

My Language Is Better Than Yours

语言特征和编译优化对性能的影响和度量

周志明 (@icyfenix)

关于我.....

- 周志明 (icyfenix)
 - ✓ 目前在远光软件任开发部部门总经理。
 - ✓ 《深入理解JAVA虚拟机》和《深入理解OSGi》作者，《Java虚拟机规范 (Java SE 7) 》译者。
 - ✓ InfoQ.cn上撰写技术专栏。
 - ✓ 对编程语言、编译原理等感兴趣，目前主要使用Java，工作相关领域为企业信息系统、OSGi、工作流等。
- 很高兴与大家交流
 - ✓ weibo.com/icyfenix
 - ✓ icyfenix@gmail.com

准备分享的内容

- 语言特征对程序性能的影响
- 现代编译器优化措施对程序性能的影响
- Java语言写microbenchmark的陷阱

希望留下的观点

- “My language is better than yours” 之类的观点通常是肤浅和无意义的，“Better” 是性能和易用之间的选择，取决于你更在乎什么。
- 编译的机器码与源码有相同语意，但经过编译器优化后，可能与源码的表达方式完全不同。
- microbenchmark的结果容易产生误导，或者不能准确反映现实状况。

语言特征对程序性能的影响

C++ VS JAVA

C++ vs Java

- 从一个经典烂坑说起
 - ✓ 从初学编程的学生就开始打的口水仗，很多人支持（或反对）某种语言的最根本原因其实是“我正在使用它”。
 - ✓ 出现过无数论据，但是其中大部分都是错误的。
 - ✓ 争论过很长时间，但缺乏直接证据，缺乏有公信力的结果。
- 想一想，如果要跳这个坑，您的观点是？

一个被吐槽的例子

7.2.5 也不要过于迷信C语言

从以上3个程序证实，基本上打击了之前谭浩强主编的那个教材中所说的理论。我想，也许那个理论是SUN公司给Java做的一个广告吧！当然，本小节中Java和C语言的这样的测试是一种很极端的情况。如果C语言的程序写得非常烂，则一样会非常慢。比如再看一下下面的这两个程序是谁快？

```
int getXXX(int x, int y)
{
    return x+y;
}
```

C语言程序如下。

```
int getXXX(int x, int y)
{
    for(int i = 0; i < 100000000; i++){
        string ss = "I am so slow";
    }
    return x+y;
}
```

就上面的两个程序而言，当然还是Java比C快。因此，在处理程序开发的具体问题时也不要过分地迷信C语言快。

经常遇到的争论

- C/C++认为.....
 - ✓ Java是解释执行，比编译执行的C/C++要慢。
 - ✓ Window是C写的，Linux是C写的，甚至Java虚拟机都是C++写的，这些都是Java性能不如C/C++的侧面证据。
 - ✓ 有某权威网站的评测数据说明.....
- Java认为.....
 - ✓ 今主流的Java虚拟机涉及性能热点的代码必然不是解释执行的。
 - ✓ 选用哪门语言去实现，绝大多数都不是基于性能考虑的，好奇号火星探测器还用Java来写软件系统呢。
 - ✓ 有某权威网站的评测数据说明.....

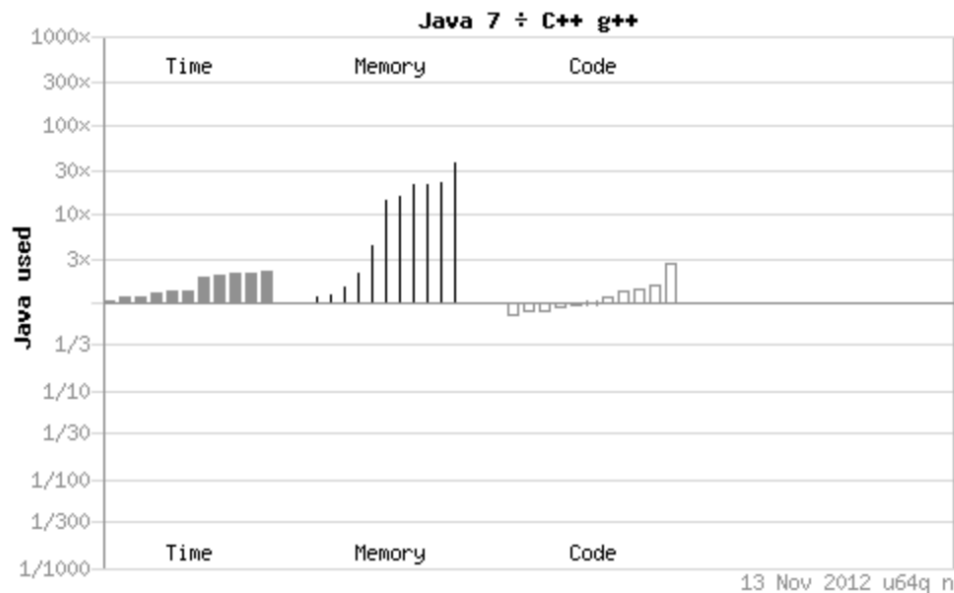
C++比Java快？

x64 Ubuntu : Intel® Q6600® quad-core
Computer Language Benchmarks Game [[More]]

- all benchmarks - Java 7
÷ C++ g++ Show

1 : Are the Java 7 programs faster? At a glance.


