

关于后续更新

后续所有更新均在微信公众号统一发布：扫描下方二维码关注，不迷路😁



掌握这些，ArrayList就不用担心了！

关于ArrayList的学习

ArrayList属于Java基础知识，面试中会经常问到，所以作为一个Java从业者，它是你不得不掌握的一个知识点。😁

可能很多人也不知道自己学过多少遍ArrayList，以及看过多少相关的文章了，但是大部分人都是当时觉得自己会了，过不了多久又忘了，真的到了面试的时候，自己回答的支支吾吾，自己都不满意😞

为什么会这样？对于ArrayList这样的知识点的学习，不要靠死记硬背，你要做的是真的理解它！😁

我这里建议，如果你真的想清楚的理解ArrayList的话，可以从它的构造函数开始，一步步的读源码，最起码你要搞清楚add这个操作，记住，是源码😁

一个问题看看你对ArrayList掌握多少

很多人已经学习过ArrayList了，读过源码的也不少，这里给出一个问题，大家可以看看，以便测试下自己对ArrayList是否真的掌握：

请问在ArrayList源码中
DEFAULTCAPACITY_EMPTY_ELEMENTDATA和
EMPTY_ELEMENTDATA是什么？它们有什么区别？

怎么样？如果你能很轻松的回答上来，那么你掌握的不错，不想再看本篇文章可以直接出门右拐（我也不知道到哪），如果你觉得不是很清楚，那就跟着我继续往下，咱们再来把ArrayList中那些重点过一遍！😁

你觉得ArrayList的重点是啥？

在我看来，ArrayList的一个相当重要的点就是**数组扩容技术**，我们之前学习过数组，想一下数组是个什么玩意以及它有啥特点。

随机访问，连续内存分布等等，这些学过的都知道，这里说一个似乎很容易被忽略的点，那就是数组的删除，想一下，数组怎么做删除？😞

关于数组删除的一些思考

关于数组的删除，我之前也是有疑惑，后来也花时间思考了一番，算是比较通透了，这里就提一点，数组并没有提供删除元素的方法，我们都是怎么做删除的？

比如我们要删除中间的一个元素，怎么操作，首先我们可以把这个元素置为null，也就把这个元素删除掉了，此时数组上就空出了一个位置，这样行吗？

当我们再次遍历这个数组的时候是不是还是会遍历到这个位置，那么就会报空指针异常，怎么办？是的我们可以先判断，但是这样的做法不好，怎么办呢？

那就是我们可以把这个元素后面的所有元素统一的向前复制，有的地方这里会说移动，我觉得不够合理，为啥？

复制是把一个元素拷贝一份放到其他位置，原来位置元素还存在，而移动呢？区别就是移动了，原本的元素就不存在了，而数组这里是复制，把元素统一的各自向前复制，最终结果就是倒数第一和第二位置上的元素是相同的。

此时的删除的本质实际上是要删除的这个元素的后一个元素把要删除的这个元素给覆盖了，后面依次都是这样的操作，可能有点绕，自己想一下。

所以就引出了数组的删除操作是要进行数组元素的复制操作，也就导致数组删除操作最坏的时间复杂度是 $O(n)$ 。

为什么说这个？因为对理解数组扩容技术很有帮助！

数组扩容技术

上面我们谈到了关于数组的删除操作，我们只是分析了该如何去删除，但是数组并未提供这样的方法，如果我们要搞个数组，这个删除操作还是要我们自己写代码去实现的。

不过好在已经有实现了，谁嘞，就是我们今天的主角ArrayList，其实ArrayList就可以看作是数组的一个升级版，ArrayList底层也是使用数组来实现，然后加上了很多操作数组的方法，比如我们上面分析的删除操作，当然除此之外，还实现了一些其他的方法，然后这就形成了一个新的物种，这就是ArrayList。

本质上ArrayList就是一个普通的类，对数组进行的封装，扩展其功能

对于数组，我们还了解一点那就是数组一旦确定就不能再被改变，而这个ArrayList却可以实现自动扩容，有木有觉得很高级，其实也没啥，因为数组本身特性决定，ArrayList所谓的自动扩容其实也是新创建一个数组而已，因为ArrayList底层就是使用的数组。

