热部署技术调研报告

目录

小组成员

隆晋威

吴昊

魏天心

伊昕宇

一、项目背景

热部署 (hot deployment) 简介

热部署是指让应用能够在无需重新安装的情况实现更新，帮助应用快速建立动态修改能力。由于用户总是希望服务进程能保持稳定，如果服务器可以 24 小时工作的话，用户会希望永远不要重启服务器。但是，产品的功能总是在不断的丰富，只要产品仍在生命周期以内，就不可避免会面临版本升级的问题。此外，产品运营能力的提升也是靠版本更新迭代来实现，轻量级版本更新更是被视为整个产品精细化运营的基础。另一方面，如果发生大规模的宕机或者运行大型程序时出现问题，通过热部署可以快速修补尽并最大可能的减少损失，而不用重新启动应用。对于移动 APP 等轻型应用，热部署可以能够缩短用户取得新版客户端的流程，改善用户体验，省去用户自行更新客户端的步骤。总而言之，如果更新过程中，业务进程不用重启，那么这将会极大地方便我们的生产、生活；从此以后，升级将不再是一个令人烦恼的事情了。

热部署技术的应用现状

热部署技术是一项非常有意义的技术，工业界已经有一些软件具备热部署的能力。需要插件来进行功能扩展的应用程序，插件的安装和部署很多都是不需要重启应用的，例如：Firefox 、 Chrome、Eclipse、IntelliJ IDEA 等。当然，也有的软件不支持热部署，如 Visual Studio Code，虽然是用 nodejs 开发的，但插件安装之后却需要重启程序才能生效，这一点饱受用户诟病。

然而，现有的热部署技术大都和平台高度相关，对于动态语言来说，每个语言都有自己的一套解决方案，例如：Python、PHP、Lua 等动态语言，大都有自己的动态加载框架和策略，有的还有 JIT (即时编译技术)。Python 和 PHP 常用于构建 web 服务程序，这些程序如果在更新时停机，会造成严重的后果。Java 是静态语言，它的灵活性比不上动态语言，但因为 Java 有强大的虚拟机，因此也能在一定程度上进行热部署。

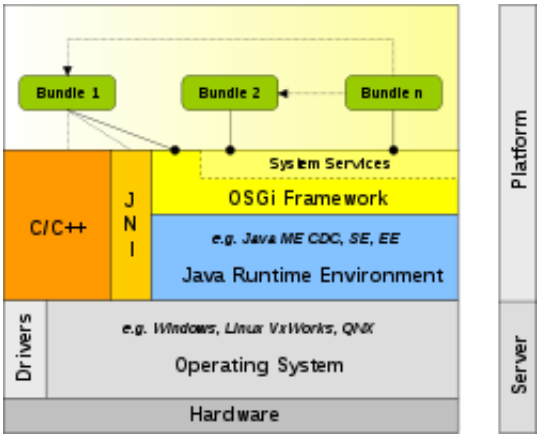
相关行业规范：OSGi 规范

OSGi (Open Service Gateway Initiative) 技术是 Java 动态化模块化系统的一系列规范。OSGi 一方面指维护 OSGi 规范的 OSGI 官方联盟，另一方面指的是该组织维护的基于 Java 语言的服务（业务）规范。简单来说，OSGi 可以认为是 Java 平台的模块层。

OSGi 服务平台向 Java 提供服务，这些服务使 Java 成为软件集成和软件开发的首选环境。Java 提供在多个平台支持产品的可移植性。OSGi 技术提供允许应用程序使用精炼、可重用和可协作的组件构建的标准化原语，这些组件能够组装进一个应用和部署中。

OSGi 服务平台提供在多种网络设备上无需重启的动态改变构造的功能。为了最小化耦合度和促使这些耦合度可管理，OSGi 技术提供一种面向服务的架构，它能使这些组件动态地发现对方。OSGi 联盟已经开发了例如像 HTTP 服务器、配置、日志、安全、用户管理、XML等很多公共功能标准组件接口。这些组件的兼容性插件实现可以从进行了不同优化和使用代价的不同计算机服务提供商得到。然而，服务接口能够基于专有权基础上开发。

下图是 OSGi 架构



OSGi 框架涵盖三个主要方面：软件包，生命周期和服务。 Bundle 层是 OSGi 框架中最明显，最常用的部分。简而言之，Bundle 层被表示为一个 JAR 文件（Java ARchive），其中包含一些额外信息，以及它的依赖关系。生命周期层定义了 Bundle 在安装，启动，更新，停止和卸载时所经历的一系列步骤。明确定义生命周期允许包中的代码管理自己的资源。同样重要的是，生命周期层可以帮助管理员及早发现问题，因为OSGi强制要求在使用包之前解决所有外部依赖关系。如果依赖关系无法解析，则在软件包启动之前记录错误。Services层公开了运行代码对象的服务，这些代码对象可以从OSGi服务器中运行的其他代码调用。 OSGi服务的最大区别在于，框架允许服务实现在运行时更改，即热部署。

然而，OSGi 规范并不是专为实现热部署而确立的，但它低耦合的架构和代码规范为各个模块提供了较好的可迁移性。再配合上强大的 Java 虚拟机，使得遵循 OSGi 规范的软件系统都具备一定程度的热部署能力。Apache Felix，Equinox，Knopflerfish 等都提供了运行 OSGi 程序的容器。

二、立项依据

热部署技术的应用场景

1. 轻量而快速的升级

热部署技术可以理解为一个动态修改代码与资源的通道，它适合于修改量较少的情况。以微信的多次发布为例，补丁大小均在300K以内，它相对于传统的发布有着很大的优势。

	普通升级	补丁升级
数据大小	32.5M	150K
更新速度 (0-50%)	10天	1天 (70%)
自动升级	WIFI	移动

2. 在不宕机的情况下更新应用

在某些应用场景中，生产环境宕机会造成非常严重的后果。

比如，2012年12月22日，全球知名的开源托管服务GitHub意外遭遇了历史上最严重的一次宕机事件，而GitHub官方确认了本次宕机是在例行的软件升级过程中发生的。众所周知，GitHub一旦宕机，会使得全球各地近 2400 万名开发者的协作开发被迫中断，如果发生了代码库丢失等事故，造成的后果更是不堪设想。

热部署技术实现思路

A. 从 Unix 系统层面入手

要在系统层面上实现热部署，业界主流方案是通过动态链接库进行代码的修改和动态加载。Windows 系统使用`.dll`文件作为其执行程序的动态链接库(Dynamic Link Library)，而 Unix 操作系统使用`.so`文件作为其动态库。我们计划针对 Unix 平台实现热部署功能，而 Unix 操作系统中的动态库`.so`文件是一种 **ELF 文件**，因此我们需要对 **ELF 文件** 有足够深入的研究。