Each chart bar shows, for one unidentified benchmark, how much the fastest **Java 7** program *used* compared to the



These are not the only compilers and interpreters. These are not the only programs that could be written. These are **tiny examples**.

Reference: <http://shootout.alioth.debian.org>

C++不比Java快？

**JAVAWORLD**
SOLUTIONS FOR JAVA DEVELOPERS

Google™ Custom Search

Research Centers

- Core Java
- Enterprise Java
- Mobile Java
- Tools & Methods
- JavaWorld Archives

Site Resources

- Featured Articles
- News & Views
- Community
- Java Q&A
- JW Blogs
- Podcasts
- Site Map
- Newsletters
- Whitepapers
- RSS Feeds




About JavaWorld

- Advertise
- Write for JW

Performance tests show Java as fast as C++

Java has endured criticism for its laggard performance (relative to C++) since its birth, but the performance gap is closing

By Carmine Mangione, JavaWorld.com, 02/01/98

 [Print](#)  [Feedback](#)  [+ Briefcase](#)

How does the performance of Java applications compare with similar fully optimized C++ programs in theory, benchmarks, and real-world applications?

Most industry analysts make the blanket assumption that Java will always suffer performance disadvantages compared with other languages because Java was developed to allow Java programs to run on multiple platforms. You can find this assumption woven through almost all mainstream articles and opinions regarding Java and NCs in modern enterprises.

Originally published in NC World

So we decided to find out for ourselves just how much of a disadvantage there is between Java and C++, the language to which Java is most often compared. We examined the architectural components of Java and compared the performance of programs written in Java to

JW's Most Read

Recommended: Tech industry graveyard tour, 2012

JW's top 5

- JVM performance optimization, Part 3: Garbage collection
- Modern threading: A Java concurrency primer
- Maximize NIO and NIO.2 for Java application responsiveness
- Design patterns: Mogwai or gremlins?
- JavaOne gems: The Checker Framework

Featured White Papers

- Do You Really Get Classloaders?
- Pragmatic Continuous Delivery with Jenkins, Nexus, and LiveRebel
- Your Next Java Web App: Less XML, No Long Restarts, Fewer Hassles
- Coding with JRebel, Java Forever Changed
- 2012 Developer Productivity Report: Java Tools, Tech, Devs, and Data

Newsletter sign-up [View all newsletters](#)
Enterprise Java Newsletter
Stay up to date on the latest tutorials and Java

Referece: <http://www.javaworld.com/javaworld/jw-02-1998/jw-02-jperf.html>

语言特征带来的负担

- 以上的争论场景多半是得不到结论的，很少人在争论中能拿出直接论据，这个片子里，我们将从最基本的程序行为来分析这个问题。
- 目的不在于分出谁快谁慢，而在于了解为什么不同语言写出来的程序，会有不一样的运行性能。
- 笔者观点：许多基本程序行为（如访问数据、调用方法、创建对象、执行代码等等）中，Java要比C/C++有更高的执行负担，通俗地讲，就是要做更多的事情。在不公平的基础上做性能比较也是不公平的。

访问数据

```
// Java Code:
class Bar { public int i; }

public static void main(String[] args) {
    Bar bar = new Bar();
    System.out.print(bar.i);
}

// C++ Code:
class Bar { public: int i;};

int main(int argc, char** argv) {
    Bar* bar = new Bar();
    std::cout << bar->i;
    return 0;
}
```

两段代码的语意等同吗？

访问数据

- Java为安全访问数据做更多的检查
 - ✓ 空指针检查
 - ✓ 数组越界检查
 - ✓ 类型转换检查
 - ✓ 同步锁状态检查
 - ✓ 算术运算合法性检查
 - ✓

```
// Java Code:
static class Bar { public int i;}

public static void main(String[] args) {
    Bar bar = new Bar();
    System.out.println(bar.i);
}

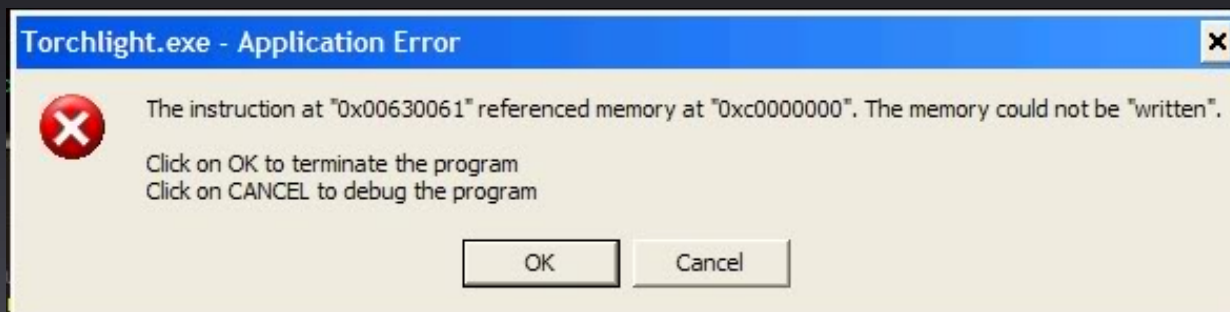
// C++ Code:
class Bar { public: int i;};

int main(int argc, char** argv) {
    Bar* bar = new Bar();
    if ( bar == NULL) {
        throw NPE();
    } else {
        std::cout << bar->i;
    }
    return 0;
}
```

