OpenJDK对于RISC-V的支持现状以及路线图-补充报告

De-Optimization

定义：

- De-Optimization（去优化）是将优化的堆栈帧更改为未优化的帧的过程。对于已编译的方法，则是丢弃无效优化代码，然后用未优化、更健壮的代码替换之前优化的代码的过程。[1]
- 通常会重新编译未优化的方法，以适应不断变化的应用程序行为，一种方法原则上可以去优化数十次。

可能触发的情形

- 需要调试的情况：如果代码正在进行单个步骤的调试，那么之前被编译成为机器码的代码需要去优化，从而能够调试。
- 代码废弃的情况：当一个被编译过的方法，因为种种原因不可用了，这个时候就需要将其去优化。
- 加载了新类后类型继承结构出现变化。
- 出现"罕见陷阱"（Uncommon Trap）

  例：编译器最初假定引用值永远不会为null，然后使用陷阱内存访问对其进行测试。之后应用程序使用了空值，则会对该方法进行去优化和重新编译，以使用显式的"测试和分支"风格来检测此类空值。

OpenJDK官方提出的几种情形[1]

- 编译器可能会将未采用的分支进行预测，而如果它曾经采用的话则会进行去优化
- 从未失败的低级别安全检查
- 如果一个调用点或者转换发生了类型异常，则编译器会进行去优化
- 如果加载的类使早期的类层次结构分析无效，则任何线程中的所有受影响的方法调用都将被强制移至安全点并进行去优化。
- 这种间接的非优化是由依赖系统来实现的。如果编译器做出了未经检查的假设，则它必须记录一个可检查的依赖项。

  例：

  ```
  Foo foo = getFoo();
  foo.doSomething();
  ```

  编译器不一定能从这段代码中分辨出`doSomething()`将执行哪个版本，它可能是在类`Foo`中的实现，或者是在Foo子类中的实现。JVM可以使用全局信息来做出更好的决策。假设没有已加载的扩展类`Foo`在此应用程序中，JVM会认为更像是`doSomething()`是`Foo`中的一个`final`方法，编译器可以将虚拟方法调用转换为直接调度（效率已经有所改进），并且还可以选择inline `doSomething()`。

  而如果编译器进行了这样的优化，然后又加载了一个扩展类，或者在工厂方法`getFoo()`中执行了此操作，`getFoo()`则会返回新`Foo`子类的实例，生成的代码将会是不正确的。但是JVM可以解决这个问题，同时将基于现在无效的假设使之前优化生成的代码无效，并恢复为解释代码（或重新编译无效的代码路径），也就是去优化。[2]

避免“去优化”

- 去优化的消耗大，主要是因为重新优化的消耗非常大。
- 如果我们不恰当的使用类型反馈信息，那么我们就会陷入去优化的怪圈：函数不停地去优化，然后再重新优化，直到我们达到了重优化的次数限制，这时我们的函数将再也不会JVM优化。

LoopTest代码优化

修改后的Java代码

```
public class LoopTest{
    public static int j=0;
    public static void main(String[] args) {
        //System.currentTimeMillis()获取毫秒值，但是其精度依赖操作系统，更改为
System.nanoTime()获取纳秒（ns），然后转换为毫秒（ms）
        long start = System.nanoTime()/1000000L;
        spendTime();
        long end = System.nanoTime()/1000000L;
        System.out.println(end-start);
    }
    private static void spendTime(){
        //int j=0;  //局部变量会被优化，效果见图2
        for (int i =500000000;i>0;i--) {
            j++;    //如果是空循环的话在编译模式下会被优化，见图3
        }
        System.out.println(System.getProperty("java.vm.name")); //获取JVM名字和类
型
        System.out.println(System.getProperty("java.vm.info")); //获取JVM的工作模
式
    }
}
```

