http://hllvm.group.iteye.com/group/topic/39806

[讨论] JIT编译以及执行native code的流程

yuyinyang 2014-03-27

最近看了解释器执行bytecode的流程，想进一步了解JIT是如何工作的。但深入源码之后发现JIT实在是复杂太多太多，越看越迷惑，希望能得到撒加的指点。

大致来说我想弄明白两件事情：

1、JIT生成native code的流程是怎样的？

正如对于那些解释执行的方法，templateInterpreter通过如下的途径：

```
generate_method_entry() ->
        generate_normal_entry()->
        dispatch_next()
```

会为当前的method逐条生成所需bytecode的opcode。所以我想知道JIT生成native code（诸如emit_code()之类的）的流程是怎样的，C1和C2做法有何不同？

2、VM执行JIT编译好的native code的流程是怎样的？

我认为每次调用Java方法都是通过call_helper进去的（如有不对敬请指出），实际执行调用的代码：

C++代码

```
1.  void JavaCalls::call_helper(JavaValue* result, methodHandle* m, JavaCallArguments* args, TRAPS) {
2.    ...
3.    // do call
4.    ...
5.      StubRoutines::call_stub()(
6.        (address)&link,
7.        // (intptr_t*)&(result->_value), // see NOTE above (compiler problem)
8.        result_val_address,        // see NOTE above (compiler problem)
9.        result_type,
10.       method(),
11.       entry_point,
12.       args->parameters(),
13.       args->size_of_parameters(),
14.       CHECK
15.     );
16.   ...
17. }
```

我知道这是解释执行的调用入口，编译执行的入口是否也在这？

call_stub之前有这样一段与compiler有关的代码：

C++代码

```
1.  void JavaCalls::call_helper(...) {
2.    ...
3.    if (CompilationPolicy::must_be_compiled(method)) {
4.      CompileBroker::compile_method(method, InvocationEntryBci,
5.                    CompilationPolicy::policy()->initial_compile_level(),
6.                    methodHandle(), 0, "must_be_compiled", CHECK);
7.    }
8.    ...
9.  }
```

这段代码是否就是为java方法生成native code的？我试图在compile_method()方法内部打印出一些信息：

C++代码

```
1.  nmethod* CompileBroker::compile_method(...) {
2.    ...
3.    tty->print("Compile level is %d\n", comp_level);
4.    ...
5.  }
```

然后无论是 java -Xcomp -version或者是运行一个java程序（内有100000次的循环以确保触发JIT），上面的打印信息都不会输出，不知是何原因。

上面是我的一些理解，感觉自己的逻辑很混乱，求R大详解。

RednaxelaFX 2014-03-28
哈哈，欢迎掉进HotSpot VM的执行引擎的大坑。
楼主先读读这帖：http://hllvm.group.iteye.com/group/topic/37707 (java_main的汇编入口在哪里)
看是否回答了您的一些疑问，然后请继续把您还想进一步了解的问题提出来

yuyinyang 2014-03-28
我认真地看了您的那篇帖子，真的是受益颇多，不过我还是有一些疑惑希望能得到R大的指点.
1、您介绍了执行引擎是如何选择解释执行和编译执行的入口，以及JIT模式下如何安装已编译好的native代码

> **RednaxelaFX 写道**
>
> JIT编译的产物包装在nmethod对象里。编译完成后“安装”的逻辑在：
> // Install compiled code. Instantly it can execute.
> void methodOopDesc::set_code(methodHandle mh, nmethod *code) {
> ...
> }

那么JIT生成这段native code的逻辑在哪？(也就是那些诸如0x0F的x86 opcode是如何生成的)，C1和C2的这部分逻辑有何不同吗？
2、你提到当使用-Xcomp启动VM的时候会在call_helper()中调用compile_method()对当前方法进行编译，那么如果不是用-Xcomp方式启动，而是当某个热点方法的invocationCounter达到了设定的CompileThreshold而触发JIT的情况下，这个过程编译及执行本地代码的逻辑是怎样的？对应的hotspot源码在什么地方？
3、我尝试在-Xcomp模式下分别使用-client和-server启动VM，然后在CompilerBroker::compile_method()里打印compile_level，我发现无论是client还是server模式下所有方法的compile_level都是4（CompLevel_full_optimization = 4, // C2 or Shark）：

> **引用**
>
> The comp_level of method java.lang.Object.<clinit>()V is 4
> The comp_level of method java.lang.Object.registerNatives()V is 4
> The comp_level of method java.lang.String.<clinit>()V is 4
> The comp_level of method java.lang.String$CaseInsensitiveComparator.<init>(Ljava/lang/String$1;)V is 4
> The comp_level of method java.lang.String$CaseInsensitiveComparator.<init>()V is 4
> The comp_level of method java.lang.Object.<init>()V is 4
> ...

请问-client和-server是分别对应C1和C2吗？为什么对上面的compile_level没有影响呢？这两个参数的解析在哪，影响了JIT的哪些内容？

先问这些吧，感觉思路还是不太清晰。。R大见谅

RednaxelaFX 2014-03-28
先回答你最后一个问题。你打开 -XX:+TieredCompilation 就知道了。详细可以看我之前写的PrintCompilation文档：
https://gist.github.com/rednaxelafx/1165804#file-notes-md

yuyinyang 2014-03-29
好的，静待您对前两个问题的解答。
您那篇gist提到：

> **RednaxelaFX 写道**
> Starting from early versions of JDK 6, HotSpot incorporates a new mode of execution: the "tiered" mode. In this mode, the interpreter, the client compiler (C1) and the server compiler (C2) would work together in a single VM, to get the best of the world in all phases of execution: start up, warm, steady.
>
> Before tiered compilation, a HotSpot VM could either use C1 (the Client VM) or C2 (the Server VM) as the JIT compiler, but not both in a single VM. With tiered compilation, the HotSpot Server VM could use both compilers at the same time, for different compilation tasks.

也就是说在-client模式下同样会有CompLevel_full_optimization的方法，那么有没有什么开关可以指定只使用C1或者C2？

C++代码

```
1.  // Enumeration to distinguish tiers of compilation
2.  enum CompLevel {
3.    CompLevel_any               = -1,
4.    CompLevel_all               = -1,
5.    CompLevel_none              = 0,         // Interpreter
6.    CompLevel_simple            = 1,         // C1
7.    CompLevel_limited_profile   = 2,         // C1, invocation & backedge counters
8.    CompLevel_full_profile      = 3,         // C1, invocation & backedge counters + mdo
9.    CompLevel_full_optimization = 4,         // C2 or Shark
10.
11. #if defined(COMPILER2) || defined(SHARK)
12.   CompLevel_highest_tier      = CompLevel_full_optimization,  // pure C2 and tiered
13. #elif defined(COMPILER1)
14.   CompLevel_highest_tier      = CompLevel_simple,             // pure C1
15. #else
16.   CompLevel_highest_tier      = CompLevel_none,
17. #endif
18.
19. #if defined(TIERED)
20.   CompLevel_initial_compile   = CompLevel_full_profile        // tiered
21. #elif defined(COMPILER1)
22.   CompLevel_initial_compile   = CompLevel_simple              // pure C1
23. #elif defined(COMPILER2) || defined(SHARK)
24.   CompLevel_initial_compile   = CompLevel_full_optimization   // pure C2
25. #else
26.   CompLevel_initial_compile   = CompLevel_none
27. #endif
28. };
```

COMPILER1或COMPILER2是由-client和-server指定的吗？

RednaxelaFX 2014-03-31
关于JIT编译是如何触发的，楼主可以参考下我以前一个很长的演示稿：
http://www.valleytalk.org/2011/07/28/java-%E7%A8%8B%E5%BA%8F%E7%9A%84%E7%BC%96%E8%AF%91%EF%BC%8C%E5%8A%A0%E8%BD%BD-%E5%92%8C-%E6%89%A7%E8%A1%8C/
中间有提到JIT编译是如何触发的.那个描述在当前HotSpot VM不使用TieredCompilation的话还是适用的.

