一般来说，Java虚拟机的即时编译过程对用户和程序都是完全透明的，虚拟机是通过解释来执行代码还是通过编译来执行代码，对于用户来说并没有什么影响（对执行结果没有影响，速度上会有显著差别），大多数情况下用户也没有必要知道。但是HotSpot虚拟机还是提供了一些参数用来输出即时编译和某些优化措施的运行状况，以满足调试和调优的需要。

本次实战部分运行参数需要FastDebug或SlowDebug优化级别的HotSpot虚拟机才能够支持，Product级别的虚拟机无法使用这部分参数。现在Oracle和OpenJDK网站上都已经不再直接提供FastDebug的JDK下载了（从JDK 6 Update 25之后官网上就没有再提供下载），所以除了自己动手编译外，就只能到网上搜索非官方编译的版本了。

测试代码：

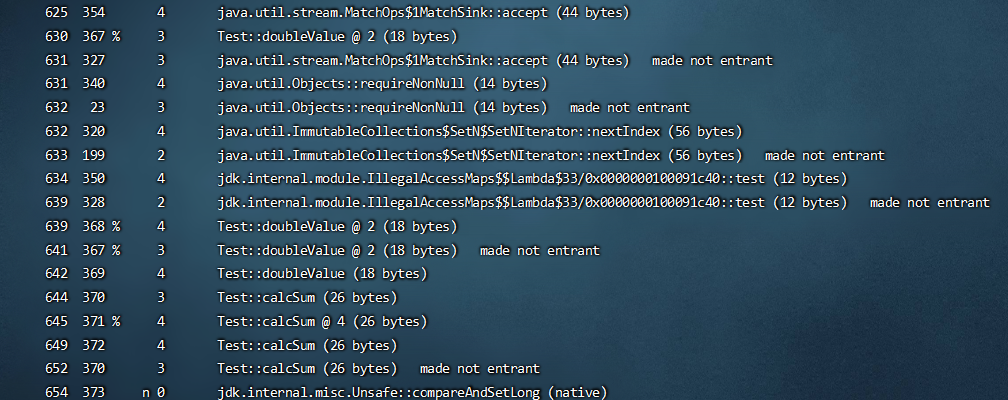
|  |
| --- |
| public static final int NUM = 15000;  public static int doubleValue(int i) {  // 这个空循环用于后面演示JIT代码优化过程  for (int j = 0; j < 100000; j++) ;  return i \* 2;  }  public static long calcSum() {  long sum = 0;  for (int i = 1; i <= 100; i++) {  sum += doubleValue(i);  }  return sum;  }  public static void main(String[] args) {  for (int i = 0; i < NUM; i++) {  calcSum();  }  } |

用自己编译的HotSpot虚拟机编译：

|  |
| --- |
| javac Test.java -encoding utf-8 |

要知道某个方法是否被编译过，可以使用参数-XX：+PrintCompilation要求虚拟机在即时编译时将被编译成本地代码的方法名称打印出来。其中带有“%”的输出说明是由回边计数器触发的栈上替换编译

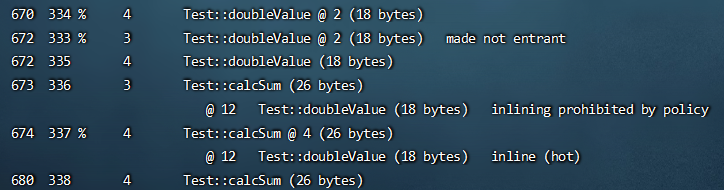
|  |
| --- |
| java -XX:+PrintCompilation Test |



