零、编译

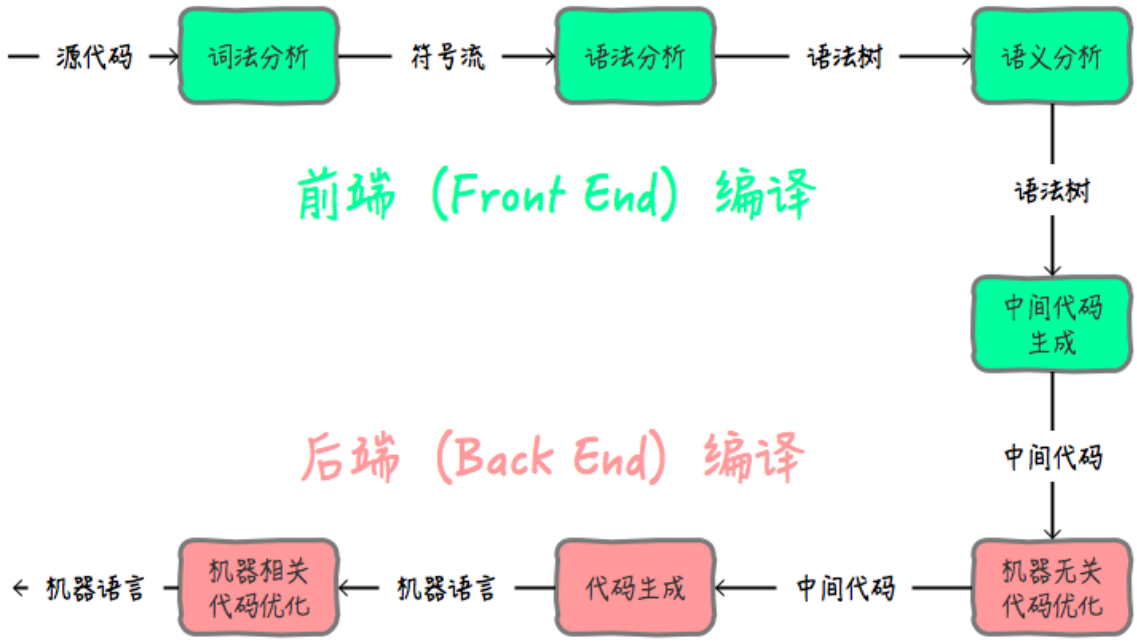
1、编译器

（1）前端编译器：*.java文件转变为*.class文件Sun的javacEclipse JDT中的增量编译器（ECJ）

（2）后端编译器：*.class文件转变为机器码HotSpot VM的C1编译器HotSpot VM的C2编译器*

（3）AOT编译器：.java文件按直接转变为机器码GNU Compiler for Java(GCJ)Excelsior JET

2、编译过程



一、前端编译

1、Javac编译过程

- 解析与填充符号表过程
  - 语法、词法分析
  - 填充符号表
- 插入式注解处理器的注解处理过程
- 分析与字节码生成过程
  - 标注检查
  - 数据及控制流分析
  - 解语法糖
  - 字节码生成

图 10-4　Javac 的编译过程

```
initProcessAnnotations(processors);  // 准备过程：初始化插入注解处理器
// These methoad calls must be chained to avoid memory leaks
delegateCompiler =
    processAnnotations( //2：执行注解处理
        enterTrees(stopIfError(CompileState.PARSE, // 1.2：填充符号表
            parseFiles(sourceFileObjects))), // 1.1：词法分析与语法分析
        classnames);

delegateCompiler.compile2(); //3：分析及字节码生成


// inner compile2
case BY_TODO:
   while(!todo.isEmpty())
        generate( // 3.4：生成字节码
            desuger( // 3.3：解语法糖
                flow( //3.2：  数据流分析
                    attribute(todo.remove())))); // 3.1：标注
   break;
```

2、解析与填充符号表

（1）词法分析

- 定义：词法分析是将源代码的字符流转变为标记（Token）集合，单个字符是程序编写的最小元素，而标记是编译过程的最小元素。如：关键字、变量名、字面量、运算符都能成为标记。
- 解析类：com.sun.tools.javac.parser.Scanner

