

二

22 HotSpot虚拟机的intrinsic

前不久，有同学问我，`String.indexOf` 方法和自己实现的 `indexOf` 方法在字节码层面上差不多，为什么执行效率却有天壤之别呢？今天我们就来看一看。

```
public int indexOf(String str) {  
    if (coder() == str.coder()) {  
        return isLatin1() ? StringLatin1.indexOf(value, str.value)  
                           : StringUTF16.indexOf(value, str.value);  
    }  
    if (coder() == LATIN1) { // str.coder == UTF16  
        return -1;  
    }  
    return StringUTF16.indexOfLatin1(value, str.value);  
}
```

为了解答这个问题，我们来读一下 `String.indexOf` 方法的源代码（上面的代码截取自 Java 10.0.2）。

在 Java 9 之前，字符串是用 char 数组来存储的，主要为了支持非英文字符。然而，大多数 Java 程序中的字符串都是由 Latin1 字符组成的。也就是说每个字符仅需占据一个字节，而使用 char 数组的存储方式将极大地浪费内存空间。

Java 9 引入了 Compact Strings[1] 的概念，当字符串仅包含 Latin1 字符时，使用一个字节代表一个字符的编码格式，使得内存使用效率大大提高。

假设我们调用 `String.indexOf` 方法的调用者以及参数均为只包含 Latin1 字符的字符串，那么该方法的关键在于对 `StringLatin1.indexOf` 方法的调用。

下面我列举了 `StringLatin1.indexOf` 方法的源代码。你会发现，它并没有使用特别高明的算法，唯一值得注意的便是方法声明前的 `@HotSpotIntrinsicCandidate` 注解。

```
@HotSpotIntrinsicCandidate  
public static int indexOf(byte[] value, byte[] str) {  
    if (str.length == 0) {  
        return 0;  
    }  
    if (value.length == 0) {
```

```

        return -1;
    }
    return indexOf(value, value.length, str, str.length, 0);
}

@HotSpotIntrinsicCandidate
public static int indexOf(byte[] value, int valueCount, byte[] str, int strCount, i
    byte first = str[0];
    int max = (valueCount - strCount);
    for (int i = fromIndex; i <= max; i++) {
        // Look for first character.
        if (value[i] != first) {
            while (++i <= max && value[i] != first);
        }
        // Found first character, now look at the rest of value
        if (i <= max) {
            int j = i + 1;
            int end = j + strCount - 1;
            for (int k = 1; j < end && value[j] == str[k]; j++, k++);
            if (j == end) {
                // Found whole string.
                return i;
            }
        }
    }
    return -1;
}

```

在 HotSpot 虚拟机中，所有被该注解标注的方法都是 HotSpot intrinsic。对这些方法的调用，会被 HotSpot 虚拟机替换成高效的指令序列。而原本的方法实现则会被忽略掉。

换句话说，HotSpot 虚拟机将为标注了 `@HotSpotIntrinsicCandidate` 注解的方法额外维护一套高效实现。如果 Java 核心类库的开发者更改了原本的实现，那么虚拟机中的高效实现也需要进行相应的修改，以保证程序语义一致。

需要注意的是，其他虚拟机未必维护了这些 intrinsic 的高效实现，它们可以直接使用原本的较为低效的 JDK 代码。同样，不同版本的 HotSpot 虚拟机所实现的 intrinsic 数量也大不相同。通常越新版本的 Java，其 intrinsic 数量越多。

你或许会产生这么一个疑问：为什么不直接在源代码中使用这些高效实现呢？

这是因为高效实现通常依赖于具体的 CPU 指令，而这些 CPU 指令不好在 Java 源程序中表达。再者，换了一个体系架构，说不定就没有对应的 CPU 指令，也就无法进行 intrinsic 优化了。

下面我们便来看几个具体的例子。

intrinsic 与 CPU 指令

在文章开头的例子中，`StringLatin1.indexOf` 方法将在一个字符串（byte 数组）中查找另一个字符串（byte 数组），并且返回命中时的索引值，或者 -1（未命中）。

“恰巧”的是，X86_64 体系架构的 SSE4.2 指令集就包含一条指令 `PCMPESTRQ`，让它能够在 16 字节以下的字符串中，查找另一个 16 字节以下的字符串，并且返回命中时的索引值。

因此，HotSpot 虚拟机便围绕着这一指令，开发出 X86_64 体系架构上的高效实现，并替换原本对 `StringLatin1.indexOf` 方法的调用。

