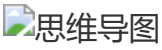
Android 热修复原理

版本号:2019/3/3-2:10

        - navtive层替换掉原方法
            - replaceMethod()
            - ArtMethod
        - 为什么替换ArtMethod的内容就能实现热修复?
            - 解释模式
            - AOT机器码模式
    - ArtMethod的兼容性问题
        - ArtMethod的整体替换
            - ArtMethod的精确尺寸
            - 线性结构不能变
    - 访问权限检查
        - 方法调用时的权限检查
        - 同包名下的权限问题
        - 反射调用非静态方法
            - 静态方法不会有该问题
            - 解决办法
    - 即时生效的限制
- 热修复与Java(68题)
    - 内部类编译
        - 静态内部类和非静态内部类的区别
        - 内部类和外部类的互相访问
        - 热部署(底层替换方案)
    - 匿名内部类编译
        - 编译期的命名规则
        - 热部署方案
    - 域编译
        - 静态field初始化/静态代码块
        - 非静态field初始化/非静态代码块
        - 热部署方案
    - final static field编译
        - static和final static修饰的区别
        - final static优化原理
        - 热部署方案
    - 方法编译
        - 混淆
        - 方法内联
        - 方法裁剪
            - 代码规范
        - 热部署方案
    - switch case语句编译
        - 热部署方案: 反编译
    - 泛型

思维导图



技术介绍(31题)

热修复基本概念

1、传统BUG修复流程的弊端?

> 1. 重新发布版本代价太大
> 2. 用户下载安装成本太高
> 3. BUG修复不及时，用户体验差。

2、对于这些弊端，有哪些合适的解决办法?(or 有哪些方案能够进行BUG的快速修复？)

方案	内容	缺点
Hybird方案	将需要经常变更的业务逻辑通过H5进行独立	1. 有学习成本， 需要对原有逻辑进行合理的抽象和转换。 2. 对于无法转为H5的代码依旧无法修复
插件化方案	例如 `Atlas` 以及 `DroidPlugin` 方案	1.移植成本高 2.需要学习插件化工具 3. 改造老代码的功能量大
热修复	APP直接从云端下拉补丁和更新	

三大优势

3、热修复的3大优势

> 1. 无需重新发版，实时高效热修复。
> 2. 用户无感知修复，无需下载新的应用，代价小。
> 3. 修复成功率高

三大领域

4、Android 热修复的3大领域

> 1. 代码修复
> 2. 资源修复
> 3. so修复

传统框架实现方式

5、传统热修复框架的实现方式

框架	方案	缺点
Xposed	手淘， 底层结构替换方案 ， 针对Dalvik虚拟机开发的Java Method Hook技术-Dexposed	1.对于底层Dalvik结构过于依赖 2. 无法继续兼容ART虚拟机(Android 5.0起)
Andfix	支付宝， 底层结构替换方案 ， 做到了Dalvik和ART环境的全版本兼容。	
Hotfix	阿里百川，Andfix升级版，业务逻辑解耦	1. 底层结构的替换方案``稳定性差 2. 使用范围限制多 3. 不支持资源和so修复
超级补丁技术	QQ控件	
Tinker	微信，	
Amigo	饿了么，	
Robust	美团，	

Sophix概览

6、Sophix的设计理念

> 1. 核心理念：非侵入性
> 2. 打包过程不会侵入到apk的build流程中。也不会增加任何AOP代码，对开发者透明化。

优势

7、Sophix框架的优势

> 1. 支持代码修复、资源修复、so修复
> 2. 集成非常简单，没有侵入性。

缺点

8、Sophix的缺点

1. 唯一缺点，是不支持四大组件的增加。但是支持四大组件的增加必然导致代码侵入性过强。
2. 一般热修复也使用于修复故障。而不是增加很多新功能。因此也不需要。
3. 可以通过增加Fragment，增加新功能。

代码修复

9、代码修复的两大主要方案

1. 阿里系的 底层替换方案 。
2. 腾讯系的 类加载方案 。

10、底层替换方案和类加载方案的优劣

方案	优点	缺点
底层替换	1.时效性最好 2.加载轻快 3.立即见效	限制很多
类加载	1.修复范围广 2.限制少	1.时效性差，需要冷启动才能见效

底层替换方案

传统方案

11、底层替换方案是什么?

1. 在已经加载了的类中直接替换掉原有的方法
2. 是在原有类的基础上进行的修改，因此无法进行 方法 和 字段 的增减(这会破坏原有类的结构)
3. 该方案的底层替换具有 不稳定性

类方法的增减

12、为什么底层替换方案无法增减原有类的方法?

1. 会导致 该类 和 整个Dex 的 方法数 变化
2. 方法数的变化 会造成 方法索引的变化 ， 这样访问方法时，就无法正常所引导正确的方法。

类字段的增减

13、为什么底层替换方案无法增减原有类的字段?

1. 增加和减少了 字段 和增减方法一样，会导致 所有字段 的索引发生变化。
2. 最严重的是，在app运行时某个类突然增加了字段，而原先已经产生的该类的实例还是 原来的结构 (这是无法改变的)，后续对这个老实例对象访问 新增字段 是很致命的。

不稳定性

14、底层替代方案是如何实现的?

1. 无论是 Dexposed 、 AndFix 以及其他的 Hook方案 都是直接修改虚拟机方法的具体字段。
2. 例如修改 Dalvik 方法的 jni函数指针 、修改类的访问权限、修改方法的访问权限

15、底层替代方案的不稳定性?

> 1.这种依赖于具体字段的Hook方案，各个厂商会对源代码进行改造，从而导致不匹配。
>
> 1. 例如 `Andfix` 里 `ArtMethod`的结构 是根据开源Android源码中的结构写死的。如果结构发生改变，就会导致 替换机制 出错。

无视底层结构的替换方案

16、无视底层具体结构的替换方法

> 1. 忽略底层 `ArtMethod` 结构的差异
> 2. 所有Android版本都不需要区分
>   即使Android版本不断修改ArtMethod的成员，只要保证ArtMethod数据仍然是 线性结构排序 就没问题

类加载方案

传统方案的原理

17、传统类加载方案原理是什么?

> 1. app重新启动后让 `ClassLoader` 去加载新的类
> 2. 不重启app，原来的类还在虚拟机中，就无法加载新的类。

18、腾讯系三类加载方案的实现原理

> 1. QQ控件会侵入打包流程，增加无用信息，不优雅。
> 2. QFix方案，获取底层虚拟机的函数，不够稳定可靠，且无法新增public函数
> 3. 微信Tinker，完整的 全量Dex加载 。会对Dex内容非常精细的比较(方法和指令的维度)，性能消耗严重。

Dex比较维度

19、Dex的比较维度有三种

> 1. 方法和指令的维度: 粒度过细，性能差
> 2. bsbiff: 粒度粗糙
> 3. 类的维度: 粒度最合适，能够达到 时间和空间平衡 的最佳效果

Sophix的方案

20、Sophix的类加载方案

> 1. dex的比较维度：类的维度
> 2. 采用全量合成dex:
>   1. 利用Android原先的类查找和合成机制，快速合成新的全量Dex-不需要处理合成时方法数超过的问题，也不会破坏性重构dex的结构。

2. 重新排列包中dex的顺序。虚拟机查找时优先找到classes.dex中的类，然后才是classes2.dex、classes3.dex

类插桩

21、Sophix中的dex文件级别的类插桩方案

1. 将 旧包 和 补丁包 中的classes.dex的顺序进行了重排
2. 让系统自动实现类覆盖的目的，大大减少合成补丁的开销

双剑合璧

22、两个方案的合并

1. 底层替换方案和类加载方案合并使用
2. 补丁工具根据实际代码变动情况：
   1. 小修改，在底层替代方案的适用范围内： 底层替代方案 -即时生效
   1. 其余： 类加载方案 -即时性差
3. Sophix底层会判断机型是否支持热修复：如果机型底层虚拟机构造不支持，依旧走 类加载修复

资源修复

23、热修复的方案大部分都参考了 `Instant Run` 的实现

Instant Run

24、Instant Run中的资源热修复的原理?

