链滴

《码出高效》系列笔记（四）：数据结构与集合的框架篇

作者：matthewhan

原文链接：https://ld246.com/article/1584006217728

来源网站：链滴

<img src="https://ld246.com/images/img-loading.svg" alt="EasyCoding.jpg" data-src="https://b3logfile.com/file/2020/03/EasyCoding-9c52a47e.jpg?imageView2/2/interlace/1/format/pg">
<blockquote>
良好的编码风格和完善统一的规约是最高效的方式。
</blockquote>
<h2 id="前言">前言</h2>
本篇汲取了本书中较为精华的知识要点和实践经验加上读者整理，作为本系列里的第四篇章第一：数据结构与集合的框架篇。
本系列目录：

《码高效》系列笔记（一）：面向对象中的类
《码出效》系列笔记（一）：面向对象中的方法
《码出效》系列笔记（一）：面向对象中的其他知识点
《码出效》系列笔记（二）：代码风格
《出高效》系列笔记（三）：异常与日志
《码高效》系列笔记（四）：数据结构与集合的框架
《码高效》系列笔记（四）：数据结构与集合的数组和泛型
《码高效》系列笔记（四）：元素的比较

<h2 id="数据结构">数据结构</h2>
Java 中的集合不同于数学概念，可以是有序的，也可以是重复的。而集合作为数据结构的载体同时也作为程序的主要构成，是所有编程语言的基础。
数据结构的分类：

线性结构：0 至 1 个直接前继和直接后继。当线性非空时，有唯一的首元素和尾元素，除两者外所有的元素都有唯一的直接前继和直接后继。该类结构一般有：顺序表、链表、栈、队列等，其中栈、队列是访问受限的结构。
树结构：0 至 1 个直接前继和 0 至 n 个直接后继（n 大于等于 2）具有层次、稳定的特性，类似大自然的树木。
图结构：0 至 n 个直接前继和直接后继（n 大于等于 2）。该类结一般有：简单图、多重图、有向图、无向图等。
哈希结构：没有直接前继和直接后继。该结构是通过某种特定的哈希函数将索引与存储的值关联来，是一种查找效率非常非常高的数据结构。

复杂度：
数据结构的复杂度分为时间复杂度和空间复杂度两种，因为目前存储设备的技术进步，时间复杂成为了重点考量的因素。

时间复杂度是一种衡量计算性能的指标。通常用大写的 O 和一个函数述，比如 O(n³)表示程序执行时间随输入规模呈现三次倍的增长，这是比较差的算法实现。
从好到坏的常用算法复杂度排序如下：常数级 O(1)、对数级 O(log em>n)、线性级 O(n)、线性对数级 O(n</m>logn)、平方级 O(n²)、立方级 (n³)、指数级 O(2ⁿ。
<h2 id="集合框架">集合框架</h2>
Java 中的集合是用于存储对象的工具类容器，实现了我们上述所说的这些的数据结构，提供了系列的公开方法用于增删改查以及遍历数据，让宁 ⑧ 用自己写轮子辣！
这里本来应该有一张 Java 结合框架图的，不过都应该烂熟于心了 ⑧。除了 <code>Map</cod> 没有直接继承 <code>collection</code> 接口，<code>Queue</code>、<code>List</code>、<code>Set</code> 均是直接继承 <code>collection</code> 接口。
<h2 id="List-集合">List 集合</h2>
List 集合是线性数据结构的主要实现，所以集合的元素通常明确上一个和下一个元素，也存在第个和最后一个元素。
<code>ArrayList</code> 是容量可变的非线程安全集合。内部使用数组行存储，扩容时会创建更大的数组空间，然后再把原来的数据复制到新的数组中。该集合支持对元素随机访问，插入与删除速度通常很慢，因为涉及到移动元素（数组）。
<code>LinkedList</code> 的本质是双向链表。与 <code>ArrayList</code> 相比，插入和除的速度更快，但是随机访问的速度很慢。<code>LinkedList</code> 包含了 3 个重要的成员：siz、first、last。size 是双向链表中节点的个数。first 和 last 分别指向第一个和最后一个节点的应用。
<h2 id="Queue-集合">Queue 集合</h2>
先进先出，一种特殊的线性表，只允许表在一端进行获取操作，在另一端进行插入操作。当不存元素时，则为空队列。自从 <code>BlockingQueue</code>（阻塞队列）问世以来，队列的地位到极大地提升，在各种高并发编程的场景，经常被作为 Buffer（数据缓冲区）使用。
<h2 id="Map-集合">Map 集合</h2>
Map 集合是以 Key-Value 键值对作为存储元素实现的哈希结构，Key 安某种哈希函数计算后是一的，Value 是可以重复的。

