链滴

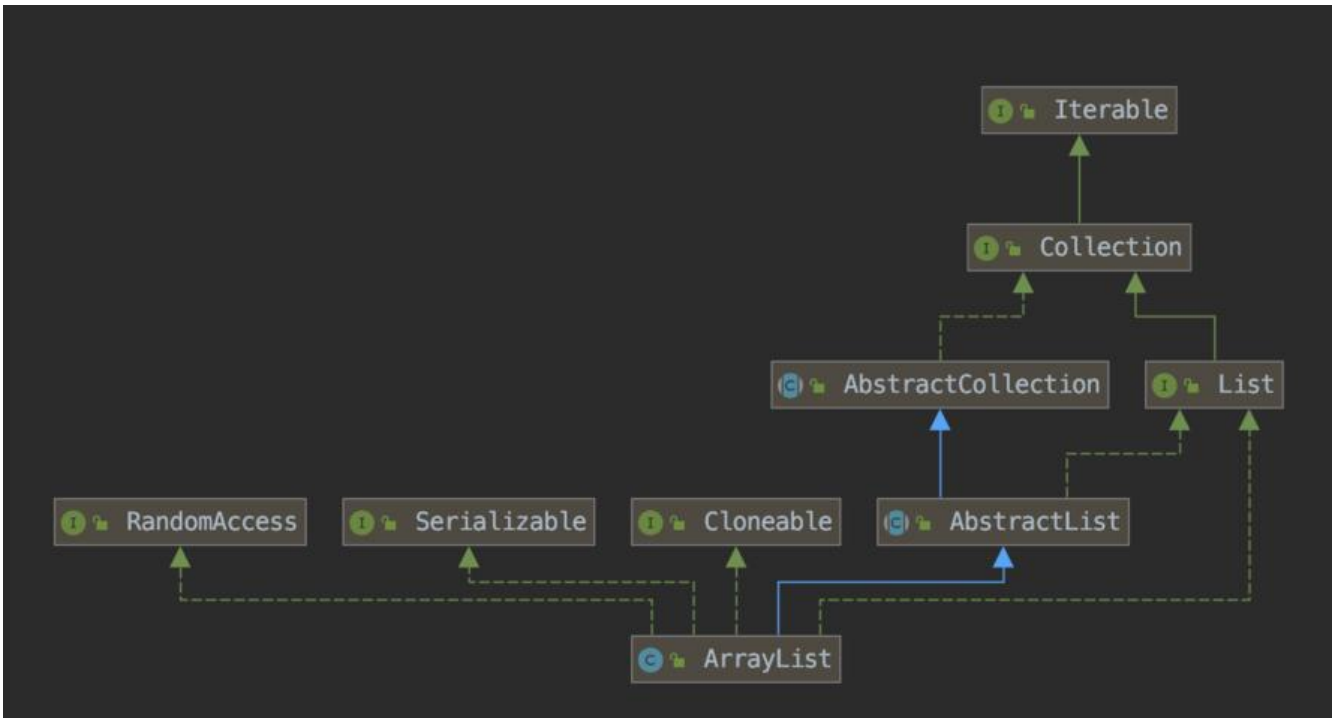
Java 源码之旅——ArrayList

作者：lonelyant

原文链接：https://ld246.com/article/1574328934737

来源网站：链滴

先看一眼关系图，有个大概印象。



ArrayList整体架构比较简单，底层实现就是一个数组。

建议打开ArrayList源码，对照着看。本文仅解析部分核心代码，并未涉及到ArrayList的所有代码。

[TOC]

关键类变量

```
/**
* Default initial capacity.
* 默认初始容量为10，这个值得记住
*/
private static final int DEFAULT_CAPACITY = 10;

/**
* ArrayList中实际负责存储的数组对象
* transient修饰代表该字段不参与序列化
*/
transient Object[] elementData;

/**
* The size of the ArrayList (the number of elements it contains).
* 表示当前数组的大小，类型int,没有使用volatile 修饰，非线程安全的
*/
private int size;

/**
* 统计当前数组被修改的版本次数，数组结构有变动，就会+1
* 定义在AbstractList中
*/
```

```
protected transient int modCount = 0;
```