1. ELF 文件介绍

ELF 文件是可执行与可链接格式的英文缩写(英语：Executable and Linkable Format)，这是一种在 Unix 系统是一种用于可执行文件、目标代码、共享库和核心转储(Core Dump)的标准文件格式。

2. ELF 文件分类

i. 可重定位的对象文件(Relocatable File)

这是由汇编器汇编生成的 .o 文件。后面的链接器拿它作为输入，经链接处理后，生成一个可执行的对象文件 (Executable File) 或者一个可被共享的对象文件(Shared Object File)

ii. 可执行的对象文件(Executable File)

很常见，如文本编辑器vi等等。可执行的脚本(如shell脚本)不是 Executable Object File，它们只是文本文件。

iii. 可被共享的对象文件(Shared Object File)

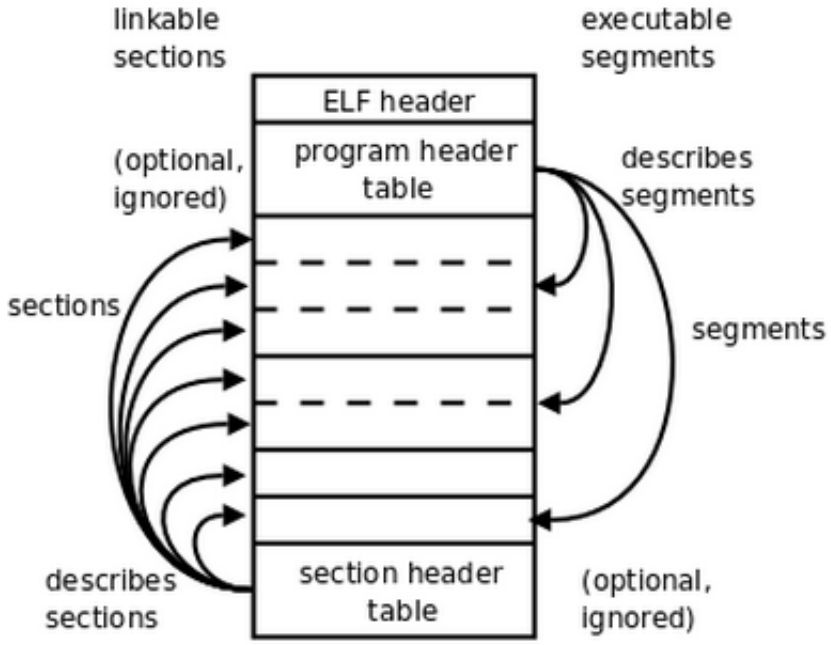
这就是所谓的动态库文件，也即 .so 文件。动态库在发挥作用的过程中，必须经过两个步骤：

1. 链接编辑器拿它和其他可重定位对象文件(Relocatable Object File)以及共享对象文件(Shared Object File)作为输入，经链接处理后，生成另外的共享对象文件或者可执行文件。
2. 运行时，动态链接器(Dynamic Linker)拿它和一个Executable File以及另外一些 Shared Object File 来一起处理，在Linux系统里面创建一个进程映像。

其中，实现热部署的关键，就是对 .so 文件的修改与重新加载

3. ELF 文件组成

ELF 文件大都包含 ELF 头部、程序头部表、节区或段、节区头部表。 ELF 头部用来描述整个文件的组织。节区部分包含链接视图的指令、数据、符号表、重定位信息等。程序头部表告诉系统如何创建进程映像。节区头部表包含了描述文件节区的信息，如名称、大小。



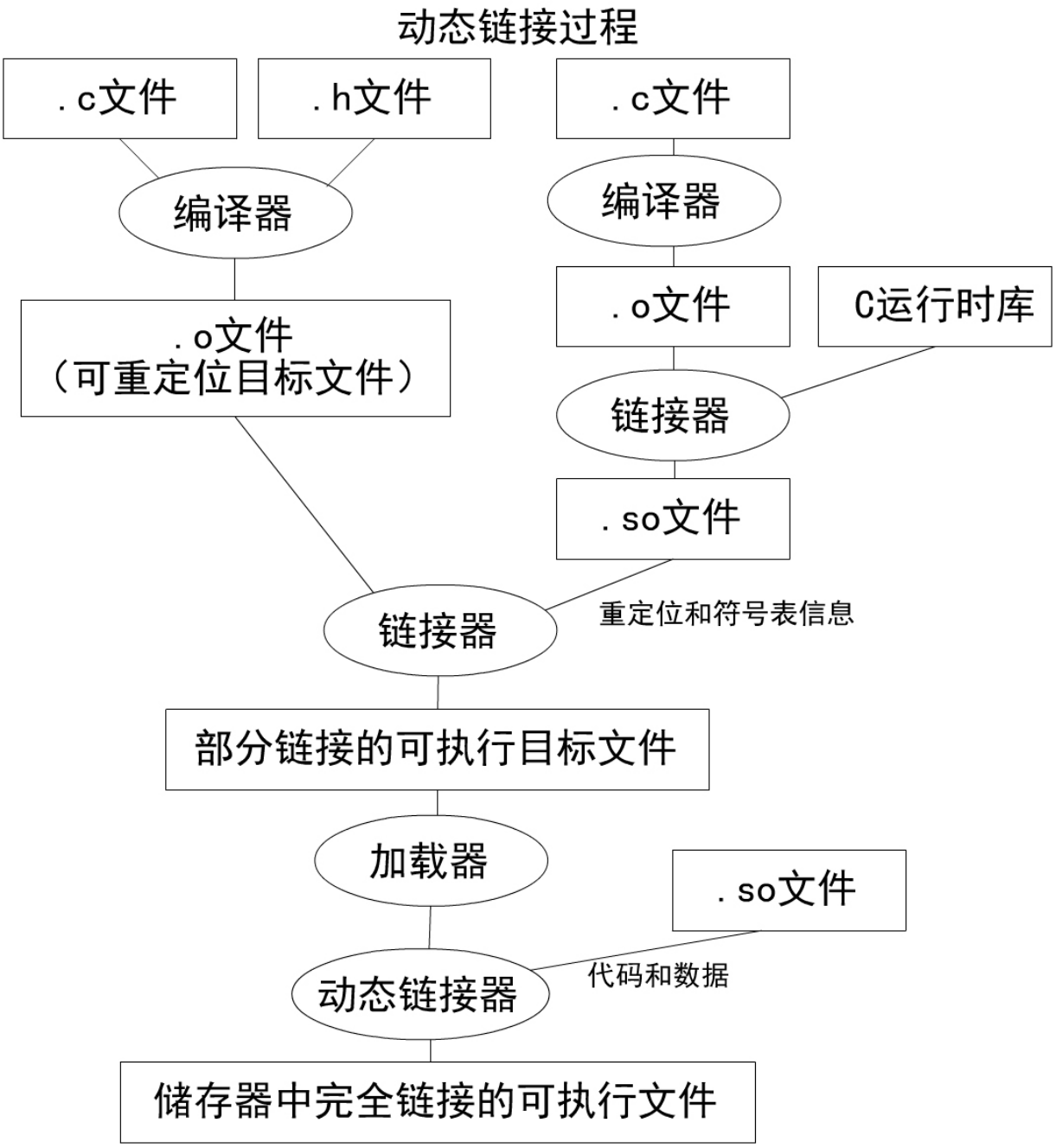
- 组成不同的可重定位文件（.o 文件）参与可执行文件或者可被共享的对象文件的链接构建。不需要程序头部表。必须包含节区头部表。以节为单位。
- 组成可执行文件或者可被共享的对象文件（.so文件）在运行时内存中进程映像的构建。 必须具有程序头部表。可以无节区头部表。以段为单位。