可以使用 <code>keySet()</code> 方法查看所有的 Key；
使用 <code>values()</code> 方法查看所有的 Value；
使用 <code>entrySet()</code> 方法查看所有的键值对。

<code>Hashtable</code> 因为性能瓶颈原因已被淘汰，如今广泛使用 <code>HashMap</cde>，但是是线程不安全的，底层也就是数组 + 链表。<code>ConcurrentHashMap</code> 是程是安全的，在 JDK8 中进行了锁的大幅度优化，体现了 8 错的性能。
在多线程并发的场景中，优先推荐使用 <code>ConcurrentHashMap</code>，而不是 <cod>HashMap</code>。
<code>TreeMap</code> 是 Key 有序的 Map 类集合，树结构保证有序，Key ⑧ 能为 <code null</code>。
<code>LinkedHashMap</code> 底层是链表结构，较与 <code>HashMap</code> 可以元素的有序。
<h2 id="Set-集合">Set 集合</h2>
Set 是不允许出现重复元素的集合类型。Set 体系最常用的是 <code>HashSet</code>、<cod>TreeSet</code> 和 <code>LinkedHashSet</code> 三个集合类。
<code>HashSet</code> 与 <code>HashMap</code> 较为类似，只是 Value 固定为一个态对象，使用 Key 保证集合元素的唯一性，但它不保证集合元素的顺序。
<code>TreeSet</code> 也是如此，与 <code>TreeMap</code> 类似，同样底层是树结构保证有序，元素不能为 <code>null</code>。
<code>LinkedHashSet</code> 继承自 <code>HashSet</code>，具有 <code>HashSet<code> 的优点，使用链表维护了元素插入顺序。

<h2 id="集合初始化">集合初始化</h2>
集合初始化通常进行分配容量、设置特定参数等相关工作。
该书中特别强调例如 <code>ArrayList</code> 在初始化的过程中，需要显式地设定集合容量小。
<h2 id="ArrayList-部分源码解析">ArrayList 部分源码解析</h2>
<blockquote>
本书分析的底层源码基本来源于较新的 JDK11，而在我本地的源码是 JDK8，下面分析的是本地的 JDK8 源码。
</blockquote>
<pre><code class="language-java highlight-chroma">/**
 * Increases the capacity to ensure that it can hold at least the
 * number of elements specified by the minimum capacity argument.
 *
 * @param minCapacity the desired minimum capacity
 */
private void grow(int minCapacity) {
    // overflow-conscious code
    int oldCapacity = elementData.length;
    int newCapacity = oldCapacity + (oldCapacity >> 1);
    // ①
    if (newCapacity - minCapacity < 0)
        newCapacity = minCapacity;
    // ②
    if (newCapacity -</code>

an> MAX_ARRAY_SIZE ><
span> 0)
        <span class=
highlight-n">newCapacity = <span class="highl
ght-n">hugeCapacity(<span class="highlight-n
>minCapacity);
    <span class="h
ghlight-c1">// minCapacity is usually close to size, so this is a win:
<span cla
s="highlight-c1">   elementData <span class="
ighlight-o">= Arrays
copyOf(<s
an class="highlight-n">elementData, <span clas
="highlight-n">newCapacity);
<span class="high
ight-o">}
<span class="high
ight-kd">private static <span class="highlight-
t">int hugeCapacity
int minCapacity</s
an>) {
  <span class="hi
hlight-k">if (minCap
city < 0
span class="highlight-o">) // overflow
<span cla
s="highlight-c1">   throw <span class="highlig
t-k">new OutOfMemoryError<span class="highl
ght-o">();
  <span class="hi
hlight-k">return (mi
Capacity > MAX_
RRAY_SIZE) ?</span

    <span class="h
ghlight-n">Integer.<span class="highlight-na"
MAX_VALUE :
      <span class="
ighlight-n">MAX_ARRAY_SIZE;
<span class="high
ight-o">}
</code></pre>
上述代码是 JDK8 中的源码，扩容在 <code>grow()</code> 方法里完成。而在 JDK11 中，
容的具体实现则是由 <code>newCapacity()</code> 方法实现。
首先我们明确几个概念：

<code>oldCapacity</code>：当前容量，由于要扩容了所以是老的容量数值；
<code>newCapacity</code>：扩容后的容量大小；
<code>minCapacity</code>：扩容的必须满足最小的要求！源码是 <code>size + 1</code
，也就是当前的容量 + 1。

什么时候扩容？

JDK8 的源码部分没贴上来，调用的方法太多了占内容空间。如果原始容量为 10，当第 11 个元即将调用 <code>add()</code> 方法时会启动 <code>grow()</code> 方法启动扩容机制，JDK11 同理。