解释器触发JIT编译的时候是向CompileBroker（编译代理商？）发出请求的。接到CompileBroker::compile_method()请求后它会把编译任务插入队列（CompileQueue）里，然后编译器线程会逐个领取这些任务然后开始编译。

洗澡去...明儿再继续回复

---

yuyinyang 2014-03-31
我看过您这篇talk,但是这pdf版本里好多slides都没有图,很多十分想了解的地方就只有一个标题,实在有点闹心. 我也找过您这talk的视频,貌似是在一个人的博客里,但那网站已经挂了,R大能否把视频share一下啊？

RednaxelaFX 2014-04-07
之前就一直想花点功夫好好回复一下这帖，但是最近好难抽出时间来。感谢家人的支持总算让我在周末找到了点时间来码字

> **yuyinyang 写道**
> 我看过您这篇talk，但是这pdf版本里好多slides都没有图，很多十分想了解的地方就只有一个标题，实在有点闹心.
> 我也找过您这talk的视频，貌似是在一个人的博客里，但那网站已经挂了，R大能否把视频share一下啊？

抱歉让您闹心了

那段视频在毕玄老大的博客上，抱歉我也没有原本的视频文件了。
现在有效的视频链接为：
Part I
Part II
Part III

Part IV
Part V
Part VI
（在国内访问打不开的话请自备梯子）
不过不看那个视频也没啥损失，因为那是我第一次讲这组演示稿，当时还不够熟练而且内容还不够充分，而且当时的拍摄效果也不太好，所以不看也罢啦。

不知道楼主想知道的细节到底要到哪一层，现在总算有点时间，我再随便写点。

先总结一下前面我的回帖，然后选几点展开来讲。下面的讨论全部限定于Oracle/Sun HotSpot VM的具体实现。对OpenJDK同样有效。
下面引用代码的地方我放的链接都是当前JDK9代码仓库里的最新版。这跟下半年将要发布的JDK8u20基本上一致。

==========

解释与编译的混合模式执行引擎

先前说的另一帖我已经讲解了解释器的生成以及每个方法的from_interpreted_entry和from_compiled_entry两种入口。
在调用一个Java方法时，HotSpot VM的解释器总是从from_interpreted_entry进入一个方法，而已编译的方法则总是从from_compiled_entry进入。

当一个目标方法还没被JIT编译时，它的两个入口如下：
```
from_interpreted_entry ----------> 解释器的入口
                                  /
from_compiled_entry -> c2i adapter /
```

而当目标方法已经被JIT编译后，它的两个入口如下：
```
from_interpreted_entry -> i2c adapter \
                                       \
from_compiled_entry ------------------> 编译后的方法的入口
```

i2c与c2i adapter的作用就是在解释器与已编译的方法之间对接，让参数能传递到正确的位置上。解释器只用栈来传递参数，而已编译方法则主要用寄存器来传递参数，两者的差异得靠适配器来抹平。

楼主最初提到的JavaCalls::call_helper()只用于从Java代码之外调用Java方法，例如从JNI的Invocation API，或者是JVM自己要主动调用某些Java方法（如类的静态初始化器、invokedynamic的bootstrap方法之类）。
JavaCalls::call_helper()总是从Java方法的from_interpreted_entry进入的。当然，楼主也看到了那个入口有个检查看目标是否必须被编译的地方，它保证-Xcomp模式下所有Java在首次执行前都会先被编译，而编译后它仍然得从from_interpreted_entry通过i2c adapter去调用到被编译后的方法。
（不过要小心的是C2并不“喜欢”别人强迫它提前编译代码。看起来已经编译了的代码可能实际上只是变了个样子的解释器入口。）

然后在这个演示稿里我提到了HotSpot VM的混合模式执行引擎的构成方式：
每个Java方法都有两个计数器，一个是调用计数器（invocation counter），另一个是循环计数器（backedge counter）。两个计数器都是InvocationCounter类型的。在解释器解释执行方法的时候，每次经过方法的入口都会递增调用计数器，每次经过循环的向回跳转（backedge）都会递增循环计数器。要说明的是每个方法只有一个入口但可能有任意多个循环，而且循环可能嵌套，但循环计数器并不关心是哪个循环在往回跳，只要有往回跳的就递增该计数器。这俩计数器在读写时都没做同步，所以在多线程读写时其值并不准确，只是个大概值，但通常认为够用了。
每当递增任何一个计数器之后，解释器都会检查一下**两个计数器之和**，看有没有超过阈值，如果超过了阈值就会先检查有没有对应这个位置的已经编译好的版本，有就跳到编译好的版本，没则提交编译任务。如果是在方法入口处发现计数器之和超过了CompileThreshold，就会触发一个**标准编译任务**；如果是在循环回跳的地方发现超过了InterpreterBackwardBranchLimit，就会触发一个**OSR编译任务**。

这些阈值的计算方式可以参考源码：
http://hg.openjdk.java.net/jdk9/jdk9/hotspot/file/ee8a3f5fbe3d/src/share/vm/interpreter/invocationCounter.cpp#l136

回答楼主的问题：
关于HotSpot VM的模板解释器：

> **yuyinyang 写道**
> 正如对于那些解释执行的方法，templateInterpreter通过如下的途径：
> generate_method_entry() ->
> generate_normal_entry()->
> dispatch_next()
> 会为当前的method逐条生成所需bytecode的opcode。

楼主这个描述是错的。HotSpot VM的模板解释器在VM初始化阶段就整个生成好了，不会在解释执行Java方法的时候逐条字节码生成机器码去执行。
当一个Java方法没有被JIT编译的时候，它的两个入口直接或间接都指向模板解释器的方法入口处理函数，跳进去就开始解释执行了。

更直观的图解请参考我的演示稿的第176-182页。

> **yuyinyang 写道**
> 2、你提到当使用-Xcomp启动VM的时候会在call_helper()中调用compile_method()对当前方法进行编译，那么如果不是用-Xcomp方式启动，而是当某个热点方法的invocationCounter达到了设定的CompileThreshold而触发JIT的情况下，这个过程编译及执行本地代码的逻辑是怎样的？对应的hotspot源码在什么地方？

以HotSpot在x86-64上的实现为例，解释器在方法入口处检查计数器是否已达到阈值的逻辑在例如InterpreterGenerator::generate_normal_entry()，在循环回跳处检查计数器是否已达到阈值的逻辑在这里。
它们都会在计数器到达阈值时调用InterpreterRuntime::frequency_counter_overflow()来触发编译任务。它会进一步调用到CompilationPolicy::event()，这里会根据当前的编译模式选择合适的编译器去编译。

要看更直观的流程图请参考我的演示稿的第205页。

CompilationPolicy是选择哪个方法要用什么程度优化的抽象层。
在实现多层编译之前，它主要有两种实现，一个是SimpleCompPolicy，直接哪个方法触发编译就编译哪个方法；另一个是StackWalkCompPolicy，在触发编译的时候会爬一下调用栈，看看是不是向上几层找个caller来编译的受益更大，找最大受益的caller来编译。
在实现多层编译之后，CompilationPolicy的任务还加上了为当前要编译的方法选择一个合适的优化层次去编译。