1. 构造一个新的 `AssetManager` .
2. 反射调用 `addAssetPath` ， 将这个完整的新资源包加入到 新`AssetManager` 中。
3. 找到所有引用 旧`AssetManager` 的地方，通过反射，将引用处替换为 新`AssetManager` -该Manager包含所有新资源

25、Instant Run的资源热修复主要工作都是在处理 兼容性 和查找到AssetManager引用处，替换逻辑很简单。

Sophix方案

26、Sophix的资源修复方案

1. 构造一个 `package id = 0x66` 的资源包，包含两种资源：1.新增资源 2.原有内容发生改变的资源
2. 直接在原有AssetManager中addAssetPath `0x66`资源包 ， 不和已经加载的0x7f冲突。不再需要去找到所有 引用`AssetManager`的地方
3. Android 4.4及以下：需要在原有的AssetManager对象上进行析构和重构。保证 `addAssetPath` 生效。Android 5.0开始， `addAssetPath(0x66`资源包`)` 会直接加载和解析资源。

27、Sophix资源修复方案的优势

1. 不修改AssetManager的引用处，替换更快更安全(对比Instant Run以及所有copycat的实现)
2. 不必下发完整包，补丁包只包含改动的资源(对比Instant Run、Amigo等方式的实现)
3. 不需要在运行时合成完整包。不占用运行时资源。(对比Tinker的实现)

不修改AssetManager的引用处

28、不修改AssetManager的引用处

> 直接在原有的AssetManager对象上进行析构和重构。不再需要去替换所有 旧`AssetManager`的引用

不必下发完整包

29、不必下发完整包

1. 构造一个 `package id = 0x66` 的资源包，包含 新增资源 和 原有内容发生改变的资源
2. 直接在原有AssetManager中addAssetPath `0x66`资源包 ， 会优先找到0x66资源包中的资源

不需要在运行时合成完整包

30、不需要在运行时合成完整包

1. 采用dex文件级别的类插桩方案
2. 重新排列包中dex的顺序。虚拟机查找时优先找到classes.dex中的类，然后才是classes2.dex、classes3.dex。系统自动实现类覆盖。

SO库修复

31、SO库修复的原理

1. 本质是对native方法的修复和替换
2. 采用类似 类修复的反射注入方式 ， 把 补丁so库 的路径插入到 `nativeLibraryDirectories`数组 的最前方，这样加载so库的时候是补丁so库
3. 该方案在启动期间，反射注入补丁so库，而不是其他方案手动替换系统的 `System.load()` 来实现替换目的

代码热修复

底层热替换原理(23题)

Andfix即时生效的原理

navtive层替换掉原方法

1、Andfix的即时生效原理

1. Andfix即时生效，不需要重新启动，但是也有使用限制(不能增减方法和字段，只能替换掉原方法)。
2. 方法：在已经加载的类中，直接在navtive层替换掉原方法，

replaceMethod()

2、AndFix的核心：replaceMethod()

> 1. 获取到原有方法的 `Method对象` ， 并且替换为新方法 `dest`
> 2. 根据虚拟机类型是 `art` 还是 `dalvik` ， 调用对应替换的方法(art/dalvik_replaceMethod)。
> 3. Android 4.4以下是dalvik， 4.4及以上是ART虚拟机

```
@AndFix /src/com/alipay/enuler/andfix/AndFix.java
// src = 原有方法
// dest = 新方法
private static native void replaceMethod(Method src, Method dest);

@AndFix /jni/andfix.cpp
static void replacMethod(JNIEnv* env, jclass clazz, jobject src, jobject dest){
  is(isArt){
    art_replaceMethod(env, src, dest);
  }else{
    dalvik_replaceMethod(env, src, dest);
  }
}

@AndFix /jni/art/art_method_replace.cpp
extern void art_replaceMethod(JNIEnv* env, jobject src, jobject dest){
  if(apilevel > 23){
    replace_7_0(env, src, dest);
  }else if(apilevel > 22){
    replace_6_0(env, src, dest);
  }else if(apilevel > 21){
    replace_5_1(env, src, dest);
  }else if(apilevel > 19){
    replace_5_0(env, src, dest);
  }else{
    replace_4_4(env, src, dest);
  }
}
```

3、Android 6.0为例解析替换函数：replace_6_0

> 1. 每个Java方法在art中都一个对应的 `ArtMethod`
> 2. `ArtMethod` 记录着Java方法的所有信息： 所属类、访问权限、代码执行地址 等等。
> 3. 利用ArtMethod指针对所有成员进行修改。
> 4. 这样后续调用 该Java方法 就会走到 新的方法实现 中

```
@AndFix /jni/art/art_method_replace_6_0.cpp
void replace_6_0(JNIEnv* env, jobject src, jobject dest){

  /**==========================================
   * 1、通过Method对象得到Java函数在底层对应的ArtMethod的真实地址
   *    1. 通过`FromReflectedMethod()`获得Method对象对应的ArtMethod的真实起始地址。
   *    2. 利用ArtMethod指针对所有成员进行修改。
   *==========================================*/
  art::mirror::ArtMethod* srcMeth = (art::mirror::ArtMethod*)env->FromReflectedMethod(src);
  art::mirror::ArtMethod* destMeth = (art::mirror::ArtMethod*)env->FromReflectedMethod(dest);

  /**==========================================
   * 2、将原方法的ArtMethod内部所有信息都替换为dest ArtMethod的内容
   *      1. 所属类
   *      2. 访问权限
   *      3. 代码执行地址
   *      ......
   *==========================================*/
   srcMeth->declaring_class_ = destMeth->declaring_class_;
   srcMeth->method_index_ = destMeth->method_index_;
   // xxx
}
```

ArtMethod

4、ArtMethod是什么?

> `ArtMethod` 记录着Java方法的所有信息： 所属类、访问权限、代码执行地址 等等。

5、字段declaring_class就是方法所属的类

> 1. 类Student的test()方法的declaring_class就是Student.class

为什么替换ArtMethod的内容就能实现热修复?

6、为什么替换了原Java方法对应的ArtMethod的内容就能实现热修复？虚拟机调用方法的原理?

> - Android6.0，art虚拟机中 `ArtMethod` 的结构如下：包含 方法的执行入口

```
@art /runtime/art_method.h

class ArtMethod FINAL{
  // 1、方法执行的入口
  void* entry_point_from_interpreter_;
  void* entry_point_from_quick_compiled_code_;
}
```

> - Java代码在Android中被编译为 `Dex Code` ， `art` 中可以采用 解释模式 或者 `AOT`机器码模式 执行
>   1. 解释模式: 执行方法时，取出ArtMethod的 `entry_point_from_interpreter_` 的方法执行入口地址，跳转过去执行。
>   2. AOT机器码模式: 执行方法时，取出ArtMethod 的 `entry_point_from_quick_compiled_code_` 的方法执行入口地址，跳转过去执行。

- 简单的替换 entry_point_* 字段表明的入口地址，不能够 实现方法的替换 。
  - 因为运行期间还会用到ArtMethod里面的 其他成员字段
- 即使是 AOT机器码模式 ， 编译出的 AOT机器码 的执行构成，依旧会有对 ArtMethod很多成员字段的依赖
- 结论：只有替换掉所有原ArtMethod中的成员字段，在所有执行到旧方法的地方，才能完整获取到所有新方法的信息: 执行入口、所属class、方法索引号、所属dex信息等，完美地去跳转到新方法。

解释模式

7、什么是解释模式执行

1. 取出 DEX Code 逐条解释执行。

AOT机器码模式

8、说什么是AOT机器码模式

1. 预先编译好 Dex code 对应的 机器码 ， 运行时直接运行机器码

ArtMethod的兼容性问题

9、AndFix等Hook方案采取的native替换的方法都具有不稳定性

1. 使用的 ArtMethod 结构完全根据 Android源码中ArtMethod 的结构写死的。
2. 一些厂商修改了 ArtMethod的内容和结构 就会导致 热修复失效---兼容性很差

ArtMethod的整体替换

10、native替换方法的兼容性的解决办法

1. 原native替换方法是 替换ArtMethod 的所有成员，因此需要依赖具体结构。
2. 解决办法：不构造出 ArtMethod具体的成员字段 ， 将 ArtMethod进行整体替换

```
memcpy(srcMeth, destMeth, sizeof(ArtMethod));
```

ArtMethod的精确尺寸

11、整体替换ArtMethod的核心在于如何精确计算出 sizeof(ArtMethod)

1. 该整体替换ArtMethod的方案，在于如果 ArtMethod 的size计算有偏差，会导致： 部分成员没有替换、替换区域超出了边界
2. 应用开发者无法知道具体Andorid设备的系统里 ArtMethod的尺寸
3. 通过 class_linker.cc 源码中 LoadClassMembers()->AllocArtMethodArray() 中可以知道 ArtMethod Array(数组) 的 ArtMethod 是紧密相连的。通过相邻两个 ArtMethod 的 起始地址的差值 就是 ArtMethod的精准大小
4. 类方法分为 Direct方法 和 Virtual方法 ， 各自有各自的 ArtMethod数组
   - direct方法: static方法和所有不可继承的对象方法

> - virtual方法: 所有可以继承的对象方法

12、借助ArtMethod紧密相连的特性，如何精准计算出ArtMethod的大小?

> 1. 构造一个辅助的类，并具有两个空方法：
>   - f1()、f2()都是 `static` 方法，都属于 `direct ArtMethod Array`
>   - NativeStructsModel中只有 这两个方法 ， 因此肯定是相邻的

```
// f1()、f2()都是`static`方法，都属于
public class NativeStructsModel{
  final public static void f1(){}
  final public static void f2(){}
}
```

> 2. 在 `JNI层` 计算出 `f1()`和`f2()` 地址的差值。

```
  size_t firstMid = (size_t) env->GetStaticMethodId(nativeStructModelClazz, "f1", "()V");
  size_t secondMid = (size_t) env->GetStaticMethodId(nativeStructModelClazz, "f2", "()V");
  // 第二个方法起始地址 - 第一个方法起始地址
  size_t methodSize = secondMid - firstMid;
```

> 3. 该 `Size`就可以直接作为`ArtMethod`的尺寸

```
//  memcpy(srcMeth, destMeth, sizeof(ArtMethod));

// 替换为:
  memcpy(srcMeth, destMeth, methodSize);
```

线性结构不能变

13、利用技巧获取到ArtMethod尺寸的优缺点

> 1. 优势：对于所有Android版本都不需要区分
> 2. 注意点：只要 `ArtMethod`数组 依旧是 线性结构 ， 无论 `ArtMethod`的成员 如何改变，都完美兼容。
> 3. `ArtMethod`数组 的 线性结构 会被修改的可能性极低!

访问权限检查

方法调用时的权限检查

14、只替换ArtMethod的内容，被替换的方法有权限访问该类的其他private方法吗?