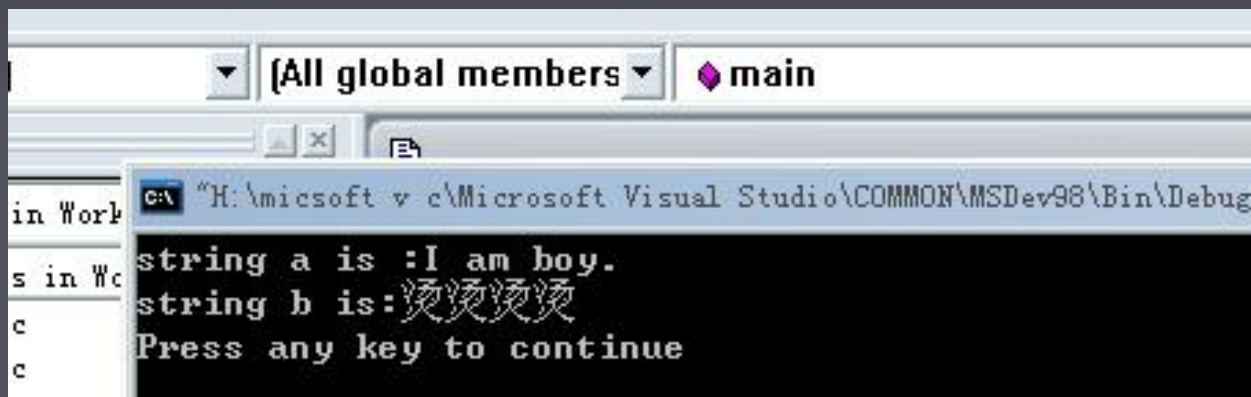
PS：这里仅仅是一个简化例子，后面会提到这段代码中，NPE检查是可以被编译器优化掉的

访问数据

- C/C++ 容忍不安全的内存访问
 - ✓ 触及操作系统的内存保护机制



- ✓ 溢出访问到预期外的数据（EXP：VC6中经典的“烫”和“屯”）



创建对象

```
// Java Code:
class Bar { public int i; }

public static void main(String[] args) {
    Bar bar = new Bar();
    System.out.print(bar.i);
}

// C++ Code:
class Bar { public: int i;};

int main(int argc, char** argv) {
    Bar* bar = new Bar();
    std::cout << bar->i;
    return 0;
}
```

两个new的语意等同吗？

创建对象

- Java中创建对象比C++更复杂。
 - ✓ Java对象的内存布局比C++的更加复杂，这些复杂性让Java可以做反射，可以快速获取HashCode，可以原生支持对象锁，可以为GC收集提供信息（如对象年龄）.....但是，它确实需要做更多的事情。
 - ✓ 堆内存的分配过程，与所选用的GC收集器有关，这增加了用户深入了解Java的对象内存模型的困难，也增加了内存分配过程的复杂度。
 - ✓ Java中只在Heap中创建对象，而C++中许多对象会选择在Stack中创建。
- 话外音：Java中创建对象要比C++更慢。

创建对象

- 一个Java对象的诞生，需要.....
 - ❶ 检查new指令参数中的符号引用（代表的类）是否已被加载、解析和初始化过。
 - ❷ 计算对象长度，根据目前使用的GC收集器，使用适当的算法（Bump The Pointer、Free List）在堆中划分空间，这个操作要保证线程安全。
 - ❸ 把申请到的空间填充零值，这步保证对象的实例字段可以不赋初始值就直接使用。
 - ❹ 将对象头的Klass字段指向该对象的类信息。
 - ❺ 填充对象头数据，例如哈希码（HashCode）、GC分代年龄、锁状态标志、线程持有的锁、偏向线程ID、偏向时间戳、所属类型信息。
 - ❻ 执行实例构造器（<init>方法）。

创建对象

- 一个Java对象的诞生，需要.....
 - ❶ 检查new指令参数中的符号引用（代表的类）是否已被加载、解析和初始化过。
 - ❷ 计算对象长度，根据目前使用的GC收集器，使用适当的算法（Bump The Pointer、Free List）在堆中划分空间，这个操作要保证线程安全。
 - ❸ 把申请到的空间填充零值，这步保证对象的实例字段可以不赋初始值就直接使用。
 - ❹ 将对象头的Klass字段指向该对象的类信息。
 - ❺ 填充对象头数据，例如哈希码（HashCode）、GC分代年龄、锁状态标志、线程持有的锁、偏向线程ID、偏向时间戳、所属类型信息。
 - ❻ 执行实例构造器（<init>方法）。

