# 历史背景

Graal虚拟机以及Graal编译器仍在实验室中尚未商用，但未来其有望代替或成为HotSpot下一代技术基础。Graal编译器最初是在Maxine虚拟机中作为C1X编译器的下一代编译器而设计的，所以它理所当然地使用于Java语言来编写。2012年，Graal编译器从Maxine虚拟机项目中分离，成为一个独立发展的Java编译器项目，Oracle Labs希望它最终能够成为一款高编译效率、高输出质量、支持提前编译和即时编译，同时支持应用于包括HotSpot在内的不同虚拟机的编译器。

由于这个编译器使用Java编写，代码清晰，又继承了许多来自HotSpot的服务端编译器的高质量优化技术，所以无论是科技企业还是高校研究院，都愿意在它上面研究和开发新编译技术。HotSpot服务端编译器的创造者Cliff Click自己就对Graal编译器十分推崇，并且公开表示再也不会用C、C++去编写虚拟机和编译器了。Twitter的Java虚拟机团队也曾公开说过C2目前犹如一潭死水，亟待一个替代品，因为在它上面开发、改进实在太困难了。

可以从github克隆openjdk镜像：

|  |
| --- |
| $ git clone https://github.com/dmlloyd/openjdk.git |

C2代码位于openjdk/hotspot/src/share/vm/opto

Graal编译器在JDK 9时以Jaotc提前编译工具的形式首次加入到官方的JDK中，从JDK 10起，Graal编译器可以替换服务端编译器，成为HotSpot分层编译中最顶层的即时编译器。这种可替换的即时编译器架构的实现，得益于HotSpot编译器接口的出现。

早期的Graal曾经同C1及C2一样，与HotSpot的协作是紧耦合的，这意味着每次编译Graal均需重新编译整个HotSpot。JDK 9时发布的JEP 243：Java虚拟机编译器接口（Java-Level JVM Compiler Interface，JVMCI）使得Graal可以从HotSpot的代码中分离出来。JVMCI主要提供如下三种功能：

* 响应HotSpot的编译请求，并将该请求分发给Java实现的即时编译器。
* 允许编译器访问HotSpot中与即时编译相关的数据结构，包括类、字段、方法及其性能监控数据等，并提供了一组这些数据结构在Java语言层面的抽象表示。
* 提供HotSpot代码缓存（Code Cache）的Java端抽象表示，允许编译器部署编译完成的二进制机器码。

综合利用上述三项功能，我们就可以把一个在HotSpot虚拟机外部的、用Java语言实现的即时编译器（不局限于Graal）集成到HotSpot中，响应HotSpot发出的最顶层的编译请求，并将编译后的二进制代码部署到HotSpot的代码缓存中。

此外，单独使用上述第三项功能，又可以绕开HotSpot的即时编译系统，让该编译器直接为应用的类库编译出二进制机器码，将该编译器当作一个提前编译器去使用（如Jaotc）。

# 自己编译Graal编译器

由于Graal编译器要同时支持Graal VM下的各种子项目，如Truffle、Substrate VM、Sulong等，还要支持作为HotSpot和Maxine虚拟机的即时编译器，所以只用Maven或Gradle的话，配置管理过程会相当复杂。为了降低代码管理、依赖项管理、编译和测试等环节的复杂度，Graal团队专门用Python 2写了一个名为mx的小工具来自动化做好这些事情。我们要构建Graal的调试环境，第一步要先把构建工具mx安装好，这非常简单，进行如下操作即可：

|  |
| --- |
| $ git clone <https://github.com/graalvm/mx.git>  $ cd mx; git checkout 7353064  $ export PATH=`pwd`/mx:$PATH |

'mx vm'等同于'java'

mx需要python2：

|  |
| --- |
| apt-get install python2.7 |

既然Graal编译器是以Java代码编写的，那第二步自然是要找一个合适的JDK来编译。考虑到Graal VM项目是基于OpenJDK 8开发的，而JVMCI接口又在JDK 9以后才会提供，所以Graal团队提供了一个带有JVMCI功能的OpenJDK 8版本，我们可以选择这个版本的JDK 8来进行编译。

|  |
| --- |
| https://github.com/graalvm/openjdk8-jvmci-builder/releases  https://github.com/graalvm/graal-jvmci-8  $ export JAVA\_HOME=/development/oraclejdk1.8.0\_212-jvmci-20-b01 |

但是本次实验使用的是jdk9：

|  |
| --- |
| <https://www.oracle.com/java/technologies/javase/javase9-archive-downloads.html>  root@deng4j-virtual-machine:/development# java -version  java version "9"  Java(TM) SE Runtime Environment (build 9+181)  Java HotSpot(TM) 64-Bit Server VM (build 9+181, mixed mode) |

graal使用的接口叫做jvmci，由JEP 243Java-Level JVM Compiler Interface提案加入到Java中。第一个实现提案的版本是java9

第三步是获取Graal编译器代码，编译器部分的代码是与整个Graal VM放在一块的，我们把Graal VM复制下来，大约有700MB，操作如下：

|  |
| --- |
| $ git clone https://github.com/graalvm/graal.git --branch vm-enterprise-0.28.2 |

其他目录中存放着Truffle、Substrate VM、Sulong等其他项目，这些在本次实战中不会涉及。进入compiler子目录，使用mx构建Graal编译器，操作如下：

|  |
| --- |
| $ cd graal/compiler  $ mx build |

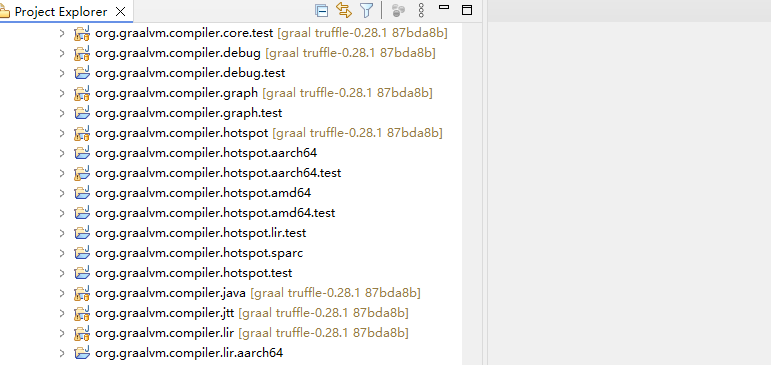
由于整个构建过程需要的依赖项都可以自动处理，需要手动处理的只有OpenJDK一个，所以编译一般不会出现什么问题，大概两三分钟编译即可完成。此时其实已经可以修改、调试Graal编译器了，但写Java代码不同于C、C++，应该没有人会直接用VIM去做Java开发调试，我们还是需要一个IDE来支持本次实战的。

mx工具能够支持Eclipse、Intellij IDEA和NetBeans三种主流的Java IDE项目的创建（Graal团队中使用Eclipse占多数，支持也最好），操作如下：

|  |
| --- |
| $ cd graal/compiler  # 生成指定编辑器的工程配置文件  $ mx eclipseinit  $ mx intellijinit  $ mx netbeansinit |

无论使用哪种IDE，都需要把IDE配置中使用的Java堆修改到2GB或以上，才能保证Graal在IDE中的编译构建能够顺利进行。譬如Eclipse默认配置（eclipse.ini文件）下的Java堆最大为1GB，在Eclipse中选择File->Open Projects from File System，再选择Graal项目的根目录，将