> 1. 可以
> 2. 在 `dex2oat` 生成 `AOT`机器码 时已经做过检查和优化，因此机器码中 不存在权限检查
> 3. 例如下面：即使func()方法偷梁换柱为其他方法，依旧可以调用 `private`的`func()`

```
public class Demo{
  Demo(){
    func();
  }

  private void func(){
  }
}
```

同包名下的权限问题

15、补丁中的类在访问同包名下的类时，会出现 访问权限异常：

> 1. 具有类 com.patch.demo.BaseBug 和 com.path.demo.MyClass 是同一个包 com.patch.demo 下面的。
> 2. 此时替换了 com.patch.demo.BaseBug 的方法 test ， 因为该方法的 ArtMethod 被完全替换，因此指向的是 新的补丁类 。
> 3. 该 补丁包中的BaseBug 是补丁包的 Classloader加载的 ， 和原先的包不是同一个Classloader，判定为不同包。 BaseBug.test() 中访问 MyClass类 ， 会导致提示无法访问 com.path.demo.MyClass 。
> 4. 校验逻辑在虚拟机代码的 Class::IsInSamePackage 中：会要求 Classloader 必须相同

16、只需要设置new Class的Classloader为old Class的Classloader就可以解决该问题:

> 1. 不需要在 JNI层 处理底层的结构
> 2. 只需要通过反射进行设置

```
// 1. 获取classloader的Field
Field classLoaderField = Class.class.getDecalredField("classLoader");
// 2. 允许访问权限
classLoaderField.setAccessible(true);
// 3. 将新类的classloader设置为旧类的classloader
classLoaderField.set(newClass, oldClass.getClassLoader());
```

反射调用非静态方法

17、非静态方法被热替换后，再反射调用该方法，会抛出异常。

> 1-下面会报错: 新BaseBug 的test()传入 旧BaseBug ， 不匹配就会报错。

```
// BaseBug的test()方法已经被热替换
// ...
// 1、该对象bb是原始的BaseBug类对象
BaseBug bb = new BaseBug();
// 2、该test()是补丁包中BaseBug的test()方法
Method testMeth = BaseBug.class.getDeclaredMethod("test");
// 3、新BaseBug的test()传入就BaseBug，导致报错
testMeth.invoke(bb);
```

> 2- invoke()->InvokeMethod()->VerifyObjectIsClass(): 会检测Method.invoke()参数传入的目标对象(旧类的对象)，是否是方法对应的ArtMethod所属的Class(新类)。

```
// object = 旧类的对象bb
// C = ArtMethod的declaring_class = 新类
inline bool VerifyObjectIsClass(Object object, Class* c){
  if(UNLIKELY(!object->InstanceOf(c))){
    // 报错
    return flase;
  }
  // xxx
}
```

静态方法不会有该问题

18、静态方法为什么不会有该问题?

> 是在 类的级别 直接调用的，不会接受 对象实例 作为参数，也不会有该方面的检查。

解决办法

19、非静态方法被热替换后，再反射调用该非静态方法，会抛出异常。解决办法是:

> 冷启动机制

即时生效的限制

20、即时生效这种运行期间修改底层结构的方案具有的限制有哪些?

> 1. 只能支持方法的替换：已存在类的方法增/减和字段增/减都不适用
> 2. 反射调用非静态方法会抛出异常

21、哪些场景是支持的?

> 1. 方法的替换
> 2. 新增一个完整的，原先包里不存在的新类

22、优点

> 1. 一旦符合使用条件，性能极佳，补丁小，加载迅速

23、不满足即时生效的场景该如何如何处理?

> 1. 冷启动修复

热修复与Java(68题)

内部类编译

1、外部类有个方法，将其修改为访问内部类的某方法，会导致补丁包新增一个方法。

2、 内部类 在编译期会被编译为跟 外部类 一样的 顶级类

静态内部类和非静态内部类的区别

3、静态内部类和非静态内部类的区别

> 1. 静态内部类 不持有外部类的引用
> 2. 非静态内部类会 持有外部类的引用
> 3. 例如: handler的实现需要采用静态内部类，避免OOM

4、非静态内部类编译时会增加字段 `this` 用于持有外部类的引用

5、持不持有外部类引用，都不影响热部署。

> 都是一个顶级类，新增一个顶级类，不影响热部署

内部类和外部类的互相访问

6、内部类和外部类都是顶级类，是否就表示对方private的内容无法被访问到?

> 1. 外部类需要访问内部类的 `private` 域/方法 ， 编译期间会为内部类生成 `access&**` 相关方法。
>   - 外部类就能访问内部类的private内容
> 2. 内部类需要访问 外部类 的 `private` 属性/方法 ， 编译期间会为外部类生成 `access&**` 相关方法。
>   - 内部类就能访问外部类的private内容

热部署(底层替换方案)

7、补丁前的test()没有访问内部类的private属性/方法, 补丁后的test()访问了内部类的private属性/方法，会导致无法使用 热部署/底层替换方案

> 1. 会新增 `access&**` 相关方法，按照限制，在原有类中增加方法，因此无法热部署
> 2. 只要避免生成 `access&**` 相关方法，就能走热部署。

8、如何避免编译器自动生成 `access&**` 相关方法

> 1. 如果一个外部类有内部类:
>   1. 把外部类所有 `private`属性/方法 的访问权限更改为其他权限(public、protected、default)
>   2. 把内部类所有 `private`属性/方法 的访问权限更改为其他权限(public、protected、default)

匿名内部类编译

9、匿名内部类在避免新增 `access&**` 方法的基础上，依旧新增了一个内部类和新增了method方法

> 1. 热部署允许新增一个类
> 2. 热部署不允许新增方法

编译期的命名规则

10、匿名内部类的名字格式是 外部类`&` + 数字

> 下例中：Thread的匿名内部类，编译期的名字为： `Demo&1`

```
public class Demo{
    public static void test(){

        new Thread(){
            // xxx
        }.start();
    }
}
```

> 此时有两个顶级类

11、 原有的匿名内部类 前插入 新的匿名内部类 会导致混乱

> 1. 下例中：有两个匿名内部类
>   1. Demo&1 --- Callback.OnClickListener
>   2. Demo&2 --- Thread
> 2. 补丁会比较新的 `Demo&1` 和旧的 `Demo&1`，然而这两者完全不同。
>   1. 会新增OnClick()方法 --- 影响热部署(Demo&1中增加了新方法，删减了旧方法)
>   2. 会新增一个匿名内部类 --- 不影响，新增类没事(Demo&2)

```
public class Demo{
    public static void test(){

    // 新增一个内部类
        new Callback.OnClickListener{
          public void onClick(){
            // xxx
          }
        }


        new Thread(){
            // xxx
        }.start();
    }
}
```

热部署方案

12、在新增/减少匿名内部类时，如何支持热部署方案?

> 1. 唯一情况：增加的匿名内部类必须插入到 外部内末尾
> 2. 其余情况：无解，补丁工具无法区分。

域编译

静态field初始化/静态代码块

13、热部署不支持 `clinit` 的修复

1. 热部署不支持 `method/field` 的新增
2. 热部署不支持 `clinit` 的修复

14、clinit在Dalvik虚拟机中类加载的时，进行类初始化时调用。

15、静态field初始化和静态代码块会被编译到 `clinit` 方法中

> 该方法由编译器自动合成

16、静态field初始化和静态代码块在 `clinit` 中的 顺序 取决于代码中出现的先后顺序

17、最常见的三种会去加载类的情况

1. new一个类对象(new-instance指令)
2. 调用类的静态方法(invoke-static指令)
3. 获取类的静态field的值(sget指令)

18、类没有被加载过时, 加载的流程

1. dvmResolveClass()
2. dvmLinkClass()
3. dvmInitClass(): 先对父类进行初始化，再调用本类的 `clinit()`

非静态field初始化/非静态代码块

19、非静态field初始化/非静态代码块会被编译到 `init无参构造函数` 中，顺序和源码中一致

20、构造函数会自动编译成init方法

热部署方案

21、任何静态field初始化和静态代码块的变更都会编译到clinit中，无法热部署，只能冷启动(处于类加载的初始化期间)

22、非静态field初始化和非静态代码块的变更都会编译到init中，只被当作一个普通方法的变更，对热部署无影响(普通的方法)

final static field编译

23、 `final static` 修饰的field编译时是否会编译到 `clinit` 中?

1. 作为 静态域 ， 应该都被编译到 `clinit` 中，但是并不完全正确
2. 修饰的 基本类型/String常量类型 ， 不会编译到 `clinit` 中

24、下例中类中的field哪些会被编译到 `clinit` 方法中？哪些不会?

```
public class Demo{

    static Object o1 = new Object(); // √
    final static Object o2 = new Object(); // √

    static int i1 = 1; // √
    final static int i2 = 2;// ×不会


    final static String s1 = new String("new String"); // √
    final static String s2 = "常量"; // ×不会
}
```

> 1. finalt static修饰的 基本类型和String常量类型 不会编译到 clinit 中

25、 final static 修饰的 基本类型/String常量类型 是在哪里初始化的?

> 1. 类加载初始化的 dvmInitClass 在执行 clinit 之前，调用 initSFields 对 static域设置默认值 。
> 2. initSFields 设置默认值的目标包括 静态域的所有引用类型/基本类型/String常量类型 ， 但是 基本类型/String常量类型 在后面的 clinit 中就不会设置了

static和final static修饰的区别

26、static和final static修饰的区别

> 1. final static修饰的 原始类型和String类型(非引用类型)的field ， 不会编译到 clinit 中，会提前在 类初始化执行的initSField 中进行初始化赋值。
> 2. final static修饰的 引用类型 和 static修饰的所有类型 ， 仍然在 clinit中初始化

final static优化原理

27、对于常量使用 final static 修饰就能达到优化效果?

