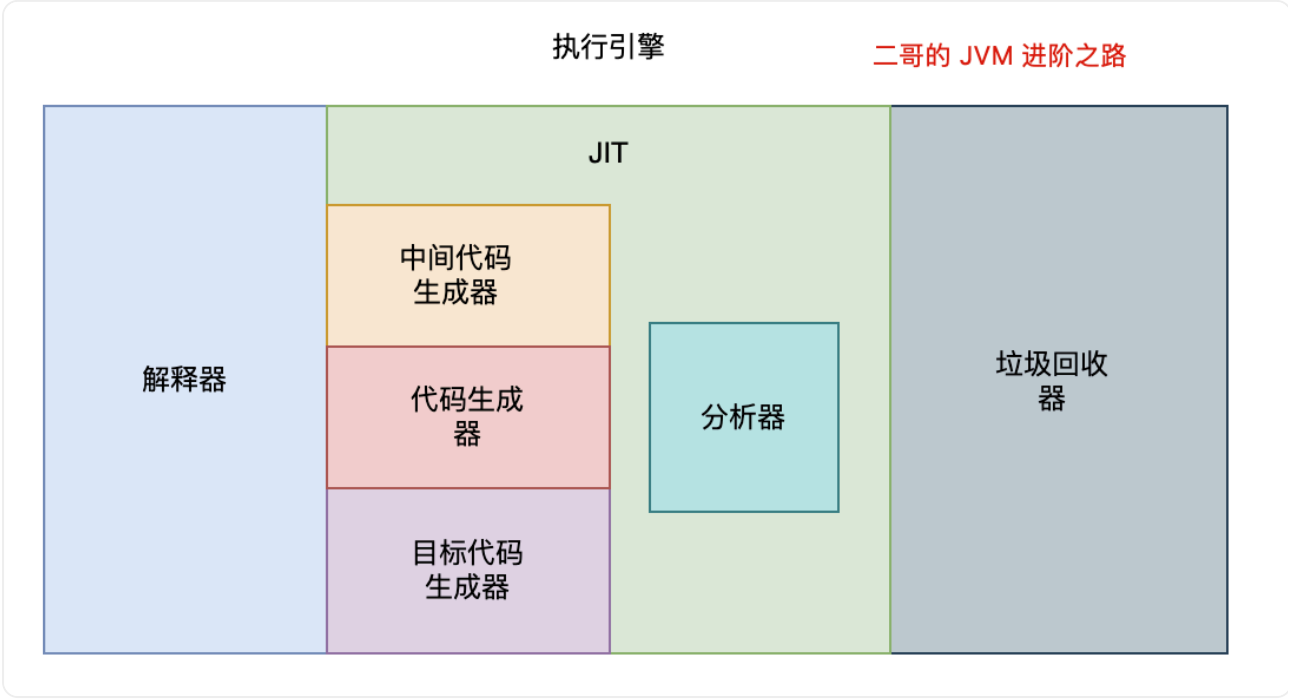
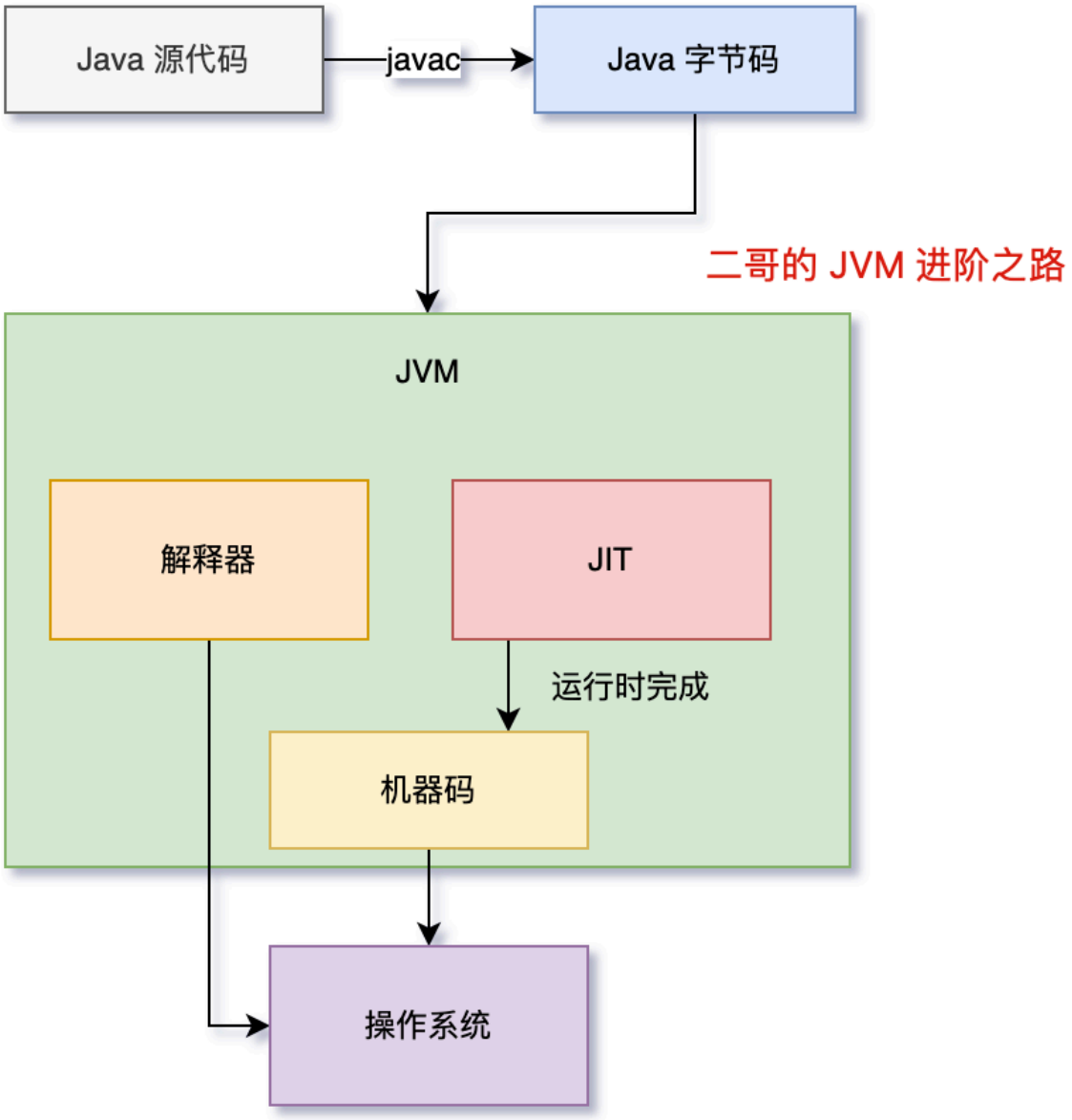
10 张手绘图 8000 字深入理解 JIT（即时编译器）

沉默王二 2022年3月27日 Java核心 Java虚拟机 约 7535 字 大约 25 分钟

前面我们讲了 ，为了提升 Java 运行时的性能，JVM 引入了 JIT，也就是即时编译（Just In Time）技术。



Java 代码首先被编译为字节码，JVM 在运行时通过解释器执行字节码。当某部分的代码被频繁执行时，JIT 会将这些热点代码编译为机器码，以此来提高程序的执行效率。