默认的容量大小是什么？

如果没有显式的初始化容量大小，那么在最开始，容量大小其实是 0 而不是默认的 10 哦。当显式的设定了容量大小，那么容量大小赋设定的值。只有当调用 <code>add()</code> 方法时才会启动扩容，变成默认 <code>DEFAULT CAPACITY = 10</code>，大小为 10。所以不显式的初始化容量大小，调用 <code>add()</code> 方法的话时必定会扩容一次的。

<pre><code class="language-java highlight-chroma">List<String> list = new ArrayList<>();
Class<?> listClazz = list.getClass();
Field elementData = listClazz.getDeclaredField("elementData");
elementData.setAccessible(true);
Object[] objects1 = (Object[])elementData.get(list);
System.out.println("不显式的初始化，容量大小为：" + objects1.length);
</code></pre>

上述代码输出 <code>不显式的初始化，容量大小为：0</code>

<pre><code class="language-java highlight-chroma">// add一个元素
<span cla

s="highlight-c1">list
add(<span
class="highlight-s">"回家做80个俯卧撑！");
<span class="high
ight-n">Object[] obj
cts2 = (<sp
n class="highlight-n">Object[])<span class="hi
hlight-n">elementData.<span class="highlight-
a">get(list<
pan class="highlight-o">);
<span class="high
ight-n">System.ou
.println<s
an class="highlight-o">("添加一个元素，现在容量大小为:
 + objects2
span class="highlight-o">.length<span class="
ighlight-o">);
</code></pre>

接着添加一个元素，运行输出 <code>添加一个元素，现在容量大小为：10</code>

<pre><code class="language-java highlight-chroma"><span c
ass="highlight-cl">List<</
pan>String> <sp
n class="highlight-n">list2 = <span class="highl
ght-k">new ArrayList<span class="highlight-o"
<>(666);</sp
n>
<span class="high
ight-n">Class<?> <span class="highlight-n
>newListClazz = list
.getClass<
pan class="highlight-o">();
<span class="high
ight-n">Field elementData2 <span class="highli
ht-o">= newListClazz<span class="highlight-o"
.getDeclaredField(<
span>"elementData");</span

<span class="high
ight-n">elementData2.<span class="highlight-n
">setAccessible(tru
);
<span class="high
ight-n">Object[] obj
cts3 = (<sp
n class="highlight-n">Object[])<span class="hi
hlight-n">elementData.<span class="highlight-
a">get(list2
span class="highlight-o">);
<span class="high
ight-n">System.ou
.println<s
an class="highlight-o">("显式初始化容量大小为666，容量

小为：" + objects3</pan>.length);
</code></pre>

显式初始化容量大小为 666，运行输出 <code>显式初始化容量大小为666，容量大小为：666<code>

如何扩容？

JDK8 之前的扩容算法与 JDK8 有所不同，源码中上述方法里，<code>int newCapacity = oldCpacity + (oldCapacity >> 1);</code> 扩容后的容量大小算法是通过右移一位再加上老容量小得到。

位移运算：JDK8 中采用了（有符号）位运算符计算，位移的过程中采用补码的形式。假设原始量为 13，二进制数是 1101，其中反码也是 1101，整数的反码、补码都是本身反码右移一位这是 110，得到十进制为 6，所以新的容量大小为 6 + 13 = 19。JDK7 之前的公式则是<code>oldCapacity * 1.5 + 1</code>；

①：如果新的容量大小比最小要求要小的话，则按照最小要求设定（size + 1）；
②：如果新的容量大小比 <code>MAX_ARRAY_SIZE</code> 还大的话，其中该变量的大小为code>Integer.MAX_VALUE - 8</code> 也就是 2³¹-1 再减 8。在走 <code>hugeapacity(int minCapacity)</code> 方法判断最后设定一个容量大小或者 OOM 了。

之所以本手册中明确强调需要显式的初始化容量大小，是因为假设有 1000 个元素需要放置在 ArayList 中，则需要被动扩容 13 次才可以完成，在这过程中数组不断地复制原有的数据再到新的数组，若能够提前给定一个合适的容量大小，就是性能的提升，也避免了一些 OOM 的风险。OS：这过更像是调优，实际开发中很难对每个 ArrayList 清晰的定义和认识吧，属于经验学的范畴。
<blockquote>
嗯？！感觉关于 ArrayList 的可以单独出一篇啊。
</blockquote>
<h2 id="HashMap-部分源码解析">HashMap 部分源码解析</h2>
<blockquote>
本书分析的底层源码基本来源于较新的 JDK11，而在我本地的源码是 JDK8，下面分析的是本地的JDK8 源码。
</blockquote>
<pre><code class="language-java highlight-chroma">/**
  * The default initial capacity - MUST be a power of two.
  */
static final int DEFAULT_INITIAL_CAPACITY = 1 &lt< 4; <spn class="highlight-c1">// aka 16

