

26 | 有哪些招惹麻烦的性能陷阱？

2019-03-04 范学雷



讲述：刘飞

时长 09:53 大小 9.05M




前面，我们讨论了改善代码性能的最基本的办法。接下来，我们讨论一些最佳实践，让我们先从一些容易被忽略的性能陷阱开始。

使用性能测试工具

今天我们的讲解需要用到一个工具，它就是 JMH。JMH 是为 Java 语言或者其他基于 JVM 的编程语言设计的一个基准测试工具。这一节，我们会使用这个工具来分析一些性能的陷阱。这里我们简单地介绍下，这个工具该怎么使用。

第一步，使用 Maven 工具建立一个基准测试项目（需要使用 Maven 工具）：


 复制代码

```
1 $ mvn archetype:generate \  
2     -DinteractiveMode=false \  
3     -DarchetypeGroupId=org.openjdk.jmh \
```

```
3 -DarchetypeGroupId=org.apache.maven.archetypes \
4 -DarchetypeArtifactId=jmh-java-benchmark-archetype \
5 -DgroupId=com.example \
6 -DartifactId=myJmh \
7 -Dversion=1.0
```

这个命令行，会生成一个 myJmh 的工程目录，和一个基准测试模板文件（myJmh/src/main/java/com/example/MyBenchmark.java）。通过更改这个测试模板，就可以得到你想要的基准测试了。

比如，你可以使用后面我们用到的基准测试代码，替换掉模板中的基准测试方法（measureStringAppend）。

 复制代码


```
1 package com.example;
2
3 import org.openjdk.jmh.annotations.Benchmark;
4
5 public class MyBenchmark {
6     @Benchmark
7     public String measureStringAppend() {
8         String targetString = "";
9         for (int i = 0; i < 10000; i++) {
10             targetString += "hello";
11         }
12
13         return targetString;
14     }
15 }
```

第二步，编译基准测试：

 复制代码


```
1 $ cd myJmh
2 $ mvn clean install
```

第三步，运行你的基准测试：

 复制代码

```
1 $ cd myJmh
2 $ Java -jar target/benchmarks.jar
```

稍微等待，基准测试结果就出来了。我们需要关注的是"Score"这一栏，它表示的是每秒钟可以执行的基准测试方法的次数。

 复制代码


```
1 Benchmark           Mode  Cnt      Score    Error  Units
2 MyBenchmark.testMethod  thrpt   25     35.945    0.694  ops/s
```

这是 JMH 工具基本的使用流程，有关这个工具更多的选项和更详细的使用，需要你参考 JMH 的相关文档。


下面，我们通过字符串连接操作和哈希值的例子，来谈论一下这个工具要怎么使用，以及对应的性能问题。同时，我们再看看其他影响性能的一些小陷阱，比如内存的泄露、未关闭的资源 and 遗漏的 hashCode。

字符串的操作

在 Java 的核心类库里，有三个字符串操作的类，分别是 String、StringBuilder 和 StringBuffer。通过下面的基准测试，我们来了解下这三种不同的字符串操作的性能差异。为了方便，我把 JMH 测试的数据，标注在每个基准测试的方法注释里了。

 复制代码

```
1 // JMH throughput benchmark: about 32 operations per second
2 @Benchmark
3 public String measureStringAppend() {
4     String targetString = "";
5     for (int i = 0; i < 10000; i++) {
6         targetString += "hello";
7     }
8
9     return targetString;
10 }
```


 复制代码

```
1 // JMH throughput benchmark: about 5,600 operations per second
2 @Benchmark
```

```

2    @Benchmark
3    public String measureStringBufferAppend() {
4        StringBuffer buffer = new StringBuffer();
5        for (int i = 0; i < 10000; i++) {
6            buffer.append("hello");
7        }
8
9        return buffer.toString();
10   }

```

 复制代码

```

1    // JMH throughput benchmark: about 21,000 operations per second
2    @Benchmark
3    public String measureStringBuilderAppend() {
4        StringBuilder builder = new StringBuilder();
5        for (int i = 0; i < 10000; i++) {
6            builder.append("hello");
7        }
8
9        return builder.toString();
10   }


```

对于字符串连接的操作，这个基准测试结果显示，使用 `StringBuffer` 的字符串连接操作，比使用 `String` 的操作快了近 200 倍；使用 `StringBuilder` 的字符串连接操作，比使用 `String` 的操作快了近 700 倍。

