

Summary of Common Java Collection Interview Questions (Part 1)

 [Guide](#)  Java  Java Collections  About 7375 words  About 25 minutes

This is a small advertisement that may be useful to you

- **Interview special edition** : Friends who are preparing for Java interviews can consider the interview special edition: "[Java Interview Guide](#)" (very high quality, specially designed for interviews, best used with JavaGuide).
- **Knowledge Planet** : Technical Column/One-on-one Questions/Resume Modification/Job Hunting Guide/Interview Check-in/Irregular Benefits, welcome to join [the JavaGuide official Knowledge Planet](#) .

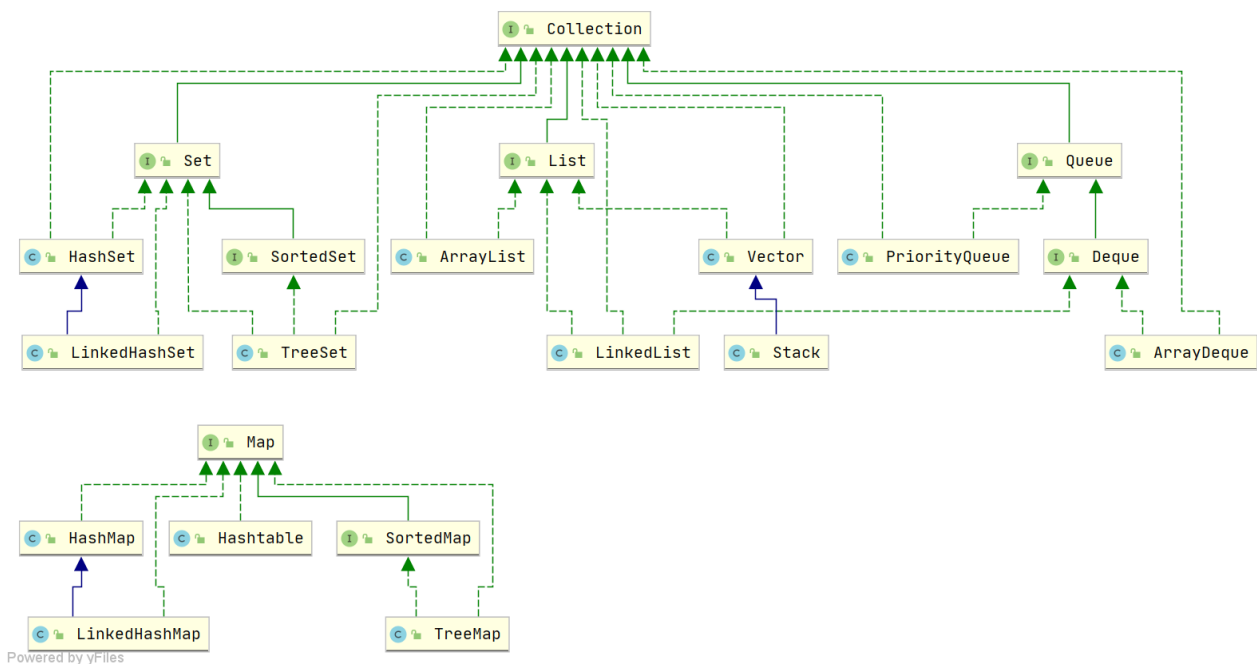
Collection Overview

Java Collections Overview

Java collections, also known as containers, are derived from two main interfaces: the `Collection` interface, which is used to store single elements, and `Map` the `Collection` interface, which is used to store key-value pairs. The `Collection` interface has three main subinterfaces: `List`, `Set` and `Queue`.

The Java Collection Framework is shown in the following figure:





Note: The diagram only lists the main inheritance and derivation relationships, not all relationships. For example `AbstractList`, `NavigableSet` abstract classes such as and some other auxiliary classes are omitted. If you want to learn more, you can view the source code yourself.

★ What are the differences between List, Set, Queue, and Map?

- **List** (A good helper for dealing with order): The stored elements are ordered and repeatable.
- **Set** (Focus on unique properties): The stored elements cannot be repeated.
- **Queue** (A calling machine that implements queuing function): The order of priority is determined according to specific queuing rules, and the stored elements are ordered and repeatable.
- **Map** (Experts who use keys to search): Use key-value pairs for storage, which is similar to the mathematical function $y=f(x)$, where "x" represents the key and "y" represents the value. The key is unordered and non-repeatable, and the value is unordered and repeatable. Each key maps to at most one value.

Summary of the underlying data structure of the collection framework



Let's first look at `Collection` the collection under the interface.

List

- `ArrayList` : `Object[]` Array. For details, see: [ArrayList Source Code Analysis](#) .
- `Vector` : `Object[]` array.
- `LinkedList` : A doubly linked list (a circular linked list before JDK 1.6, but the loop is removed in JDK 1.7). For more details, see [LinkedList Source Code Analysis](#) .

Set


- `HashSet` (unordered, unique): `HashMap` implemented based on , the underlying layer uses `HashMap` to save elements.
- `LinkedHashSet` : `LinkedHashSet` is `HashSet` a subclass of , and its internal structure is `LinkedHashMap` implemented by .
- `TreeSet` (Ordered, unique): Red-black tree (self-balancing sorted binary tree).

Queue

- `PriorityQueue` : `Object[]` array to implement a mini-heap. For details, see: [PriorityQueue source code analysis](#) .
- `DelayQueue` : `PriorityQueue` . For details, please see: [DelayQueue source code analysis](#) .
- `ArrayDeque` : Expandable dynamic bidirectional array.

Let's take a look at `Map` the collection under the interface.

Map

- `HashMap` Before JDK 1.8, `HashMap` `HashMap` consisted of an array and a linked list `HashMap` . The array was the core of `HashMap`, while the linked list was primarily used to resolve hash conflicts (using the "zipper" method). JDK 1.8 and later implemented a significant change in hash conflict resolution. When the linked list length exceeds a threshold (default 8), the linked list is converted to a red-black tree to reduce search time. (This check is performed before converting the linked list to a red-black tree. If the current array length is less than 64, the array is expanded rather than converted to red-black tree.) For more details, see [HashMap Source Code Analysis](#) . 
- `LinkedHashMap` : `LinkedHashMap` Inherited from `HashMap` , so its underlying structure is still based on the zipper hash structure, which is composed of an array and a

linked list or a red-black tree. In addition, `LinkedHashMap` based on the above structure, a bidirectional linked list is added so that the above structure can maintain the insertion order of key-value pairs. At the same time, by performing corresponding operations on the linked list, the access order-related logic is implemented. For details, please see: [LinkedHashMap Source Code Analysis](#)

- `Hashtable` : It is composed of array + linked list. The array is `Hashtable` the main body of , and the linked list exists mainly to resolve hash conflicts.
- `TreeMap` : Red-black tree (self-balancing sorted binary tree).

