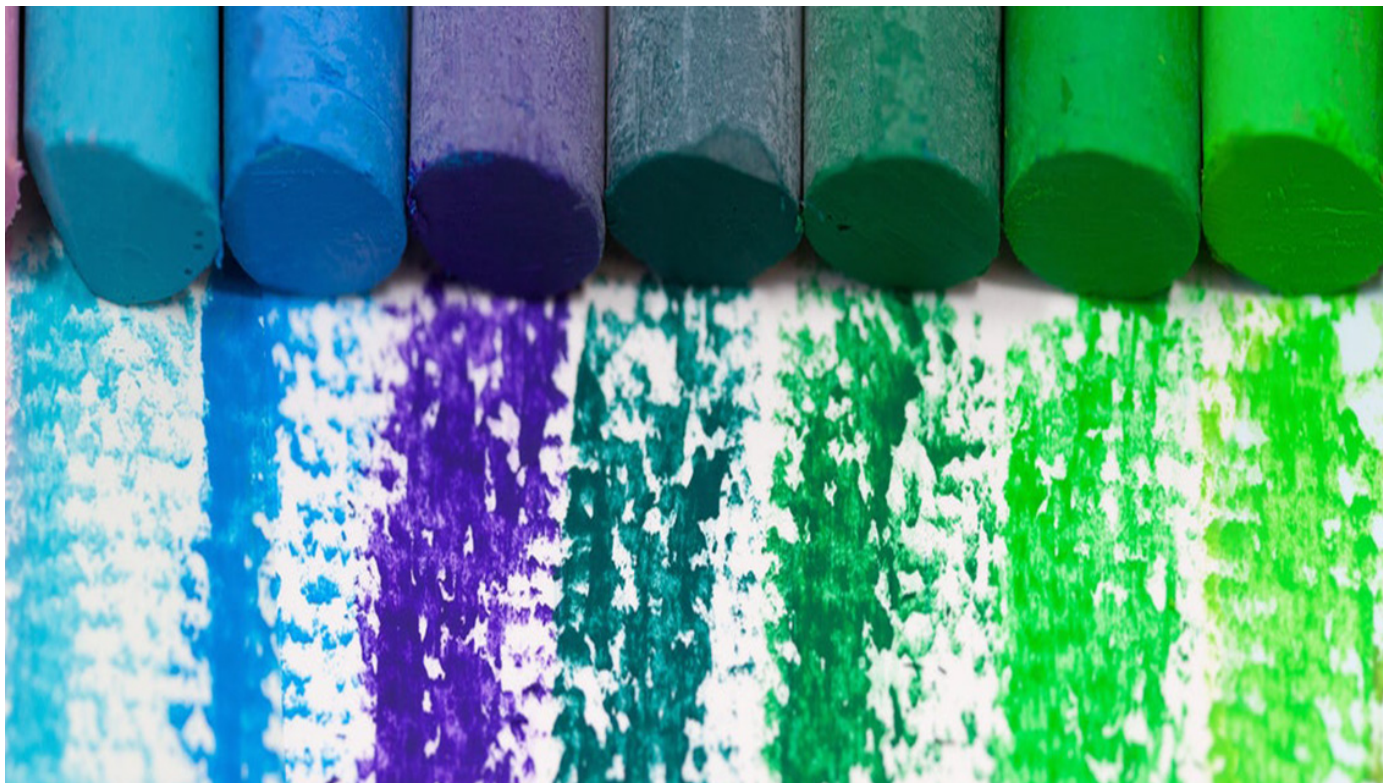


讲堂 > 深入拆解 Java 虚拟机 > 文章详情

05 | JVM是如何执行方法调用的？（下）

2018-07-30 郑雨迪



05 | JVM是如何执行方...

朗读者：郑雨迪 13'01" |
5.97M

0:00 / 0:00

我在读博士的时候，最怕的事情就是被问有没有新的 Idea。有一次我被老板问急了，就随口说了一个。

这个 Idea 究竟是什么呢，我们知道，设计模式大量使用了虚方法来实现多态。但是虚方法的性能效率并不高，所以我就说，是否能够在此基础上写篇文章，评估每一种设计模式因为虚方法调用而造成的性能开销，并且在文章中强烈谴责一下？