新增shell脚本run_looptest.sh

```
#!/bin/sh
echo '------Xint参数-------'
starttime=`date +'%Y-%m-%d %H:%M:%S'`
/opt/jdk-13.0.2/bin/java -Xint /home/linux/code/test/LoopTest.java
endtime=`date +'%Y-%m-%d %H:%M:%S'`
start_seconds=$(date --date="$starttime" +%s);
end_seconds=$(date --date="$endtime" +%s);
echo "shell脚本运行时间：  "$((end_seconds-start_seconds))"s"
echo '----------------------'

echo '------Xcomp参数-------'
starttime=`date +'%Y-%m-%d %H:%M:%S'`
/opt/jdk-13.0.2/bin/java -Xcomp /home/linux/code/test/LoopTest.java
endtime=`date +'%Y-%m-%d %H:%M:%S'`
start_seconds=$(date --date="$starttime" +%s);
end_seconds=$(date --date="$endtime" +%s);
echo "shell脚本运行时间：  "$((end_seconds-start_seconds))"s"
echo '----------------------'
```

- 同一脚本内分别执行Xint和Xcomp

- 统计shell脚本执行时间，即Java从启动到运行结束的时间，可以发现Xcomp在Java代码执行效率高的情况下shell脚本时间反而变长，说明Xcomp启动和编译耗时较大