创建对象

- 一个Java对象的诞生，需要.....
 - ❶ 检查new指令参数中的符号引用（代表的类）是否已被加载、解析和初始化过。
 - ❷ 计算对象长度，根据目前使用的GC收集器，使用适当的算法（Bump The Pointer、Free List）在堆中划分空间，这个操作要保证线程安全。
 - ❸ 把申请到的空间填充零值，这步保证对象的实例字段可以不赋初始值就直接使用。
 - ❹ 将对象头的Class字段指向该对象的类信息。
 - ❺ 填充对象头数据，例如哈希码（HashCode）、GC分代年龄、锁状态标志、线程持有的锁、偏向线程ID、偏向时间戳、所属类型信息。
 - ❻ 执行实例构造器（<init>方法）。

方法调用

```
// Java Code:
class Bar { public int getValue(){ return 1; } }

public static void main(String[] args) {
    Bar bar = new Bar();
    bar = null;
    System.out.print(bar.getValue());
}

// C++ Code:
class Bar { public: int getValue(){ return 1; } }

int main(int argc, char** argv) {
    Bar* bar = new Bar();
    bar = NULL;
    std::cout << bar->getValue();
    return 0;
}
```

两段代码的效果等同吗？

方法调用

- C++版本中，方法能正确调用，因为：

执行bar = NULL，放到栈偏移0x28中；

```
movq    $0x0, -0x28(%rbp)
mov     -0x28(%rbp), %rax
mov     %rax, %rdi
callq   0x100000c5c <dyld_stub___ZN3Bar8getValueEv>
```

方法调用

- C++版本中，方法能正确调用，因为：

准备调用getValue()，将bar的引用放到rdi寄存器，作为方法的第一个参数（this指针）

```
movq    $0x0, -0x28(%rbp)
mov     -0x28(%rbp), %rax
mov     %rax, %rdi
callq   0x100000c5c <dyld_stub__ZN3Bar8getValueEv>
```

方法调用

- C++版本中，方法能正确调用，因为：

调用编译器静态生成的getValue()方法Stub

```
movq    $0x0, -0x28(%rbp)
mov     -0x28(%rbp), %rax
mov     %rax, %rdi
callq   0x10000c5c <dyld_stub__ZN3Bar8getValueEv>
```

C++中实例方法调用默认是最朴素的早期绑定（非虚方法）。方法调用期间没有访问过bar对象（下称接收者），所以它是不是NULL并没有任何影响。

方法调用

- Java版本中，形式和C++版本看起来很相似，但是有本质区别：

执行 `bar = null` ; 放到局部变量表Slot 1中

```
8: aconst_null  
9: astore 1  
13: aload_1  
14: invokevirtual #25 // Method Bar.getValue():I
```


方法调用

- Java版本中，形式和C++版本看起来很相似，但是有本质区别：

准备调用getValue()，将bar的引用放到操作数栈顶，作为方法的第一个参数（this指针）

```
8: aconst_null
9: astore 1
13: aload_1
14: invokevirtual #25 // Method Bar.getValue():I
```

方法调用

- Java版本中，形式和C++版本看起来很相似，但是有本质区别：

根据invokevirtual语意，查找getValue()方法的正确版本。

```
8: aconst_null
9: astore_1
13: aload 1
14: invokevirtual #25 // Method Bar.getValue():I
```

确定操作数栈
顶对象的类型



在该类型中查
找方法



在该类型的父
类中查找方法



进入适合的方
法入口

方法调用

- 结论是Java版本会抛出NPE异常，C++版本正常执行。
- 这个结论掩盖下的结论：大多数情况下，Java中方法调用负担更重
 - ✓ Java中的方法默认是虚方法，运行时根据接收者的vtable查找调用方法。
 - ✓ 更多的运行时安全检查，例如调用的方法是protected，需检查接收者必须是当前类或其子类；调用的方法是synchronized，需检查接收者必须是未被锁定或当前是可重入（Reentered）的。
 - ✓ 虚方法和编译优化中最重要的优化“方法内联”相矛盾。

执行代码

- 在最普通的程序语句执行时，Java要做的事情也可能会更多，以普通的步进循环为例。

```
// C++ Code:
for(int i = 0; i < 100; i++){
    // do sth;
}

// Java Code:
for(int i = 0; i < 100; i++){
    // do sth;
}
```

两段代码的效果等同吗？

执行代码

- 即使一个简单的循环，虚拟机也可能要做额外的事情：
 - ✓ 例如需要对循环进行执行计数，以便触发OSR编译.....
 - ✓ 例如需要监视后台编译线程是否完成，是否有新的循环入口.....
 - ✓ 例如.....
- 即使仅考虑JIT编译后的本地代码，不考虑解释器阶段：
 - ✓ 那来看看轮询Safepoint怎么样？
 - ✓ 每次循环都要多一次Safepoint轮询的开销。☹