当时呢，我老板教的是一门高级程序设计的课，其中有好几节课刚好在讲设计模式的各种好处。所以，我说完这个 Idea，就看到老板的神色略有不悦了，脸上写满了“小郑啊，你这是舍本逐末啊”，于是，我就连忙挽尊，说我是开玩笑的。

在这里呢，我犯的错误其实有两个。第一，我不应该因为虚方法的性能效率，而放弃良好的设计。第二，通常来说，Java 虚拟机中虚方法调用的性能开销并不大，有些时候甚至可以完全消除。第一个错误是原则上的，这里就不展开了。至于第二个错误，我们今天便来聊一聊 Java 虚拟机中虚方法调用的具体实现。

首先，我们来看一个模拟出国边检的小例子。

```
1 abstract class 乘客 {
2     abstract void 出境 ();
3     @Override
4     public String toString() { ... }
5 }
6 class 外国人 extends 乘客 {
7     @Override
8     void 出境 () { /* 进外国人通道 */ }
9 }
10 class 中国人 extends 乘客 {
11     @Override
12     void 出境 () { /* 进中国人通道 */ }
```

[复制代码](#)

```
13 void 买买买 () { /* 避免税店 */ }
14 }
15
16 乘客 某乘客 = ...
17 某乘客. 出境 ();
18
```

这里我定义了一个抽象类，叫做“乘客”，这个类中有一个名为“出境”的抽象方法，以及重写自 Object 类的 toString 方法。

然后，我将“乘客”粗暴地分为两种：“中国人”以及“外国人”。这两个类分别实现了“出境”这个方法，具体来说，就是中国人走中国人通道，外国人走外国人通道。由于咱们储蓄较多，所以我在“中国人”这个类中，还特意添加了一个叫做“买买买”的方法。

那么在实际运行过程中，Java 虚拟机是如何高效地确定每个“乘客”实例应该去哪条通道的呢？我们一起来看一下。

1. 虚方法调用

在上一篇中我曾经提到，Java 里所有非私有实例方法调用都会被编译成 invokevirtual 指令，而接口方法调用都会被编译成 invokeinterface 指令。这两种指令，均属于 Java 虚拟机中的虚方法调用。

在绝大多数情况下，Java 虚拟机需要根据调用者的动态类型，来确定虚方法调用的目标方法。这个过程我们称之为动态绑定。那么，相对于静态绑定的非虚方法调用来说，虚方法调用更加耗时。

在 Java 虚拟机中，静态绑定包括用于调用静态方法的 invokestatic 指令，和用于调用构造器、私有实例方法以及超类非私有实例方法的 invokespecial 指令。如果虚方法调用指向一个标记为 final 的方法，那么 Java 虚拟机也可以静态绑定该虚方法调用的目标方法。

Java 虚拟机中采取了一种用空间换取时间的策略来实现动态绑定。它为每个类生成一张方法表，用以快速定位目标方法。那么方法表具体是怎样实现的呢？

2. 方法表

在介绍那篇类加载机制的链接部分中，我曾提到类加载的准备阶段，它除了为静态字段分配内存之外，还会构造与该类相关联的方法表。

这个数据结构，便是 Java 虚拟机实现动态绑定的关键所在。下面我将以 `invokevirtual` 所使用的虚方法表 (virtual method table, vtable) 为例介绍方法表的用法。`invokeinterface` 所使用的接口方法表 (interface method table, itable) 稍微复杂些，但是原理其实是类似的。

方法表本质上是一个数组，每个数组元素指向一个当前类及其祖先类中非私有的实例方法。

这些方法可能是具体的、可执行的方法，也可能是没有相应字节码的抽象方法。方法表满足两个特质：其一，子类方法表中包含父类方法表中的所有方法；其二，子类方法在方法表中的索引值，与它所重写的父类方法的索引值相同。

我们知道，方法调用指令中的符号引用会在执行之前解析成实际引用。对于静态绑定的方法调用而言，实际引用将指向具体的目标方法。对于动态绑定的方法调用而言，实际引用则是方法表的索引值（实际上并不仅是索引值）。

在执行过程中，Java 虚拟机将获取调用者的实际类型，并在该实际类型的虚方法表中，根据索引值获得目标方法。这个过程便是动态绑定。

乘客的方法表

0	乘客.toString()
1	乘客.出境() (备注: 抽象方法, 不可执行)

外国人的方法表

0	乘客.toString()
1	外国人.出境()

中国人的方法表


0	乘客.toString()
1	中国人.出境()
2	中国人.买买买()