可以看到main()、calcSum()和doubleValue()等方法已经被编译。

还可以加上参数-XX:+PrintInlining要求虚拟机输出方法内联信息。

|  |
| --- |
| java -XX:+PrintCompilation -XX:+PrintInlining Test |

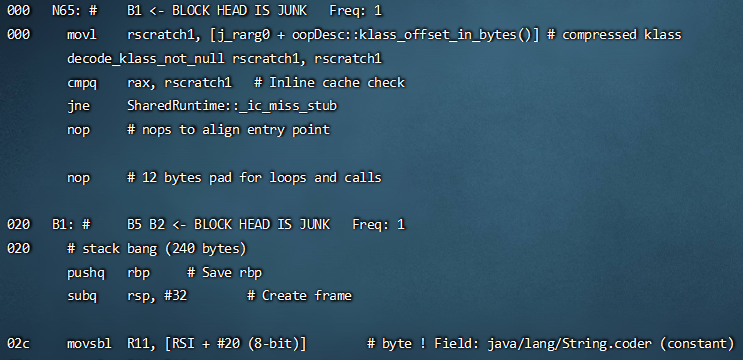


可以看到doubleValue()方法已被内联编译到calcSum()方法中，所以虚拟机再次执行calcSum()方法的时候，doubleValue()方法是不会再被实际调用的，没有任何方法分派的开销，它的代码逻辑都被直接内联到calcSum()方法里面了。

除了查看哪些方法被编译之外，我们还可以更进一步看到即时编译器生成的机器码内容。不过得到的是即时编译器输出一串0和1。机器码至少要反汇编成基本的汇编语言才可能被人类阅读。虚拟机提供了一组通用的反汇编接口，可以接入各种平台下的反汇编适配器，如使用32位x86平台应选用hsdis-i386适配器，64位则需要选用hsdis-amd64，其余平台的适配器还有如hsdis-sparc、hsdis-sparcv9和hsdis-aarch64等，可以下载或自己编译出与自己机器相符合的反汇编适配器，之后将其放置在JAVA\_HOME/lib/amd64/server下，只要与jvm.dll或libjvm.so的路径相同即可被虚拟机调用。为虚拟机安装了反汇编适配器之后，就可以使用-XX: +PrintAssembly参数要求虚拟机打印编译方法的汇编代码了。

如果没有HSDIS插件支持，也可以使用-XX:+PrintOptoAssembly（用于服务端模式的虚拟机）或-XX:+PrintLIR（用于客户端模式的虚拟机）来输出比较接近最终结果的中间代码表示

|  |
| --- |
| java -XX:+PrintOptoAssembly Test |



前面提到的使用-XX：+PrintAssembly参数输出反汇编信息需要FastDebug或SlowDebug优化级别的HotSpot虚拟机才能直接支持，如果使用Product版的虚拟机，则需要加入参数-XX:+UnlockDiagnosticVMOptions打开虚拟机诊断模式