可重定位文件	可执行或可被共享的对象文件
ELF头部	ELF头部
程序头部表（可无）	程序头部表（必须）
很多节区	很多段
节区头部表（必须）	节区头部表（可无）

4. 动态链接过程

一个程序要想装入内存运行必然要先经过编译、链接和装入这三个阶段。人们考虑将如果两个程序用到相同的函数库，那么理想的情况是系统中只保存一份函数库的拷贝（无论是在内存中还是在硬盘上），于是人们想到了动态链接的方法来实现函数库的复用：动态链接使得两个进程在内存中通过将地址映射到相同的 .o 文件实现对其的共享。动态链接的这一特性对于库的升级（比如错误的修正）是至关重要的。当一个库升级到一个新版本时，所有用到这个库的程序将自动使用新的版本；如果不使用动态链接技术，那么所有这些程序都需要被重新链接才能得以访问新版的库。这样的系统被称作共享库系统。我们在热部署的实现中正好可以利用这个特性来实现我们的目的，即如果我们要修改一个函数，我们将新的函数存储在动态链接库中，然后将旧的函数名重新链接（这是个动态链接过程）到

在动态链接库中的函数地址。



B. 借鉴 JIT 实现思路

JIT 是什么

代码有两种常见的执行方式：一种叫做编译执行，也就是直接将代码转换为CPU指令，然后连续执行这些指令，好处当然是非常快，一条多余的指令都没有；坏处则是难以支持许多动态特性。一种叫做解释执行，也就是对每条语句在运行时用解释器去执行它相应的动作，好处是实现起来非常简单，也很容易添加新特性，坏处则是执行得非常慢，大部分CPU时间花在了解释器运行上面。JIT技术是两者的结合，首先让代码解释执行，同时收集信息，在收集到足够信息的时候，将代码动态编译成CPU指令，然后用CPU指令替代解释执行的过程，因为编译发生在马上要执行之前，所以叫做Just-In-Time Compiler。编译之后速度就是编译执行的速度了，自然比解释执行要快得多。javac 把 java 的源文件翻译成了 class 文件，而 class 文件中全都是 Java 字节码。那么，JVM 在加载了这些 class 文件以后，针对这些字节码，逐条取出，逐条执行。

还有一种，就是把这些 Java 字节码重新编译优化，生成机器码，让 CPU 直接执行。这样编出来的代码效率会更高。通常，我们不必把所有的 Java 方法都编译成机器码，只需要把调用最频繁，占据 CPU 时间最长的方法找出来将其编译成机器码。这种调用最频繁的 Java 方法就是我们常说的热点方法（Hotspot，说不定这个虚拟机的名字就是从这里来的）。

这种在运行时按需编译的方式就是 Just In Time。

JIT的好处

JIT的好处在于兼顾了语言的动态特性和程序的执行效率。例如， PyPy 的性能在没有 JIT 的情况下和 CPython 是差不多的 (大概慢一到四倍 [1])，用了 JIT 就能超出几十到数百倍都有可能，这是因为 CPython 不支持 JIT，因此为了满足动态特性，不得不考虑传入参数的各种类型（也就是多态）。一个例子如下： 为了实现下面这段 Python 代码

```
def add(x, y):
    return x + y
```

CPython 的实现：

```
if (instance_has_method(x, '__add__')) {
    return call(x, '__add__', y);
    }
else if (isinstance_has_method(super_class(x), '__add__') {
    return call(super_class, '__add__', y);
} else if (isinstance(x, str) and isinstance(y, str)) {
    return concat_str(x, y);
} else if (isinstance(x, float) and isinstance(y, float)) {
    return add_float(x, y);
} else if (isinstance(x, int) and isinstance(y, int)) {
    return add_int(x, y);
} else
  return 0;
```

如果我们仅仅执行整数的加法，CPython 在条件判断上耗费的时间会很多，而应用了 JIT 特性的 PyPython 会这样实现：

```
int add_int_int(int x, int y) {
  return x + y;
}
```

这样就能节省大量的时间了。

JIT 主要技术点

其实 JIT 的主要技术点，从大的框架上来说，非常简单，就是申请一块既有写权限又有执行权限的内存，然后把你要编译的 Java 方法，翻译成机器码，写入到这块内存里。当再需要调用原来的 Java 方法时，就转向调用这块内存。一个经典的例子是，我们使用 mmap 来申请了一块有写权限和执行权限的内存，然后把我们手写的机器码拷进去，然后使用一个函数指针指向这块内存，并且调用它。通过

这种方式我们就可以执行这一段手写的机器码了。比如这个例子：

```
#include<stdio.h>
#include<memory.h>
#include<sys/mman.h>

typedef int (* inc_func)(int a);

int main() {
    char code[] = {
        0x55,             // push rbp
        0x48, 0x89, 0xe5, // mov rsp, rbp
        0x89, 0xf8,       // mov edi, eax
        0x83, 0xc0, 0x01, // add $1, eax
        0x5d,             // pop rbp
        0xc3              // ret
    };

    void * temp = mmap(NULL, sizeof(code), PROT_WRITE | PROT_EXEC,
            MAP_ANONYMOUS | MAP_PRIVATE, -1, 0);

    memcpy(temp, code, sizeof(code));
    inc_func p_inc = (inc_func)temp;
    printf("%d\n", p_inc(7));

    return 0;
}
```

这样，我们就通过手写机器码把原来的 inc 函数代替掉了。在新的例子中，我们是使用程序中定义的数据来重新造了一个 inc 函数。这种在运行的过程创建新的函数的方式，就是JIT的核心操作。