那为什么 JIT 就能提高程序的执行效率呢，解释器不也是将字节码翻译为机器码交给操作系统执行吗？

解释器在执行程序时，对于每一条字节码指令，都需要进行一次解释过程，然后执行相应的机器指令。这个过程在每次执行时都会重复进行，因为解释器不会记住之前的解释结果。

与此相对，JIT 会将频繁执行的字节码编译成机器码。这个过程只发生一次。一旦字节码被编译成机器码，之后每次执行这部分代码时，直接执行对应的机器码，无需再次解释。

除此之外，JIT 生成的机器码更接近底层，能够更有效地利用 CPU 和内存等资源，同时，JIT 能够在运行时根据实际情况对代码进行优化（如内联、循环展开、分支预测优化等），这些优化是在机器码级别上进行的，可以显著提升执行效率。

换句话说，解释器是一个循规蹈矩的人，每次都要按照规则来执行，而 JIT 是一个“偷奸耍滑”的人，他会根据实际情况来做出最优的选择。

好，我们再来梳理一下。

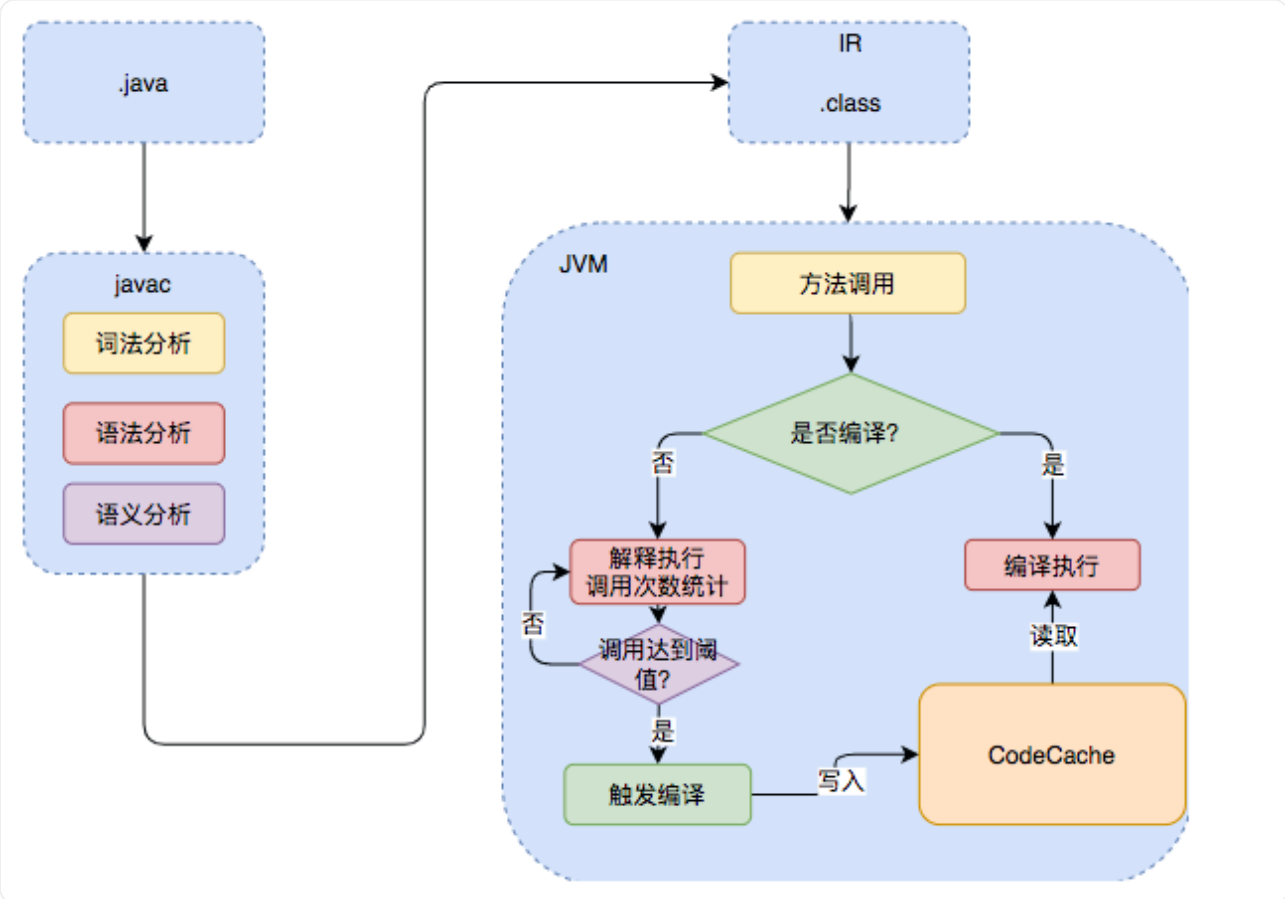
Java 的执行过程分为两步，第一步由 javac 将源码编译成字节码，在这个过程中会进行词法分析、语法分析、语义分析。

第二步，解释器会逐行解释字节码并执行，在解释执行的过程中，JVM 会对程序运行时的信息进行收集，在这些信息的基础上，JIT 会逐渐发挥作用，它会把字节码编译成机器码，但不是所有的代码都会被编译，只有被 JVM 认定为热点代码，才会被编译。

怎么样才会被认为是热点代码呢?

JVM 中有一个阈值，当方法或者代码块的在一定时间内的调用次数超过这个阈值时就会被认定为热点代码，然后编译存入 codeCache 中。当下次执行时，再遇到这段代码，就会从 codeCache 中直接读取机器码，然后执行，以此来提升程序运行的性能。

整体的执行过程大致如下图所示：



这里的 codeCache 让我想起了 Redis ，Redis 也是将热点数据存储在内存中，以此来提升访问速度。

OK，解释清楚了 JIT 的原理，我们来看看 JIT 的实现。

JVM 的编译器

JVM 中集成了两种编译器，一种是 Client Compiler，另外一种是 Server Compiler。

Client Compiler 注重启动速度和局部的优化，Server Compiler 则更加关注全局的优化，性能会更好，但由于会进行更多的全局分析，所以启动速度会慢一些。

两种编译器相辅相成，互为臂膀，共同把 JVM 的性能带到了一个新的高度。

Client Compiler

就那虚拟机中的太子 HotSpot 来说吧，它就带有一个 Client Compiler，被称为 C1 编译器，启动速度极快。

C1 通常会做这三件事：

①、**局部简单可靠的优化**，比如在字节码上进行一些基础优化，方法内联、常量传播等。

我们来举例看一下什么是方法内联，假设我们有两个简单的方法：

```java
public class Example {
    public int add(int a, int b) {
        return a + b;
    }

    public void run() {
        int result = add(5, 3);
        System.out.println(result);
    }
}
```

在执行 run 方法时，会调用 add 方法，方法内联优化后，会将 add 方法的字节码直接插入到 run 方法中，这样就不用再去调用 add 方法了，直接执行 run 方法就可以了。

```java
public class Example {
    public void run() {
        int a = 5;
        int b = 3;
        int result = a + b;  // 这里是内联后的 add 方法体
```

```
        System.out.println(result);
    }
}
```

②、**将字节码编译成 HIR**（High-level Intermediate Representation），别计较它中文名叫什么，我觉得与其死板的翻译，不如就记住它叫 HIR，一种比较接近源代码的形式。