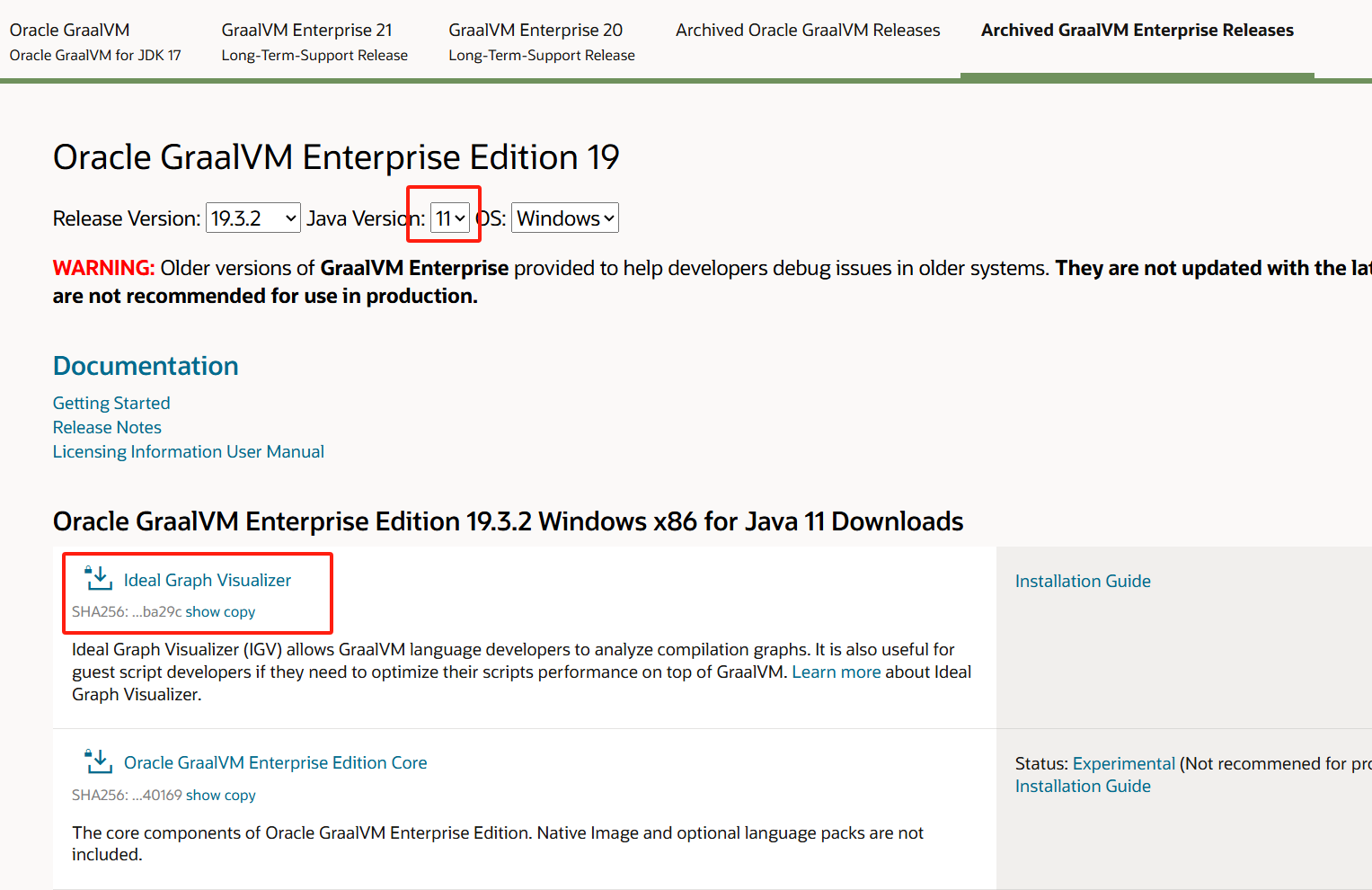
如果除了本地代码的生成结果外，还想再进一步跟踪本地代码生成的具体过程，那可以使用参数-XX:+PrintCFGToFile（用于客户端编译器）或-XX:PrintIdealGraphFile（用于服务端编译器）要求Java虚拟机将编译过程中各个阶段的数据（譬如对客户端编译器来说包括字节码、HIR生成、LIR生成、寄存器分配过程、本地代码生成等数据）输出到文件中。然后使用Java HotSpot Client Compiler Visualizer（用于分析客户端编译器）或Ideal Graph Visualizer（用于分析服务端编译器）打开这些数据文件进行分析。

使用Ideal Graph Visualizer（新版本要求高版本jdk）

|  |
| --- |
| # 下载Ideal Graph Visualizer源码  <https://github.com/openjdk/jdk/tree/master/src/utils>  # 在IdealGraphVisualizer文件夹下使用maven安装  mvn install  # 启动  ./igv.sh |