==========

异步的后台编译（background compilation）

触发编译任务是异步的。HotSpot VM的JIT编译器跑在自己独立的线程上。Java线程在解释执行Java方法要触发JIT编译时，解释器调用编译代理商（CompileBroker）提交任务，后者生成任务后推到编译队列（CompileQueue）就返回了。然后解释器会继续解释执行当前方法；而编译器线程从编译队列拿到任务后会进行编译，在编译完成后把结果（nmethod）“安装”到方法上，这样下次解释器去检查有没有已编译好的版本时就会找到。

异步编译的行为可以通过启动参数-XX:-BackgroundCompilation禁用，变为同步编译。这样的话Java线程在提交编译任务后就会等着，一直等到编译结果被安装到目标Java方法上之后再跳进编译后的版本去执行。

要看更直观的流程图请参考我的演示稿的第206页。

异步编译是混合模式或多层编译的执行引擎里常见的做法，在不影响程序响应性的前提下为高优化层的JIT编译器争取到更多时间，于是就可以做更多更强但更耗时的优化。
除HotSpot VM之外的另外两个桌面/服务器端主流JVM，IBM J9 VM和Oracle JRockit VM也都支持后台编译。

现代JavaScript引擎也都用同一思路实现了异步/后台编译。IE9的Chakra从一开始就宣传它通过后台编译更好的利用多线程资源。最近V8也终于赶上趟了。

==========

Client VM 与 Server VM

在HotSpot VM有多层编译之前，HotSpot有两种VM：Client VM与Server VM。它们共享大部分核心运行时系统和GC的代码，但是JIT编译器不一样——Client VM里只包含C1（Client Compiler），而Server VM里只包含C2（Server Compiler）。
Client VM与Server VM是两个不同的库文件，可以看到在JRE里有client和server目录，里面各有一个libjvm.so（或libjvm.dylib / jvm.dll），它们分别是Client VM与Server VM的实体。
关于Client VM/Server VM与C1、C2的关系，我的演示稿的第103页有图示。

在构建Client VM与Server VM时，使用的编译参数略有不同。其中Cient VM有定义COMPILER1宏，而Server VM有定义COMPILER2宏。

回答楼主的问题：

> **yuyinyang 写道**
> COMPILER1或COMPILER2是由-client和-server指定的吗？

大家平时用java -client或java -server实际上是向“java”这个启动程序（launcher）指定要加载的JVM的名字，让它去找到对应的JVM动态链接库。这是个运行时选项。
关于java launcher，我的演示稿的第109页有提到。

而源码里的COMPILER1和COMPILER2宏，如前面所述，是构建JVM的编译时选项。

> **yuyinyang 写道**
> 请问-client和-server是分别对应C1和C2吗？为什么对上面的compile_level没有影响呢？这两个参数的解析在哪，影响了JIT的哪些内容？

-client / -server参数的解析不在HotSpot VM，而在java launcher里，CheckJvmType()，进一步调用KnownVMIndex()去查找jvm.cfg里配置的已知JVM：
http://hg.openjdk.java.net/jdk9/jdk9/jdk/file/6f87125d6e50/src/share/bin/java.c#l489
client和server都是jvm.cfg里有配置的。之前HLLVM群组的一帖也有对此的讨论。

> **yuyinyang 写道**
> 3、我尝试在-Xcomp模式下分别使用-client和-server启动VM，然后在CompilerBroker::compile_method()里打印compile_level，我发现无论是client还是server模式下所有方法的compile_level都是4（CompLevel_full_optimization = 4, // C2 or Shark）：
>
> > **引用**
> > The comp_level of method java.lang.Object.<clinit>()V is 4
> > The comp_level of method java.lang.Object.registerNatives()V is 4
> > The comp_level of method java.lang.String.<clinit>()V is 4
> > The comp_level of method java.lang.String$CaseInsensitiveComparator.<init>(Ljava/lang/String$1;)V is 4
> > The comp_level of method java.lang.String$CaseInsensitiveComparator.<init>()V is 4
> > The comp_level of method java.lang.Object.<init>()V is 4
> > ...

这是因为Oracle/Sun JDK在64位上只有Server VM，没有Client VM。
不知道楼主是在什么平台上测试的32还是64位的JDK。如果楼主在64位平台上运行64位Oracle/Sun JDK，里面就只有Server VM。就算用-client想指定用Client VM，实际上还是会用到Server VM。
Oracle/Sun JDK6与JDK7的Server VM默认不启用多层编译模式。而comp_level只在多层编译模式才有用，否则总是使用当前VM的CompLevel_highest_tier，在Client VM里它是1即CompLevel_simple，在Server VM里它是4即CompLevel_full_optimization。

题外话，HotSpot VM并不是只有上述的Client VM、Server VM这两种形式，还有一些其它变种。
例如说在iOS上运行的Oracle HotSpot VM就只有解释器而没有任何JIT编译器。Java SE Embedded里的HotSpot VM通常也只有C1而没有C2，但它们又跟桌面版的HotSpot Client VM不一样。

==========

多层编译系统（tiered compilation system）

这帖我提到JDK6u25开始包含新的多层编译系统，可以用-XX:+TieredCompilation打开。
这篇文档里我讲解了PrintCompilation日志的格式，里面顺带提到了多层编译模式的概念。

在多层编译模式下,HotSpot Server VM(此时应该叫做HotSpot Tiered VM了)会同时用上解释器、C1和C2：
Tier 0：CompLevel_none，解释器
Tier 1：CompLevel_simple，C1的正常编译（没有任何profiling）。这跟在Client VM里C1的工作模式**几乎**一样。进入该层的方法无法继续升级到更高层，除非先逆优化到解释器里。
Tier 2：CompLevel_limited_profile，C1带基本profiling的编译（有方法调用和循环计数器，跟解释器用的计数器的位置和作用一样）。可以升级到更高层。
Tier 3：CompLevel_full_profile，C1带所有profiling（包括空指针检查、条件分支、虚方法调用、类型检查等。最近还新加了参数类型和返回值类型profiling），同时禁用少量优化。可以升级到更高层。
Tier 4：CompLevel_full_optimization，C2编译。

在这种执行模型中，最常见的执行过程是：
0 -> 3 -> 4
或
0 -> 2 -> 3 -> 4
也就是说方法一开始由解释器执行；
达到第一个编译阈值的时候由C1编译带有基本profiling的版本，这种编译得到的代码比不带profiling的慢一点，但它的计数器能触发进一步的编译；
达到第二个编译阈值时再由C1编译，这次是带有完整profiling的版本。这个版本比第2层更慢，但是它能收集非常详细的profile，有助后续编译做精准的优化；
最后是由C2编译的优化版本。它会利用前面第3层的C1编译收集到的profile做激进的优化。

C2编译了之后并不是一定就一直保持在这个版本上了。C2的激进可能在它编译的时候还正确，但到后来运行一段时间可能就变得不正确了例如没执行过的分支可能执行到了，之前没遇到空指针的地方可能遇到了空指针，或者新加载的类可能让原本只有一个版本的实现的方法变成有多个版本。这些新状况都需要抛弃C2激进优化的代码而回到低层的版本，通常是回到第0层的解释器。这叫做逆优化（deoptimization）。
所以会有：
4 -> 0