在我们的例子中，“乘客”类的方法表包括两个方法：“toString”以及“出境”，分别对应 0 号和 1 号。

之所以方法表调换了“toString”方法和“出境”方法的位置，是因为“toString”方法的索引值需要与 Object 类中同名方法的索引值一致。为了保持简洁，这里我就不考虑 Object 类中的其他方法。

“外国人”的方法表同样有两行。其中，0 号方法指向继承而来的“乘客”类的“toString”方法。1 号方法则指向自己重写的“出境”方法。

“中国人”的方法表则包括三个方法，除了继承而来的“乘客”类的“toString”方法，自己重写的“出境”方法之外，还包括独有的“买买买”方法。

 复制代码

```
1 乘客 某乘客 = ...  
2 某乘客. 出境 ();
```

这里，Java 虚拟机的工作可以想象为导航员。每当来了一个乘客需要出境，导航员会先问是中国人还是外国人（获取动态类型），然后翻出中国人 / 外国人对应的小册子（获取动态类型的方法表），小册子的第 1 页便写着应该到哪条通道办理出境手续（用 1 作为索引来查找方法表所对应的目标方法）。

实际上，使用了方法表的动态绑定与静态绑定相比，仅仅多出几个内存解引用操作：访问栈上的调用者，读取调用者的动态类型，读取该类型的方法表，读取方法表中某个索引值所对应的目标方法。相对于创建并初始化 Java 栈帧来说，这几个内存解引用操作的开销简直可以忽略不计。

那么我们是否可以认为虚方法调用对性能没有太大影响呢？

其实是不能的，上述优化的效果看上去十分美好，但实际上仅存在于解释执行中，或者即时编译代码的最坏情况中。这是因为即时编译还拥有另外两种性能更好的优化手段：内联缓存（inlining cache）和方法内联（method inlining）。下面我便来介绍第一种内联缓存。

3. 内联缓存

内联缓存是一种加快动态绑定的优化技术。它能够缓存虚方法调用中调用者的动态类型，以及该类型所对应的目标方法。在之后的执行过程中，如果碰到已缓存的类型，内联缓存便会直接调用该类型所对应的目标方法。如果没有碰到已缓存的类型，内联缓存则会退化至使用基于方法表的动态绑定。

在我们的例子中，这相当于导航员记住了上一个出境乘客的国籍和对应的通道，例如中国人，走了左边通道出境。那么下一个乘客想要出境的时候，导航员会先问是不是中国人，是的话就走左边通道。如果不是的话，只好拿出外国人的小册子，翻到第 1 页，再告知查询结果：右边。

在针对多态的优化手段中，我们通常会提及以下三个术语。

1. 单态 (monomorphic) 指的是仅有一种状态的情况。
2. 多态 (polymorphic) 指的是有限数量种状态的情况。二态 (bimorphic) 是多态的其中一种。
3. 超多态 (megamorphic) 指的是更多种状态的情况。通常我们用一个具体数值来区分多态和超多态。在这个数值之下，我们称之为多态。否则，我们称之为超多态。

对于内联缓存来说，我们也有对应的单态内联缓存、多态内联缓存和超多态内联缓存。单态内联缓存，顾名思义，便是只缓存了一种动态类型以及它所对应的目标方法。它的实现非常简单：比较所缓存的动态类型，如果命中，则直接调用对应的目标方法。

多态内联缓存则缓存了多个动态类型及其目标方法。它需要逐个将所缓存的动态类型与当前动态类型进行比较，如果命中，则调用对应的目标方法。

一般来说，我们会将更加热门的动态类型放在前面。在实践中，大部分的虚方法调用均是单态的，也就是只有一种动态类型。为了节省内存空间，Java 虚拟机只采用单态内联缓存。

前面提到，当内联缓存没有命中的情况下，Java 虚拟机需要重新使用方法表进行动态绑定。对于内联缓存中的内容，我们有两种选择。一是替换单态内联缓存中的纪录。这种做法就好比 CPU 中的数据缓存，它对数据的局部性有要求，即在替换内联缓存之后的一段时间内，方法调用的调用者的动态类型应当保持一致，从而能够有效地利用内联缓存。

因此，在最坏情况下，我们用两种不同类型的调用者，轮流执行该方法调用，那么每次进行方法调用都将替换内联缓存。也就是说，只有写缓存的额外开销，而没有用缓存的性能提升。