我们的重点需要关注的是这个自动扩容的过程，就是怎么创建一个新的数组，创建完成之后又是怎么做的，这才是我们关注的重点。

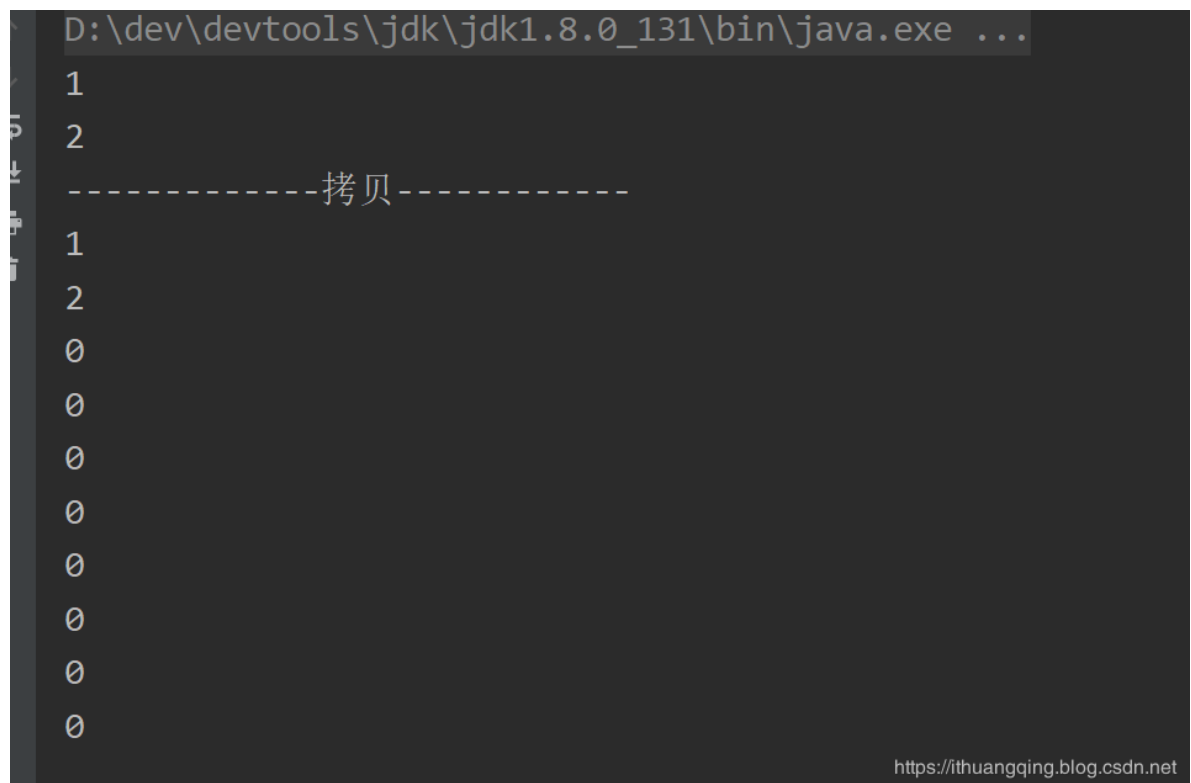
接下来我们看两种数组扩容方式。

Arrays.copyOf

不知道你使用过没，我们直接看代码：

```
public static void main(String[] args) {  
    int[] a1 = new int[]{1, 2};  
    for (int a : a1) {  
        System.out.println(a);  
    }  
    System.out.println("-----拷贝-----");  
    int[] b1 = Arrays.copyOf(a1, 10);  
    for (int b : b1) {  
        System.out.println(b);  
    }  
}
```

代码不多，很简单，看看输出结果你就明白了



```
D:\dev\devtools\jdk\jdk1.8.0_131\bin\java.exe ...  
1  
2  
-----拷贝-----  
1  
2  
0  
0  
0  
0  
0  
0  
0  
0  
0
```

<https://ithuangqing.blog.csdn.net>

ok，是不是很简单，知道这个简单用法就ok了，接下来看另外一种

System.arraycopy()

这个方法我们看看是个啥：

```
public static native void arraycopy(Object src,  int  srcPos,  
                                     Object dest, int destPos,  
                                     int length);
```

看见没，native修饰的，一般是使用c/c++写的，性能很高，我们看看这里的这几个参数都是啥意思：

src：要拷贝的数组
srcPos：要拷贝的数组的起始位置
dest：目标数组
destPos：目标数组的起始位置
length：你要拷贝多少个数据

怎么样，知道这几个参数什么意思了，那使用就简单了，我这里就不显示了。

ps：以后复制数组别再傻傻的遍历了，用这个多香😋

以上两个方法都是进行数组拷贝的，这个对理解数组扩容技术很重要，而且在ArrayList中也有应用，我们等会会详细说。

下面咱们开始看看ArrayList的一些源码，加深我们对ArrayList的理解！

源码中的ArrayList

一般我们是怎么用ArrayList的呢？看下面这些代码：

```
ArrayList arrayList = new ArrayList();  
    arrayList.add("hello");  
    arrayList.add(1);  
  
ArrayList<String> stringArrayList = new ArrayList<>();  
    stringArrayList.add("hello");
```

简单，都会吧，就是new一个出来，不过上面的代码我还想说明一个问题，当你不指定具体类型的时候是可以存储任意类型的数据的，指定的话就只能存储特定类型，为啥不指定可以存储任意类型？

这个问题不做解释，等会看源码你就明白了。

看看ArrayList的无参构造函数

一般我们看ArrayList的源码，都是从它的无参构造函数开始看起的，也就是这个：

```
new ArrayList();
```

好啦，走进去看看这个new ArrayList();构造函数长啥样吧。

```
/**
 * Constructs an empty list with an initial capacity of ten.
 */
public ArrayList() {
    this.elementData = DEFAULTCAPACITY_EMPTY_ELEMENTDATA;
}
```

咋一看，代码不多，简单，里面就是个赋值操作啊，有两个新东西elementData和DEFAULTCAPACITY_EMPTY_ELEMENTDATA，这是啥？🤔

不着急，我们点进去看看

```
private static final Object[] DEFAULTCAPACITY_EMPTY_ELEMENTDATA = {};
transient Object[] elementData; // non-private to simplify nested
class access
```

这不就是Object数组嘛，好像还真的是，那transient啥意思？它啊，你就记住被它修饰序列化的时候会被忽略掉。

好了，除此之外，就是个数组，对Object类型的。

不好像有点区别啊，DEFAULTCAPACITY_EMPTY_ELEMENTDATA已经指定是个空数组了，而elementData只是声明，在new一个ArrayList的时候进行了赋值，也就是这样：

```
this.elementData = DEFAULTCAPACITY_EMPTY_ELEMENTDATA;
```

咋样？明白了吧，之前不就说了嘛，ArrayList底层就是一个数组的，这里你看，new之后不就给你弄个空数组出来嘛，也就是说啊，你要使用ArrayList，一开始先new一下，然后给你搞个空数组出来。

啥？空数组？空数组怎么行呢？毕竟我们还需要用它存数据嘞，所以啊，重点来了，我们看它的add，也就是添加数据的操作。

看看ArrayList的add

```
public boolean add(E e) {  
    ensureCapacityInternal(size + 1); // Increments modCount!!  
    elementData[size++] = e;  
    return true;  
}
```

就是这个啦，ArrayList不就是使用add来添加数据嘛，我们看看是怎么操作的，咋一看这段代码，让我们感到比较陌生的就是这个方法了

```
ensureCapacityInternal(size + 1);
```

这是啥玩意，翻译一下😁



确保内部容量？什么鬼，这里还有个size，我们看看是啥？

```
private int size;
```

就是一个变量啊，我们再看看这段代码

```
public boolean add(E e) {  
    ensureCapacityInternal(size + 1); // Increments modCount!!  
    elementData[size++] = e;  
    return true;  
}
```

