> 前言
> 关于什么是 JVMTI？怎么使用 JVMTI 都不在本主题的讨论范围之内，大家可自行百度或者谷歌。本主题主要通过使用 JVMTI 技术实现对本地方法进行增强的功能，该主题分两部分

- 第一部分也就是本章节，给出一个简单的实现；
- 第二部分会结合 JVM 底层源码和大家分享，底层是如何实现这一功能的。

目标描述

利用 JVMTI 技术实现对 Java native 方法增强， 话不多少，咱直接从代码入手或许会看的更清楚，请看如下 Java 类

```
public class NormalClazz{
    private static void normalMethod(){
        long start = System.currentTimeMillis();
        for(int i = 0 ; i < Integer.MAX_VALUE; ++i){
            // do something calculation
        }
        long cost = System.currentTimeMillis() - start;
        System.out.println("normalMethod cost: " + cost);
    }
}
```

上面这种代码，在很多时候会出现在我们的调试代码中，即用于调试某个具体的方法（如：normalMethod）的执行耗时，但是这有个很明显的缺陷，那就是需要硬编码这些测试代码，耗时费力，且是重复劳动，而且当项目正式发布时还需要去掉这些代码，于是就产生了使用字节码增强技术，在方法的开头和结尾处插入相关的逻辑，达到用于计算方法调用耗时，监控方法入参、返回值等的目的，同过字节码增强，可以产生类似如下代码：

```
public class NormalClazz2{
    private static void normalMethod(){
        // 自动插入
        long start = System.currentTimeMillis();
        for(int i = 0 ; i < Integer.MAX_VALUE; ++i){
            // do something calculation
        }
        // 自动插入
        long cost = System.currentTimeMillis() - start;
        // 自动插入
        System.out.println("normalMethod cost: " + cost);
    }
}
```

上面打有"自动插入"注释的代码，则可以认为是字节码增强技术在方法运行时自动插入的代码，省去了很多无畏的硬编码，大大节省了开发和调试的时间。可谓非常强大，非常方便。
那么问题来了，如果有一个本地方法，这个时候你想监控这个本地方法的执行时间，我们该怎么做？ 如下：

```
public class NormalClazz3{
    private static native void nativeMethod();
}
```

nativeMethod 是一个本地方法，我们知道 java 中本地方法是不存在 java 代码的，其具体实现被移动到了 C/C++ 代码中。这个时候，我们该如何获取这个方法的执行时间呢？大家可以先停留 2 分钟，自己思考一下，看看有没有什么思路。

解决方案

有些同学可能找到了解决方案或思路，有些可能没有。没关系，这就是本节要和大家分享的主题。既然本地方法没有代码，那咱们能不能把这个本地方法先转成 java 方法普通方法，然后在这个普通方法中调用原来的本地方法，这样一来

- 原来本地方法的调用者不需要做任何的修改
- 方便在 java 普通方法中插入代码来获取本地方法执行的时长了

这段话要表述的意思，用代码描述如下：

```
public class NormalClazz3{
    // 原本的 native 方法，转换为普通的 java 方法
    private static void nativeMethod(){
        // 在这里调用原来的 native 方法
        $$wrapped$$_nativeMethod();
    }

    private static native void $$wrapped$$_nativeMethod();
}
```

看到这里，是不是有思路了？只需要采用字节码增强技术对 `nativeMethod` 方法采用前面 `NormalClazz2` 中的 `normalMethod` 一样的操作即可获取 `$$wrapped$$_nativeMethod()` 的执行时间了？是不是很 nice？ 别高兴的太早，事情往往没有想象的那么简单，细心的同学肯定发现了，原来的 `nativeMethod` 变成了 `$$wrapped$$_nativeMethod()` 那底层是如何通过变名字后的方法调用到原来的本地方法的呢？
幸运的是，JVMTI 底层提供了通过设置本地方法前缀名来改变 JVM 对本地方法的寻址方式，对应的 JVMTI 接口就是 `SetNativeMethodPrefix`，针对本例，只需要通过该接口设置本地方法的前缀为 `$$wrapped$$_` 就能帮助 JVM 找到 `$$wrapped$$_nativeMethod()` 对应的底层实现。
针对上面的描述做一个简短的总结

