实验二十　ArkTS 容器类库

一、实验目的

1. 了解 DevEco Studio 的使用
2. 学习 ArkTS 语言及容器类库
3. 编写代码
4. 编译运行
5. 在模拟器上运行

二、实验原理

1. 鸿蒙开发原理
2. ArkTS，ArkUI 开发原理
3. 鸿蒙应用运行原理

三、实验仪器材料

1. 计算机实训室电脑一台
2. DevEco Studio 开发环境及鸿蒙手机模拟器

四、实验步骤

容器类库官方资料 https://developer.huawei.com/consumer/cn/doc/harmonyos-guides/container-overview

容器类库，用于存储各种数据类型的元素，并具备一系列处理数据元素的方法，作为**纯数据结构容器来使用**具有一定的优势。

容器类采用了类似静态语言的方式来实现，并通过对存储位置以及属性的限制，让每种类型的数据都能在完成自身功能的基础上去除冗余逻辑，保证了数据的高效访问，提升了应用的性能。

当前提供了**线性**和**非线性**两类容器。**线性类容器底层通过数组实现，非线性类容器底层通过 hash 或者红黑树实现**。线性容器和非线性容器都**不是多线程安全**的。

- 线性容器

线性容器实现能按顺序访问的数据结构，其底层主要通过数组实现，包括 **ArrayList、Vector、List、LinkedList、Deque、Queue、Stack** 七种。

线性容器，充分考虑了数据访问的速度，运行时（Runtime）通过一条字节码指令就可以完成增、删、改、查等操作。

- 非线性容器

非线性容器实现能快速查找的数据结构，其底层通过 hash 或者红黑树实现，包括 **HashMap、HashSet、TreeMap、TreeSet、LightWeightMap、LightWeightSet、PlainArray** 七种。非线性容器中的 key 及 value 的类型均满足 ECMA 标准。

我们主要通过实例来学习。

1. 打开 DevEco Studio，点击 Create Project 创建工程

设置项目名称为 ContainerDemo。

2. 线性容器

- ArrayList

ArrayList 是一种线性数据结构，底层基于数组实现。ArrayList 会根据实际需要动态调整容量，**每次扩容增加 50%**。

ArrayList 和 Vector 相似，都是基于数组实现。它们都可以动态调整容量，但 **Vector 每次扩容增加 1 倍**。

ArrayList 和 LinkedList 相比，ArrayList 的随机访问效率更高。但由于 ArrayList 的增删操作会影响数组内其他元素的移动，LinkedList 的增加和删除操作效率更高。

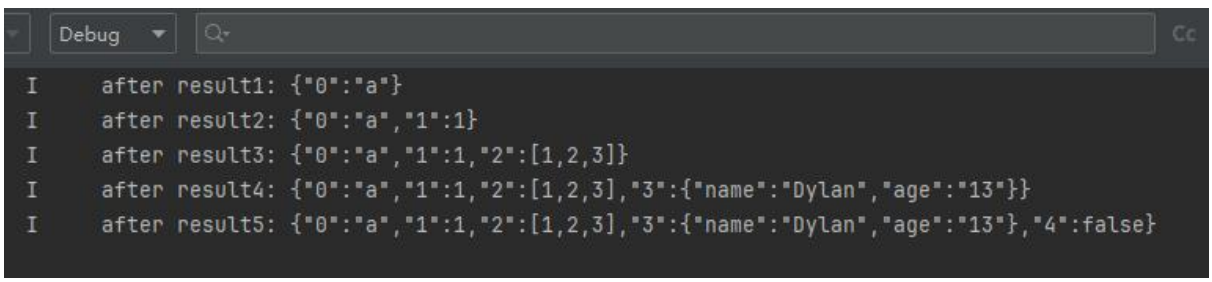
推荐使用场景： **当需要频繁读取集合中的元素时，推荐使用 ArrayList**。

创建一个新的文件 ArrayListDemo.ets，创建基本的页面 Component，然后在@Entry 前面加入代码：

```
import { ArrayList } from '@kit.ArkTS';


class C1 {
```

```
  name: string = ""
  age: string = ""
}

let arrayList: ArrayList<string | number | boolean | Array<number> | C1> = new ArrayList();
let result1 = arrayList.add("a");
console.log("after result1:", JSON.stringify(arrayList));
let result2 = arrayList.add(1);
console.log("after result2:", JSON.stringify(arrayList));
let b = [1, 2, 3];
let result3 = arrayList.add(b);
console.log("after result3:", JSON.stringify(arrayList));
let c : C1 = {name: "Dylan", age: "13"}
let result4 = arrayList.add(c);
console.log("after result4:", JSON.stringify(arrayList));
let result5 = arrayList.add(false);
console.log("after result5:", JSON.stringify(arrayList));
```

