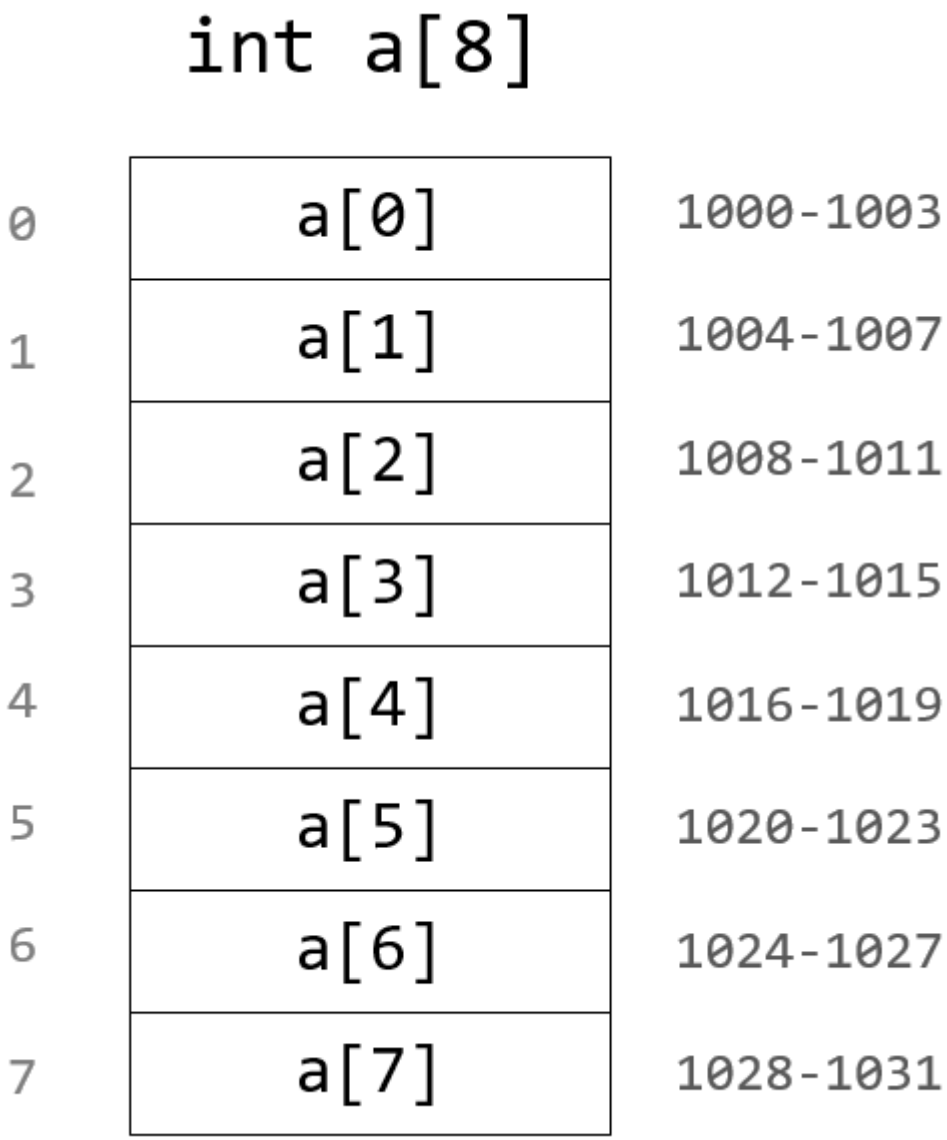
> 2018年10月14日

基本上每一种编程语言都有数组这种数据类型，数组就是用一组连续的内存空间，来存储一组具有相同类型的数据。

1，数组随机访问

在大部分编程语言中，如C/C++，Java，其数组下标从0开始编号。以一个长度为8的`int`型数组为例：`int []a = new int[8]`；其中分配了一块连续内存空间`1000-1031`，内存块的首地址`base_address=1000`。