`String` 的字符串连接操作为什么慢呢？这是因为每一个字符串连接的操作（`targetString += "hello"`），都需要创建一个新的 `String` 对象，然后再销毁，再创建。这种模式对 CPU 和内存消耗都比较大。

`StringBuilder` 和 `StringBuffer` 为什么快呢？因为 `StringBuilder` 和 `StringBuffer` 的内部实现，预先分配了一定的内存。字符串操作时，只有预分配内存不足，才会扩展内存，这就大幅度减少了内存分配、拷贝和释放的频率。

`StringBuilder` 为什么比 `StringBuffer` 还要快呢？`StringBuffer` 的字符串操作是多线程安全的，而 `StringBuilder` 的操作就不是。如果我们看这两个方法的实现代码，除了线程安全的同步以外，几乎没有差别。

 复制代码

```


1 public final class StringBuffer

```

```

2     extends AbstractStringBuilder
3     implements java.io.Serializable, Comparable<StringBuffer>, CharSequence {
4     // snipped
5
6     @Override
7     @HotSpotIntrinsicCandidate
8     public synchronized StringBuffer append(String str) {
9         toStringCache = null;
10        super.append(str);
11        return this;
12    }
13
14    // snipped
15 }

```

 复制代码


```

1 public final class StringBuilder
2     extends AbstractStringBuilder
3     implements java.io.Serializable, Comparable<StringBuilder>, CharSequence {
4     // snipped
5
6     @Override
7     @HotSpotIntrinsicCandidate
8     public StringBuilder append(String str) {
9         super.append(str);
10        return this;
11    }
12
13    // snipped
14 }

```

JMH 的基准测试，并没有涉及到线程同步问题，难道使用 `synchronized` 关键字也会有性能损耗吗？

我们再来看看另外一个基准测试。这个基准测试，使用线程不安全的 `StringBuilder` 以及同步的字符串连接，部分模拟了线程安全的 `StringBuffer.append()` 方法的实现。为了方便你对比，我把没有使用同步的代码也拷贝在下面。

 复制代码

```


1     // JMH throughput benchmark: about 21,000 operations per second
2     @Benchmark
3
4     public String measureStringBuilderApend() {
5         StringBuilder builder = new StringBuilder();

```

```

5         for (int i = 0; i < 10000; i++) {
6             builder.append("hello");
7         }
8
9         return builder.toString();
10    }

```

 复制代码

```

1    // JMH throughput benchmark: about 16,000 operations per second
2    @Benchmark
3    public String measureStringBuilderSynchronizedAppend() {
4        StringBuilder builder = new StringBuilder();
5        for (int i = 0; i < 10000; i++) {
6            synchronized (this) {
7                builder.append("hello");
8            }
9        }
10
11        return builder.toString();
12    }

```

这个基准测试结果显示，虽然基准测试并没有使用多个线程，但是使用了线程同步的代码比不使用线程同步的代码慢。线程同步，就是 `StringBuffer` 比 `StringBuilder` 慢的原因之一。


通过上面的基准测试，我们可以得出这样的结论：

1. 频繁的对象创建、销毁，有损代码的效率；
2. 减少内存分配、拷贝、释放的频率，可以提高代码的效率；
3. 即使是单线程环境，使用线程同步依然有损代码的效率。


从上面的基准测试结果，是不是可以得出结论，我们应该使用 `StringBuilder` 来进行字符串操作呢？我们再来看几个基准测试的例子。

下面的例子，测试的是常量字符串的连接操作。从测试结果，我们可以看出，使用 `String` 的连接操作，要比使用 `StringBuilder` 的字符串连接快 5 万倍，这是一个让人惊讶的性能差异。


```
1 // JMH throughput benchmark: about 1,440,000,000 operations per second
2 @Benchmark
3 public void measureSimpleStringApend() {
4     for (int i = 0; i < 10000; i++) {
5         String targetString = "Hello, " + "world!";
6     }
7 }
```

 复制代码


```
1 // JMH throughput benchmark: about 26,000 operations per second
2 @Benchmark
3 public void measureSimpleStringBuilderApend() {
4     for (int i = 0; i < 10000; i++) {
5         StringBuilder builder = new StringBuilder();
6         builder.append("hello, ");
7         builder.append("world!");
8     }
9 }
```

 复制代码


这个巨大的差异，主要来自于 Java 编译器和 JVM 对字符串处理的优化。" Hello, " + " world! " 这样的表达式，并没有真正执行字符串连接。编译器会把它处理成一个连接好的常量字符串"Hello, world!"。这样，也就不存在反复的对象创建和销毁了，常量字符串的连接显示了超高的效率。

