在 Android Runtime (ART) 上验证应用行为

本文内容

另请参阅

Android Runtime (ART) 是运行 Android 5.0（API 级别 21）及更高版本的设备的默认运行时。 此运行时提供了多种可改善 Android 平台和应用的性能和流畅度的功能。 您可以在 ART 简介中找到关于 ART 新功能的更多信息。

不过，部分适合 Dalvik 的技术并不适用于 ART。本文档可帮助您了解在迁移现有应用，使其与 ART 兼容时需要注意的事项。 大多数应用在使用 ART 运行时都能正常工作。

解决垃圾回收 (GC) 问题

在 Dalvik 中，应用常常发现显式调用 `System.gc()` 非常有用，可促进垃圾回收 (GC)。对 ART 而言这种做法的必要性低得多，尤其是当您需要通过垃圾回收来预防出现 `GC_FOR_ALLOC` 类型或减少碎片时。 您可以通过调用 `System.getProperty("java.vm.version")` 来验证正在使用哪种运行时。 如果使用的是 ART，则该属性值将是 `"2.0.0"` 或更高。

而且，Android 开源项目 (AOSP) 中正在开发一种紧凑型垃圾回收器，以改善内存管理。 因此，您应该避免使用与紧凑型 GC 不兼容的方法（例如保存对象实例数据的指针）。 这对于使用 Java 原生接口 (JNI) 的应用而言尤其重要。 如需了解详细信息，请参阅预防 JNI 问题。

预防 JNI 问题

ART 的 JNI 比 Dalvik 的 JNI 更为严格一些。使用 CheckJNI 模式来捕获常见问题是一种特别实用的方法。 如果您的应用使用 C/C++ 代码，您应该阅读以下文章：

使用 CheckJNI 调试 Android JNI

检查 JNI 代码中的垃圾回收问题

ART 在 Android 开源项目 (AOSP) 有正在开发中的紧凑型垃圾回收器。 一旦该紧凑型垃圾回收器投入使用，便可在内存中移动对象。 如果您使用 C/C++ 代码，请勿执行与紧凑型 GC 不兼容的操作。 我们对 CheckJNI 进行了增强，以识别一些潜在的问题（如 ICS 中的 JNI 局部引用更改中所述）。

需要特别注意的一个方面是 `Get...ArrayElements()` 和 `Release...ArrayElements()` 函数的使用。 在包含非紧凑型 GC 的运行时中，`Get...ArrayElements()` 函数通常返回支持数组对象的实际内存的引用。 如果对其中一个返回的数组元素执行更改，数组对象本身将被更改（并且 `Release...ArrayElements()` 的参数往往会被忽略）。 但如果正在使用的是紧凑型 GC，则 `Get...ArrayElements()` 函数可能返回内存的副本。 如果您在使用紧凑型 GC 的情况下误用引用方法，可能会导致内存崩溃或其他问题。 例如：

- 如果您对返回的数组元素执行任何更改，则在完成更改后必须调用相应的 `Release...ArrayElements()` 函数，以确保您所做的更改已正确地复制回基础数组对象。
- 在您释放内存数组元素时，必须根据所做的更改使用相应的模式：
  - 如果您没有对数组元素执行任何更改，请使用 `JNI_ABORT` 模式，该模式会释放内存，而不将更改复制回基础数组元素。
  - 如果您对数组执行了更改，并且不再需要该引用，请使用代码 `0`（它将更新数组对象并释放内存副本）。
  - 如果您对您想要提交的数组执行了更改，并且您希望保留该数组的副本，请使用 `JNI_COMMIT`（它将更新基础数组对象并保留该副本）。
- 调用 `Release...ArrayElements()` 时，将返回最初由 `Get...ArrayElements()` 返回的相同指针。 例如，递增原始指针（以扫描所有返回的数组元素），然后将递增的指针传递至 `Release...ArrayElements()` 是不安全的做法。 传递此修改后的指针可能导致释放错误的内存，进而导致内存崩溃。

错误处理

ART 的 JNI 会在多种情况下引发错误，而 Dalvik 则不然。（同样地，您可以通过使用 CheckJNI 执行测试来捕获大量此种情况）。

例如，如果使用不存在的方法（可能由于该方法已被 **ProGuard** 等工具移除）调用 `RegisterNatives`，ART 现在会正确地引发 `NoSuchMethodError`：

```
08-12 17:09:41.082 13823 13823 E AndroidRuntime: FATAL EXCEPTION: main
08-12 17:09:41.082 13823 13823 E AndroidRuntime: java.lang.NoSuchMethodError:
    no static or non-static method
    "Lcom/foo/Bar;.native_frob(Ljava/lang/String;)I"
08-12 17:09:41.082 13823 13823 E AndroidRuntime:
    at java.lang.Runtime.nativeLoad(Native Method)
08-12 17:09:41.082 13823 13823 E AndroidRuntime:
    at java.lang.Runtime.doLoad(Runtime.java:421)
08-12 17:09:41.082 13823 13823 E AndroidRuntime:
    at java.lang.Runtime.loadLibrary(Runtime.java:362)
08-12 17:09:41.082 13823 13823 E AndroidRuntime:
    at java.lang.System.loadLibrary(System.java:526)
```