计算机给每个内存单元分配一个地址，计算机通过地址来访问内存中的数据；当计算机需要随机访问数组中的某个元素时，它会通过下面的寻址公式，计算出该元素存储的内存地址：$a[i]\_address = base\_address + i * data\_type\_size$ 其中`data_type_size`为数组中每个元素所占字节数；在我们所举的例子中采用的是`int`类型，所以`data_type_size`为4个字节。

如果数组的下标是从`1`开始，那么其寻址公式为：$a[i]\_address = base\_address + (i-1) * data\_type\_size$ 则每次随机访问数组时都会多做一次减法运算；由于数组作为一种非常基础的数据结构，其性能优化就要做到极致，因此数组下标一般从`0`开始，避免进行减法运算。

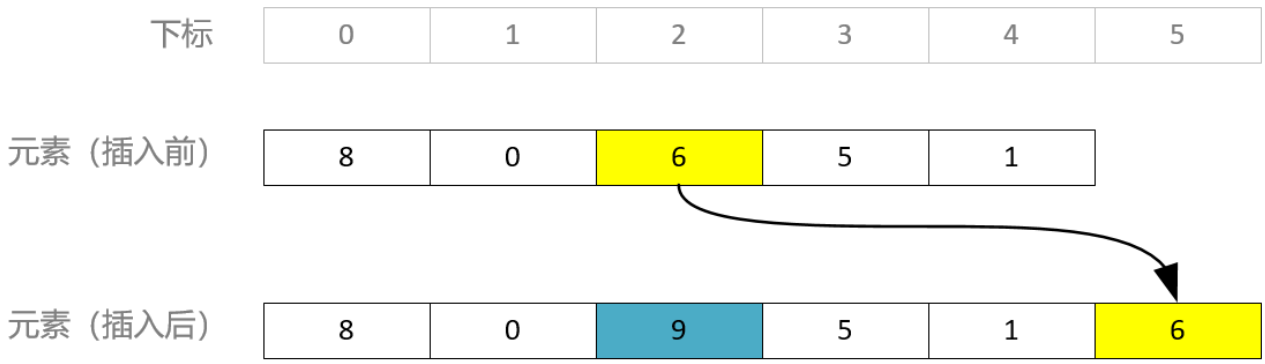
因此数组进行随机访问时，只需计算出该元素的内存地址即可，其时间复杂度为 $O(1)$；而对于有序数组进行二分查找，其时间复杂度为 $O(logN)$。

2，插入与删除

假设数组的长度为`N`，若我们需要将一个数据插入到数组的第`k`个位置，那么为了将第`k`个位置腾出来，需要先将`k~n`这部分元素都往后挪一位。那么其时间复杂度为：

- **最好时间复杂度**：在数组的末尾插入元素，此时不用挪动其他任何元素，此时时间复杂度为 $O(1)$。
- **最坏时间复杂度**：在数组的头部插入元素，此时数组中所有元素都要往后挪一位，此时时间复杂度为 $O(N)$。
- **平均时间复杂度**：由于一共有`N+1`中插入情况，那么其平均时间复杂度为：$\frac{N+...+2+1+0}{N+1} = \frac{N(N+1)}{2(N+1)} = O(N)$

若数组中元素是有序的，那么我们在某一个位置插入元素时，就必须按照上述方法对其后面的元素进行挪动。但是，若数组中的元素没有任何规律，为了避免元素的大规模挪动，我们可以先将位置`k`上的元素插入到数组末尾，在将位置`k`上的元素替换为我们要插入的元素；举个栗子：假设`a[8]`中存储了以下5个元素：`8，0，6，5，1`；现在需要将元素`9`插入到第`3`个位置，那么就只需将`6`放入到`a[5]`中，然后将`a[2]`赋值为`9`即可，如下图：



与插入操作类似，若要删除数组中第`k`个位置的元素，为了内存的连续性，需要挪动相应位置的元素；删除操作的时间复杂度与插入操作类似：最好时间复杂度即在数组末尾删除为 $O(1)$，最坏时间复杂度即在数组头部进行删除为 $O(N)$，平均时间复杂度为 $O(N)$。