刷新 Previewer，查看 Log：



首先是需要导入 ArrayList，然后创建一个新的 ArrayList，在这个数组中可以存放的类型可以是多种的，包括自己定义的类 C1。通过 add 函数添加。查看结果可以看到，JSON.stringify 输出的实际上是一个 KV 键值对，包括位置和值。

再添加代码，理解 insert 函数的使用：

```
arrayList.insert("A", 0);

console.log("after insert:", JSON.stringify(arrayList));

arrayList.insert(0, 1);

console.log("after insert:", JSON.stringify(arrayList));

arrayList.insert(true, 2);

console.log("after insert:", JSON.stringify(arrayList));
```

结果：

```
I     after insert: {"0":"A","1":"a","2":1,"3":[1,2,3],"4":{"name":"Dylan","age":"13"},"5":false}
I     after insert: {"0":"A","1":0,"2":"a","3":1,"4":[1,2,3],"5":{"name":"Dylan","age":"13"},"6":false}
I     after insert: {"0":"A","1":0,"2":true,"3":"a","4":1,"5":[1,2,3],"6":{"name":"Dylan","age":"13"},"7":false}
```

添加代码，看一下 remove 的效果：

```
let res1: boolean = arrayList.remove(b); //前面 b=[1,2,3]

console.log("after remove:", JSON.stringify(arrayList));

let res2: boolean = arrayList.remove("a");

console.log("after remove:", JSON.stringify(arrayList));
```

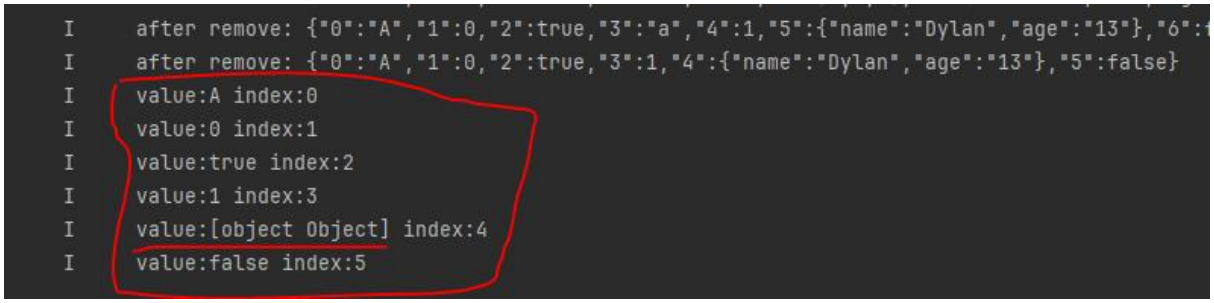
结果：

```
I     after insert: {"0":"A","1":0,"2":"a","3":1,"4":[1,2,3],"5":{"name":"Dylan","age":"13"},"6":false}
I     after insert: {"0":"A","1":0,"2":true,"3":"a","4":1,"5":[1,2,3],"6":{"name":"Dylan","age":"13"},"7":false}
I     after remove: {"0":"A","1":0,"2":true,"3":"a","4":1,"5":{"name":"Dylan","age":"13"},"6":false}
I     after remove: {"0":"A","1":0,"2":true,"3":1,"4":{"name":"Dylan","age":"13"},"5":false}
```

添加代码，看一下 forEach 的作用：

```
arrayList.forEach((value: string | number | boolean | Array<number> | C1, index?: number)
=> {
  console.log("value:" + value, "index:" + index);
});
```