尤其是

```
elementData[size++] = e;
```

知道了嘛？我们之前不是已经创建了一个空数组，不就是elementData嘛，这好像是在往数组里面放数据啊，不过不对啊，不是空数组嘛？咋能放数据，这不是前面还有这一步嘛

```
ensureCapacityInternal(size + 1);
```

是不是有想法了，这一步应该就是把数组的容量给确定下来的，赶紧进去看看

```
private void ensureCapacityInternal(int minCapacity) {  
    if (elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA) {  
        minCapacity = Math.max(DEFAULT_CAPACITY, minCapacity);  
    }  
  
    ensureExplicitCapacity(minCapacity);  
}
```

就是这个了，这一步很重要：

```
if (elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA) {  
    minCapacity = Math.max(DEFAULT_CAPACITY, minCapacity);  
}
```

也好理解吧，就是先判断下现在这个ArrayList的底层数组elementData 是不是刚创建的的空数组，这里肯定是啊，然后开始执行

```
minCapacity = Math.max(DEFAULT_CAPACITY, minCapacity);
```

minCapacity是个啥（重要）

说这个之前，你先得搞清楚这个minCapacity 是啥，它现在其实就是底层数组将要添加的第几个元素，看看上一步

```
ensureCapacityInternal(size + 1);
```

这里size+1了，所以现在minCapacity 相当于是1，也就是说将要向底层数组添加第一个元素，这一点理解很重要，所以从minCapacity 的字面意思理解也就是“最小容量”，我现在将要添加第一个元素，那你至少给我保证底层数组有一个空位置，不然怎么放数据嘞。

重点来了，因为第一次添加，底层数组没有一个位置，所以需要先确定下来一共有多少个位置，就是献给数组一个默认的长度

于是这里给重新赋值了（只有第一次添加数据才会执行这步，这一步就是为了指定默认数组长度的，指定一次就ok了）

```
minCapacity = Math.max(DEFAULT_CAPACITY, minCapacity);
```

这怎么赋值的应该知道嘛，哪个大取哪个，那我们要看看DEFAULT_CAPACITY 是多少了

```
/**
 * Default initial capacity.
 */
private static final int DEFAULT_CAPACITY = 10;
```

ok，明白了，这就是ArrayList的底层数组elementData初始化容量啊，是10，记住了哦，那么现在minCapacity就是10了，我们再接着看下面的代码，也即是：

```
ensureExplicitCapacity(minCapacity);
```

进去看看吧：

```
private void ensureExplicitCapacity(int minCapacity) {  
    modCount++;  
  
    // overflow-conscious code  
    if (minCapacity - elementData.length > 0)  
        grow(minCapacity);  
}
```

也比较简单，现在底层数组长度肯定还不到10啊，所以我们继续看grow方法

```
private void grow(int minCapacity) {  
    // overflow-conscious code  
    int oldCapacity = elementData.length;  
    int newCapacity = oldCapacity + (oldCapacity >> 1);  
    if (newCapacity - minCapacity < 0)  
        newCapacity = minCapacity;  
    if (newCapacity - MAX_ARRAY_SIZE > 0)  
        newCapacity = hugeCapacity(minCapacity);  
    // minCapacity is usually close to size, so this is a win:  
    elementData = Arrays.copyOf(elementData, newCapacity);  
}
```

咋一看，判断不少啊，干啥的都是，突然看到了Arrays.copyOf，知道这是啥吧，上面可是特意讲过的，原来这是要进行数组拷贝啊，那这个elementData就是原来的数组，newCapacity就是新数组的容量

我们一步步来看代码，首先是

```
int oldCapacity = elementData.length;
```

得到原来数组的容量，接着下一步：

```
int newCapacity = oldCapacity + (oldCapacity >> 1);
```

这是得到新容量的啊，不过后面的这个oldCapacity >> 1有点看不懂啊，其实这oldCapacity >> 1就相当于oldCapacity / 2，这是移位运算，感兴趣的自行搜索学习。

知道了，也就是扩容为原来的1.5倍，接下来这一步：

```
if (newCapacity - minCapacity < 0)
    newCapacity = minCapacity;
```

因为目前数组长度为0，所以这个新的容量也是0，而minCapacity 则是10，所以会执行方法体内的赋值操作，也就是现在的新容量成了10。

接着这句代码就知道怎么回事了

```
elementData = Arrays.copyOf(elementData, newCapacity);
```

不知道你发现没，这里饶了一大圈，就是为了创建一个默认长度为10的底层数组。

底层数组长度要看ensureCapacityInternal

ensureCapacityInternal这个方法就像个守卫，时刻监视着数组容量，然后过来一个数值，也就是说要向数组添加第几个数据，那ensureCapacityInternal需要思考思考了，思考啥呢？当然是看底层数组有没有这么大容量啊，比如你要添加第11个元素了，那底层数组长度最少也得是11啊，不然添加不了啊，看它是怎么把关的

```
private void ensureCapacityInternal(int minCapacity) {
    if (elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA) {
        minCapacity = Math.max(DEFAULT_CAPACITY, minCapacity);
    }

    ensureExplicitCapacity(minCapacity);
}
```

记住了这段代码

```
if (elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA) {
    minCapacity = Math.max(DEFAULT_CAPACITY, minCapacity);
}
```

它的存在就是为了一开始创建默认长度为10的数组的，当添加了一个数据之后就不会再执行这个方法，所以重难点是这个方法：

```
ensureExplicitCapacity(minCapacity);
```

也就是真正的把关在这里，看它的实现：

```
private void ensureExplicitCapacity(int minCapacity) {  
    modCount++;  
  
    // overflow-conscious code  
    if (minCapacity - elementData.length > 0)  
        grow(minCapacity);  
}
```

怎么样，看明白了吧，比如你要添加第11个元素，可是我的底层数组长度只有10，不够啊，然后执行grow方法，干嘛执行这个方法，它其实就是用来扩容的，不信你再看看它的实现，上面已经分析过了，这里就不说了。

假如你要添加第二个元素，这里底层数组长度为10，就不需要执行grow方法，因为根本不需要扩容啊，所以这一步实际啥也没做（有个计数操作）：

```
public boolean add(E e) {  
    ensureCapacityInternal( minCapacity: size + 1); // Increments modCount!!  
    elementData[size++] = e;  
    return true;  
}
```

