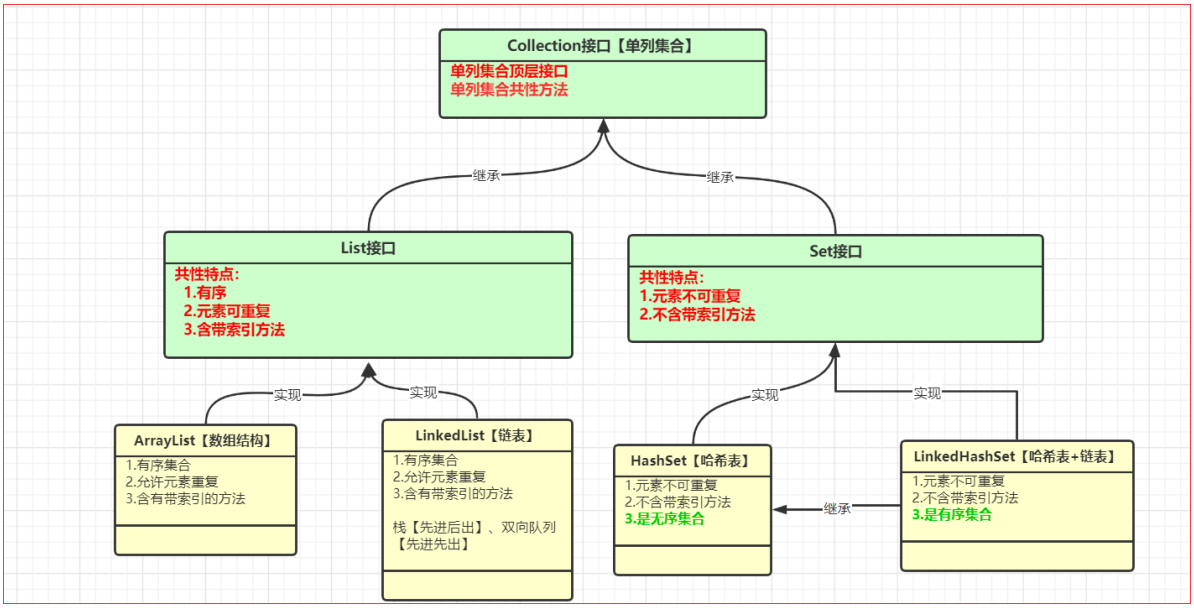
第一章 ArrayList源码分析

目标:

- 理解ArrayList的底层数据结构
- 深入掌握ArrayList查询快，增删慢的原因
- 掌握ArrayList的扩容机制
- 掌握ArrayList初始化容量过程
- 掌握ArrayList出现线程安全问题原因及解决方案
- 掌握ArrayList的Fail-Fast机制

一、ArrayList的简介



ArrayList集合是Collection和List接口的实现类。底层的数据结构是数组。数据结构特点 : 增删慢，查询快。线程不安全的集合!

许多程序员开发的时候，使用集合基本上无脑选取ArrayList！不建议这种用法。

ArrayList的特点:

- 单列集合 : 对应与Map集合来说【双列集合】
- 有序性 : 存入的元素和取出的元素是顺序是一样的
- 元素可以重复 : 可以存入两个相同的元素
- 含带索引的方法 : 数组与生俱来含有索引【下角标】

二、ArrayList原理分析

2.1 ArrayList的数据结构源码分析

```java
//空的对象数组
private static final Object[] EMPTY_ELEMENTDATA = {};
//默认容量空对象数组，通过空的构造参数生成ArrayList对象实例
private static final Object[] DEFAULTCAPACITY_EMPTY_ELEMENTDATA = {};
//ArrayList对象的实际对象数组!
transient Object[] elementData; // non-private to simplify nested class access
//1、为什么是Object类型呢？利用面向对象的多态特性，当前ArrayList的可以存储任意引用数据类型。
//2、ArrayList有一个问题，不能存储基本数据类型！就是数组的类型是Object类型
```

2.2 ArrayList默认容量&最大容量

```java
//默认的初始化容量是10
private static final int DEFAULT_CAPACITY = 10;
//最大容量 : 2^31 - 1 - 8 = 21 4748 3639【21亿】
private static final int MAX_ARRAY_SIZE = Integer.MAX_VALUE - 8;
```

> 为什么最大容量要-8呢?
>
> 目的是为了存储ArrayList集合的基本信息，比如list集合的最大容量!