3，数组越界问题

首先来分析一段c语言代码：

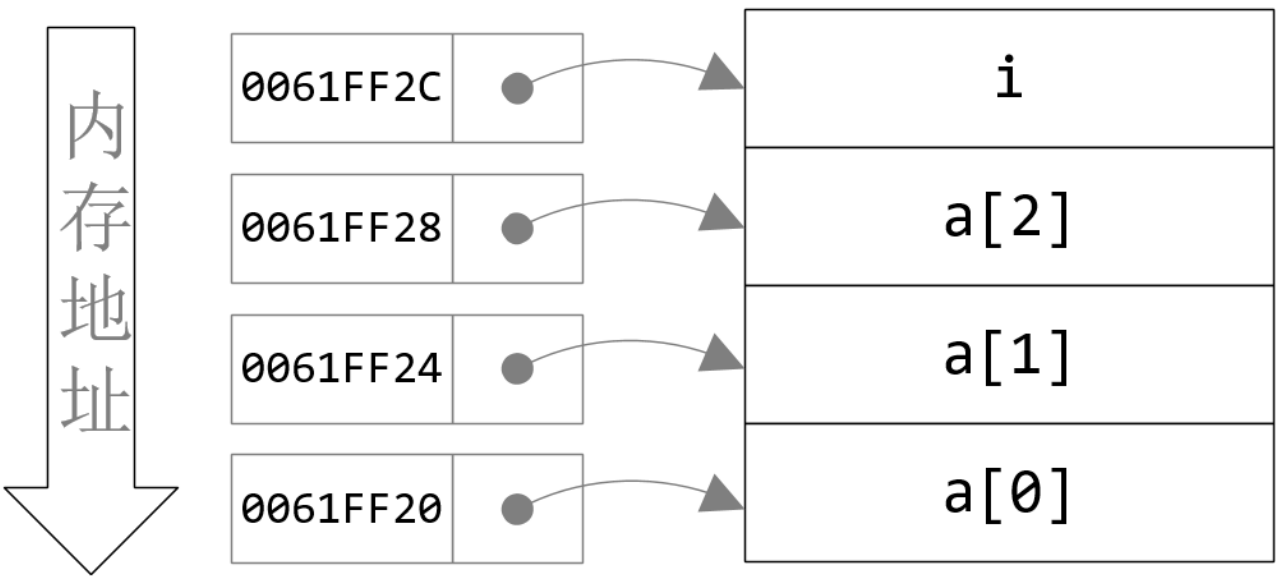
```c
#include <stdio.h>
int main(void) {
    int i = 0;
    printf("i的内存地址: %p\n", &i);
    int array[3] = {0, 1, 2};
    for (; i <= 3; i++) {
        if (i == 3) {
            printf("a[3]的内存地址: %p\n", &array[3]);
            array[i] = 0;
        }
        printf("a[%d]=%d, 其内存地址: %p\n", i,array[i], &array[i]);
    }
    return 0;
}
```

其运行结果为：

```c
#include <stdio.h>
int main(void) {
    int i = 0;
    printf("i的内存地址: %p\n", &i);
    int array[3] = {0, 1, 2};
    for (; i <= 3; i++) {
        if (i == 3) {
            printf("a[3]的内存地址: %p\n", &array[3]);
            array[i] = 0;
        }
        printf("a[%d]=%d, 其内存地址: %p\n", i,array[i], &array[i]);
    }
    return 0;
}
```

```
main
Run: myTest
D:\CCode\myTest\cmake-build-debug\myTest.exe
i的内存地址: 0061FF2C
a[0]=0, 其内存地址: 0061FF20
a[1]=1, 其内存地址: 0061FF24
a[2]=2, 其内存地址: 0061FF28
a[3]的内存地址: 0061FF2C
a[0]=0, 其内存地址: 0061FF20
a[1]=1, 其内存地址: 0061FF24
a[2]=2, 其内存地址: 0061FF28
```