会导入整个Graal VM，导入的工程如图：



本次实验采用jdk9编译，记得在Eclipse中也必须将那个带有JVMCI功能的特殊JDK9用作Eclipse里面的环境配置（Windows -> Preferences -> Java -> Install JREs -> Execution Environments）。此外，还需要手工将以其他版本号结尾的工程关闭。

# JVMCI编译器接口

## 介绍

来思考一下，如果让您来设计JVMCI编译器接口，它应该是怎样的？既然JVMCI面向的是Java语言的编译器接口，那它至少在形式上是与我们已经见过无数次的Java接口是一样的。

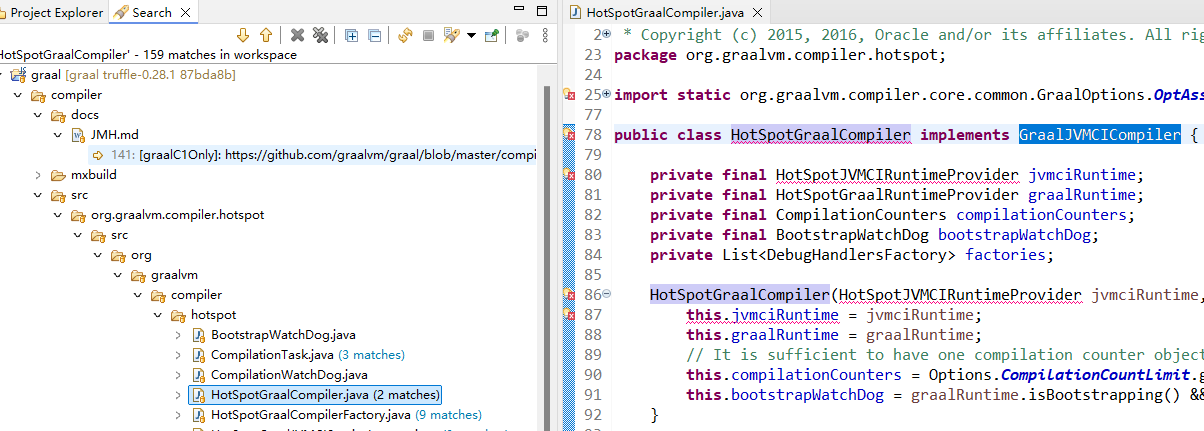
我们来考虑即时编译器的输入是什么。答案当然是要编译的方法的字节码。既然叫字节码，顾名思义它就应该是“用一个字节数组表示的代码”。那接下来它输出什么？这也很简单，即时编译器应该输出与方法对应的二进制机器码，二进制机器码也应该是“用一个字节数组表示的代码”。这样的话，JVMCI接口就应该看起来类似于下面这种样子：

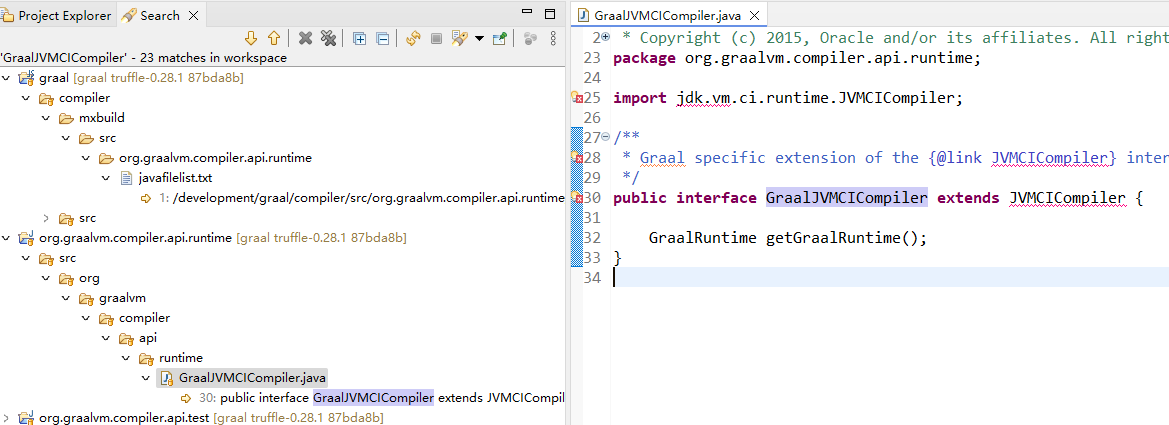
|  |
| --- |
| interface JVMCICompiler {  byte[] compileMethod(byte[] bytecode);  } |

事实上JVMCI接口只比上面这个稍微复杂一点点，因为其输入除了字节码外，HotSpot还会向编译器提供各种该方法的相关信息，譬如局部变量表中变量槽的个数、操作数栈的最大深度，还有分层编译在底层收集到的统计信息等。因此JVMCI接口的核心内容实际是：

|  |
| --- |
| interface JVMCICompiler {  void compileMethod(CompilationRequest request);  }  interface CompilationRequest {  JavaMethod getMethod();  }  interface JavaMethod {  byte[] getCode();  int getMaxLocals();  int getMaxStackSize();  ProfilingInfo getProfilingInfo();  ... // 省略其他方法  } |

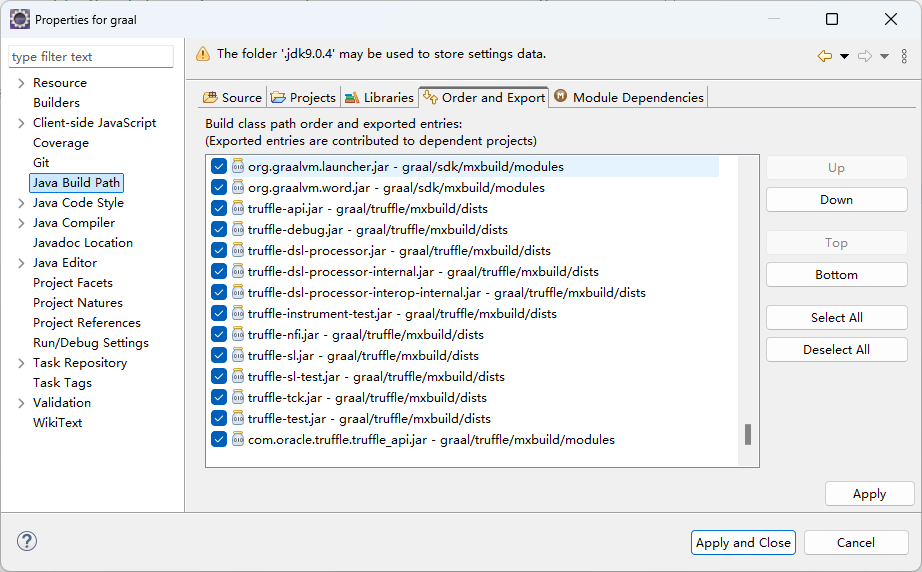
在Eclipse中找到JVMCICompiler接口，通过继承关系分析，可以清楚地看到有一个实现类HotSpotGraalCompiler实现了JVMCI，这个就是我们要分析的代码的入口。