Read the above article about Java Memory management, which clearly states

10

I think this applies to ArrayList as it is the Resizable array implemenation.

Anatomy of a Java array object

> The shape and structure of an array object, such as an array of int values, is similar to that of a standard Java object. The primary difference is that the array object has an additional piece of metadata that denotes the array's size. An array object's metadata, then, consists of: Class : A pointer to the class information, which describes the object type. In the case of an array of int fields, this is a pointer to the int[] class.
>
> Flags : A collection of flags that describe the state of the object, including the hash code for the object if it has one, and the shape of the object (that is, whether or not the object is an array).
>
> Lock : The synchronization information for the object — that is, whether the object is currently synchronized.
>
> Size : The size of the array.

max size

```
2^31 = 2,147,483,648
```

as the Array it self needs `8 bytes` to stores the size `2,147,483,648`

so

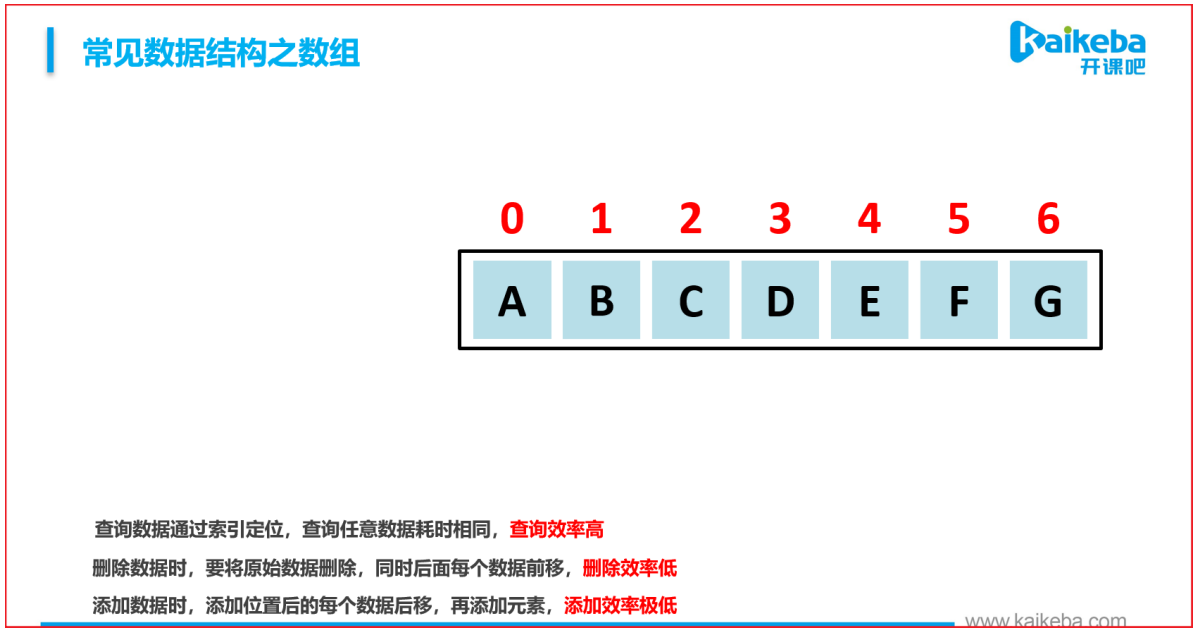
```
2^31 -8 (for storing size ),
```

so maximum array size is defined as Integer.MAX_VALUE - 8

2.3 为什么ArrayList查询快，增删慢?

ArrayList的底层数据结构就是一个Object数组，一个可变的数组，对于其的所有操作都是通过数组来实现的。

- 数组是一种，查询快、增删慢!
- 查询数据是通过索引定位，查询任意数据耗时均相同。**查询效率贼高!**
- 删除数据时，要将原始数据删除，同时后面的每个数据迁移。**删除效率就比较低!**
- 新增数据，在添加数组的位置加入数组，同时在数组后面位置后移以为！**添加效率极低!**



2.4 ArrayList初始化容量

ArrayList底层是数组，动态数组!