结果：

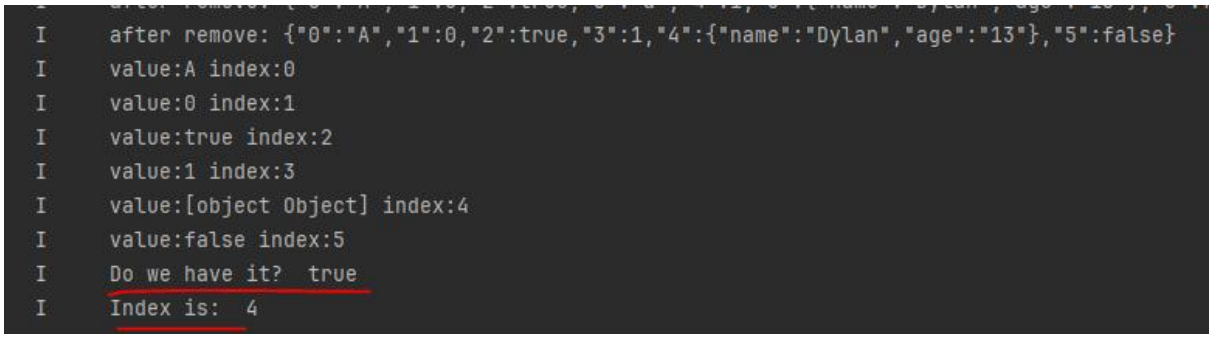


思考：怎么显示 object?

再看一下函数 has 和 getIndexOf 的作用：

```
let res3: boolean = arrayList.has("A");
console.log("Do we have it? ", res3);
let res4: number = arrayList.getIndexOf(c);
console.log("Index is: ", res4);
```

结果：



添加代码，看 replaceAllElements 的作用：

```
let arrayList1: ArrayList<number>= new ArrayList();

arrayList1.add(2);

arrayList1.add(4);

arrayList1.add(5);

arrayList1.add(4);

console.log("before replace:", JSON.stringify(arrayList1));


arrayList1.replaceAllElements((value: number, index: number)=> {

  return value = 2 * value;

});

console.log("after replace:", JSON.stringify(arrayList1));


arrayList1.replaceAllElements((value: number, index: number) => {

  return value = value - 2;

});

console.log("after replace:", JSON.stringify(arrayList1));
```

结果：

```
I     before replace: {"0":2,"1":4,"2":5,"3":4}
I     after replace: {"0":4,"1":8,"2":10,"3":8}
I     after replace: {"0":2,"1":6,"2":8,"3":6}
```

还有其他的一些函数，可以自己在这里继续探索：

https://developer.huawei.com/consumer/cn/doc/harmonyos-references-V5/js-apis-arraylist-V5#replaceallelements

- Vector

- List

List 可用来构造一个**单向链表**对象，**即只能通过头结点开始访问到尾节点**。List 依据泛型定义，在**内存中的存储位置可以是不连续**的。List 底层通过单向链表实现，每个节点有一个指向后一个元素的引用。当需要查询元素时，必须从头遍历，插入、删除效率高，查询效率低。**List 允许元素为 null**。

List 和 LinkedList 相比，**LinkedList 是双向链表**，可以快速地在头尾进行增删，而 List 是单向链表，无法双向操作。

当**需要频繁的插入删除元素**，并且需要使用单向链表时，推荐使用 List 高效操作。

创建一个新的文件 ListDemo.ets，创建基本的页面 Component，然后在@Entry 前面加入代码：

```
import { List } from '@kit.ArkTS';

let list: List<string | number | boolean | object> = new List();

let result1 = list.add("a");

console.log("after result1:", JSON.stringify(list));

let result2 = list.add(1);

console.log("after result2:", JSON.stringify(list));

let b = [1, 2, 3];

let result3 = list.add(b);

console.log("after result3:", JSON.stringify(list));
```

```
class C {
  name: string = ''
  age: string = ''
}
let c: C = {name : "Dylan", age : "13"};
let result4 = list.add(c);
console.log("after result4:", JSON.stringify(list));
let result5 = list.add(false);
console.log("after result5:", JSON.stringify(list));
```

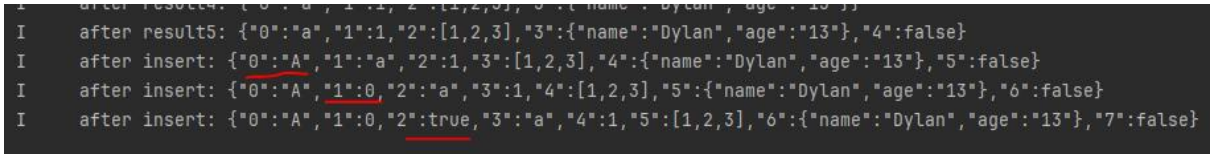
结果（和 ArrayList 很类似）：

```
after result1: {"0":"a"}
after result2: {"0":"a","1":1}
after result3: {"0":"a","1":1,"2":[1,2,3]}
after result4: {"0":"a","1":1,"2":[1,2,3],"3":{"name":"Dylan","age":"13"}}
after result5: {"0":"a","1":1,"2":[1,2,3],"3":{"name":"Dylan","age":"13"},"4":false}
```