<https://ithuangqing.blog.csdn.net>

然后就直接在相应位置赋值了。

小结

所以这里很重要的一点就是理解这一步传入的值的意义：

```
ensureCapacityInternal(size + 1);
```

简单点就是要向底层数组中添加第几个元素了，然后开始进行一系列的判断，容量够的话直接返回，直接赋值，不够的话就执行grow方法开始扩容。

主要判断就在这里：

```
private void ensureExplicitCapacity(int minCapacity) {  
    modCount++;  
  
    // overflow-conscious code  
    if (minCapacity - elementData.length > 0)  
        grow(minCapacity);  
}
```

具体的扩容是这里

```
private void grow(int minCapacity) {  
    // overflow-conscious code  
    int oldCapacity = elementData.length;  
    int newCapacity = oldCapacity + (oldCapacity >> 1);  
    if (newCapacity - minCapacity < 0)  
        newCapacity = minCapacity;  
    if (newCapacity - MAX_ARRAY_SIZE > 0)  
        newCapacity = hugeCapacity(minCapacity);  
    // minCapacity is usually close to size, so this is a win:  
    elementData = Arrays.copyOf(elementData, newCapacity);  
}
```

这里需要注意这段代码

```
if (newCapacity - minCapacity < 0)  
    newCapacity = minCapacity;
```

这段代码只有在第一次添加数据的时候才会执行，也是为创建默认长度为10的数组做准备的，因为这个时候原本数组长度为0，扩容后也是0，而minCapacity 为默认值10，所以会执行这段代码。

但是一旦添加数据之后，底层数组默认就是10了，再加上之前的判断，这里的新newCapacity 一定会比minCapacity 大，这个点需要了解。

看看ArrayList的有参构造函数

我们上面着重分析了ArrayList的无参构造函数，下面再来看看它的有参构造函数：

```
ArrayList arrayList1 = new ArrayList(100);
```

看看这个构造函数长啥样？

```
public ArrayList(int initialCapacity) {  
    if (initialCapacity > 0) {  
        this.elementData = new Object[initialCapacity];  
    } else if (initialCapacity == 0) {  
        this.elementData = EMPTY_ELEMENTDATA;  
    } else {  
        throw new IllegalArgumentException("Illegal Capacity: "+  
                                           initialCapacity);  
    }  
}
```

我去，这不就是直接创建嘛，然后还有这个：

```
else if (initialCapacity == 0) {  
    this.elementData = EMPTY_ELEMENTDATA;  
}
```

我们看看这个EMPTY_ELEMENTDATA

```
private static final Object[] EMPTY_ELEMENTDATA = {};  
private static final Object[] DEFAULTCAPACITY_EMPTY_ELEMENTDATA = {};
```

ok，现在你可以回答我们开篇提的那个问题了。

我们以上对ArrayList的源码有了一定的认识之后，我们再来看看ArrayList的读取，替换和删除操作时怎样的？

ArrayList的其他操作

经过上面的分析，我相信你对ArrayList的其他诸如读取删除等操作也没啥问题，一起来看下。

读取操作

[看源码](#)

```
public E get(int index) {  
    rangeCheck(index);  
  
    return elementData(index);  
}
```

代码很简单，rangeCheck就是用来判断数组是否越界的，然后直接返回下标对应的值。

删除操作

[看源码](#)

```
public E remove(int index) {  
    rangeCheck(index);  
  
    modCount++;  
    E oldValue = elementData(index);  
  
    int numMoved = size - index - 1;  
    if (numMoved > 0)  
        System.arraycopy(elementData, index+1, elementData,  
index,  
                                numMoved);  
    elementData[--size] = null; // clear to let GC do its work  
  
    return oldValue;  
}
```

代码相对来说多一些，要理解这个，可以仔细看看我上面对“关于数组删除的一些思考”的分析，这里是一样的道理。

替换操作

[看源码](#)

```
public E set(int index, E element) {  
    rangeCheck(index);  
  
    E oldValue = elementData(index);  
    elementData[index] = element;  
    return oldValue;  
}
```

其实就是把原来的值覆盖，没啥问题吧😊

和vector很像

这个想必大家都知道，ArrayList和vector是很像的，前者是线程不安全，后者是线程安全，我们看一下vector一段源码就明白了

```
public synchronized boolean add(E e) {  
    modCount++;  
    ensureCapacityHelper(elementCount + 1);  
    elementData[elementCount++] = e;  
    return true;  
}
```

没错，区别就是这么明显！

总结

到这里，我们基本上把ArrayList的相关重点都过了一遍，对于ArrayList来说，重点就是分析它的无参构造函数的执行，经过分析，我们知道了它有个数组拷贝的操作，这块是会影响到它的一些操作的时间复杂度的，关于这点，就留给大家去思考吧！

好了，今天就到这里，大家如果有什么问题，欢迎留言，一起交流学习！

害怕面试被问HashMap? 这一篇就搞定了!

搞定HashMap

作为一个Java从业者，面试的时候肯定会被问到过HashMap，因为对于HashMap来说，可以说是Java 集合中的精髓 了，如果你觉得自己对它掌握的还不够好，我想今天这篇文章会非常适合你，至少，看了今天这篇文章，以后不怕面试被问HashMap了

其实在我学习HashMap的过程中，我个人觉得HashMap还是挺复杂的，如果真的想把它搞得明明白白的，没有足够的内力怕是一时半会儿做不到，不过我们总归是在不断的学习，因此真的不必强迫自己把现在遇到的一些知识点全部搞懂。