- 底层是Object对象数组，数组存储的数据类型是Object，数组名字为elementData。

```
transient Object[] elementData;
```

1、创建ArrayList对象分析：无参数

> 创建ArrayList的之后，ArrayList容量是多少呢？回答10是错误的！回答0是正确【限定条件，在JDK1.8中】
>
> 如何 初始化 动态数组的容量？10个

构造方法

```
/**
     * Constructs an empty list with an initial capacity of ten.
     */
//初始化的ArrayList的容量，是10个!
public ArrayList() {
    this.elementData = DEFAULTCAPACITY_EMPTY_ELEMENTDATA;
}
//空数组!
private static final Object[] DEFAULTCAPACITY_EMPTY_ELEMENTDATA = {};
```

在执行add()方法的时候初始化！【懒加载】

判断当前数组的容量是否有存储空间，如果没有初始化一个10的容量。

```
//想数组中，添加一个元素
public boolean add(E e) {
    //确保有容量，如果第一次添加，会初始化一个容量为10的list
    //size当前集合元素的个数，随着添加的元素递增
    ensureCapacityInternal(size + 1);  // Increments modCount!!
    //添加元素
    elementData[size++] = e;
    return true;
}
//ensureCapacityInternal确保有容量，如果第一次添加，会初始化一个容量为10的list
private void ensureCapacityInternal(int minCapacity) {
    //两个方法
    ensureExplicitCapacity(calculateCapacity(elementData, minCapacity));
}
// calculateCapacity(elementData, minCapacity) 拿着当前ArrayList的数组，与当前数组中的元素个数。计算容量
private static int calculateCapacity(Object[] elementData, int minCapacity)
{
    //ArrayList的数组 与默认的数组进行比较。、
    //{}  == {}
    if (elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA) {/true
        //DEFAULT_CAPACITY = 10
        //minCapacity 1
        //1和10比谁大 10
        return Math.max(DEFAULT_CAPACITY, minCapacity);//计算之后，返回的初始化容量是10
    }
    return minCapacity;
}
// ensureExplicitCapacity() 确保不会超过数组的真实容量
private void ensureExplicitCapacity(int minCapacity) {
    //minCapacity 当前计算后容量 10
    modCount++;//对当前数组操作计数器

    // overflow-conscious code
    //最小的容量 : 10 - 当前数组的容量{} 0
    if (minCapacity - elementData.length > 0)
        grow(minCapacity);//做了扩容
}
```

2、创建ArrayList对象分析：带有初始化容量构造方法

```
//创建ArrayList集合，并且设置固定的集合容量
public ArrayList(int initialCapacity) {
    //initialCapacity 手动设置的初始化容量
    if (initialCapacity > 0) {//判断容量是否大于0，如果大于0
        //创建一个对象数组位指定容量大小，并且交给ArrayList对象
        this.elementData = new Object[initialCapacity];
        //如果设置的容量为0，设置默认数组
    } else if (initialCapacity == 0) {
        this.elementData = EMPTY_ELEMENTDATA;//默认的元素数据数组{}
    } else {
        //如果不是0，也不是大于0的数，会抛出非法参数异常！
```

```
        throw new IllegalArgumentException("Illegal Capacity:
"+initialCapacity);
    }
}
```

注意 : 使用ArrayList的集合，建议如果知道集合的大小，最好提前设置。提示集合的使用效率!

2.5 ArrayList扩容原理

add方法先要确保数组的容量足够，防止数组已经填满还往里面添加数据造成数组越界:

1. 如果数组空间足够，直接将数据添加到数组中
2. 如果数组空间不够了，则进行扩容。**扩容1.5倍扩容。**
3. 扩容 : 原始数组copy新数组中，同时向新数组后面加入数据

