一、避免在循环条件中使用复杂表达式

在不做编译优化的情况下，在循环中，循环条件会被反复计算，如果不使用复杂表达式，而使循环条件值不变的话，程序将会运行的更快。

例子：

```
import java.util.vector;
class cel {
    void method (vector vector) {
        for (int i = 0; i < vector.size (); i++)  // violation
            ; // ...
    }
}
```

更正：

```
class cel_fixed {
    void method (vector vector) {
        int size = vector.size ()
        for (int i = 0; i < size; i++)
            ; // ...
    }
}
```

二、为'vectors' 和 'hashtables'定义初始大小

jvm 为 vector 扩充大小的时候需要重新创建一个更大的数组，将原原先数组中的内容复制过来，最后，原先的数组再被回收。可见 vector 容量的扩大是一个颇费时间的事。
通常，默认的 10 个元素大小是不够的。你最好能准确的估计你所需要的最佳大小。

例子：

```
import java.util.vector;
public class dic {
    public void addobjects (object[] o) {
        // if length > 10, vector needs to expand
        for (int i = 0; i< o.length;i++) {
            v.add(o);   // capacity before it can add more elements.
        }
    }
    public vector v = new vector();  // no initialcapacity.
}
```

更正：
自己设定初始大小。

```
    public vector v = new vector(20);
    public hashtable hash = new hashtable(10);
```

参考资料：
dov bulka, "java performance and scalability volume 1: server-side programming techniques" addison wesley, isbn: 0-201-70429-3 pp.55 – 57

三、在 finally 块中关闭 stream

程序中使用到的资源应当被释放，以避免资源泄漏。这最好在 finally 块中去做。不管程序执行的结果如何，finally 块总是会执行的，以确保资源的正确关闭。

例子：

```
import java.io.*;
public class cs {
    public static void main (string args[]) {
        cs cs = new cs ();
        cs.method ();
    }
    public void method () {
        try {
            fileinputstream fis = new fileinputstream ("cs.java");
            int count = 0;
            while (fis.read () != -1)
                count++;
            system.out.println (count);
            fis.close ();
        } catch (filenotfoundexception e1) {
        } catch (ioexception e2) {
        }
    }
}
```

更正：
在最后一个 catch 后添加一个 finally 块

参考资料：
peter haggar: "practical java - programming language guide".
addison wesley, 2000, pp.77-79

四、使用'system.arraycopy ()'代替通过来循环复制数组

'system.arraycopy ()' 要比通过循环来复制数组快的多。

例子：

```
public class irb
{
    void method () {
        int[] array1 = new int [100];
        for (int i = 0; i < array1.length; i++) {
            array1 [i] = i;
        }
        int[] array2 = new int [100];
        for (int i = 0; i < array2.length; i++) {
            array2 [i] = array1 [i];                // violation
        }
    }
}
```

更正：

```
public class irb
```

```
{
    void method () {
        int[] array1 = new int [100];
        for (int i = 0; i < array1.length; i++) {
            array1 [i] = i;
        }
        int[] array2 = new int [100];
        system.arraycopy(array1, 0, array2, 0, 100);
    }
}
```

参考资料：
http://www.cs.cmu.edu/~jch/java/speed.html

五、让访问实例内变量的 getter/setter 方法变成"final"

简单的 getter/setter 方法应该被置成 final，这会告诉编译器，这个方法不会被重载，所以，可以变成"inlined"

例子：

```
class maf {
    public void setsize (int size) {
         _size = size;
    }
    private int _size;
}
```

更正：

```
class daf_fixed {
    final public void setsize (int size) {
         _size = size;
    }
    private int _size;
}
```

参考资料：
warren n. and bishop p. (1999), "java in practice", p. 4-5
addison-wesley, isbn 0-201-36065-9

六、避免不需要的 instanceof 操作

如果左边的对象的静态类型等于右边的，instanceof 表达式返回永远为 true。

例子：

```
public class uiso {
    public uiso () {}
}
class dog extends uiso {
    void method (dog dog, uiso u) {
        dog d = dog;
```

```
            if (d instanceof uiso) // always true.
                system.out.println("dog is a uiso");
            uiso uiso = u;
            if (uiso instanceof object) // always true.
                system.out.println("uiso is an object");
        }
    }
```

更正：
删掉不需要的 instanceof 操作。

```
    class dog extends uiso {
        void method () {
            dog d;
            system.out.println ("dog is an uiso");
            system.out.println ("uiso is an uiso");
        }
    }
```

七、避免不需要的造型操作

所有的类都是直接或者间接继承自 object。同样，所有的子类也都隐含的"等于"其父类。那么，由子类造型至父类的操作就是不必要的了。
例子：

```
    class unc {
        string _id = "unc";
    }
    class dog extends unc {
        void method () {
            dog dog = new dog ();
            unc animal = (unc)dog;  // not necessary.
            object o = (object)dog;         // not necessary.
        }
    }
```

更正：

```
    class dog extends unc {
        void method () {
            dog dog = new dog();
            unc animal = dog;
            object o = dog;
        }
    }
```

参考资料：
nigel warren, philip bishop: "java in practice - design styles and idioms for effective java". addison-wesley, 1999. pp.22-23

八、如果只是查找单个字符的话，用 charat()代替 startswith()

用一个字符作为参数调用 startswith()也会工作的很好，但从性能角度上来看，调用用 string api 无疑是错误的!