但是，对于HashMap来说，你所掌握的应该足够可以让你应对面试，所以今天咱们的侧重点就是学会那些经常被问到的知识点。

我猜，你肯定看过不少分析HashMap的文章了，那么你掌握多少了呢？从一个问题开始吧

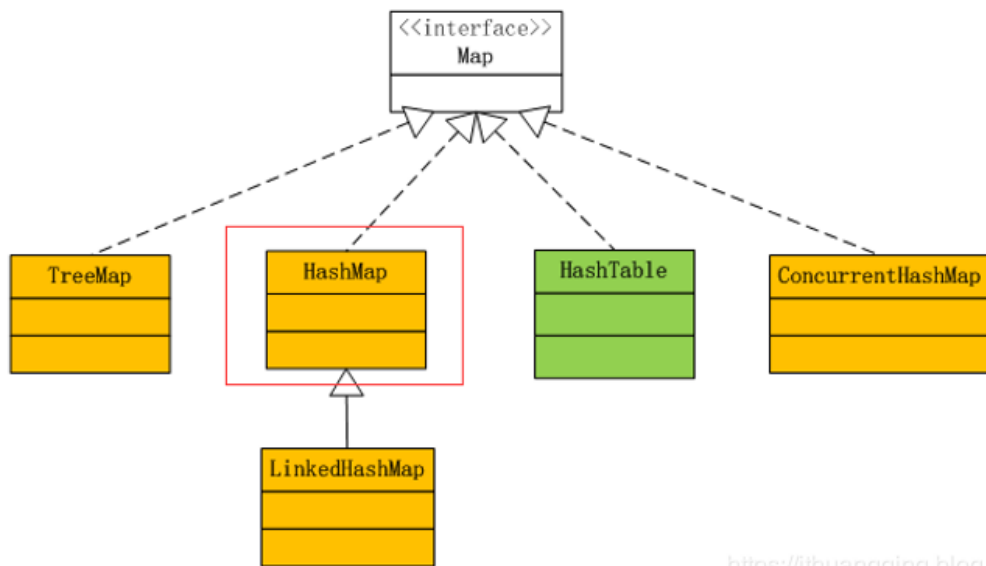
新的节点在插入链表的时候，是怎么插入的?

怎么样，想要回答这个问题，还是需要你对HashMap有个比较深入的了解的，如果仅仅知道什么key和value的话，那么回答这个问题就比较难了。

这个问题大家可以先想想，后面我会给出解答，下面我们一步步的来看HashMap中几个你必须知道的知识点。

Map是个啥?

HashMap隶属于Java中集合这一块，我们知道集合这块有list，set和map，这里的HashMap就是Map的实现类，那么在Map这个大家族中还有哪些重要角色呢?



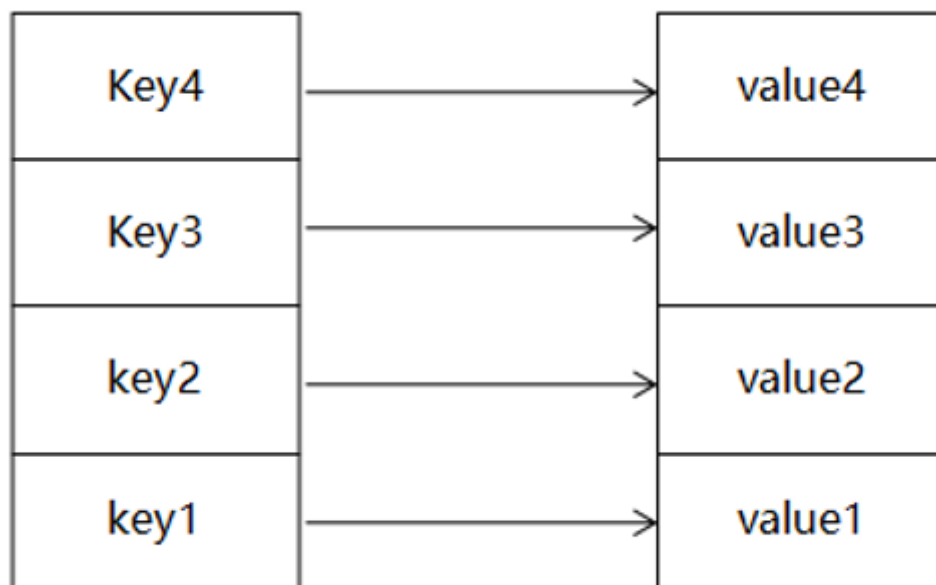
<https://ithuangqing.blog.csdn.net>

上图展示了Map的家族，都是狠角色啊，我们对这些其实都要了解并掌握，这里简单的介绍下这几个狠角色：

TreeMap从名字上就能看出来是与树有关，它是基于树的实现，而HashMap，HashTable和ConcurrentHashMap都是基于hash表的实现，另外这里的HashTable和HashMap在代码实现上，基本上是一样的，还记得之前在讲解ArrayList的时候提到过和Vector的区别嘛？这里他们是很相似的，一般都不怎么用HashTable，会用ConcurrentHashMap来代替，这个也需要好好研究，它比HashTable性能更好，它的锁粒度更小。

由于这不是本文的重点，只做简单说明，后续会发文单独介绍。

简单来说，Map就是一个映射关系的数据集合，就是我们常见的k-v的形式，一个key对应一个value，大致有这样的图示



<https://ithuangqing.blog.csdn.net>

这只是简单的概念，放到具体的实例当中，比如在HashMap中就会衍生出很多其他的问题，那么HashMap又是个啥？

HashMap是个啥

上面简单提到过，HashMap是基于Hash表的实现，因此，了解了什么是Hash表，那对学习HashMap是相当重要。

之前特意写了一篇介绍哈希表的，不了解的赶紧去看看：[来吧！一文彻底搞定哈希表！](#)

建议了解了哈希表之后再学习HashMap，这样很多难懂的也就不那么难理解了。

接着，HashMap是基于hash表的实现，而说到底，它也是用来存储数据供我们使用的，那么底层是用什么来存储数据的呢？可能有人猜到了，还是数组，为啥还是数组？想想之前的ArrayList，怎么，对ArrayList也不了解。

没事，刚好我也写了一篇：[掌握这些，ArrayList就不用担心了！](#)

所以，对于HashMap来说，底层也是基于数组实现，只不过这个数组可能和你印象中的数组有些许不同，我们平常整个数组出来，里面会放一些数据，比如基础数据类型或者引用数据类型，数组中的每个元素我们没啥特殊的叫法。