JIT 解释器和 C1，C2

解释器为每一个字节码生成一小段机器码，在执行 Java 方法的过程中，每次取一条指令，然后就去执行这一个指令所对应的那一段机器码。256 条指令，就组成了一个表，在这个表里，每一条指令都对应一段机器码，当执行到某一条指令时，就从这个表里去查这段机器码，并且通过 jmp 指令去执行这段机器码就行了。

这种方式被称为模板解释器。

模板解释器生成的代码有很多冗余，就像我们上面的第一个例子那样。为了生成更精简的机器码，我们可以引入编译器优化手段，例如全局值编码，死代码消除，标量展开，公共子表达式消除，常量传播等等。这样生成出来的机器码会更加优化。

但是，生成机器码的质量越高，所需要的时间也就越长。JIT 线程也是要挤占 Java 应用线程的资源的。所以 C1 是一个折衷，编译时间既不会太长，生成的机器码的指令也不是最优化的，但肯定比解释器的效率要高很多。

如果一个 Java 方法调用得足够频繁，那就更值得花大力气去为它生成更优质的机器码，这时就会触发 C2 编译，C2 是一个运行得更慢，但却能生成更高效代码的编译器。

由此，我们看到，其实 Java 的运行，几乎全部都依赖运行时生成的机器码上。而 Java 的运行依赖模板解释器和 JIT 编译器。为了能动态修改，动态加载我们执行的二进制代码，我们也不妨参考这种运行时编译（JIT）的方式。

三、重要性以及前瞻性分析

热部署是关联性强，实用性高并能显著提升各程序执行效率和开发效率的战略性技术，从谷歌微软中的大型服务器，到各类游戏厂商和手机 APP 的更新下载，再到我们生活中的日常使用，总会有功能不够完善需要升级，或者是犯错想要修改的时候。而如果此时程序正在运行，重启更新代价又过高，这就是热部署发挥功效的时候。

游戏市场

举一些例子，比如在游戏市场中，刚出的全新游戏产品（如 2017年风靡全国 PC 端游戏市场的绝地求生），在巨量用户体验游戏时，由于玩家的各种行为是不可预料的，当玩家作出一些设计师尚未考虑到的行为时候，就会出现不可预料的 bug。如果你的游戏不支持热更新技术，修复 bug 的补丁分发流程大致如下：

> pc用户：
>
> 更新客户端 -> 等待下载 -> 安装客户端 -> 等待安装 -> 启动游戏 -> 开始游戏
>
> 手机用户：
>
> 商城更新APP -> 等待下载 -> 等待安装更新 -> 启动游戏 -> 开始游戏

为了不影响玩家体验，来修复一些轻微bug，使用热部署技术之后的操作流程如下：

> pc用户：
>
> 启动 -> 联网安装热部署补丁 -> 开始游戏
>
> 手机用户：
>
> 启动游戏 -> 联网安装热部署补丁 -> 开始游戏

对用户来说能获得更好游戏体验，对开发者来说又能保证服务器稳定运行，适应上线需求，节省时间。

APP市场

再比如同样对手机上的很多 app，不支持热更新的话，如果开发者发布了一些小的漏洞修正补丁，用户需要重新下载安装包，浪费了用户的时间，会造成用户流失以及版本更新率降低，这样就降低了产品的竞争力，而如果使用热部署技术的话，比如使用 **Robust** （美团）或者是 **Tinker** （腾讯）这些方案，就无需重新发版，实时高效修复升级，可以做到用户无感知，无需下载新的应用，而大小也会因为只包含了新增代码而比原始安装包要小得多，代价小，成功率高，把损失降到最低。

服务器等应用

还有时常能从新闻上看到的大型服务器的宕机，对这些大公司来说，每一秒的停滞都是成千上万的损失，而更甚的是可能会造成用户隐私的泄漏和公司信誉的破产，而在这种情况热部署的实现并迅速修改显得尤为重要，就比如前面所提到的github宕机事件，GitHub 的用户中包括大部分美国科技巨头例如 Twitter、Facebook、Google 以及微软等。GitHub 员工曾在微博上透露，中国已是 Github 第二大用户，国内的科技巨头均是 Github 的用户，例如，支付宝网页用到的前端模块 SeaJS、腾讯的移动 Web 前端知识库 Mars 以及百度的百度图说。此外，不少创业公司也把部分软件代码放到 GitHub，方便平时开发协作，包括各大高校和很多学生也将自己的代码数据上传其中，而这一宕机如果造成这些代码数据的丢失，损失会是难以估计的，如果整个停止重写就会造成很长时间的浪费，而通过热部署就可以快速地更新，尽量地减少损失。

再比如科研等工作中有时候会用到超级计算器这种成本很高的工具，如果程序执行到一半发现代码中有疏漏，或者你正在爬取处理大量数据，已经执行了很多工作，这时重新开始就会浪费大量时间、算力，得不偿失，而热部署可以很好地解决这些问题,可以在不停止运行的情况下来更新解决现有的问题，降低了风险，也减轻了负担。

总结规划

热部署这一强有力的技术已经在不知不觉间影响着我们的生活，并发挥着不可或缺的作用，虽然谷歌微软等巨头都在不断地开发更新更强力的热部署工具，工业界也已经有一些软件具备热部署的能力，需要插件来进行功能扩展的应用程序，插件的安装和部署很多都是不需要重启应用的，但仍存在着他们力所不能及的一些有市场，有价值的需求和亟待解决的问题，而我们的目标就是通过动态链接库进行代码的修改和动态加载，对 .so 动态库文件进行修改与重新加载，在操作系统层面提供动态链接的支持来给程序提供一个通用的热部署的框架，众所周知，已经支持热部署的更新很简单，但我们的重点是尽可能地做到给开发时不支持热部署的程序提供热部署服务，让原本只要更新或安装插件后就需要重启而饱受诟病的软件能得到改善，不仅用来方便我们的工作和生活，也希望能够被用以改良原有的工作，焕发出新的活力。

四、相关工作

Java 虚拟机上的热部署

Java 程序在执行之前，会被 Java 编译器编译为一种中间语言: Java 字节码。这些字节通常以即 *.class 文件的形式存在，在 Java 虚拟机执行前，*.class 文件中有较多与类相关的元信息，例如类的字段表、属性表等。Java 字节码是一种基于栈的中间语言，既可以直接被 Java 虚拟机解释执行，也可以通过 JIT (Just-in-time compilation) 技术编译成机器码之后执行。

Java 虚拟机原生支持动态加载代码，ClassLoader 类可以用来在运行期装载新的类。但因为 Java 虚拟机的某些设计原因，它并不支持直接替换原有的类，而只能加载一个新的类。为了能实现热部署，各大组织和厂商提供了一些自己的解决方案。