例子：

```
public class pcts {
    private void method(string s) {
        if (s.startswith("a")) { // violation
            // ...
        }
    }
}
```

更正
将'startswith()' 替换成'charat()'.

```
public class pcts {
    private void method(string s) {
        if ('a' == s.charat(0)) {
            // ...
        }
    }
}
```

参考资料：
dov bulka, "java performance and scalability volume 1: server-side programming techniques" addison wesley, isbn: 0-201-70429-3

九、使用移位操作来代替'a / b'操作

"/"是一个很"昂贵"的操作，使用移位操作将会更快更有效。

例子：

```
public class sdiv {
    public static final int num = 16;
    public void calculate(int a) {
        int div = a / 4;            // should be replaced with "a >> 2".
        int div2 = a / 8;          // should be replaced with "a >> 3".
        int temp = a / 3;
    }
}
```

更正：

```
public class sdiv {
    public static final int num = 16;
    public void calculate(int a) {
        int div = a >> 2;
        int div2 = a >> 3;
        int temp = a / 3;       // 不能转换成位移操作
    }
}
```

十、使用移位操作代替'a * b'

同上。
[i]但我个人认为，除非是在一个非常大的循环内，性能非常重要，而且你很清楚你自己在做什么，方可使用这种方法。否则提高性能所带来的程序晚读性的降低将是不合算的。

例子：

```
public class smul {
    public void calculate(int a) {
        int mul = a * 4;            // should be replaced with "a << 2".
        int mul2 = 8 * a;          // should be replaced with "a << 3".
        int temp = a * 3;
    }
}
```

更正：

```
package opt;
public class smul {
    public void calculate(int a) {
        int mul = a << 2;
        int mul2 = a << 3;
        int temp = a * 3;       // 不能转换
    }
}
```

十一、在字符串相加的时候，使用 '' 代替 " "，如果该字符串只有一个字符的话

例子：

```
public class str {
    public void method(string s) {
        string string = s + "d"  // violation.
        string = "abc" + "d"      // violation.
    }
}
```

更正：
将一个字符的字符串替换成''

```
public class str {
    public void method(string s) {
        string string = s + 'd'
        string = "abc" + 'd'
    }
}
```

十二、不要在循环中调用 synchronized(同步)方法

方法的同步需要消耗相当大的资料，在一个循环中调用它绝对不是一个好主意。

例子：

```
import java.util.vector;
public class syn {
    public synchronized void method (object o) {
    }
    private void test () {
        for (int i = 0; i < vector.size(); i++) {
            method (vector.elementat(i));    // violation
        }
    }
    private vector vector = new vector (5, 5);
}
```

更正：
不要在循环体中调用同步方法，如果必须同步的话，推荐以下方式：

```
import java.util.vector;
public class syn {
    public void method (object o) {
    }
private void test () {
    synchronized{//在一个同步块中执行非同步方法
            for (int i = 0; i < vector.size(); i++) {
                method (vector.elementat(i));
            }
        }
    }
    private vector vector = new vector (5, 5);
}
```

十三、将 try/catch 块移出循环

把 try/catch 块放入循环体内，会极大的影响性能，如果编译 jit 被关闭或者你所使用的是一个不带 jit 的 jvm，性能会将下降 21%之多!

例子：

```
import java.io.fileinputstream;
public class try {
    void method (fileinputstream fis) {
        for (int i = 0; i < size; i++) {
            try {                                          // violation
                _sum += fis.read();
            } catch (exception e) {}
        }
    }
    private int _sum;
}
```

更正：
将 try/catch 块移出循环

```
 void method (fileinputstream fis) {
        try {
            for (int i = 0; i < size; i++) {
```

```
                    _sum += fis.read();
                }
            } catch (exception e) {}
        }
```

参考资料：
peter haggar: "practical java - programming language guide".
addison wesley, 2000, pp.81 – 83

十四、对于 boolean 值，避免不必要的等式判断

将一个 boolean 值与一个 true 比较是一个恒等操作(直接返回该 boolean 变量的值). 移走对于 boolean 的不必要操作至少会带来 2 个好处：
1)代码执行的更快 (生成的字节码少了 5 个字节)；
2)代码也会更加干净 。

例子：

```
public class ueq
{
    boolean method (string string) {
        return string.endswith ("a") == true;   // violation
    }
}
```

更正：

```
class ueq_fixed
{
    boolean method (string string) {
        return string.endswith ("a");
    }
}
```

十五、对于常量字符串，用'string' 代替 'stringbuffer'

常量字符串并不需要动态改变长度。
例子：

```
public class usc {
    string method () {
        stringbuffer s = new stringbuffer ("hello");
        string t = s + "world!";
        return t;
    }
}
```

更正：
把 stringbuffer 换成 string，如果确定这个 string 不会再变的话，这将会减少运行开销提高性能。

十六、用'stringtokenizer' 代替 'indexof()' 和'substring()'

字符串的分析在很多应用中都是常见的。使用 indexof()和 substring()来分析字符串容易导致 stringindexoutofboundsexception。而使用 stringtokenizer 类来分析字符串则会容易一些，效率也会高一些。

例子：

```
public class ust {
    void parsestring(string string) {
        int index = 0;
        while ((index = string.indexof(".", index)) != -1) {
            system.out.println (string.substring(index, string.length()));
        }
    }
}
```