（2）语法分析

- 定义：语法分析是根据Token序列构造抽象语法树的过程，抽象语法树（AST）是一种用来描述程序代码中的一个语法结构，如包、类型、修饰符、运算符、接口、返回值等。
- 解析类：com.sun.tools.javac.parser.JavacParser

3、填充符号表

（1）定义：符号表（Symbol Table）是一组符号地址与符号信息构成的表格。

（2）符号表中记录的信息在编译的不同阶段都要用到，如：语义分析时，符号表中的内容用于语义检查（名字与原先的说明是否一致）与生成中间代码；在目标代码生成阶段，对地址名进行地址分配就是根据符号表的记录。

（3）解析类：com.sun.tools.javac.comp.Enter

4、注解处理器

（1）JDK1.5，java语言提供了注解的支持，但当时只能在运行期发挥作用。

（2）JDK 1.6，提供了插入式注解处理器的标准API在编译期间对注解进行处理。这些注解处理器能在处理注解期间对语法树进行修改，所以需要回到解析以及填充符号表的过程，这称为一个Round。

（3）处理类：com.sun.tools.javac.processing.JavacProcessingEnvironment

5、语义分析：分析对结构正确的源程序进行上下文有关性质的审查，如：类型审查。

（1）标注检查

- 检查内容：变量使用前是否被声明、变量与赋值之间的数据类型是否能匹配等
- 常量折叠：常量相加变为一个常量
- 例子：int a = 1 + 2; => int a = 3;
- 解析类：com.sun.tools.javac.comp.Attr、com.sun.tools.javac.comp.Check

（2）数据及控制流分析

- 数据及控制流分析是对程序上下文逻辑更进一步的验证
- 验证内容：局部变量在使用前是否赋值、方法的每条路径是否有返回值、是否所有的受检异常被正确处理。
- 例子：final 只在编译期间保证变量的不变性
- 解析类：com.sun.tools.javac.comp.Flow

（3）解语法糖

- 语法糖：JVM不支持的语法，但为了让程序员编程简单而添加的高级语法，所以编译过程需要将高级语法还原为简单的基础语法结构。
- 例子：增强for循环 => 迭代器循环
- 解析类：com.sun.tools.javac.comp.TransTypes 、com.sun.tools.javac.comp.Lower

6、字节码生成

- 前面各个步骤的信息（语法树、符号表）转化为字节码写入磁盘
- 少量的添加和转换工作
    - 如：字符串加法 => StringBuilder的append方法；
    - 如：类构造器和实例构造器的生成（顺序为父类的构造器先执行）
- 关联类：com.sun.tools.javac.jvm.Gen、com.sun.tools.javac.jvm.ClassWriter

7、常见语法糖的奥秘

（1）泛型与类型擦除

Java的泛型基于类型擦除，在编译期间就把泛型变为原来的裸类型。

```
List<String> list = new ArrayList<>();
list.add("hello");
list.add("world");
System.out.println(list.get(0));

============================>

List list = new ArrayList();
list.add("hello");
list.add("world");
System.out.println((String)list.get(0));
```

(2) 自动装箱、拆箱与遍历循环

```
List<Integer> list2 = Arrays.asList(1,2,3,4,5,6);
int sum = 0;
for (int i : list2)
   sum += i;

============================>

List list2 = Arrays.asList(new Integer[] { Integer.valueOf(1), Integer.valueOf(2),
Integer.valueOf(3), Integer.valueOf(4), Integer.valueOf(5), Integer.valueOf(6) });
int sum = 0;
for (Iterator localIterator = list2.iterator(); localIterator.hasNext(); ) {
      int i = ((Integer)localIterator.next()).intValue();
      sum += i;
}

//  自动装箱 int -> Integer
1 -> Integer.valueOf(1)

// 自动拆箱 Integer -> int
Integer::intValue()

// 增强for循环
for(int i : list)
    ->
for(Iterator localIterator = list.iterator(); localIterator.hasNext(); ){
    int i = localIterator.next();
}

// 自动装箱以及自动拆箱的陷阱
Integer a = 1;
Integer b = 2;
Integer c = 3;
Integer d = 3;
Integer e = 321;
Integer f = 321;
Long g = 3L;
System.out.println(c == d); // true
System.out.println(e == f); // false,==不遇到算术运算符不自动拆箱（即两个Integer比较）
```

```
System.out.println(c == (a + b)); // true
System.out.println(c.equals(a + b)); // true
System.out.println(g == (a + b)); // true
System.out.println(g.equals(a + b)); // false
```