如果字符串的连接里，出现了变量，编译器和 JVM 就没有办法进行优化了。这时候，StringBuilder 的效率优势才能体现出来。下面的两个基准测试结果，就显示了变量对于字符长连接操作效率的影响。

```
1 // JMH throughput benchmark: about 9,000 operations per second
2 @Benchmark
3 public void measureVariableStringApend() {
4     for (int i = 0; i < 10000; i++) {
5         String targetString = "Hello, " + getAppendix();
6     }
7 }
```

 复制代码


```
1 // JMH throughput benchmark: about 26,000 operations per second
2 @Benchmark
```

 复制代码

```

3     public void measureVariableStringBuilderApend() {
4         for (int i = 0; i < 10000; i++) {
5             StringBuilder builder = new StringBuilder();
6             builder.append("hello, ");
7             builder.append(getAppendix());
8         }
9     }
10

```

 复制代码

```

1     private String getAppendix() {
2         return "World!";
3     }

```

通过上面的基准测试，我们可以总结出下面的几条最佳实践：

1. Java 的编译器会优化常量字符串的连接，我们可以放心地把长的字符串换成多行；
2. 带有变量的字符串连接，StringBuilder 效率更高。如果效率敏感的代码，建议使用 StringBuilder。String 的连接操作可读性更高，效率不敏感的代码可以使用，比如异常信息、调试日志、使用不频繁的代码；
3. 如果涉及大量的字符串操作，使用 StringBuilder 效率更高；
4. 除非有线程安全的需求，不推荐使用线程安全的 StringBuffer。

内存的泄露

内存泄漏是 C 语言的一个大问题。为了更好地管理内存，Java 提供了自动的内存管理和垃圾回收机制。但是，Java 依然会泄露内存。这种内存泄漏的主要表现是，如果一个对象不再有用处，而且它的引用还没有清零，垃圾回收器就意识不到这个对象需要及时回收，这时候就引发了内存泄露。

生命周期长的集合，是 Java 容易发生内存泄漏的地方。比如，可以扩张的静态的集合，或者存活时间长的缓存等。如果不能及时清理掉集合里没有用处的对象，就会造成内存的持续增加，引发内存泄漏问题。

比如下面这两个例子，就容易发生内存泄露。

静态的集合：


```
1 static final List<Object>
2     staticCachedObjects = new LinkedList<>();
3
4 // snipped
5 staticCachedObjects.add(...);
```

长寿的缓存：

```
1 final List<Object>
2     longLastingCache = new LinkedList<>();
3
4 // snipped
5 longLastingCache.add(...);
```

解决这个问题的办法通常是使用 `SoftReference` 和 `WeakReference` 来存储对象的引用，或者主动地定期清理。

静态的集合：

```
1 static final List<WeakReference<Object>>
2     staticCachedObjects = new LinkedList<>();
3
4 // snipped
5 staticCachedObjects.add(...);
```

长寿的缓存：

```
1 final List<WeakReference<Object>>
2     longLastingCache = new LinkedList<>();
3
4 // snipped
5 longLastingCache.add(...);
```

需要注意的是，缓存的处理是一个复杂的问题，使用 SoftReference 和 WeakReference 未必能够满足你的业务需求。更有效的缓存解决方案，依赖于具体的使用场景。

未关闭的资源

有很多系统资源，需要明确地关闭，要不然，占用的系统资源就不能有效地释放。比如说，数据库连接、套接字连接和 I/O 操作等。原则上，所有实现了 Closable 接口的对象，都应该调用 close() 操作；所有需要明确关闭的类，都应该实现 Closable 接口。

需要注意的是，close() 操作，一定要使用 try-finally 或者 try-with-resource 语句。要不然，关闭资源的代码可能很复杂。

try-finally	<pre>Socket socket = new Socket(); try { ... } finally { socket.close(); }</pre>
try-with-resource	<pre>try (Socket socket = new Socket()) { ... }</pre>
复杂，不推荐	<pre>Socket socket = new Socket(); try { ... socket.close(); // close the socket after all operations } catch (Exception ex) { ... socket.close(); // close the socket if exception was thrown }</pre>

