35 个 Java 代码性能优化总结

来源：五月的仓颉

代码优化，一个很重要的课题。可能有些人觉得没用，一些细小的地方有什么好修改的，改与不改对于代码的运行效率有什么影响呢？这个问题我是这么考虑的，就像大海里面的鲸鱼一样，它吃一条小虾米有用吗？没用，但是，吃的小虾米一多之后，鲸鱼就被喂饱了。代码优化也是一样，如果项目着眼于尽快无 BUG 上线，那么此时可以抓大放小，代码的细节可以不精打细磨；但是如果有足够的时间开发、维护代码，这时候就必须考虑每个可以优化的细节了，一个一个细小的优化点累积起来，对于代码的运行效率绝对是有提升的。

代码优化的目标是：

1、减小代码的体积

2、提高代码运行的效率

代码优化细节

1、尽量指定类、方法的 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，整个类都是 final 的。为类指定 final 修饰符可以让类不可以被继承，为方法指定 final 修饰符可以让方法不可以被重写。如果指定了一个类为 final，则该类所有的方法都是 final 的。Java 编译器会寻找机会内联所有的 final 方法，内联对于提升 Java 运行效率作用重大，具体参见 Java 运行期优化。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，出现字符串连接时应该使用 StringBuilder/StringBuffer 代替。由于 Java 虚拟机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、尽可能使用局部变量

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中速度较快，其他变量，如静态变量、实例变量等，都在堆中创建，速度较慢。大讲台，混合式自适应 IT 职业教育开创者。另外，栈中创建的变量，随着方法的运行结束，这些内容就没了，不需要额外的垃圾回收。

4、及时关闭流

Java 编程过程中，进行数据库连接、I/O 流操作时务必小心，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，稍有不慎，将会导致严重的后果。

5、尽量减少对变量的重复计算

明确一个概念，对方法的调用，即使方法中只有一句语句，也是有消耗的，包括创建栈帧、调用方法时保护现场、调用方法完毕时恢复现场等。所以例如下面的操作：

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

{...}
```

建议替换为：

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

{...}
```

这样，在 list.size()很大的时候，就减少了很多的消耗

6、尽量采用懒加载的策略，即在需要的时候才创建

例如：

```
String str = "aaa";if (i == 1)

{

list.add(str);

}
```

建议替换为：

```
if (i == 1)

{

String str = "aaa";

list.add(str);
```

```
}
```

7、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()的本地同步方法，fillInStackTrace()方法检查堆栈，收集调用跟踪信息。只要有异常被抛出，Java 虚拟机就必须调整调用堆栈，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。

8、不要在循环中使用 try…catch…，应该把其放在最外层

除非不得已。如果毫无理由地这么写了，只要你的领导资深一点、有强迫症一点，八成就要骂你为什么写出这种垃圾代码来了

9、如果能估计到待添加的内容长度，为底层以数组方式实现的集合、工具类指定初始长度

比如 ArrayList、LinkedLlist、StringBuilder、StringBuffer、HashMap、HashSet 等等，以 StringBuilder 为例：

（1）StringBuilder()　　// 默认分配 16 个字符的空间

（2）StringBuilder(int size)　　// 默认分配 size 个字符的空间

（3）StringBuilder(String str)　// 默认分配 16 个字符+str.length()个字符空间

可以通过类（这里指的不仅仅是上面的 StringBuilder）的来设定它的初始化容量，这样可以明显地提升性能。比如 StringBuilder 吧，length 表示当前的 StringBuilder 能保持的字符数量。因为当 StringBuilder 达到最大容量的时候，它会将自身容量增加到当前的 2 倍再加 2，无论何时只要 StringBuilder 达到它的最大容量，它就不得不创建一个新的字符数组然后将旧的字符数组内容拷贝到新字符数组中—-这是十分耗费性能的一个操作。试想，如果能预估到字符数组中大概要存放 5000 个字符而不指定长度，最接近 5000 的 2 次幂是 4096，每次扩容加的 2 不管，那么：

（1）在 4096 的基础上，再申请 8194 个大小的字符数组，加起来相当于一次申请了 12290 个大小的字符数组，如果一开始能指定 5000 个大小的字符数组，就节省了一倍以上的空间

（2）把原来的 4096 个字符拷贝到新的的字符数组中去

这样，既浪费内存空间又降低代码运行效率。所以，给底层以数组实现的集合、工具类设置一个合理的初始化容量是错不了的，这会带来立竿见影的效果。但是，注意，像 HashMap 这种是以数组+链表实现的集合，别把初始大小和你估计的大小设置得一样，因为一个 tabl

e 上只连接一个对象的可能性几乎为 0。初始大小建议设置为 2 的 N 次幂，如果能估计到有 2000 个元素，设置成 new HashMap(128)、new HashMap(256)都可以。