可以看出，该段代码循环输出第`11`行，这是怎么回事呢？我们来看看变量的栈帧结构：



如上图所示，栈地址是由高到低增长的，由上面的寻址公式可知：
$a[3]\_address = 0061FF20 + 3 \times 4 = 0061FF2C$，即`a[3]`与`i`的内存地址相同，因此第`9`行代码`array[i] = 0`也就是将`i`的值赋为`0`，因此该段代码就陷入死循环，这是数组越界带来的危害。

而在Java语言中，会做越界检查，如下面的Java代码：

```
int[] array = new int[8];
a[8] = 8;
```

此段代码会抛出java.lang.ArrayIndexOutOfBoundsException异常。

4，容器

JAVA中的ArrayList类将很多数组操作的细节封装起来，如插入、删除时需要挪动其他元素等，而且，还支持动态扩容。

由于数组在定义时需要预先指定大小，因为需要分配连续的空间。如果申请了大小为8的数组，那么当第9个元素需要存储到数组时，就需要重新分配一块更大的空间，并将原来的元素复制过去，然后再将新的元素插入。

在JDK中实现的ArrayList中，每次空间不够时，会将空间自动扩容为原来的1.5倍；因为扩容需要内存申请和数据复制，比较耗时，所以，若实现能确定需要存储的数据大小，那么最好在创建ArrayList时应指定大小。

在使用ArrayList时要注意fail-fast问题，也就是快速失败，这是一种JAVA集合检测错误的机制，即当一个线程在使用迭代器遍历集合中的元素时，集合自身的方法修改了集合结构（如使用add或remove方法），或另一个线程中的迭代器或集合自身的方法修改了集合的结构，就会抛出一个ConcurrentModificationException异常。

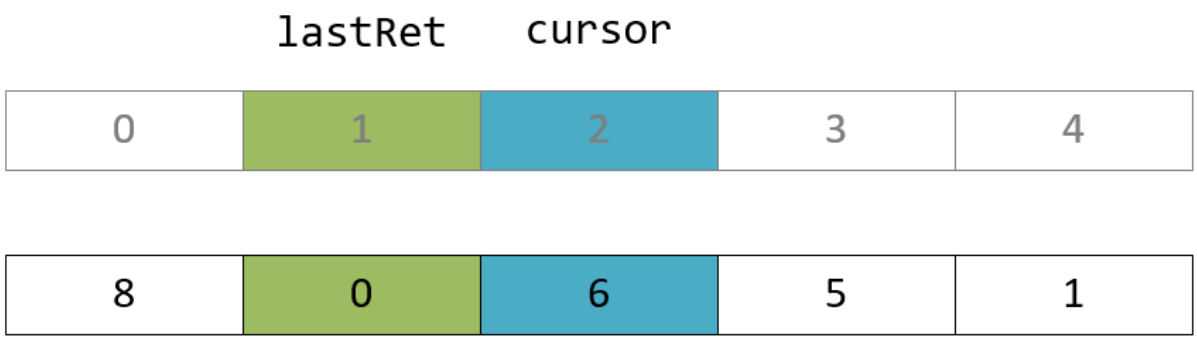
如何判断ConcurrentModificationException异常？在集合中有一个modCount变量，集合中每次有元素添加或删除，都会使modCount自增；而在迭代器中有一个变量expectedModCount，在初始化iterator时，expectedModCount = modCount，所以，一旦集合中结构发生改变，expectedModCount != modCount，当触发这个条件时，就会抛出ConcurrentModificationException异常。