Safepoint是什么？

- Safepoint就是代码中可以停下来、开始GC的位置
 - ✓ Safepoint的技术背景：ByteCode通过JIT编译为NativeCode后，虚拟机需要通过额外的信息才能知道某个对象的某个偏移位置上的数据到底是reference、int、float还是别的含义，GC收集器做可达性分析时，必须要找出其中reference。这些数据在HotSpot中称为OopMap，基于空间成本考虑，不可能在每一条指令之后都生成一份OopMap，所以只能选出一部分的代码作为Safepoint。
 - ✓ Safepoint选取原则：既不能太少以至于让GC收集器等待时间太长，也不能过于频繁以至于过分增大运行时的负荷。
 - ✓ Safepoint主要出现于：
 - 方法调用
 - 循环跳转
 - 异常跳转

结论？

- 每次循环，Java比C++都要增加额外的Safepoint轮询的开销。
- 在4个例子（访问数据、调用方法、创建对象、执行代码）中，Java都要比C++做更多的事情，这样的例子还可以再举许多，那Java的运行速度会因此而悲剧吗？



是Java？还是杯具？

现代编译器优化措施程序性能的影响

源码与实现之间的差异

源码与实现之间的差异

- 完成与源码相同语意的行为，可以有更优化的方式。
- 有一些情况下，与源码语意不一致，但只要这些情况没有发生，那实现是可以作弊的。

Safepoint轮询的实现

- 当执行ByteCode时
 - ✓ 通过dispath_table实现，HotSpot设置了3个dispath_table（active table、normal table、safept table），平时用normal table充当active table，需要暂停时替换为safept table作为active table。
- 当执行NativeCode时
 - ✓ 通过poling page实现，在每个Safepoint中生成一条访问这个poling page的指令，需要暂停时将这个内存页设置为不可读，那到Safepoint就会发生一个错误信号（SIGSEGV），进到虚拟机注册好的Handler中，再在Handler中暂停线程。

来看代码吧

```
// C++ Code:  
for(int i = 0; i < 100; i++){  
    // do sth;  
}
```

```
// Java Code:  
for(int i = 0; i < 100; i++){  
    // do sth;  
}
```

;C++ Compiled Code, gcc version 4.2.1 (LLVM 2336.11.00)

movl \$0x0,-0x24(%rbp)

mov -0x24(%rbp),%eax

jmp 0x10000097f

0x100000976: add \$0x1,%eax

0x10000097f: cmp \$0x63,%eax

jle 0x100000976

;Java Compiled Code, HotSpot Client VM build 21.0-b17

mov \$0x0,%eax

jmp 0x0189dedf

0x0189dedc: inc %eax

0x0189dedf: test %eax,0x100100

cmp \$0x64,%eax

j1 0x0189dedc

循环初值int i=0 ;

来看代码吧

// C++ Code:

```
for(int i = 0; i < 100; i++){  
    // do sth;  
}
```

// Java Code:

```
for(int i = 0; i < 100; i++){  
    // do sth;  
}
```

;C++ Compiled Code, gcc version 4.2.1 (LLVM 2336.11.00)

movl \$0x0,-0x24(%rbp)

mov -0x24(%rbp),%eax

jmp 0x10000097f

0x100000976: add \$0x1,%eax

0x10000097f: cmp \$0x63,%eax

jle 0x100000976

;Java Compiled Code, HotSpot Client VM build 21.0-b17

mov \$0x0,%eax

jmp 0x0189dedf

0x0189dedc: inc %eax

0x0189dedf: test %eax,0x100100

cmp \$0x64,%eax

j1 0x0189dedc

增加循环变量, i++ ;

来看代码吧

```
// C++ Code:  
for(int i = 0; i < 100; i++){  
    // do sth;  
}
```

```
// Java Code:  
for(int i = 0; i < 100; i++){  
    // do sth;  
}
```

;C++ Compiled Code, gcc version 4.2.1 (LLVM 2336.11.00)

```
    movl    $0x0,-0x24(%rbp)  
    mov     -0x24(%rbp),%eax  
    jmp     0x10000097f  
0x100000976: add     $0x1,%eax  
0x10000097f: cmp     $0x63,%eax  
    jle     0x100000976
```

;Java Compiled Code, HotSpot Client VM build 21.0-b17

```
    mov     $0x0,%eax  
    jmp     0x0189dedf  
0x0189dedc: inc     %eax  
0x0189dedf: test    %eax,0x100100  
    cmp     $0x64,%eax  
    jl      0x0189dedc
```

比较循环条件，根据结果跳转

来看代码吧

```
// C++ Code:  
for(int i = 0; i < 100; i++){  
    // do sth;  
}
```

```
// Java Code:  
for(int i = 0; i < 100; i++){  
    // do sth;  
}
```

;C++ Compiled Code, gcc version 4.2.1 (LLVM 2336.11.00)

```
    movl    $0x0,-0x24(%rbp)  
    mov     -0x24(%rbp),%eax  
    jmp     0x10000097f  
0x100000976: add     $0x1,%eax  
0x10000097f: cmp     $0x63,%eax  
    jle     0x100000976
```