10、当复制大量数据时，使用 System.arraycopy()命令

11、乘法和除法使用移位操作

例如：

```
for (val = 0; val < 100000; val += 5)

{

a = val * 8;

b = val / 2;

}
```

用移位操作可以极大地提高性能，因为在计算机底层，对位的操作是最方便、最快的，因此建议修改为：

```
for (val = 0; val < 100000; val += 5)

{

a = val << 3;

b = val >> 1;

}
```

移位操作虽然快，但是可能会使代码不太好理解，因此最好加上相应的注释。

12、循环内不要不断创建对象引用

例如：

```
for (int i = 1; i <= count; i++)

{

Object obj = new Object();
```

```
}
```

这种做法会导致内存中有 count 份 Object 对象引用存在，count 很大的话，就耗费内存了，建议为改为：

```
Object obj = null;for (int i = 0; i <= count; i++) { obj = new Object(); }
```

这样的话，内存中只有一份 Object 对象引用，每次 new Object()的时候，Object 对象引用指向不同的 Object 罢了，但是内存中只有一份，这样就大大节省了内存空间了。

13、基于效率和类型检查的考虑，应该尽可能使用 array，无法确定数组大小时才使用 ArrayList

14、尽量使用 HashMap、ArrayList、StringBuilder，除非线程安全需要，否则不推荐使用 Hashtable、Vector、StringBuffer，后三者由于使用同步机制而导致了性能开销

15、不要将数组声明为 public static final

因为这毫无意义，这样只是定义了引用为 static final，数组的内容还是可以随意改变的，将数组声明为 public 更是一个安全漏洞，这意味着这个数组可以被外部类所改变

16、尽量在合适的场合使用单例

使用单例可以减轻加载的负担、缩短加载的时间、提高加载的效率，但并不是所有地方都适用于单例，简单来说，单例主要适用于以下三个方面：

（1）控制资源的使用，通过线程同步来控制资源的并发访问

（2）控制实例的产生，以达到节约资源的目的

（3）控制数据的共享，在不建立直接关联的条件下，让多个不相关的进程或线程之间实现通信

17、尽量避免随意使用静态变量

要知道，当某个对象被定义为 static 的变量所引用，那么 gc 通常是不会回收这个对象所占有的堆内存的，如：

```
public class A

{
```

```
private static B b = new B();

}
```

此时静态变量 b 的生命周期与 A 类相同，如果 A 类不被卸载，那么引用 B 指向的 B 对象会常驻内存，直到程序终止

18、及时清除不再需要的会话

为了清除不再活动的会话，许多应用服务器都有默认的会话超时时间，一般为 30 分钟。当应用服务器需要保存更多的会话时，如果内存不足，那么操作系统会把部分数据转移到磁盘，应用服务器也可能根据 MRU（最近最频繁使用）算法把部分不活跃的会话转储到磁盘，甚至可能抛出内存不足的异常。如果会话要被转储到磁盘，那么必须要先被序列化，在大规模集群中，对对象进行序列化的代价是很昂贵的。因此，当会话不再需要时，应当及时调用 HttpSession 的 invalidate()方法清除会话。

19、实现 RandomAccess 接口的集合比如 ArrayList，应当使用最普通的 for 循环而不是 foreach 循环来遍历

这是 JDK 推荐给用户的。JDK API 对于 RandomAccess 接口的解释是：实现 RandomAccess 接口用来表明其支持快速随机访问，此接口的主要目的是允许一般的算法更改其行为，从而将其应用到随机或连续访问列表时能提供良好的性能。实际经验表明，实现 RandomAccess 接口的类实例，假如是随机访问的，使用普通 for 循环效率将高于使用 foreach 循环；反过来，如果是顺序访问的，则使用 Iterator 会效率更高。可以使用类似如下的代码作判断：

```
if (list instanceof RandomAccess)

{

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

}

Else

{

Iterator<?> iterator = list.iterable(); while (iterator.hasNext()){iterator.next()}

}
```

foreach 循环的底层实现原理就是迭代器 Iterator，参见 Java 语法糖 1：可变长度参数以及 foreach 循环原理。所以后半句”反过来，如果是顺序访问的，则使用 Iterator 会效率更高”的意思就是顺序访问的那些类实例，使用 foreach 循环去遍历。

20、使用同步代码块替代同步方法

这点在多线程模块中的 synchronized 锁方法块一文中已经讲得很清楚了，除非能确定一整个方法都是需要进行同步的，否则尽量使用同步代码块，避免对那些不需要进行同步的代码也进行了同步，影响了代码执行效率。