有报红，是没有导入源码中的jar

选中项目 -> File -> Properties

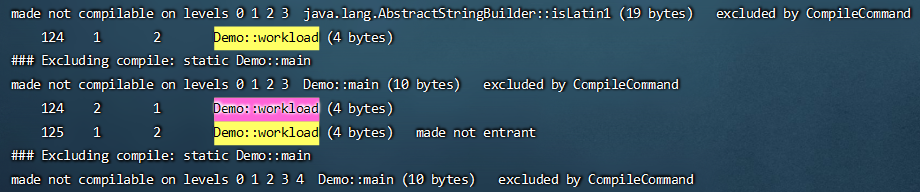


为了后续调试方便，我们先准备一段简单的代码，并让它触发HotSpot的即时编译，以便我们跟踪观察编译器是如何工作对的。

|  |
| --- |
| class Demo {  public static void main(String[] args) {  while (true) {  workload(14, 2);  }  }  private static int workload(int a, int b) {  return a + b;  }  } |

由于存在无限循环，workload()方法肯定很快就会被虚拟机发现是热点代码因而进行编译。实际上除了workload()方法以外，这段简单的代码还会导致相当多的其他方法的编译，因为一个最简单的Java类的加载和运行也会触发数百个类的加载。为了避免干扰信息太多，笔者加入了参数-XX: CompileOnly来限制只允许workload()方法被编译。

|  |
| --- |
| javac Demo.java  java -XX:+PrintCompilation -XX:CompileOnly=Demo::workload Demo |



上面显示wordload()方法确实被分层编译了多次，“made not entrant”的输出就表示了方法的某个已编译版本被丢弃过。从这段信息中清楚看到，分层编译机制及最顶层的服务端编译都已经正常工作了，下一步就是用我们在Eclipse中的Graal编译器代替HotSpot的服务端编译器。

## 使用自己编译的Graal编译器

为简单起见，加上-XX:-TieredCompilation关闭分层编译，让虚拟机只采用有一个JVMCI编译器而不是由客户端编译器和JVMCI混合分层。然后使用参数-XX:+EnableJVMCI、-XX:+UseJVMCICompiler来启用JVMCI接口和JVMCI编译器。

由于这些目前尚属实验阶段的功能，需要再使用-XX: +UnlockExperimentalVMOptions参数进行解锁。最后，也是最关键的一个问题，如何让HotSpot找到Graal编译器的位置呢？

如果采用带有jvmci的JDK8 ，那虚拟机将会自动去查找JAVA\_HOME/jre/lib/jvmci目录。假如这个目录不存在，那就会从-Djvmci.class.path.append参数中搜索。它查找的目标，即Graal编译器的JAR包。通过mx build命令成功编译出来后，在带有jvmci的JDK8下使用启动参数：

带有jvmci的JDK8的运行配置：

|  |
| --- |
| -Djvmci.class.path.append=~/graal/compiler/mxbuild/dists/jdk1.8/graal.jar:~/graal/sdk/mxbuild/dists/jdk1.8/graal  -XX:+UnlockExperimentalVMOptions  -XX:+EnableJVMCI  -XX:+UseJVMCICompiler  -XX:-TieredCompilation  -XX:+PrintCompilation  -XX:CompileOnly=Demo::workload |

如果采用JDK 9或以上版本，那原本的Graal编译器是实现在jdk.internal.vm.compiler模块中的，只要用--upgrade-module-path参数指定这个模块的升级包即可。

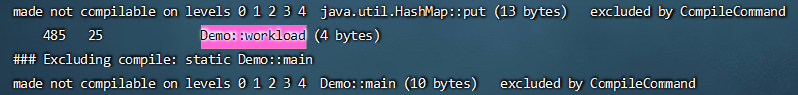
JDK9或以上版本的运行配置：

|  |
| --- |
| --module-path=~/graal/sdk/mxbuild/dists/jdk11/graal.jar  --upgrade-module-path=~graal/compiler/mxbuild/dists/jdk11/jdk.internal.vm.compiler.jar  -XX:+UnlockExperimentalVMOptions  -XX:+EnableJVMCI  -XX:+UseJVMCICompiler  -XX:-TieredCompilation  -XX:+PrintCompilation  -XX:CompileOnly=Demo::workloa |

通过上述参数，HotSpot就能顺利找到并应用Graal编译器了。

本次实验采用的jdk9，自己编译的graal编译器：

|  |
| --- |
| java \  --module-path=/development/graal/sdk/mxbuild/modules/org.graalvm.graal\_sdk.jar:/development/graal/truffle/mxbuild/modules/com.oracle.truffle.truffle\_api.jar \  --upgrade-module-path=/development/graal/compiler/mxbuild/modules/jdk.internal.vm.compiler.jar \  -XX:+UnlockExperimentalVMOptions \  -XX:+EnableJVMCI \  -XX:+UseJVMCICompiler \  -XX:-TieredCompilation \  -XX:+PrintCompilation \  -XX:CompileOnly=Demo::workload \  Demo |



对HotSpotGraalCompiler类的compileMethod()方法做一个简单改动，输出编译的方法名称：

/development/graal/compiler/src/org.graalvm.compiler.hotspot/src/org/graalvm/compiler/hotspot/HotSpotGraalCompiler.java：

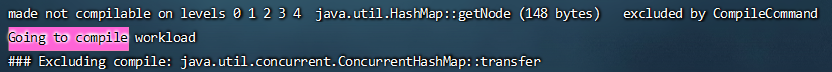
|  |
| --- |
| @Override  public CompilationRequestResult compileMethod(CompilationRequest request) {  System.err.println("Going to compile " + request.getMethod().getName());  return compileMethod(request, true);  } |

* eclipse调试方式：如果用eclipse调试，就不需要重新mx build，可以立刻把修改好的编译器插入到现有的jvm。
* 重新构建方式：

|  |
| --- |
| $ mx build |

运行：

|  |
| --- |
| $ java \  --module-path=/development/graal/sdk/mxbuild/modules/org.graalvm.graal\_sdk.jar:/development/graal/truffle/mxbuild/modules/com.oracle.truffle.truffle\_api.jar \  --upgrade-module-path=/development/graal/compiler/mxbuild/modules/jdk.internal.vm.compiler.jar \  -XX:+UnlockExperimentalVMOptions \  -XX:+EnableJVMCI \  -XX:+UseJVMCICompiler \  -XX:-TieredCompilation \  -XX:+PrintCompilation \  -XX:CompileOnly=Demo::workload \  Demo |



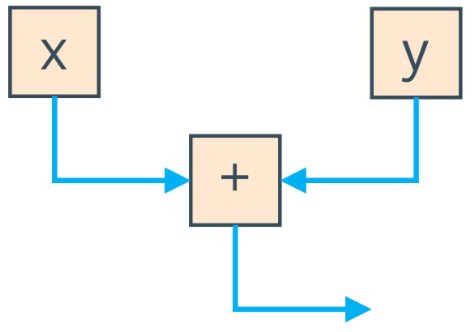
重新运行后就可以看到刚刚修改的东西了

# 代码中间表示

Graal编译器在设计之初就刻意采用了与HotSpot服务端编译器一致（略有差异但已经非常接近）的中间表示形式，也即是被称为Sea-of-Nodes的中间表示，或者与其等价的被称为理想图（Ideal Graph，在代码中称为Structured Graph）的程序依赖图（Program Dependence Graph，PDG）形式。