/**
 * The maximum capacity, used if a higher value is implicitly specified
 * by either of the constructors with arguments.
 * MUST be a power of two <= 1<<30.
 */
static final int MAXIMUM_CAPACITY = 1 << 30;

/**
 * The load factor used when none specified in constructor.
 */
static final float DEFAULT_LOAD_FACTOR = 0.75f;

/* ---------------- Fields -------------- */
/**
  * The table, initialized on first use, and resized as
  * necessary. When allocated, length is always a power of two.
  * (We also tolerate length zero in some operations to allow
  * bootstrapping mechanics that are currently not needed.)
  */
transient Node<K,V>[] table;
</code></pre>

<code>DEFAULT_INITIAL_CAPACITY</code>：默认初始容量 <code>1 << 4</code> aka 16，必须是 2ⁿ幂，注解里很直白的写着，所以你即使在初始化的过程这样写 <code>new HashMap<>(3);</code>，实际上也会被改成 2²，比 3 大且最近的一 2 的幂次方；
<code>MAXIMUM_CAPACITY</code>：最大容量大小，默认 <code>1 << 30</code>

，相当于 2³⁰；
<code>DEFAULT_LOAD_FACTOR</code>：负载因子，也叫填充比例，默认 0.75，可在初化过程中自定一个 float 类型的数值。
<code>table</code>：

HashMap 和 ArrayList 一样，容量不是在 new 的时候分配，而是在第一次 <code>put</code 的时候。
<code>put</code>、<code>putVal()</code>、<code>reszie()</code> 方法有些晦涩杂就不贴上来了。很多细节，:older_man: 属实有点看不明白。这里有个概念 <code>threshold</cde> 作为临界值就是 <code>loadfactory</code>（负载因子）和 <code>capacity</code>（容）的乘积。也就是说默认情况下，当 HashMap 中元素个数达到了容量的 3/4 的时候就会进行自动扩。

为什么负载因子是 0.75？

JDK 源码里这样写，一定有它的道理，这里不太想去探究。查阅网上的资料，和哈希冲突有很大系，以及一些数学计算、log2、对数之类的有一定关系，反正一般不要去自己去设定就 vans 了。</li

什么时候扩容？

和 ArrayList 一样，到了某个临界值才会被动扩容，而且扩容的过程中会重新计算所有元素的哈值。扩容的条件是达到了 <code>threshold</code> 这个参数，而它是 <code>capacity</code> 和 <code>loadfactory</code> 的乘积，所以我们可以通过代码来验证一下：

<pre><code class="language-java highlight-chroma">Map<String, String> map = new/span> HashMap<>(</pan>0);
Class<?> <span class="highlight-n>mapClazz = map</pan>.getClass<spa class="highlight-o">();
Method capacity = mapClazz.</san>getDeclaredMethod(</san>"capacity");
capacity.seAccessible(true</spn>);
System.ou.println<san class="highlight-o">("容量大小为：" + capacity.invoke(</spn>map));

</code></pre>

输出 <code>容量大小为：1</code>，因为是 2⁰

<pre><code class="language-java highlight-chroma"><span c
ass="highlight-cl">map.</sp
n>put(<span class
"highlight-s">"MatthewHan", <span class="high
ight-s">"developer");
<span class="high
ight-n">System.ou
.println<s
an class="highlight-o">("添加一个元素后，容量大小为："<
span> + capacity<s
an class="highlight-o">.invoke<span class="hi
hlight-o">(map));</
pan>
</code></pre>

put 一个元素之后，查看 <code>capacity</code> 大的大小，输出 <code>添加一个元素后
容量大小为：2</code>。为什么不是 1 呢，因为 <code>capacity</code> 和 <code>loadfactor
</code> 的乘积是 <code>0.75 * 1 < 1</code>，满足扩容的条件。所以从 2⁰
容成了 2¹。当然你这样写 <code>new HashMap<>(0, 1f);</code> 输出的
是 1 了。

以前看到过的一道美团笔试题：如果一个 HashMap 要装载 100 个元素，那么初始化
量大小是设定最佳？

最佳的大小，一定是满足不会多次扩容调用 <code>resize()</code> 方法的。所以就是一定要
于 <code>100 ÷ 0.75 = 133.333...</code>，比该数大且最接近的 2ⁿ的是 2<sup>
</sup> = 256。而 2⁷ = 128 看起来比 100 大，但是需要多扩容一次，全部重新计
哈希算法，属实 ⑧ 行。上面写了目前对时间性能的要求远远大于空间，用空间换时间。
或者可以这样 <code>new HashMap<>(128, 0.79f);</code> 但是一般不会改变负载因
的值，该方法实际表现未知。