> 错误!
>
> - 只有 final static 修饰的 原始类型和tring类型常量 才能得到优化。

28、final static进行优化的原理

> 1. 可以优化的情况中：要访问该常量通过 const/4 指令实现，该指令非常简单
> 2. 不可优化的情况中：访问这些field，通过 sget 指令。内部包含解析，解析类等操作，属于重操作。

29、final对于final static修饰的引用类型的唯一作用就是避免该field被修改

热部署方案

30、final static修饰的field如何进行热部署?

> 1. 可以热部署:
>   1. 基本类型: 引用该基本类型的地方都会被 立即数 替换

> 2. String常量：所有引用该常量的地方都被 常量池索引id 替换
> 3. 热部署中将所有引用到该 final static field 的方法都进行替换，走热部署没问题。
> 2. 不可热部署：
> 1. final static 修饰的 引用类型 都被翻译到 clinit 中，不会热部署。

方法编译

混淆

31、混淆可能导致方法内联和裁剪，而导致 method 的增减

方法内联

32、哪些场景会导致方法内联?

> 1. 方法没有被其他任何地方引用
> 2. 方法足够简单，例如只有一行，会在任何调用该方法的地方用该方法的实现进行替换
> 3. 方法只有一个地方引用到，会在调用处用实现进行替换

33、方法内联为什么会导致方法的增减？以及导致热部署失效?

> 1. 原Class中具有一个 test() 方法，因为内联，所以编译后不再有 test()方法
> 2. 新Ckass中，因为不满足 内联的条件 导致 tets()不被内联 ， 因此多出来 test()方法
> 3. 前后对比，因为 新增方法 导致不能 热部署 ， 只能 冷启动
> 4. 反过来 方法内联 也会导致 方法的减少

方法裁剪

34、方法裁剪

> 1. test(context) 方法中由于 context参数 没有被使用到，因此 混淆任务 会先生成 裁剪过后无参的test()方法 ， 然后再进行混淆。
> 2. 如果新代码中，正好使用了 参数 ， 不会导致方法裁剪，因此会新增一个 具有参数的test(context) 方法
> 3. 方法裁剪导致 方法增减 ， 导致 不嗯呢刚热部署

35、如何避免方法裁剪?

> 1. 保证所有 参数被使用 ， 或者进行特殊处理：

```
public void test(Context context){
 if(Boolean.FALSE.booleanValue()){
   context.getApplicationContext();
 }
}
```

代码规范

热部署方案

36、如何避免混淆时的方法内联和方法裁剪导致热部署失效的问题?

> 混淆配置文件中加上配置项 `-dontoptimize` 就可以关闭方法的裁剪和内联

37、混淆库的预编译会拖累打包速度，Android虚拟机有自己的一套代码校验逻辑

> 需要加上配置项 `-dontpreverify`

switch case语句编译

38、资源修复方案中需要对新旧ID进行替换，但是 `switch case` 中的 `id` 不会被替换

39、switch case 语句编译实例中解析编译规则

> 1-第一个方法较为连续。第二个方法不连续.
> 2-第一个testContinue()方法中，因为1、3、5连续，使用指令 `packed-switch` ， 会影响 热部署
> 3-第二个testNotContinue()中，1、3、10不连续，使用指令 `sparse-switch`

```
public void testContinue(){
    int temp = 2;
    int result = 0;
    switch (temp){
        case 1:
            result = 1;
            break;
        case 3:
            result = 1;
            break;
        case 5:
            result = 1;
            break;
    }
}

public void testNotContinue(){
    int temp = 2;
    int result = 0;
    switch (temp){
        case 1:
            result = 1;
            break;
        case 3:
            result = 1;
            break;
        case 10:
            result = 1;
            break;
    }
}
```

热部署方案: 反编译

40、为什么资源id替换不完全？如何解决?

1. 资源id 肯定是 const final static 变量，导致 switch case 被翻译成 packed-switch 指令
2. 采用方案： 反编译(强行替换指令) -> 资源id替换 -> 重新编译
    1. 修改反编译流程： 遇到 packed-switch指令 就强转为 sparse-switch指令 ； :pswitch_N 等标签指令强转为 :sswitch_N 指令
    2. 资源ID的暴力替换
    3. 重新编译为 Dex

泛型

41、泛型可能会导致 method 的新增

为什么需要泛型

42、Java中的泛型完全在 编译器 中实现

1. 由编译器执行 类型检查 和 类型推断
2. 然后生成普通的无泛型的字节码。泛型知识为了保证类型安全。
3. 这种技术就是 擦除(erasure)

43、Java的泛型为什么要采用 擦除技术 来实现?

1. 泛型从Java5才引入
2. 通过扩展虚拟机指令集来支持泛型是不可以的，也会导致升级JVM具有很多障碍

Object实现泛型

44、Object实现泛型

```
public class ObjectGeneric {
    private Object obj;
    public void setValue(Object value){
        obj = value;
    }
    public Object getValue(){
        return obj;
    }

    public static void main(String args[]){
        ObjectGeneric generic = new ObjectGeneric();
        generic.setValue(true);
        // 1、获取数值
        boolean bool = (boolean) generic.getValue();
        // 2、获取到Int值
        int n = (int) generic.getValue();
    }
}
```

1. 上面1和2在编译期间都不会报错，因为符合Java语法。
2. 但是在实际运行中， 2 会出现 java.lang.ClassCastException 的异常:

```
Exception in thread "main" java.lang.ClassCastException: java.base/java.lang.Boolean cannot be cast to java.ba
        at ObjectGeneric.main(ObjectGeneric.java:16)
```

45、Java5泛型提出之前采用 Object 实现该效果，但是会导致 编译器 无法检测出 类型不匹配的问题

> 泛型在 编译时 就进行 类型安全检测

泛型

46、泛型会在编译期间进行检查, 实例:

```
public class ObjectGeneric<T> {
    private T obj;
    public void setValue(T value){
        obj = value;
    }
    public T getValue(){
        return obj;
    }

    public static void main(String args[]){
        ObjectGeneric<Boolean> generic = new ObjectGeneric();
        generic.setValue(true);
        // 1、获取数值
        boolean bool = (boolean) generic.getValue();
        // 2、获取到Int值
        // int n = (int) generic.getValue();
    }
}
```

> 情况2获取到Int值，会报错。

类型擦除

47、下面的例子中的类型擦除:

> 1. 方法 setValue(T value) 会被处理为 setValue(Object value)，因此编写一个方法为 setValue(Object value) 会 报错！
> 2. 泛型设置类具体类型 <Integer> 本质在字节码中生成的还是 Object类型的参数 ， 只是利用这个进行了类型检查。

```
public class ObjectGeneric<T> {
    private T obj;
    public void setValue(T value){
        obj = value;
    }

    // 报错：Error:(9, 17) java: 名称冲突: setValue(java.lang.Object)和setValue(T)具有相同
    public void setValue(Object value){
        obj = value;
    }

}
```

类型擦除和多态的冲突

48、类型擦除会导致本来是想 重写 ，结果变成了 重载

> 1. setValue(T value) 在字节码上是 setValue(Object obj)
> 2. 结果 setValue(Integer value) 是对父类 setValue(Object obj) 的重载
> 3. 然而需要的效果是 setValue(Integer value) 是对父类 setValue(T obj) 的重写

```
public class ObjectGeneric<T> {
    private T obj;
    public void setValue(T value){
        obj = value;
    }
    public T getValue(){
        return obj;
    }

    class B extends ObjectGeneric<Integer>{
        private Integer n;
        public void setValue(Integer value) {
            n = value;
        }
    }
}
```

49、使用 @Override 能实现 重写

```
    class B extends ObjectGeneric<Integer>{
        private Integer n;

        @Override
        public void setValue(Integer value) {
            n = value;
        }
    }
```

bridge

50、编译器会自动合成bridge方法来实现 重写 的效果

> 编译器自动生成一个 setValue(Object Value) 来重写 父类 的该方法

```
    class B extends ObjectGeneric<Integer>{
        private Integer n;

//        public void setValue(Object obj){
//            // xxx
//        }

        @Override
        public void setValue(Integer value) {
            n = value;
        }

// 自动生成
//        public void setValue(Object value){
//            n = value;
//        }
    }
```

51、虚拟机是通过 参数类型+返回类型 来确定一个方法，和Java语言规则不同。

> 该方法用于解决泛型中类型擦除和多态的冲突问题

52、泛型的隐形类型转换，编译器会自动加上 check-cast 类型转换

> 不需要程序员进行 显式地类型转换 ， 而是自动进行类型转换。

```
    public static void main(String args[]){
        ObjectGeneric<Boolean> generic = new ObjectGeneric();
        generic.setValue(true);
        // 1、获取数值
        boolean bool = generic.getValue();
    }
```

热部署的方案

53、泛型对热部署的影响

> 1. 类型擦除的过程中可能会 新增bridge方法 ， 导致热部署失败
> 2. 另一方面 泛型方法内部 会生成一个 dalvik/annotation/Signature 的系统注解，方法逻辑没出现变化，但是该方法的注解发生了变化。补丁工具进行判断会走 热部署进行修复 ， 然而并没有什么意义(方法逻辑没有变化，根本不需要修复)

Lambda表达式编译

54、Lambda表达式简介

> 1. Java 7 才引入的一种表达式
> 2. 类似 匿名内部类 ， 却 有巨大的区别

3. 会导致方法的增减，影响热部署

55、函数式接口的两大特征

1. 是一个街口
2. 具有唯一的一个抽象方法
3. 典型的函数式接口 Runnable和Comparator

和匿名内部类的区别

56、Lambda表达式和匿名内部类的区别?