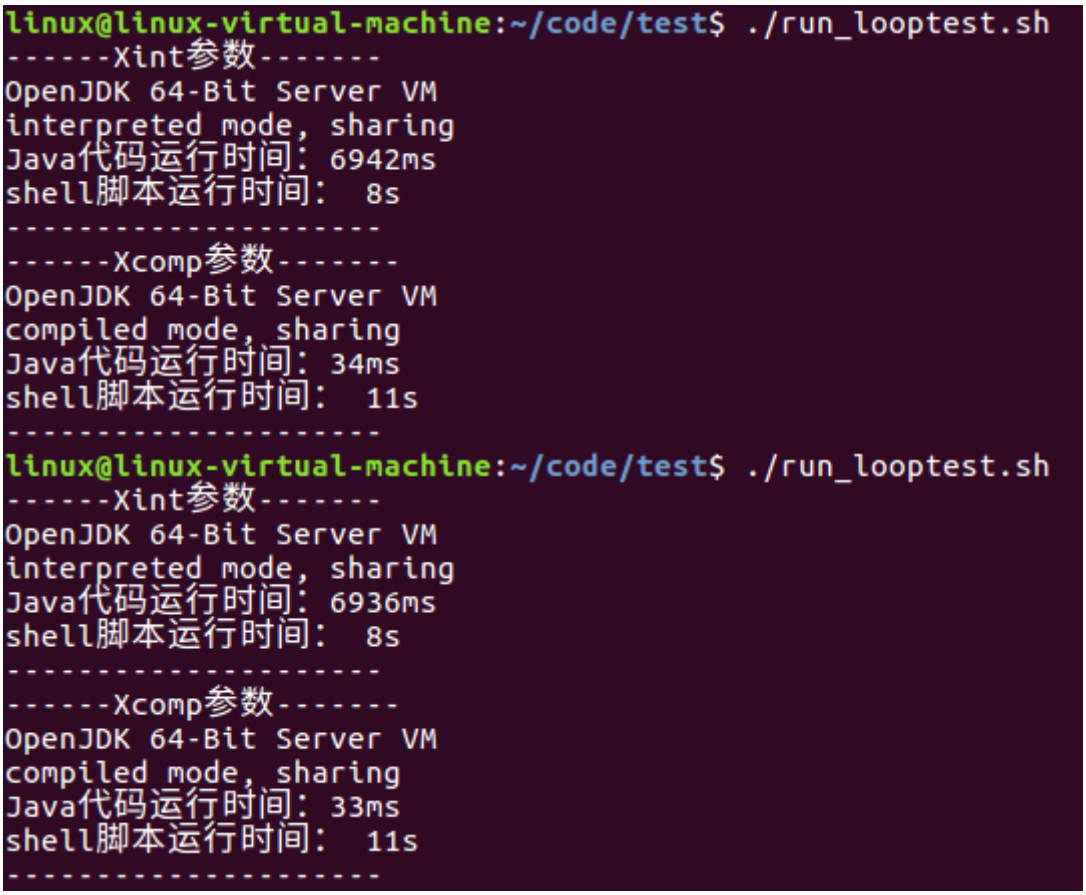

图1 修改后代码执行结果

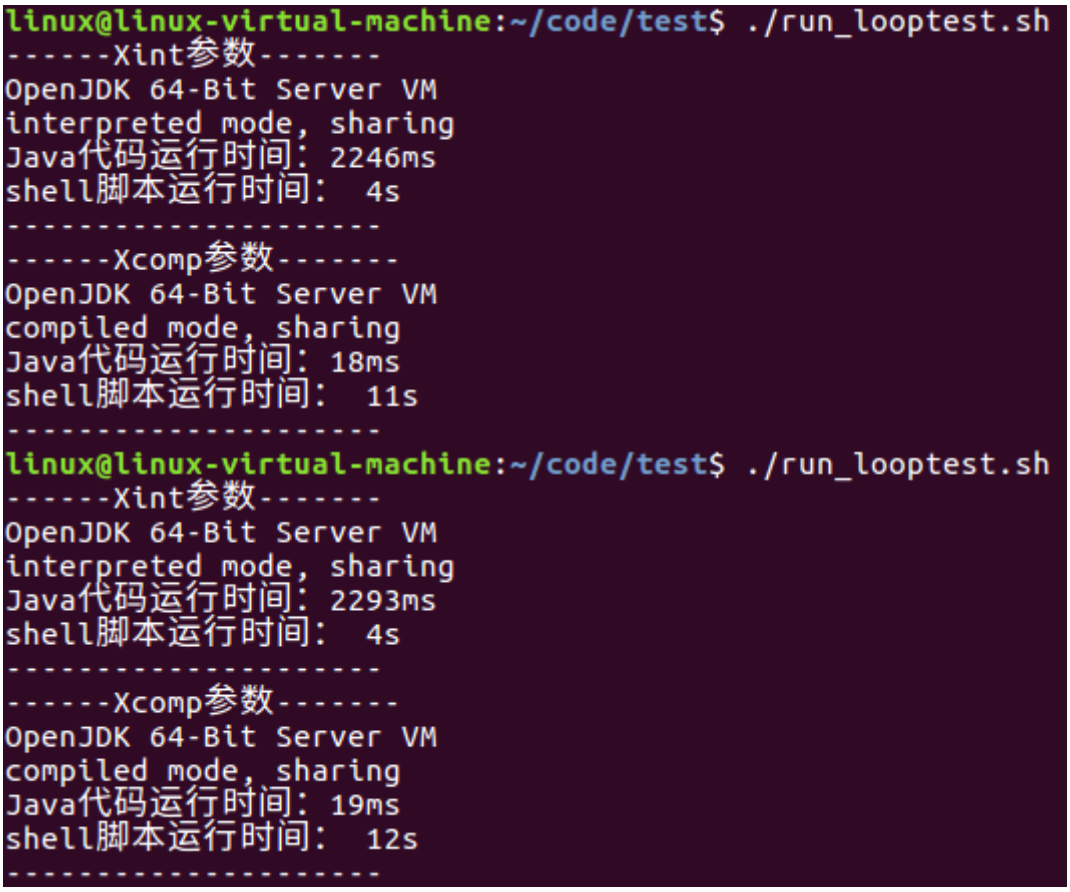

图2 int j=0为局部变量时执行结果

图3 空循环时执行结果

---

1. 参考资料：https://wiki.openjdk.java.net/display/HotSpot/PerformanceTechniques ↩ ↩

2. 参考资料：https://www.ibm.com/developerworks/library/j-jtp12214/ ↩