注意：equals方法不处理数据转换，==方法不遇到算术运算符不会自动拆箱。

```
Integer a = Integer.valueOf(1);
Integer b = Integer.valueOf(2);
Integer c = Integer.valueOf(3);
Integer d = Integer.valueOf(3);
Integer e = Integer.valueOf(321);
Integer f = Integer.valueOf(321);
Long g = Long.valueOf(3L);
System.out.println(c == d);
System.out.println(e == f);
System.out.println(c.intValue() == a.intValue() + b.intValue());
System.out.println(c.equals(Integer.valueOf(a.intValue() + b.intValue())));
System.out.println(g.longValue() == a.intValue() + b.intValue());
System.out.println(g.equals(Integer.valueOf(a.intValue() + b.intValue()))); // Integer 与 Long比较
```

（3）条件编译：com.sun.tools.javac.comp.Lower完成

```
if (true){
    System.out.println("true");
}else{
    System.out.println("false");
}

==============================>

System.out.println("true");
```

二、后端编译

1、JIT编译器

- 概述：JIT编译期能在JVM发现**热点代码**时，将这些热点代码编译成与**本地平台相关的机器码**，并进行各个层次的优化，从而提高热点代码的执行效率。
- 热点代码：某个方法或代码块运行频繁。
- JIT编译器（Just In Time Compiler）：即时编译器。
- 目的：提高热点代码的执行效率。

2、解释器与编译器

(

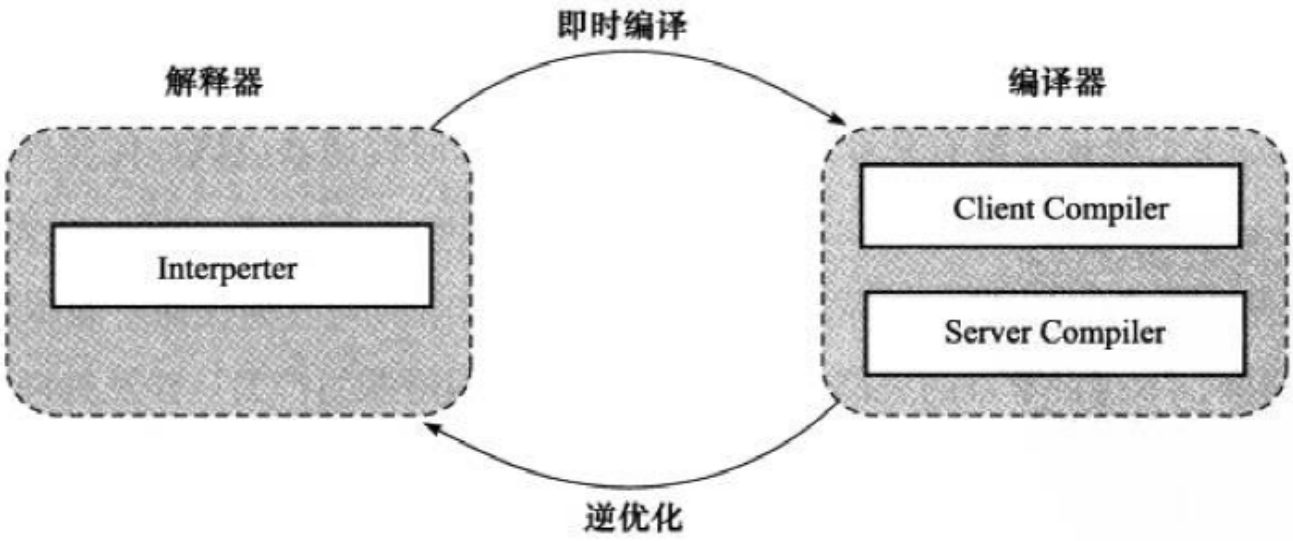


图 11-1　解释器与编译器的交互

)

