目录

第1章 基础

01 开篇词：为什么学习本专栏

02 String、Long 源码解析和面试题

03 Java 常用关键字理解

04 Arrays、Collections、Objects 常用方法源码解析

第2章 集合

05 ArrayList 源码解析和设计思路

06 LinkedList 源码解析

07 List 源码会问哪些面试题

08 HashMap 源码解析

09 TreeMap 和 LinkedHashMap 核心源码解析

10 Map源码会问哪些面试题

11 HashSet、TreeSet 源码解析

12 彰显细节：看集合源码对我们实际工作的帮助和应用 最近阅读

13 差异对比：集合在 Java 7 和 8 有何不同和改进

14 简化工作：Guava Lists Maps 实际工作运用和源码

第3章 并发集合类

15 CopyOnWriteArrayList 源码解析和设计思路

16 ConcurrentHashMap 源码解析和设计思路

17 并发 List、Map源码面试题

18 场景集合：并发 List、Map的应用

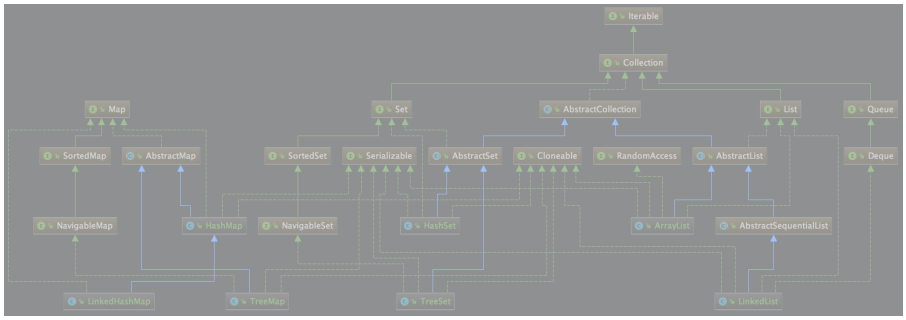
12 彰显细节：看集合源码对我们实际工作的帮助和应用

更新时间：2019-09-17 10:19:41

> 劳动是一切知识的源泉。
>
> ——陶铸

本节中，我们先跳出源码的视角，来看看集合类的类图，看看在设计层面上，是否有可疑借鉴之处，接着通过源码来找找工作中的集合坑，提前扫雷。

1 集合类图



上图是目前我们已学的集合类图，大概可以看出以下几点：

1. 每个接口做的事情非常明确，比如 Serializable，只负责序列化，Cloneable 只负责拷贝，Map 只负责定义 Map 的接口，整个图看起来虽然接口众多，但职责都很清晰；
2. 复杂功能通过接口的继承来实现，比如 ArrayList 通过实现了 Serializable、Cloneable、RandomAccess、AbstractList、List 等接口，从而拥有了序列化、拷贝、对数组各种操作定义等各种功能；
3. 上述类图只能看见继承的关系，组合的关系还看不出来，比如说 Set 组合封装 Map 的底层能力等。