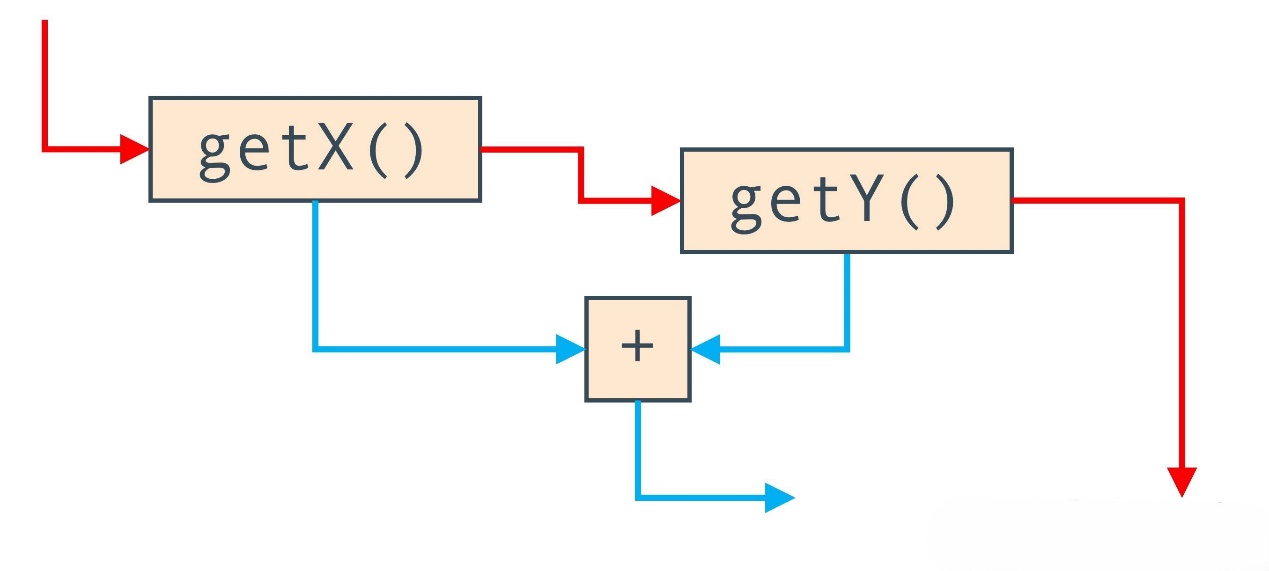
通过可视化工具Ideal Graph Visualizer看到过在理想图上翻译和优化输入代码的整体过程，从编译器内部来看即：字节码→理想图→优化→机器码（以Mach Node Graph表示）的转变过程。

不熟悉编译原理与编译器设计，可能会不太容易读懂每个阶段所要做的工作。下面用例子和对照Graal源码的形式，详细讲解输入代码与理想图的转化对应关系，以便理解Graal是如何基于理想图去优化代码的。



蓝色的边表示数据流，它们读取局部变量的值，流入加法node。可以使用边表示程序运行的顺序。

如果是调用两个方法而不是读取两个局部变量，比如getX()，getY()。这时候除了数据流向之外，还必须要考虑方法调用的顺序。在理想图中用另外一条边来表示方法的调用，即下面的红边表示控制流。



理想图本质上就是这种将数据流图和控制流图以某种方式合并到一起，用一种边来表示数据流向，另一种边来表示控制流向的图形表示。

现在在运行参数中再增加一个参数-Dgraal.Dump，要求Graal编译器把构造的理想图输出来：

|  |
| --- |
| java \  --module-path=/development/graal/sdk/mxbuild/modules/org.graalvm.graal\_sdk.jar:/development/graal/truffle/mxbuild/modules/com.oracle.truffle.truffle\_api.jar \  --upgrade-module-path=/development/graal/compiler/mxbuild/modules/jdk.internal.vm.compiler.jar \  -XX:+UnlockExperimentalVMOptions \  -XX:+EnableJVMCI \  -XX:+UseJVMCICompiler \  -XX:-TieredCompilation \  -XX:+PrintCompilation \  -XX:CompileOnly=Demo::workload \  -Dgraal.Dump \  Demo |

加入后编译时将会产生类似如下的输出，提示了生成的理想图的存储位置：



生成的文件：

|  |
| --- |
| HotSpotCompilation-25[Demo.workload(int,int)].bgv  HotSpotCompilation-25[Demo.workload(int,int)].cfg |

* \*.bgv文件：这是Graal编译器生成的理想图文件
* \*.cfg文件：控制流图文件

使用mx工具启动IGV，如果没有会下载：

|  |
| --- |
| $ mx -p /development/graal/compiler igv |

* mx：这是一个用于构建和管理Graal项目的工具
* /development/graal/compiler：配置Graal编译器
* igv：表示启动Ideal Graph Visualizer工具

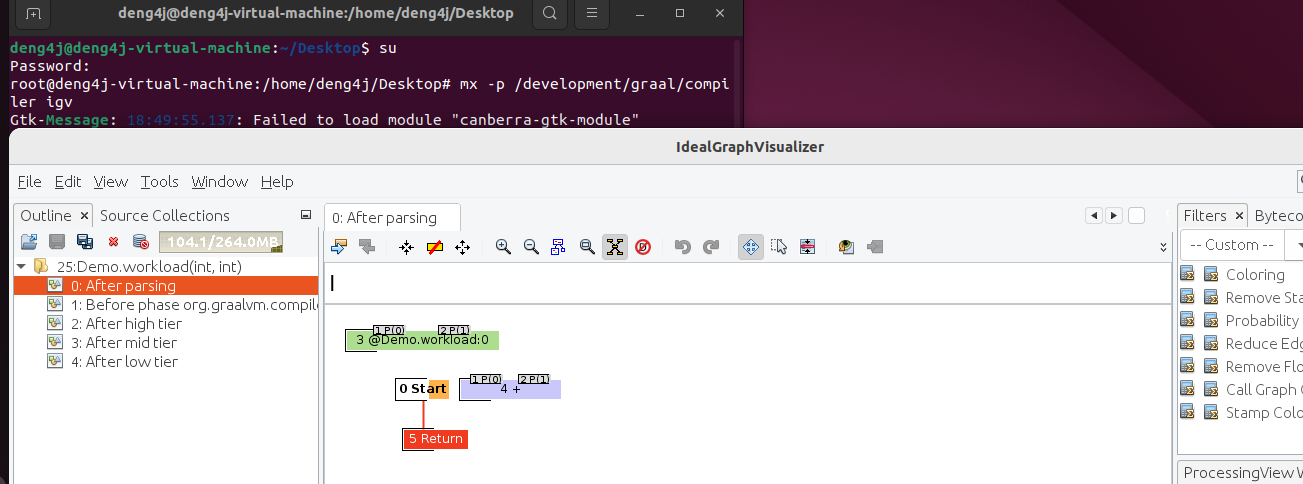
下载后启动报错：

|  |
| --- |
| $ mx -p /development/graal/compiler igv  /development/graal/compiler/mxbuild/idealgraphvisualizer/platform/lib/nbexec: WARNING: environment variable DISPLAY is not set |