通过借助 HIR 我们可以实现冗余代码消除、死代码删除等编译优化工作，我们同样通过代码来看一下。

```java
public class OptimizationExample {
    public int calculate(int x, int y) {
        int a = x + y;
        int b = x + y;  // 冗余计算
        return a * b;
    }
}
```

很明显，上面的代码中，b 的值是可以直接通过 a 的值计算出来的，所以 b 的计算就是冗余的，我们可以通过 HIR 来消除这种冗余计算。

```java
public class OptimizationExample {
    public int calculate(int x, int y) {
        int a = x + y;
        return a * a;  // 使用一个计算结果
    }
}
```

在 HIR 优化阶段，编译器识别到 x + y 的计算是冗余的，因此它将第二次计算的结果用第一次的结果替换。

③、**最后将 HIR 转换成 LIR**（Low-level Intermediate Representation），比较接近机器码了。这期间会做一些寄存器分配、窥孔优化等。

寄存器分配是指在编译时将程序中的变量分配到 CPU 的寄存器上。由于寄存器的访问速度远快于内存，因此合理的寄存器分配可以显著提高程序的执行效率。

来看这段代码：

```java
int a = 5;
int b = 10;
int c = a + b;
```

```
System.out.println(c);
```

在没有寄存器优化的情况下，编译器会将变量 a、b、c 分配到内存中，然后在执行时，再从内存中读取变量的值。有了寄存器分配优化呢?

```java
R1 = 5     // 将 5 赋值给寄存器 R1
R2 = 10    // 将 10 赋值给寄存器 R2
R3 = R1 + R2 // 将 R1 和 R2 的和赋值给寄存器 R3
```

这样，变量 a、b、c 就被分配到了寄存器 R1、R2、R3 上，而不是内存中，寄存器的访问速度远快于内存，所以这样的优化可以提高程序的执行效率。

窥孔优化（Peephole Optimization）是一种在生成机器码阶段进行的局部优化技术。编译器“窥视”一小段生成的机器码，并尝试找出并替换更高效的指令序列。

假设有这样一段简单的机器码：

```java
MOV R1, 0
ADD R1, 5
```

这段代码首先将寄存器 R1 置零，然后再将 5 加到 R1 上，窥孔优化会将这两条指令合并成一条：

```java
MOV R1, 5
```

这样，仅用一条指令就完成了同样的操作，显然会提高代码执行的效率。

Server Compiler

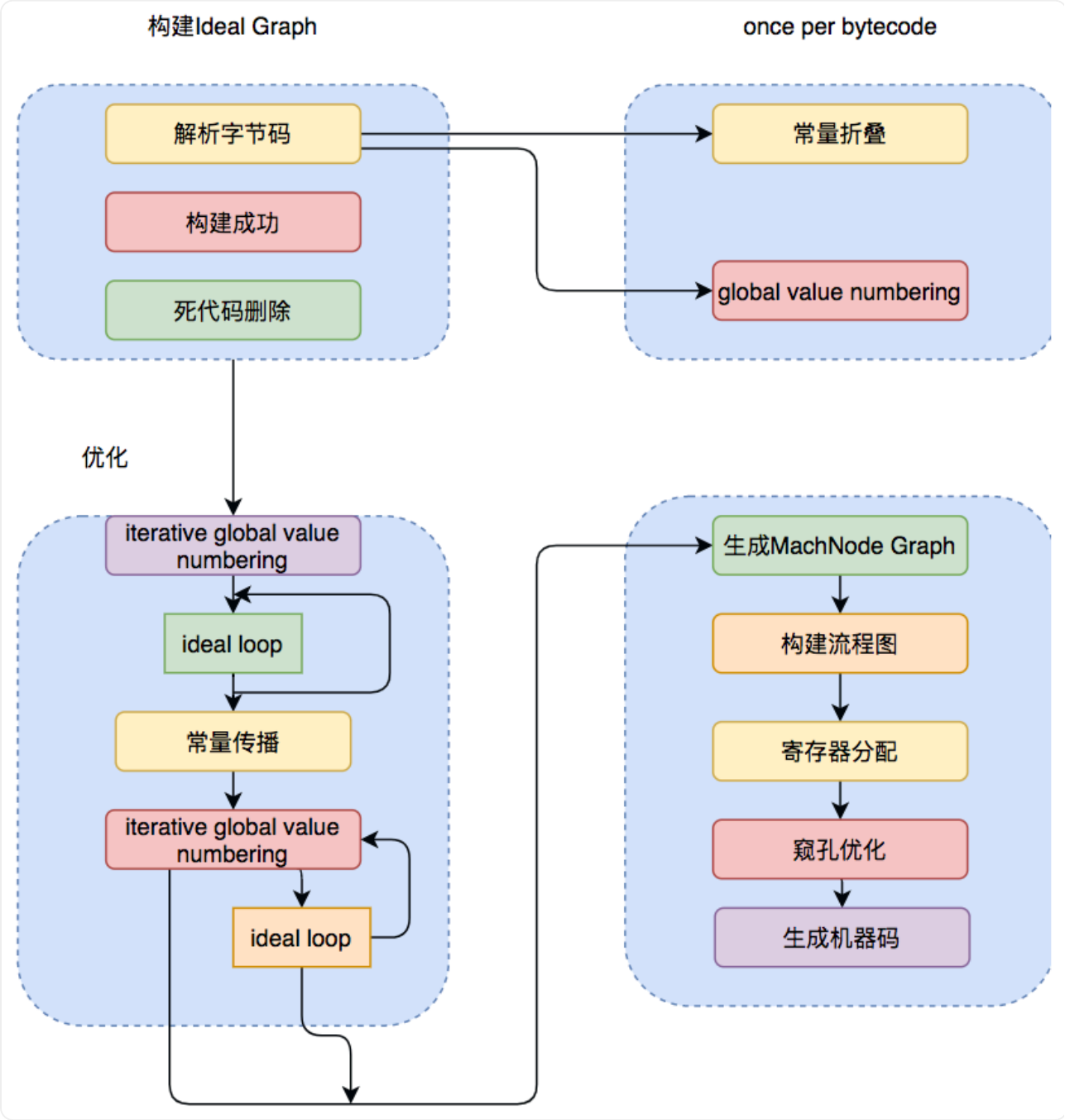
Server Compiler 主要关注一些编译耗时较长的全局优化，甚至会还会根据程序运行的信息进行一些不可靠的激进优化。这种编译器的启动时间长，适用于长时间运行的后台程序，它的性能通常比 Client Compiler 高 30%以上。目前，Hotspot 虚拟机中使用的 Server Compiler 有两种：C2 和 Graal。

C2 Compiler

Hotspot 中，默认的 Server Compiler 是 C2 编译器。

C2 编译器在进行编译优化时，会使用一种控制流与数据流结合的图数据结构，称为 Ideal Graph，我愿称之为“理想图”。

Ideal Graph 表示当前程序的数据流向和指令间的依赖关系，依靠这种图结构，某些优化步骤（尤其是涉及浮动代码块的优化步骤）会变得不那么复杂。



解析字节码的时候，C2 会向一个空的 Graph 中添加节点，Graph 中的节点通常对应一个指令块，每个指令块包含多条相关联的指令，JVM 会利用一些优化技术对这些指令进行优化，比如 Global Value Numbering、常量折叠等，解析结束后，还会进行一些死代码剔除的操作。