或者下载二进制应用：

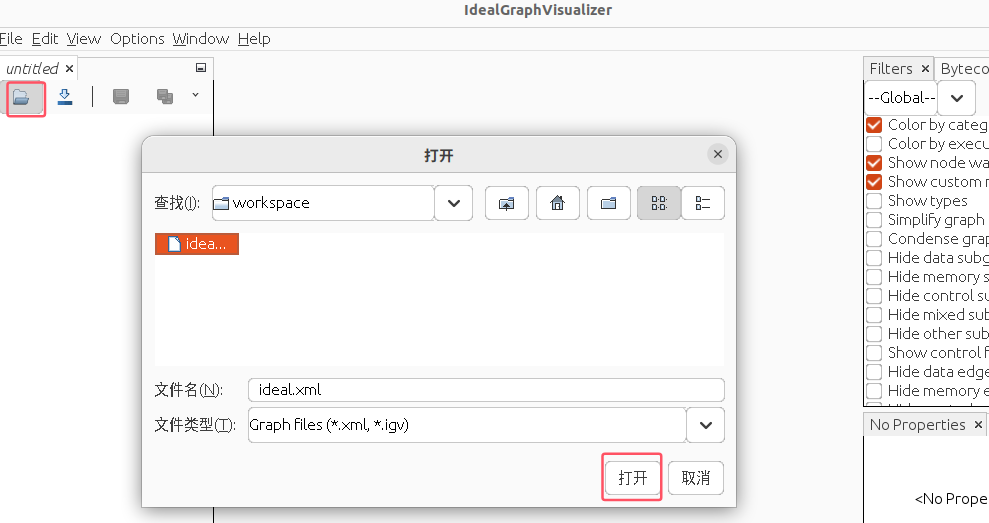
|  |
| --- |
| https://ssw.jku.at/General/Staff/TW/igv.html （OpenJdk提供）  https://www.oracle.com/downloads/graalvm-downloads.html （graal VM提供）  解压后修改文件"\idealgraphvisualizer\etc\idealgraphvisualizer.conf"，添加jdk路径（OpenJdk提供：只能是JDK6或者7；graal VM提供：链接中的jdk版本）  jdkhome="D:\program file\Java\jdk7" |