注意 : new的ArrayList的对象没有容量的，在第一次添加的add，会进行第一次扩容。0 -> 10!

```
//grow扩容数组
private void grow(int minCapacity) {
    //minCapacity 当前数组的最小容量，存储了多少个元素
    // overflow-conscious code
    //获取当前存储数据数组的长度
    int oldCapacity = elementData.length;
    //新的容量 = 旧的容量 + 扩容的容量【旧容量/2 = 0.5旧容量】
    //扩容1.5倍扩容
    int newCapacity = oldCapacity + (oldCapacity >> 1);
    //极端情况过滤 :  新的容量 - 旧的容量小于0【int值移除】
    if (newCapacity - minCapacity < 0)
        newCapacity = minCapacity;//不扩容了
    //新的容量，比ArrayList的最大值，还要打
    if (newCapacity - MAX_ARRAY_SIZE > 0)
        //设置新的容量为ArrayList的最大值，以ArrayList最大值为当前容量
        newCapacity = hugeCapacity(minCapacity);
    // minCapacity is usually close to size, so this is a win:
    elementData = Arrays.copyOf(elementData, newCapacity);
}
```

总结:

1. 扩容的规则并不是翻倍，是原来容量的1.5倍
2. ArrayList的数组最大值Integer.MAX_VALUE。不允许超过这个最大值
3. 新增元素时，没有严格的数据值的检查。所有可用设置null

三、ArrayList线程安全问题及解决方案

3.1 错误复现

ArrayList 我们都知道底层是以数组方式实现的，实现了可变大小的数组，它允许所有元素，包括null。看下面一个例子：开启多个线程操作List集合，向ArrayList中增加元素，同时去除元素

```
/**
 * 目标 : 线程安全问题复现
 */
public class Demo04 {
    //全局线程共享集合ArrayList
    protected static ArrayList<Object> arrayList = new ArrayList<>();
```

```
7
8
9       public static void main(String[] args) {
10          //1.创建线程数组【500】
11          Thread[] threads = new Thread[500];
12          //2.遍历数组，想线程中添加500线程对象
13          for (int i = 0; i < threads.length; i++) {
14              threads[i] = new MyThread();
15              threads[i].start();//启动线程
16          }
17          //3.遍历线程，等待线程执行完毕【等待所有线程执行完毕】
18          for (int i = 0; i < threads.length; i++) {
19              try {
20                  threads[i].join();//等待线程执行完毕
21              } catch (InterruptedException e) {
22                  e.printStackTrace();
23              }
24          }
25          //线程执行内容 : 向集合中添加自己的线程名称
26          //4.遍历list集合，获取所有线程的名称
27          for (Object threadName : arrayList) {
28              System.out.println("threadName = " + threadName);
29          }
30      }
31  }
32
33  //线程执行内容，是想集合中添加自己的线程名称
34  class MyThread extends Thread {
35      @Override
36      public void run() {
37          try {
38              //线程休眠1000
39              Thread.sleep(1000);
40              //向集合中添加自己的线程名称【操作共享内容，会出现线程安全问题】
41              Demo04.arrayList.add(Thread.currentThread().getName());
42          } catch (InterruptedException e) {
43              e.printStackTrace();
44          }
45
46      }
47  }
```

运行代码结果可知，会出现以下几种情况:

- ①打印null
- ②某些线程并未打印
- ③数组角标越界异常

3.2 导致ArrayList线程不安全的源码分析

ArrayList成员变量

```
public class ArrayList<E> extends AbstractList<E>
        implements List<E>, RandomAccess, Cloneable, java.io.Serializable
{
    //ArrayList的Object的数组存所有元素。
    transient Object[] elementData; // non-private to simplify nested class access
    //size变量保存当前数组中元素个数。
    private int size;
//...
}
```