这个错误提示意味着系统中没有设置DISPLAY环境变量。DISPLAY环境变量通常用于指定图形输出的位置。解决：

|  |
| --- |
| $ echo $DISPLAY  $ export DISPLAY=localhost:0.0 |

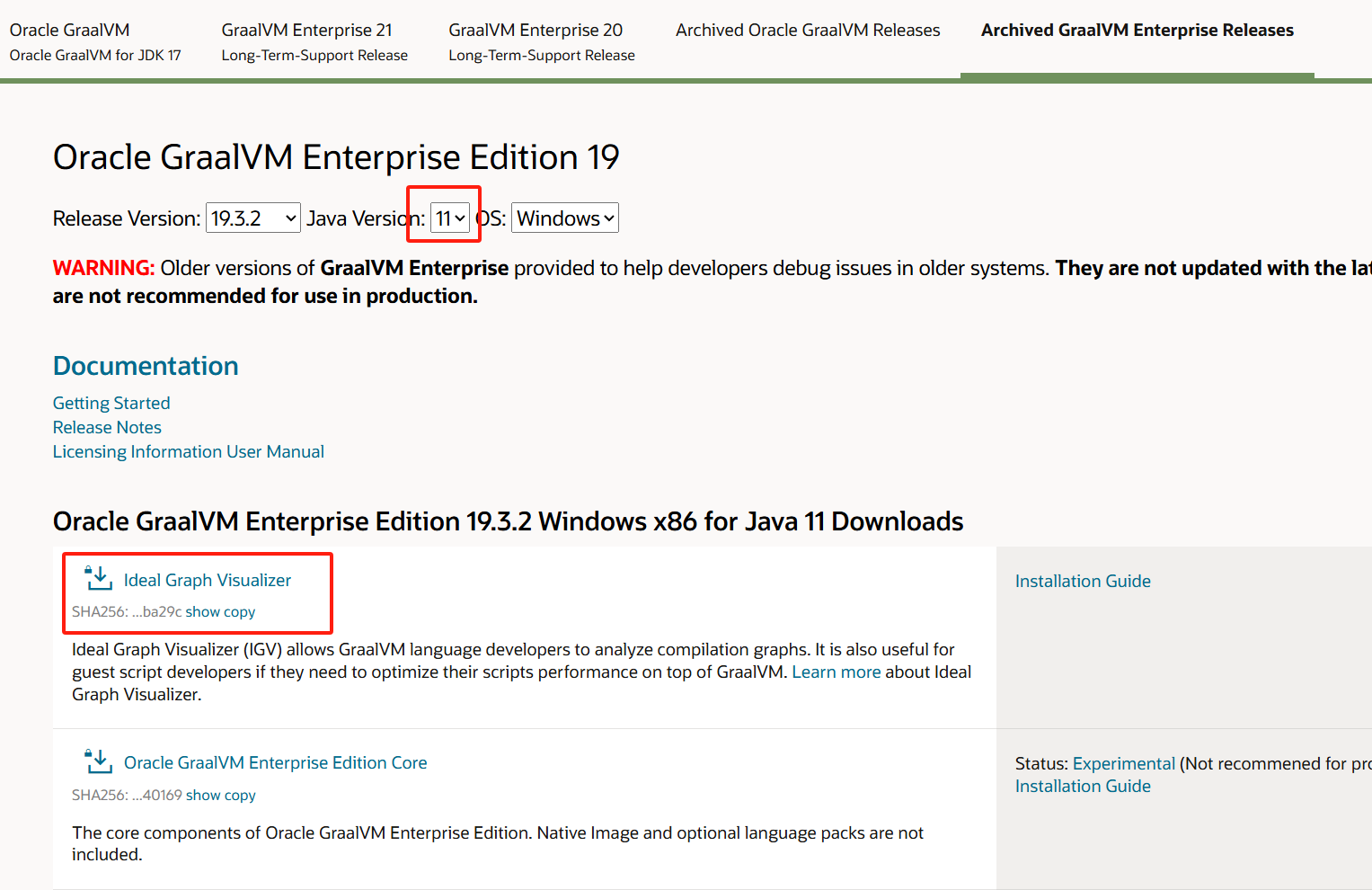
重新启动IGV，打开\*.bgv文件：



打开IGV后，Graal编译器把构造的理想图输出来后，GraalVM会在终端输出“Connected to the IGV on 127.0.0.1:4445”，这表明它连接上了idealgraphvisualizer。接着，在即时编译之后，idealgraphvisualizer就接收到了编译过程中生成的图。

但是太糊了，看不了，用graal VM提供的：

|  |
| --- |
| <https://www.oracle.com/downloads/graalvm-downloads.html>  解压后修改文件"\idealgraphvisualizer\etc\idealgraphvisualizer.conf"，添加jdk路径（链接中的jdk版本）  jdkhome="D:\program file\Java\jdk7" |

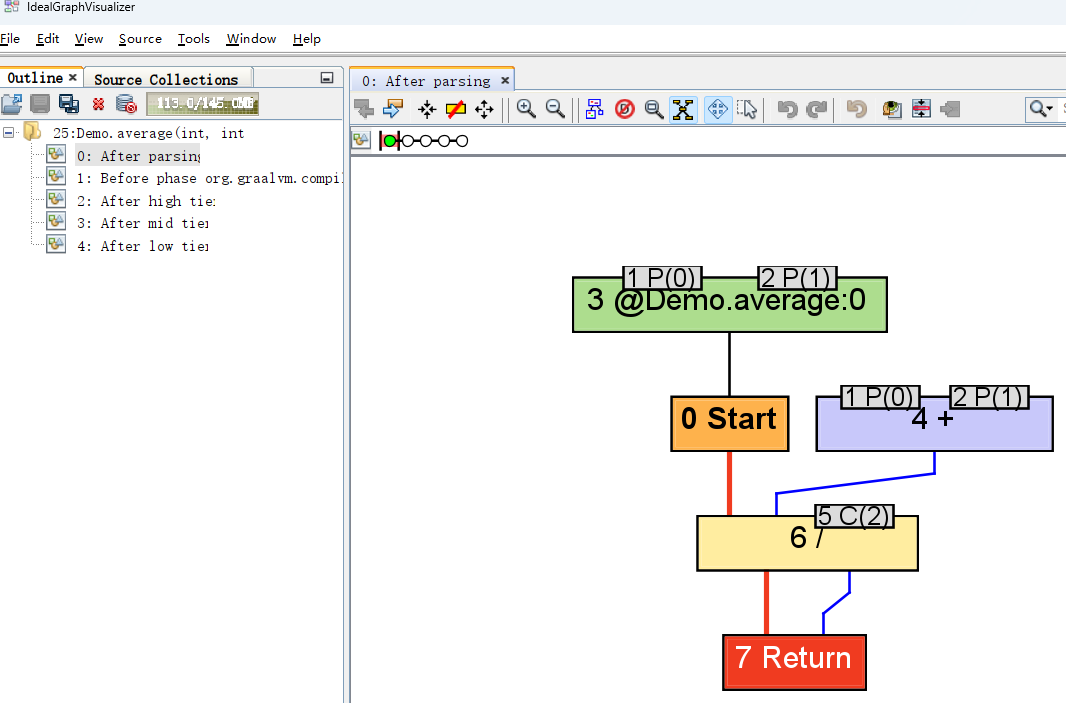


可以写个简单的表达式看看数据流：

|  |
| --- |
| class Demo {  public static void main(String[] args) {  while (true) {  average(14, 2);  }  }  private static int average(int a, int b) {  return (a + b) / 2;  }  } |