1. 关键字this:
    1. 匿名内部类的this指向 匿名类
    2. lambda表达式的this指向 包围lambda表达式的类
2. 编译方式:
    1. 编译器将 匿名内部类 编译成 新类，名称为 外部类名+&number
    2. 编译器将 lambda表达式 编译成 类的私有方法，使用 Java7的invokedynamic 字节码指令进行动态绑定该方法

invokedynamic

57、实例解析lanmbda表达式

1. 编译期间会自动生成私有静态的 lambda$test$ + number(参数类型) 的方法
2. invokedynamic执行 lambda表达式
3. 相比于 匿名内部类，不会生成 外部类名 + & + number 的新类。

```
public class Test{
  public static void test(){
    new Thread( ()->{
          // xxx
    }).start();
  }
}
```

metafctory

58、invokedynamic指令简介

1. java7新增，用于支持 动态语言： 允许方法调用可以在运行时指定类和方法，不需要编译时确定。
2. 每个 invokedynamic 指令出现的位置被称为 动态调用点
3. invokedynamic 指令后会跟着一个 指向常量池的调用点限定符(#3, #6)
4. 调用点限定符 会被解析为 一个动态调用点
5. invokedynamic指令 最终会去执行 java.lang.invoke.LambdaMetafactory类 的 静态方法: metafctory()，该方法 会在运行时声称一个实现函数式接口的具体类

6. 该 具体类-例如：Test$$Lambda$1.java 会调
   用 私有静态方法： lambda$test$ + number(参数类型)， 执行 lambda表达式的逻辑

```
final class Test$$Lambda$1 implements Runnable{
  @Hidden
  public void run(){
    // 去执行自动生成的lambda表达式相关的方法，该方法内部就是自定义的逻辑
    Test.lambda$test$0();
  }
}
```

Android虚拟机中的lambda

59、Android虚拟机下是如何解释lambda表达式的?

1. android虚拟
   机 首先通过javac把源代码.java编译成.class，在通过dx工具优化成适合移动设备的dex字节码文件
2. android中 如果要使用java8语言特性， 需要使用 新的jack工具链 来替代老的工具链进行 编译
3. Jack 会将 .java文件 编译成 .jack 文件，最后直接编译成 .dex 文件(Dalvik字节码文件)

60、构建 Android Dalvik可执行文件 可使用的两种工具链对比

1. 旧版javac工具链
   javac (.java -> .class)-> dx (.class ->. dex)
2. 新版Jack工具链
   Jack (.java -> .class -> .dex)

Jack

61、Jack是什么?

> Java Android Compiler Kit

62、Jack工具链中处理lambda的异同

1. 相同点:
   - 编译期间都会 为外部类合成一个static辅助方法， 内部逻辑就是 lambda表达式的内容
2. 不同点:
   1. 老版本中通过 invokedynamic指令 执行 lambda；Jack的 .dex 中执
      行 lambda表达式 和 普通方法调用没有区别
   2. 老版本是在 运行中生成新类； Jack是在 编译期间 生成 新类

热部署方案

63、Lambda表达式会导致热部署失效的原因

1. 方法的增减: 新增一个lambda表达式，会导致 外部类新增一个辅助方法
2. 顺序混乱: 合成类的命名规则 = “外部类雷鸣 + Lambda + Lambda所在方法的签名 +
   LambdaImpl + 出现的序号 ”，和 匿名内部类一样的问题

64、不增减lambda表达式，不改变lambda表达式的顺序，只是更改Lambda原有内部逻辑，能否走热部署?

> 在一定情况下，依旧会出问题，不能走热部署：
>
> 1. 如果 lambda表达式 访问 外部类非静态的field和method
>   1. 编译期间在.dex文件 中会自动生成 新的辅助类(Test$$Lambda$1.java)，该类 没有持有外部类的引用
>   2. 为了访问非静态的field和method，会导致 需要持有外部类的引用，从而增加一个字段来持有
> 2. 辅助类的field的增减 导致无法热部署

```
final class Test$$Lambda$1 implements Runnable{
  @Hidden
  public void run(){
    // 去执行自动生成的lambda表达式相关的方法，该方法内部就是自定义的逻辑
    Test.lambda$test$0();
  }
}
```

65、Lambda表达式对热部署影响的总结

> 1. 增加/减少 一个lambda表达式会导致 类方法的错乱 。热部署失败!
> 2. 修改 一个原有lambda表达式，因为可能访问/取消访问 外部类的非静态field和method 的情况，可能导致 辅助类的field 的 增加/减少 。热部署失败!
> 3. 调整原有lambda表达式的顺序 ，会导致 类方法的错乱 。热部署失败!

访问权限检查

66、一个类的加载必须经历 resolve 、 link 、 init 三个阶段

类加载阶段

67、类加载阶段中对父类和当前类实现的接口的权限检查主要在link阶段

> 1. 如果当前类、实现的接口、父类是非public的，并且加载两者的 classLoader 不一样的情况，直接return
> 2. 代码热修复方案是基于新classLoader的，类加载阶段就会报错

类校验阶段

68、如果补丁类中存在非public类的访问、非public方法的调用、非public field的调用都会导致失败

> 1. 这些错误在 补丁加载阶段 是检测不出来的，补丁会被视作正常加载
> 2. 直到运行阶段，会直接crash

冷启动类加载原理(26题)

1、冷启动方案的作用?

1. 热部署有很多限制
2. 在超出限制的情况下，再通过冷启动进行补充，使得热修复一定能成功。

传统实现方案

Tinker

2、Tinker如何实现冷启动的?

1. 提供Dex差量包，并整体替换Dex的方案。
2. 通过差量的方式生成patch.dex(补丁dex文件)，然后将 `patch.dex` 和应用的 `classes.dex` 合并成一个完整的dex
3. 加载 新dex文件 得到 `dexFile`对象 并以此构造出 `Element`对象 ， 然后整体替换掉 旧的`dex Elements`数组

3、Tinker方案的优点

1. 自研dex差异算法，补丁包小，不影响类加载性能。

4、Tinker方案的缺点

dex合并，在VM Heap上消耗内存，容易OOM，导致dex合并失败

5、Tinker如何避免OOM导致的dex合并失败的问题?

1. 可以在jni层面进行dex的合并，从而避免OOM导致dex合并失败
2. 但是JNI层实现比较复杂。

插桩实现

6、如果仅仅把补丁类打入补丁包中而不做任何处理会出什么问题？该问题是啥意思?

1. 运行时 类加载 的时候会异常退出

dexopt

odex

7、加载一个dex文件到本地内存的流程

1. 如果不存在 `odex`文件 ， 首先会执行 `dexopt`
2. `dexopt` 的入口在 `davilk/opt/OptMain.cpp` 的 `main`方法
3. 最后调用 `verifyAndOptimizeClass` 进行真正的 `verify(验证)`和`optimize(优化)` 操作

8、dexopt的流程
dexopt

verifyAndOptimizeClass

9、Apk第一次安装时的流程

1. 对原Dex执行 dexopt -> 执行到verifyAndOptimizeClass()
2. 会先进行 类校验-dvmVerifyClass()：校验成功，则所有类都会打上 CLASS_ISPREVERIFIED 标志
3. 接着执行 类优化-dvmOptimizeClass()，并且打上 CLASS_ISOPTIMIZED 标志

dvmVerifyClass

10、dvmVerifyClass()方法的作用

1. 类校验，目的是: 防止类被篡改校验类的合法性
2. 会对 类的每个方法 进行校验， 类的所有方法 中 直接 引用的类和当前类都在同一个dex中：return true

dvmOptimizeClass

11、dvmOptimizeClass()方法的作用

1. 类优化，将部分指令优化成 虚拟机的内部指令
2. 例如: 方法调用指令
    1. invoke-* 指令变成了 invoke-*-quick 指令
    1. quick指令直接从 vtable 表中取，该表是类的所有方法的表(包括继承的方法)， 加快了方法的执行速度

dvmResolveClass

12、加载阶段中为什么会出现 dvmThrowllegalAccessError (运行时异常)?

1. 原Dex中的 类B 中的某个方法引用到 补丁包中的类A
2. 执行到该方法时，会尝试解析类A:
    1. 类B具有 CLASS_ISPREVERIFIED 标志
    2. 然后判断 类A 和 类B 所属的 dex，因为不同，抛出异常 dvmThrowllegalAccessError

13、为什么原Dex类B能引用到补丁类A的方法？明明没打补丁前，都不知道有这个补丁类A?

补丁类A作为补丁，说明原包中肯定有一个原始类A

插桩

14、如何解决dvmThrowllegalAccessError问题?