21、将常量声明为 static final，并以大写命名

这样在编译期间就可以把这些内容放入常量池中，避免运行期间计算生成常量的值。另外，将常量的名字以大写命名也可以方便区分出常量与变量

22、不要创建一些不使用的对象，不要导入一些不使用的类

这毫无意义，如果代码中出现”The value of the local variable i is not used”、”The import java.util is never used”，那么请删除这些无用的内容

23、程序运行过程中避免使用反射

关于，请参见反射。反射是 Java 提供给用户一个很强大的功能，功能强大往往意味着效率不高。不建议在程序运行过程中使用尤其是频繁使用反射机制，特别是 Method 的 invoke 方法，如果确实有必要，一种建议性的做法是将那些需要通过反射加载的类在项目启动的时候通过反射实例化出一个对象并放入内存——用户只关心和对端交互的时候获取最快的响应速度，并不关心对端的项目启动花多久时间。

24、使用数据库连接池和线程池

这两个池都是用于重用对象的，前者可以避免频繁地打开和关闭连接，后者可以避免频繁地创建和销毁线程

25、使用带缓冲的输入输出流进行 IO 操作

带缓冲的输入输出流，即 BufferedReader、BufferedWriter、BufferedInputStream、BufferedOutputStream，这可以极大地提升 IO 效率

26、顺序插入和随机访问比较多的场景使用 ArrayList，元素删除和中间插入比较多的场景使用 LinkedList

这个，理解 ArrayList 和 LinkedList 的原理就知道了

27、不要让 public 方法中有太多的形参

public 方法即对外提供的方法，如果给这些方法太多形参的话主要有两点坏处：

1、违反了面向对象的编程思想，Java 讲求一切都是对象，太多的形参，和面向对象的编程思想并不契合

2、参数太多势必导致方法调用的出错概率增加

至于这个”太多”指的是多少个，3、4 个吧。大讲台，混合式自适应 IT 职业教育开创者。比如我们用 JDBC 写一个 insertStudentInfo 方法，有 10 个学生信息字段要插如 Student 表中，可以把这 10 个参数封装在一个实体类中，作为 insert 方法的形参

28、字符串变量和字符串常量 equals 的时候将字符串常量写在前面

这是一个比较常见的小技巧了，如果有以下代码：

```
String str = "123";

if (str.equals("123"))

{

...

}
```

建议修改为：

```
String str = "123";

if ("123".equals(str))

{

...

}
```

这么做主要是可以避免空指针异常

29、请知道，在 java 中 if (i == 1)和 if (1 == i)是没有区别的，但从阅读习惯上讲，建议使用前者

平时有人问，”if (i == 1)”和”if (1== i)”有没有区别，这就要从 C/C++讲起。

在 C/C++中，”if (i == 1)”判断条件成立，是以 0 与非 0 为基准的，0 表示 false，非 0 表示 true，如果有这么一段代码：

```
int i = 2;

if (i == 1)

{

...

}

else{

...

}
```

C/C++判断”i==1″不成立，所以以 0 表示，即 false。但是如果：

```
int i = 2;if (i = 1) { ... }else{ ... }
```

万一程序员一个不小心，把”if (i == 1)”写成”if (i = 1)”，这样就有问题了。在 if 之内将 i 赋值为 1，if 判断里面的内容非 0，返回的就是 true 了，但是明明 i 为 2，比较的值是 1，应该返回的 false。这种情况在 C/C++的开发中是很可能发生的并且会导致一些难以理解的错误产生，所以，为了避免开发者在 if 语句中不正确的赋值操作，建议将 if 语句写为：

```
int i = 2;if (1 == i) { ... }else{ ... }
```

这样，即使开发者不小心写成了”1 = i”，C/C++编译器也可以第一时间检查出来，因为我们可以对一个变量赋值 i 为 1，但是不能对一个常量赋值 1 为 i。

但是，在 Java 中，C/C++这种”if (i = 1)”的语法是不可能出现的，因为一旦写了这种语法，Java 就会编译报错”Type mismatch: cannot convert from int to boolean”。但是，尽管 Java 的”if (i == 1)”和”if (1 == i)”在语义上没有任何区别，但是从阅读习惯上讲，建议使用前者会更好些。

30、不要对数组使用 toString()方法

看一下对数组使用 toString()打印出来的是什么：

```
public static void main(String[] args)

{ int[] is = new int[]{1, 2, 3};

System.out.println(is.toString());

}
```

结果是：

```
[I@18a992f
```

本意是想打印出数组内容，却有可能因为数组引用 is 为空而导致空指针异常。不过虽然对数组 toString()没有意义，但是对集合 toString()是可以打印出集合里面的内容的，因为集合的父类 AbstractCollections<E>重写了 Object 的 toString()方法。