目录

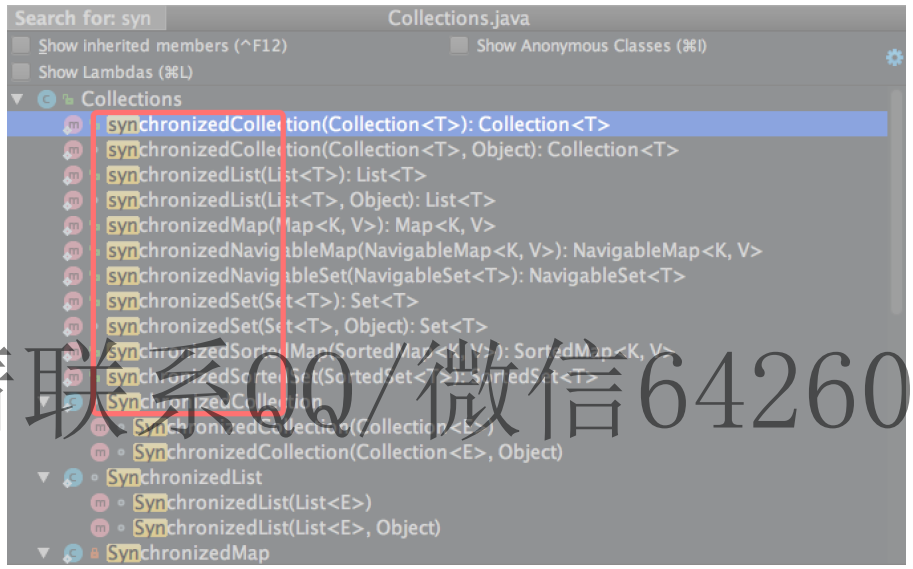
口，进行一些简单的组装，从而加快开发速度。

这种思想在平时的工作中也经常被使用，我们会把一些通用的代码块抽象出来，沉淀成代码块池，碰到不同的场景的时候，我们就从代码块池中，把我们需要的代码块提取出来，进行简单的编排和组装，从而实现我们需要的场景功能。

2 集合工作中一些注意事项

2.1 线程安全

我们说集合都是非线程安全的，这里说的非线程安全指的是集合类作为共享变量，被多线程读写的时候，才是不安全的，如果要实现线程安全的集合，在类注释中，JDK 统一推荐我们使用 Collections.synchronized* 类， Collections 帮我们实现了 List、Set、Map 对应的线程安全的方法， 如下图：



图中实现了各种集合类型的线程安全的方法，我们以 synchronizedList 为例，从源码上来看下，Collections 是如何实现线程安全的：

```
// mutex 就是我们需要锁住的对象
final Object mutex;
static class SynchronizedList<E>
        extends SynchronizedCollection<E>
        implements List<E> {
        private static final long serialVersionUID = -7754090372962971524L;
        // 这个 List 就是我们需要保证线程安全的类
        final List<E> list;
        SynchronizedList(List<E> list, Object mutex) {
            super(list, mutex);
            this.list = list;
        }
                        // 我们可以看到，List 的所有操作都使用了 synchronized 关键字，来进行加
                        // synchronized 是一种悲观锁，能够保证同一时刻，只能有一个线程能够获
        public E get(int index) {
            synchronized (mutex) {return list.get(index);}
        }
        public E set(int index, E element) {
            synchronized (mutex) {return list.set(index, element);}
        }
        public void add(int index, E element) {
            synchronized (mutex) {list.add(index, element);}
```

目录

从源码中我们可以看到 Collections 是通过 synchronized 关键字给 List 操作数组的方法加上锁，来实现线程安全的。

2.2 集合性能

集合的单个操作，一般都没有性能问题，性能问题主要出现的批量操作上。

2.2.1 批量新增

在 List 和 Map 大量数据新增的时候，我们不要使用 for 循环 + add/put 方法新增，这样子会有很大的扩容成本，我们应该尽量使用 addAll 和 putAll 方法进行新增，以 ArrayList 为例写了一个 demo 如下，演示了两种方案的性能对比：

```
@Test
public void testBatchInsert(){
  // 准备拷贝数据
  ArrayList<Integer> list = new ArrayList<>();
  for(int i=0;i<3000000;i++){
    list.add(i);
  }

  // for 循环 + add
  ArrayList<Integer> list2 = new ArrayList<>();
  long start1 = System.currentTimeMillis();
  for(int i=0;i<list.size();i++){
    list2.add(list.get(i));
  }
  log.info("单个 for 循环新增 300 w 个，耗时{}",System.currentTimeMillis()-start1);

  // 批量新增
  ArrayList<Integer> list3 = new ArrayList<>();
  long start2 = System.currentTimeMillis();
  list3.addAll(list);
  log.info("批量新增 300 w 个，耗时{}",System.currentTimeMillis()-start2);
}
```


最后打印出来的日志为：

16:52:59.865 [main] INFO demo.one.ArrayListDemo - 单个 for 循环新增 300 w 个，耗时1518
16:52:59.880 [main] INFO demo.one.ArrayListDemo - 批量新增 300 w 个，耗时8