- JVMTI : agent 机制

  官方规范中提供了 agent 机制，用户可以通过 `retransformClass` / `redefineClass` 操作可以在加载前和加载后动态修改类的内容，但适用范围非常受限。

- DCEVM : JVM enhancement

DCEVM，即 Dynamic Code Evolution VM，意为代码可以动态“进化”，即我们想要的热部署。Java 社区希望能做一个增强版的 JVM，使其原生支持代码动态更新和部署。但这个项目进展比较缓慢，Full version 只支持到 Java 7 ，Light version 支持到 Java 8，而最新的 Java 版本是 Java 10。

图一是代码修改的层次图：

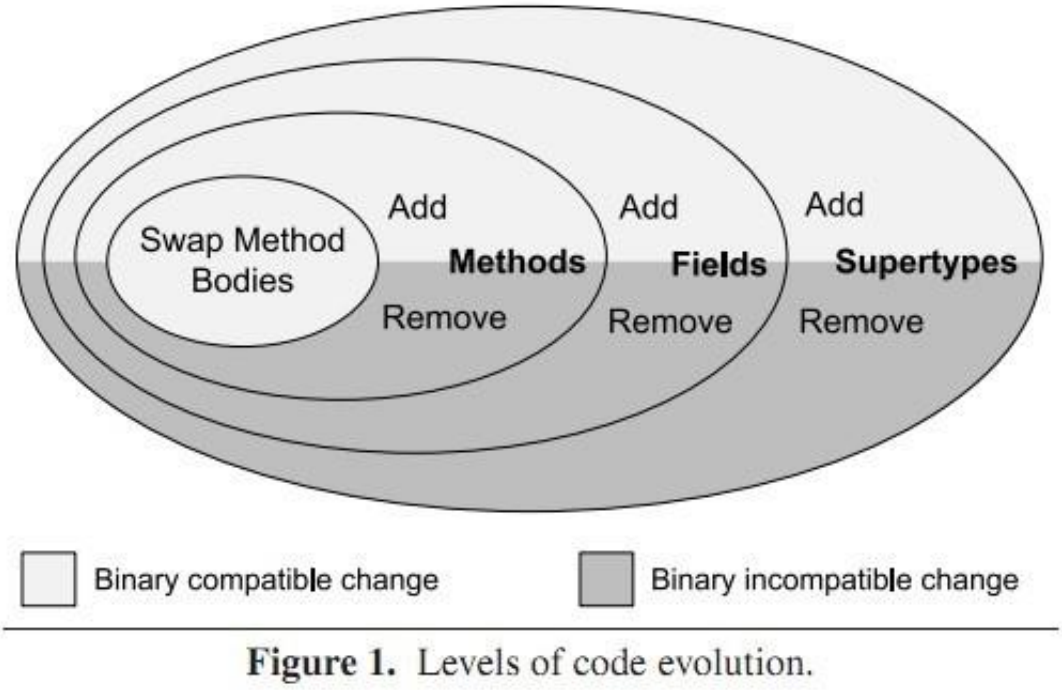


Figure 1. Levels of code evolution.

图中白色部分表示相容的修改，灰色部分表示不相容的修改。交换函数体始终是相容的，而对于方法、字段和超类的而言，增加都是相容的，删除都是不相容的。但值得注意的是，相容并不能保证代码正确、安全地运行，涉及到资源、递归、锁、并发等时，还是应该小心谨慎。

图二是 DECVM 的实现架构：

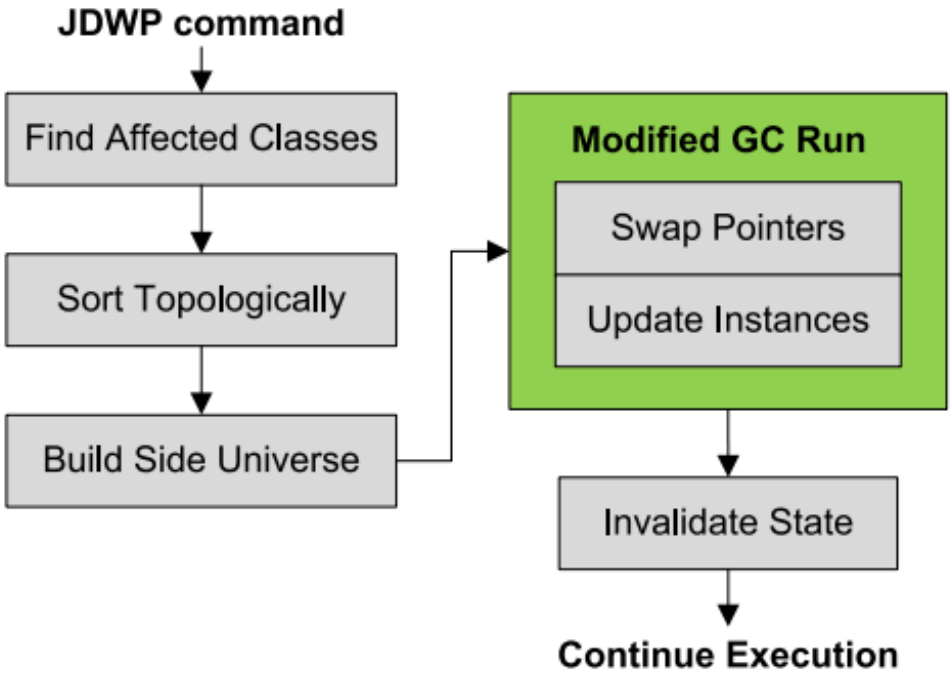
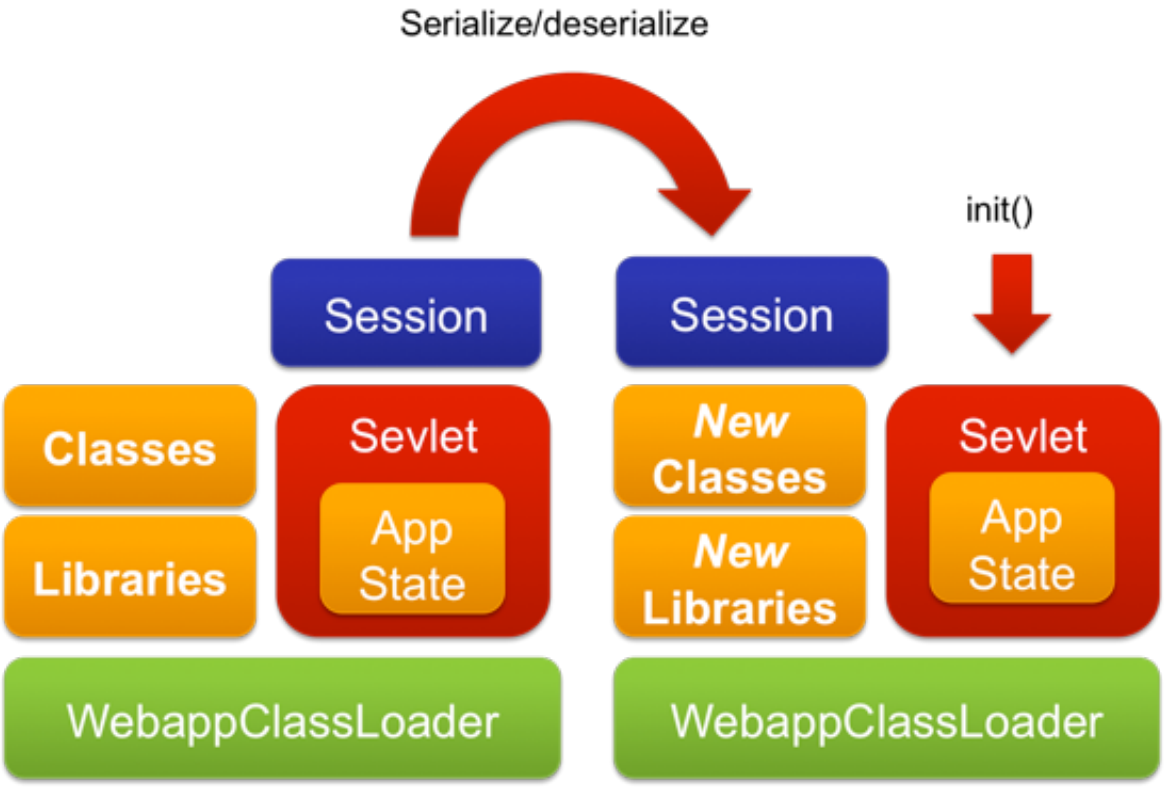