生成 Ideal Graph 后，会在这个基础上结合收集到的程序运行信息来进行一些全局的优化。

无论是否进行全局优化，Ideal Graph 都会被转化为一种更接近机器层面的 MachNode Graph，最后编译的机器码就是从 MachNode Graph 中得到的。

Graal Compiler

从 JDK 9 开始，Hotspot 中集成了一种新的 Server Compiler，也就是 Graal 编译器。相比 C2，Graal 有这样几种关键特性：

①、JVM 会在解释执行的时候收集程序运行的各种信息，然后根据这些信息进行一些基于预测的激进优化，比如分支预测，根据程序不同分支的运行概率，选择性地编译一些概率较大的分支。Graal 比 C2 更加青睐这种优化，所以 Graal 的峰值性能通常要比 C2 更好。

②、与 C2（主要用 C++ 编写）不同，Graal 使用 Java 语言编写。这样做的好处是，Graal 可以直接使用 JVM 的内存管理机制，不需要像 C2 那样自己实现内存管理，这样就可以避免一些内存管理上的问题。

③、Graal 引入了许多现代化的编译优化技术，例如更复杂的内联策略、循环优化等，这些在某些情况下可以比 C2 产生更优化的代码。

④、改进的逃逸分析有助于更好地进行栈上分配和锁消除，从而提升性能。

⑤、Graal 不仅能作为 JIT 编译器使用，还支持 Ahead-of-Time（AOT）编译，这有助于减少 Java 应用的启动时间和内存占用。

Graal 编译器可以通过 JVM 参数 `-XX:+UnlockExperimentalVMOptions -XX:+UseJVMCICompiler` 启用。当启用时，它将替换掉 HotSpot 中的 C2，并响应原本由 C2 负责的编译请求。

JVM 的分层编译

Java 7 引入了分层编译的概念，它结合了 C1 和 C2 的优势，追求启动速度和峰值性能的一个平衡。分层编译将 JVM 的执行状态分为了五个层次。五个层级分别是：

Java 7 中引入的分层编译（Tiered Compilation）确实是一种结合了 C1 编译器（Client Compiler）和 C2 编译器（Server Compiler）优势的技术。分层编译旨在优化 Java 程序的启动速度和长期运行时的性能。这一机制通过在不同的层级应用不同的编译策略，以达到快速启动和最高性能的平衡。在 HotSpot JVM 中，分层编译将程序执行状态分为五个层次：

1. **层级 0 - 解释器（Interpreter）**：这是程序最初执行的阶段，代码通过解释器逐行解释执行。这一阶段的目的是尽快开始执行而不等待编译完成。

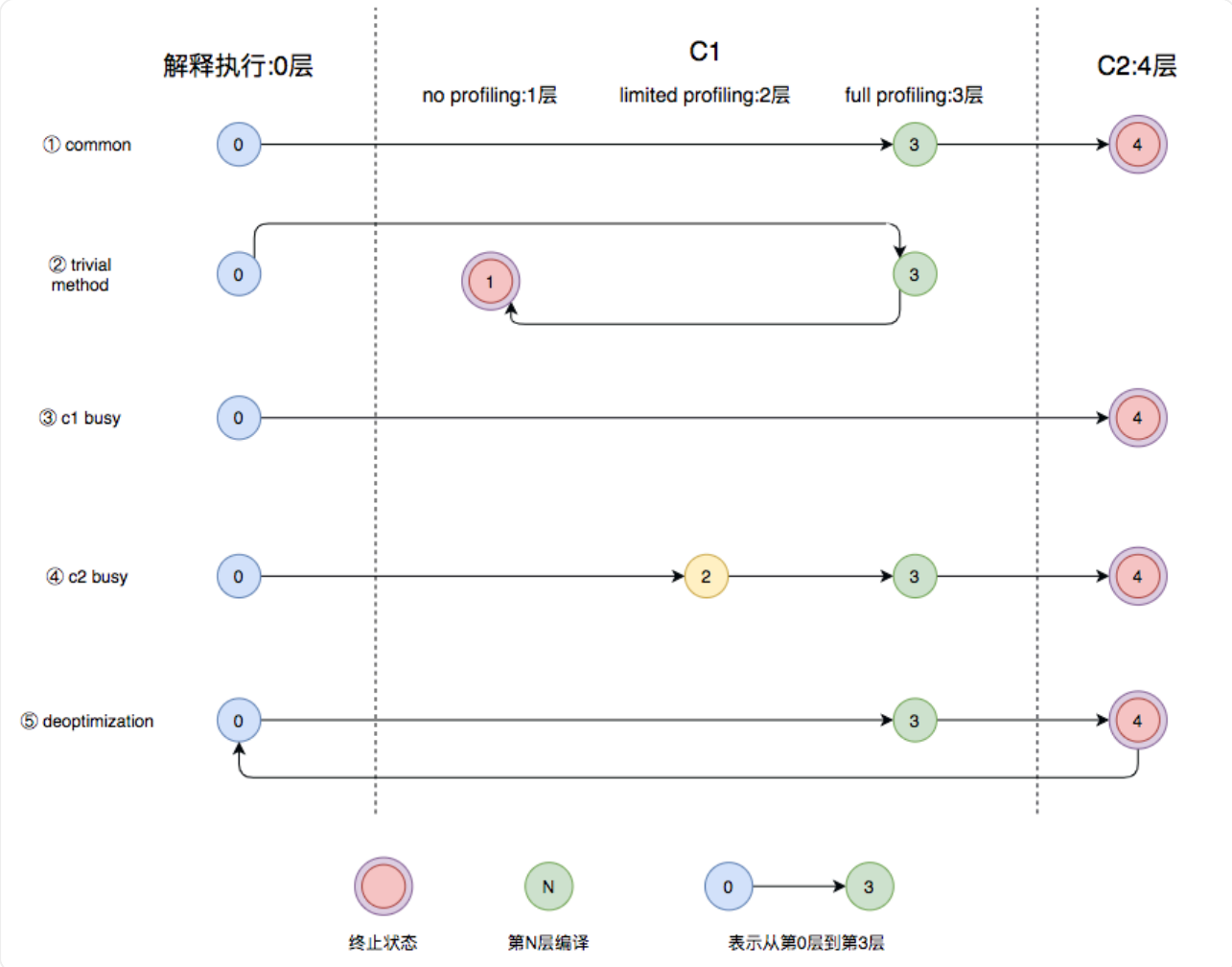
2. **层级 1 - C1 编译器带有轻量级优化（C1 with Simple Optimizations）**：在这一层级，代码首次由 C1 编译器编译，应用了一些基本的优化，如方法内联。这一阶段的编译速度较快，能迅速提供优于解释执行的性能。

3. **层级 2 - C1 编译器带有完整优化（C1 with Full Optimizations）**：此层级仍由 C1 编译器处理，但应用了更多优化技术，如逃逸分析。虽然这些优化需要更长的编译时间，但能进一步提升运行性能。

4. **层级 3 - C1 编译器带有分析数据收集（C1 with Profiling）**： 在这个层级，C1 编译器除了执行优化，还收集方法执行的详细分析数据（如分支频率、热点代码等）。这些数据将用于 C2 编译器的后续优化。