（1）并存的优势

- 程序需要迅速启动和执行时，解释器先发挥作用，省去编译时间，且随时间推移，编译器将热点代码编译本地代码，提高执行效率。
- 当运行环境内存资源限制较大（嵌入式）时，使用解释执行节约内存，反之使用编译执行提升效率。
- 解释器能作为编译器激进优化的逃生门，如：编译优化出现问题，能退化为解释器状态执行 。

（2）Hotspot虚拟机内置的JIT编译器

- C1编译器（Client Compiler）：更高的编译速度
- C2编译器（Server Compiler，Opto编译器）：更好的编译质量

（3）JVM的运行模式

- 混合模式（Mixed Mode）：使用解释器 + 其中一个JIT编译器（-client / -server 指定使用哪个）
- 解释模式（Interpreted Mode）：只使用解释器（-Xint 强制JVM使用解释模式）
- 编译模式（Compiled Mode）：只使用编译器（-Xcomp JVM优先使用编译模式，解释模式作为备用）

（4）编译层次：

- 第0层：程序解释执行，解释器不开启性能监控功能，可触发第1层编译
- 第1层：C1编译，将字节码编译为本地代码，进行简单、可靠的优化，如有必要将加入性能监控逻辑
- 第2层：C2编译，同1层优化，但启动了一些编译耗时较长的优化，甚至根据性能监控信息进行不可靠的激进优化

3、编译对象与触发条件

（1）编译对象（热点代码）

- 被多次调用的方法：整个方法为编译对象
- 被多次执行的循环体（一个方法中）：整个方法为编译对象

> 循环体编译优化发生在方法执行过程中，称为栈上替换（On Stack Replacement,简称OSR编译，机方法栈帧还在栈上，方法就被替换了）

（2）热点代码探测判定

- 基于采样的热点探测：周期检查各个线程的栈顶，发现某个方法经常出现栈顶，即热点方法。
    - 实现简单，高效
    - 容易获取方法调用关系（堆栈中展开即可）
    - 但很难准备确定一个方法的热度
- 基于计数器的热点探测（Hotspot JVM采用）：为每个方法（代码块）建立计数器，统计方法的执行次数，次数超过阈值就认定为热点方法。
    - 实现复杂，每个方法维护一个计数器
    - 不能直接获取方法调用关系
    - 但统计准确和严谨

（3）基于计数器的热点探测分类

- 方法调用计数器：统计方法调用次数
    - 阈值修改：-XX：CompileThreshold
    - 设置半衰周期（周期内没有达到阈值将减半）：-XX：CounterHalfLifeTime （单位 s）
    - 关闭热度衰减：-XX：-UseCounterDecay
- 回边计数器：统计一个方法中的循环体代码执行次数，字节码中遇到控制流向后跳转的执行称为”回边“。
    - 阈值修改：-XX：BackEdgeThreshold
    - 间接调整阈值： -XX:OnStackReplacePercentage
        - Client模式下：方法调用计数器阈值 * OSR比率 / 100
        - Server模式下：方法调用计数器阈值 * （OSR比率 - 解释器监控比率<默认33>） / 100