1. 构造一个单独没啥用的 帮助类 放到一个单独的Dex中
2. 原Dex中所有类的构造函数都引用这个类
3. 这里需要侵入dex打包流程，利用 .class字节码修改技术，在所有 .class 文件的构造函数中引用该 帮助类
4. 在加载Dex文件时，会走dexopt流程，在 dvmVerifyClass 校验时，校验失败(类B的所有方法中引用到的类-帮助类，和类B不在一个Dex中)。原dex中所有类没有 CLASS_ISPREVERIFIED 标志。并且后续流程也不走，不会打上 CLASS_ISOPTIMIZED
5. 因此引用到补丁类A时，解析类A，不会进入 CLASS_ISPREVERIFIED 标志的后续判断，也不会抛出异常 dvmThrowllegalAccessError

插桩导致类加载性能差

15、插桩为什么会导致类加载的效率很低?

> 1. 类的加载需要三个阶段：dvmResolveClass->dvmLinkClass->dvmInitClass
> 2. 如果类因为插桩没有打上 `CLASS_ISPREVERIFIED` 和 `CLASS_ISOPTIMIZED` 标志，在类的初始化阶段，还会重新进行 类的`verify(验证)`和`optimize(优化)`
> 3. 原来验证和优化操作只有在第一次apk安装执行dexopt时，才会进行。结果如今每次进行类加载时，都会重复处理，过多的类加载同时进行，性能消耗会更大。

插桩具体性能影响

16、插桩技术对性能影响的具体测试数据

> 1. 整体上有8~9倍的性能差距
> 2. 应用启动上，容易导致白屏。

	不插桩	插桩
加载700个类	84ms	685ms
启动应用耗时	4934ms	7240ms

避免插桩的手Q方案

17、手Q方案中避免插桩的思路是什么?

> 1. 避免在 `dvmResolveClass` 中走校验dex一致性的流程.
> 2. 也就是提前将 `补丁类` 加入到数组中，让其能直接返回 `补丁类`

```
void dvmResolve(){

    ClassObject patchClass = null;

    // 1、提前将patch类加入到数组中，让patchClass!=null。
    patchClass = dvmDexGetResolved(xxx);
    if(patchClass != null){
        // 2、只要这里拿到了patchClass，就可以直接返回。
        return patchClass;
    }

    // 3、检查dex的一致性
    // xxx
    // throw dvmThrowIllegalAccessError

}
```

18、手Q方案的缺陷?

> 1. 在 `dexopt` 后进行绕过的，dexopt会改变原先的很多逻辑

2. odex层面的优化会写死字段和方法的访问偏移，就会导致严重的BUG

ART下冷启动实现

Dalvik和Art加载dex分解的区别

19、Dalvik在尝试加载一个压缩文件的时候只会把 `classes.dex` 文件加载到内存中

1. 如果压缩文件中有多个dex文件，除了 `classes.dex` 文件，其他的dex文件都会被无视

20、Art支持压缩文件中包含多个dex的加载问题

1. 会优先加载 `classes.dex` 文件
2. 然后在按顺序加载 `classes2.dex` 、 `classes3.dex` 文件
3. 如果多个dex中有同一个 类 ，只有第一个出现的 类 才会被加载，不会重复加载

Art中的方案

21、Art中进行冷启动的方案

1. 把 补丁dex 文件命名为 `classes.dex`
2. 原 `apk` 中的 `dex` 依次命名为 `classes(2,3,4...).dex` ，并一起打包为一个压缩文件。
3. 再通过 `DexFile.loadDex()` 得到 `DexFile`对象 ，并将其整个替换 旧的`dexElements`数组 即可

22、Art冷启动方案的注意点

1. 补丁dex必须命名为 `classes.dex`
2. loadDex得到的新DexFile必须完全替换掉dexElements数组，而不是插入

Tinker方案的比较

23、Tinker的冷启动方案和Sophix新方案的比较图


odex和dex

24、虚拟机真正执行的是dex文件吗?

1. `DexFile.loadDex()` 会尝试将 `dex`文件 解析并加载到 `native`内存中
2. 如果 `native`内存中 不存在dex对应的odex，Dalvik和Art分别通过 `dexopt` 、 `dexoat` 得到一个优化后的 `odex`
3. VM真正执行的是 `odex` 还不是 `dex`

25、patch不定的安全性如何保证?

1. 对补丁包进行签名校验，能保证补丁包不被篡改。
2. 但是虚拟机执行的是odex文件，而不是dex文件，还需要对 `odex`文件进行`md5`完整性校验 ，防止odex被篡改。

完整的方案

26、Dalvik和Art中完美兼容的冷启动方案

> 1. 代码采用同一套，不会根据Dalvik和Art分开处理。
> 2. Dalvik: 采用自行研发的全量Dex方案
> 3. Art：本身支持多Dex加载，只需要改名即可。

多态对冷启动类加载的影响(16题)

多态

1、多态是如何实现的?(利用的是什么技术？)

> 1. 实现多态的技术是 动态绑定
> 2. 动态绑定是指，在执行期间判断所引用对象的实际类型，根据 实际类型调用对应方法

2、field和静态方法不具备多态性

> Field如下, static方法同理:

```
class A{
  String name = "SuperClass";
}

public class B extends A{
  String name = "B";
}

A obj = new B();
System.out.println(obj.name);
// name = "SuperClass"
```

3、非静态非private方法才具有多态性

方法多态性的实现

4、方法多态性的实现流程:

```
People p = new Man();
p.talk(); // table为非静态非private的方法
```

> 1. `p.talk();` 通过指令 `invokeVirtual` 执行
> 2. 调用 `p` 的方法talk()，会拿到 该`talk()`在父类`People`的`vtable` 中的索引(`methodIndex`)
> 3. 然后在 子类`Man` 的 `vtable[methodIndex]` 中得到虚方法talk，并且执行。
> 4. 构成了多态

Virtual方法

5、Virtual方法是什么?

> 1. Virtual方法就是当前类和继承自父类的所有方法中，为 `public/protected/default`的方法

6、类加载时会创建vtable

> 1. `new B()` 时会加载类B:
>   1. 方法调用链：devmResolveClass -> dvmLinkClass -> createVtable()
>   2. createVtable(): 创建vtable，存放当前类所有 `Virtual`方法

7、createVtable的流程

> 1. 复制父类的vtable到子类的vtable
> 2. 遍历子类的virtual方法集合:
>   1. 方法原型一致 ， 表明是 重写父类方法 ， 在 相同索引处 ， 用子类方法覆盖原有 父类的方法
>   2. 方法原型不一致 ， 将子类该方法添加到 `vtable`末尾

invokeVirtual

field/static方法不具有多态

8、为什么field/static方法不具有多态性?

> 1. iget/invoke-static(虚拟机指令)是直接在 引用类型 中查找，而 不是从实际类型 中查找
> 2. 如果找不到，再去 父类中 递归查找

冷启动方案的限制

9、如果新增了一个public/protected/default方法会出现什么情况?

```
class A{
  // 新增一个方法method1
  void method1{
    // 打印method1
  }
  void method2{
    // 打印method2
  }
}

public class Demo{
  public static void test_addMethod(){
    A obj = new A();
    obj.method2();
  }
}
```

> 1. 打补丁前: 调用 方法-`method2`
> 2. 打补丁后：调用 方法-`method1`

类优化(dvmOptimizeClass)

10、类优化阶段时对虚方法调用的影响

> 1. dex文件第一次加载时，会执行 `dexopt： verify + optimize`
> 2. 类优化阶段 时，会将 `invoke-virtual` 指令替换为 `invoke-virtual-quick` 指令

11、invoke-virtual-quick指令为什么会提高方法的执行效率?

> 1. `-quick` 指令后面跟着 该方法在类`vtable`中的索引值
> 2. 会直接从类的 `vtable` 中取出方法，加快执行效率
> 3. 节省拿到 索引值 的流程

12、invoke-virtual指令的方法调用的流程?

> 1. 多了在 引用类型的`vtable`中的索引 的步骤
> 2. 然后才到 子类的`vtable` 中取出方法

13、为什么新增了public/protected/default方法会出现方法调用错乱?

> 上例分析:
>
> 1. `obj.method2()` 对应的 `-quick` 指令保存的索引值是 `0` ， 对应 `vtable[0]`
> 2. 补丁前: `vtable[0] = method2`
> 3. 补丁后: `vtable[0] = method1, vtable[1] = method2`
> 4. 最终导致方法调用错乱。

终极方案

插桩方案的失败

14、插桩方案为什么不能采用?

> 1. 通过 `Art`和`Dalvik`的冷启动方案 ， 能对 补丁类 进行加载，但是在运行时类加载的时候会出现 `dvmThrowllegalAccessError` 异常
> 2. 采用插桩方案能处理该问题，但是性能极差

非插桩手Q方案的失败

15、手Q的非插桩方案的为什么不能采用?

> 1. 通过非插桩的方法来绕过 `dex`一致性检查 ， 虽然不会抛出异常.
> 2. 但是在 多态的情况下 因为 `dexopt`的优化 导致方法调用错乱。

完整DEX方案

16、需要采用类似Tinker的完整Dex方案

> 1. google开源的 `dexmerge`方案 能将 补丁`dex` 和 原`dex` 合并成一个完整的dex
> 2. 会出现多dex下方法数超过 `65535` 的异常
> 3. dexmerge占用内存，且内存不足时有可能会失败。

4. 在移动端需要合成完整的Dex，实现较为复杂。

Dalvik中全量Dex方案(16题)

冷启动类加载修复

1、Android的冷启动类加载方案是如何实现的?