参考资料：
graig larman, rhett guthrie: "java 2 performance and idiom guide"
prentice hall ptr, isbn: 0-13-014260-3 pp. 282 – 283

十七、使用条件操作符替代"if (cond) return; else return;" 结构

条件操作符更加的简捷
例子：

```
public class if {
    public int method(boolean isdone) {
        if (isdone) {
            return 0;
        } else {
            return 10;
        }
    }
}
```

更正：

```
public class if {
    public int method(boolean isdone) {
        return (isdone ? 0 : 10);
    }
}
```

十八、使用条件操作符代替"if (cond) a = b; else a = c;" 结构

例子：

```
public class ifas {
    void method(boolean istrue) {
        if (istrue) {
            _value = 0;
        } else {
            _value = 1;
        }
    }
```

```
        private int _value = 0;
    }
```

更正：

```
    public class ifas {
        void method(boolean istrue) {
            _value = (istrue ? 0 : 1);       // compact expression.
        }
        private int _value = 0;
    }
```

十九、不要在循环体中实例化变量

在循环体中实例化临时变量将会增加内存消耗

例子：

```
    import java.util.vector;
    public class loop {
        void method (vector v) {
            for (int i=0;i < v.size();i++) {
                object o = new object();
                o = v.elementat(i);
            }
        }
    }
```

更正：
在循环体外定义变量，并反复使用

```
    import java.util.vector;
    public class loop {
        void method (vector v) {
            object o;
            for (int i=0;i<v.size();i++) {
                o = v.elementat(i);
            }
        }
    }
```

二十、确定 stringbuffer 的容量

stringbuffer 的构造器会创建一个默认大小(通常是 16)的字符数组。在使用中，如果超出这个大小，就会重新分配内存，创建一个更大的数组，并将原先的数组复制过来，再丢弃旧的数组。在大多数情况下，你可以在创建 stringbuffer 的时候指定大小，这样就避免了在容量不够的时候自动增长，以提高性能。

例子：

```
    public class rsbc {
        void method () {
            stringbuffer buffer = new stringbuffer(); // violation
            buffer.append ("hello");
        }
```

```
    }
```

更正：
为 stringbuffer 提供寝大小。

```
public class rsbc {
    void method () {
        stringbuffer buffer = new stringbuffer(max);
        buffer.append ("hello");
    }
    private final int max = 100;
}
```

参考资料：
dov bulka, "java performance and scalability volume 1: server-side programming techniques" addison wesley, isbn: 0-201-70429-3 p.30 – 31

二十一、尽可能的使用栈变量

如果一个变量需要经常访问，那么你就需要考虑这个变量的作用域了。static? local?还是实例变量？访问静态变量和实例变量将会比访问局部变量多耗费 2-3 个时钟周期。

例子：

```
public class usv {
    void getsum (int[] values) {
        for (int i=0; i < value.length; i++) {
            _sum += value[i];           // violation.
        }
    }
    void getsum2 (int[] values) {
        for (int i=0; i < value.length; i++) {
            _staticsum += value[i];
        }
    }
    private int _sum;
    private static int _staticsum;
}
```

更正：
如果可能，请使用局部变量作为你经常访问的变量。
你可以按下面的方法来修改 getsum()方法：

```
void getsum (int[] values) {
    int sum = _sum;  // temporary local variable.
    for (int i=0; i < value.length; i++) {
        sum += value[i];
    }
    _sum = sum;
}
```

参考资料：
peter haggar: "practical java - programming language guide".
addison wesley, 2000, pp.122 – 125

二十二、不要总是使用取反操作符(!)

取反操作符(!)降低程序的可读性，所以不要总是使用。

例子：

```
public class dun {
    boolean method (boolean a, boolean b) {
        if (!a)
            return !a;
        else
            return !b;
    }
}
```

更正：
如果可能不要使用取反操作符(!)

二十三、与一个接口 进行 instanceof 操作

基于接口的设计通常是件好事，因为它允许有不同的实现，而又保持灵活。只要可能，对一个对象进行 instanceof 操作，以判断它是否某一接口要比是否某一个类要快。

例子：

```
public class insof {
    private void method (object o) {
        if (o instanceof interfacebase) { }  // better
        if (o instanceof classbase) { }   // worse.
    }
}

class classbase {}
interface interfacebase {}
```

参考资料：
graig larman, rhett guthrie: "java 2 performance and idiom guide"
prentice hall ptr, 2000. pp.207

- 17:26
- 浏览 (26)
- 评论 (0)
- 分类: java

2010-05-12
缩略显示

Java 性能优化技巧

文章分类:Java 编程
转载：http://blog.csdn.net/kome2000/archive/2010/04/28/5537591.aspx

[size=small]在 JAVA 程序中，性能问题的大部分原因并不在于 JAVA 语言，而是程序本身。养

成良好的编码习惯非常重要，能够显著地提升程序性能。

1. 尽量使用 final 修饰符。
带有 final 修饰符的类是不可派生的。在 JAVA 核心 API 中，有许多应用 final 的例子，例如 java.lang.String。为 String 类指定 final 防止了使用者覆盖 length()方法。另外，如果一个类是 final 的，则该类所有方法都是 final 的。java 编译器会寻找机会内联（inline）所有的 final 方法（这和具体的编译器实现有关）。此举能够使性能平均提高 50%。

