链滴

JAVA 代码优化十九式

作者：jianzh5

原文链接：https://ld246.com/article/1599613157159

来源网站：链滴

在JAVA中好的代码可以带来性能的提升，本节将讲解一些常用的代码优化原则，从而在编码中保持的习惯，让代码保持最优状态，当然也可以将这些原则引入到代码评审中，让整个团队都写出更好的码。

1.使用局部变量可避免在堆上分配

由于堆资源是多线程共享的，是垃圾回收器工作的主要区域，过多的对象会造成 GC 压力。可以通过部变量的方式，将变量在栈上分配。这种方式变量会随着方法执行的完毕而销毁，能够减轻 GC 的压。

2.减少变量的作用范围

注意变量的作用范围，尽量减少对象的创建。如下面的代码，变量 a 每次进入方法都会创建，可以将移动到 if 语句内部。

```
public void test1(String str) {
    final int a = 100;
    if (!StringUtils.isEmpty(str)) {
        int b = a * a;
    }
}
```

3.访问静态变量直接使用类名

有的同学习惯使用对象访问静态变量，这种方式多了一步寻址操作，需要先找到变量对应的类，再找类对应的变量，如下面的代码：

```
public class StaticCall {
    public static final int A = 1;

    void test() {
        System.out.println(this.A);
        System.out.println(StaticCall.A);
    }
}
```

对应的字节码为：

```
void test();
    descriptor: ()V
    flags:
    Code:
      stack=2, locals=1, args_size=1
         0: getstatic     #2                  // Field java/lang/System.out:Ljava/io/PrintStream;
         3: aload_0
         4: pop
         5: iconst_1
         6: invokevirtual #3                  // Method java/io/PrintStream.println:(I)V
         9: getstatic     #2                  // Field java/lang/System.out:Ljava/io/PrintStream;
        12: iconst_1
        13: invokevirtual #3                  // Method java/io/PrintStream.println:(I)V
        16: return
```

```
      LineNumberTable:
        line 5: 0
        line 6: 9
        line 7: 16
```

可以看到使用 this 的方式多了一个步骤。

4.字符串拼接使用 StringBuilder

字符串拼接，使用 StringBuilder 或者 StringBuffer，不要使用 + 号。比如下面这段代码，在循环中接了字符串。

```
public String test() {
    String str = "-1";
    for (int i = 0; i < 10; i++) {
        str += i;
    }
    return str;
}
```

从下面对应的字节码内容可以看出，它在每个循环里都创建了一个 StringBuilder 对象。所以，我们平常的编码中，显式地创建一次即可。

```
 5: iload_2
 6: bipush        10
 8: if_icmpge     36
11: new           #3                  // class java/lang/StringBuilder
14: dup
15: invokespecial #4                  // Method java/lang/StringBuilder."<init>":()V
18: aload_1
19: invokevirtual #5                  // Method java/lang/StringBuilder.append:(Ljava/lang/String;)L
ava/lang/StringBuilder;
22: iload_2
23: invokevirtual #6                  // Method java/lang/StringBuilder.append:(I)Ljava/lang/String
uilder;
26: invokevirtual #7                  // Method java/lang/StringBuilder.toString:()Ljava/lang/String;
29: astore_1
30: iinc          2, 1
33: goto          5
```

5.重写对象的 HashCode，不要简单地返回固定值

在代码 review 的时候，我发现有开发重写 HashCode 和 Equals 方法时，会把 HashCode 的值返回定的 0，而这样做是不恰当的。

当这些对象存入 HashMap 时，性能就会非常低，因为 HashMap 是通过 HashCode 定位到 Hash，有冲突的时候，才会使用链表或者红黑树组织节点。固定地返回 0，相当于把 Hash 寻址功能给废了。

6.HashMap 等集合初始化的时候，指定初始值大小

这样的对象有很多，比如 ArrayList，StringBuilder 等，通过指定初始值大小可减少扩容造成的性能耗。

7.遍历 Map 的时候，使用 EntrySet 方法

使用 EntrySet 方法，可以直接返回 set 对象，直接拿来用即可；而使用 KeySet 方法，获得的是key 集合，需要再进行一次 get 操作，多了一个操作步骤。所以更推荐使用 EntrySet 方式遍历 Map。