如果一个类需要关闭，但是又没有实现 Closable 接口，就比较麻烦，比如 URLConnection。URLConnection.connect() 能够建立连接，该连接需要关闭，但是 URLConnection 没有实现 Closable 接口，关闭的办法只能是关闭对应的 I/O 接口，可是关闭 I/O 输入和输出接口中的一个，还不能保证整个连接会完全关闭。谨慎的代码，需要把 I/O 输入和输出都关闭掉，哪怕不需要输入或者输出。但是这样一来，我们的编码负担就会加重。所以最好的方法就是实现 Closable 接口。

双向关闭 I/O：

```
1 URL url = new URL("http://www.google.com/");
2 URLConnection conn = url.openConnection();
3 conn.connect();
4
5 try (InputStream is = conn.getInputStream()) {
6     // sinnped
7 }
8
9 try (OutputStream os = conn.getOutputStream()) {
10     // sinnped
11 }
```

单向关闭 I/O :

```
1 URL url = new URL("http://www.google.com/");
2 URLConnection conn = url.openConnection();
3 conn.connect();
4
5 try (InputStream is = conn.getInputStream()) {
6     // sinnped
7 }
8
9 // The output stream is not close, the connection may be still alive.
```

遗漏的 hashCode

在使用 Hashtbale、HashMap、HashSet 这样的依赖哈希 (hash) 值的集合时，有时候我们会忘记要检查产生哈希值的对象，一定要实现 hashCode() 和 equals() 这两个方法。缺省的 hashCode() 实现，返回值是每一个对象都不同的数值。即使是相等的对象，不同的哈希值，使用基于哈希值的集合时，也会被看作不同的对象。这样的行为，可能不符合我们的预期。而且，使用没有实现 hashCode() 和 equals() 这两个方法的对象，可能会造成集合的尺寸持续增加，无端地占用内存，甚至会造成内存的泄漏。

所以，我们使用基于 hash 的集合时，一定要确保集合里的对象，都正确地实现了 hashCode() 和 equals() 这两个方法。

<pre> public static void main(String[] args) { Map<String, String> map = new HashMap<>(); for (int i = 0; i < 10; i++) { map.put("key", "value"); } System.out.println("map size: " + map.size()); } </pre>	<p>使用String的 hashCode()和 String.equals()实 现。</p> <p>结果符合预期： map size: 1</p>
<pre> public static void main(String[] args) { Map<Key, String> keyMap = new HashMap<>(); for (int i = 0; i < 10; i++) { keyMap.put((new Key("key")), "value"); } System.out.println("map size: " + keyMap.size()); } private static class Key { final String key; Key(String key) { this.key = key; } } </pre>	<p>没有实现 hashCode()和 equals()这两个方 法。</p> <p>结果不符合预期： map size: 10</p>
<pre> public static void main(String[] args) { Map<HashedKey, String> hashedKeyMap = new HashMap<>(); for (int i = 0; i < 10; i++) { hashedKeyMap.put((new HashedKey("key")), "value"); } System.out.println("map size: " + hashedKeyMap.size()); } private static class HashedKey { final String key; HashedKey(String key) { this.key = key; } @Override public boolean equals(Object obj) { if (obj == this) { return true; } if (obj instanceof HashedKey) { return key.equals(((HashedKey)obj).key); } } } </pre>	<p>实现了hashCode() 和equals()这两个方 法。</p> <p>结果符合预期： map size: 1</p>

```
        return false;
    }


    @Override
    public int hashCode() {
        return key.hashCode();
    }
}
```

撞车的哈希值

实现 hashCode() 这个方法的，并没有要求不相等对象的返回值也必须是不相等的。但是如果返回的哈希值不同，对集合的性能就会有比较大的影响。

下面的两个基准测试结果显示，如果 10,000 个对象，只有 10 个不同的哈希值，它的集合运算的性能是令人担忧的。和使用了不用哈希值的实现相比，性能有几百倍的差异。

这种性能差异，主要是由基于哈希值的集合的实现方式决定的。哈希值如果相同，就要调用其他的方法来识别一个对象。哈希值如果不同，哈希值本身就可以确定一个对象的索引。如果哈希值撞车比例大，这种检索和计算的差距就会很大。


 复制代码

```
1    // JMH throughput benchmark: about 5,000 operations per second
2    @Benchmark
3    public void measureHashMap() throws IOException {
4        Map<HashedKey, String> map = new HashMap<>();
5        for (int i = 0; i < 10000; i++) {
6            map.put(new HashedKey(i), "value");
7        }
8    }
9
10   private static class HashedKey {
11       final int key;
12
13       HashedKey(int key) {
14           this.key = key;
15       }
16
17       @Override
18       public boolean equals(Object obj) {
19           if (obj == this) {
20               return true;
21           }
22
```

```

23         if (obj instanceof HashedKey) {
24             return key == ((HashedKey)obj).key;
25         }
26
27         return false;
28     }
29
30     @Override
31     public int hashCode() {
32         return key;
33     }
34 }