;Java Compiled Code, HotSpot Client VM build 21.0-b17

```
    mov     $0x0,%eax  
    jmp     0x0189dedf  
0x0189dedc: inc     %eax  
0x0189dedf: test    %eax,0x100100  
    cmp     $0x64,%eax  
    jl      0x0189dedc
```

两者的差异：轮询Safepoint，被简化至只有一条test指令的程度

消除方法调用开销

- 内联是最重要最基础的优化措施
 - ✓ 可以消除调用方法调用的开销（如创建栈帧）
 - ✓ 增大其它优化措施的作用范围
 - ✓ Java中遍地都是虚方法，如果不消除这部分开销，势必对Java的性能影响很大。
- 解决虚方法和内联之间的矛盾
 - ✓ HotSpot的解决方案：激进优化（非稳定优化）
 - 类继承关系分析（Class Hierarchy Analysis）
 - 内联缓存（Inline Cache）

类继承关系分析

- CHA的作用

- ✓ 一种全程序分析手段。
- ✓ 可以提供诸如“某个接口是否有多于一种的实现，某个类是否存在子类、子类是否为抽象类等信息”之类的信息。
- ✓ 如果能够通过CHA确定调用的方法只有唯一一个版本（实践中大部分情况都是这样），那就可以不把虚方法当虚方法了，这时候称为守护内联（Guarded Inlining）。

- 不是稳定优化

- ✓ CHA是基于“当前虚拟机所加载的程序便是程序的全部”这一假设为前提的，但虚拟机可以动态扩展，随时通过ClassLoader加载更多的新类进来，当加载了新类，上一次CHA得出的结果便可能不再可靠。

内联缓存

- 听起来更不靠谱、更激进的优化策略。
- 内联缓存的依据是“即使存在多个方法版本，但许多情况下都只会调用其中一个”。
- ✓ 在未发生方法调用之前，内联缓存状态为空，当第一次调用发生后，缓存记录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，如果以后进来的每次调用的方法接收者版本都是一样的，那这个内联还可以一直用下去。如果发生了方法接收者不一致的情况，就说明程序真正使用到了虚方法的多态特性，这时候才会取消内联，查找虚方法表进行方法分派。

逆优化

- 激进优化失败时的回退策略，例如：
 - ✓ 守护内联中的Guard条件失败。
 - ✓ 内联缓存中方法版本比较失败。
- 后果：退回到解释器执行，等待JIT重新生成新的编译代码。
- 激进优化和逆优化的存在，是Java有与C++比较性能的重要进攻武器之一。

虚方法内联实例

- MicroBenchmark : 1000*1000000次加法运算 , Java版本:

```
private static final int MAX = 10000000;  
private static final int TIMES = 1000;  
  
static class Foo { int val() { return 2; } }  
  
static int calc() {  
    Foo x = new Foo();  
    int sum = 0;  
    for (int i = 0; i < MAX; i++)  
        sum += x.val();  
    return sum;  
}  
  
public static void main(String[] args) throws Exception {  
    long time = System.currentTimeMillis();  
  
    for (int i = 0; i < TIMES; i++)  
        calc();  
  
    time = System.currentTimeMillis() - time;  
    System.out.println("Run 1000 add in: " + time + " millsecs");  
}
```

虚方法内联实例

- MicroBenchmark : 1000*1000000次加法运算 , C++版本:

```
#define MAX 10000000
#define TIMES 1000

class Foo { public: virtual int val() { return 2; } };

int calc() {
    Foo* x = new Foo();
    int sum = 0;
    for (int i = 0; i < MAX; i++)
        sum += x->val();
    return sum;
}

int main(int argc, char** argv) {
    long long time = getSystemTime();

    for (int i = 0; i < TIMES; i++)
        calc();

    time = getSystemTime() - time;
    std::cout << "Run 1000 add in: " << time << " msecs" << std::endl;
}
```

虚方法内联实例

- Java版本运行结果：
 - ✓ Run 1000 add in: 15 millsecs
- C++版本运行结果：
 - ✓ Run 1000 add in: 17676 millsecs
- 结论：
 - ✓ Java掌握了天顶星科技？？！！！！



虚方法内联实例

- Java版本的JIT编译器优化过程：

```
private static final int MAX = 10000000;  
private static final int TIMES = 1000;  
  
static class Foo { int val() { return 2; } }  
  
static int calc() {  
    Foo x = new Foo();  
    int sum = 0;  
    for (int i = 0; i < MAX; i++)  
        sum += x.val();  
    return sum;  
}  
  
public static void main(String[] args) throws Exception {  
    long time = System.currentTimeMillis();  
  
    for (int i = 0; i < TIMES; i++)  
        calc();  
  
    time = System.currentTimeMillis() - time;  
    System.out.println("Run 1000 add in: " + time + " millsecs");  
}
```

1.CHA得出当前全程序内只有一个Foo类型，不存在任何子类：
int val() { return 2; } → final int val() { return 2; }