构造方法

ArrayList有三个构造方法，分别是无参数直接初始化、指定大小初始化、指定初始数据初始化。

无参数直接初始化

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

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

这个没什么好说的，赋值了一个空数组。

指定大小初始化

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

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);
  }
}
```

指定初始数据初始化

```
public ArrayList(Collection<? extends E> c) {
  elementData = c.toArray();
  if ((size = elementData.length) != 0) {
    // c.toArray might (incorrectly) not return Object[] (see 6260652)
    if (elementData.getClass() != Object[].class)
      elementData = Arrays.copyOf(elementData, size, Object[].class);
  } else {
    // replace with empty array.
    this.elementData = EMPTY_ELEMENTDATA;
  }
}
```

当给定的集合长度为0的时候，初始化为空数组。

当集合长度不为0时，如果集合元素类型不是 Object 类型，会转成 Object。这里会有一点问题，ArrayList 初始化之后（ArrayList 元素非 Object 类型），再次调用 toArray 方法，得到 Object 数组，并

往 Object 数组赋值时，会触发此 bug，导致ArrayStoreException。这一点已经在Java 9中被修复。

新增和扩容

从上面我们能看到，ArrayList真正存储数据的地方其实是elementData，而elementData是一个数，数组的长度是固定的。当数组装满之后还要再往ArrayList中添加元素的时候，就需要进行扩容操作。

所以往数组中新增元素，主要分为两步：

- 判断是否需要扩容，如果需要，执行扩容操作；
- 赋值

代码

```
public boolean add(E e) {
  // 确保容量足够，如果不够，则扩容
  ensureCapacityInternal(size + 1);  // Increments modCount!!
  // 将新元素追加到数组尾部。注意，这里是线程不安全的。
  elementData[size++] = e;
  return true;
}

// 保证容量足够，容量不足会扩容
private void ensureCapacityInternal(int minCapacity) {
  ensureExplicitCapacity(calculateCapacity(elementData, minCapacity));
}

private static int calculateCapacity(Object[] elementData, int minCapacity) {
  if (elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA) {
    return Math.max(DEFAULT_CAPACITY, minCapacity);
  }
  return minCapacity;
}

private void ensureExplicitCapacity(int minCapacity) {
  modCount++;    // 每进行一次操作，版本号就会加1

  // overflow-conscious code
  // 如果我们期望的最小容量大于目前数组的长度，那么就扩容
  if (minCapacity - elementData.length > 0)
    // 扩容
    grow(minCapacity);
}
```

添加这一块，ArrayList实际上是花了绝大部分精力在扩容上面，在进行真正的扩容操作（grow）之，已经计算出了需要扩充到的容量。

```
// 扩容，并把现有数据拷贝到新的数组里面去
private void grow(int minCapacity) {
  // overflow-conscious code
  int oldCapacity = elementData.length;
  // oldCapacity >> 1 是把 oldCapacity 除以 2 的意思
  int newCapacity = oldCapacity + (oldCapacity >> 1);
```

```
  // 如果扩容后的值 < 我们的期望值，扩容后的值就等于我们的期望值
  if (newCapacity - minCapacity < 0)
    newCapacity = minCapacity;
  // 如果扩容后的值 > jvm 所能分配的数组的最大值，那么就用 Integer 的最大值
  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);
}

private static int hugeCapacity(int minCapacity) {
  if (minCapacity < 0) // overflow
    throw new OutOfMemoryError();
  return (minCapacity > MAX_ARRAY_SIZE) ?
    Integer.MAX_VALUE :
  MAX_ARRAY_SIZE;
}
```

MAX_ARRAY_SIZE

```
/**
* 要分配的数组的最大大小。
* 一些虚拟机在数组中保留一些标题字。
* 尝试分配更大的数组可能会导致
* OutOfMemoryError: Requested array size exceeds VM limit
*/
private static final int MAX_ARRAY_SIZE = Integer.MAX_VALUE - 8;
```

扩容本质 Arrays.copyOf

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

newCapacity是我们期望的新数组容量，该行代码会新建一个容量为newCapacity的数组，然后将原组的数据填充到新数组，最后返回这个新数组。该方法底层通过 System.arraycopy 方法进行拷贝，方法是 native 的方法，源码如下：