类似地，尝试 insert:

```
list.insert("A", 0);
console.log("after insert:", JSON.stringify(list));
list.insert(0, 1);
console.log("after insert:", JSON.stringify(list));
list.insert(true, 2);
console.log("after insert:", JSON.stringify(list));
```

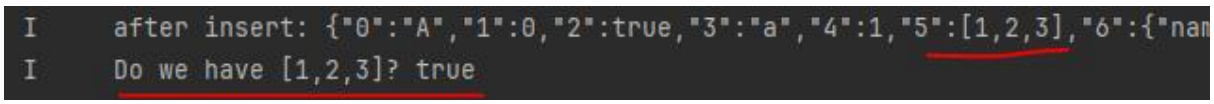
结果：



尝试 has:

```
let res1 = list.has(b);

console.log("Do we have [1,2,3]?", res1);
```

结果：

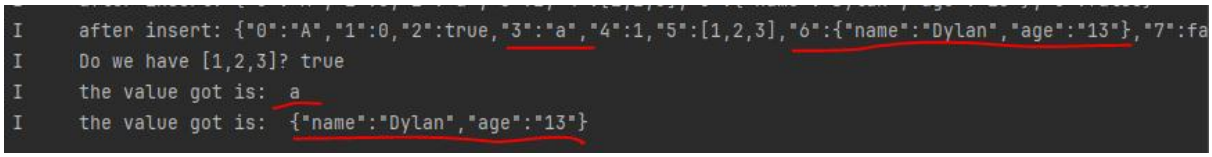


尝试 get，注意：

> **注意**
> 在List中使用[index]的方式虽然能够获取对应位置的元素，但这会导致未定义结果。推荐使用get()方法。

```
let res2 = list.get(3);

console.log("the value got is: ", res2);

let res3 = list.get(6);

console.log("the value got is: ", JSON.stringify(res3));
```

结果：



类似地，也有 remove, forEach，replaceAllElements 等函数，请到这里查看：

https://developer.huawei.com/consumer/cn/doc/harmonyos-references-V5/js-apis-list-V5

- LinkedList

LinkedList **底层通过双向链表实现**，双向链表的每个节点都包含对前一个元素和后一个元素的引用。当需要查询元素时，可以从头遍历，也可以从尾部遍历，**插入、删除效率高**，查询效率低。LinkedList 允许元素为 null。

LinkedList 和 List 相比，LinkedList 是双向链表，可以快速地在头尾进行增删，而 List 是单向链表，无法双向操作。

LinkedList 和 ArrayList 相比，插入数据效率 LinkedList 优于 ArrayList，而查询效率 ArrayList 优于 LinkedList。

可以查看这里自己尝试实例：

https://developer.huawei.com/consumer/cn/doc/harmonyos-references-V5/js-apis-linkedlist-V5

- Deque

官方文档：https://developer.huawei.com/consumer/cn/doc/harmonyos-references-V5/js-apis-deque-V5

Deque（double ended queue）根据**循环队列的数据结构**实现，符合**先进先出以及先进后出**的特点，支持两端的元素插入和移除。Deque 会根据实际需要动态调整容量，每次进行两倍扩容。

Deque 和 Queue 相比，Queue 的特点是先进先出，只能在头部删除元素，尾部增加元素。

与 Vector 相比，它们都支持在两端增删元素，但 Deque 不能进行中间插入的操作。对头部元素的插入删除效率高于 Vector，而 Vector 访问元素的效率高于 Deque。

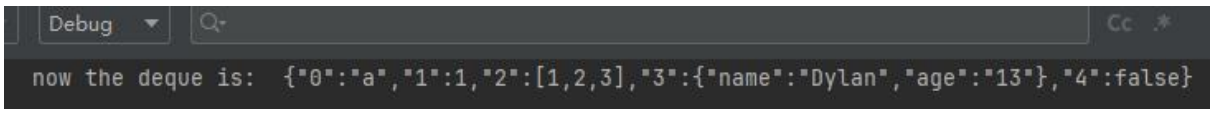
推荐使用场景： 需要频繁在集合两端进行增删元素的操作时，推荐使用 Deque。

创建一个新的文件 DequeDemo.ets，创建基本的页面 Component，然后在@Entry 前面加入代码：

```
import { Deque } from '@kit.ArkTS';


class C1 {
  name: string = ""
  age: string = ""
}
```

```
let deque: Deque<string | number | boolean | Array<number> | C1> = new Deque();

deque.insertEnd("a");

deque.insertEnd(1);

let b = [1, 2, 3];

deque.insertEnd(b);

let c: C1 = {name : "Dylan", age : "13"};

deque.insertEnd(c);

deque.insertEnd(false);


console.log("now the deque is: ", JSON.stringify(deque));
```