How to choose a collection?

We mainly choose the appropriate collection based on the characteristics of the collection. For example:

- When we need to obtain the element value according to the key value, we choose `Map` the collection under the interface. When we need to sort `TreeMap` , we choose it. When we don't need to sort `HashMap` , we choose it. When we need to ensure thread safety, we choose it `ConcurrentHashMap` .
- When we only need to store element values, we choose `Collection` a collection that implements the interface. When we need to ensure that the elements are unique, we choose `Set` a collection that implements the interface, such as `TreeSet` or `HashSet` . If we don't need it, we choose a collection that implements `List` the interface, such as `ArrayList` or `LinkedList` . Then we choose according to the characteristics of the collections that implement these interfaces.

Why use collections?

When we need to store a group of data of the same type, arrays are one of the most commonly used and fundamental containers. However, using arrays to store objects has some drawbacks, as in real-world development, the types of data stored vary and the number of data stored is uncertain. This is where Java collections come in handy. Compared to arrays, Java collections provide a more flexible and efficient way to store multiple data objects. The various collection classes and interfaces in the Java Collections Framework can store objects of varying types and quantities, while also offering a wide variety of operations. Compared to arrays, Java collections offer advantages such as resizable size, support for generics, and built-in algorithms. Overall, Java collections provide increased flexibility in data storage and processing, better adapting to the diverse data needs of modern software development and supporting high-quality code development.



List

★ What is the difference between ArrayList and Array?

ArrayList Internally, it is implemented based on a dynamic array, which **Array** is more flexible to use than a static array:

- **ArrayList** It will dynamically expand or shrink according to the actual elements stored, and **Array** its length cannot be changed after it is created.
- **ArrayList** Allowing you to use generics to ensure type safety, **Array** you can't.
- **ArrayList** Only objects can be stored in it. For primitive data types, you need to use their corresponding wrapper classes (such as Integer, Double, etc.). **Array** You can store primitive data types directly or objects.
- **ArrayList** It supports common operations such as insertion, deletion, and traversal, and provides a rich set of API operation methods, such as `add()` , `remove()` etc.
Array It is just an array of fixed length, and the elements can only be accessed according to the subscript, and it does not have the ability to dynamically add or delete elements.
- **ArrayList** The size does not need to be specified when creating it, but **Array** it must be specified when creating it.

Here is a simple comparison of the two:

Array :

```
1 // 初始化一个 String 类型的数组
2 String[] stringArr = new String[]{"hello", "world", "!"};
3 // 修改数组元素的值
4 stringArr[0] = "goodbye";
5 System.out.println(Arrays.toString(stringArr)); // [goodbye, world,
6 !]
7 // 删除数组中的元素, 需要手动移动后面的元素
8 for (int i = 0; i < stringArr.length - 1; i++) {
9     stringArr[i] = stringArr[i + 1];
10 }
11 stringArr[stringArr.length - 1] = null;
    System.out.println(Arrays.toString(stringArr)); // [world, !, null]
```

ArrayList :

```
1 // 初始化一个 String 类型的 ArrayList java
2 ArrayList<String> stringList = new ArrayList<>
3 (Arrays.asList("hello", "world", "!"));
4 // 添加元素到 ArrayList 中
5 stringList.add("goodbye");
6 System.out.println(stringList); // [hello, world, !, goodbye]
7 // 修改 ArrayList 中的元素
8 stringList.set(0, "hi");
9 System.out.println(stringList); // [hi, world, !, goodbye]
10 // 删除 ArrayList 中的元素
11 stringList.remove(0);
   System.out.println(stringList); // [world, !, goodbye]
```

What is the difference between ArrayList and Vector?

(Just understand)

- ArrayList It is List the main implementation class of , uses Object[] storage at the bottom layer, is suitable for frequent search work, and is not thread-safe.
- Vector It is List an old implementation class of , which uses Object[] storage at the bottom layer and is thread-safe.

What is the difference between Vector and Stack? (Just understand)

- Vector Both Stack are thread-safe and use synchronized the keyword for synchronization.
- Stack Inherited from Vector , is a last-in-first-out stack, while Vector is a list.

With the development of Java concurrent programming, Vector and Stack have been eliminated. It is recommended to use concurrent collection classes (such as ConcurrentHashMap , CopyOnWriteArrayList etc.) or manually implement thread-safe methods to provide safe multi-threaded operation support.



Can null values be added to ArrayList?

`ArrayList` You can store any type of object in a , including `null` values. However, `ArrayList` adding `null` values to a is not recommended `null` ; it's meaningless and makes the code difficult to maintain. For example, forgetting to check for nulls can result in a `NullPointerException`.

Sample code:

```
1 ArrayList<String> listOfStrings = new ArrayList<>();  
2 listOfStrings.add(null);  
3 listOfStrings.add("java");  
4 System.out.println(listOfStrings);
```

java

Output:

```
1 [null, java]
```

plain

★ What is the time complexity of inserting and deleting elements in ArrayList?

For insert:

- Head insertion: Since all elements need to be moved back one position in sequence, the time complexity is $O(n)$.
- Tail insertion: When `ArrayList` the capacity of the list has not reached its limit, the time complexity of inserting an element to the end of the list is $O(1)$, because it only requires adding one element to the end of the array; when the capacity has reached its limit and needs to be expanded, it is necessary to perform an $O(n)$ operation to copy the original array to a new larger array, and then perform an $O(1)$ operation to add the element.
- Insert at a specified position: All elements after the target position need to be moved back one position, and then the new element is inserted into the specified position. This process requires moving an average of $n/2$ elements, so the time complexity is $O(n)$.

For deletion:



- Head deletion: Since all elements need to be moved forward one position in sequence, the time complexity is $O(n)$.
- Tail deletion: When the deleted element is at the end of the list, the time complexity is $O(1)$.
- Deletion at a specified position: All elements after the target element need to be moved forward one position to fill the deleted blank position. Therefore, an average of $n/2$ elements need to be moved, and the time complexity is $O(n)$.