- ArrayList的Object的数组存所有元素。
- size变量保存当前数组中元素个数。

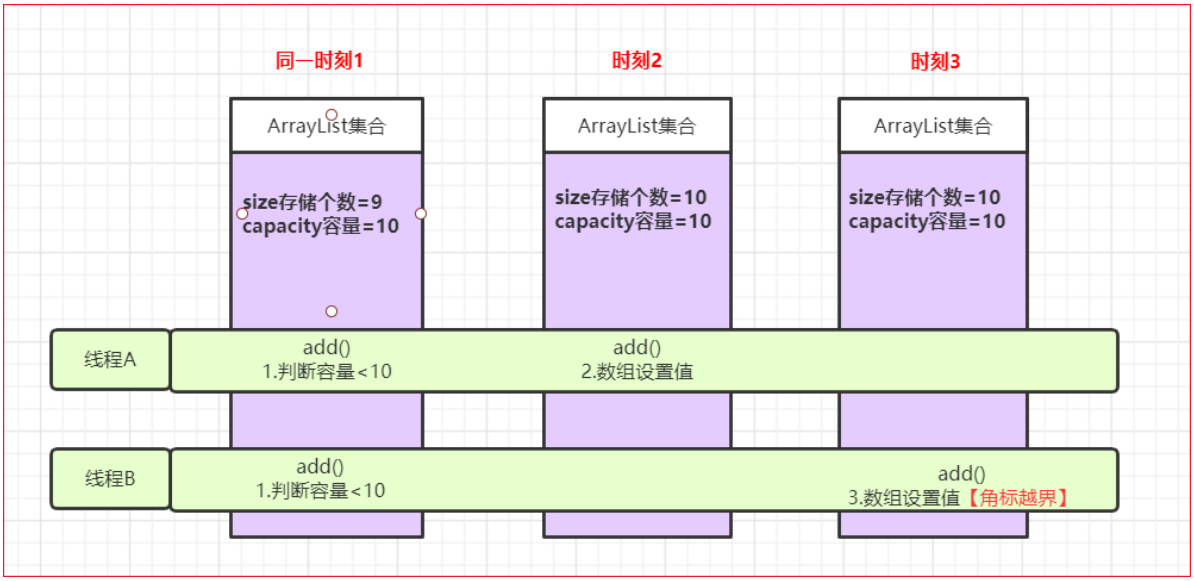
出现线程不安全源码之一：add()方法

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

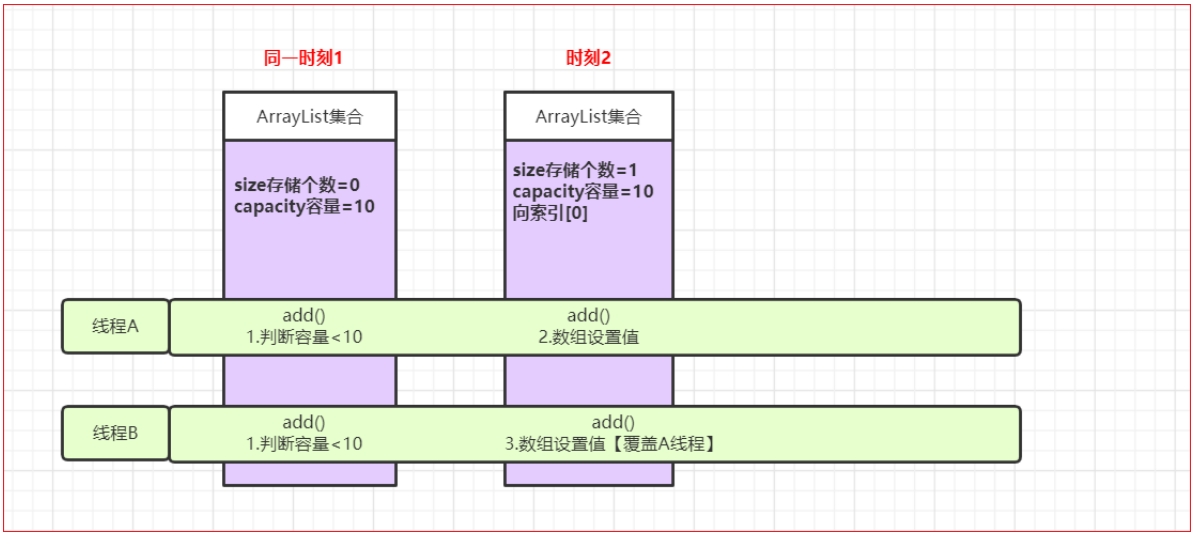
add添加元素，实际做了两个大的步骤：

1. 判断elementData数组容量是否满足需求
2. 在elementData对应位置上设置值

线程不安全的隐患【1】，导致③数组下标越界异常



线程不安全的隐患【2】，导致①Null、②某些线程并未打印



由此我们可以得出，在多线程情况下操作ArrayList 并不是线性安全的。

那如何解决呢?