如果一个方法被逆优化太多次，或者可能C2无法编译它(例如方法含有完全空的循环，**while** (**true**) { })，那么多层编译系统可能会选择只用C1来编译不带profiling的版本，这样就会有：
0 -> 1

一个方法被第1层C1编译后通常就会一直用那一层了。它没有profiling用的计数器所以不会触发新的高优化层的编译。唯一跳出它的情况就是新的类加载使它也需要逆优化。

更新：在豆瓣笔记里也写了一段关于多层编译系统的笔记：http://book.douban.com/annotation/31392220/

要更详细的了解HotSpot VM现在实现的多层编译系统，可以参考它的作者Igor Veresov做的演示稿。

Sun在JDK 6给HotSpot添加了多层编译模式之后，Client VM还是跟以前一样，而名为Server VM的那个实际上变成了Tiered VM：它同时包含C1与C2，而且它所包含的CompilationPolicy知道如何分派编译任务给这俩编译器。

所以，现在的HotSpot Server VM跟大家以前印象中的不一样了，并不只含有C2，而是两个JIT编译器都有。
在JDK8之前，Server VM默认还是跟以前一样，只用解释器+C2的混合模式执行引擎来执行Java代码。而显式指定-XX:+TieredCompilation来启动的话，它就变成Tiered VM，使用解释器+C1+C2的多层编译模式。
从JDK8开始，多层编译模式变成Server VM的默认执行模式了。

要在HotSpot Server VM上禁用C1只用C2的话很简单，只要指定-XX:-TieredCompilation即可；
而要禁用C2只用C1的话就稍微麻烦点，要指定-XX:+TieredCompilation -XX:TieredStopAtLevel=1。注意这个参数要在Oracle JDK6u25或以上的版本才可以用。

继续回答楼主的问题：

> **yuyinyang 写道**
> 您那篇gist提到：
>
> > **RednaxelaFX 写道**
> > Starting from early versions of JDK 6, HotSpot incorporates a new mode of execution: the "tiered" mode. In this mode, the interpreter, the client compiler (C1) and the server compiler (C2) would work together in a single VM, to get the best of the world in all phases of execution: start up, warm, steady.

> Before tiered compilation, a HotSpot VM could either use C1 (the Client VM) or C2 (the Server VM) as the JIT compiler, but not both in a single VM. With tiered compilation, the HotSpot Server VM could use both compilers at the same time, for different compilation tasks.

> 也就是说在-client模式下同样会有CompLevel_full_optimization的方法，那么有没有什么开关可以指定只使用C1或者C2？

java -client选择了Client VM，它只包含解释器与C1的混合模式执行引擎，没有C2，也没有多层编译系统。
在Client VM里，CompLevel_highest_tier为CompLevel_simple，也就是第1层的C1。
而在带有多层编译系统的Server VM里，CompLevel_highest_tier为CompLevel_full_optimization，也就是第4层的C2。也可以通过-XX:TieredStopAtLevel=<n>来限制多层编译系统多高可以用到哪一层。

其实从JDK6的早期开发开始，Sun就在研究给HotSpot VM添加多层编译系统，由Steve Goldman负责开发。当时的开发状况可以通过这个JavaOne 2006的演示稿一窥端倪。
（本来他还有几篇写得不错的博文描述当时的开发状况的，可惜他的博客从Sun迁移到Oracle之后被删除了。可恶。）
不幸的是Steve在2008年因病去世，这个系统的研发也就耽搁了下来，直到后来Igor接下任务重新实现了一个多层编译系统，也就是我们现在看到的这个。

再题外话一点：HotSpot VM并不总是代表着JVM技术的最先进程度，至少在JIT编译器这块的某些部分不是。多层编译系统就是这样一例。
IBM自从JDK5改用J9 VM作为其Java SE的默认VM，就在J9的JIT编译器“Testarossa”里实现了多层编译。这是个颇为可配置的灵巧的编译器，单个编译器就可以配置在多种优化程度上使用。
Sun自家的Exact VM也在很早就实现了解释器+初级编译器（baseline compiler）+优化编译器的多层编译系统。可惜这个VM在跟HotSpot VM的内部竞争中失败了，后来没有发布到生产环境（只在Sun JDK1.2.2里发布了一个版本，但发布的版本包含的功能没有Sun Labs做的许多先进研究）。

Oracle HotSpot开发组里其实有一种声音，觉得维护两个JIT编译器的成本太高了，什么优化都在分开实现，测试也得多测一份。如果能像IBM J9/TR那样只用一个编译器就实现多种优化程度就好了。
而且C1用在多层编译系统里总觉得稍微勉强了点。就像这篇文章所述，Mozilla SpiderMonkey早先有过一个多层编译系统，是解释器+JaegerMonkey+IonMonkey，其中JaegerMonkey用作初级编译器，而IonMonkey是优化编译器。问题是JaegerMonkey在研发之初并不是打算在多层编译系统里只充当初级编译器角色的，而是在TraceMonkey无法优化的地方替补上去作优化编译。这就使得它用作初级编译器时显得做了太多激进优化，不太合适。C1的初衷是为Client VM服务，做快速有效的优化编译；把它用作初级编译器跟JaegerMonkey有着类似的困境。
另外，C2虽然做了很多优化，但编译速度还是有点太慢了。而且即便是HotSpot VM开发组里觉得C2太复杂不好维护的人也有许多。
所以有可能会在未来开发C1、C2之外的另一个新JIT编译器来一统HotSpot VM的编译需求。

==========

JIT编译器

CompilationPolicy到CompileBroker，然后选择合适的编译器，最后会调用到AbstractCompiler::compile_method()。
C1与C2各自有实现AbstractCompiler接口，分别是C1的Compiler与C2的C2Compiler。

> **yuyinyang 写道**
> 1、您介绍了执行引擎是如何选择解释执行和编译执行的入口，以及JIT模式下如何安装已编译好的native代码
> **RednaxelaFX 写道**
>
> JIT编译的产物包装在nmethod对象里。编译完成后“安装”的逻辑在：
> // Install compiled code. Instantly it can execute.
> void methodOopDesc::set_code(methodHandle mh, nmethod *code) {
> ...
> }
>
> 那么JIT生成这段native code的逻辑在哪？(也就是那些诸如0x0F的x86 opcode是如何生成的)，C1和C2的这部分逻辑有何不同吗？

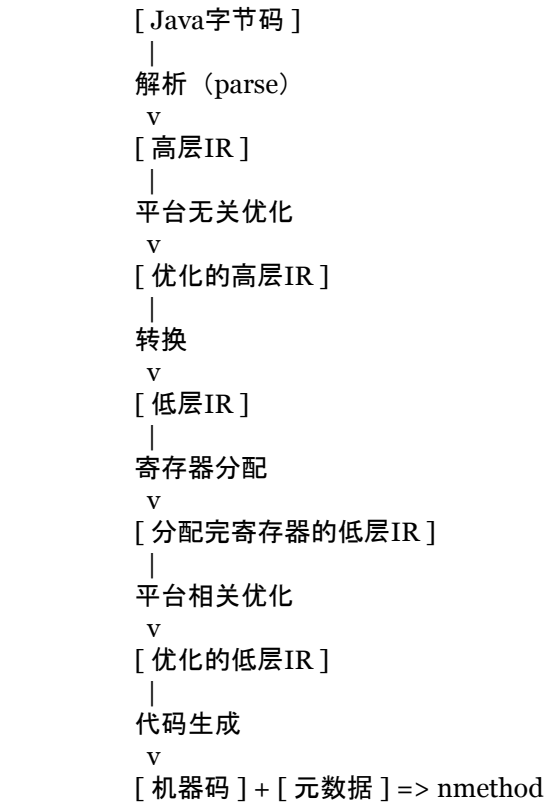
上面提到了，C1与C2编译Java方法的入口就在它们对AbstractCompiler::compile_method()的实现那里。后面具体的编译动作就各有各不同了。