启动，生成理想图：

|  |
| --- |
| java \  --module-path=/development/graal/sdk/mxbuild/modules/org.graalvm.graal\_sdk.jar:/development/graal/truffle/mxbuild/modules/com.oracle.truffle.truffle\_api.jar \  --upgrade-module-path=/development/graal/compiler/mxbuild/modules/jdk.internal.vm.compiler.jar \  -XX:+UnlockExperimentalVMOptions \  -XX:+EnableJVMCI \  -XX:+UseJVMCICompiler \  -XX:-TieredCompilation \  -XX:+PrintCompilation \  -XX:CompileOnly=Demo::average \  -Dgraal.Dump \  Demo |



虽然没有了箭头，但是节点上列明了代表执行顺序的序号，仍然是蓝色线表示数据流、红色线表示控制流。从图中可以看到参数0（记作P(0)）和参数1（记作P(1)）是如何送入加法操作的，然后结果是如何和常量2（记作C(2)）一起送入除法操作的。

再下一步我们就会开始接触真实的代码编译和优化了。前面介绍编译器优化技术时提到过公共子表达式消除。

公共子表达式被消除的应用范围：

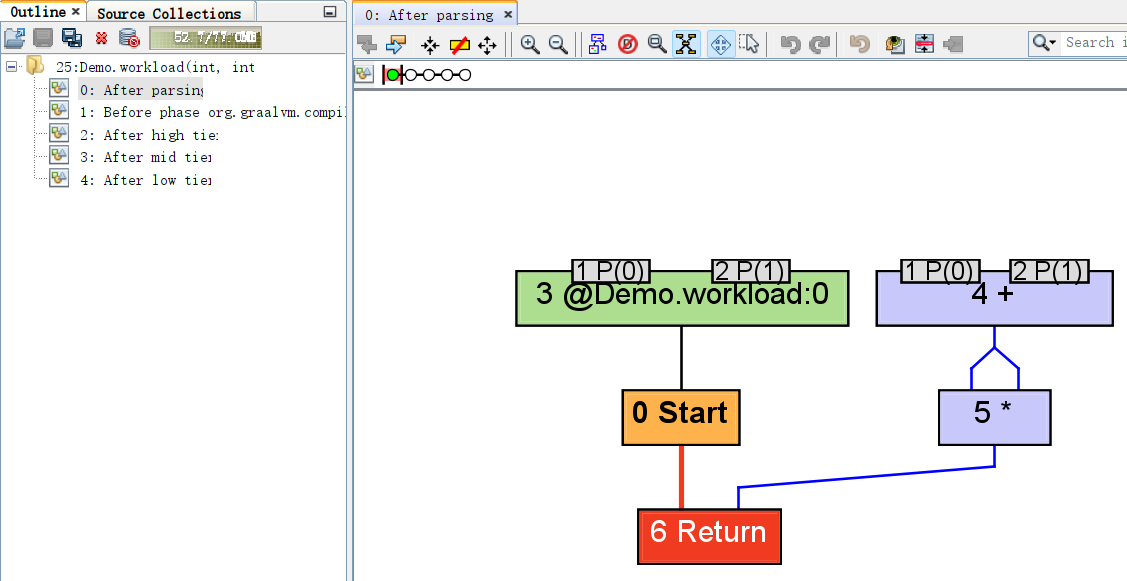
|  |
| --- |
| // 以下代码的公共子表达式能够被消除  private static int workload(int a, int b) {  return (a + b) \* (a + b);  }  // 以下代码的公共子表达式是不可以被消除的  private static int workload() {  return (getA() + getB()) \* (getA() + getB());  } |

对于第一段代码，a+b是公共子表达式，可以通过优化使其只计算一次而不会有任何的副作用。

但是对于第二段代码，由于getA()和getB()方法内部所蕴含的操作是不确定的，它是否被调用、调用次数的不同都可能会产生不同返回值或者其他影响程序状态的副作用（譬如改变某个全局的状态变量），这种代码只能内联了getA()和getB()方法之后才能考虑更进一步的优化措施，仍然保持函数调用的情况下是无法做公共子表达式消除的。我们可以从Graal生成的理想图中清晰地看到这一点。

第一段代码：

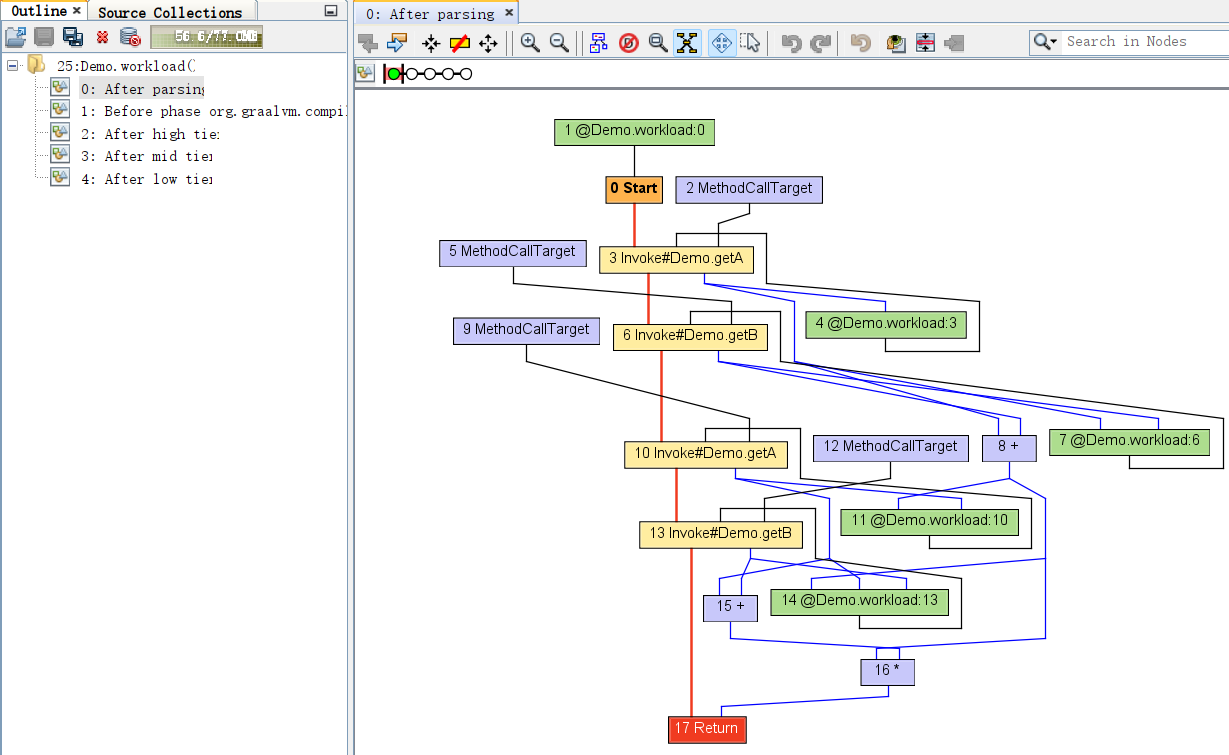
|  |
| --- |
| class Demo {  public static void main(String[] args) {  while (true) {  workload(14, 2);  }  }  private static int workload(int a, int b) {  return (a + b) \* (a + b);  }  } |