Here is a simple example:

```

1 // ArrayList的底层数组大小为10, 此时存储了7个元素
2 +---+---+---+---+---+---+---+---+---+---+
3 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |   |   |   |
4 +---+---+---+---+---+---+---+---+---+---+
5     0   1   2   3   4   5   6   7   8   9
6 // 在索引为1的位置插入一个元素8, 该元素后面的所有元素都要向右移动一位
7 +---+---+---+---+---+---+---+---+---+---+
8 | 1 | 8 | 2 | 3 | 4 | 5 | 6 | 7 |   |   |
9 +---+---+---+---+---+---+---+---+---+---+
10    0   1   2   3   4   5   6   7   8   9
11 // 删除索引为1的位置的元素, 该元素后面的所有元素都要向左移动一位
12 +---+---+---+---+---+---+---+---+---+---+
13 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |   |   |   |
14 +---+---+---+---+---+---+---+---+---+---+
15    0   1   2   3   4   5   6   7   8   9

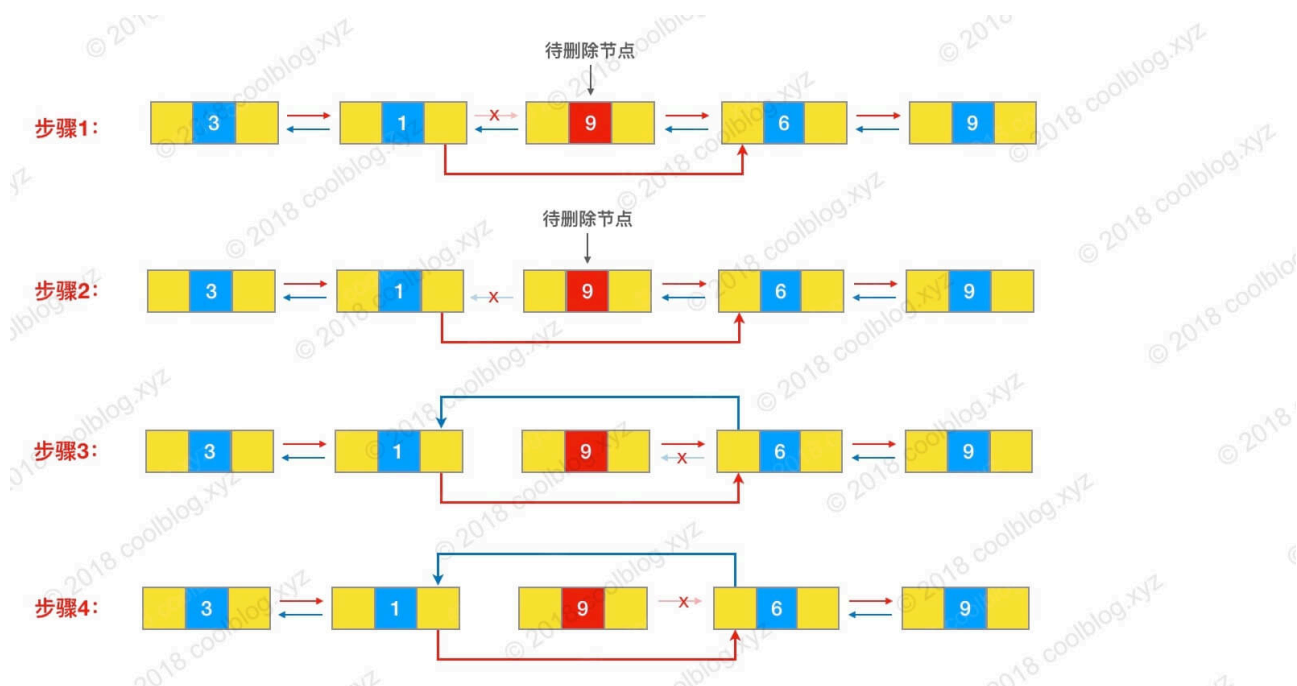
```

★ What is the time complexity of inserting and deleting elements in LinkedList?

- Head insertion/deletion: You only need to modify the pointer of the head node to complete the insertion/deletion operation, so the time complexity is $O(1)$.
- Tail insertion/deletion: The insertion/deletion operation can be completed by simply modifying the pointer of the tail node, so the time complexity is $O(1)$.
- Insertion/deletion at a specified position: You need to move to the specified position first, and then modify the pointer of the specified node to complete the insertion/deletion. However, since there are head and tail pointers, you can start from the closer pointer, so you need to traverse an average of $n/4$ elements, and the time complexity is $O(n)$.



Here's a simple example: if we want to delete node 9, we need to first traverse the linked list to find the node. Then, we need to modify the pointer to the corresponding node. For the specific source code, please refer to: [LinkedList Source Code Analysis](#) .



Why can't LinkedList implement the RandomAccess interface?

RandomAccess This is a marker interface used to indicate that the class implementing this interface supports random access (i.e., elements can be quickly accessed by index). Since **LinkedList** the underlying data structure is a linked list, the memory addresses are discontinuous and can only be located by pointers, which does not support random and fast access, so **RandomAccess** the interface cannot be implemented.

★ What is the difference between ArrayList and LinkedList?

- **Is thread safety guaranteed:** **ArrayList** Both **LinkedList** are not synchronized, that is, thread safety is not guaranteed;
- **Underlying data structure:** **ArrayList** The underlying data structure is **Object an array**; **LinkedList** the underlying data structure is **a doubly linked list** (before JDK 1.6, it was a circular linked list, and JDK 1.7 canceled the loop. Note the difference between a doubly linked list and a doubly circular linked list, which is introduced below!)
- **Whether insertion and deletion are affected by element position:**

- **ArrayList** Because array storage is used, the time complexity of inserting and deleting elements is affected by the element's position. For example, `add(E e)` when executing a method, `ArrayList` the specified element is appended to the end of the list by default, resulting in a time complexity of $O(1)$. However, if you insert or delete an element at a specific position `i` (`add(int index, E element)`), the time complexity becomes $O(n)$. This is because the elements after the i -th element in the collection must be shifted forward or backward during these operations.
- **LinkedList** It uses linked list storage, so inserting or deleting elements at the head or tail is not affected by the element position (,,,) `add(E e)`, `addFirst(E e)` and the time complexity is $O(1)$. If you want to insert and delete elements at a specified position (,,), the time complexity is $O(n)$, because you need to move to the specified position first and then insert and delete. `addLast(E e)` `removeFirst()` `removeLast()` `i add(int index, E element)` `remove(Object o)` `remove(int index)`
- **Support for fast random access:** `LinkedList` Efficient random element access is not supported, but `ArrayList` (implementing `RandomAccess` the interface) is supported. Fast random access is to quickly obtain the element object by the element's sequence number (corresponding to `get(int index)` the method).
- **Memory space usage:** `ArrayList` The space waste is mainly reflected in the fact that a certain amount of capacity space will be reserved at the end of the list, while the space cost of `LinkedList` is reflected in the fact that each of its elements requires more space than `ArrayList` (because it needs to store the direct successor and direct predecessor as well as data).