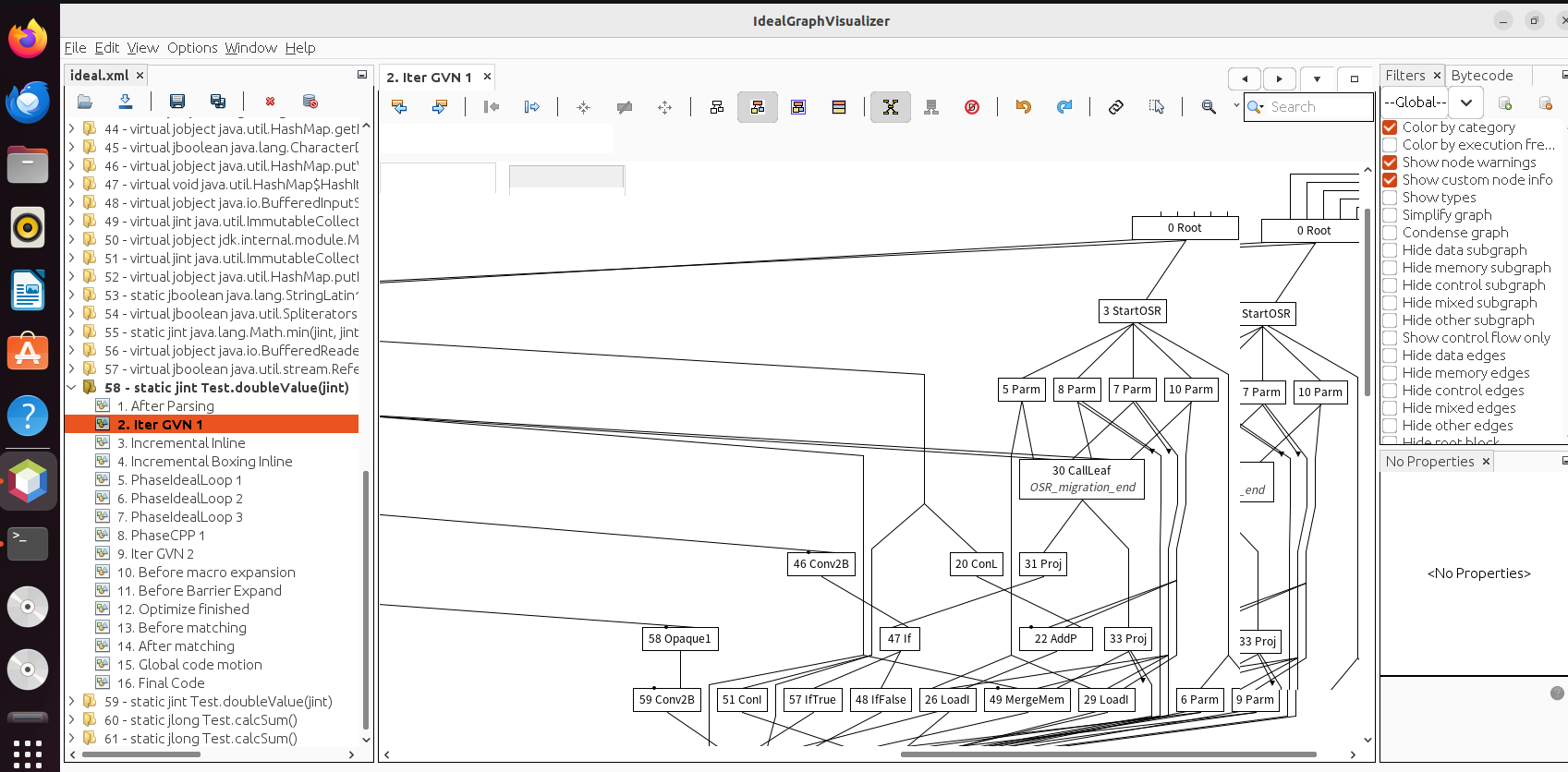
服务端编译器的中间代码表示是一种名为理想图（Ideal Graph）的程序依赖图（Program Dependence Graph，PDG），在运行Java程序的FastDebug或SlowDebug优化级别的虚拟机上的参数中加入“-XX:PrintIdealGraphLevel=2 -XX:PrintIdealGraphFile=ideal.xml”，即时编译后将会产生一个名为ideal.xml的文件，它包含了服务端编译器编译代码的全过程信息，可以使用Ideal Graph Visualizer对这些信息进行分析。

|  |
| --- |
| /development/jdk12-06222165c35f/build/linux-x86\_64-server-fastdebug/jdk/bin/java -Xbatch -XX:PrintIdealGraphLevel=2 -XX:PrintIdealGraphFile=ideal.xml Test |

* xbatch:禁止后台编译，也就是说解释执行与编译过程必须按顺序执行
* -XX:PrintIdealGraphLevel=2：设置编译输出结果级别
* -XX:PrintIdealGraphFile=ideal.xml表示将理想图文件输出到ideal.xml