从图中可以看到，参数1、2的加法操作只进行了一次，然后同时流出了两条数据流指向乘法操作的输入中。

第二段代码：

|  |
| --- |
| class Demo {  public static void main(String[] args) {  while (true) {  workload();  }  }  private static int workload() {  return (getA() + getB()) \* (getA() + getB());  }  private static int getA() {  return 5;  }  private static int getB() {  return 3;  }  } |

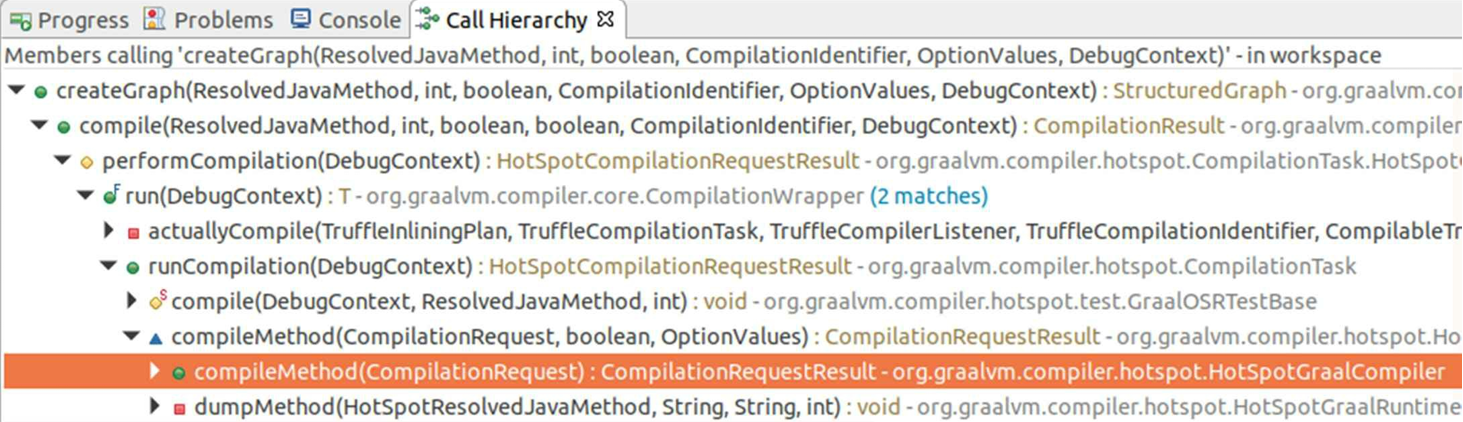


从图中代表控制流的红色边（以实线表示）可以看出，四次方法调用全部执行了，代表数据流的蓝色边也明确看到了两个独立加法操作节点，由此看出这个版本是不会把它当作公共子表达式来消除的。

# 代码优化与生成

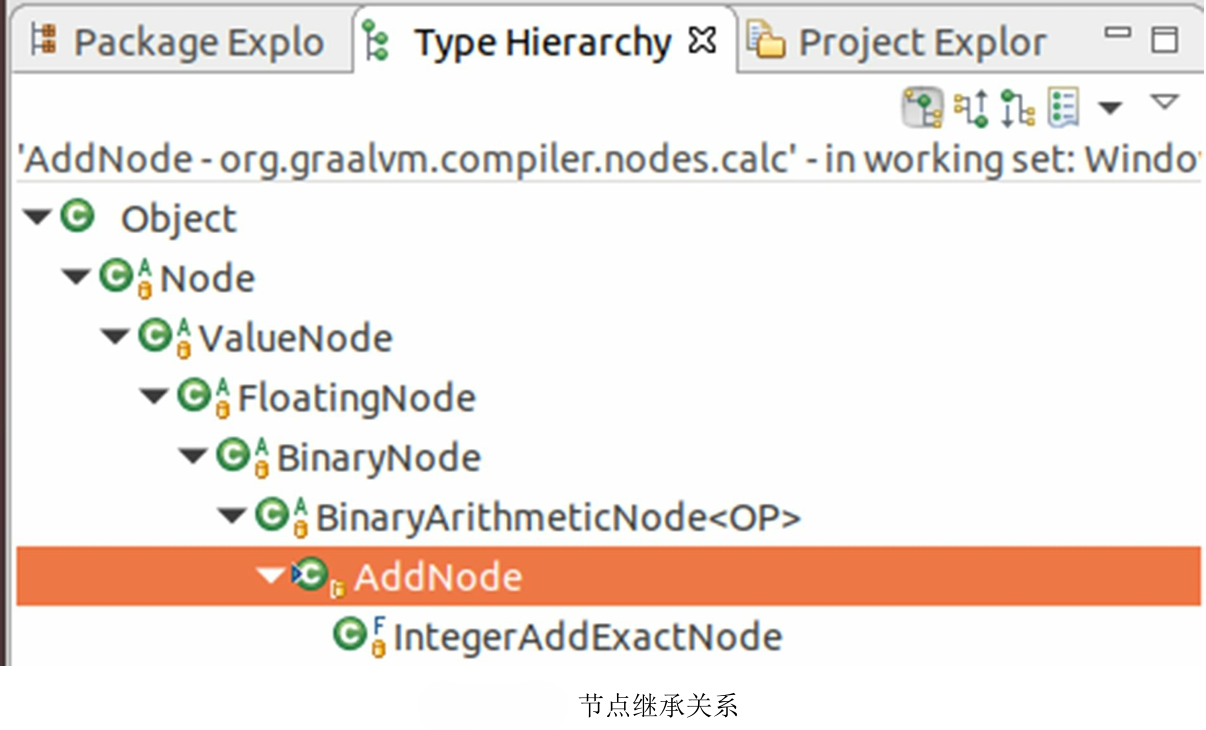
现在已经能够基本看明白Graal理想图的中间表示了，那对应到代码上，Graal编译器是如何从字节码生成理想图？又如何在理想图基础上进行代码优化的呢？这时候就充分体现出了Graal编译器在使用Java编写时对普通Java程序员来说具有的便捷性了，在Outline视图中找到创建理想图的方法是greateGraph()，可以从Call Hierarchy视图中轻易地找到从JVMCI的入口方法compileMethod()到greateGraph()之间的调用关系，如图。

greateGraph()方法的代码也很清晰，里面调用了StructuredGraph::Builder()构造器来创建理想图。这里要关注的关键点有两个：



第一是理想图本身的数据结构。它是一组不为空的节点的集合，它的节点都是用ValueNode的不同类型的子类节点来表示的。

仍然以x+y表达式为例，譬如其中的加法操作，就由AddNode节点来表示，从“节点继承关系图”所示的Type Hierarchy视图中可以清楚地看到加法操作是二元算术操作节点（BinaryArithmeticNode）的一种，而二元算术操作节点又是二元操作符（BinaryNode）的一种，以此类推直到所有操作符的共同父类ValueNode（表示可以返回数据的节点）。



第二就是如何从字节码转换到理想图。该过程被封装在BytecodeParser类中，这个解析器我们可以按照字节码解释器的思路去理解它。如果这真的是一个字节码解释器，执行一个整数加法操作，按照《Java虚拟机规范》所定义的iadd操作码的规则，应该从栈帧中出栈两个操作数，然后相加，再将结果入栈。而从BytecodeParser::genArithmeticOp()方法上我们可以看到，其实现与规则描述没有什么差异，如图所示。