另外一种选择则是劣化为超多态状态。这也是 Java 虚拟机的具体实现方式。处于这种状态下的内联缓存，实际上放弃了优化的机会。它将直接访问方法表，来动态绑定目标方法。与替换内联缓存

纪录的做法相比，它牺牲了优化的机会，但是节省了写缓存的额外开销。

具体到我们的例子，如果来了一队乘客，其中外国人和中国人依次隔开，那么在重复使用的单态内联缓存中，导航员需要反复记住上个出境的乘客，而且记住的信息在处理下一乘客时又会被替换掉。因此，倒不如一直不记，以此来节省脑细胞。

虽然内联缓存附带内联二字，但是它并没有内联目标方法。这里需要明确的是，任何方法调用除非被内联，否则都会有固定开销。这些开销来源于保存程序在该方法中的执行位置，以及新建、压入和弹出新方法所使用的栈帧。

对于极其简单的方法而言，比如说 getter/setter，这部分固定开销占据的 CPU 时间甚至超过了方法本身。此外，在即时编译中，方法内联不仅仅能够消除方法调用的固定开销，而且还增加了进一步优化可能性，我们会在专栏的第二部分详细介绍方法内联的内容。

总结与实践

今天我介绍了虚方法调用在 Java 虚拟机中的实现方式。

虚方法调用包括 `invokevirtual` 指令和 `invokeinterface` 指令。如果这两种指令所声明的目标方法被标记为 `final`，那么 Java 虚拟机会采用静态绑定。否则，Java 虚拟机将采用动态绑定，在运行过程中根据调用者的动态类型，来决定具体的目标方法。

Java 虚拟机的动态绑定是通过方法表这一数据结构来实现的。方法表中每一个重写方法的索引值，与父类方法表中被重写的方法的索引值一致。在解析虚方法调用时，Java 虚拟机会纪录下所声明的目标方法的索引值，并且在运行过程中根据这个索引值查找具体的目标方法。

Java 虚拟机中的即时编译器会使用内联缓存来加速动态绑定。Java 虚拟机所采用的单态内联缓存将纪录调用者的动态类型，以及它所对应的目标方法。

当碰到新的调用者时，如果其动态类型与缓存中的类型匹配，则直接调用缓存的目标方法。否则，Java 虚拟机将该内联缓存劣化为超多态内联缓存，在今后的执行过程中直接使用方法表进行动态绑定。

在今天的实践环节，我们来观测一下单态内联缓存和超多态内联缓存的性能差距。为了消除方法内联的影响，请使用如下的命令。

```
1 // Run with: java -XX:CompileCommand='dontinline,*. 出境' 乘客
2 public abstract class 乘客 {
3     abstract void 出境 ();
4     public static void main(String[] args) {
5         乘客 a = new 中国人 ();
6         乘客 b = new 外国人 ();
7         long current = System.currentTimeMillis();
8         for (int i = 1; i <= 2_000_000_000; i++) {
9             if (i % 100_000_000 == 0) {
10                 long temp = System.currentTimeMillis();
11                 System.out.println(temp - current);
12                 current = temp;
13             }
14             乘客 c = (i < 1_000_000_000) ? a : b;
15             c. 出境 ();
16         }
17     }
18 }
19 class 中国人 extends 乘客 { @Override void 出境 () {} }
20 class 外国人 extends 乘客 { @Override void 出境 () {} }
```

[复制代码](#)

 极客时间

深入拆解Java虚拟机

Oracle 高级研究员 手把手带你入门 JVM



郑雨迪 Oracle Labs高级研究员，计算机博士

新版升级：点击「 请朋友读」，10位好友免费读，邀请订阅更有**现金**奖励。

版权归极客邦科技所有，未经许可不得转载

写留言

精选留言



啊一大狗

这套课很好，谢谢！

2018-07-31

👍 3



Tony

同提建议，代码使用英文。刚学java基础时，有老师为了便于理解用中文命名。现在都来学jvm，对java很熟悉了，看到中文不仅不会觉得通俗易懂，反而特别别扭。

👍 27

2018-07-30

| 作者回复

多谢建议!

原本是英文的，录音的时候觉得老要切换，就给换了。。

2018-07-30



Jchriss

👍 10

提个小建议，能否在代码中都使用英文？毕竟使用中文作对象名不值得提倡

2018-07-30

| 作者回复

谢谢建议!