查看doubleValue的理想图：



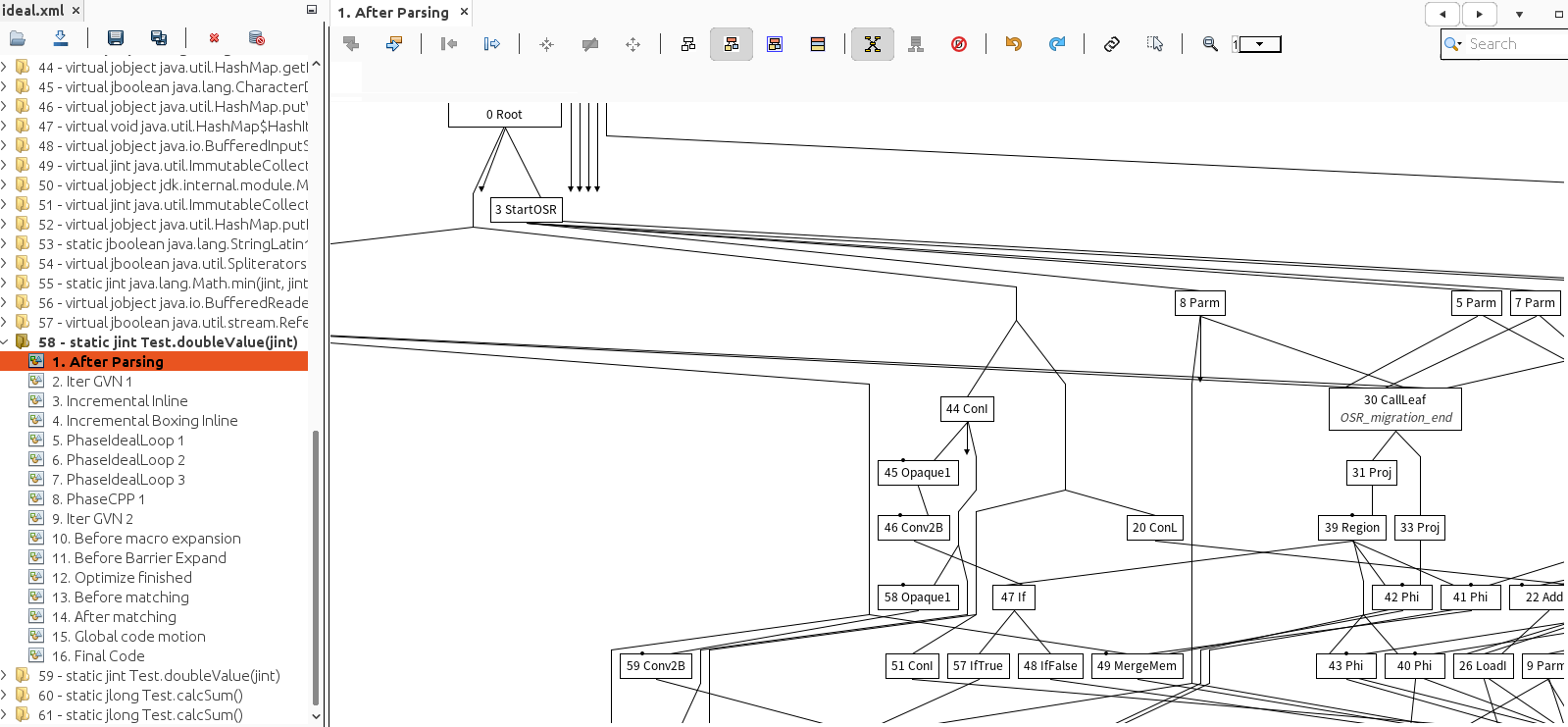
Ideal Graph Visualizer加载ideal.xml文件后，在Outline面板上将显示程序运行过程中编译过的方法列表。其中doubleValue()方法出现了两次，这是由于该方法的编译结果存在标准编译和栈上替换编译两个版本。

专门为doubleValue()方法增加了一个空循环，这个循环对方法的运算结果不会产生影响，但如果没有任何优化，执行该循环就会耗费处理器时间。

直到今天还有不少程序设计的入门教程会把空循环当作程序延时的手段来介绍。

展开方法根节点，可以看到下面罗列了方法优化过程的各个阶段（根据优化措施的不同，每个方法所经过的阶段也会有所差别）的理想图，我们先打开“After Parsing”这个阶段。即时编译器编译一个Java方法时，首先要把字节码解析成某种中间表示形式，然后才可以继续做分析和优化，最终生成代码。“After Parsing”就是服务端编译器刚完成解析，还没有做任何优化时的理想图表示。