```
/**
 * @param src     被拷贝的数组
 * @param srcPos  从数组那里开始
 * @param dest    目标数组
 * @param destPos 从目标数组那个索引位置开始拷贝
 * @param length  拷贝的长度
 * 此方法是没有返回值的，通过 dest 的引用进行传值
 */
public static native void arraycopy(Object src,  int  srcPos,
                                     Object dest, int destPos,
                                     int length);
```

添加元素到指定索引位置

```
public void add(int index, E element) {
  // 检查索引合法性
```

```
  rangeCheckForAdd(index);

  ensureCapacityInternal(size + 1);  // Increments modCount!!
  System.arraycopy(elementData, index, elementData, index + 1,
                   size - index);
  elementData[index] = element;
  size++;
}

private void rangeCheckForAdd(int index) {
  if (index > size || index < 0)
    throw new IndexOutOfBoundsException(outOfBoundsMsg(index));
}
```

流程就是进行扩容检查，该扩容就扩容，之后使用刚刚讲到的native方法System.arraycopy将原数组处于index位置与之后的所有元素向后移动一格，最后把需要新增的元素放到index就好了，非常的简。

小结

新增的话还有addAll(Collection<? extends E> c)等方法，大同小异，有兴趣的朋友可以自行查看。

扩容里面需要注意的几点：

- 扩容的规则并不是翻倍，是原来容量大小 + 容量大小的一半，直白来说，**扩容后的大小是原来容的 1.5 倍**；
- ArrayList 中的 **数组的最大值是 Integer.MAX_VALUE**，超过这个值，JVM 就不会给数组分配内空间了。
- 新增时，并没有对值进行严格的校验，所以 **ArrayList 是允许 null 值**的。
- **溢出意识**，源码中已经在相应位置标明了overflow-conscious code。就是说扩容后数组的大小下不能小于 0，上界不能大于 Integer 的最大值，这种意识我们可以学习。

删除

ArrayList 删除元素有很多种方式，比如根据数组索引删除、根据值删除或批量删除等等，原理和思都差不多，这里选取根据值删除方式来进行源码解读。

删除这一块我从源码里面没看出来哪里缩容了，1.8以后ArrayList删除操作不缩容了？希望有大佬可赐教。

代码

```
public boolean remove(Object o) {
  // 如果要删除的值是 null，找到第一个值是 null 的删除
  if (o == null) {
    for (int index = 0; index < size; index++)
      if (elementData[index] == null) {
        fastRemove(index);
        return true;
      }
  } else {
```

```
    // 如果要删除的值不为 null，找到第一个和要删除的值相等的删除
    for (int index = 0; index < size; index++)
      // 这里是根据 equals 来判断值相等的，相等后再根据索引位置进行删除
      if (o.equals(elementData[index])) {
        fastRemove(index);
        return true;
      }
  }
  return false;
}

/*
* Private remove method that skips bounds checking and does not
* return the value removed.
*/
private void fastRemove(int index) {
  modCount++;
  // numMoved 表示删除 index 位置的元素后，需要从 index 后移动多少个元素到前面去
  // 减 1 的原因，是因为 size 从 1 开始算起，index 从 0开始算起
  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
}
```

移除操作通过fastRemove实现，该方法借助native方法，将index+1位置及其后面的元素，依次赋给前一格。然后将位于--size的元素置空，方便GC。

小结

- 新增的时候是没有对 null 进行校验的，所以删除的时候也是 **允许删除 null 值**
- 判断元素是否相等，是 **通过 equals 来判断**的，如果数组元素不是基本类型，需要我们关注 equal 的具体实现。

迭代器

```
public Iterator<E> iterator() {
  return new Itr();
}
```

这里的Itr是ArrayList的内部类，该内部类实现了java.util.Iterator接口。

Itr类有如下几个成员变量