- 添加一个和原有本地方法同名的普通方法
- 将原有的本地方法通过 `$$wrapped$$_` 前缀进行包装，改名为 `$$wrapped$$_nativeMethod()`
- 通过 JVMTI 接口设置前缀 `$$wrapped$$_` 来帮助 JVM 直找到原来底层和之前 `nativeMethod` 对应的本地方法实现

代码实现

Java 代码

```
package com.bytecode.newi;
public class NormalClazz3{
    // 原本的 native 方法，转换为普通的 java 方法，这部分可以通过 ASM、Javassist 等字节码框架在运行时修改方法的修饰符，去掉native
    private static void nativeMethod(){
        // 在这里调用原来的 native 方法
        $$wrapped$$_nativeMethod();
    }

    // 这 $$wrapped$$_nativeMethod 方法可以通过 ASM、Javassist 等字节码框架在运行时新建
    private static native void $$wrapped$$_nativeMethod();
}
```

C++ 代码

HPP 头文件

com_bytecode_newi_NormalClazz3.hpp

```
#ifndef _Included_com_bytecode_newi_NormalClazz3
#define _Included_com_bytecode_newi_NormalClazz3
#ifdef __cplusplus
extern "C" {
#endif
JNIEXPORT jlong JNICALL Java_com_bytecode_newi_NormalClazz3_nativeMethod
  (JNIEnv *, jclass);
#ifdef __cplusplus
}
#endif
#endif
```

CPP 实现

com_bytecode_newi_NormalClazz3.cpp

```
#include <stdio.h>
#include <jvmti.h>
#include "com_bytecode_newi_NormalClazz3.hpp"
const char* WRAPPER_PREFIX = "$$wrapper$$_";
jvmtiEnv *jvmti = NULL;

JNIEXPORT void JNICALL Java_com_bytecode_newi_NormalClazz3_nativeMethod
  (JNIEnv * jnienv, jclass clazz){
    printf("NormalClazz3_nativeMethod \n");
}

extern "C"
JNIEXPORT jint JNICALL
Agent_OnAttach(JavaVM* vm, char *options, void *reserved) {
  return Agent_OnLoad(vm,options,reserved);
}

extern "C"
JNIEXPORT jint JNICALL
Agent_OnLoad(JavaVM* vm, char *options, void *reserved) {
  jint ret = vm->GetEnv((void**)&jvmti, JVMTI_VERSION);
  if(ret != JNI_OK) {
    printf("Unable to access JVMTI, error: %d \n", ret);
    return -1;
  }
  jvmtiCapabilities caps;
  caps.can_set_native_method_prefix = 1;
  jvmtiError error = jvmti->AddCapabilities(&caps);
  if(error != JVMTI_ERROR_NONE) {
    printf("Unable to add capabilities: can_set_native_method_prefix. \n");
    return -1;
  }

  // 设置前缀，帮助 JVM 通过前缀再去找到原来的本地方法
  error = jvmti->SetNativeMethodPrefix(WRAPPER_PREFIX);
  if(error != JVMTI_ERROR_NONE) {
    printf("Unable to SetNativeMethodPrefix: %s\n", WRAPPER_PREFIX);
    return -1;
  }

  return 0;
}
```

动态库编译的指令

```
g++ -I${JAVA_HOME}/include -I${JAVA_HOME}/include/darwin \
        com_bytecode_newi_NormalClazz3.cpp -fPIC -shared -ldl -lpthread -o libnew.dylib
```

在运行的加上 java `-agentpath:/????/libnew.dylib` com.bytecode.newi.NewInstruction

总结

本文通过一个简单的案例演示了如何通过使用 JVMTI 提供的能力来跟踪和增强本地方法。当然，其中有很大一部分的实现没在本文的讨论范围，比如如何通过 `ASM`、`Javassist` 等字节码框架来实现运行时方法的修改、增强以及方法的添加等操作，这些东西大家感兴趣的话，欢迎留言，和大家一起探讨，也可以自行百度、Google。

> 由于时间仓促，本文中难免存在一些错误，还望大家不吝指出，同时如果对文中描述存在任何疑问，也欢迎大家提出来讨论。

参考

JVMTI 参考原文
JVMTI 参考中文