5. **层级 4 - C2 编译器优化（C2 Optimizations）**：最终阶段由 C2 编译器处理，它使用收集的分析数据进行深入优化。C2 编译器的优化更加彻底和复杂，适用于长时间运行的代码，能够提供最佳的运行性能。

下图中列举了几种常见的编译路径：



1）图中第 ① 条路径，代表编译的一般情况，热点方法从解释执行到被 3 层的 C1 编译，最后被 4 层的 C2 编译。

2）如果方法比较小（比如 getter/setter），3 层的 profiling 没有收集到有价值的数据，JVM 就会断定该方法对于 C1 代码和 C2 代码的执行效率相同，就会执行图中第 ② 条路径。

在这种情况下，JVM 会在 3 层编译之后，放弃进入 C2 编译，直接选择用 1 层的 C1 编译运行。

3）在 C1 忙碌的情况下，执行图中第 ③ 条路径，在解释执行过程中对程序进行 profiling，根据信息直接由第 4 层的 C2 编译。

4）C1 中的执行效率是 1 层>2 层>3 层，第 3 层一般要比第 2 层慢 35%以上，所以在 C2 忙碌的情况下，执行图中第 ④ 条路径。这时方法会被 2 层的 C1 编译，然后再被 3 层的 C1 编译，以减少方法在 3 层的执行时间。

5）如果编译器做了一些比较激进的优化，比如分支预测，在实际运行时发现预测出错，这时就会进行反优化，重新进入解释执行，图中第 ⑤ 条执行路径代表的就是反优化。

分层编译通过在不同的阶段应用不同程度的优化，既提供了较快的应用启动时间，又确保了长时间运行的应用能达到峰值性能。这种动态适应的编译策略是 Java 平台持续优化性能的关键手段之一。

从 JDK 8 开始，JVM 默认开启分层编译。

JIT 的触发

JVM 根据方法的调用次数以及循环回边的执行次数来触发 JIT。

循环回边是一个控制流图中的概念，程序中可以简单理解为往回跳转的指令，比如下面这段代码：

```java
public void nlp(Object obj) {
  int sum = 0;
  for (int i = 0; i < 200; i++) {
    sum += i;
  }
}
```

上面这段代码经过编译生成下面的字节码 。

```
public void nlp(java.lang.Object);
    Code:
       0: iconst_0
       1: istore_1
       2: iconst_0
       3: istore_2
       4: iload_2
       5: sipush        200
       8: if_icmpge     21
      11: iload_1
      12: iload_2
      13: iadd
      14: istore_1
      15: iinc          2, 1
      18: goto          4
      21: return
```

其中，偏移量为 18 的字节码将往回跳至偏移量为 4 的字节码中。在解释执行时，每当运行一次该指令，JVM 便会将该方法的循环回边计数器加 1。

在即时编译过程中，编译器会识别循环的头部和尾部。上面这段字节码中，循环体的头部和尾部分别为偏移量为 11 的字节码和偏移量为 15 的字节码。编译器将在循环体结尾增加循环回边计数器的代码，来对循环进行计数。

当方法的调用次数和循环回边的次数的和，超过由参数 `-XX:CompileThreshold` 指定的阈值时，就会触发即时编译。

> C1 默认值为 1500；C2 默认值为 10000。

开启分层编译的情况下， `-XX:CompileThreshold` 参数设置的阈值将会失效，触发及时编译会由以下的条件来判断：

- 方法调用次数大于由参数 `-XX:TierXInvocationThreshold` 指定的阈值乘以系数。
- 方法调用次数大于由参数 `-XX:TierXMINInvocationThreshold` 指定的阈值乘以系数，并且方法调用次数和循环回边次数之和大于由参数 `-XX:TierXCompileThreshold` 指定的阈值乘以系数时。

分层编译触发条件公式(i 为调用次数，b 是循环回边次数，s 是系数)：

```
i > TierXInvocationThreshold * s || (i > TierXMinInvocationThreshold * s  && i + b > TierXCompileThreshold * s)
```

满足其中一个条件就会触发即时编译，并且 JVM 会根据当前的编译方法数以及编译线程数动态调整系数 s。

JIT 的编译优化

即时编译器会对正在运行的程序进行一系列优化，包括：

- 字节码解析过程中的分析
- 根据编译过程中代码的一些中间形式来做局部优化
- 根据程序依赖图进行全局优化

最后才会生成机器码。

中间表达形式

在编译原理中，通常会把编译器分为前端和后端，前端编译经过词法分析、语法分析、语义分析生成中间表达形式 IR（Intermediate Representation），后端会对 IR 进行优化，生成目标代码。

Java 字节码 就是一种 IR，但是字节码的结构复杂，也不适合做全局的分析优化。

现代编译器一般采用图结构的 IR，也就是所谓的静态单赋值——Static Single Assignment，SSA 是目前比较常用的一种 IR。这种 IR 的特点是每个变量只能被赋值一次，而且只有当变量被赋值之后才能使用。

举个例子（前面也讲过，这里再强调一遍）：

```java
{
  a = 1;
  a = 2;
  b = a;
}
```

我们可以轻易地发现 a = 1 的赋值是冗余的。传统的编译器需要借助数据流分析，从后至前依次确认哪些变量的值被覆盖掉了。不过，如果借助了 SSA IR，编译器则可以很容易识别冗余赋值。

上面代码的 SSA IR 形式的伪代码可以表示为：

```java
{
  a_1 = 1;
  a_2 = 2;
  b_1 = a_2;
}
```

由于 SSA IR 中每个变量只能赋值一次，所以代码中的 a 在 SSA IR 中会分成 a_1、a_2 两个变量来赋值，这样编译器就可以很容易通过扫描这些变量来发现 a_1 的赋值后并没有使用，由此认定该赋值是冗余的。

除此之外，SSA IR 对其他优化方式也有很大的帮助，例如下面这个死代码删除（Dead Code Elimination）的例子：

```java
public void DeadCodeElimination{
  int a = 2;
  int b = 0
  if(2 > 1){
    a = 1;
  } else{
    b = 2;
  }
  add(a,b)
}
```

可以得到 SSA IR 伪代码：

```
a_1 = 2;
b_1 = 0
if true:
  a_2 = 1;
else
  b_2 = 2;
add(a,b)
```

编译器通过执行字节码可以发现 else 分支不会被执行。经过死代码删除后就可以得到代码：

```java
public void DeadCodeElimination{
  int a = 1;
  int b = 0;
  add(a,b)
```