2.尽量重用对象。
特别是 String 对象的使用中，出现字符串连接情况时应使用 StringBuffer 代替，由于系统不仅要花时间生成对象，以后可能还需要花时间对这些对象进行垃圾回收和处理。因此生成过多的对象将会给程序的性能带来很大的影响。

3. 尽量使用局部变量。
调用方法时传递的参数以及在调用中创建的临时变量都保存在栈（Stack）中，速度较快。其他变量，如静态变量，实例变量等，都在堆（Heap）中创建，速度较慢。

4.不要重复初始化变量。
默认情况下，调用类的构造函数时，java 会把变量初始化成确定的值，所有的对象被设置成 null，整数变量设置成 0，float 和 double 变量设置成 0.0，逻辑值设置成 false。当一个类从另一个类派生时，这一点尤其应该注意，因为用 new 关键字创建一个对象时，构造函数链中的所有构造函数都会被自动调用。
这里有个注意，给成员变量设置初始值但需要调用其他方法的时候，最好放在一个方法比如 initXXX()中，因为直接调用某方法赋值可能会因为类尚未初始化而抛空指针异常，public int state = this.getState();

5.在 java+Oracle 的应用系统开发中，java 中内嵌的 SQL 语言应尽量使用大写形式，以减少 Oracle 解析器的解析负担。

6.java 编程过程中，进行数据库连接，I/O 流操作，在使用完毕后，及时关闭以释放资源。因为对这些大对象的操作会造成系统大的开销。

7.过分的创建对象会消耗系统的大量内存，严重时，会导致内存泄漏，因此，保证过期的对象的及时回收具有重要意义。
JVM 的 GC 并非十分智能，因此建议在对象使用完毕后，手动设置成 null。

8.在使用同步机制时，应尽量使用方法同步代替代码块同步。

9.尽量减少对变量的重复计算。
比如

```
for(int i=0;i<list.size();i++)
```

应修改为

```
for(int i=0,len=list.size();i<len;i++)
```

10. 采用在需要的时候才开始创建的策略。
例如：

```
String str="abc";
if(i==1){ list.add(str);}
```

应修改为：

```
if(i==1){String str="abc"; list.add(str);}
```

11.慎用异常，异常对性能不利。
抛出异常首先要创建一个新的对象。Throwable 接口的构造函数调用名为 fillInStackTrace()的本地方法，fillInStackTrace()方法检查栈，收集调用跟踪信息。只要有异常被抛出，VM 就必须调整调用栈，因为在处理过程中创建了一个新的对象。
异常只能用于错误处理，不应该用来控制程序流程。

12.不要在循环中使用 Try/Catch 语句，应把 Try/Catch 放在循环最外层。
Error 是获取系统错误的类，或者说是虚拟机错误的类。不是所有的错误 Exception 都能获取到的，虚拟机报错 Exception 就获取不到，必须用 Error 获取。

13.通过 StringBuffer 的构造函数来设定他的初始化容量，可以明显提升性能。
StringBuffer 的默认容量为 16，当 StringBuffer 的容量达到最大容量时，她会将自身容量增加到当前的 2 倍+2，也就是 2*n+2。无论何时，只要 StringBuffer 到达她的最大容量，她就不得不创建一个新的对象数组，然后复制旧的对象数组，这会浪费很多时间。所以给 StringBuffer 设置一个合理的初始化容量值，是很有必要的！

14.合理使用 java.util.Vector。
Vector 与 StringBuffer 类似，每次扩展容量时，所有现有元素都要赋值到新的存储空间中。Vector 的默认存储能力为 10 个元素，扩容加倍。
vector.add(index,obj) 这个方法可以将元素 obj 插入到 index 位置，但 index 以及之后的元素依次都要向下移动一个位置（将其索引加 1）。 除非必要，否则对性能不利。
同样规则适用于 remove(int index)方法，移除此向量中指定位置的元素。将所有后续元素左移（将其索引减 1）。返回此向量中移除的元素。所以删除 vector 最后一个元素要比删除第 1 个元素开销低很多。删除所有元素最好用 removeAllElements()方法。
如果要删除 vector 里的一个元素可以使用 vector.remove(obj)；而不必自己检索元素位置，再删除，如 int index = indexOf（obj）;vector.remove(index)；

15.当复制大量数据时，使用 System.arraycopy();

16.代码重构，增加代码的可读性。

17.不用 new 关键字创建对象的实例。
用 new 关键词创建类的实例时，构造函数链中的所有构造函数都会被自动调用。但如果一个对象实现了 Cloneable 接口，我们可以调用她的 clone() 方法。clone()方法不会调用任何类构造函数。
下面是 Factory 模式的一个典型实现。

```
public static Credit getNewCredit()
{
    return new Credit();
}
```

改进后的代码使用 clone() 方法，

```
private static Credit BaseCredit = new Credit();
public static Credit getNewCredit()
{
    return (Credit)BaseCredit.clone();
}
```

18. 乘除法如果可以使用位移，应尽量使用位移，但最好加上注释，因为位移操作不直观，难于理解。

19.不要将数组声明为：public static final。

20.HaspMap 的遍历。

```
Map<String, String[]> paraMap = new HashMap<String, String[]>();
for( Entry<String, String[]> entry : paraMap.entrySet() )
```

```
    {
        String appFieldDefId = entry.getKey();
        String[] values = entry.getValue();
    }
```