1. 把 新dex 插入到 ClassLoader 索引路径的最前面
2. 在load一个class时，优先加载补丁中的类。

2、遇到的pre-verify问题

1. 一个类中 直接引用到的所有非系统类 都和 该类 在同一个 dex 中，该类会被打上 CLASS_ISPREVERIFIED 标志
2. 具体判定代码在虚拟机中的 verifyAndOptimizeClass 函数

3、腾讯三大热修复方案如何解决 CLASS_ISPREVERIFIED 导致的异常问题?

	方案	缺点
QQ空间	插桩。 在每个类中都插入来自于一个特殊dex的 hack.，让所有类都无法满足 pre-verified 条件	性能差
Tinker	合成全量的Dex文件， 所有class都在一个dex中， 消除class重复的问题。	从dex的方法和指令的维度进行全量合成， 比较粒度过细，实现复杂， 性能消耗严重。
QFix	非插桩。利用虚拟机底层方法， 绕过 pre-verify 检查	1. 不能增加public函数

新的全量Dex方案

4、全量Dex方案

1. 将原本基线包的dex里面去除掉 补丁包中也有的class
2. 补丁 + 去除补丁类的基线包 = 新app中所有类
3. 不变的class需要用到补丁类的时候，自动地去找补丁dex
4. 新补丁类需要用到不变的class时，直接去基线包dex中寻找
5. 这样没用到补丁类的基线包class，继续通过dexopt进行处理，最大的保证了效果

5、全量Dex方案的核心在于：如何在基线包的dex文件中去除掉补丁包中的所有类

1. 从Dex Head中获取到dex的各个重要属性
2. 对于需要移除Class，不需要将其所有信息都从dex移除，只需要移除 定义的入口 即可
3. 不需要删除Class具体内容

6、如何找到某个dex的所有类定义？虚拟机在dexopt过程中是如何找到的?

> 1. dexopt的 `verifyAndOptimizeClass()` 中通过 `dexGetClassDef()` 找到的类的定义(DexClassDef *)
> 2. 内部是pHeader->classDefsOff偏移处开始，依次线性排列。

7、如何从基线包中删除目标类的 定义的入口

> 1. 直接找到 `pHeader->classDefsOff` 偏移处，遍历所有 `DexClassDef`
> 2. 如果类名包含在补丁中，就将该 `dexGetClassDef` 移除

8、Sophix是如何处理 CLASS_ISPREVERIFIED 问题的?

> 1. 补丁dex文件在补丁压缩包中，名称为 `classes.dex` ，会将该 `dex` 加载到 `dexElements`数组 中
> 2. 原apk的所有dex文件，都会被Dalvik生成 `DexFile` 加载到 `dexElements`数组 中
> 3. 这样所有类，都可以从所有dex中的某一个dex中找到。
> 4. loadDex加载删除了补丁类的原apk的dex文件时，会重新dexopt生成odex文件(CLASS_ISPREVERIFIED)标志只有满足条件的才会打上。
> 5. 当原apk中的类引用到补丁类时，因为没有 `CLASS_ISPREVERIFIED` 标志，不会出现dex一致性检查而抛出异常的情况。

9、实例解析该全量Dex方案如何解决异常问题

> 1. 错误场景：直接将补丁打入补丁包，不做额外处理
>   1. 原本APK的dex中有类A、类B
>   2. 现在类B有一个补丁类B
>   3. 单纯将补丁类打入补丁包时，此时APK的dex中有类A、类B，补丁包中有补丁类B
>   4. 程序运行时会对Apk中的类A和类B，进行 校验和优化 ，类A引用了类B，且两者位于同一个dex，给类A打上 已经校验的标志
>   5. 后续进行了处理，让类A引用类B时，能指向补丁类B。
>   6. 类A引用类B时，根据类A的 特殊标志 ，将类A和补丁类B的Dex进行校验，dex不同，抛出异常。
> 2. Sophix场景：
>   1. 原本APK中dex有类A、类B
>   2. 打补丁后，原本APK的Dex中只有类A，补丁包中有 补丁类B
>   3. Apk重新运行时，dexopt校验，类A引用补丁类B，但是两者不在同一个dex中。 校验失败
>   4. 后续运行时，类A引用补丁类B，类A不具有 特殊标志 ，不走检查dex一致性的流程，直接走重新 校验和优化 。

multidex的原理

10、multidex的原理

> 1. 将一个apk中所有类拆分到 `classes.dex、classes2.dex、classes3.dex...`
> 2. 然后将 `dex`文件 都加载进去，在运行时遇到本dex不存在的类，可以到其他dex中找

对Application的处理

11、Application的处理

> 1. Application必然是加载在原来的老dex里面。
> 2. 加载补丁后，如果Application类使用其他在新dex里的类，由于不在一个dex中，application如果被打伤了 `CLASS_ISPREVERIFIED` 标志，就会抛出异常
>   - java.lang.IllegalAccessError: Class ref in pre-verified class resolved to unexpected implementation

12、如何处理Application的pre-verified标志问题?

> 1. 将其标志位进行清除。
> 2. 在JNI层清除：类的标志位于ClassObject的accessFlags成员中

```
claszzObj->accessFlags &= ~CLASS_ISPREVERIFIED;
```

dvmOptResolveClass的问题

13、如果入口Application没有pre-verified，会有更严重的问题

> 1. Dalvik虚拟机发现某个类没有 `pre-verified` ，会在初始化类的时候，进行Verify操作
> 2. 会扫描类中使用到的所有类，并执行 `dvmOptResolveClass()`
> 3. 会在Application类初始化的时候，补丁还没加载，提前加载到原始Dex中的类
> 4. 补丁加载完毕后，已经加载的类如果用到新dex中的类，遇到 `pre-verified` 就会报错

14、问题原因

> 1. 无法把补丁加载提前到 `dvmOptResolveClass` 之前，也就是比入口Application初始化更早的时期。
> 2. 常见于多dex的情形，存在多dex时，无法保证Application用到的类和其在同一个dex中

15、多dex情况下如何解决该问题

> 1. 方法1：将Application用到的非系统类都和Application同一个dex里。保证具有 `pre-verified`标志 ，补丁加载完毕后，再清除标志。
> 2. 方法2：
>   1. 将Application中除了热修复框架的代码，都放到其他单独类中，让Application不直接调用非系统类。让其处于同一个dex。
>   2. 保险起见，Application用反射访问这个单独类中，让Application和其他类彻底隔绝。

16、Android multi-dex机制对方法1的支持

> 1. multi-dex机制会自动将Application用到的类都打包到主dex中
> 2. 只要把热修复初始化放在 `attachContext` 最前面，就OK。

资源热修复技术(25题)

普通的实现方式

Instant Run

1、Instant Run中的资源热修复的原理?

> 1. 构造一个新的 `AssetManager` .
> 2. 反射调用 `addAssetPath` ， 将这个完整的新资源包加入到 新`AssetManager` 中。
> 3. 找到所有引用 旧`AssetManager` 的地方，通过反射，将引用处替换为 新`AssetManager` -该Manager包含所有新资源

2、Instant Run的资源热修复主要工作都是在处理 兼容性 和查找到AssetManager引用处，替换逻辑很简单。

3、AssetManager是什么?

> 1. Android中有所资源包都通过AssetManager的addAssetPath()将资源路径添加进去。
> 2. Java层的AssetManager只是一个包装。底部navtive层由C++ AssetManager。

4、addAssetPath的解析流程

> 1. 通过传入的`资源包`路径，得到其中的`resources.arsc`
> 1. 解析`resources.arsc`的格式
> 1. 存放在底层`AssetManager`的`mResources`成员中

AssetManager

5、AssetManager的结构

> 1. 一个Android进程只包含一个 `ResTable`
> 2. ResTable具有成员变量 `mPackageGroups`： 包含所有解析过的资源包
> 3. 任何一个资源包中都包含 `resource.arsc`：记录所有资源的id分配情况和所有资源中的字符串
> 4. 底层AssetManager就是解析该 `resource.arsc` ， 将解析后的信息存储到 `mPackageGroup` 中

```
class AssetManager : public AAssetManager{
  mutable ResTable* mResoucres;
}
class ResTable{
  Vector<PackageGroup*> mPackageGroups;
}
```

资源文件的格式

resources.arsc

6、resources.arsc文件是什么?

> 1. 实际上由一个个 `ResChunk` 拼接起来

2. 从文件头开始，每个 ResChunk的头部 都是一个 ResChunk_header 结构，表明了 ResChunk的大小和数据类型

ResChunk_header

7、resources.arsc的解析流程

1. 通过 ResChunk_header 的 type 成员判断出其 实际类型 ，采用相应方法进行解析
2. 解析完毕后，通过 size 成员，从 ResChunk + size 得到下一个 ResChunk的起始位置
3. 依次解析完整个文件的数据内容

资源id

8、resources.arsc中包含若干个package

1. package中包含所有资源信息
2. 资源信息指:
    1. 资源名称
    2. 资源ID

9、默认情况下有aapt工具打包出来的包只有一个package

10、资源id的是一个32位数字，可以通过aapt工具解析可以看到。

1. 十六进制表示: 0xPPTTEEEE, 如: 0x7f 04 0019
2. PP是package id，如：0x7f
3. TT是类型 id，如：0x04
4. EEEE是资源项id，如：0x0019

package id

11、package id是什么?

1. 表明了是哪个资源包，如0x7f就是id = 0x7f的资源包

type id

12、type id是什么?

1. 表明资源的具体类型。
2. 依赖于 Type String Pool-类型字符串池 中具体的内容
3. 例如池中依次是 attr、drawable、mipmap、layout
4. 0x04 就表示是 layout布局类型的资源

entry id

13、 entry id是什么?