以下是ArrayList中迭代器的hasNext(), next(), remove()方法源码：

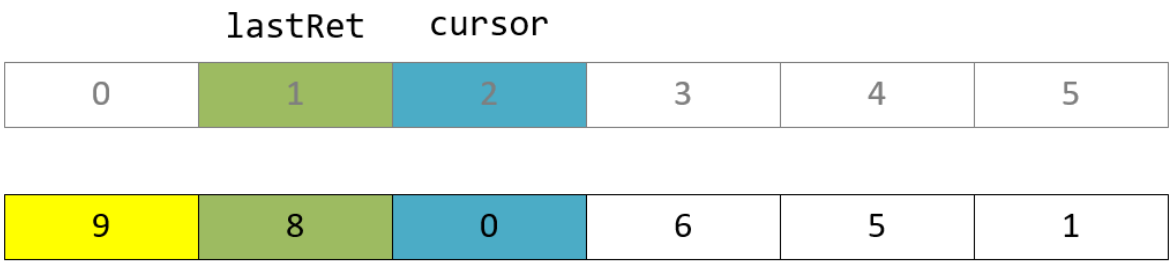
```
        public boolean hasNext() {
            return cursor != size;
        }

        @SuppressWarnings("unchecked")
        public E next() {
            checkForComodification();
            int i = cursor;
```

```
            if (i >= size)
                throw new NoSuchElementException();
            Object[] elementData = ArrayList.this.elementData;
            if (i >= elementData.length)
                throw new ConcurrentModificationException();
            cursor = i + 1;
            return (E) elementData[lastRet = i];
        }

        public void remove() {
            if (lastRet < 0)
                throw new IllegalStateException();
            checkForComodification();

            try {
                ArrayList.this.remove(lastRet);
                cursor = lastRet;
                lastRet = -1;
                expectedModCount = modCount;
            } catch (IndexOutOfBoundsException ex) {
                throw new ConcurrentModificationException();
            }
        }
```

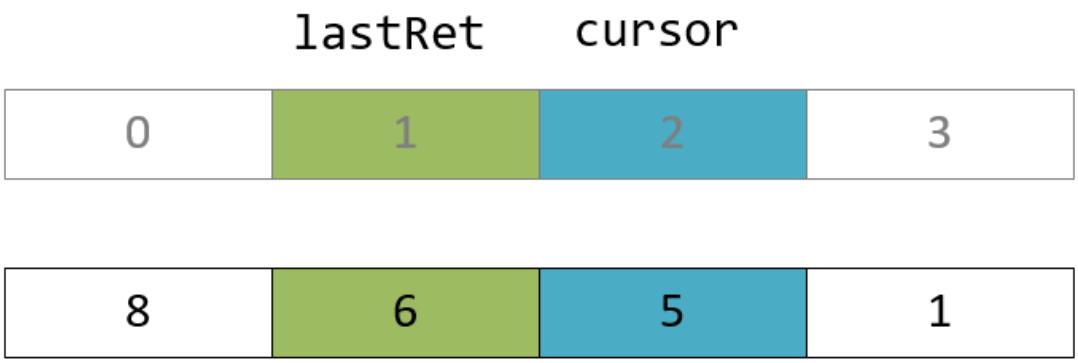
其中`cursor`是下一个将要遍历元素的下标，`lastRet`为刚刚遍历的元素下标；我们使用迭代器来遍历元素，使用两次`next`后，当前元素下标`lastRest`为`1`，那么下一个要遍历元素的下标`cursor`为`2`，此时已遍历的元素为`8, 0`：



- 使用集合中方法`add`在头部插入元素`9`后：



  因为插入元素后，集合结构发生改变，相应的元素都往后挪动了，此时的`lastRet`应为2，`cursor`应为3；但是，集合中`add`方法并不知晓迭代器中元素下标情况，没有对`lastRet`和`cursor`做出相应修改，所以，此时，`cursor仍然=2`，那么此时使用`next`，`i=cursor=2`，返回的元素：`elementData[lastRet=i]=0`，已遍历的元素为：`8，0，0`，重复遍历了元素`0`。
- 使用集合中的`remove`删除元素`0`后：