利用散列值取出相应的 Entry 做比较得到结果，取得 entry 的值之后直接取 key 和 value。

21.array(数组)和 ArrayList 的使用。
array 数组效率最高，但容量固定，无法动态改变，ArrayList 容量可以动态增长，但牺牲了效率。

22.单线程应尽量使用 HashMap, ArrayList,除非必要，否则不推荐使用 HashTable,Vector，她们使用了同步机制，而降低了性能。

23.StringBuffer,StringBuilder 的区别在于：java.lang.StringBuffer 线程安全的可变字符序列。一个类似于 String 的字符串缓冲区，但不能修改。StringBuilder 与该类相比，通常应该优先使用 StringBuilder 类，因为她支持所有相同的操作，但由于她不执行同步，所以速度更快。为了获得更好的性能，在构造 StringBuffer 或 StringBuilder 时应尽量指定她的容量。当然如果不超过 16 个字符时就不用了。
相同情况下，使用 StringBuilder 比使用 StringBuffer 仅能获得 10%~15%的性能提升，但却要冒多线程不安全的风险。综合考虑还是建议使用 StringBuffer。

24. 尽量使用基本数据类型代替对象。

25.用简单的数值计算代替复杂的函数计算，比如查表方式解决三角函数问题。

26.使用具体类比使用接口效率高，但结构弹性降低了，但现代 IDE 都可以解决这个问题。

27.考虑使用静态方法，
如果你没有必要去访问对象的外部，那么就使你的方法成为静态方法。她会被更快地调用，因为她不需要一个虚拟函数导向表。这同事也是一个很好的实践，因为她告诉你如何区分方法的性质，调用这个方法不会改变对象的状态。

28.应尽可能避免使用内在的 GET,SET 方法。
android 编程中，虚方法的调用会产生很多代价，比实例属性查询的代价还要多。我们应该在外包调用的时候才使用 get，set 方法，但在内部调用的时候，应该直接调用。

29. 避免枚举，浮点数的使用。

30.二维数组比一维数组占用更多的内存空间，大概是 10 倍计算。

31.SQLite 数据库读取整张表的全部数据很快，但有条件的查询就要耗时 30-50MS,大家做这方面的时候要注意，尽量少用，尤其是嵌套查找！ [/size][align=left][/align]

《java 解惑》转

文章分类:Java 编程
转载于:http://jiangzhengjun.javaeye.com/blog/652623

数值表达式

1. 奇偶判断

不要使用 i % 2 == 1 来判断是否是奇数，因为 i 为负奇数时不成立，请使用 i % 2 != 0 来判断是否是奇数，或使用

高效式 (i & 1) != 0 来判断。

2. 小数精确计算

```
System.out.println(2.00 -1.10);//0.8999999999999999
```

上面的计算出的结果不是 0.9，而是一连串的小数。问题在于 1.1 这个数字不能被精确表示为一个 double，因此它被表

示为最接近它的 double 值，该程序从 2 中减去的就是这个值，但这个计算的结果并不是最接近 0.9 的 double 值。

一般地说，问题在于并不是所有的小数都可以用二进制浮点数精确表示。

二进制浮点对于货币计算是非常不适合的，因为它不可能将 1.0 表示成 10 的其他任何负次幂。

解决问题的第一种方式是使用货币的最小单位（分）来表示：

```
System.out.println(200-110);//90
```

第二种方式是使用 BigDecimal，但一定要用 BigDecimal(String)构造器，而千万不要用 BigDecimal(double)来构造（也不能将 float 或 double 型转换成 String 再来使用 BigDecimal(String)来构造，因为在将 float 或 double 转换成 String 时精度已丢失）。
例如 new BigDecimal(0.1)，
它将返回一个 BigDecimal，
也即 0.1000000000000000055511151231257827021181583404541015625，
正确使用 BigDecimal，程序就可以打印出我们所期

望的结果 0.9：

```
System.out.println(new BigDecimal("2.0").subtract(new BigDecimal("1.10")));// 0.9
```

另外，如果要比较两个浮点数的大小，要使用 BigDecimal 的 compareTo 方法。

3. int 整数相乘溢出

我们计算一天中的微秒数：

```
long microsPerDay = 24 * 60 * 60 * 1000 * 1000;// 正确结果应为：86400000000
System.out.println(microsPerDay);// 实际上为：500654080
```

问题在于计算过程中溢出了。这个计算式完全是以 int 运算来执行的，并且只有在运算完成之后，其结果才被提升为 long，而此时已经太迟：计算已经溢出。
解决方法使计算表达式的第一个因子明确为 long 型，这样可以强制表达式中所有的后续计算都用 long 运算来完成，这样结果就不会溢出：

```
long microsPerDay = 24L * 60 * 60 * 1000 * 1000;
```

4. 负的十六进制与八进制字面常量

“数字字面常量”的类型都是 int 型，而不管他们是几进制，所以“2147483648”、“0x180000000（十六进制，共 33 位，所以超过了整数的取值范围）”字面常量是错误的，编译时会报超过 int 的取值范围了，所以要确定以 long 来表示“2147483648L”“0x180000000L”。