但是在HashMap中人家就有了新名字，我发现这个知识点其实很多人都不太清楚：

在HashMap中的底层数组中，每个元素在jdk1.7及之前叫做Entry，而在jdk1.8之后人家又改名叫做Node。

这里可能还是会有人好奇这Entry和Node长啥样，这个看看源码就比较清楚了，后面我们会说。

到了这里你因该就能简单的理解啥是HashMap了，如果你看过什么是哈希表了，你就会清楚，在HashMap中同样会出现哈希表所描述的那些问题，比如：

1. 如何确定添加的元素在底层数组的哪个位置？
2. 怎么扩容？
3. 出现冲突了怎么处理？
4. 。。。

没事，这些问题我们后续都会谈到。

HashMap初始化大小是多少

先来看HashMap的基础用法：

```
HashMap map = new HashMap();
```

就这样，我们创建好了一个HashMap，接下来我们看看new之后发生了什么，看看这个无参构造函数吧

```
public HashMap() {  
    this.loadFactor = DEFAULT_LOAD_FACTOR; // all other fields  
    defaulted  
}
```

解释下新面孔：

1. loadFactor：负载因子，之前聊哈希表的时候说过这个概念
2. DEFAULT_LOAD_FACTOR：默认负载因子，看源码知道是0.75

很简单，当你新建一个HashMap的时候，人家就是简单的去初始化一个负载因子，不过我们这里想知道的是底层数组默认是多少嘞，显然我们没有得到我们的答案，我们继续看源码。

在此之前，想一下之前ArrayList的初始化大小，是不是在add的时候才创建默认数组，这里会不会也一样，那我们看看HashMap的添加元素的方法，这里是put

```
public V put(K key, V value) {  
    return putVal(hash(key), key, value, false, true);  
}
```

这里大眼一看，有两个方法：

1. putVal 重点哦
2. hash

这里需要再明确下，这是我们往HashMap中添加第一个元素的时候，也就是第一次调用这个put方法，可以猜想，现在数据已经过来了，底层是不是要做存储操作，那肯定要弄个数组出来啊，好，离我们想要的结果越来越近了。

先看这个hash方法：

```
static final int hash(Object key) {  
    int h;  
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);  
}
```

记得之前聊哈希表的时候说过，哈希表的数据存储有个很明显的特点，就是根据你的key使用哈希算法计算得出一个下标值，对吧，不懂得赶紧看：[来吧！一文彻底搞定哈希表！](#)

而这里的hash就是根据key得到一个hash值，并没有得到下标值哦。

重点要看这个putVal方法，可以看看源码：

```
final V putVal(int hash, K key, V value, boolean onlyIfAbsent,  
               boolean evict) {  
    Node<K,V>[] tab; Node<K,V> p; int n, i;  
    if ((tab = table) == null || (n = tab.length) == 0)  
        n = (tab = resize()).length;
```

```

        if ((p = tab[i = (n - 1) & hash]) == null)
            tab[i] = newNode(hash, key, value, null);
        else {
            Node<K,V> e; K k;
            if (p.hash == hash &&
                ((k = p.key) == key || (key != null &&
key.equals(k))))
                e = p;
            else if (p instanceof TreeNode)
                e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash,
key, value);
            else {
                for (int binCount = 0; ; ++binCount) {
                    if ((e = p.next) == null) {
                        p.next = newNode(hash, key, value, null);
                        if (binCount >= TREEIFY_THRESHOLD - 1) // -1
for 1st
                            treeifyBin(tab, hash);
                        break;
                    }
                    if (e.hash == hash &&
                        ((k = e.key) == key || (key != null &&
key.equals(k))))
                            break;
                    p = e;
                }
            }
            if (e != null) { // existing mapping for key
                V oldValue = e.value;
                if (!onlyIfAbsent || oldValue == null)
                    e.value = value;
                afterNodeAccess(e);
                return oldValue;
            }
        }
        ++modCount;
        if (++size > threshold)
            resize();
        afterNodeInsertion(evict);
        return null;
    }

```

咋样，是不是感觉代码一下变多了，我们这里逐步的有重点的来看，先看这个：

```
if ((tab = table) == null || (n = tab.length) == 0)
    n = (tab = resize()).length;
```

这个table是啥？

```
transient Node<K,V>[] table;
```

看到了，这就是HashMap底层的那个数组，之前说了jdk1.8中数组中的每个元素叫做Node，所以这就是个Node数组。

那么上面那段代码啥意思嘞？其实就是我们第一次往HashMap中添加数据的时候，这个Node数组肯定是null，还没创建嘞，所以这里会去执行resize这个方法。

resize方法的主要作用就是初始化和增加表的大小，说白了就是第一次给你初始化一个Node数组，其他需要扩容的时候给你扩容

看看源码：

```
final Node<K,V>[] resize() {
    Node<K,V>[] oldTab = table;
    int oldCap = (oldTab == null) ? 0 : oldTab.length;
    int oldThr = threshold;
    int newCap, newThr = 0;
    if (oldCap > 0) {
        if (oldCap >= MAXIMUM_CAPACITY) {
            threshold = Integer.MAX_VALUE;
            return oldTab;
        }
        else if ((newCap = oldCap << 1) < MAXIMUM_CAPACITY &&
            oldCap >= DEFAULT_INITIAL_CAPACITY)
            newThr = oldThr << 1; // double threshold
    }
    else if (oldThr > 0) // initial capacity was placed in threshold
        newCap = oldThr;
    else {
        // zero initial threshold signifies
        using defaults
    }
}
```

```

        newCap = DEFAULT_INITIAL_CAPACITY;
        newThr = (int) (DEFAULT_LOAD_FACTOR *
DEFAULT_INITIAL_CAPACITY);
    }
    if (newThr == 0) {
        float ft = (float) newCap * loadFactor;
        newThr = (newCap < MAXIMUM_CAPACITY && ft <
(float) MAXIMUM_CAPACITY ?
            (int) ft : Integer.MAX_VALUE);
    }
    threshold = newThr;
    @SuppressWarnings({"rawtypes", "unchecked"})
        Node<K,V>[] newTab = (Node<K,V>[]) new Node[newCap];
    table = newTab;
    if (oldTab != null) {
        for (int j = 0; j < oldCap; ++j) {
            Node<K,V> e;
            if ((e = oldTab[j]) != null) {
                oldTab[j] = null;
                if (e.next == null)
                    newTab[e.hash & (newCap - 1)] = e;
                else if (e instanceof TreeNode)
                    ((TreeNode<K,V>) e).split(this, newTab, j,
oldCap);

                else { // preserve order
                    Node<K,V> loHead = null, loTail = null;
                    Node<K,V> hiHead = null, hiTail = null;
                    Node<K,V> next;
                    do {
                        next = e.next;
                        if ((e.hash & oldCap) == 0) {
                            if (loTail == null)
                                loHead = e;
                            else
                                loTail.next = e;
                            loTail = e;
                        }
                        else {
                            if (hiTail == null)
                                hiHead = e;
                            else
                                hiTail.next = e;
                            hiTail = e;
                        }
                    } while ((e = next) != null);
                    if (loHead != null)
                        newTab[loHead.hash & (newCap - 1)] = loHead;
                    if (hiHead != null)
                        newTab[hiHead.hash & (newCap - 1)] = hiHead;
                }
            }
        }
    }