8.不要在多线程下使用同一个 Random

Random 类的 seed 会在并发访问的情况下发生竞争，造成性能降低，建议在多线程环境下使用 ThredLocalRandom 类。

在 Linux 上，通过加入 JVM 配置 **-Djava.security.egd=file:/dev/./urandom**，使用 urandom 机生成器，在进行随机数获取时，速度会更快。

9.自增推荐使用 LongAddr

自增运算可以通过 synchronized 和 volatile 的组合，或者也可以使用原子类（比如 AtomicLong）。

后者的速度比前者要高一些，AtomicLong 使用 CAS 进行比较替换，在线程多的情况下会造成过多效自旋，所以可以使用 LongAdder 替换 AtomicLong 进行进一步的性能提升。

10.不要使用异常控制程序流程

异常，是用来了解并解决程序中遇到的各种不正常的情况，它的实现方式比较昂贵，比平常的条件判语句效率要低很多。

这是因为异常在字节码层面，需要生成一个如下所示的异常表（Exception table），多了很多判断步。

```
Exception table:
   from    to  target type
   7    17    20   any
   20    23    20   any
```

所以，尽量不要使用异常控制程序流程。

11.不要在循环中使用 try catch

道理与上面类似，很多文章介绍，不要把异常处理放在循环里，而应该把它放在最外层，但实际测试况表明这两种方式性能相差并不大。

既然性能没什么差别，那么就推荐根据业务的需求进行编码。比如，循环遇到异常时，不允许中断，就是允许在发生异常的时候能够继续运行下去，那么异常就只能在 for 循环里进行处理。

12.不要捕捉 RuntimeException

Java 异常分为两种，一种是可以通过预检查机制避免的 RuntimeException；另外一种就是普通异常。

其中，RuntimeException 不应该通过 catch 语句去捕捉，而应该使用编码手段进行规避。

如下面的代码，list 可能会出现数组越界异常。是否越界是可以通过代码提前判断的，而不是等到发异常时去捕捉。提前判断这种方式，代码会更优雅，效率也更高。

```
//BAD
public String test1(List<String> list, int index) {
    try {
        return list.get(index);
    } catch (IndexOutOfBoundsException ex) {
        return null;
    }
}

//GOOD
public String test2(List<String> list, int index) {
    if (index >= list.size() || index < 0) {
        return null;
    }
    return list.get(index);
}
```

13.合理使用 PreparedStatement

PreparedStatement 使用预编译对 SQL 的执行进行提速，大多数数据库都会努力对这些能够复用的询语句进行预编译优化，并能够将这些编译结果缓存起来。

这样等到下次用到的时候，就可以很快进行执行，也就少了一步对 SQL 的解析动作。

PreparedStatement 还能提高程序的安全性，能够有效防止 SQL 注入。

但如果你的程序每次 SQL 都会变化，不得不手工拼接一些数据，那么 PreparedStatement 就失去了的作用，反而使用普通的 Statement 速度会更快一些。

14.日志打印的注意事项

我们平常会使用 debug 输出一些调试信息，然后在线上关掉它。如下代码：

```
logger.debug("xjjdog:"+ topic + "  is  awesome"  );
```

程序每次运行到这里，都会构造一个字符串，不管你是否把日志级别调试到 INFO 还是 WARN，这效率就会很低。

可以在每次打印之前都使用 isDebugEnabled 方法判断一下日志级别，代码如下：

```
if(logger.isDebugEnabled()) {
    logger.debug("xjjdog:"+ topic + "  is  awesome"  );
}
```

使用占位符的方式，也可以达到相同的效果，就不用手动添加 isDebugEnabled 方法了，代码也优得多。

```
logger.debug("xjjdog:{}  is  awesome"  ,topic);
```

对于业务系统来说，日志对系统的性能影响非常大，不需要的日志，尽量不要打印，避免占用 I/O 资。

15.减少事务的作用范围

如果的程序使用了事务，那一定要注意事务的作用范围，尽量以最快的速度完成事务操作。这是因为事务的隔离性是使用锁实现的。

```
@Transactional
public void test(String id){
    String value = rpc.getValue(id); //高耗时
    testDao.update(sql,value);
}
```