可以看到，批量新增方法性能是单个新增方法性能的 189 倍，主要原因在于批量新增，只会扩容一次，大大缩短了运行时间，而单个新增，每次到达扩容阀值时，都会进行扩容，在整个过程中就会不断的扩容，浪费了很多时间，我们来看下批量新增的源码：

```
public boolean addAll(Collection<? extends E> c) {
  Object[] a = c.toArray();
  int numNew = a.length;
  // 确保容量充足，整个过程只会扩容一次
  ensureCapacityInternal(size + numNew);
  // 进行数组的拷贝
  System.arraycopy(a, 0, elementData, size, numNew);
```

目录

以上是 ArrayList 批量新增的演示，我们可以看到，整个批量新增的过程中，只扩容了一次，HashMap 的 putAll 方法也是如此，整个新增过程只会扩容一次，大大缩短了批量新增的时间，提高了性能。

所以如果有人问你当碰到集合批量拷贝，批量新增场景，如何提高新增性能的时候 ，就可以从目标集合初始化方面应答。

这里也提醒了我们，在容器初始化的时候，最好能给容器赋上初始值，这样可以防止在 put 的过程中不断的扩容，从而缩短时间，上章 HashSet 的源码给我们演示了，给 HashMap 赋初始值的公式为：取括号内两者的最大值（期望的值/0.75+1，默认值 16）。

2.2.2 批量删除

批量删除 ArrayList 提供了 removeAll 的方法，HashMap 没有提供批量删除的方法，我们一起来看下 removeAll 的源码实现，是如何提高性能的：

```
// 批量删除，removeAll 方法底层调用的是 batchRemove 方法
// complement 参数默认是 false,false 的意思是数组中不包含 c 中数据的节点往头移动
// true 意思是数组中包含 c 中数据的节点往头移动，这个是根据你要删除数据和原数组大小的比例来
// 如果你要删除的数据很多，选择 false 性能更好，当然 removeAll 方法默认就是 false。
private boolean batchRemove(Collection<?> c, boolean complement) {
  final Object[] elementData = this.elementData;
  // r 表示当前循环的位置、w 位置之前都是不需要被删除的数据，w 位置之后都是需要被删除的数据
  int r = 0, w = 0;
  boolean modified = false;
  try {
    // 从 0 位置开始判断，当前数组中元素是不是要被删除的元素，不是的话移到数组头
    for (; r < size; r++)
      if (c.contains(elementData[r]) == complement)
        elementData[w++] = elementData[r];
  } finally {
    // r 和 size 不等，说明在 try 过程中发生了异常，在 r 处断开
    // 把 r 位置之后的数组移动到 w 位置之后(r 位置之后的数组数据都是没有判断过的数据，这样不会
    if (r != size) {
      System.arraycopy(elementData, r,
                       elementData, w,
                       size - r);
      w += size - r;
    }
    // w != size 说明数组中是有数据需要被删除的
    // 如果 w、size 相等，说明没有数据需要被删除
    if (w != size) {
      // w 之后都是需要删除的数据，赋值为空，帮助 gc。
      for (int i = w; i < size; i++)
        elementData[i] = null;
      modCount += size - w;
      size = w;
      modified = true;
    }
  }
  return modified;
}
```

目录

元素正好是数组最后一个元素时除外），当数组越大，需要删除的数据越多时，批量删除的性能会越差，所以在 ArrayList 批量删除时，强烈建议使用 removeAll 方法进行删除。

2.3 集合的一些坑

1. 当集合的元素是自定义类时，自定义类强制实现 equals 和 hashCode 方法，并且两个都要实现。

在集合中，除了 TreeMap 和 TreeSet 是通过比较器比较元素大小外，其余的集合类在判断索引位置和相等时，都会使用到 equals 和 hashCode 方法，这个在之前的源码解析中，我们有说到，所以当集合的元素是自定义类时，我们强烈建议覆写 equals 和 hashCode 方法，我们可以直接使用 IDEA 工具覆写这两个方法，非常方便；