```

```

        }
    } while ((e = next) != null);
    if (loTail != null) {
        loTail.next = null;
        newTab[j] = loHead;
    }
    if (hiTail != null) {
        hiTail.next = null;
        newTab[j + oldCap] = hiHead;
    }
}
}
}
}
return newTab;
}

```

感觉代码也是比较多的啊，同样，我们关注重点代码：

```
newCap = DEFAULT_INITIAL_CAPACITY;
```

有这么一个赋值操作，DEFAULT_INITIAL_CAPACITY字面意思理解就是初始化容量啊，是多少呢？

```
static final int DEFAULT_INITIAL_CAPACITY = 1 << 4; // aka 16
```

这里是个移位运算，就是16，现在已经确定具体的默认容量是16了，那具体在哪创建默认的Node数组呢？继续往下看源码，有这么一句

```
Node<K,V>[] newTab = (Node<K,V>[])new Node[newCap];
```

ok，到这里我们发现，第一次使用HashMap添加数据的时候底层会创建一个长度为16的默认Node数组。

那么新的问题来了？

为啥初始化大小是16

这个问题想必你在HashMap相关分析文章中也看到过，那么该怎么回答呢？

想搞明白为啥是16不是其他的，那首先要知道为啥HashMap的容量要是2的整数次幂？

为什么容量要是 2 的整数次幂？

先看这个16是怎么来的：

```
static final int DEFAULT_INITIAL_CAPACITY = 1 << 4; // aka 16
```

这里使用了位运算，为啥不直接16嘞？这里主要是位运算的性能好，为啥位运算性能就好，那是因为位运算人家直接操作内存，不需要进行进制转换，要知道计算机可是以二进制的形式做数据存储啊，知道了吧，那16嘞？为啥是16不是其他的？想要知道为啥是16，我们得从HashMap的数据存放特性来说。

对于HashMap而言，存放的是键值对，所以做数据添加操作的时候会根据你传入的key值做hash运算，从而得到一个下标值，也就是以这个下标值来确定你的这个value值应该存放在底层Node数组的哪个位置。

那么这里一定会出现的问题就是，不同的key会被计算得出同一个位置，那么这样就冲突啦，位置已经被占了，那么怎么办嘞？

首先就是冲突了，我们要想办法看看后来的数据应该放在哪里，就是给它找个新位置，这是常规方法，除此之外，我们是不是也可以聚焦到hash算法这块，就是尽量减少冲突，让得到的下标值能够均匀分布。

好了，以上巴拉巴拉说一些理念，下面我们看看源码中是怎么计算下标值得：

```
i = (n - 1) & hash
```

这是在源码中第629行有这么一段，它就是计算我们上面说的下标值的，这里的n就是数组长度，默认的就是16，这个hash就是这里得到的值：

```
static final int hash(Object key) {  
    int h;  
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);  
}
```

继续看它：

```
i = (n - 1) & hash
```

这里是做位与运算，接着我们还需要先搞明白一个问题

为什么要进行取模运算以及位运算

要知道，我们最终是根据key通过哈希算法得到下标值，这个是怎么得到的呢？通常做法就是拿到key的hashcode然后与数组的容量做取模运算，为啥要做取模运算呢？

比如这里默认是一个长度为16的Node数组，我们现在要根据传进来的key计算一个下标值出来然后把value放入到正确的位置，想一下，我们用key的hashcode与数组长度做取模运算，得到的下标值是不是一定在数组的长度范围之内，也就是得到的下标值不会出现越界的情况。

要知道取模是怎么回事啊！明白了这点，我们再来看：

```
i = (n - 1) & hash
```

这里就是计算下标的，为啥不是取模运算而是位与运算呢？使用位与运算的一方面原因就是它的性能比较好，另外一点就是这里有这么一个等式：

```
(n - 1) & hash = n % hash
```

因此，总结起来就是使用位与运算可以实现和取模运算相同的效果，而且位与运算性能更高！

接着，我们再看一个问题

为什么要减一做位运算

理解了这个问题，我们就快接近为什么容量是2的整数次幂的答案了，根据上面说的，这里的n-1是为了实现与取模运算相同的效果，除此之外还有很重要的原因在里面。

在此之前，我们需要看看什么是位与运算，因为我怕这块知识大家之前不注意忘掉了，而它对理解我们现在所讲的问题很重要，看例子：

比如拿5和3做位与运算，也就是 $5 \& 3 = 1$ （操作的是二进制），怎么来的呢？

5转换为二进制：0000 0000 0000 0000 0000 0000 0000 0101

3转换为二进制：0000 0000 0000 0000 0000 0000 0000 0011

1转换为二进制：0000 0000 0000 0000 0000 0000 0000 0001

所以啊，位与运算的操作就是：第一个操作数的第n位与第二个操作数的第n位如果都是1，那么结果的第n位也为1，否则为0

看懂了吧，不懂得话可以去补补这块的知识，后续我也会单独发文详细说说这块。

我们继续回到之前的问题，为什么做减一操作以及容量为啥是2的整数次幂，为啥嘞？

告诉你个秘密，2的整数次幂减一得到的数非常特殊，有啥特殊嘞，就是2的整数次幂得到的结果的二进制，如果某位上是1的话，那么2的整数次幂减一的结果的二进制，之前为1的后面全是1