打开这个图后，读者会看到其中有很多有颜色的方块。每一个方块代表了一个程序的基本块（Basic Block）。基本块是指程序按照控制流分割出来的最小代码块，它的特点是只有唯一的一个入口和唯一的一个出口，只要基本块中第一条指令被执行了，那么基本块内所有指令都会按照顺序全部执行一次。

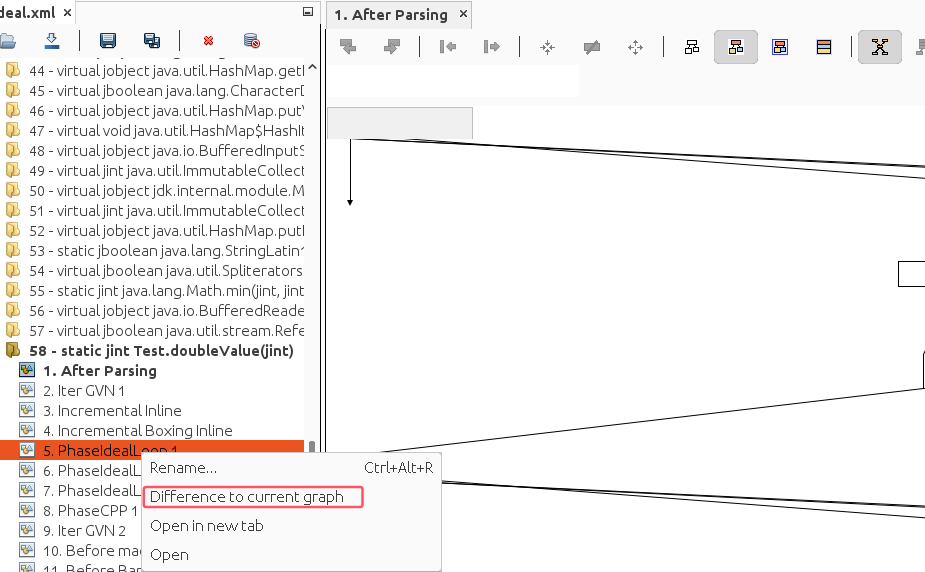


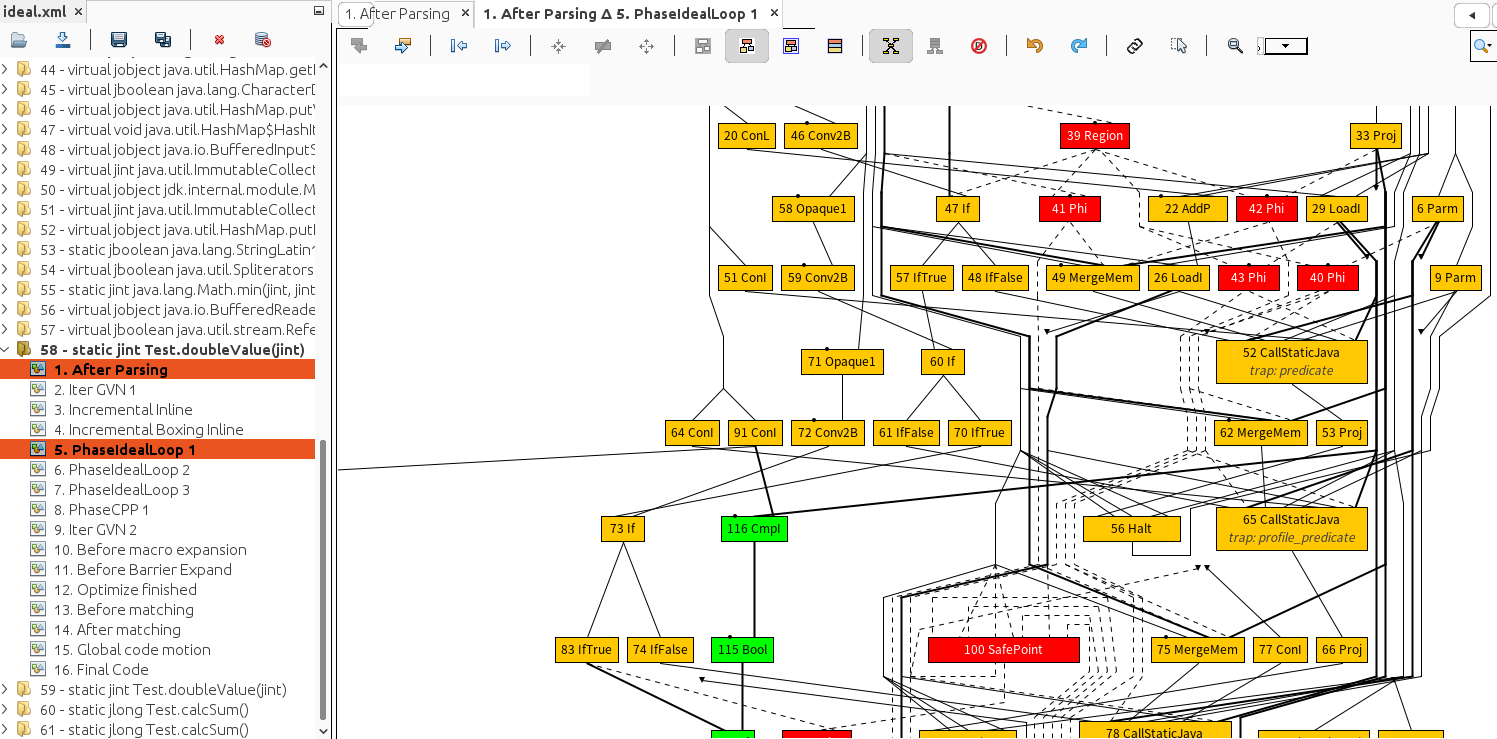
doubleValue()方法虽然只有简单的两行字，但是按基本块划分后，形成的图形结构却要比想象中复杂得多，这是因为一方面要满足Java语言所定义的安全需要（如类型安全、空指针检查）和Java虚拟机的运作需要（如Safepoint轮询），另一方面有些程序代码中一行语句就可能形成几个基本块（例如循环语句）。对于例子中的doubleValue()方法，如果忽略语言安全检查的基本块，可以简单理解为按顺序执行了以下几件事情：

1. 程序入口，建立栈帧
2. 设置j=0，进行安全点（Safepoint）轮询，跳转到4的条件检查
3. 执行j++
4. 条件检查，如果j<100000，跳转到3
5. 设置i=i\*2，进行安全点轮询，函数返回

若想看空循环是否被优化掉，或者何时被优化掉，只要观察代表循环的基本块是否被消除掉，以及何时被优化掉就可以了。

右击指定阶段，然后点击“Difference to current graph” ，让软件自动分析指定阶段与当前打开的理想图之间的差异，如果基本块被消除了，将会以红色显示。





对比“After Parsing”和“PhaseIdealLoop 1”阶段的理想图进行差异分析，会发现在“PhaseIdealLoop 1”阶段循环操作就被消除了，这也就说明空循环在最终的本地代码里实际上是不会被执行的。

从“After Parsing”阶段开始，一直到最后的“Final Code”阶段都可以看到doubleValue()方法的理想图从繁到简的变迁过程，这也反映了Java虚拟机即时编译器尽力优化代码的过程。到了最后的“Final Code”阶段，不仅空循环的开销被消除了，许多语言安全保障措施和GC安全点的轮询操作也被一起消除了，因为编译器判断到即使不做这些保障措施，程序也能得到相同的结果，不会有可观察到的副作用产生，虚拟机的运行安全也不会受到威胁。