2018-07-30



陈晨

👍 6

没用过中文写代码，居然认为中文会编译错误T____T

老师是为了课件方便这样写，自己写作业就改下呗，又没规定要每个字照抄

```
[root@localhost cq]# javac Passenger.java
```

```
[root@localhost cq]# java Passenger
```

```
cost time : 1167
```

```
cost time : 3156
```

```
[root@localhost cq]# java -XX:CompileCommand='dontinline,*.exit' Passenger
```

```
CompilerOracle: dontinline *.exit
```

```
cost time : 3709
```

```
cost time : 7557
```

2018-07-30

| 作者回复

哈，我以前也认为无法编译，直到有一次我看到一个俄语的方法名。。

另外，如果你用javap -v查看常量池的话，你会发现类名方法名以及方法描述符都是用UTF8来存的。

2018-07-30



杨军

👍 4

一直不太理解一个问题：“Java的动态类型运行期才可知”，在编译期代码写完之后应该就已经确定了吧，比如A是B的子类，“B b = new B(); b = new A()”这种情况下b的动态类型是A，Java编译器在编译阶段就可以确定啊，为什么说动态类型直到运行期才可知？

诚心求老师解惑，这个问题对我理解Java的动态绑定机制很关键-.-

2018-08-13



lxz

👍 4

建议结合java代码及其对应的字节码来讲解，比如常量池，方法表在字节码中对应的位置，干讲一点印象也没有

2018-08-02



杨春鹏

👍 4

关于单态内联缓存中的记录，hotspot采用了超多态。也就是如果该调用者的动态类型不是缓存中的类型的话，直接通过基于方法表来找到具体的目标方法。那么内联缓存中的类型是永久不变，一直是第一次缓存的那个调用者类型吗？

2018-07-31



godtrue

👍 3

1:虚方法

方法重写的方法，可认为就是虚方法

2:JVM怎么执行虚方法

通过方法表，一个二维表结构，标示出类的类型、虚方法的序号。当调用虚方法的时候，先确定类型，再根据类型找方法

2018-07-30



和风暖林

👍 2

代码用汉语也挺好的呀。来这都是学jvm的，没有来学编码规范的吧.....

2018-07-30

作者回复

哈，多谢支持。不过汉语编程有个问题，没办法区分大小写，因此变量名和类名容易混淆

2018-07-31



左岸 开

👍 2

为什么调用超类非私有实例方法会属于静态绑定呢？

2018-07-30

作者回复

通过super关键字来调用父类方法，本意就是想要调用父类的特定方法，而不是根据具体类型决定目标方法。

2018-07-31



吾是锋子

👍 1

郑老师，您好。有个具体的问题想请教下，String类里面indexOf(String str)调用的是自己类里面indexOf(String str, int fromIndex)方法，但我自己在测试的时候却发现两个方法的速度有很明显的差异，看字节码也没有发现什么特殊。

不知道是不是我忽略了什么，希望您能抽空点拨下，感谢！

2018-08-14

作者回复

HotSpot里有String.indexOf intrinsic，用了很多向量化指令，所以性能会快很多的。

关于intrinsic的概念，你可以理解为HotSpot识别指定方法后，将其替代为语意等价的高效实现。

2018-08-15



方枪枪

👍 1

一直不能明确一个问题，执行哪个方法，是不是都是在运行的时候确定的，如果是的话，coding的时候，写一个不存在的方法or传入不存在的参数，编译会报错，那这个合法性的检测，是一个

什么逻辑？另外关于方法的确定，对于Java来说，是按照传入的形参确定执行哪个重写的方法，对于 groovy 是按照实际类型确定执行哪个方法，这两个区别在JVM层面是如何实现的？

2018-08-01

作者回复

合法性检测是根据编译器能找到的class文件来判定的。你可以在编译后，移除掉相应的class文件或者库文件，就会出现你所说的不存在的方法的情况了。

第二个问题，在各自的编译器中已经作出区分了。在Java字节码中就只是根据类名，方法名和方法描述符来定位方法的。

2018-08-02



有时也，命也，运也，如之奈何？

👍 0

豁然开朗，拆解的够透彻。

2018-10-15



likun

👍 0

你好 复习测试遇到了一个问题 课后题修改了一下当类型又切回已缓存的单态类型时耗时介于两者之间且有明显的跨度 这是为什么呢？望有空可以解答一下 谢谢

2018-10-08



崔不武

👍 0

请问郑老师，单态内联缓存什么时候触发呢？这个缓存的作用范围是单线程还是全虚拟机呢？

2018-09-30



ZY

👍 0

- 单态(monomorphic)内联缓存：（缓存遇到的第一个，永不替换）
- 多态(polymorphic)内联缓存：（HotSpot里不存在，猜想就是缓存前N个，估计也永不替换）
- 超多态状态/超太态内联缓存：（直接方法表）