十进制字面常量只有一个特性，即所有的十进制字面常量都是正数，如果想写一个负的十进制，则需要在正的十进制

字面常量前加上“-”即可。

十六进制或八进制字面常量可就不一定是正数或负数，是正还是负，则要根据当前情况看：如果十六进制和八进制字

面常量的最高位被设置成了 1，那么它们就是负数：

```
System.out.println(0x80);//128
//0x81 看作是 int 型，最高位(第 32 位)为 0，所以是正数
System.out.println(0x81);//129
System.out.println(0x8001);//32769
System.out.println(0x70000001);//1879048193
//字面量 0x80000001 为 int 型，最高位(第 32 位)为 1，所以是负数
System.out.println(0x80000001);//-2147483647
//字面量 0x80000001L 强制转为 long 型，最高位（第 64 位）为 0，所以是正数
System.out.println(0x80000001L);//2147483649
//最小 int 型
System.out.println(0x80000000);//-2147483648
//只要超过 32 位，就需要在字面常量后加 L 强转 long，否则编译时出错
System.out.println(0x8000000000000000L);//-9223372036854775808
```

从上面可以看出，十六进制的字面常量表示的是 int 型，如果超过 32 位，则需要在后面加“L”，否则编译过不过。如果为 32，则为负 int 正数，超过 32 位，则为 long 型，但需明确指定为 long。

```
System.out.println(Long.toHexString(0x100000000L + 0xcafebabe));// cafebabe
```

结果为什么不是 0x1cafebabe？该程序执行的加法是一个混合类型的计算：左操作数是 long 型，而右操作数是 int 类型。为了执行该计算，Java 将 int 类型的数值用拓宽原生类型转换提升为 long 类型，然后对两个 long 类型数值相加。因为 int 是有符号的整数类型，所以这个转换执行的是符号扩展。

这个加法的右操作数 0xcafebabe 为 32 位，将被提升为 long 类型的数值 0xffffffffcafebabeL，之后这个数值加上了左操

作 0x100000000L。当视为 int 类型时，经过符号扩展之后的右操作数的高 32 位是-1，而左操作数的第 32 位是 1，两个数

值相加得到了 0：

```
  0x 0xffffffffcafebabeL
+0x 0000000100000000L
----------------------------
0x 00000000cafebabeL
```

如果要得到正确的结果 0x1cafebabe，则需在第二个操作数组后加上“L”明确看作是正的 long 型即可，此时相加时拓

展符号位就为 0：

```
System.out.println(Long.toHexString(0x100000000L + 0xcafebabeL));// 1cafebabe
```

5. 窄数字类型提升至宽类型时使用符号位扩展还是零扩展

```
System.out.println((int)(char)(byte)-1);// 65535
```

结果为什么是 65535 而不是-1？

窄的整型转换成较宽的整型时符号扩展规则：如果最初的数值类型是有符号的，那么就执行符号扩展（即如果符号位

为 1，则扩展为 1，如果为零，则扩展为 0）；如果它是 char，那么不管它将要被提升成什么类型，都执行零扩展。

了解上面的规则后，我们再来看看迷题：因为 byte 是有符号的类型，所以在将 byte 数值-1（二进制为：11111111）提

升到 char 时，会发生符号位扩展，又符号位为 1，所以就补 8 个 1，最后为 16 个 1；然后从 char 到 int 的提升时，由于是

char 型提升到其他类型，所以采用零扩展而不是符号扩展，结果 int 数值就成了 65535。

如果将一个 char 数值 c 转型为一个宽度更宽的类型时，只是以零来扩展，但如果清晰表达以零扩展的意图，则可以考虑

使用一个位掩码：

```
int i = c & 0xffff;//实质上等同于：int i = c ;
```

如果将一个 char 数值 c 转型为一个宽度更宽的整型，并且希望有符号扩展，那么就先将 char 转型为一个 short，它与

char 上个具有同样的宽度，但是它是有符号的：

```
int i = (short)c;
```

如果将一个 byte 数值 b 转型为一个 char，并且不希望有符号扩展，那么必须使用一个位掩码来限制它：

```
char c = (char)(b & 0xff);// char c = (char) b;为有符号扩展
```

6. ((byte)0x90 == 0x90)?

答案是不等的，尽管外表看起来是成立的，但是它却等于 false。为了比较 byte 数值(byte)0x90 和 int 数值 0x90，Java

通过拓宽原生类型将 byte 提升为 int，然后比较这两个 int 数值。因为 byte 是一个有符号类型，所以这个转换执行的是

符号扩展，将负的 byte 数值提升为了在数字上相等的 int 值（10010000◊11111111111111111111111 10010000）。在本例中，该转换将(byte)0x90 提升为 int 数值-112，它不等于 int 数值的 0x90，即+144。

解决办法：使用一个屏蔽码来消除符号扩展的影响，从而将 byte 转型为 int。

```
((byte)0x90 & 0xff)== 0x90
```

7. 三元表达式（?:）

```
char x = 'X';
int i = 0;
System.out.println(true ? x : 0);// X
System.out.println(false ? i : x);// 88
```

条件表达式结果类型的规则：
（1） 如果第二个和第三个操作数具有相同的类型，那么它就是条件表达式的类型。
（2） 如果一个操作的类型是 T，T 表示 byte、short 或 char，而另一个操作数是一个 int 类型的“字面常量”，并且