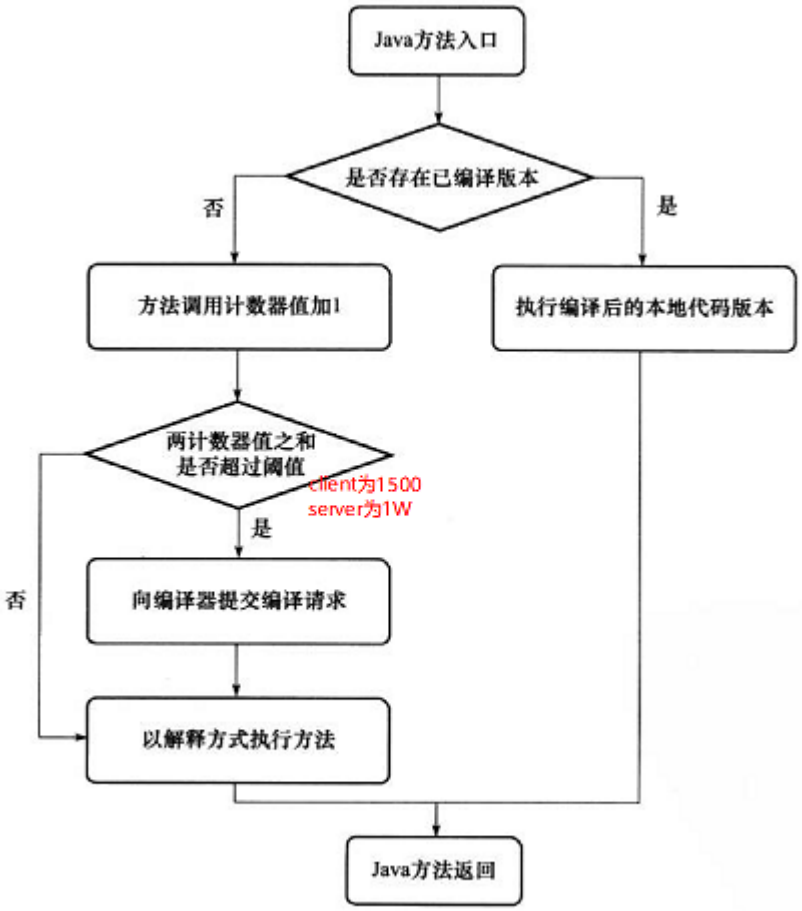
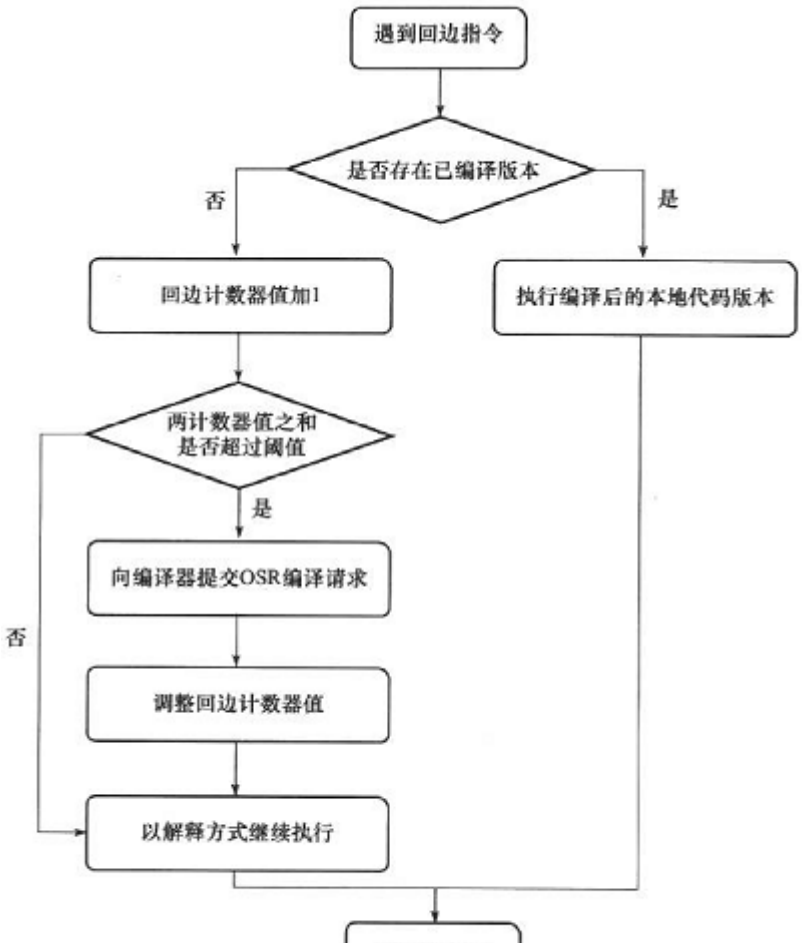