2. 所有集合类，在 for 循环进行删除时，如果直接使用集合类的 remove 方法进行删除，都会快速失败，报 ConcurrentModificationException 的错误，所以在任意循环删除的场景下，都建议使用迭代器进行删除；

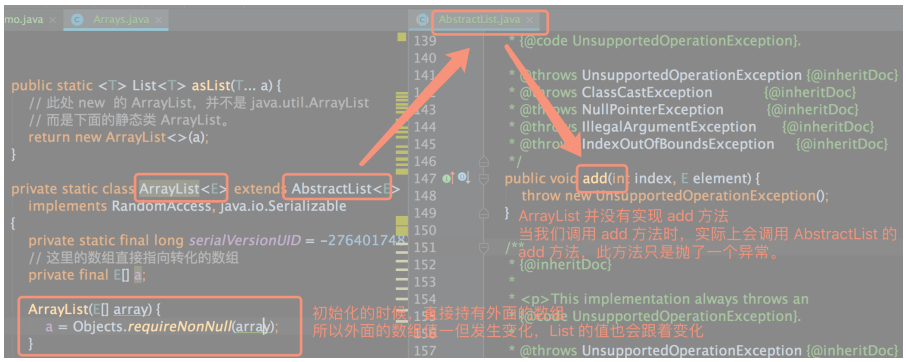
3. 我们把数组转化成集合时，常使用 Arrays.asList(array)，这个方法有两个坑，代码演示坑为：

```
public void testArrayToList(){
  Integer[] array = new Integer[]{1,2,3,4,5,6};
  List<Integer> list = Arrays.asList(array);

  // 坑1：修改数组的值，会直接影响原 list
  log.info("数组被修改之前，集合第一个元素为：{}",list.get(0));
  array[0] = 10;
  log.info("数组被修改之前，集合第一个元素为：{}",list.get(0));

  // 坑2：使用 add、remove 等操作 list 的方法时，
  // 会报 UnsupportedOperationException 异常
  list.add(7);
}
坑 1：数组被修改后，会直接影响到新 List 的值。
坑 2：不能对新 List 进行 add、remove 等操作，否则运行时会报 UnsupportedOperationExceptio
```


我们来看下 Arrays.asList 的源码实现，就能知道问题所在了，源码如下图：



从上图中，我们可以发现，Arrays.asList 方法返回的 List 并不是 java.util.ArrayList，而是自己内部的一个静态类，该静态类直接持有数组的引用，并且没有实现 add、remove 等方法，这些就是坑 1 和 2 的原因。

目录

```
public void testListToArray(){
  List<Integer> list = new ArrayList<Integer>(){{
    add(1);
    add(2);
    add(3);
    add(4);
  }};

  // 下面这行被注释的代码这么写是无法转化成数组的，无参 toArray 返回的是 Object[],
  // 无法向下转化成 List<Integer>，编译都无法通过
  // List<Integer> list2 = list.toArray();

  // 演示有参 toArray 方法，数组大小不够时，得到数组为 null 情况
  Integer[] array0 = new Integer[2];
  list.toArray(array0);
  log.info("toArray 数组大小不够，array0 数组[0] 值是{},数组[1] 值是{},",array0[0],array0[1]);

  // 演示数组初始化大小正好，正好转化成数组
  Integer[] array1 = new Integer[list.size()];
  list.toArray(array1);
  log.info("toArray 数组大小正好，array1 数组[3] 值是{}",array1[3]);

  // 演示数组初始化大小大于实际所需大小，也可以转化成数组
  Integer[] array2 = new Integer[list.size()+2];
  list.toArray(array2);
  log.info("toArray 数组大小多了，array2 数组[3] 值是{}，数组[4] 值是{}",array2[3],array2[4]);
}
19:33:07.687 [main] INFO demo.one.ArrayListDemo - toArray 数组大小不够，array0 数组[0] 值
19:33:07.697 [main] INFO demo.one.ArrayListDemo - toArray 数组大小正好，array1 数组[3] 值
19:33:07.697 [main] INFO demo.one.ArrayListDemo - toArray 数组大小多了，array2 数组[3] 值
```