```

 复制代码

```

1 // JMH throughput benchmark: about 9.5 operations per second
2 @Benchmark
3 public void measureCollidedHashMap() throws IOException {
4     Map<CollidedKey, String> map = new HashMap<>();
5     for (int i = 0; i < 10000; i++) {
6         map.put(new CollidedKey(i), "value");
7     }
8 }
9
10 private static class CollidedKey {
11     final int key;
12
13     CollidedKey(int key) {
14         this.key = key;
15     }
16
17     @Override
18     public boolean equals(Object obj) {
19         if (obj == this) {
20             return true;
21         }
22
23         if (obj instanceof CollidedKey) {
24             return key == ((CollidedKey)obj).key;
25         }
26
27         return false;
28     }
29
30     @Override
31     public int hashCode() {
32         return key % 10;
33     }
34 }

```

小结


今天，我们主要讨论了一些容易被忽略的性能陷阱。比如，字符串怎么操作才是高效的；Java 常见的内存泄漏；资源关闭的正确方法以及集合的相关性能问题。

我们虽然使用了 Java 作为示例，但是像集合和字符串操作这样的性能问题，并不局限于特定的编程语言，你也可以看看你熟悉的编程语言有没有类似的问题。

一起来动手

这一次的练手题，我们来练习使用 JMH 工具，分析更多的性能问题。在“撞车的哈希值”这一小节，我们测试了 HashMap 的 put 方法，你能不能测试下其他方法以及其他基于哈希值的集合（HashSet，Hashtable）？我们测试的是 10,000 个对象，只有 10 个哈希值。如果 10,000 个对象，有 5,000 个哈希值，性能影响有多大？

下面的这段代码，你能够找到它的性能问题吗？

 复制代码

```
1 package com.example;
2
3 import java.util.Arrays;
4 import java.util.Random;
5
6 public class UserId {
7     private static final Random random = new Random();
8
9     private final byte[] userId = new byte[32];
10
11     public UserId() {
12         random.nextBytes(userId);
13     }
14
15     @Override
16     public boolean equals(Object obj) {
17         if (obj == this) {
18             return true;
19         }
20
21         if (obj instanceof UserId) {
22             return Arrays.equals(this.userId, ((UserId)obj).userId);
23         }
24     }
25 }
```




```

25         return false;
26     }
27
28     @Override
29     public int hashCode() {
30         int retVal = 0;
31
32         for (int i = 0; i < userId.length; i++) {
33             retVal += userId[i];
34         }
35
36         return retVal;
37     }
38 }

```

我们前面讨论了下面这段代码的性能问题，你能够使用 JMH 测试一个你的改进方案带来的效率提升吗？

 复制代码

```

1  import java.util.HashMap;
2  import java.util.Map;
3
4  class Solution {
5      /**
6       * Given an array of integers, return indices of the two numbers
7       * such that they add up to a specific target.
8       */
9      public int[] twoSum(int[] nums, int target) {
10         Map<Integer, Integer> map = new HashMap<>();
11         for (int i = 0; i < nums.length; i++) {
12             int complement = target - nums[i];
13             if (map.containsKey(complement)) {
14                 return new int[] { map.get(complement), i };
15             }
16             map.put(nums[i], i);
17         }
18         throw new IllegalArgumentException("No two sum solution");
19     }
20 }
21

```

另外，你也可以检查一下你手头的代码，看看有没有踩到类似的坑。如果遇到类似的陷阱，看一看能不能改进。

容易被忽略的性能陷阱，有很多种。这些大大小小的经验，需要我们日复一日的积累。如果你有这方面的经验，或者看到这方面的技术，请你分享在留言区，我们一起来学习、积累这些经验。

也欢迎点击“请朋友读”，把这篇文章分享给你的朋友或者同事，一起交流一下。



代码精进之路

你写的每一行代码都是你的名片

范学雷

Oracle 首席软件工程师
Java SE 安全组成员
OpenJDK 评审成员



新版升级：点击「 请朋友读」，10位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得转载

上一篇 25 | 使用有序的代码，调动异步的事件

精选留言 (1)

写留言



往事随风，...

2019-03-05

存在拆箱和装箱的转换问题，比较耗费资源

展开 ∨