这是我的想法。。不知道对不对

2018-09-28





ZY

👍 0

前5个为有内联缓存的情况，后5个被劣化为超多态内联缓存（其实就是方法表访问），这个缓存压根没有替换的过程。。这个超多态内联缓存。。是挂羊头卖狗肉吗。。

2018-09-28



ZY

👍 0

不太明白这段代码为啥体现了单态内联缓存和超多态内联缓存的性能差距？。。既然说了Java虚拟机只用了单态内联缓存。。为什么后面的b.出境()没有被替换之前的缓存？

2018-09-28



班纳睿

👍 0

java -XX:CompileCommand='dontinline,*. 出境' 乘客
试验了下发现命令里“出境”前面多个空格，得去掉才行。

2018-09-22



掌心童话

👍 0

告诉了我：打老板脸有多爽😁

2018-08-29



allwmh

👍 0

问下方法的内联缓存 是什么样的结构，它的作用范围是什么（全vm，类或者其他）？

2018-08-24



乔毅

👍 0

感觉还是没理解。虚函数调用开销主要是查表带来的，文中又说查表只是内存解引用成本可以忽略不计。但从课后的例子看有没有查表的缓存，差距有小一倍，所以结论是查表的影响还是挺大的？

2018-08-17



东方

👍 0

清晰易懂，赞

2018-08-11



hacker time

👍 0

单态的用内联缓存，超多态的劣化为方法表+索引，那多态的呢？

2018-08-11

作者回复

这里指的是HotSpot的情况，它不存在多态内联缓存。

2018-08-15



Bale

👍 0

老师，我只有第一次能成功出现，关掉inline 功能会变的慢一些。

如果我不清除缓存的，执行命令和不带命令的时间几乎相等。

但是我清除内存，就会重现结果。

Jvm会缓存 我执行过的java类嘛？

2018-08-06



沉淀的梦想

👍 0

为什么在我电脑上运行实验的结果是这样

java 乘客

打印:

961

1740

java -XX:CompileCommand="dontinline, *.*" 乘客

打印:

3164

3506

禁止方法内联之后，反而差别不大了

2018-08-05



一只智障

👍 0

内联缓存再碰到没有缓存的目标类以及目标方法，那么会去方法表中寻找，那为什么内联表不把
这个没有命中的方法缓存下来供下次使用呢？

2018-08-05



丝竹悠扬

👍 0

挺好的，好想可以听懂点

2018-08-05



蠢蠢欲动的腹肌

👍 0

老师，请问下，在准备阶段生成的方法表存储在哪里？

2018-08-03



Grubby🐶

👍 0

内联缓存是针对每个类的吗？就是说两个不同父类的子类，在jvm里有两个内联缓存？

2018-08-02



熊猫酒仙

👍 0

想问下老师，方法作为参数传递，传递的是方法地址吗？这种情况，执行时是否不存在查找方法的过程？

另外假如两个类的方法互相调用，互相需要对方的方法信息，(排除其他依赖的情况)那么这两个类的加载是并行还是串行呢？

2018-08-02



godtrue

👍 0

3:缓存

凡是需要提高性能的地方都需要使用，这个方法也是人类经常使用的方式，计算机中使用的也比较多，使用缓存的基本理念是，一将需要的东西提前加工好，二将加工好的东西放在获取速度更快更方便的地方

4:内联缓存

是JVM为了提高动态绑定或者根据动态的类类型找目标方法的一种方式，这是以空间换时间的优化思路，需要权衡利弊，视场景使用

2018-08-01



礼貌

👍 0

汉语编程?

2018-07-30

作者回复

哈，这个对于VM实现者来说可是feature，毕竟要存储UTF8。不过以后的代码会换到英文的。

2018-07-30



王旭林

👍 0

老师，打印耗时的System.out.println 用的太多了吧？

2018-07-30

作者回复

你是指课后作业吗？

打印语句每一亿次循环只会运行一次，相对来说并不耗时。

2018-07-30