  删除元素后，相应的元素往前挪动了，此时的`cursor`应为1，同理，集合中`remove`方法并不对`cursor`做出修改；所以，此时`cursor仍然=2`，此时使用`next`，`i=cursor=2`，返回的元素`elementData[lastRet=i]=5`，那么此时已遍历的元素为：`8，0，5`，从而遗漏了元素`6`。

因此，在单线程中使用迭代器进行元素遍历时，若要对集合进行修改，应使用`iterator`中的`add`或`remove`方法；在多线程中，使用`CopyOnWriteArrayList`来替代`ArrayList`，该集合读写分离，写操作在一个复制的数组上进行，读操作还是在原始数组中进行，读写分离，互不影响。写操作需要加锁，防止并发写入时导致写入数据丢失。写操作结束之后需要把原始数组指向新的复制数组。

下面是实现了一个自定义的`ArrayList`，只是简单的实现了相应的功能，具体的细节还是得看`JDK`中的源码：

```
1 import java.util.Iterator;
```

```java
import java.util.NoSuchElementException;

/**
 * @author: Hello World
 * @date: 2018/10/11 22:12
 */
public class MyArrayList<AnyType> implements Iterable<AnyType> {
    private static final int DEFAULT_CAPACITY = 10;

    private AnyType[] theItems;
    private int theSize;

    public MyArrayList() {
        doClear();
    }

    public void clear() {
        doClear();
    }

    private void doClear() {
        theSize = 0;
        ensureCapacity(DEFAULT_CAPACITY);
    }

    public int size() {
        return theSize;
    }

    public boolean isEmpty() {
        return size() == 0;
    }

    public void trimToSize() {
        ensureCapacity(size());
    }

    public AnyType get(int index) {
        if (index <= 0 || index >= size()) {
            throw new ArrayIndexOutOfBoundsException();
        }
        return theItems[index];
```

```
    }

    public AnyType set(int index, AnyType element) {
        if (index < 0 || index >= size()) {
            throw new ArrayIndexOutOfBoundsException();
        }
        AnyType old = theItems[index];
        theItems[index] = element;
        return old;
    }

    private void ensureCapacity(int newCapacity) {
        if (newCapacity < size()) {
            return;
        }
        AnyType[] old = theItems;
        theItems = (AnyType[]) new Object[newCapacity];
        for (int i = 0; i < size(); i++) {
            theItems[i] = old[i];
        }
    }

    public boolean add(AnyType element) {
        add(size(), element);
        return true;
    }

    public void add(int index, AnyType element) {
        if (theItems.length == size()) {
            //+1是针对size()=0的情况(使用remove将数组元素移空了)
            ensureCapacity(size() * 2 + 1);
        }
        for (int i = size(); i > index; i--) {
            theItems[i] = theItems[i - 1];
        }
        theItems[index] = element;

        theSize++;
    }

    public AnyType remove(int index) {
        AnyType removedElement = theItems[index];
```

```
        for (int i = index; i < size() - 1; i++) {
            theItems[i] = theItems[i + 1];
        }

        //GC时将其数组末尾标记为垃圾进行回收
        theItems[--theSize] = null;
        return removedElement;
    }

    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder("[ ");

        //增强for循环调用的是iterator实现的
        for (AnyType element : this) {
            sb.append(element + " ");
        }
        sb.append("]");

        return sb.toString();
    }

    @Override
    public Iterator<AnyType> iterator() {
        return new ArrayListIterator();
    }

    private class ArrayListIterator implements Iterator<AnyType> {
        private int current = 0;
        //迭代器中进行remove操作前必须进行next操作
        private boolean okToRemove = false;

        @Override
        public boolean hasNext() {
            return current < size();
        }

        @Override
        public AnyType next() {
            if (!hasNext()) {
                throw new NoSuchElementException();
            }
```

```
128
129             okToRemove = true;
130             return theItems[current++];
131         }
132
133         @Override
134         public void remove() {
135             if (!okToRemove) {
136                 throw new IllegalStateException();
137             }
138
139             MyArrayList.this.remove(--current);
140             okToRemove = false;
141         }
142     }
143 }
```