We generally don't use in our projects `LinkedList` . `LinkedList` In almost all scenarios where we need to use , we can use `ArrayList` instead, and the performance is usually better! Even `LinkedList` Josh Bloch, the author of , says he never uses it `LinkedList` .

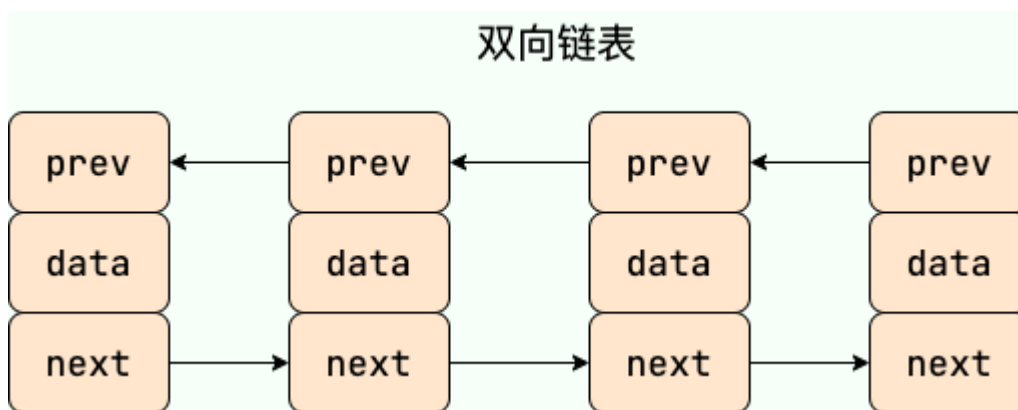




Also, don't automatically assume `LinkedList` that linked lists are best suited for adding and deleting elements. As I mentioned above, `LinkedList` the time complexity is approximately $O(1)$ only when inserting or deleting elements at the head or tail. Otherwise, the average time complexity for adding and deleting elements is $O(n)$.

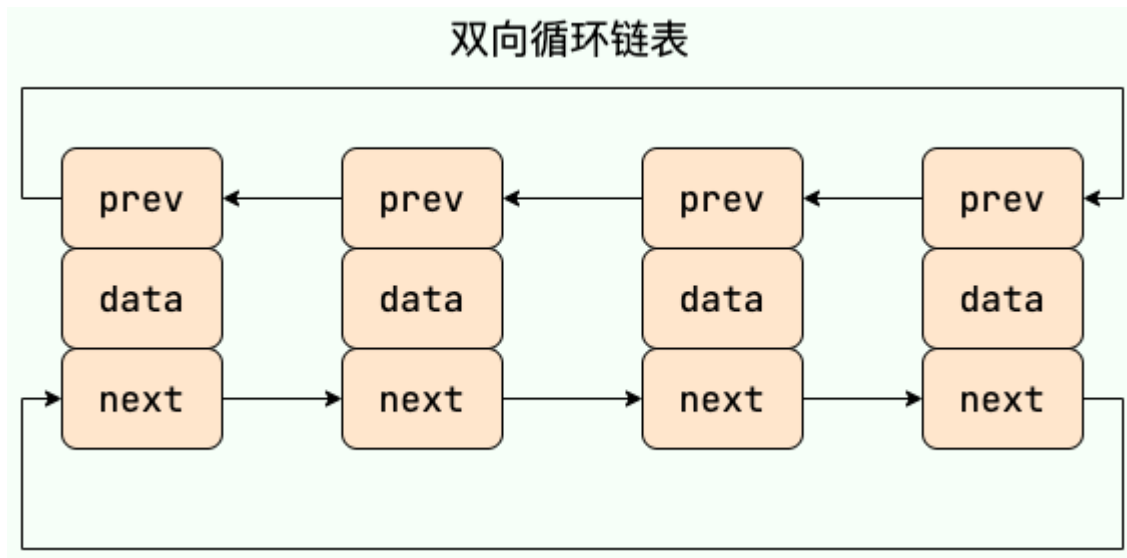
Supplementary content: Doubly linked lists and doubly circular linked lists

Doubly linked list: contains two pointers, a `prev` pointing to the previous node and a `next` pointing to the next node.



Bidirectional circular linked list: The `next` of the last node points to the head, and the `prev` of the head points to the last node, forming a ring.





Additional content: RandomAccess interface

```

1  public interface RandomAccess {
2  }

```

java

Looking at the source code, we see that `RandomAccess` the interface actually defines nothing. So, in my opinion, `RandomAccess` the interface is just a marker. What does it mark? It indicates that the class implementing this interface has random access capabilities.


In `binarySearch()` the method, it determines whether the passed list `RandomAccess` is an instance of `RandomAccess`. If so, call `indexedBinarySearch()` the method. If not, call `iteratorBinarySearch()` the method

```

1  public static <T>
2  int binarySearch(List<? extends Comparable<? super T>> list, T
3  key) {
4      if (list instanceof RandomAccess || list.size()
5  < BINARYSEARCH_THRESHOLD)
6          return Collections.indexedBinarySearch(list, key);
7      else
          return Collections.iteratorBinarySearch(list, key);
8  }

```

java

`ArrayList` The interface is implemented `RandomAccess`, but `LinkedList` is not.  think it has something to do with the underlying data structure! `ArrayList` The underlying structure is an array, while `LinkedList` the underlying structure is a linked list. Arrays naturally support random access, with a time complexity of $O(1)$, so it is called

fast random access. Linked lists require traversing to a specific position to access the element at that position, with a time complexity of $O(n)$, so they do not support fast random access. `ArrayList` Implementing `RandomAccess` the interface indicates that it has fast random access capabilities. `RandomAccess` The interface is just a marker; it does not mean that only when the interface `ArrayList` is implemented `RandomAccess` can it have fast random access capabilities!