Figure 2. Steps performed by the code evolution algorithm.

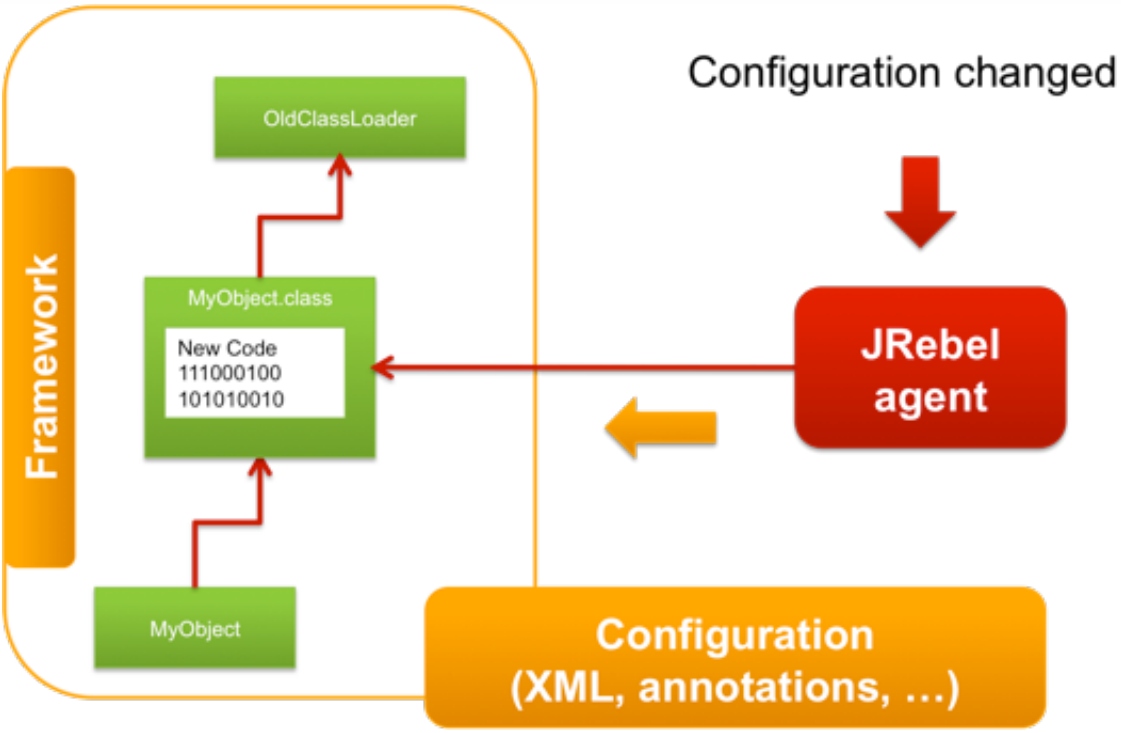
DECVM 的实现比较复杂，代码热部署时涉及符号查找，代码替换，实例更新，垃圾回收等方面。JDWP 的命令触发后，首先查找受影响的类，然后对他们的继承关系进行拓扑排序，接着构建一个新的环境。接下来就进入 GC （垃圾回收）系统，修改指针，更新实例，然后废弃掉有害的状态。最后把控制流交回程序，继续执行。由此可见，DEVCM 上 Java 的一次热部署需要的开销并不小。

- JBoss : 自己开发 ClassLoader



JBoss 是一个开源的应用服务器，它自定义了类加载过程 (WebappClassLoader) ，使其支持热部署功能。但这带来了一些的副作用：PermGen out of memory 异常。因为 JVM 的限制，热部署其实是在不断地装载新的类，所以给存储造成了较大的负担，某些特定功能的存储分区发生了溢出，引发了异常。

- JRebel：JVM 插件



> JRebel 是一个 JVM 插件，在检测到 Configuration（配置）改变时，JRebel agent 会作用于 .class 文件。由上图可以看出，实例 MyObject 引用了 MyObject.class ，而 MyObject.class 引用 OldClassLoader。如果需要更改 class，则需要关注它和 ClassLoader，Object 的联系。
>
> JRebel 与 JBoss 的做法接近，都是添加了一个动态链接层，这种链接层对虚拟机执行性能的影响非常大。

Java 热部署中的难点：

> 1. 两个全限定名相同的类如何加载?
> 2. 类的实例化如何获取到新的类?
> 3. 更新类的声明后，如增加类方法、实例方法；修改类方法、实例方法签名、方法体、方法注解；新增、修改类变量、实例变量；修改接口、类的继承关系，调用点怎么指向新的类?
> 4. 如果JVM使用了内联优化技术呢?
> 5. 如何保证反射正确，比如调用 `Class` 的 `getName()`、`getMethods()`、`getField()` 等方法时如何获取到新的类?
> 6. 如果用了容器或者框架，修改JavaConfig或者XML后，怎么反映到容器里?