我以前写过一篇关于HotSpot VM源码目录结构的笔记，C1和C2的源码分别在c1与opto目录中。它们共享的源码只有编译器跟运行时系统打交道的抽象层ci（Compiler Interface），真正做编译器本职工作的逻辑完全没有共享代码。

C1与C2是HotSpot VM自带的两个JIT编译器。要理解它们需要事先充分掌握编译原理的知识，特别是编译器后端的知识。如果这部分基础还不扎实的话，请先读读相关书籍、动手写点代码做做实验，再回来看HotSpot的JIT编译器的源码会轻松许多。JIT编译器相对于传统静态编译器因为资源更受限，不能用太复杂太耗时的优化，所以通常也更简单好懂。
这俩JIT编译器相当于传统编译器的后端；相应的，javac或ecj之类的Java源码层编译器相当于传统编译器的前端，而连接前后端的IR就是Java字节码/Class文件。

C1的取舍倾向使用开销小、见效快的优化，因而很少使用需要迭代收敛的算法。
而C2则倾向于更彻底的优化，用了许多迭代算法。
现在的C1和C2其实已经比刚开始的时候要相似得多，主要是C1比以前先进多了。

从抽象概念说，这俩JIT编译器的任务一样，都是拿到Java字节码，转换成自己的中间表现形式（intermediate representation，IR），做一些优化，然后生成机器码与相关元数据（metadata），打包成nmethod对象安装到目标方法上。



C1与C2的编译都要走过这样的流程。细节上有许多共通点，也有差异相当大的地方。

C1与C2的IR都是图形式的IR，而不是线性形式或者树形式的。
IR的形式极大的影响编译器的各种优化实现的难度和开销，是编译器很重要的方面。

C1的高层IR名为HIR（High-level Intermediate Representation），是一种比较传统的IR，有基本块构成的控制流图（control-flow graph，CFG）和基本块内的SSA形式的数据依赖图。
HIR的控制流图是双向链接的，也就是说每个基本块都有指向前驱节点（predecessor）和指向后继节点（successor）的指针。
而HIR的数据依赖图则是单向链接的，只有use-def链，而不显式维护def-use链。每个数据节点持有指向它的参数的指针，而不知道它自己的值被什么节点使用。这使得C1很容易做前向数据流分析（forward dataflow analysis），而不那么方便做后向数据流分析（backward dataflow analysis）。它需要用一个单独的趟来得到def-use信息。

C2的高层IR名为Ideal Graph，是一种比较少见的sea-of-nodes形式的IR，属于PDG（program dependence graph，程序依赖图），在同一层IR里显式记述了控制流、数据流与内存副作用依赖，而没有显式的基本块结构。其数据流的部分也是SSA形式的。
Ideal Graph的所有依赖都通过显式的双向链接来维护，无论是前向还是后向分析做起来都很方便。而且，由于没有显式的基本块结构，它在优化过程中不维护代码调度顺序（schedule），所以使许多原本只适用在局部（基本块内）的简单优化变得可以在全局（跨基本块）适用，使C2能维持相对简单的结构来达到更好的优化效果。

C1与C2都在解析（parse）字节码的时候通过抽象解释把Java字节码转换为SSA形式的IR。转换成SSA形式的过程隐含了复写传播（copy propagation）优化。
两者都在解析字节码的过程中做方法内联（method inlining）以及值标号（value numbering）优化。

C1在方法内联上只用了比较简单的策略：它能内联可以静态判定实现者的、字节码大小不大于MaxInlineSize（= 35字节）的方法。这包括静态方法、private的实例方法以及final的虚方法。同时它也可以依赖类层次分析（class hierarchy analysis，CHA）来内联只有单一实现者的虚方法。
C2在方法内联上则更为激进：除了C1能内联的之外，C2还能使用profile信息来内联无法静态判定实现者的方法——假如profiling发现某个虚方法调用点实际只调用到了1、2个实现者，那么C2就可以把这1、2个实现者给内联进来。

对于无法内联的虚方法调用，C1和C2都会生成单态内联方法调用缓存（monomorphic inline-cache）来加速虚方法调用。代码里是CompiledIC。

值标号主要用于消除冗余表达式的计算，并常跟表达式简化（algebraic simplification）和常量折叠（constant folding）并用。
在这方面，C1在解析字节码时只做局部值标号（local value numbering，LVN），也就是只在基本块内做值标号。
而C2此时做的是全局值标号（global value numbering，GVN），也就是在方法内可以跨基本块边界做值标号——Ideal Graph根本就没有基本块边界，值标号很自然就是全局的。

另外值得一提的是，C1解析字节码得到的IR包含整个方法的逻辑，而C2则可能只包含方法的一部分逻辑——没执行过的或者很少执行的分支、涉及尚未加载或已卸载的类的地方、许多抛异常的地方等等，这些部分C2假定不会发生所以都不编译；一旦在运行时真的执行到了这些地方，代码就会通过逆优化（deoptimization）回到解释器去继续执行。这样可以让C2的IR更小、类型更精准，以便容忍后续的更耗时的优化。

接下来是得到高层IR后的平台无关优化。

C1做的优化还是比较简单，主要有这么一些：
* 消除冗余的空指针检查（null check elimination）
* 消除条件表达式（conditional expression elimination，CEE）
* 合并基本块
* 基于必经节点的全局值标号（dominator-based GVN）
最新版的HotSpot C1新加了
* 消除数组边界检查（range check elimination，RCE）
* 比较简单的循环不变量外提（loop invariant code motion，LICM）。

C2则是做迭代式优化。这包括：
* 迭代式全局值标号（iterative GVN）
* 条件常量传播（conditional constant propagation，CCP）
* 循环优化（Ideal Loop），包括消除数组边界检查（RCE）、循环不变量外提（LICM）、循环展开（loop unrolling）、循环剥离（loop peeling）、基于superword的循环再合并、循环向量化，等等。
* 逃逸分析（escape analysis）与标量替换（scalar replacement）、锁消除（lock elision）搭配使用
* Java代码模式优化，例如字符串拼接优化（string concatenation optimization）、消除自动装箱（autoboxing elimination）等
* 增量式方法内联（incremental inlining）
大致数了一下，不保证全部优化都列举出来了。

然后要从高层IR转换到低层IR。
C1与C2的低层IR最重要的功能都是支持寄存器分配，所以它们都显式刻画所有对运算对寄存器的使用。

C1的低层IR叫LIR（Low-level Intermediate Representation）。它虽然叫做低层IR，但语义相对其它优化编译器的LIR来说还是比较高级（也就是说比较抽象，没那么贴近目标机器指令），只是能够显式刻画寄存器的使用而已。HIR与LIR之间有固定的对应关系。LIR不是SSA形式的，在从HIR转换过来的时候需要退出SSA形式。

C2的低层IR叫Mach Node Graph。它非常贴近目标机器指令，几乎能一对一的直接映射到机器码上。从Ideal Graph转换到Mach Node Graph需要做指令选择（instruction selection），还有在代码调度（scheduling）之后重新创建带有显式基本块结构的控制流图。
C2的指令选择通过树改写的方式实现，这种系统叫做bottom-up rewrite system，BURS。