如果不使用任何方法调用 `RegisterNatives`，ART 也会记录错误（在 logcat 中可见）：

```
W/art     ( 1234): JNI RegisterNativeMethods: attempt to register 0 native
methods for <classname>
```

此外，JNI 函数 `GetFieldID()` 和`GetStaticFieldID()` 现在会正确地引发 `NoSuchFieldError`，而不是仅仅返回 null。 类似地，`GetMethodID()` 和 `GetStaticMethodID()` 现在会正确地引发 `NoSuchMethodError`。这可能会导致 CheckJNI 由于未处理的异常或引发至原生代码的 Java 调用函数的异常而失败。 这让使用 CheckJNI 模式测试 ART 兼容型应用变得格外重要。

ART 预期 JNI `CallNonvirtual...Method()` 方法（例如 `CallNonvirtualVoidMethod()`）的用户按照 JNI 规范的要求，使用该方法的声明类而不是子类。

预防堆栈大小问题

Dalvik 具有单独的原生代码堆栈和 Java 代码堆栈，并且默认的 Java 堆栈大小为 32KB，默认的原生堆栈大小为 1MB。 ART 具有统一的堆栈以改善局部性。 通常情况下，ART `Thread` 堆栈大小应该与 Dalvik 堆栈大小近乎相同。 但如果您显式设置了堆栈大小，则可能需要针对 ART 中运行的应用重新访问这些值。

- 在 Java 中，查看用于指定显式堆栈大小的 `Thread` 构造函数的调用。 例如，如果发生 `StackOverflowError`，您将需要增加该大小。
- 在 C/C++ 中，查看如何将 `pthread_attr_setstack()` 和 `pthread_attr_setstacksize()` 用于同时通过 JNI 运行 Java 代码的线程。以下是某个应用在 pthread 过小的情况下尝试调用 JNI `AttachCurrentThread()` 时记录的错误示例：

  ```
  F/art: art/runtime/thread.cc:435]
      Attempt to attach a thread with a too-small stack (16384 bytes)
  ```

对象模型更改

对象模型更改

Dalvik 错误地允许子类覆盖包私有的方法。ART 在这类情况下会发出警告：

```
Before Android 4.1, method void com.foo.Bar.quux()
would have incorrectly overridden the package-private method in
com.quux.Quux
```

如果您希望在另一个包中覆盖某个类的方法，请将该方法声明为 `public` 或 `protected`。

`Object` 现在包含私有字段。对于反射其类层次中的字段的应用，应小心避免尝试查看 `Object` 的字段。 例如，如果您正在向上迭代某个作为串行化框架一部分的类层次，以下情况下请停止迭代操作

```
Class.getSuperclass() == java.lang.Object.class
```

而不是继续操作，直至该方法返回 `null`。

如果没有任何参数，代理 `InvocationHandler.invoke()` 现在将会收到 `null`，而不是空数组。 之前记录过此行为，但在 Dalvik 中未得到正确处理。 之前版本的 Mockito 难以处理这一问题，因此在使用 ART 测试时请使用更新的 Mockito 版本。

修复 AOT 编译问题

ART 的提前 (AOT) Java 编译应适用于所有标准 Java 代码。 编译由 ART 的 `dex2oat` 工具执行，如果您在安装时遇到任何与 `dex2oat` 有关的问题，请联系我们（请参阅报告问题），以便我们能够尽快将其修复。 需要注意的几个问题：

- ART 会在安装时执行比 Dalvik 更严格的字节代码验证。 Android 构建工具生成的代码应该没有问题。但一些后期处理工具（尤其是执行模糊处理的工具）可能会生成被 Dalvik 容忍而被 ART 拒绝的无效文件。 我们已经与工具供应商合作，查找并修复此类问题。 在许多情况下，获取最新版本的工具并重新生成 DEX 文件可以修复这些问题。
- 一些被 ART 验证器标记的典型问题包括：
  - 无效的控制流
  - 失衡的 `moniterenter`/`moniterexit`
  - 0 长度参数类型列表大小
- 一些应用对 `/system/framework`、`/data/dalvik-cache` 中或 `DexClassLoader` 的优化输出目录中的安装的 `.odex` 文件格式具有依赖性。 这些文件现在是 ELF 文件，而不是 DEX 文件的扩展形式。 尽管 ART 努力遵循与 Dalvik 相同的命名和锁定规则，但应用不能依赖于文件格式，因为该格式可能未经通知便发生更改。

报告问题

如果您遇到任何不是由于应用 JNI 问题而导致的问题，请通过位于 https://code.google.com/p/android/issues/list 的 Android 开源项目问题跟踪器报告这些问题。请包含 `"adb bugreport"` 和 Google Play 商店中的应用链接（如果可用）。 否则，如果可能，请附加用于重现该问题的 APK。 请注意，这些问题（包括附件）是公开可见的。