Lua 等动态语言的热部署

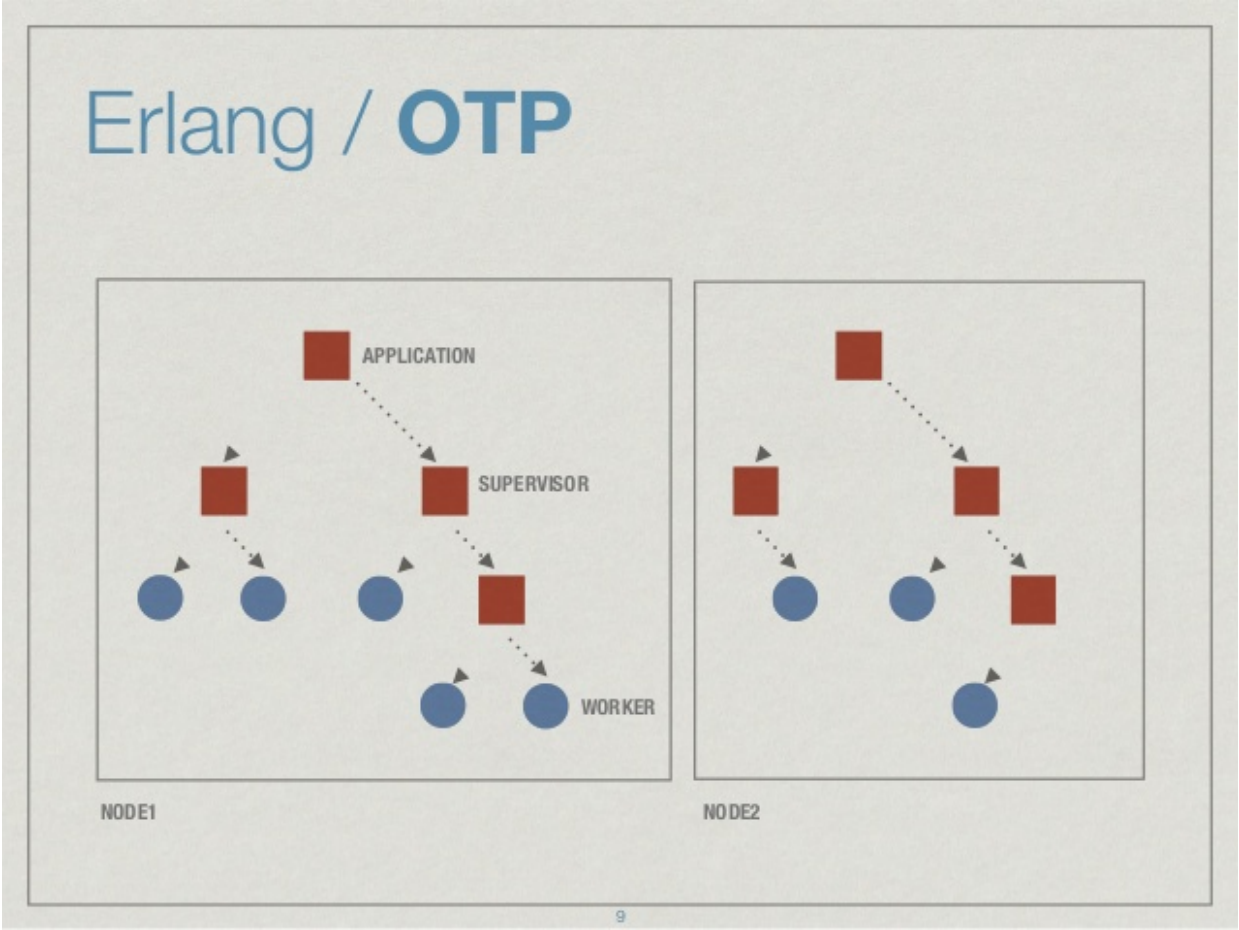
动态语言如 Lua, Python, PHP, Closure 等都有自己的热部署方案。因为对于动态语言来说，代码就是数据，什么时候想改代码了，直接重新加载即可，因为虚拟机的代码是没有发生改变的，只不过是虚拟机所持有的数据发生了变化，导致它的控制流发生改变。对操作系统来说，这个更新是不可见的。

Python，PHP 等的代码热部署常用于服务器的热部署，而 Lua 因为其常常作为游戏的脚本语言而经常用于游戏的热部署。Closure 是 JVM 上的动态语言，因此有时也作为 JVM 程序热部署的替代方案。

动态语言执行代码的方式一般有两种，一种是直接执行，一种是通过 JIT 技术编译后执行。总的来说，动态语言数据格式灵活，运行期反射强大。

Erlang 与 Actor 并发模型

Erlang 是爱立信公司为了电话系统的热部署而专门开发的编程语言，采用 Actor 并发模型，在虚拟机中实现轻量级进程。



一个 Application (应用) 下面有一些 Supervisor (监控进程) 和一些 Worker (工作进程)，这里的进程并不是操作系统意义下的进程，实际上它比线程还轻。它自己有一些显式声明的状态和一个信箱，信箱用于接收来着其它进程的消息，然后对消息进行处理。在没有消息的时候，进程处于阻塞状态，而不是空转状态，这为系统节省了开销。Actor 的启动非常迅速，而且因为是基于消息的，所以一个 Actor 可以迅速的关闭和重启，并继续处理刚才信箱中的消息，而这一切在外界调用者看来就像没有发生一样。再加上每个 Worker 都有 Supervisor，因此 Worker 在重新启动后很容易继续刚才的工作。对于整个系统而言，如果需要热部署，只需分批换下各个 Actor 即可。这是开销很小而且安全的热部署方案。