另外一个例子则是整数加法的溢出处理。一般我们在做整数加法时，需要考虑结果是否会溢出，并且在溢出的情况下作出相应的处理，以保证程序的正确性。

Java 核心类库提供了一个 `Math.addExact` 方法。它将接收两个 `int` 值（或 `long` 值）作为参数，并返回这两个 `int` 值的和。当这两个 `int` 值之和溢出时，该方法将抛出 `ArithmeticException` 异常。

```
@HotSpotIntrinsicCandidate
public static int addExact(int x, int y) {
    int r = x + y;
    // HD 2-12 Overflow iff both arguments have the opposite sign of the result
    if (((x ^ r) & (y ^ r)) < 0) {
        throw new ArithmeticException("integer overflow");
    }
    return r;
}
```

在 Java 层面判断 `int` 值之和是否溢出比较费事。我们需要分别比较两个 `int` 值与它们的和的符号是否不同。如果都不同，那么我们便认为这两个 `int` 值之和溢出。对应的实现便是两个异或操作，一个与操作，以及一个比较操作。

在 X86_64 体系架构中，大部分计算指令都会更新状态寄存器（`FLAGS register`），其中就有表示指令结果是否溢出的溢出标识位（`overflow flag`）。因此，我们只需在加法指令之后比较溢出标志位，便可以知道 `int` 值之和是否溢出了。对应的伪代码如下所示：

```
public static int addExact(int x, int y) {
    int r = x + y;
    jo LABEL_OVERFLOW; // jump if overflow flag set
    return r;
LABEL_OVERFLOW:
    throw new ArithmeticException("integer overflow");
    // or deoptimize
}
```

```
}
```

最后一个例子则是 `Integer.bitCount` 方法，它将统计所输入的 `int` 值的二进制形式中有多少个 1。

```
@HotSpotIntrinsicCandidate
public static int bitCount(int i) {
    // HD, Figure 5-2
    i = i - ((i >>> 1) & 0x55555555);
    i = (i & 0x33333333) + ((i >>> 2) & 0x33333333);
    i = (i + (i >>> 4)) & 0x0f0f0f0f;
    i = i + (i >>> 8);
    i = i + (i >>> 16);
    return i & 0x3f;
}
```

我们可以看到，`Integer.bitCount` 方法的实现还是很巧妙的，但是它需要的计算步骤也比较多。在 X86_64 体系架构中，我们仅需要一条指令 `popcnt`，便可以直接统计出 `int` 值中 1 的个数。

intrinsic 与方法内联

HotSpot 虚拟机中，intrinsic 的实现方式分为两种。

一种是独立的桩程序。它既可以被解释执行器利用，直接替换对原方法的调用；也可以被即时编译器所利用，它把代表对原方法的调用的 IR 节点，替换为对这些桩程序的调用的 IR 节点。以这种形式实现的 intrinsic 比较少，主要包括 `Math` 类中的一些方法。

另一种则是特殊的编译器 IR 节点。显然，这种实现方式仅能够被即时编译器所利用。

在编译过程中，即时编译器会将原方法的调用的 IR 节点，替换成特殊的 IR 节点，并参与接下来的优化过程。最终，即时编译器的后端将根据这些特殊的 IR 节点，生成指定的 CPU 指令。大部分的 intrinsic 都是通过这种方式实现的。

这个替换过程是在方法内联时进行的。当即时编译器碰到方法调用节点时，它将查询目标方法是不是 intrinsic。

如果是，则插入相应的特殊 IR 节点；如果不是，则进行原本的内联工作。（即判断是否需要内联目标方法的方法体，并在需要内联的情况下，将目标方法的 IR 图纳入当前的编译范围之中。）

也就是说，如果方法调用的目标方法是 intrinsic，那么即时编译器会直接忽略原目标方法的

字节码，甚至根本不在乎原目标方法是否有字节码。即便是 native 方法，只要它被标记为 intrinsic，即时编译器便能够将之 " 内联 " 进来，并插入特殊的 IR 节点。

事实上，不少被标记为 intrinsic 的方法都是 native 方法。原本对这些 native 方法的调用需要经过 JNI (Java Native Interface)，其性能开销十分巨大。但是，经过即时编译器的 intrinsic 优化之后，这部分 JNI 开销便直接消失不见，并且最终的结果也十分高效。