★ Let's talk about the expansion mechanism of `ArrayList`

For details, please see this article by the author: [Analysis of ArrayList expansion mechanism](#) .

★ What are fail-fast and fail-safe in collections?

Regarding fail-fast the quote medium from an article about fail-fast and fail-safe :

Fail-fast systems are designed to immediately stop functioning upon encountering an unexpected condition. This immediate failure helps to catch errors early, making debugging more straightforward.

The idea of fast failure is to indicate failures in advance and stop operations for possible exceptions, thereby reducing the risk of cascading failure systems by discovering and stopping errors as early as possible.

Most collections under `java.util` the package do not support thread safety. In order to detect thread safety risks caused by concurrent operations in advance, it is proposed to maintain a `modCount` record of the number of modifications. During iteration, the expected number of modifications is compared with the expected number of modifications `expectedModCount` to `modCount` determine whether there is concurrent operation, thereby achieving fast failure and avoiding the execution of unnecessary complex code in the event of an exception.

Correspondingly, we give the following example. We first insert 100 an operation element, a thread iterates the element, and a thread deletes the element. The final output is thrown as expected `ConcurrentModificationException` :



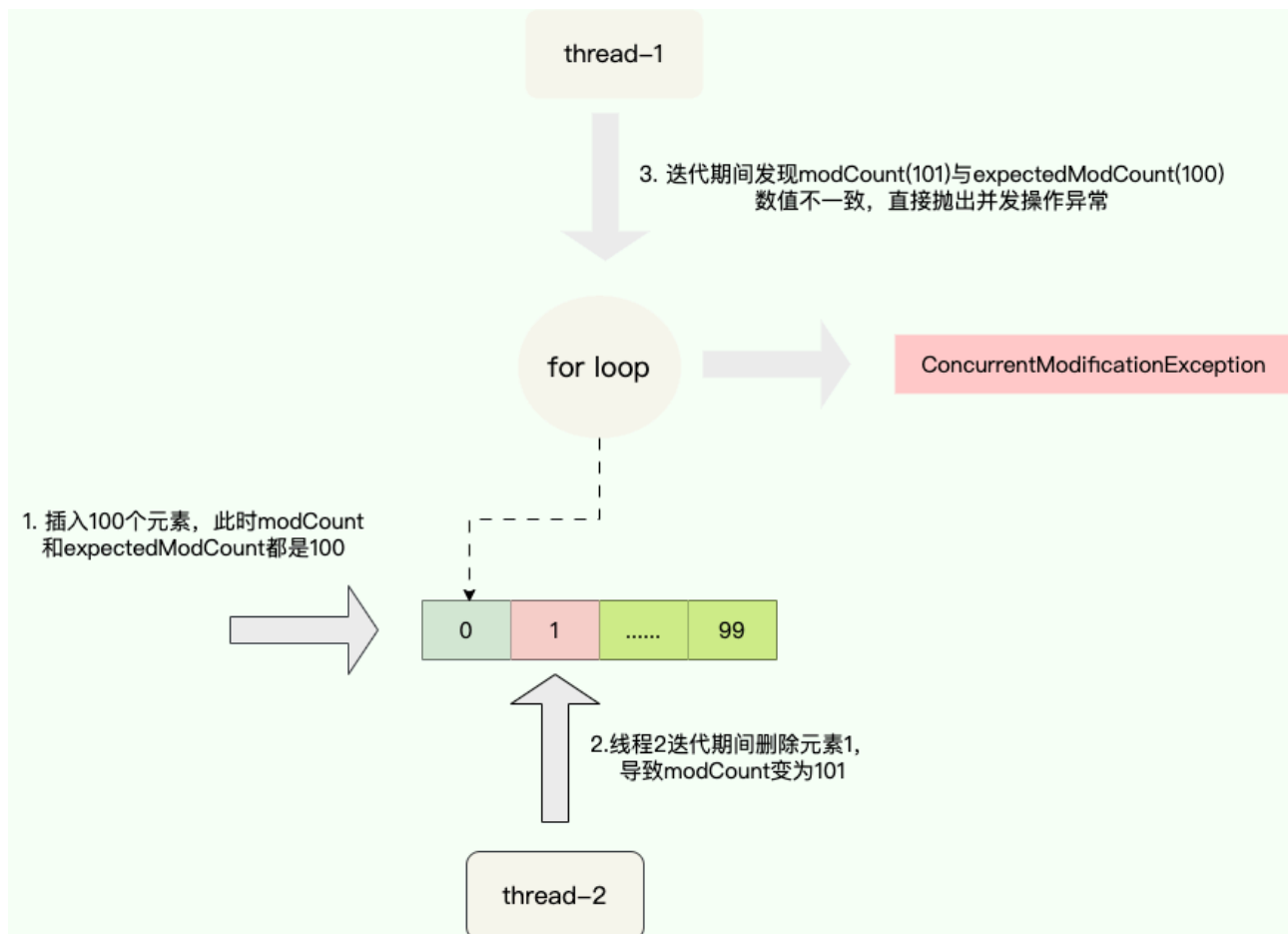
```
1 // 使用线程安全的 CopyOnWriteArrayList 避免 ConcurrentModificationException
2
3 List<Integer> list = new CopyOnWriteArrayList<>();
4 CountDownLatch countDownLatch = new CountDownLatch(2);
5
6 // 添加元素
7 for (int i = 0; i < 100; i++) {
8     list.add(i);
9 }
10
11 Thread t1 = new Thread(() -> {
12     // 迭代元素 (注意: Integer 是不可变的, 这里的 i++ 不会修改 list 中的值)
13     for (Integer i : list) {
14         i++; // 这行代码实际上没有修改list中的元素
15     }
16     countDownLatch.countDown();
17 });
18
19 Thread t2 = new Thread(() -> {
20     System.out.println("删除元素1");
21     list.remove(Integer.valueOf(1)); // 使用 Integer.valueOf(1) 删除
22     // 指定值的对象
23     countDownLatch.countDown();
24 });
25
26 t1.start();
27 t2.start();
28 countDownLatch.await();
```

We inserted 100 elements during initialization, and the corresponding modification modCount count is 100 . Thread 2 then deleted elements during the iteration of thread 1, and the corresponding count modCount became 101 .