啥意思嘞，可能有点绕，我们先看2的整数次幂啊，有2，4，8，16，32等等，我们来看，首先是16的二进制是：**10000**，接着16减一得15，15的二进制是：**1111**，再形象一点就是：

16转换为二进制：0000 0000 0000 0000 0000 0000 0001 0000

15转换为二进制：0000 0000 0000 0000 0000 0000 0000 1111

再对照我给你说的秘密，看看懂了吧，可以再举个栗子：

32转换为二进制：0000 0000 0000 0000 0000 0000 0010 0000

31转换为二进制：0000 0000 0000 0000 0000 0000 0001 1111

这会总该懂了吧，然后我们再看计算下标的公式：

```
(n - 1) & hash = n % hash
```

n 是容量，它是2的整数次幂，然后与得到的hash值做位与运算，因为 n 是2的整数次幂，减一之后的二进制最后几位都是1，再根据位与运算的特性，与hash位与之后，得到的结果是不是可能是0也可能是1，，也就是说最终的结果取决于hash的值，如此一来，只要输入的hashcode值本身是均匀分布的，那么hash算法得到的结果就是均匀的。

啥意思？这样得到的下标值就是均匀分布的啊，那冲突的几率就减少啦。

而如果容量不是2的整数次幂的话，就没有上述说的那个特性，这样冲突的概率就会增大。

所以，明白了为啥容量是2的整数次幂了吧。

那为啥是16嘞？难道不是2的整数次幂都行嘛？理论上是都行，但是如果是2，4或者8会不会有点小，添加不了多少数据就会扩容，也就是会频繁扩容，这样岂不是影响性能，那为啥不是32或者更大，那不就浪费空间了嘛，所以啊，16就作为一个非常合适的经验值保留了下来！

出现哈希冲突怎么解决

我们上面也提到了，在添加数据的时候尽管为实现下标值的均匀分布做了很多努力，但是势必还是会存在冲突的情况，那么该怎么解决冲突呢？

这就牵涉到哈希冲突的解决办法了，详情建议阅读：[来吧！一文彻底搞定哈希表！](#)

了解了哈希冲突的解决办法之后我们还要关注一个问题，那就是新的节点在插入到链表的时候，是怎么插入的？

回答开篇的问题

现在你应该知道，当出现hash冲突，可以使用链表来解决，那么这里就有问题，新来的Node是应该放在之前Node的前面还是后面呢？

Java8之前是头插法，啥意思嘞，就是放在之前Node的前面，为啥要这样，这是之前开发者觉得后面插入的数据会先用到，因为要使用这些Node是要遍历这个链表，在前面的遍历的会更快。

为什么使用尾插法？

但是在Java8及之后都使用尾插法了，就是放到后面，为啥这样？

这里主要是一个链表成环的问题，啥意思嘞，想一下，使用头插法是不是会改变链表的顺序，你后来的就应该在后面嘛，如果扩容的话，由于原本链表顺序有所改变，扩容之后重新hash，可能导致的情况就是扩容转移后前后链表顺序倒置，在转移过程中修改了原来链表中节点的引用关系。

这样的话在多线程操作下就会出现死循环，而使用尾插法，在相同的前提下就不会出现这样的问题，因为扩容前后链表顺序是不变的，他们之间的引用关系也是不变的。

关于扩容

下面我们继续说HashMap的扩容，经过上面的分析，我们知道第一次使用HashMap是创建一个默认长度为16的底层Node数组，如果满了怎么办，那就需要进行扩容了，也就是之前谈及的resize方法，这个方法主要就是初始化和增加表的大小，关于扩容要知道这两个概念：

1. Capacity: HashMap当前长度。
2. LoadFactor: 负载因子，默认值0.75f。

这里怎么扩容的呢？首先是达到一个条件之后会发生扩容，什么条件呢？就是这个负载因子，比如HashMap的容量是100，负载因子是0.75，乘以100就是75，所以当你增加第76个的时候就需要扩容了，那扩容又是怎么样步骤呢？

首先是创建一个新的数组，容量是原来的二倍，为啥是2倍，想一想为啥容量是2的整数次幂，这里扩容为原来的2倍不正好符合这个规则嘛。

然后会经过重新hash，把原来的数据放到新的数组上，至于为啥要重新hash，那必须啊，你容量变了，相应的hash算法规则也就变了，得到的结果自然不一样了。

关于链表转红黑树

在Java8之前是没有红黑树的实现的，在jdk1.8中加入了红黑树，就是当链表长度为8时会将链表转换为红黑树，为6时又会转换成链表，这样时提高了性能，也可以防止哈希碰撞攻击，这些知识在[来吧！一文彻底搞定哈希表！](#)都有详细讲解，强烈推荐阅读

HashMap增加新元素的主要步骤

下面我们分析一下HashMap增加新元素的时候都会做哪些步骤：

1. 首先肯定时根据key值，通过哈希算法得到value应该放在底层数组中的下标位置
2. 根据这个下标定位到底层数组中的元素，当然，这里可能时链表，也可能时树，知道为啥吧，给你个提醒，链表转红黑树
3. 拿到当前位置上的key值，与要放入的key比较，是否==或者equals，如果成立的话就替换value值，并且需要返回原来的值
4. 当然，如果是树的话就要循环树中的节点，继续==和equals的判断，成立替换，否则添加到树里
5. 链表的话就是循环遍历了，同样的判断，成立替换，否则就添加到链表的尾部

所以啊，这里面的重点就是判断放入HashMap中的元素要不要替换当前节点的元素，那怎么判断呢？总结起来只要满足以下两点即可替换：

- 1、hash值相等。
- 2、==或equals的结果为true。