虚方法内联实例

- Java版本的JIT编译器优化过程：

```
private static final int MAX = 10000000;  
private static final int TIMES = 1000;  
  
static class Foo { int val() { return 2; } }  
  
static int calc() {  
    Foo x = new Foo();  
    int sum = 0;  
    for (int i = 0; i < MAX; i++)  
        sum += x.val();  
    return sum;  
}  
  
public static void main(String[] args) throws Exception {  
    long time = System.currentTimeMillis();  
  
    for (int i = 0; i < TIMES; i++)  
        calc();  
  
    time = System.currentTimeMillis() - time;  
    System.out.println("Run 1000 add in: " + time + " millsecs");  
}
```

2.方法内联:

sum += x.val(); → sum += 2;

虚方法内联实例

- Java版本的JIT编译器优化过程：

```
private static final int MAX = 10000000;  
private static final int TIMES = 1000;  
  
static class Foo { int val() { return 2; } }  
  
static int calc() {  
    Foo x = new Foo();  
    int sum = 0;  
    for (int i = 0; i < MAX; i++)  
        sum += x.val();  
    return sum;  
}  
  
public static void main(String[] args) throws Exception {  
    long time = System.currentTimeMillis();  
  
    for (int i = 0; i < TIMES; i++)  
        calc();  
  
    time = System.currentTimeMillis() - time;  
    System.out.println("Run 1000 add in: " + time + " millsecs");  
}
```

3.循环展开:

for (int i = 0; i < MAX; i++) sum += 2;
→ sum = MAX*2

虚方法内联实例

- Java版本的JIT编译器优化过程：

```
private static final int MAX = 10000000;  
private static final int TIMES = 1000;  
  
static class Foo { int val() { return 2; } }
```

```
static int calc() {  
    Foo x = new Foo();  
    int sum = 0;  
    for (int i = 0; i < MAX; i++)  
        sum += x.val();  
    return sum;  
}
```

4.常量折叠:

return sum → return 10000000*2 → return 20000000

```
public static void main(  
    long time = System.currentTimeMillis();  
  
    for (int i = 0; i < TIMES; i++)  
        calc();  
  
    time = System.currentTimeMillis() - time;  
    System.out.println("Run 1000 add in: " + time + " millsecs");  
}
```

优化结果的直接论据

创建栈帧

```
;Java HotSpot(TM) Server VM (build 22.0-b02-fastdebug, mixed mode)
; # {method} 'calc' '()I' in 'org/fenixsoft/Test'
```

```
0x019a9200: push    %ebp
0x019a9201: sub     $0x8,%esp          ;*synchronization monitor
                                ; - org.fenixsoft.Test.calc
0x019a9207: mov     $0x77359400,%eax
0x019a920c: add     $0x8,%esp
0x019a920f: pop     %ebp
0x019a9210: test    %eax,0x150000      ; {poll_return}
0x019a9216: ret
```

0x77359400 = 200000000
木有计算过程，直接将运算结果赋到
EAX寄存器上返回

销毁栈帧
轮询Safepoint
方法返回

消除数据访问安全检查开销

- 隐式异常检查
 - ✓ 主要应用于NPE、被零除两类异常。
 - ✓ 不会在需要检查的地方生成显示的代码，而是通过注册异常Handler来完成。如果异常不出现，则是零代价的，如果异常出现了，则需要付出很高代价（内核态/用户态切换的代价）
 - ✓ 通过收集运行数据来判断是否要进行这项优化（ImplicitNullCheckThreshold参数）
- 根据上下文进行优化
 - ✓ 上下文的语境决定是否要生成检查代码

消除对象创建开销

- 基于逃逸分析的优化
 - ✓ 什么是逃逸分析 (Escape Analysis) ?
 - 方法逃逸
 - 线程逃逸
 - ✓ 建立在逃逸分析基础上的优化措施
 - 栈上分配 (Stack Allocations)
 - 标量替换 (Scalar Replacement)
 - 锁消除 (Lock Elimination)
 -

标量替换示例

```
class Point {  
    private int x,y;  
  
    public Point(int x, int y) { this.x = x; this.y = y; }  
    public int getX() { return x; }  
    public int getY() { return y; }  
}  
  
// 优化前代码  
public int test(int x) {  
    int xx = x + 2;  
    Point p = new Point(xx, 42);  
    return p.getX();  
}
```

标量 (Scalar) : 无法再进行分解的原始类型。

聚合量 (Aggregate) : 由标量和其他聚合量组成的数据类型，对象是典型聚合量。

标量替换：如果把一个Java对象拆散，将其成员变量恢复为分散的标量，这就叫做标量替换。拆散后的变量便可以被单独分析与优化，原本的对象就无需整体分配空间了。

标量替换示例

// 优化前代码

```
public int test(int x) {  
    int xx = x + 2;  
    Point p = new Point(xx, 42);  
    return p.getX();  
}
```



// 优化1：内联Point的构造器，Inline

```
public test(int x) {  
    int xx = x + 2;  
    Point p = new_Point(); // 创建一个Point对象但不调用其构造器  
    p.x = xx;               // Point构造器内容被内联到这里  
    p.y = 42;  
    return p.x;  
}
```