Scala 语言的库 Akka Actor 同样提供了 Actor 进程模型，Akka Actor 的文档中提到 Akka Actor 也支持安全的热部署。

安卓平台上的热修复

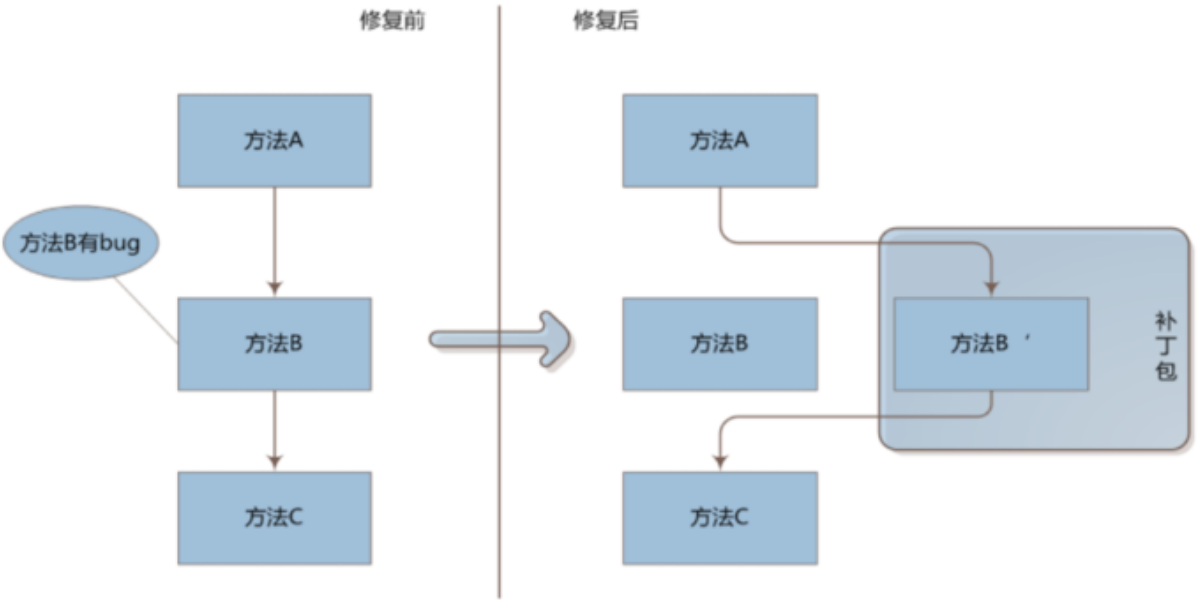
安卓平台上有热修复技术，指的是对已经发布的 APP 进行热修复，与我们探讨的热部署略有差别。热修复主要解决的问题是如何给已发布的 APP 打补丁，而不是重新发布一个新的版本。不管是安卓还是 iOS ，发布一个 APP 版本都是一件成本不低的事情，然而，生产中迭代开发的要求使得一个 APP 不可能经过很长时间的测试才发布。就算经历了充分的测试，也难免会遇到各种各样的小 bug 。

图 1-1 和 图 1-2 展示了正常开发流程和热修复开发流程的差别，可以看出，热修复可以节约开发成本，改善用户体验。

图 1-1　正常开发流程

图 1-2　热修复开发流程

热修复的实现原理



主被动分区技术

这是与系统更新相关的技术，用两个分区来安装系统，一个叫主动分区，是实际加载的分区，一个叫被动分区，用于等待更新。一旦更新发布，被动分区就应用更新，并在下一次启动时与之前的主动分区交换位置。这一技术与热部署并没有直接的关系，但其思路值得借鉴。

启发

热部署在实现上的重点在于动态加载代码和原有控制流的改变，对于动态语言来说，因为控制流完全由虚拟机控制，所以可以在虚拟机层面提供热部署机制。而对于 native 程序，加载和控制流有一部分是由操作系统控制，一部分是硬件控制的，因此我们可以借鉴 Lua 等动态语言的思路，对已运行的程序进行热部署。

热部署在理论上的难点在于状态的管理，即前后程序的无缝衔接，已有的工作如何转交，如何保证热部署的安全。对于无状态的程序，如动态链接库中的库函数，如果不更改函数类型，那么更新就是较为安全的。但对于状态丰富的程序而已，如何进行安全地热更新仍然是一个需要解决的问题。Actor 模型提供了一种可行的解决方案，但有一些学习成本和已有代码的迁移成本。

五、参考文献

1. Chopra, Shelesh, et al. "Automated hotfix handling model." U.S. Patent No. 8,713,554. 29 Apr. 2014.
2. Snow, Kevin Z., et al. "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization." Security and Privacy (SP), 2013 IEEE Symposium on. IEEE, 2013.
3. Krall, Andreas. "Efficient JavaVM just-in-time compilation." Parallel Architectures and Compilation Techniques, 1998. Proceedings. 1998 International Conference on. IEEE, 1998.
4. Gal, Andreas, et al. "Trace-based just-in-time type specialization for dynamic languages." ACM Sigplan Notices 44.6 (2009): 465-478.
5. Cramer, Timothy, et al. "Compiling Java just in time." IEEE Micro 17.3 (1997): 36-43.
6. Suganuma, Toshio, et al. "A dynamic optimization framework for a Java just-in-time compiler." ACM SIGPLAN Notices. Vol. 36. No. 11. ACM, 2001.
7. RednaxelaFX. Java是否可以做到修改类而不用重启JVM？[EB/OL]. https://www.zhihu.com/question/28833796.
8. 丁, 志君. 深入探索 Java 热部署[EB/OL]. https://www.ibm.com/developerworks/cn/java/j-lo-hotdeploy/.
9. Jevgeni, Kabanov. Reloading Java Classes 101: Objects, Classes and ClassLoaders[EB/OL]. https://zeroturnaround.com/rebellabs/reloading-objects-classes-classloaders/.
10. Renato, Athaydes. 4 free ways to hot-swap code on the JVM[EB/OL]. https://sites.google.com/a/athaydes.com/renato-athaydes/posts/4freewaystohot-swapcodeonthejvm.
11. Eddie. What makes hot deployment a “hard problem”?[EB/OL]. What makes hot deployment a “hard problem”?.
12. 杨广翔. ELF格式可执行程序的代码嵌入技术[J]. 程序员, 2008, (3): 104-106
13. 桂阳. CoreOS 实战：CoreOS 及管理工具介绍[EB/OL]. http://www.infoq.com/cn/articles/what-is-coreos.
14. Learn You some Erlang[EB/OL]. http://learnyousomeerlang.com/content.
15. Actors (Java)[EB/OL]. https://doc.akka.io/docs/akka/2.0/java/untyped-actors.html.
16. luciandun. Android热更新之初探[EB/OL]. https://www.jianshu.com/p/a4bf979cce3b.
17. 海纳. JVM杂谈之JIT[EB/OL]. https://zhuanlan.zhihu.com/p/28476709.
18. Jiadong. 几种常见的JVM热部署技术及实现难点浅谈[EB/OL]. https://segmentfault.com/a/1190000011174467.
19. skybber. DCEVM[EB/OL]. https://dcevm.github.io/.
20. redcreen. Dynamic Code Evolution for Java dcevm 原理[EB/OL]. http://www.cnblogs.com/redcreen/archive/2011/06/14/2080718.html.
21. winner_0715. HotSwap和JRebel原理[EB/OL]. https://www.cnblogs.com/winner-0715/p/5136489.html.
22. argan. 实现增强的java class hotswap (三) 解决方案[EB/OL]. http://argan.iteye.com/blog/443611.
23. argan. 实现增强的java class hotswap (三) 解决方案 续[EB/OL]. http://argan.iteye.com/blog/444633.