再下来终于到寄存器分配了。

C1使用几乎线性时间开销的线性扫描寄存器分配（linear scan register allocation，LSRA）。它具体使用的变种近似与原始LSRA和后来的second chance binpacking的混合形态。

C2则使用了更耗时的、比较传统的Chaitin-Briggs式图着色寄存器分配（graph-coloring register allocation）。
C2的寄存器分配非常慢，大概能占掉C2整个编译时间的一半…

在寄存器分配后，C1与C2都会做一趟窥孔优化（peephole optimization）来小范围的优化代码序列。这算是平台相关优化的一部分。

最后就到代码生成（code generation）了。这里C1和C2都是像套用模板似的把低层IR映射为机器码。
C1的LIR跟目标机器码还有一定距离，所以不一定能一对一映射过去。简单的运算，如算数运算，通常是一对一映射过去的；而一些比较复杂的操作，例如分配新对象、类型检查之类的则有大块的机器码模板。
C2的Mach Node跟目标机器码已经非常接近了，大部分能直接映射为机器码；有少量需要大块机器码模板的情况。

另外，C1与C2都要在平台相关优化完成后生成元数据，以便支持异常处理、GC与逆优化等的需求。这方面可以参考我之前写的一篇笔记。

到此算是很概括的过了一遍C1与C2的编译过程。
感觉如何？上面列举的名词大多是编译原理的后端部分的常见知识点，请让我重复一次，如果想要深入了解但缺乏基础知识的话还是先补充一下的好。

==========

HotSpot Client Compiler（C1）

我还没发过啥现成的博文或者帖子写C1的细节的。找时间再写写。

前面提到的优化多数都是课本上有的常规优化，但也有少量是只在Java上才有意义的。

C1做的“消除条件表达式（CEE）”就是这种特定优化的例子。它非常简单，作用是把Java源码里的三元条件表达式在HIR里直接表现出来。

您可能会觉得：啥？不就是条件表达式嘛？
问题就是Java字节码并不能直接表达这种表达式，而要绕点弯，使得

Java代码

```
x > y ? x : y
```

在字节码里看起来跟

Java代码

```
int temp;
if (x > y) {
  temp = x;
} else {
  temp = y;
}
```

几乎一样。实际字节码大致会是：

Java bytecode代码

```
1.  iload x
2.  iload y
3.  if_icmple Label_false_branch
4. Label_true_branch:
5.  iload x
6.  goto Label_done
7. Label_false_branch:
8.  iload y
9.  # fallthrough
10. Label_done:
11.  # ...
```

类似这样。挺繁琐的。
CEE要做的事情就是字节码里这种比较繁琐的代码序列恢复为跟Java源码类似的形式，最终让一些Java三元条件表达式生成出cmp + cmov的指令序列。

这种优化就是属于给Java字节码的设计“缺陷”来擦屁股的情况……

==========

HotSpot Server Compiler（C2）

HLLVM群组之前有一帖讨论C2的，请参考。

==========

今天就先写到这儿。欢迎楼主和其他对此感兴趣的同学继续提问

---

yuyinyang 2014-04-11
感谢R大如此详细的回复。关于C1C2的问题您已经解释的很清楚了，我还有一个小问题：

> **RednaxelaFX 写道**
> 这是因为Oracle/Sun JDK在64位上只有Server VM，没有Client VM。
> 不知道楼主是在什么平台上测试的32还是64位的JDK。如果楼主在64位平台上运行64位Oracle/Sun JDK，里面就只有Server VM。就算用-client想指定用Client VM，实际上还是会用到Server VM。

您之后又提到：

> **RednaxelaFX 写道**
> 要在HotSpot Server VM上禁用C1只用C2的话很简单，只要指定-XX:-TieredCompilation即可；
> 而要禁用C2只用C1的话就稍微麻烦点，要指定-XX:+TieredCompilation -XX:TieredStopAtLevel=1。

也就是说其实是可以通过使用分层编译并指定TieredStopAtLevel的方法在64位平台上使用client VM(C1)的，不知道我这个理解是否正确？

我看了您推荐的那篇C2的帖子，读了Thomas Würthinger硕士论文讲解server的部分，同时还有几篇Cliff关于ideal graph的paper，但感觉还是有不少疑问，其中我对编译过程中的code_gen比较关心，您在那篇帖子里主要是介绍parse阶段，能否对code_gen阶段也展开讲解一下呢？

RednaxelaFX 2014-04-12

> **yuyinyang 写道**
> 也就是说其实是可以通过使用分层编译并指定TieredStopAtLevel的方法在64位平台上使用client VM(C1)的，不知道我这个理解是否正确？

不对。是可以用分层编译和TieredStopAtLevel来指定只使用C1，但它仍然跟Cloent VM不一样。Client VM与Server VM之间最大的差异是JIT编译器，但其它方面也有细微差异，例如说ergonomics不一样…

> **yuyinyang 写道**
> 我看了您推荐的那篇C2的帖子，读了Thomas Würthinger硕士论文讲解server的部分，同时还有几篇Cliff关于ideal graph的paper，但感觉还是有不少疑问，其中我对编译过程中的code_gen比较关心，您在那篇帖子里主要是介绍parse阶段，能否对code_gen阶段也展开讲解一下呢？

如果您有具体问题的话可以到那帖去讨论。
但是先确定一点：您说的code_gen()跟我前面的介绍里的“代码生成”阶段对应吗？
如果是的话那其实没啥好说的，太简单了...就是模版套上去

yuyinyang 2014-04-13
额，怎么说呢。。
Thomas那篇硕士论文的3.5节提到：
> **Thomas Würthinger 写道**
> The server compiler selects subtrees out of the ideal graph and converts them one by one. It selects specific nodes as root nodes and transforms their related tree using tree selection rules.

这个阶段是否就是ideal graph node转换成MachNode的阶段，对应的源码在什么地方？还有论文以及代码里都提到的一个“matcher”具体是如何工作的？
另外您说的代码生成阶段套模板是指根据ad文件中描述的规则生成指令吗？

RednaxelaFX 2014-04-14
> **yuyinyang 写道**
> Thomas那篇硕士论文的3.5节提到：
> > **Thomas Würthinger 写道**
> > The server compiler selects subtrees out of the ideal graph and converts them one by one. It selects specific nodes as root nodes and transforms their related tree using tree selection rules.
>
> 这个阶段是否就是ideal graph node转换成MachNode的阶段，

对的。这段是instruction selection，在C2里通过一个bottom-up rewrite system（BURS）来实现。 在之前我提到的C2讨论帖里也有讲到BURS，一些相关资料啥的。请参考那边。

> **yuyinyang 写道**
> 对应的源码在什么地方？

对应的源码，平台无关的部分在hotspot/src/share/vm/adlc的整个目录，和hotspot/src/share/vm/opto/matcher.[hpp|cpp]。平台相关的部分在hotspot/src/cpu/<arch>/vm/<arch>.ad文件里。
ad文件里很重要的部分就是指定树的匹配模式及其对应的MachNode节点。
在构建HotSpot VM时，会先编译adlc（Architecture Description Language Compiler），然后执行adlc把ad文件编译成一个DFA，作为平台相关部分的Matcher的实现，以及MachNode的类型声明啥的。