如上面的代码，由于 rpc 服务耗时高且不稳定，就应该把它移出到事务之外，改造如下：

```
public void test(String id){
    String value = rpc.getValue(id); //高耗时
    testDao(value);
}
@Transactional
public void testDao(String value){
    testDao.update(value);
}
```

这里有一点需要注意的地方，由于 SpringAOP 的原因，@Transactional 注解只能用到 public 方上，如果用到 private 方法上，将会被忽略。

16.使用位移操作替代乘除法

计算机是使用二进制表示的，位移操作会极大地提高性能。

```
<< 左移相当于乘以 2；
>> 右移相当于除以 2；
>>> 无符号右移相当于除以 2，但它会忽略符号位，空位都以 0 补齐。

int a = 2;
int b = (a++) << (++a) + (++a);
System.out.println(b);
```

注意：位移操作的优先级非常低，所以上面的代码，输出是 1024。

17.不要打印大集合或者使用大集合的 toString 方法

有的开发喜欢将集合作为字符串输出到日志文件中，这个习惯是非常不好的。

拿 ArrayList 来说，它需要遍历所有的元素来迭代生成字符串。在集合中元素非常多的情况下，这不会占用大量的内存空间，执行效率也非常慢。我曾经就遇到过这种批量打印方式造成系统性能直线下的实际案例。

下面这段代码，就是 ArrayList 的 toString 方法。它需要生成一个迭代器，然后把所有的元素内容拼成一个字符串，非常浪费空间。

```
public String toString() {
    Iterator<E> it = iterator();
    if (! it.hasNext())
        return "[]";

    StringBuilder sb = new StringBuilder();
```

```
    sb.append('[');
    for (;;) {
        E e = it.next();
        sb.append(e == this ? "(this Collection)" : e);
        if (! it.hasNext())
            return sb.append(']').toString();
        sb.append(',').append(' ');
    }
}
```

18.程序中少用反射

反射的功能很强大，但它是通过解析字节码实现的，性能就不是很理想。

现实中有很多对反射的优化方法，比如把反射执行的过程（比如 Method）缓存起来，使用复用来加反射速度。

Java 7 之后，加入了新的包 java.lang.invoke，同时加入了新的 JVM 字节码指令 invokedynamic，来支持从 JVM 层面，直接通过字符串对目标方法进行调用。

如果你对性能有非常苛刻的要求，则使用 invoke 包下的 MethodHandle 对代码进行着重优化,但它编程不如反射方便，在平常的编码中，反射依然是首选。

下面是一个使用 MethodHandle 编写的代码实现类。它可以完成一些动态语言的特性，通过方法名和传入的对象主体，进行不同的调用，而 Bike 和 Man 类，可以是没有任何关系的。

```
import java.lang.invoke.MethodHandle;
import java.lang.invoke.MethodHandles;
import java.lang.invoke.MethodType;

public class MethodHandleDemo {
    static class Bike {
        String sound() {
            return "ding ding";
        }
    }

    static class Animal {
        String sound() {
            return "wow wow";
        }
    }


    static class Man extends Animal {
        @Override
        String sound() {
            return "hou hou";
        }
    }


    String sound(Object o) throws Throwable {
        MethodHandles.Lookup lookup = MethodHandles.lookup();
```

```
        MethodType methodType = MethodType.methodType(String.class);
        MethodHandle methodHandle = lookup.findVirtual(o.getClass(), "sound", methodType);

        String obj = (String) methodHandle.invoke(o);
        return obj;
    }

    public static void main(String[] args) throws Throwable {
        String str = new MethodHandleDemo().sound(new Bike());
        System.out.println(str);
        str = new MethodHandleDemo().sound(new Animal());
        System.out.println(str);
        str = new MethodHandleDemo().sound(new Man());
        System.out.println(str);
    }
}
```

19.正则表达式可以预先编译，加快速度

Java 的正则表达式需要先编译再使用，典型代码如下：

```
Pattern pattern = Pattern.compile({pattern});
Matcher pattern = pattern.matcher({content});
```

Pattern 编译非常耗时，它的 Matcher 方法是线程安全的，每次调用方法这个方法都会生成一个新的 Matcher 对象。所以，一般 Pattern 初始化一次即可，可以作为类的静态成员变量。