标量替换示例

```
// 优化1: 内联Point的构造器, Inline
public test(int x) {
    int xx = x + 2;
    Point p = new_Point(); // 创建一个Point对象但不调用其构造器
    p.x = xx;               // Point构造器内容被内联到这里
    p.y = 42;
    return p.x;
}
```



```
// 优化2: 通过逃逸分析得知对象p不会逃逸, Scalar Replacement
public static void test(int x) {
    int xx = x + 2;
    int px = xx;    // Point消失了, 留下其成员为分散的局部变量
    int py = 42;
    return px;
}
```



```
// 优化3: 通过数据流分析得知py是完全无用的, 做Dead Code Elimination
public static void test(int x) {
    return x + 2;
}
```

度量程序片段的性能

MICROBENCHMARK的陷阱

Microbenchmark

- 什么是microbenchmark
 - ✓ 用于度量某个代码片段性能的测试代码片段。
- 你需要的是microbenchmark还是macrobenchmark?
- "microbenchmarks can be very misleading"
 - ✓ 你所测试的代码，与你所见所想的代码，可能有非常大的差异。
 - ✓ 结果容易受到干扰，可以与它实际运行时的性能有非常大的差异。

所见 ≠ 所测

- 实例：测试Java语言做除法的性能

```
// 外部传入N, 获取进行N次“除10”运算的平均性能
long start = Sys.CTM();
for( int i=0; i<N; i++ )
    int x = i/10;
return N*1000/(Sys.CTM()-start);
```

- 结果：输出每秒能进行的“除10”运算次数？

所见 ≠ 所测

- 实例：测试Java语言做除法的性能

```
// 外部传入N，获取进行N次“除10”运算的平均性能
long start = Sys.CTM();
for( int i=0; i<N; i++ )
    int x = i/10;
return N*1000/(Sys.CTM()-start);
```

- 实际上：
 1. 解释执行一段时间（假设10ms）
 2. JIT编译器发现x不会被使用，做无用代码消除优化（dead code elimination），循环没有了。
 3. 输出结果，输入多大的N，就获取多好的测试成绩

结论？

- "microbenchmarks can be very misleading"
 - ✓ 你所测试的代码，与你所见所想的代码，可能有非常大的差异。

即使编译的影响

- 实例：测试缓存循环终值是否能提高性能？

```
List<Object> list = new ArrayList<Object>();  
// 填充数据  
for (int i = 0; i < 200000; i++) {  
    list.add(new Object());  
}  
long start;  
  
start = System.nanoTime();  
// 初始化时已经计算好条件  
for (int i = 0, n = list.size(); i < n; i++) {  
}  
System.out.println("判断条件外计算：" + (System.nanoTime() - start) + " ns");  
  
start = System.nanoTime();  
// 在判断条件中计算  
for (int i = 0; i < list.size(); i++) {  
}  
System.out.println("判断条件中计算：" + (System.nanoTime() - start) + " ns");
```

即使编译的影响

- 实例：测试缓存循环终值是否能提高性能？

在Client虚拟机下：

判断条件外计算：118100 ns

判断条件中计算：1456400 ns

在Server虚拟机下：

判断条件外计算：805100 ns

判断条件中计算：2200800 ns

- 结论：Server虚拟机比Client虚拟机慢？

即使编译的影响

- 造成 “Server虚拟机比Client虚拟机慢” 的原因

```
VM option '+PrintCompilation'
```

```
169 1      java.lang.String::hashCode (67 bytes)
172 2      java.lang.String::charAt (33 bytes)
174 3      java.lang.String::indexOf (87 bytes)
179 4      java.lang.Object::<init> (1 bytes)
185 5      java.util.ArrayList::add (29 bytes)
185 6      java.util.ArrayList::ensureCapacityInternal (26 bytes)
186 1%     Client1::main @ 21 (79 bytes)
```

```
VM option '+PrintCompilation'
```

```
203 1      java.lang.String::charAt (33 bytes)
218 2      java.util.ArrayList::add (29 bytes)
218 3      java.util.ArrayList::ensureCapacityInternal (26 bytes)
221 1%     Client1::main @ 21 (79 bytes)
230 1%     made not entrant Client1::main @ -2 (79 bytes)
231 2%     Client1::main @ 51 (79 bytes)
233 2%     made not entrant Client1::main @ -2 (79 bytes)
233 3%     Client1::main @ 65 (79 bytes)
```

Server VM中OSR编译发生了3次，丢弃了其中2次

结论？

- "microbenchmarks can be very misleading"
 - ✓ 结果容易受到干扰，可以与它实际运行时的性能有非常大的差异。

总结

- 写microbenchmark的建议：
 - ✓ 注意编译器优化的结果，编译器优化后的代码，可能与你写的代码有非常大的差异。
 - ✓ 注意“预热”测试代码，保证它们进行了JIT编译。可以开启-XX:+PrintCompilation等参数监视虚拟机编译动作。
 - ✓ 注意虚拟机运行模式（-server/-client）和编译模式（OSR编译、标准编译、多层编译）的差别。
 - ✓ 注意初始化动作的影响，譬如类加载、静态初始化等。
 - ✓ 注意GC对测试的影响，可以开启-verbose:gc等参数观察虚拟机GC。

Q&A