3.3 解决方案

第一种方案：使用Vector集合，Vector集合是线程安全的

```
1 //线程安全问题解决方案1
2 protected static Vector<Object> vector = new Vector<>();
```

第二种方案：使用Collections.synchronizedList。它会自动将我们的list方法进行改变，最后返回给我们一个加锁了List

```
1 //线程安全问题解决方案2
2 //将集合改为同步集合
3 protected static List<Object> synList =
  Collections.synchronizedList(arrayList);
```

第三种方案：使用JUC中的CopyOnWriteArrayList类进行替换。【】

```
//线程安全问题解决方案3 JUC 【最佳选择】
protected static CopyOnWriteArrayList<Object> copyOnWriteArrayList = new CopyOnWriteArrayList<>();
```

四、ArrayList的Fail-Fast机制深入理解

什么是Fail-Fast机制?

"快速失败"即Fail-Fast机制，它是Java中一种错误检测机制!

当多钱程对集合进行结构上的改变，或者在迭代元素时直接调用自身方法改变集合结构而没有通知迭代器时，有可能会触发Fail-Fast机制并抛出异常【ConcurrentModificationException】。注意，是有可能触发Fail-Fast，而不是肯定!

触发时机：在迭代过程中，集合的结构发生改变，而此时迭代器并不知情，或者还没来得及反应，便会产生Fail-Fast事件。

再次强调，迭代器的快速失败行为无法得到保证！一般来说，不可能对是否出现不同步并发修改，或者自身修改做出任何硬性保证。快速失败迭代器会尽最大努力抛出 ConcurrentModificationException。

Java.util包中的所有集合类都是快速失败的，而java.util.concurrent包中的集合类都是安全失败的；快速失败的迭代器抛出ConcurrentModificationException，而安全失败的迭代器从不抛出这个异常。

上代码：

ArrayList的Fast-Fail事件复现及解决方案

```
/**
 * 目标 : 复现Fast_Fail机制
 * 1.产生条件 :
 *   当多线程操作同一个集合
 *   同时遍历这个集合，该集合被修改!
 * 2.解决方案 :使用并发编程包中的集合，替换愿有集合CopyOnWriteArrayList
 */
public class Demo06 {
    //定义全局共享集合 :
    //static ArrayList<String> list = new ArrayList<>();
    //Fast_Fail机制CopyOnWriteArrayList
    static CopyOnWriteArrayList<String> copyOnWriteArrayList = new CopyOnWriteArrayList<>();

    public static void main(String[] args) {
        //创建线程1，并且向集合添加元素，打印集合中的内容
        Thread thread1 = new Thread(() -> {
            //并且向集合添加元素
            for (int i = 0; i < 6; i++) {
                copyOnWriteArrayList.add("" + i);
                // 打印集合中的内容
                printAll();
            }
        });
        thread1.start();//启动线程1
```

```
25          //创建线程2，并且向集合添加元素，打印集合中的内容
26          Thread thread2 = new Thread(() -> {
27              //并且向集合添加元素
28              for (int i = 10; i < 16; i++) {
29                  copyOnWriteArrayList.add("" + i);
30                  // 打印集合中的内容
31                  printAll();
32              }
33          });
34          thread2.start();//启动线程2
35      }
36      /**
37       * 使用迭代器打印集合
38       */
39      public static void printAll() {
40          //获取当前集合的迭代器
41          Iterator<String> iterator = copyOnWriteArrayList.iterator();
42          //通过迭代器遍历集合
43          while (iterator.hasNext()) {
44              String value = iterator.next();
45              System.out.println(value + ",");
46          }
47
48      }
49  }
```

```
14,
15,
java.util.ConcurrentModificationException
    at java.util.ArrayList$Itr.checkForComodification(ArrayList.java:909)
    at java.util.ArrayList$Itr.next(ArrayList.java:859)
    at com.kkb.Demo06.printAll(Demo06.java:47)
    at com.kkb.Demo06.lambda$main$0(Demo06.java:24)
    at java.lang.Thread.run(Thread.java:748)
```