其中，genIntegerAdd()方法中就只有一行代码，即调用AddNode节点的create()方法，将两个操作数作为参数传入，创建出AddNode节点，如下所示：

|  |
| --- |
| protected ValueNode genIntegerAdd(ValueNode x, ValueNode y) {  return AddNode.create(x, y, NodeView.DEFAULT);  } |

每一个理想图的节点都有两个共同的主要操作，一个是规范化（Canonicalisation），另一个是生成机器码（Generation）。生成机器码顾名思义，就不必解释了，规范化则是指如何缩减理想图的规模，也即在理想图的基础上优化代码所要采取的措施。这两个操作对应了编译器两项最根本的任务：代码优化与代码翻译。

AddNode节点的规范化是实现在canonical()方法中的，机器码生成则是实现在generate()方法中的，从AddNode的创建方法上可以看到，在节点创建时会调用canonical()方法尝试进行规范化缩减图的规模，如下所示：

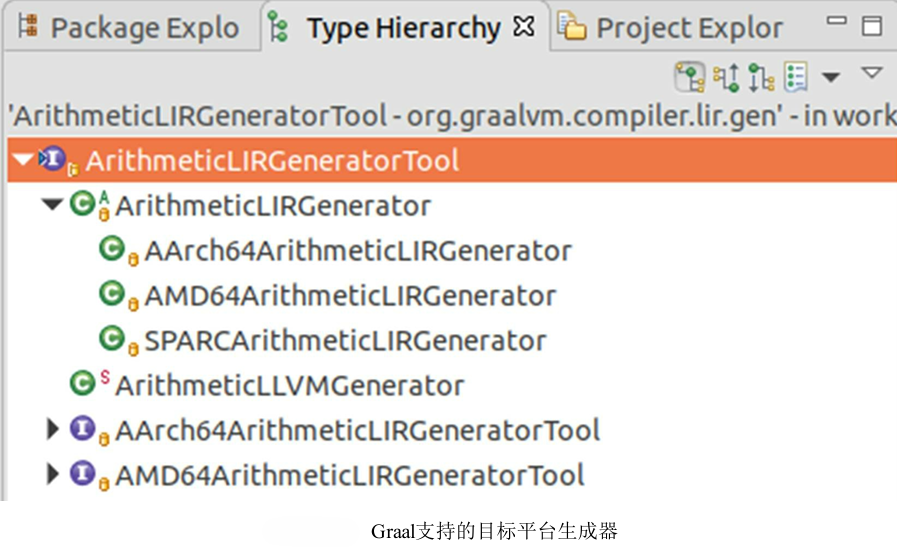
|  |
| --- |
| public static ValueNode create(ValueNode x, ValueNode y, NodeView view) {  BinaryOp<Add> op = ArithmeticOpTable.forStamp(x.stamp(view)).getAdd();  Stamp stamp = op.foldStamp(x.stamp(view), y.stamp(view));  ConstantNode tryConstantFold = tryConstantFold(op, x, y, stamp, view);  if (tryConstantFold != null) {  return tryConstantFold;  }  if (x.isConstant() && !y.isConstant()) {  return canonical(null, op, y, x, view);  } else {  return canonical(null, op, x, y, view);  }  } |

从AddNode的canonical()方法中我们可以看到为了缩减理想图的规模而做的相当多的努力，即使只是两个整数相加那么简单的操作，也尝试过了常量折叠（如果两个操作数都为常量，则直接返回一个常量节点）、算术聚合（聚合树的常量子节点，譬如将(a+1)+2聚合为a+3）、符号合并（聚合树的相反符号子节点，譬如将(a-b)+b或者b+(a-b)直接合并为a）等多种优化，canonical()方法的内容较多。

对理想图的规范化并不局限于单个操作码的局部范围之内，很多的优化都是要立足于全局来进行的，这类操作在CanonicalizerPhase类中完成。仍然以上一节的公共子表达式消除为例，这就是一个全局性的优化，实现在CanonicalizerPhase::tryGlobalValueNumbering()方法中，其逻辑看起来已经非常清晰了：如果理想图中发现了可以进行消除的算术子表达式，那就找出重复的节点，然后替换、删除。具体代码如下所示：

|  |
| --- |
| public boolean tryGlobalValueNumbering(Node node, NodeClass<?> nodeClass) {  if (nodeClass.valueNumberable()) {  Node newNode = node.graph().findDuplicate(node);  if (newNode != null) {  assert !(node instanceof FixedNode || newNode instanceof FixedNode);  node.replaceAtUsagesAndDelete(newNode);  COUNTER\_GLOBAL\_VALUE\_NUMBERING\_HITS.increment(debug);  debug.log("GVN applied and new node is %1s", newNode);  return true;  }  }  return false;  } |

至于代码生成，Graal并不是直接由理想图转换到机器码，而是和其他编译器一样，会先生成低级中间表示（LIR，与具体机器指令集相关的中间表示），然后再由HotSpot统一后端来产生机器码。譬如涉及算术运算加法的操作，就在ArithmeticLIRGeneratorTool接口的emitAdd()方法里完成。从低级中间表示的实现类上，我们可以看到Graal编译器能够支持的目标平台，目前它只提供了三种目标平台的指令集（SPARC、x86-AMD64、ARMv8-AArch64）的低级中间表示，所以现在Graal编译器也就只能支持这几种目标平台，如图所示。



为了验证代码阅读的成果，现在我们来对AddNode的代码生成做一些小改动，将原本生成加法汇编指令修改为生成减法汇编指令，即按如下方式修改AddNode::generate()方法：

|  |
| --- |
| class AddNode {  void generate(...) {  ... gen.emitSub(op1, op2, false) ... // 原来这个方法是emitAdd()  }  } |

然后在虚拟机运行参数中加上-XX:+PrintAssembly参数，因为从低级中间表示到真正机器码的转换是由HotSpot统一负责的，所以HSDIS插件仍然能发挥作用，帮助我们输出汇编代码。从输出的汇编中可以看到，在没有修改之前，AddNode节点输出的汇编代码如下所示：

|  |
| --- |
| 0x000000010f71cda0: nopl 0x0(%rax,%rax,1)  0x000000010f71cda5: add %edx,%esi ;\*iadd {reexecute=0 rethrow=0 return\_oop=0}  ; - Demo::workload@2 (line 10)  0x000000010f71cda7: mov %esi,%eax ;\*ireturn {reexecute=0 rethrow=0 return\_oop=0}  ; - Demo::workload@3 (line 10)  0x000000010f71cda9: test %eax,-0xcba8da9(%rip) # 0x0000000102b74006  ; {poll\_return}  0x000000010f71cdaf: vzeroupper  0x000000010f71cdb2: retq |

而被我们修改后，编译的结果已经变为：

|  |
| --- |
| 0x0000000107f451a0: nopl 0x0(%rax,%rax,1)  0x0000000107f451a5: sub %edx,%esi ;\*iadd {reexecute=0 rethrow=0 return\_oop=0}  ; - Demo::workload@2 (line 10)  0x0000000107f451a7: mov %esi,%eax ;\*ireturn {reexecute=0 rethrow=0 return\_oop=0}  ; - Demo::workload@3 (line 10)  0x0000000107f451a9: test %eax,-0x1db81a9(%rip) # 0x000000010618d006  ; {poll\_return}  0x0000000107f451af: vzeroupper  0x0000000107f451b2: retq |