Thread 1 found that was in the foreach second loop , which was not equal to the expected count. It was determined to be a concurrent operation exception, so it failed quickly and threw: modCount 101 expectedModCount(值为100因为初始化插入了元素100

个) ConcurrentModificationException





We also give a method `for` for looping the underlying iterator to get the next element `next`. You can see that it `checkForComodification` has logic for comparing the number of modifications:

```

1  public E next() {
2      //检查是否存在并发修改
3      checkForComodification();
4      //.....
5      //返回下一个元素
6      return (E) elementData[lastRet = i];
7  }
8
9  final void checkForComodification() {
10     //当前循环遍历次数和预期修改次数不一致时，就会抛出
    ConcurrentModificationException
11     if (modCount != expectedModCount)
12         throw new ConcurrentModificationException();
13 }

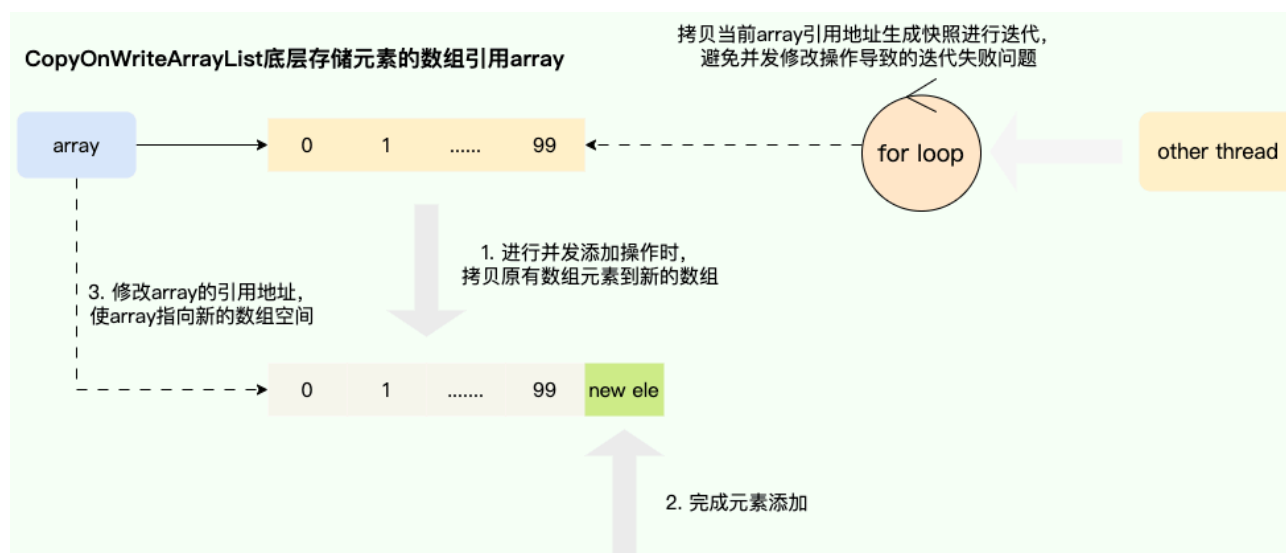
```

java

That fail-safe's what fail-safe means. It aims to recover and continue operating even in the face of unexpected situations, which makes it particularly suitable for uncertain or unstable environments:

Fail-safe systems take a different approach, aiming to recover and continue even in the face of unexpected conditions. This makes them particularly suited for uncertain or volatile environments.

This concept is often used in concurrent containers. The most classic implementation is `CopyOnWriteArrayList`. The copy-on-write concept ensures that a snapshot is copied when a modification operation is performed. After the addition or deletion operation is completed based on this snapshot, `CopyOnWriteArrayList` the underlying array reference is pointed to the new array space. This avoids concurrent operation security issues caused by interference from concurrent modifications during iteration. Of course, this approach also has its disadvantages, namely, it is impossible to obtain real-time results during traversal operations:



We also provide the core code of `CopyOnWriteArrayList` the implementation fail-safe. You can see that its implementation is to `getArray` obtain an array reference and then `Arrays.copyOf` obtain a snapshot of the array. After completing the addition operation based on this snapshot, `array` the reference address pointed to by the underlying variable is modified to complete the write-time copy:

```

1 public boolean add(E e) {
2     final ReentrantLock lock = this.lock;
3     lock.lock();
4     try {
5         //获取原有数组
        Object[] elements = getArray();
    
```

java


```
6         int len = elements.length;
7         //基于原有数组复制出一份内存快照
8         Object[] newElements = Arrays.copyOf(elements, len +
9         1);
10        //进行添加操作
11        newElements[len] = e;
12        //array指向新的数组
13        setArray(newElements);
14        return true;
15    } finally {
16        lock.unlock();
17    }
18 }
```

Set

The difference between Comparable and Comparator

Comparable The interfaces and **Comparator** are both interfaces used for sorting in Java. They play an important role in implementing size comparison and sorting between class objects:

- **Comparable** The interface actually comes from `java.lang` the package and it has a `compareTo(Object obj)` method for sorting
- **Comparator** The interface actually comes from `java.util` the package and it has a `compare(Object obj1, Object obj2)` method for sorting

Generally, when we need to use custom sorting on a collection, we need to override `compareTo()` a method or `compare()` methods. When we need to implement two sorting methods for a collection, such as song using different sorting methods for song titles and artist names in a collection, we can override `compareTo()` the method and use a custom **Comparator** method or use two **Comparator** methods to implement song title sorting and artist name sorting. The second method means that we can only use the two-parameter version `Collections.sort()`.