Matcher是从C2的Compile::Code_Gen()创建Matcher实例并调用matcher.match()来进入的。

> **yuyinyang 写道**
> 还有论文以及代码里都提到的一个“matcher”具体是如何工作的？

Matcher就是C2的BURS的核心。参照上面的描述，先读读相关资料了解一下它想做的是啥再回来看C2的实现。
> **yuyinyang 写道**
> 另外您说的代码生成阶段套模板是指根据ad文件中描述的规则生成指令吗？

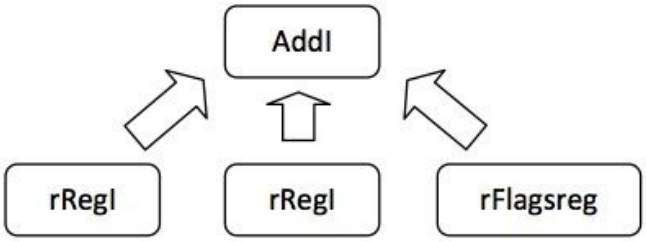
我说的是狭义的code generation，是最后从MachNode生成出机器码的地方，对应C2的代码是Compile::Output()。C2的MachNode跟机器码几乎一一对应，所以这里要做的就是把对应的机器码输出出来而已。

“Code generator”在编译器里有很多种用法，有广义有狭义。最广义的情况下，整个编译器后端都可以看作是“code generator”。最狭义的情况下，只有把最低层的IR映射到机器码的部分叫做“code generator”。所以一定得指定清楚说的是什么范围才回答得了。

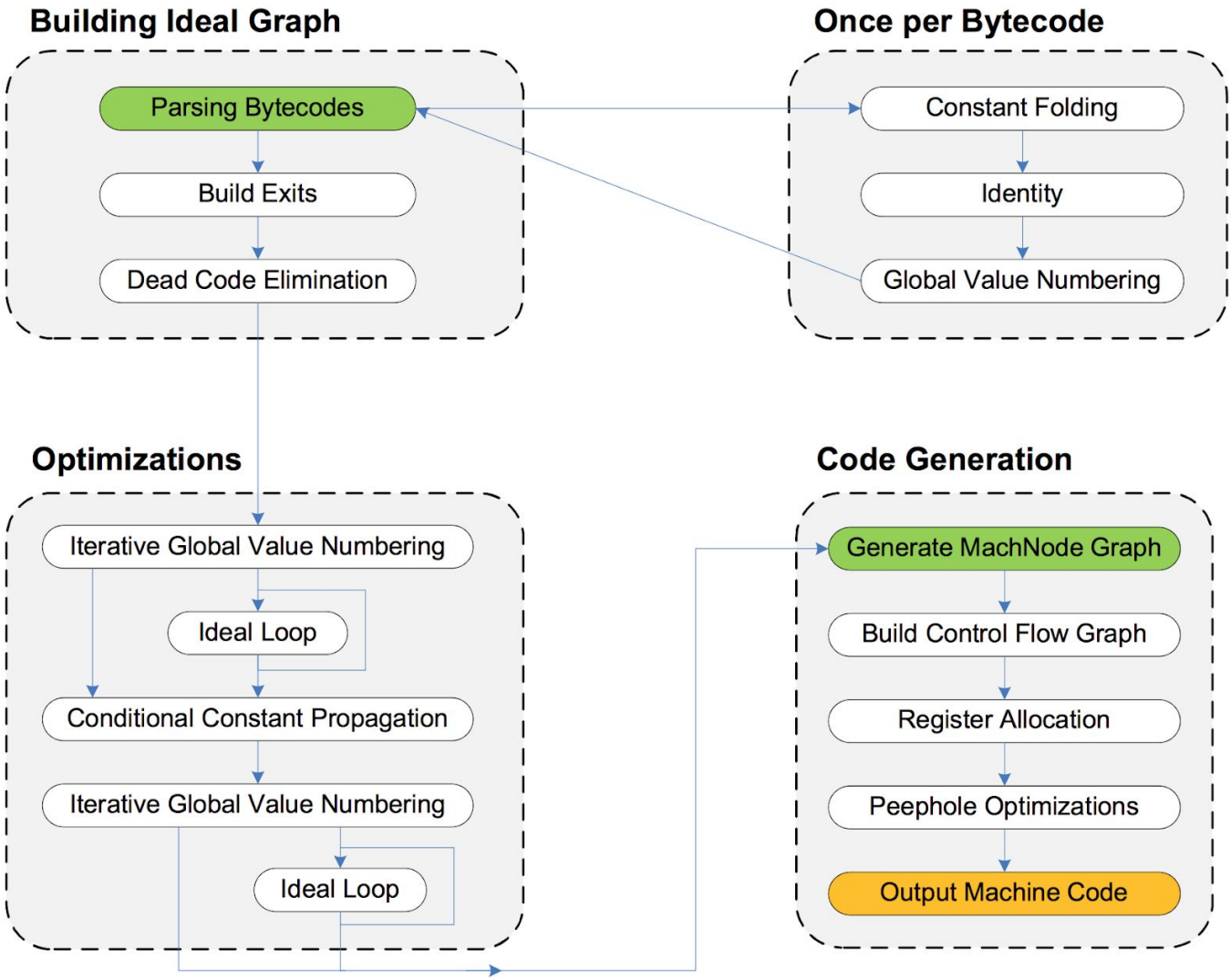
就楼主问的问题看，想问的应该是指C2的Compile::Code_Gen()所覆盖的范围？那个已经覆盖了C2的整个后端（所有平台相关的优化啥的都在里面）。

yuyinyang 2014-04-15
R神，您推荐那篇A Tutorial on Adding New Instructions to the Oracle® Java HotSpot TM Virtual Machine里有这样一段介绍编译过程的：

When compiling a Java method, HotSpot first parses the bytecodes and creates an intermediate representation (IR), which represents the instructions in the method in the form of an abstract syntax tree. The nodes in the tree represent operations (e.g., addition, multiplication, cmove) and their children represent the inputs to those operations. For example, we can visually represent the sub-tree corresponding to a basic "reg-to-reg" form of the integer addition operation:



不是先parse成ideal graph的吗，这里怎么又说是abstract syntax tree了。。而且根据后面内容讲的貌似就是根据这些tree去匹配ad文件里定义的规则然后选择指令，所以这个tree就是output()之前的最后形态了吗？。。那么它是下面这张图里哪个阶段的输出呢？总是感觉不能把Ideal Graph Node，MachNode，Subtrees，CFG与compile的各个phase确切地对应起来。。



RednaxelaFX 2014-04-16

呵呵，AMD那篇不能太追究其术语的准确性...它只有后面讲如何添加新指令的部分比较实用。

Java字节码parse之后就是Ideal Graph。这种图不是树，而是普通的有向图：它代表控制流的节点构成的图是可以有环的有向图；而其代表数据依赖的节点如果不看Phi的话是有向无环图（DAG），算上Phi节点就有可能有环了。

Thomas的Ideal Graph Visualizer论文里有提到Ideal Graph在做instruction selection之前会先变成树，因为C2的Matcher只能匹配树而不能匹配DAG。这就意味着某些节点可能会被复制出多份。多半是因为这个原因，AMD那篇文档只关心instruction selection部分，只看到了树型的输入，所以就认为输入是抽象语法树了。
这个步骤可以算在图中Geneate MachNode Graph（instruction selection）里面。但要注意它是发生在实际的instruction selection之前，算是必要的准备步骤。
具体在代码里，instruction selection的整个动作就在Matcher::match()里。其中这段代码就是把Ideal Graph变成概念上的树的形式的：