31、不要对超出范围的基本数据类型做向下强制转型

这绝不会得到想要的结果：

```
public static void main(String[] args)

{

long l = 12345678901234L;

int i = (int)l;

System.out.println(i);

}
```

我们可能期望得到其中的某几位，但是结果却是：

1942892530

解释一下。Java 中 long 是 8 个字节 64 位的，所以 12345678901234 在计算机中的表示应该是：

0000 0000 0000 0000 0000 1011 0011 1010 0111 0011 1100 1110 0010 1111 1111 0010

一个 int 型数据是 4 个字节 32 位的，从低位取出上面这串二进制数据的前 32 位是：

0111 0011 1100 1110 0010 1111 1111 0010

这串二进制表示为十进制 1942892530，所以就是我们上面的控制台上输出的内容。从这个例子上还能顺便得到两个结论：

1、整型默认的数据类型是 int，long l = 12345678901234L，这个数字已经超出了 int 的范围了，所以最后有一个 L，表示这是一个 long 型数。顺便，浮点型的默认类型是 double，所以定义 float 的时候要写成”"float f = 3.5f”

2、接下来再写一句”int ii = l + i;”会报错，因为 long + int 是一个 long，不能赋值给 int

32、公用的集合类中不使用的数据一定要及时 remove 掉

如果一个集合类是公用的（也就是说不是方法里面的属性），那么这个集合里面的元素是不会自动释放的，因为始终有引用指向它们。大讲台，混合式自适应 IT 职业教育开创者。所以，如果公用集合里面的某些数据不使用而不去 remove 掉它们，那么将会造成这个公用集合不断增大，使得系统有内存泄露的隐患。

33、把一个基本数据类型转为字符串，基本数据类型.toString()是最快的方式、String.valueOf(数据)次之、数据+”"最慢

把一个基本数据类型转为一般有三种方式，我有一个 Integer 型数据 i，可以使用 i.toString()、String.valueOf(i)、i+”"三种方式，三种方式的效率如何，看一个测试：

```
public static void main(String[] args)

{

int loopTime = 50000;



Integer i = 0; long startTime = System.currentTimeMillis(); for (int j = 0; j <
 loopTime; j++)

{

String str = String.valueOf(i);

}
```

```
System.out.println("String.valueOf(): " + (System.currentTimeMillis() - startTime) + "ms");

startTime = System.currentTimeMillis(); for (int j = 0; j < loopTime; j++)

{

String str = i.toString();

}

System.out.println("Integer.toString(): " + (System.currentTimeMillis() - startTime) + "ms");

startTime = System.currentTimeMillis(); for (int j = 0; j < loopTime; j++)

{

String str = i + "";

}

System.out.println("i + \"\": " + (System.currentTimeMillis() - startTime) + "ms");

}
```

运行结果为：

```
String.valueOf(): 11ms Integer.toString(): 5ms i + "": 25ms
```

所以以后遇到把一个基本数据类型转为 String 的时候，优先考虑使用 toString()方法。至于为什么，很简单：

1、String.valueOf()方法底层调用了 Integer.toString()方法，但是会在调用前做空判断

2、Integer.toString()方法就不说了，直接调用了

3、i + “”底层使用了 StringBuilder 实现，先用 append 方法拼接，再用 toString()方法获取字符串

三者对比下来，明显是 2 最快、1 次之、3 最慢

34、使用最有效率的方式去遍历 Map

遍历 Map 的方式有很多，通常场景下我们需要的是遍历 Map 中的 Key 和 Value，那么推荐使用的、效率最高的方式是：

```
public static void main(String[] args)

{

HashMap<String, String> hm = new HashMap<String, String>();

hm.put("111", "222");

Set<Map.Entry<String, String>> entrySet = hm.entrySet();

Iterator<Map.Entry<String, String>> iter = entrySet.iterator(); while (iter.h
asNext())

{

Map.Entry<String, String> entry = iter.next();

System.out.println(entry.getKey() + "\t" + entry.getValue());

}

}
```

如果你只是想遍历一下这个 Map 的 key 值，那用”Set<String> keySet = hm.keySet();”会比较合适一些

35、对资源的 close()建议分开操作

意思是，比如我有这么一段代码：

```
try{

XXX.close();

YYY.close();

}catch (Exception e)

{

...

}
```

建议修改为：

```
try{ XXX.close(); }catch (Exception e) { ... }try{ YYY.close(); }catch (Except
ion e) { ... }
```

虽然有些麻烦，却能避免资源泄露。我们想，如果没有修改过的代码，万一 XXX.close()抛异常了，那么就进入了 cath 块中了，YYY.close()不会执行，YYY 这块资源就不会回收了，一直占用着，这样的代码一多，是可能引起资源句柄泄露的。而改为下面的写法之后，就保证了无论如何 XXX 和 YYY 都会被 close 掉。