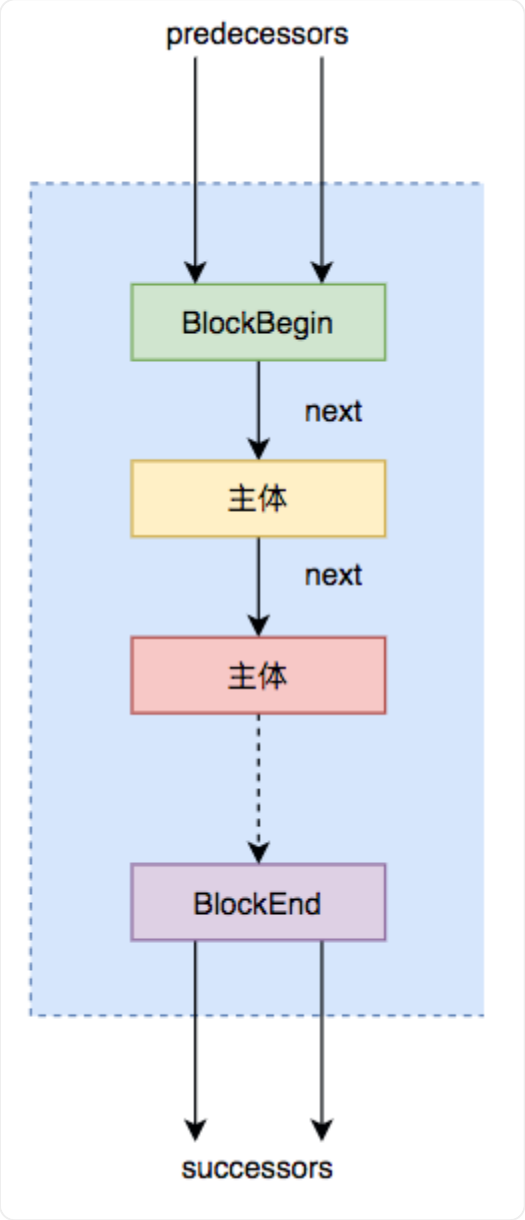
```
}
```

我们可以将编译器的每一种优化看成一个图优化算法，它接收一个 IR 图，并输出经过转换后的 IR 图。编译器优化的过程就是一个个图节点的优化串联起来的。

C1 的 HIR

前文提到了 C1 编译器内部使用了高级中间表达形式 HIR，低级中间表达形式 LIR 来进行各种优化，这两种 IR 都是 SSA 形式的。

HIR 是由很多基本块（Basic Block）组成的控制流图结构，每个块包含很多 SSA 形式的指令。基本块的结构如下图所示：



其中，predecessors 表示前驱基本块，由于前驱可能是多个，所以是 BlockList 结构，由多个 BlockBegin 组成的可扩容数组。

同样，successors 表示多个后继基本块 BlockEnd。

除了这两部分就是主体块，里面包含程序执行的指令和一个 next 指针，指向下一个执行的主体块。

从字节码到 HIR 的构造最终调用的是 GraphBuilder，GraphBuilder 会遍历字节码，将所有代码基本块存储为一个链表结构，但是这个时候的基本块只有 BlockBegin，不包括具体的指令。

第二步 GraphBuilder 会用一个 ValueStack 作为操作数栈和局部变量表 ，模拟执行字节码，构造出对应的 HIR，填充之前空的基本块，这里给出简单字节码块构造 HIR 的过程示例，如下所示：

```
字节码                    Local Value            operand stack              HIR
5: iload_1                  [i1,i2]                 [i1]
6: iload_2                  [i1,i2]                 [i1,i2]
                          ..............................................    i3:
i1 * i2
7: imul
8: istore_3                 [i1,i2, i3]              [i3]
```

可以看出，当执行 iload_1 时，操作数栈压入变量 i1，执行 iload_2 时，操作数栈压入变量 i2，执行相乘指令 imul 时弹出栈顶两个值，构造出 HIR i3 : i1 * i2，生成的 i3 入栈。

C1 编译器的大部分优化工作都是在 HIR 之上完成的。当优化完成之后它会将 HIR 转化为 LIR，LIR 和 HIR 类似，也是一种编译器内部用到的 IR，HIR 通过优化消除一些中间节点就可以生成 LIR，形式上更加简化。

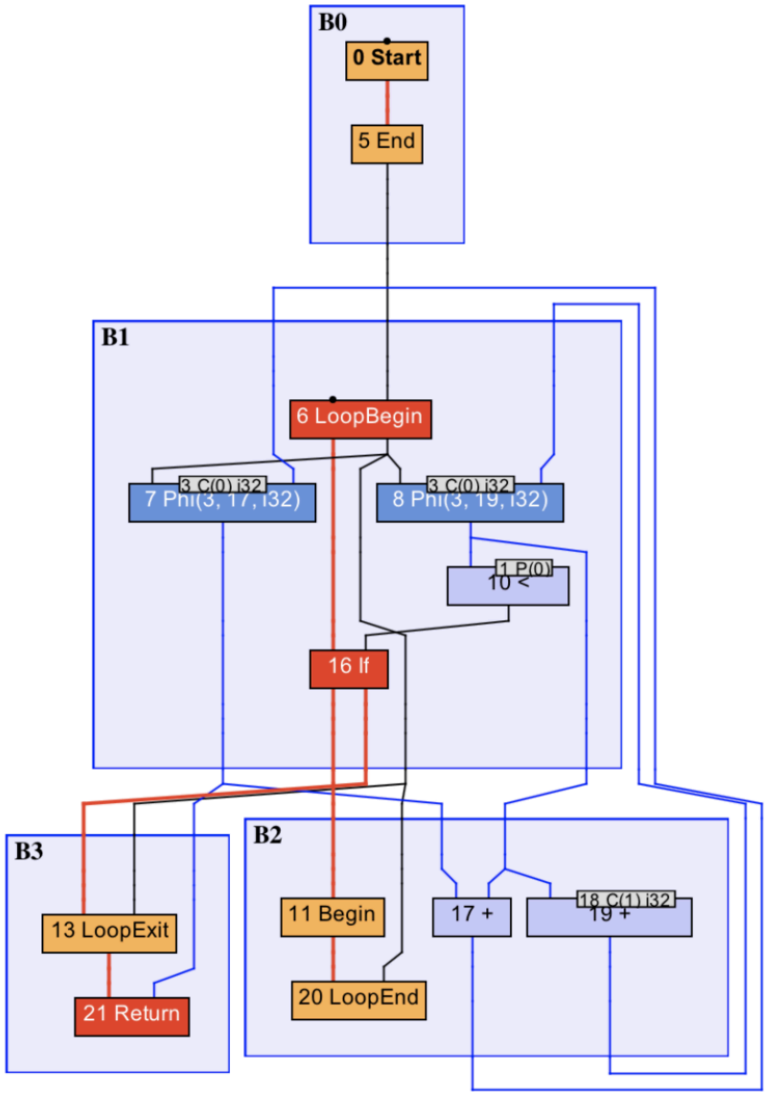
C2 的 Sea-of-Nodes IR

C2 编译器中的 Ideal Graph 采用的是一种名为 Sea-of-Nodes 中间表达形式，同样也是 SSA 形式。

它最大的特点是去除了变量的概念，直接采用值来进行运算。为了方便理解，可以利用 IR 可视化工具 Ideal Graph Visualizer（IGV） ，来展示具体的 IR 图。比如下面这段代码：

```java
public static int foo(int count) {
  int sum = 0;
  for (int i = 0; i < count; i++) {
    sum += i;
  }
  return sum;
}
```