1. 资源项id，表明在 0x7f 的资源包中，类型为 0x04(layout) 中，第 0x0019 的资源项

运行时资源的解析

14、默认的apk的资源包的package id是多少?

> 1. 由 Android SDK 编译出的apk，会经过 aapt工具 进行打包的，其 package id 就是 0x7f

15、系统资源包是什么?

> 1. 系统的资源包，就是 framework-res.jar
> 2. package id = 0x01

16、资源包重复加载导致的问题

> 1. app启动时，系统会构建AssetManager并且将 0x01和0x07 的资源包添加进去
> 2. 如果通过 AssetManager.addAssetPath() 添加补丁包的资源，会导致 0x07资源包添加两次 ，会导致的问题:
>   1. Android 5.0开始，添加不会有问题，会默默将 后来的包 添加到 原来的资源的同一个PackageGroup 下面。读取时，会发现补丁包中新增的资源会生效。修改原app的资源不会生效。
>   2. Android 4.4及以下， addAssetPath 直接将补丁包的路径添加到 mAssetPath 中，但不会进行加载解析，补丁包里面的资源会完全不生效。

资源修复方案

传统方案

17、市面上的传统的方案

> 1. 对资源做差量包，运行时合成完整包再加载。
>   1. 运行时多了合成的操作，耗时，占用内存
> 2. 类似Instant Run的方案。
> 3. 修改aapt，在打包时对补丁包资源进行重新编号。对于aapt等SDK工具包的修改，不利于日后的升级。

最佳方案

18、最佳方案

> 1. 构造一个 package id = 0x66 的资源包，包含两种资源：1.新增资源 2.原有内容发生改变的资源
> 2. 直接在原有AssetManager中addAssetPath 0x66资源包 ，不和已经加载的0x7f冲突
> 3. 直接在原有的AssetManager对象上进行析构和重构。不再需要去找到所有 引用AssetManager的地方

新增资源和id偏移

19、新增资源导致id的偏移

1. 原来具有资源 0x7f020001(A图片)、0x7f020002(B图片)
2. 新增资源后是，0x7f020001(A图片)、 `0x7f020002(新图片)` 、0x7f020003(B图片)---新增资源的插入位置是随机的，跟appt有关。
3. 因为新增的资源是在 `0x66` 资源包中，打包工具需要更正id为:
    1. 原资源保持不变: 0x7f020001(A图片)、0x7f020002(B图片)
    1. 新增资源: `0x66020001(新图片)`

内容改变的资源

20、原有内容改变的资源需要代码热修复的配合

1. 原来引用资源：setContentView(0x7f030000)
2. 引用修改后的资源: setContentView(`0x66020000`)
3. 需要将 `代码中` 引用该id的方法进行修改，通过 `代码热部署` 修改引用的id

删除的资源

21、删除的资源如何处理

1. 不需要处理
2. 新代码中没有引用，自然用不到该资源。

对于type的影响

22、对补丁包中的Type String Pool需要进行修正

1. 原来池中有: attr(0x01)、drawable(0x02)
2. 因为attr的资源没有变动，所以补丁包中只有: drawable
3. 删除池中的attr，保证0x01能引用到drawable: attr(0x01)

优雅地替换AssetManager

23、在Android 5.0开始，不需要替换AssetManager

只需要在 `AssetManager` 中add进入 `0x66` 的资源包即可

24、Android 4.4及以下的版本，需要替换AssetManager

1. 这些版本调用 `AssetManager.addAssetPath` 不会加载资源，只会添加到 `mAssetPath` 中，不会解析资源包。
2. 但是不需要和 `Instant Run` 一样构造新的 `AssetManager` ，并且进行各种兼容和反射工作。

25、利用AssetManager的析构和构造方法，实现资源的真正加载。

1. 先反射调用 `AssetManager` 的 `析构方法` : 将Java层的AssetManager置为空壳(null)
2. 反射调用 `构造方法` ，调用 `addAssetPath()` 添加所有资源包： 系统会自动加载解析所有add过的资源包。

> 3. 对 mStringBlocks 置空并且重新赋值：该成员记录了所有加载过的 资源包的String pool ， 不进行重构会导致崩溃。

SO库热修复技术(22题)

SO库加载原理

1、Java API提供两个接口来加载SO库

> 1. System.loadLibrary(String libName)
>   1. 参数-SO库名称，位于 apk压缩文件 的 libs目录
>   1. 最后复制到 apk安装目录 下
> 2. System.load(String pathName)
>   1. 参数-so库在 磁盘中的完整路径
>   1. 加载一个自定义外部so库文件

2、JNI编程中，native方法分为 动态注册 和 静态注册 两种

动态注册

3、动态注册的native方法

> 1. 必须实现 JNI_OnLoad方法
> 2. 需要实现一个 JNINativeMethod[]数组
> 3. 动态注册的native方法映射 通过加载so库过程中调用 JNI_OnLoad 方法调用完成

静态注册

4、静态注册的native方法

> 1. 必须是 Java + 类完整路径 + 方法名 的格式
> 2. 静态注册的native方法映射 是在 该native方法第一次执行时 完成。 前提该so库已经load过

SO库热部署方案

动态注册native方法

5、动态注册的native方法实时生效的方案?

> 1. 该方法每调用一次 JNI_OnLoad 方法就会 重新进行一次映射
> 2. 先加载 原来的so库 ， 再加载 补丁so库 ， 就能映射为补丁中的新方法

Art

6、ART中能做到实时生效(热部署)吗?

> 可以

Dalvik

7、Dalvik中能做到实时生效(热部署)吗?

> 1. 不能
> 2. 第二次load补丁so库，依旧执行的是 原来so库的`JNI_OnLoad()`方法

8、为什么Dalvik加载补丁so库，执行的是原始so库的load方法?

> 1. 下面两个方法可能有问题:
>   1. dlopen: 返回动态链接库的句柄
>   2. dlsym: 通过dlopen得到的句柄，来查找一个 `symbol`
> 2. `dlopen` 具有bug:
>   1. Dalvik中通过so的name去 `solist` 中查找，因为加载原始so库时，该列表中已经存储，直接返回原始so库的句柄

9、Dalvik的Bug如何规避?

> 1. 对 补丁so库 进行改名
> 2. 原始so库路径为: /data/data/.../files/libnative-lib.so
> 3. 补丁so库路径改为：/data/data/.../files/libnative-lib-时间戳.so
> 4. 通过改名，添加时间戳，保证 `name` 是 全局唯一 ，这样能正确得到 动态链接库的句柄

静态注册native方法

10、静态注册native方法的映射都是在native方法的第一次执行时完成的映射。如果native方法在加载补丁so库前已经执行过了。会出现问题。

11、如果保证静态注册native方法能够热部署?

> 1. JNI API提供了 解注册的接口
> 2. 把目标类的所有native方法都解注册，无论是动态注册还是静态注册的，后面都需要重新 映射

```
env->UnregisterNative(claze);
```

12、经过解注册处理后，热部署也不一定会成功

> 1. 补丁so库是否能成功加载，取决于在hashtable中的位置
> 2. 如果顺序是：补丁so库、原始so库。则 热部署修复 成功。
> 3. 如果顺序是: 原始so库、补丁so库。则失败。

SO库冷部署方案

接口调用替换方案

13、使用sdk提供的接口替换System加载so库的接口

1. 用 SOPatchManager.loadLibrary() 替代 System.loadLibrary()，优先加载sdk指定目录下的补丁so
2. SOPatchManager.loadLibrary() 的加载策略如下:
    1. 如果存在补丁so库，直接加载。
    2. 如果不存在补丁so库，调用 System.loadLibrary 加载apk目录下的so库

14、替换方案的优点

1. 不需要对不同sdk版本进行兼容，因为都有 System.loadLibrary 这个接口

15、替换方案的缺点

1. 调用 System接口 的地方都需要替换为 sdk的接口
2. 如果是已经混淆好的第三方库的so库，无法进行接口替换。

反射注入方案

16、System.loadLibrary("native-lib")的底层原理

1. 本质是在 nativeLibraryDirectories数组 中遍历

17、反射注入方案

1. 将 补丁so库 的路径，插入到 nativeLibraryDirectories数组的最前面
2. 就能达到加载so库时，直接加载补丁so库的目的。

sdk23前后的区别

18、sdk < 23时，只需要将 补丁so库 的路径，插入到 nativeLibraryDirectories数组的最前面

19、sdk >= 23时，需要用 补丁so库路径 来构建 Element对象 ，然后插入到 nativeLibraryPathElements数组的最前面

20、反射注入方案的优缺点

1. 优点: 不具有侵入性
2. 缺点: 需要针对sdk进行适配

机型对应的so库

21、so库具有多种cpu架构的so文件

1. 如 armeabi 、 arm64-v8a 、 x86
2. 难点在于如何根据机型，选择对应的so库文件

22、如何选择机型最合适的primaryCpuAbi so文件?

1. sdk>=21 ，反射拿到 ApplicationInfo对象 的 primaryCpuAbi 即可
2. sdk<21 ，不支持64位，直接把 Build.CPU_ABI 、 Build.CPU_ABI2 作为 primaryCpuAbi 即可

问题汇总

参考资料

1. 混淆库官方文档
2. Android 新一代编译 toolchain Jack & Jill 简介
3. 官方文档Compiling with Jack
4. gradle中如何使用Lambda
5. Android动态链接库加载原理及HotFix方案介绍