图 11-2　方法调用计数器触发即时编译



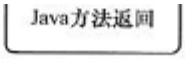


图 11-3　回边计数器触发即时编译

4、后台执行编译优化过程（-XX：BackgroudCompilation设置来禁止后台编译）

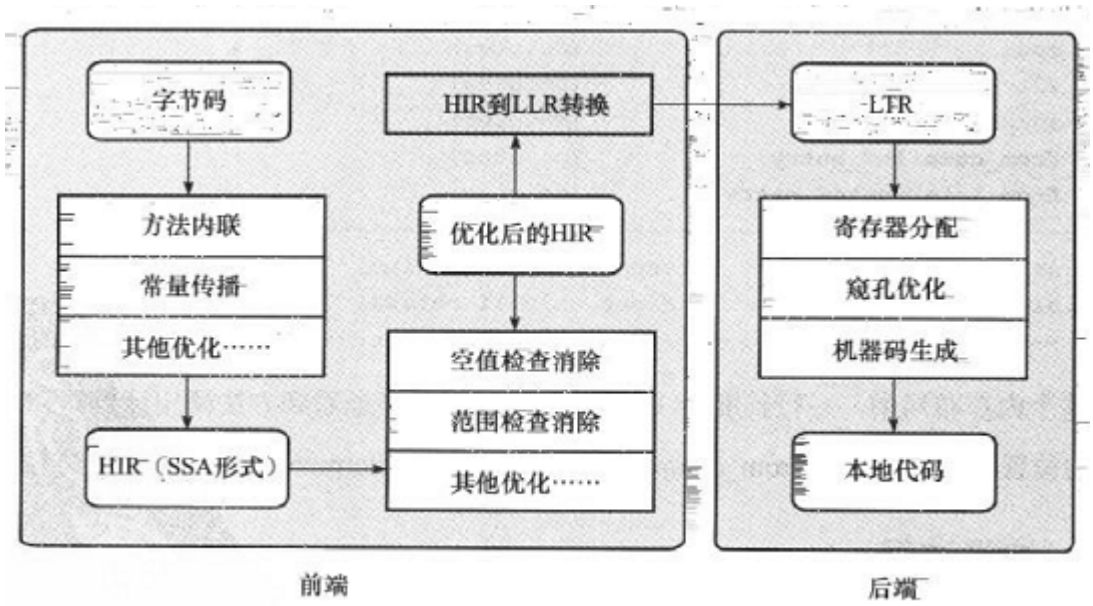


图 11-4　Client Compiler 架构

（1）一个平台独立的前端将字节码构造成一种高级中间代码（HIR,High Level Intermediate Representation）表示。HIR使用静态单分配的形式代表代码值，使得HIR的构造过程中和之后进行优化动作更容易实现。在此之前会进行基础优化，如：方法内联、常量传播等。

（2）一个平台相关的后端从HIR中产生低级中间代码（LIR）表示，而在此之前进行HIR上的优化，如：空值检查、范围检查消除等

（3）平台相关的后端使用线性扫描算法在LIR上分配寄存器，并在LIR上做窥孔优化，产生机器代码。

5、编译优化技术

（1）优化技术概述

```
// 优化前
static class B{
    int value;
    final int get(){
        return value;
    }
}

public void foo(){
    y = b.get();
    // ...do stuff
    z = b.get();
    sum = y + z;
}
```

```
// 内联优化后
public void foo(){
    y = b.value;
    // ...do stuff
    z = b.value;
    sum = y + z;
}

// 冗余访问消除或公共子表达式消除后
public void foo(){
    y = b.value;
    // ...do stuff
    z = y;
    sum = y + z;
}

// 复写传播后
public void foo(){
    y = b.value;
    // ...do stuff
    y = y;
    sum = y + y;
}

// 无用代码消除后
public void foo(){
    y = b.value;
    // ...do stuff
    sum = y + y;
}
```