对应的 IR 图如下所示：



B0 基本块中 0 号 Start 节点是方法入口，B3 中 21 号 Return 节点是方法出口。

红色加粗线条为控制流，蓝色线条为数据流，其他颜色的线条则是特殊的控制流或数据流。

被控制流所连接的是固定节点，其他的则是浮动节点。

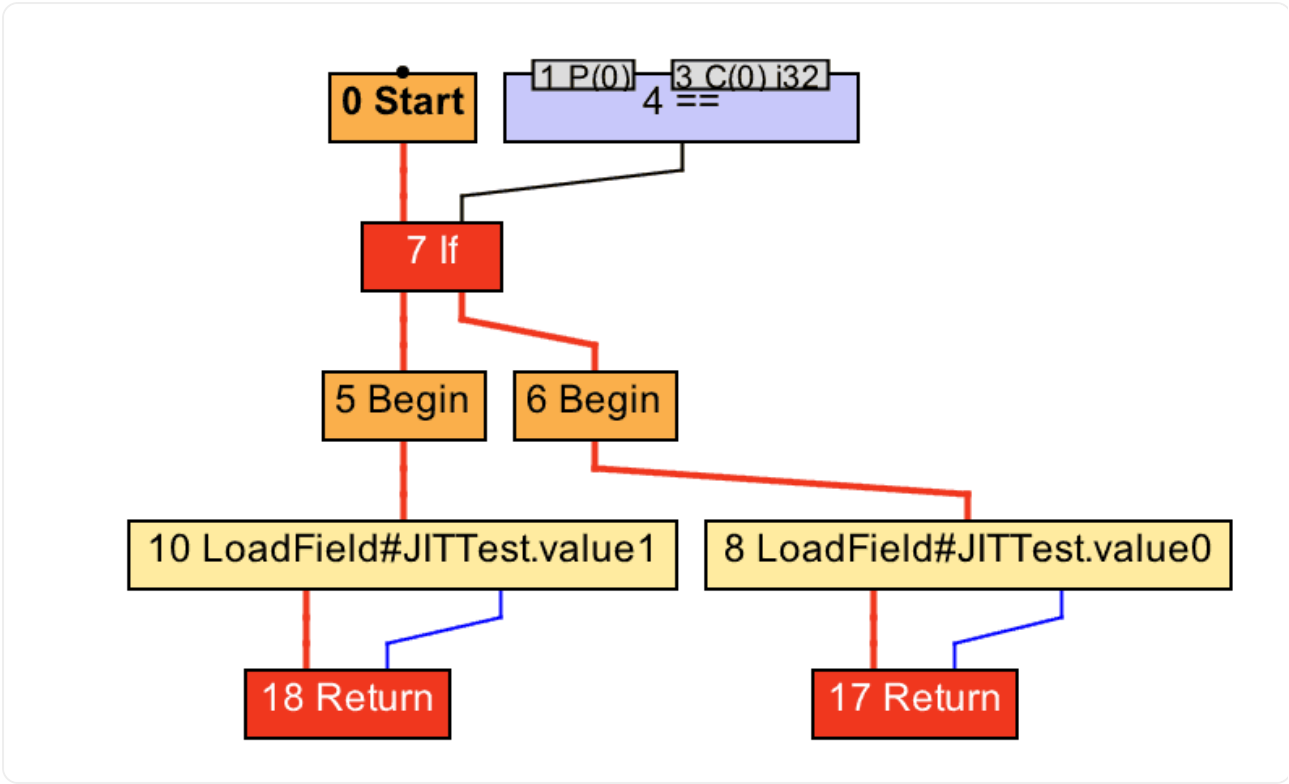
依赖于这种图结构，通过收集程序运行的信息，JVM 可以通过 Schedule 那些浮动节点，从而获得最好的编译效果。

方法内联

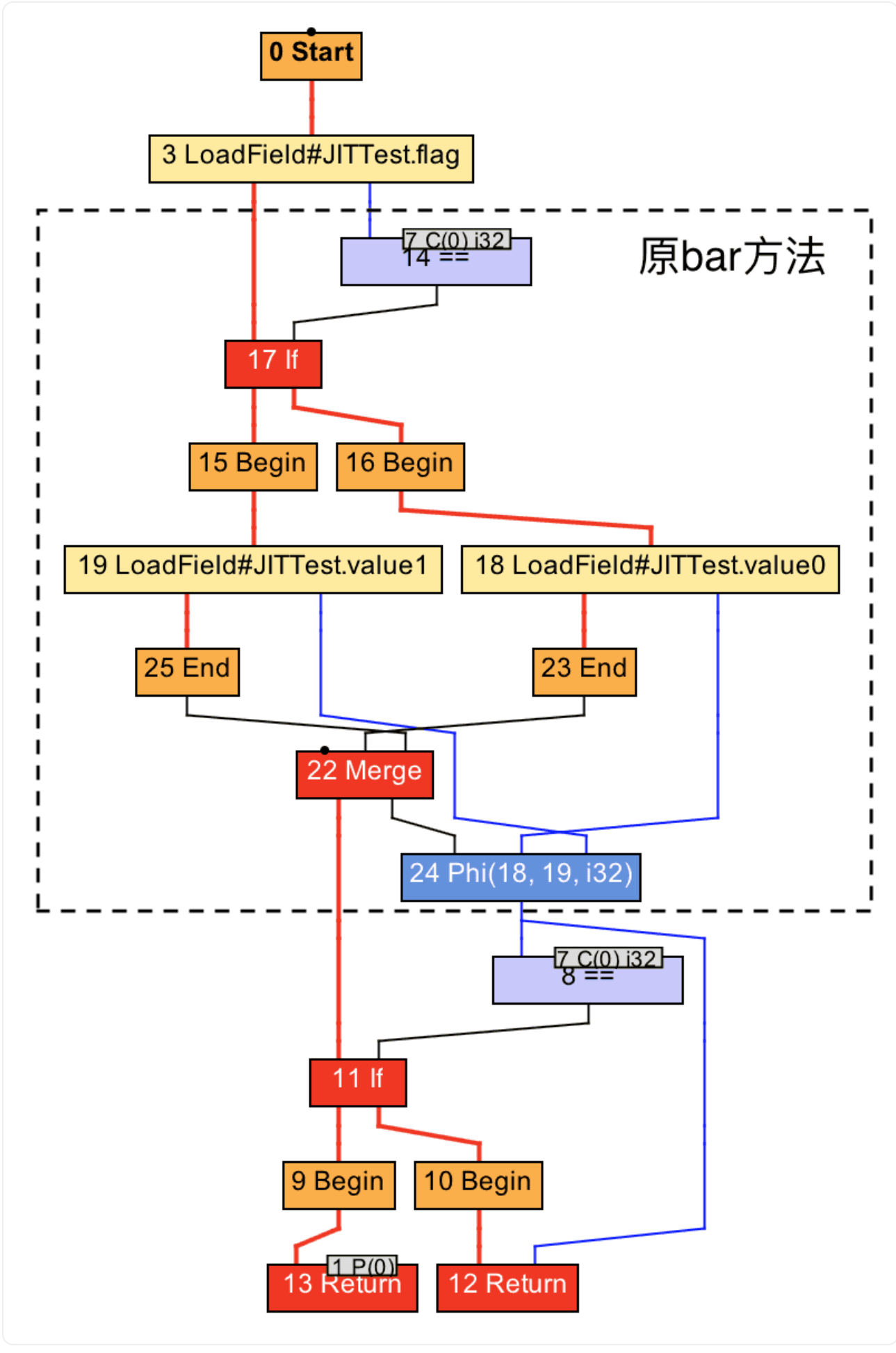
来看下面这段代码：

```java
public static boolean flag = true;
public static int value0 = 0;
public static int value1 = 1;

public static int foo(int value) {
    int result = bar(flag);
    if (result != 0) {
        return result;
    } else {
        return value;
    }
}

public static int bar(boolean flag) {
    return flag ? value0 : value1;
}
```