```
int cursor;       // index of next element to return
int lastRet = -1; // index of last element returned; -1 if no such
// expectedModCount 表示迭代过程中，期望的版本号；modCount 表示数组实际的版本号。
int expectedModCount = modCount;
```

迭代器一般来说有四个方法：

- hasNext 用来判断是否还有值可以迭代

- next 如果有值可以迭代，迭代的值是多少
- remove 删除当前迭代的值
- forEachRemaining 1.8新增，能够将Iterator中迭代剩余的元素传递给一个函数

hasNext

```
public boolean hasNext() {
  return cursor != size;//cursor 表示下一个元素的位置，size 表示实际大小，如果两者相等，说明
经没有元素可以迭代了，如果不等，说明还可以迭代
}
```

next

```
public E next() {
  //迭代过程中，判断版本号有无被修改，有被修改，抛 ConcurrentModificationException 异常
  checkForComodification();
  //本次迭代过程中，元素的索引位置
  int i = cursor;
  if (i >= size)
    throw new NoSuchElementException();
  Object[] elementData = ArrayList.this.elementData;
  if (i >= elementData.length)
    throw new ConcurrentModificationException();
  // 下一次迭代时，元素的位置，为下一次迭代做准备
  cursor = i + 1;
  // 返回元素值
  return (E) elementData[lastRet = i];
}
// 版本号比较
final void checkForComodification() {
  if (modCount != expectedModCount)
    throw new ConcurrentModificationException();
}
```

next 方法就干了两件事情，第一是检验能不能继续迭代，第二是找到迭代的值，并为下一次迭代做准（cursor+1）。

remove

```
public void remove() {
  // 如果上一次操作时，数组的位置已经小于 0 了，说明数组已经被删除完了
  if (lastRet < 0)
    throw new IllegalStateException();
  //迭代过程中，判断版本号有无被修改，有被修改，抛 ConcurrentModificationException 异常
  checkForComodification();

  try {
    ArrayList.this.remove(lastRet);
    cursor = lastRet;
    // -1 表示元素已经被删除，这里也防止重复删除
    lastRet = -1;
```

```
    // 删除元素时 modCount 的值已经发生变化，在此赋值给 expectedModCount
    // 这样下次迭代时，两者的值是一致的了
    expectedModCount = modCount;
  } catch (IndexOutOfBoundsException ex) {
    throw new ConcurrentModificationException();
  }
}
```

forEachRemaining

```
public void forEachRemaining(Consumer<? super E> consumer) {
  Objects.requireNonNull(consumer);
  final int size = ArrayList.this.size;
  int i = cursor;
  if (i >= size) {
    return;
  }
  final Object[] elementData = ArrayList.this.elementData;
  if (i >= elementData.length) {
    throw new ConcurrentModificationException();
  }
  while (i != size && modCount == expectedModCount) {
    consumer.accept((E) elementData[i++]);
  }
  // update once at end of iteration to reduce heap write traffic
  // 在迭代结束时更新一次，以减少堆写入流量
  cursor = i;
  lastRet = i - 1;
  checkForComodification();
}
```

小结

- lastRet = -1 的操作目的，是防止重复删除操作
- 删除元素成功，数组当前 modCount 就会发生变化，这里会把 expectedModCount 重新赋值，次迭代时两者的值就会一致了

时间复杂度

新增（不考虑扩容）

对于新增操作，如果是add(E e)方法，那么新元素会被直接添加到数组末尾，时间复杂度为O(1)；但是dd(int, E)会需要对index后的元素作遍历移动操作，所以时间复杂度为O(n)。

删除（没找到缩容相关代码）

同样的，从数组尾部删除为O(1)，从任意索引位置删除为O(n)。

查找

通过索引get，时间复杂度为O(1)；查找元素是否存在，时间复杂度为O(n)。

线程安全问题

只有当 ArrayList 作为共享变量时，才会有线程安全问题，当 ArrayList 是方法内的局部变量时，没线程安全问题。

ArrayList 有线程安全问题的本质，是因为 ArrayList 自身的 elementData、size、modConut 在进各种操作时，都没有加锁，而且这些变量的类型并非是可见（volatile）的，所以如果多个线程对这变量进行操作时，可能会有值被覆盖的情况。