（2）公共子表达式消除： 一个表达式已经被计算过且从先前的计算到现在都没发生变化，那么E就成为了公共子表达式。

```
// 未优化前
int d = (c * b) * 12 + a + (a + b + c);

// 公共子表达式消除后
int E = c * b;
int d = E * 12 + a + (E + a);

// 代数化简后
int d = 13 * E + 2 * a;
```

（3）数组边界检查消除

- foo[3] 数组下标为常量，编译期期间根据数据流分析来确定foo.length的值，并判断下标 3 没有越界，则执行的时候无需判断。
- 数组访问发生在循环中，循环变量来访问数组，如果编译器通过数据流分析得知循环变量的取值在区间内，则可以把循环中的数组上下界检查消除。

（4）方法内联

- 优点
  - 去除调用方法的成本（如建立栈帧）
  - 为其他优化建立基础
- 涉及技术：用于解决多态特性。
  - 类型继承关系分析（CHA，Class Hierarchy Analysis）技术：用于确定目前的加载类中，某个接口是否有多个实现，某个类是否有子类和子类是否抽象等。
  - 内联缓存：在未发生方法调用前，内联缓存为空，第一次调用后，缓存下方法接收者的版本信息，并且每次进行方法调用时都比较接收者版本，如果每次调用方法接收者版本一样，那么内联缓存可以继续用。但发现不一样时，即发生了虚函数的多态特性时，取消内联，查找虚方法表进行方法分派。

```
// 优化前
public static void foo(Object obj){
    if(obj != null){
        Sout("do something");
    }
}

public static void testInline(String[] args){
    Object obj = null;
    foo(obj);
}

// 优化后
public static void testInline(String[] args){
    Object obj = null;
    if(obj != null){
        Sout("do something");
    }
}
```

（5）逃逸分析：为其他优化提供**分析**手段

- 基本行为：分析对象的作用域
  - 方法逃逸：当一个对象在方法里面被定义后，它可能被外部方法所引用。如：作为调用参数传递到其他方法中。
  - 线程逃逸：当一个对象在方法里面被定义后，它可能被外部线程访问到。如：类变量或可被访问到实例变量

（6）根据逃逸分析证明一个对象不会逃逸到方法或线程中，则进行高效的优化

- 栈上分配：JVM中，对象一般在堆中分配，堆是线程共享的，进行垃圾回收和整理内存都是消耗时间的。所有确定一个对象不会逃逸时，让对象从栈上分配内存可以缓解垃圾回收的压力。
- 同步消除：如果确定一个变量不会逃逸出线程，则消除掉变量的同步措施。
- 标量替换：聚合量 => 拆开 => 成员变量恢复为原始变量 => 标量。
  - 标量是一个数据已经无法再分解成更小的数据，JVM中的原始数据类型（int、long等数值类型以及reference类型等）。反之为聚合量，如Java对象。

- 如果对象不会逃逸，则不创建该对象。方法执行时直接创建若干个相关的变量来替代。并且对象拆分后，对象的成员变量在栈上分配和读写，为进一步优化提供条件。

6、Java与C/C++编译器对比

（1）Java的劣势：

- JIT即时编译器运行占用用户运行时间
- Java语言时动态的类型安全语言，JVM频繁进行动态检查，如：实例方法访问时检查空指针、数组元素访问检查上下界、类型转换时检查继承关系
- Java使用虚方法频率高于C/C++，即多态选择频率大于C/C++。
- Java是可以动态拓展的语言，运行时加载新的类可能改变程序类型的继承关系，即编译器需要时刻注意类型变化并在运行时撤销或重新进行一些优化。
- Java对象在堆上分配，垃圾回收比C/C++语言由用户管理开销大。

（2）Java的优势：

- 开发效率高
- C/C++编译器属于静态优化，不能在运行期间进行优化。如：
  - 调用频率预测
  - 分支频率预测
  - 裁剪未被选择的