它的值可以用类型 T 表示，那条件表达式的类型就是 T。
（3） 否则，将对操作数类型进行提升，而条件表达式的类型就是第二个和第三个操作被提升之后的类型。

现来使用以上规则解上面的迷题，第一个表达式符合第二条规则：一个操作数的类型是 char，另一个的类型是字面常

量为 0 的 int 型，但 0 可以表示成 char，所以最终返回类型以 char 类型为准；第二个表达式符

合第三条规则：因为 i 为 int

型变量，而 x 又为 char 型变量，所以会先将 x 提升至 int 型，所以最后的结果类型为 int 型，但如果将 i 定义成 final 时，

则返回结果类型为 char，则此时符合第二条规则，因为 final 类型的变量在编译时就使用"字面常量 0"来替换三元表

达式了：

```
final int i = 0;
System.out.println(false ? i : x);// X
```

在 JDK1.4 版本或之前，条件操作符 ?: 中，当第二个和延续三个操作数是引用类型时，条件操作符要求它们其中一个

必须是另一个的子类型，那怕它们有同一个父类也不行：

```
public class T {
 public static void main(String[] args) {
  System.out.println(f());
 }
 public static T f() {
  // !!1.4 不能编译，但 1.5 可以
  // !!return true?new T1():new T2();
  return true ? (T) new T1() : new T2();// T1
 }
}

class T1 extends T {
 public String toString() {
  return "T1";
 }
}

class T2 extends T {
 public String toString() {
  return "T2";
 }
}
```

在 5.0 或以上版本中，条件操作符在延续二个和第三个操作数是引用类型时总是合法的。其结果类型是这两种类型的最

小公共超类。公共超类总是存在的，因为 Object 是每一个对象类型的超类型，上面的最小公共超类是 T，所以能编译。

在 JAVA 程序中，性能问题的大部分原因并不在于 JAVA 语言，而是程序本身。养成良好的编码习惯非常重要，能够显著地提升程序性能。

1. 尽量使用 final 修饰符。

带有 **final** 修饰符的类是不可派生的。在 **JAVA** 核心 **API** 中，有许多应用 **final** 的例子，例如 **java.lang.String**。为 **String** 类指定 **final** 防止了使用者覆盖 **length()**方法。另外，如果一个类是 **final** 的，则该类所有方法都是 **final** 的。**java** 编译器会寻找机会内联（**inline**）所有的 **final** 方法（这和具体的编译器实现有关）。此举能够使性能平均提高 **50%**。

2.尽量重用对象。

特别是 **String** 对象的使用中，出现字符串连接情况时应使用 **StringBuffer** 代替，由于系统不仅要花时间生成对象，以后可能还需要花时间对这些对象进行垃圾回收和处理。因此生成过多的对象将会给程序的性能带来很大的影响。

3. 尽量使用局部变量。

调用方法时传递的参数以及在调用中创建的临时变量都保存在栈（**Stack**）中，速度较快。其他变量，如静态变量，实例变量等，都在堆（**Heap**）中创建，速度较慢。

4.不要重复初始化变量。

默认情况下，调用类的构造函数时，**java** 会把变量初始化成确定的值，所有的对象被设置成 **null**，整数变量设置成 **0**，**float** 和 **double** 变量设置成 **0.0**，逻辑值设置成 **false**。当一个类从另一个类派生时，这一点尤其应该注意，因为用 **new** 关键字创建一个对象时，构造函数链中的所有构造函数都会被自动调用。

这里有个注意，给成员变量设置初始值但需要调用其他方法的时候，最好放在一个方法比如 **initXXX()**中，因为直接调用某方法赋值可能会因为类尚未初始化而抛空指针异常，**public int state = this.getState();**

5.在 **java+Oracle** 的应用系统开发中，**java** 中内嵌的 **SQL** 语言应尽量使用大写形式，以减少 **Oracle** 解析器的解析负担。

6.java 编程过程中，进行数据库连接，**I/O** 流操作，在使用完毕后，及时关闭以释放资源。因为对这些大对象的操作会造成系统大的开销。

7.过分的创建对象会消耗系统的大量内存，严重时，会导致内存泄漏，因此，保证过期的对象的及时回收具有重要意义。

JVM 的 **GC** 并非十分智能，因此建议在对象使用完毕后，手动设置成 **null**。

8.在使用同步机制时，应尽量使用方法同步代替代码块同步。

9.尽量减少对变量的重复计算。

比如

Java 代码

```
1.  for(int i=0;i<list.size();i++)
```

应修改为

```
Java 代码
1.  for(int i=0,len=list.size();i<len;i++)
```

10. 采用在需要的时候才开始创建的策略。
例如：

```
Java 代码
1.  String str="abc";
2.  if(i==1){ list.add(str);}
```

应修改为：

```
Java 代码
1.  if(i==1){String str="abc"; list.add(str);}
```

11.慎用异常，异常对性能不利。
抛出异常首先要创建一个新的对象。**Throwable** 接口的构造函数调用名为 **fillInStackTrace()**的本地方法，**fillInStackTrace()**方法检查栈，收集调用跟踪信息。只要有异常被抛出，**VM** 就必须调整调用栈，因为在处理过程中创建了一个新的对象。
异常只能用于错误处理，不应该用来控制程序流程。