举个例子，我们可以通过 `Thread.currentThread` 方法来获取当前线程。这是一个 native 方法，同时也是一个 HotSpot intrinsic。在 X86_64 体系架构中，R13 寄存器存放着当前线程的指针。因此，对该方法的调用将被即时编译器替换为一个特殊 IR 节点，并最终生成读取 R13 寄存器指令。

已有 intrinsic 简介

最新版本的 HotSpot 虚拟机定义了三百多个 intrinsic。

在这三百多个 intrinsic 中，有三成以上是 `Unsafe` 类的方法。不过，我们一般不会直接使用 `Unsafe` 类的方法，而是通过 `java.util.concurrent` 包来间接使用。

举个例子，`Unsafe` 类中经常会被用到的便是 `compareAndSwap` 方法 (Java 9+ 更名为 `compareAndSet` 或 `compareAndExchange` 方法)。在 X86_64 体系架构中，对这些方法的调用将被替换为 `lock cmpxchg` 指令，也就是原子性更新指令。

除了 `Unsafe` 类的方法之外，HotSpot 虚拟机中的 intrinsic 还包括下面的几种。

1. `StringBuilder` 和 `StringBuffer` 类的方法。HotSpot 虚拟机将优化利用这些方法构造字符串的方式，以尽量减少需要复制内存的情况。
2. `String` 类、`StringLatin1` 类、`StringUTF16` 类和 `Arrays` 类的方法。HotSpot 虚拟机将使用 SIMD 指令 (single instruction multiple data，即用一条指令处理多个数据) 对这些方法进行优化。举个例子，`Arrays.equals(byte[], byte[])` 方法原本是逐个字节比较，在使用了 SIMD 指令之后，可以放入 16 字节的 XMM 寄存器中 (甚至是 64 字节的 ZMM 寄存器中) 批量比较。
3. 基本类型的包装类、`Object` 类、`Math` 类、`System` 类中各个功能性方法，反射 API、`MethodHandle` 类中与调用机制相关的方法，压缩、加密相关方法。这部分 intrinsic 则比较简单，这里就不详细展开了。如果你感兴趣的，可以自行查阅资料，或者在文末留言。

如果你想知道 HotSpot 虚拟机定义的所有 intrinsic，那么你可以直接查阅 OpenJDK 代码 [2]。(该链接是 Java 12 的 intrinsic 列表。Java 8 的 intrinsic 列表可以查阅这一链接

[3]。)

总结与实践

今天我介绍了 HotSpot 虚拟机中的 intrinsic。

HotSpot 虚拟机将对标注了 `@HotSpotIntrinsicCandidate` 注解的方法的调用，替换为直接使用基于特定 CPU 指令的高效实现。这些方法我们便称之为 intrinsic。

具体来说，intrinsic 的实现有两种。一是不大常见的桩程序，可以在解释执行或者即时编译生成的代码中使用。二是特殊的 IR 节点。即时编译器将在方法内联过程中，将对 intrinsic 的调用替换为这些特殊的 IR 节点，并最终生成指定的 CPU 指令。

HotSpot 虚拟机定义了三百多个 intrinsic。其中比较特殊的有 `Unsafe` 类的方法，基本上使用 `java.util.concurrent` 包便会间接使用到 `Unsafe` 类的 intrinsic。除此之外，`String` 类和 `Arrays` 类中的 intrinsic 也比较特殊。即时编译器将为之生成非常高效的 SIMD 指令。

今天的实践环节，你可以体验一下 `Integer.bitCount` intrinsic 带来的性能提升。

```
// time java Foo
public class Foo {
    public static int bitCount(int i) {
        // HD, Figure 5-2
        i = i - ((i >>> 1) & 0x55555555);
        i = (i & 0x33333333) + ((i >>> 2) & 0x33333333);
        i = (i + (i >>> 4)) & 0x0f0f0f0f;
        i = i + (i >>> 8);
        i = i + (i >>> 16);
        return i & 0x3f;
    }
    public static void main(String[] args) {
        int sum = 0;
        for (int i = Integer.MIN_VALUE; i < Integer.MAX_VALUE; i++) {
            sum += bitCount(i); // In a second run, replace with Integer.bitCount
        }
        System.out.println(sum);
    }
}
```

[1] <http://openjdk.java.net/jeps/254> [2] <http://hg.openjdk.java.net/jdk/hs/file/46dc568d6804/src/hotspot/share/classfile/vmSymbols.hpp#l727> [3] <http://hg.openjdk.java.net/jdk8u/jdk8u/hotspot/file/2af8917ffbee/src/share/vm/classfile/vmSymbols.hpp#l647>

上一页

下一页