Comparator custom sorting




```
1  ArrayList<Integer> arrayList = new ArrayList<Integer>();
2  arrayList.add(-1);
3  arrayList.add(3);
4  arrayList.add(3);
5  arrayList.add(-5);
6  arrayList.add(7);
7  arrayList.add(4);
8  arrayList.add(-9);
9  arrayList.add(-7);
10 System.out.println("原始数组:");
11 System.out.println(arrayList);
12 // void reverse(List list): 反转
13 Collections.reverse(arrayList);
14 System.out.println("Collections.reverse(arrayList):");
15 System.out.println(arrayList);
16
17 // void sort(List list),按自然排序的升序排序
18 Collections.sort(arrayList);
19 System.out.println("Collections.sort(arrayList):");
20 System.out.println(arrayList);
21 // 定制排序的用法
22 Collections.sort(arrayList, new Comparator<Integer>() {
23     @Override
24     public int compare(Integer o1, Integer o2) {
25         return o2.compareTo(o1);
26     }
27 });
28 System.out.println("定制排序后: ");
29 System.out.println(arrayList);
```

java

Output:

```
1  原始数组:
2  [-1, 3, 3, -5, 7, 4, -9, -7]
3  Collections.reverse(arrayList):
4  [-7, -9, 4, 7, -5, 3, 3, -1]
5  Collections.sort(arrayList):
6  [-9, -7, -5, -1, 3, 3, 4, 7]
7  定制排序后:
8  [7, 4, 3, 3, -1, -5, -7, -9]
```

plain

Rewrite the compareTo method to sort by age

```
1 // person对象没有实现Comparable接口，所以必须实现，这样才不会出错，才可以使java
2 treemap中的数据按顺序排列
3 // 前面一个例子的String类已经默认实现了Comparable接口，详细可以查看String类
4 的API文档，另外其他
5 // 像Integer类等都已经实现了Comparable接口，所以不需要另外实现了
6 public class Person implements Comparable<Person> {
7     private String name;
8     private int age;
9
10    public Person(String name, int age) {
11        super();
12        this.name = name;
13        this.age = age;
14    }
15
16    public String getName() {
17        return name;
18    }
19
20    public void setName(String name) {
21        this.name = name;
22    }
23
24    public int getAge() {
25        return age;
26    }
27
28    public void setAge(int age) {
29        this.age = age;
30    }
31
32    /**
33     * 重写compareTo方法实现按年龄来排序
34     */
35    @Override
36    public int compareTo(Person o) {
37        if (this.age > o.getAge()) {
38            return 1;
```



```

39         }
40         if (this.age < o.getAge()) {
41             return -1;
42         }
43         return 0;
    }
}

```

```

1      public static void main(String[] args) {
2          TreeMap<Person, String> pdata = new TreeMap<Person, String>
3      ();
4          pdata.put(new Person("张三", 30), "zhangsan");
5          pdata.put(new Person("李四", 20), "lisi");
6          pdata.put(new Person("王五", 10), "wangwu");
7          pdata.put(new Person("小红", 5), "xiaohong");
8          // 得到key的值的的同时得到key所对应的值
9          Set<Person> keys = pdata.keySet();
10         for (Person key : keys) {
11             System.out.println(key.getAge() + "-" + key.getName());
12         }
13     }
}

```

Output:

```

1      5-小红
2      10-王五
3      20-李四
4      30-张三

```

What do disorder and non-repeatability mean?

- Disorder does not mean randomness. Disorder means that the stored data in the underlying array is not added in the order of array index, but is determined by the hash value of the data.
- Non-repeatability means that when the added element `equals()` is judged according to , it returns false, and you need to rewrite both `equals()` the method and `hashCode()` the method.



Compare the similarities and differences between HashSet, LinkedHashSet and TreeSet

- HashSet , LinkedHashSet and TreeSet are all Set implementation classes of the interface, which can ensure element uniqueness and are not thread-safe.
- HashSet The main differences between , LinkedHashSet and TreeSet lie in their underlying data structures. HashSet The underlying data structure of is a hash table (HashMap implemented based on). LinkedHashSet The underlying data structures of are linked lists and hash tables, and the insertion and removal order of elements satisfies FIFO. TreeSet The underlying data structure is a red-black tree, and the elements are ordered. Sorting methods include natural sorting and custom sorting.
- Different underlying data structures lead to different application scenarios for these three methods: HashSet they are used in scenarios where the order of element insertion and retrieval does not need to be guaranteed, LinkedHashSet they are used in scenarios where the order of element insertion and retrieval must meet FIFO, and TreeSet they are used in scenarios where custom sorting rules for elements are supported.

Queue

The difference between Queue and Deque

Queue It is a single-ended queue. Elements can only be inserted from one end and deleted from the other end. The implementation generally follows **the first-in-first-out (FIFO)** rule.

Queue The interface of is extended `Collection` . According to **the different processing methods after the operation fails due to capacity problems**, it can be divided into two categories: one will throw an exception after the operation fails, and the other will return a special value.

| Queue interface | throw an exception | Returning a special value |
|--------------------------------|-----------------------|---------------------------|
| Insert at the end of the queue | <code>add(E e)</code> | <code>offer(E e)</code> |



| Queue interface | throw an exception | Returning a special value |
|--------------------------------------|--------------------|---------------------------|
| Delete the leader | remove() | poll() |
| Query the first element of the queue | element() | peek() |

Deque It is a double-ended queue where elements can be inserted or deleted at both ends of the queue.

Deque The interface of is extended **Queue** to add methods for inserting and deleting at the head and tail of the queue. It is also divided into two categories according to the different post-failure processing methods:

| Deque interface | throw an exception | Returning a special value |
|--------------------------------------|--------------------|---------------------------|
| Insert to the head of the team | addFirst(E e) | offerFirst(E e) |
| Insert at the end of the queue | addLast(E e) | offerLast(E e) |
| Delete the leader | removeFirst() | pollFirst() |
| Delete the end of the queue | removeLast() | pollLast() |
| Query the first element of the queue | getFirst() | peekFirst() |
| Query the tail element | getLast() | peekLast() |

In fact, **Deque** there are other methods `push()` such as `pop()` and that can be used to simulate a stack.

Differences between ArrayDeque and LinkedList

ArrayDeque Both **LinkedList** implement **Deque** the interface and both have queue functionality, but what is the difference between them?

- **ArrayDeque** is implemented based on variable-length arrays and double pointers, **LinkedList** is implemented through linked lists.
- **ArrayDeque** Storing `NULL` data is not supported, but **LinkedList** is supported.



- **ArrayDeque** It was introduced in JDK1.6, but **LinkedList** it already existed in JDK1.2.
- **ArrayDeque** There may be an expansion process during insertion, but the insertion operation after amortization is still $O(1)$. Although **LinkedList** there is no need for expansion, new heap space must be allocated each time data is inserted, and the amortized performance is slower than before.

From a performance perspective, choosing **ArrayDeque** to implement a queue is **LinkedList** better than . In addition, **ArrayDeque** it can also be used to implement a stack.

Talk about PriorityQueue

PriorityQueue It was introduced in JDK1.5. **Queue** The difference between it and is that the order of elements being dequeued is related to the priority, that is, the element with the highest priority is always dequeued first.