12.不要在循环中使用 **Try/Catch** 语句，应把 **Try/Catch** 放在循环最外层。
Error 是获取系统错误的类，或者说是虚拟机错误的类。不是所有的错误 **Exception** 都能获取到的，虚拟机报错 **Exception** 就获取不到，必须用 **Error** 获取。

13.通过 **StringBuffer** 的构造函数来设定他的初始化容量，可以明显提升性能。
StringBuffer 的默认容量为 **16**，当 **StringBuffer** 的容量达到最大容量时，她会将自身容量增加到当前的 **2** 倍**+2**，也就是 **2*n+2**。无论何时，只要 **StringBuffer** 到达她的最大容量，她就不得不创建一个新的对象数组，然后复制旧的对象数组，这会浪费很多时间。所以给 **StringBuffer** 设置一个合理的初始化容量值，是很有必要的！

14.合理使用 **java.util.Vector**。
Vector 与 **StringBuffer** 类似，每次扩展容量时，所有现有元素都要赋值到新的存储空间中。**Vector** 的默认存储能力为 **10** 个元素，扩容加倍。

vector.add(index,obj) 这个方法可以将元素 **obj** 插入到 **index** 位置，但 **index** 以及之后的元素依次都要向下移动一个位置（将其索引加 **1**）。 除非必要，否则对性能不利。
同样规则适用于 **remove(int index)**方法，移除此向量中指定位置的元素。将所有后续元素左移（将其索引减 **1**）。返回此向量中移除的元素。所以删除 **vector** 最后一个元素要比删除第 **1** 个元素开销低很多。删除所有元素最好用 **removeAllElements()**方法。
如果要删除 **vector** 里的一个元素可以使用 **vector.remove(obj)**；而不必自己检索元素位置，再删除，如 **int index = indexOf（obj）;vector.remove(index)**；

15.当复制大量数据时，使用 **System.arraycopy();**

16.代码重构，增加代码的可读性。

17.不用 **new** 关键字创建对象的实例。
用 **new** 关键词创建类的实例时，构造函数链中的所有构造函数都会被自动调用。但如果一个对象实现了 **Cloneable** 接口，我们可以调用她的 **clone()** 方法。**clone()**方法不会调用任何类构造函数。
下面是 **Factory** 模式的一个典型实现。

Java 代码

```
1.  public static Credit getNewCredit()
2.  {
3.      return new Credit();
4.  }
```

改进后的代码使用 **clone()** 方法，

Java 代码

```
1.  private static Credit BaseCredit = new Credit();
2.  public static Credit getNewCredit()
3.  {
4.      return (Credit)BaseCredit.clone();
5.  }
```

18. 乘除法如果可以使用位移，应尽量使用位移，但最好加上注释，因为位移操作不直观，难于理解。

19.不要将数组声明为：**public static final**。

20.HaspMap 的遍历。

Java 代码

```
1.  Map<String, String[]> paraMap = new HashMap<String, String[]>();
2.  for( Entry<String, String[]> entry : paraMap.entrySet() )
```

```
3.  {
4.      String appFieldDefId = entry.getKey();
5.      String[] values = entry.getValue();
6.  }
```

利用散列值取出相应的 **Entry** 做比较得到结果，取得 **entry** 的值之后直接取 **key** 和 **value**。

21.array(数组)和 ArrayList 的使用。
array 数组效率最高，但容量固定，无法动态改变，**ArrayList** 容量可以动态增长，但牺牲了效率。

22.单线程应尽量使用 HashMap, ArrayList,除非必要，否则不推荐使用 **HashTable,Vector**，她们使用了同步机制，而降低了性能。

23.StringBuffer,StringBuilder 的区别在于：
java.lang.StringBuffer 线程安全的可变字符序列。一个类似于 **String** 的字符串缓冲区，但不能修改。**StringBuilder** 与该类相比，通常应该优先使用 **StringBuilder** 类，因为她支持所有相同的操作，但由于她不执行同步，所以速度更快。为了获得更好的性能，在构造 **StringBuffer** 或 **StringBuilder** 时应尽量指定她的容量。当然如果不超过 **16** 个字符时就不用了。
相同情况下，使用 **StringBuilder** 比使用 **StringBuffer** 仅能获得 **10%~15%**的性能提升，但却要冒多线程不安全的风险。综合考虑还是建议使用 **StringBuffer**。

24. 尽量使用基本数据类型代替对象。

25.用简单的数值计算代替复杂的函数计算，比如查表方式解决三角函数问题。

26.使用具体类比使用接口效率高，但结构弹性降低了，但现代 **IDE** 都可以解决这个问题。

27.考虑使用静态方法，
如果你没有必要去访问对象的外部，那么就使你的方法成为静态方法。她会被更快地调用，因为她不需要一个虚拟函数导向表。这同事也是一个很好的实践，因为她告诉你如何区分方法的性质，调用这个方法不会改变对象的状态。

28.应尽可能避免使用内在的 **GET,SET** 方法。
android 编程中，虚方法的调用会产生很多代价，比实例属性查询的代价还要多。我们应该在外包调用的时候才使用 **get**，**set** 方法，但在内部调用的时候，应该直接调用。

29. 避免枚举，浮点数的使用。

30.二维数组比一维数组占用更多的内存空间，大概是 **10** 倍计算。

31.SQLite 数据库读取整张表的全部数据很快，但有条件的查询就要耗时 **30-50MS,**大家做这方面的时候要注意，尽量少用，尤其是嵌套查找！