toArray 的无参方法，无法强转成具体类型，这个编译的时候，就会有提醒，我们一般都会去使用带有参数的 toArray 方法，这时就有一个坑，如果参数数组的大小不够，这时候返回的数组值竟然是空，上述代码中的 array0 的返回值就体现了这点，但我们去看 toArray 源码，发现源码中返回的是 4 个大小值的数据，返回的并不是空，源码如下：

```
// List 转化成数组
public <T> T[] toArray(T[] a) {
  // 如果数组长度不够，按照 List 的大小进行拷贝，return 的时候返回的都是正确的数组
  if (a.length < size)
    // Make a new array of a's runtime type, but my contents:
    return (T[]) Arrays.copyOf(elementData, size, a.getClass());
  System.arraycopy(elementData, 0, a, 0, size);
  // 数组长度大于 List 大小的，赋值为 null
  if (a.length > size)
    a[size] = null;
  return a;
}
```

从源码中，我们丝毫看不出为什么 array0 的元素值为什么是 null，最后我们去看方法的注释，发现是这样子描述的：

```
If the list fits in the specified array, it is returned therein.  Otherwise, a new array is
 allocated with the runtime type of the specified array and the size of this list。
```

目录

所以我们在使用有参 toArray 方法时，申明的数组大小一定要大于等于 List 的大小，如果小于的话，你会得到一个空数组。

3 总结

本小节，我们详细描述了集合的线程安全、性能优化和日常工作中一些坑，这些问题我们在工作中经常会碰到，稍不留神就会引发线上故障，面试的时候也经常会通过这些问题，来考察大家的工作经验，所以阅读本章时，建议大家自己动手试一试，加深印象。

← 11 HashSet、TreeSet 源码解析

13 差异对比：集合在 Java 7 和 8 有何不同和改进 →

精选留言 6

欢迎在这里发表留言，作者筛选后可公开显示

风舞炫动

亲手debug完才理解了为什么数组值小于list时，数组值为null。 还有一个问题就是通过Arrays.asList得到的ArrayList其实是Arrays内部自己实现的，跟ArrayList.class的ArrayList可以理解为没有啥关系对吧。之前一直以为是同一个ArrayList

0 回复 2019-11-20

文贺 回复 **风舞炫动**

是的，两个 List 是不同的。

回复 2019-11-23 16:41:36

Sicimike

public T[] toArray(T[] a)方法有返回值。使用该方法时，即使传入的数组长度小于list长度，返回值也是正确的。

0 回复 2019-11-12

文贺 回复 **Sicimike**

同学你理解有误哈，可以 debug 下，如果数组长度小于 list 的实际大小，是无法返回正确的值的哈。

回复 2019-11-17 10:50:39

慕盖茨4571687

final Object[] elementData = this.elementData; 老师我想问一下 这行代码把list存放的数组赋值给一个临时变量，底下代码一直操作这个临时变量也没操作原来的数组呀?

0 回复 2019-10-17

目录

回复 2019-10-17 19:07:56

Elylic

// 坑1：修改数组的值，会直接影响原 list log.info("数组被修改之前，集合第一个元素为：{}",list.get(0)); array[0] = 10; log.info("数组被修改之前，集合第一个元素为：{}",list.get(0)); 这里第二个log应该是“数组被修改之后”吧！

0 回复 2019-09-19

slvayf

老师好，请问学习专栏和网络上其他视频课程的时候，真正高效的学习方式是什么？记笔记的过程重要吗？（感觉很少会回头再看，只有常用的东西会记住，那些已经理解并自己写了demo的，不常用就会慢慢忘掉，只能回想起自己学过）

3 回复 2019-09-18

文贺 回复 **slvayf**

你说的很对，不常用的，过段时间就会忘记，所以重复温习很重要哈，你做的笔记正好是你温习的最快简捷，重复温习很重要哈，没有什么捷径的。

回复 2019-09-18 20:46:28

蜗牛Baby

顶一个，自律狂魔，老师别停！

1 回复 2019-09-18

千学不如一看，千看不如一练