C++代码

```
1.  // ----------------
2.  // Collect roots of matcher trees.  Every node for which
3.  // _shared[_idx] is cleared is guaranteed to not be shared, and thus
4.  // can be a valid interior of some tree.
5.  find_shared( C->root() );
6.  find_shared( C->top() );
```

既然要复制节点，那就得选择哪些节点可以复制而哪些不能。所有可能有副作用的节点都不可以复制，它们一定要被匹配规则匹配为一棵子树的根；而没有副作用的节点则可以被复制，它们可以被匹配到任意位置。
Matcher::find_shared(Node *n)里，被set_shared(n)的节点就是标记为不能复制的，没被标记的就是可以复制的。然后还有set_dontcare(n)的节点，那些多半是无所谓或者无法匹配的节点。

yuyinyang 2014-06-09
R神，HotSpot运行阶段（包括JIT编译生成代码）有没有那种显式地清空所有寄存器的行为？有没有某些寄存器是一直不会被用到的？比如index比较大的那些xmm寄存器？
有没有可能我在VM初始化阶段为一个不会被用到的寄存器（假设是xmm7）附上某个值，之后可以一直使用它？

RednaxelaFX 2014-06-10

> **yuyinyang 写道**
> R神，HotSpot运行阶段（包括JIT编译生成代码）有没有那种显式地清空所有寄存器的行为？有没有某些寄存器是一直不会被用到的？比如index比较大的那些xmm寄存器？
> 有没有可能我在VM初始化阶段为一个不会被用到的寄存器（假设是xmm7）附上某个值，之后可以一直使用它？

您不能依赖某个GPR没被HotSpot使用...
例如，看看ad文件，里面声明了的寄存器都会被C2的寄存器分配器使用（除了RIP/RSP/RBP）。所以那些你都不能指望值放进去了不会变。x86的ISA上暴露出来的通用寄存器那么少，当然得用到尽咯。

我印象中HotSpot在x86上没用gs段寄存器。您要放的值假如是个地址的话可以放在gs里，然后要访问它的时候指定段寄存器就是。

yuyinyang 2014-06-10
我不指望exclusive地使用某个GPR...我做了一件非常naive的事情，我把hotspot里所有关于xmm8-15的声明和使用的地方全部remove了（包括ad文件等），然后跑了一些benchmark也都顺利通过了，这是否能保证hotspot不会用到这些寄存器了？

我在看JIT生成代码的那段output方法，有个地方不太明白。JIT貌似会先为每个CFG block根据它所用到的instructions的大小计算block初始的大小blk_starts[i]，然后实际generate出来的大小是cb->insts_size()。我想知道它到底是如何确定block和instruction的初始size的呢？

yuyinyang 2014-06-16
上面的问题已经找到了，一些指令的大小是固定的写在adfile里的，没有在adfile里指定大小的会通过把指令emit到一个trash code buffer里的计算出指令实际的size
又碰到新的问题，挂在instruction selection的阶段，下面这个dump出来的东西完全看不明白。。

> **引用**
>
> 035 MulD === _ 033 034 [[036]]

```
--N: 035       MulD   === _ 033 034  [[036 ]]

  --N: 033     ConvI2D === _ 010  [[035 ]]
  REGD  100  convI2D_reg_reg
  STACKSLOTD  195  storeSSD

       --N: 010  Parm   === 03  [[033 ]] Parm0: int
       RREGI  0  RREGI
       RAX_REGI  0  RAX_REGI
       RBX_REGI  0  RBX_REGI
       RCX_REGI  0  RCX_REGI
       RDX_REGI  0  RDX_REGI
       RDI_REGI  0  RDI_REGI
       NO_RCX_REGI  0  NO_RCX_REGI
       NO_RAX_RDX_REGI  0  NO_RAX_RDX_REGI
       STACKSLOTI  100  storeSSI

  --N: 034     ConvF2D === _ 011  [[035 ]]
  REGD  100  convF2D_reg_reg
  STACKSLOTD  195  storeSSD
  _ConvF2D_regF_  0  _ConvF2D_regF_

       --N: 011  Parm   === 03  [[034 ]] Parm1: float
       REGF  0  REGF
       STACKSLOTF  95  storeSSF

# To suppress the following error report, specify this argument
# after -XX: or in .hotspotrc:  SuppressErrorAt=/matcher.cpp:1513
#
# A fatal error has been detected by the Java Runtime Environment:
#
#  Internal Error (/home/yuyinyang/openjdk-b24-28_aug_2012/hotspot/src/share/vm/opto/matcher.cpp:1513),
pid=234697, tid=140299994089216
#  assert(false) failed: bad AD file
#
# JRE version: 7.0
# Java VM: OpenJDK 64-Bit Server VM (23.2-b09-fastdebug mixed mode linux-amd64 compressed oops)
# Failed to write core dump. Core dumps have been disabled. To enable core dumping, try "ulimit -c unlimited"
before starting Java again
#
# An error report file with more information is saved as:
# /home/yuyinyang/workspace/test/hs_err_pid234697.log
#
# If you would like to submit a bug report, please visit:
#   http://bugreport.sun.com/bugreport/crash.jsp
#
Current thread is 140299994089216
Dumping core ...
Aborted
```

hs_err文件里的部分内容：

引用

```
---------------  T H R E A D  ---------------

Current thread (0x00007f9ba4309000):  JavaThread "C2 CompilerThread0" daemon [_thread_in_native,
id=234856, stack(0x00007f9a233f4000,0x00007f9a234f5000)]

Stack: [0x00007f9a233f4000,0x00007f9a234f5000],  sp=0x00007f9a234eff40,  free space=1007k
```

Native frames: (J=compiled Java code, j=interpreted, Vv=VM code, C=native code)
V [libjvm.so+0xa7cfba] VMError::report(outputStream*)+0xf6e
V [libjvm.so+0xa7e285] VMError::report_and_die()+0x5dd
V [libjvm.so+0x5362db] report_vm_error(char const*, int, char const*, char const*)+0x8c
V [libjvm.so+0x85eb7b] Matcher::Label_Root(Node const*, State*, Node*, Node const*)+0x4ef
V [libjvm.so+0x85ea7b] Matcher::Label_Root(Node const*, State*, Node*, Node const*)+0x3ef
V [libjvm.so+0x85e26a] Matcher::match_tree(Node const*)+0x1d0
V [libjvm.so+0x85cc67] Matcher::xform(Node*, int)+0x1a1
V [libjvm.so+0x85a4a0] Matcher::match()+0xb2c
V [libjvm.so+0x4cb40c] Compile::Code_Gen()+0xac
V [libjvm.so+0x4c6cd8] Compile::Compile(ciEnv*, C2Compiler*, ciMethod*, int, bool, bool)+0x1070
V [libjvm.so+0x416a22] C2Compiler::compile_method(ciEnv*, ciMethod*, int)+0xd2
V [libjvm.so+0x4d7235] CompileBroker::invoke_compiler_on_method(CompileTask*)+0x391
V [libjvm.so+0x4d6ace] CompileBroker::compiler_thread_loop()+0x32c
V [libjvm.so+0xa32250] compiler_thread_entry(JavaThread*, Thread*)+0x57
V [libjvm.so+0xa2de00] JavaThread::thread_main_inner()+0x11e
V [libjvm.so+0xa2dce0] JavaThread::run()+0x110
V [libjvm.so+0x8f29b1] java_start(Thread*)+0x1bf

Current CompileTask:
C2: 25506 26 java.lang.StringCoding::access$000 (6 bytes)