Here are some key points:

- **PriorityQueue** It is implemented using a binary heap data structure, and the underlying variable-length array is used to store data.
- **PriorityQueue** By floating up and down the heap elements, it is possible to insert elements and delete the top elements of the heap within the time complexity of $O(\log n)$.
- **PriorityQueue** It is not thread-safe and does not support storing **NULL** and non-comparable objects.
- **PriorityQueue** The default is a mini-heap, but it can accept a **Comparator** construction parameter to define the priority of the elements.

PriorityQueue In the interview, you may be more likely to encounter hand-tearing algorithms. Typical examples include heap sort, finding the K-th largest number, traversing weighted graphs, etc., so you need to be proficient in using them.

What is BlockingQueue?

BlockingQueue (**Blocking Queue**) is an interface inherited from **Queue** .

BlockingQueue The reason for blocking is that it supports blocking when there are no elements in the queue until there are elements; it also supports waiting until the queue can put new elements if the queue is full.

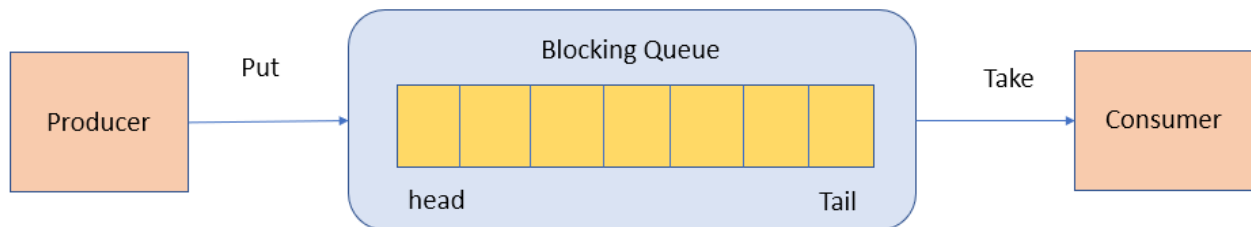


```

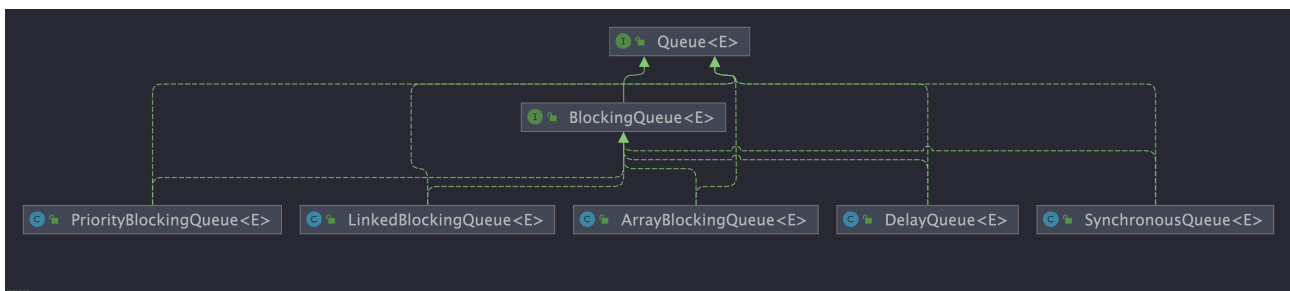
1 public interface BlockingQueue<E> extends Queue<E> {
2     // ...
3 }

```

BlockingQueue Commonly used in the producer-consumer model, the producer thread adds data to the queue, and the consumer thread takes data from the queue for processing.



What are the implementation classes of BlockingQueue?



The commonly used blocking queue implementation classes in Java are as follows:

1. **ArrayBlockingQueue** A bounded blocking queue implemented using an array. It requires a specified capacity when creating it and supports both fair and unfair lock access mechanisms.
2. **LinkedBlockingQueue** : An optionally bounded blocking queue implemented using a singly linked list. The capacity can be specified when creating it; if not, it defaults to `Integer.MAX_VALUE`. Unlike `ArrayBlockingQueue`, it only supports unfair lock access mechanisms.
3. **PriorityBlockingQueue** : An unbounded blocking queue that supports priority sorting. Elements must implement `Comparable` the interface or pass objects in the constructor `Comparator`, and null elements cannot be inserted.
4. **SynchronousQueue** A synchronous queue is a blocking queue that does not store elements. Each insertion operation must wait for the corresponding deletion operation and vice versa. Therefore, `SynchronousQueue` it is often used to transfer data directly between threads.

5. DelayQueue : Delay queue, in which elements can only be dequeued after their specified delay time.
6. ...

In daily development, these queues are not used much, so just understand them.

★ What is the difference between ArrayBlockingQueue and LinkedBlockingQueue?

ArrayBlockingQueue and LinkedBlockingQueue are two commonly used blocking queue implementations in the Java concurrency package. Both are thread-safe. However, there are some differences between them:

- The underlying implementation ArrayBlockingQueue is based on array implementation and LinkedBlockingQueue based on linked list implementation.
- Bounded: ArrayBlockingQueue A bounded queue must have a capacity specified when it is created. LinkedBlockingQueue It is possible to create a queue without specifying a capacity, Integer.MAX_VALUE which defaults to unbounded. However, you can also specify a queue size, making it bounded.
- Whether the lock is separated: ArrayBlockingQueue the lock in is not separated, that is, the production and consumption use the same lock; LinkedBlockingQueue the lock in is separated, that is, the production uses putLock and the consumption uses takeLock , which can prevent lock contention between producer and consumer threads.
- Memory usage: ArrayBlockingQueue Array memory needs to be allocated in advance, LinkedBlockingQueue while linked list node memory is dynamically allocated. This means that ArrayBlockingQueue a certain amount of memory is occupied when creating an array, and the memory requested is often larger than the actual memory used. LinkedBlockingQueue However, memory is gradually occupied as the number of elements increases.



JavaGuide官方公众号

(微信搜索JavaGuide)



- 1、公众号后台回复“**PDF**”获取原创PDF面试手册
- 2、公众号后台回复“**学习路线**”获取Java学习路线最新版
- 3、公众号后台回复“**开源**”获取优质Java开源项目合集
- 4、公众号后台回复“**八股文**”获取Java面试真题+面经

Recently Updated 2025/7/27 10:31

Contributors: guide , samho2008 , Jarvan-Song , Guide , Mr.Hope , paigeman , FengHaoJ , TedLau , chaobk , shark-chili , Xiaoxin

Copyright © 2025 Guide