来看一下 bar 方法的 IR 图：


0 Start
1 P(0)
3 C(0) i32
4 ==
7 If
5 Begin
6 Begin
10 LoadField#JITTest.value1
8 LoadField#JITTest.value0
18 Return
17 Return

内联后的 IR 图:


0 Start
3 LoadField#JITTest.flag
7 C(0) i32
14 ==
原bar方法
17 If
15 Begin
16 Begin
19 LoadField#JITTest.value1
18 LoadField#JITTest.value0
25 End
23 End
22 Merge
24 Phi(18, 19, i32)
7 C(0) i32
8 ==
11 If
9 Begin
10 Begin
1 P(0)
13 Return
12 Return

内联将被调用方法的 IR 图节点复制到调用者方法的 IR 图中。在这个例子中，bar 方法的 IR 图中的 0 号 Start 节点被复制到了 foo 方法的 IR 图中，从而避免了方法调用的开销。

逃逸分析

逃逸分析是 JIT 用于优化内存管理和同步操作的重要技术。通过分析对象是否逃逸到方法或线程的外部，编译器可以做出更智能的存储和同步决策。

逃逸分析通常是在方法内联的基础上进行的，JIT 可以根据逃逸分析的结果进行诸如锁消除、栈上分配以及标量替换的优化。

下面这段代码的就是对象未逃逸的例子：

```java
public class Example {
    public static void main(String[] args) {
        example();
    }

    public static void example() {
        Foo foo = new Foo();
        Bar bar = new Bar();
        bar.setFoo(foo);
    }
}

class Foo {}

class Bar {
    private Foo foo;
    public void setFoo(Foo foo) {
        this.foo = foo;
    }
}
```

在这个例子中， `example` 方法创建了两个对象： `Foo` 和 `Bar` 。然后， `Bar` 对象通过 `setFoo` 方法引用了 `Foo` 对象。

1. **Foo 对象的逃逸情况**：
    - `Foo` 对象被创建并传递给 `Bar` 对象的 `setFoo` 方法。
    - 一旦 `setFoo` 方法被调用， `Foo` 对象的引用存储在 `Bar` 对象的实例变量 `foo` 中。
    - 但是， `Bar` 对象本身在 `example` 方法结束后就不再使用。

- 这意味着即使 `Foo` 对象的引用被存储在另一个对象中，但由于 `Bar` 对象本身也不会逃逸出 `example` 方法，因此 `Foo` 对象实际上也没有逃逸。

2. **Bar 对象的逃逸情况**：

- `Bar` 对象在 `example` 方法中被创建并使用，但之后没有被传递到其他方法或返回。
- 因此， `Bar` 对象也没有逃逸出 `example` 方法。

根据逃逸分析的结果，JIT 可能做出以下优化决策：

①、**锁消除**：如果 `Foo` 或 `Bar` 类中有同步块 （使用 `synchronized` ），由于对象没有逃逸，编译器可以安全地消除这些锁操作。

②、**栈上分配**：由于 `Foo` 和 `Bar` 对象都没有逃逸到方法之外，编译器可以选择在栈上分配这两个对象，而非在堆上分配。这样可以提高内存分配的效率，并减少垃圾收集器的压力。

> 堆和栈的区别可以查看这篇内容：JVM 的内存数据区

我们都知道 Java 的对象是在堆上分配的，而堆是对所有对象可见的。同时，JVM 需要对所分配的堆内存进行管理，并且在对象不再被引用时回收其所占据的内存 。

如果逃逸分析能够证明某些新建的对象不逃逸，那么 JVM 完全可以将其分配至栈上，并且在 new 语句所在的方法退出时，通过弹出当前方法的栈桢 来自动回收所分配的内存空间。

这样一来，我们便无须借助垃圾收集器 来处理不再被引用的对象。

不过 Hotspot 并没有进行实际的栈上分配，而是使用了标量替换的技术。

③、**标量替换**（Scalar Replacement）

标量替换是一种优化技术，其中编译器将一个聚合对象分解为其各个字段。如果这个对象没有逃逸出方法，那么它的各个字段可以视为独立的局部变量。

这种技术允许编译器进行更细粒度的优化，如更好的寄存器分配和减少不必要的内存分配。

考虑这段代码：

```java
public class Example {

    @AllArgsConstructor
    static class Cat {
        int age;
        int weight;
```

```
    }

    public static void example() {
        Cat cat = new Cat(1, 10);
        addAgeAndWeight(cat.age, cat.weight);
    }

    public static void addAgeAndWeight(int age, int weight) {
        // 对年龄和体重进行一些操作
    }

    public static void main(String[] args) {
        example();
    }
}
```

1. **对象的使用范围**：
   - 在 `example` 方法中创建了一个 `Cat` 对象，并将其字段 `age` 和 `weight` 传递给了 `addAgeAndWeight` 方法。
   - `Cat` 对象在 `example` 方法中创建且只在该方法中使用，没有被传递到方法外部或赋值给外部引用。
2. **逃逸分析**：
   - 由于 `Cat` 对象在方法外部没有引用，它没有逃逸出 `example` 方法的作用域。
   - 这意味着 `Cat` 对象是一个局部对象，适合进行标量替换。
3. **标量替换的应用**：
   - JVM 的 JIT 编译器会分析 `Cat` 对象的使用情况。基于逃逸分析，编译器可以决定不在堆上分配 `Cat` 对象，而是将其分解为两个独立的局部变量 `age` 和 `weight` 。
   - 这样，原本由 `Cat` 对象占用的堆空间就被节省下来，而且减少了垃圾回收的压力。
4. **优化后的执行**：
   - 在执行 `example` 方法时， `Cat` 对象的字段 `age` 和 `weight` 直接作为栈上的局部变量处理，避免了堆分配。

标量替换后的伪代码如下所示：

```java
public class Example {
    public static void example() {
        int catAge = 1;
        int catWeight = 10;
        addAgeAndWeight(catAge, catWeight);
    }

    public static void addAgeAndWeight(int age, int weight) {
        // 方法实现
    }
}
```

可以看到， `Cat` 对象被分解为两个局部变量 `catAge` 和 `catWeight` ，并且直接作为参数传递给了 `addAgeAndWeight` 方法。

窥孔优化与寄存器分配

前面我们也简单分析了一下窥孔优化与寄存器分配，相信大家对这两个概念都有了一定的了解，这里简单总结下。

窥孔优化就是将编译器所生成的中间代码中的某些组合替换为效率更高的指令组，比如强度削减、常数合并等，看下面这个例子就是一个强度削减的例子：

```java
y1=x1*3
经过强度削减后得到
y1=(x1<<1)+x1
```

编译器使用移位和加法削减乘法的强度，使用更高效率的指令组。

寄存器分配也是一种编译的优化手段，在 C2 编译器中普遍的使用。它是通过把频繁使用的变量保存在寄存器中，CPU 访问寄存器的速度比内存快得多，可以提升程序的运行速度。

经过寄存器分配和窥孔优化之后，程序就会被转换成机器码保存在 codeCache 中。

小结

本文主要介绍了 JIT 即时编译的原理以及编译优化的过程，包括：

- JIT 的触发条件
- JIT 的编译优化

- JIT 的编译器

JIT 是 JVM 的重要组成部分，它可以根据程序运行的情况，对热点代码进行编译优化，从而提升程序的运行效率。

参考链接：美团技术

上次编辑于: 2024/11/5 18:28:58
贡献者: 沉默王二