结果：



可以看到，没有“add”函数，而是 insertEnd 函数。华为官方文档：

https://developer.huawei.com/consumer/cn/doc/harmonyos-references-V5/js-apis-deque-V5

可以看到：

提供的 API 不多。再尝试 insertFront，forEach 和 popFirst:

```
let deque1: Deque<number> = new Deque();

deque1.insertFront(1);

deque1.insertFront(2);

deque1.insertEnd(5);

deque1.insertFront(3);

deque1.insertFront(4);

console.log("now the deque1 is: ", JSON.stringify(deque1));

deque1.forEach((value: number, index?: number | undefined, deque?: Deque<number> |
undefined):void => {
  console.log("value:" + value, "index:" + index);
});

let result = deque1.popFirst();

console.log("result is: ", result);
```

结果：

```
I     now the deque1 is:  {"0":5,"4":4,"5":3,"6":2,"7":1}
I     value:4 index:0
I     value:3 index:1
I     value:2 index:2
I     value:1 index:3
I     value:5 index:4
I     result is:  4
```

注意，用 console.log 打印出来的结果不正确，所以我们用 forEach 循环打印，结果符合预期。popFirst 返回的是第一个元素。

试一下 popLast():

```
let result2 = deque1.popLast();

console.log("result2 is: ", result2);
```

结果：

```
I     value:4 index:0
I     value:3 index:1
I     value:2 index:2
I     value:1 index:3
I     value:5 index:4
I     result is:  4
I     result2 is:  5
```

- Queue

Queue 的特点是**先进先出，在尾部增加元素，在头部删除元素**。根据循环队列的数据结构实现。

Queue 和 Deque 相比，Queue 只能在一端删除一端增加，Deque 可以两端增删。

推荐使用场景： 一般符合先进先出的场景可以使用 Queue。

华为官方文档：

https://developer.huawei.com/consumer/cn/doc/harmonyos-references-V5/js-apis-queue-V5

可以看到提供的 API 更少，因为只能在尾部加，头部删除。

可以自己尝试文档中的实例。

- Stack

Stack 可用来构造栈对象，存储元素遵循**先进后出**的规则。

Stack 依据泛型定义，要求存储位置是一片连续的内存空间，初始容量大小为 8，并支持动态扩容，

每次扩容大小为原始容量的 1.5 倍。**Stack 底层基于数组实现，入栈出栈均从数组的一端操作。**

Stack 和 Queue 相比，Queue 基于循环队列实现，只能在一端删除，另一端插入，而 Stack 都在一端操作。

一般**符合先进后出的场景**可以使用 Stack。

华为官方文档：

https://developer.huawei.com/consumer/cn/doc/harmonyos-references-V5/js-apis-stack-V5

提供的 API 符合一个栈的逻辑。注意：

- peek(): T 获取并返回栈顶元素。
- locate: 返回指定元素第一次出现时的下标值，查找失败返回-1。
- isEmpty: 判断该栈是否为空。

可以自行尝试文档中的实例。

3. 非线性容器

- HashMap

华为官方文档：https://developer.huawei.com/consumer/cn/doc/harmonyos-references/js-apis-hashmap

HashMap 可用来**存储具有关联关系的 key-value 键值对集合**，存储元素中 **key 是唯一**的，每个 key 会对应一个 value 值。

HashMap 依据泛型定义，集合中通过 key 的 hash 值确定其存储位置，从而快速找到键值对。HashMap

的初始容量大小为 16，并支持动态扩容，每次扩容大小为原始容量的 2 倍。**HashMap 底层基于 HashTable 实现，冲突策略采用链地址法**。

HashMap 和 TreeMap 相比，HashMap 依据键的 hashCode 存取数据，访问速度较快。而 TreeMap 是有序存取，效率较低。

需要快速存取、删除以及插入键值对（KV Pair）数据时，推荐使用 HashMap。

创建一个新的文件 HashMapDemo.ets，创建基本的页面 Component，然后在@Entry 前面加入代码：

```
import { HashMap } from '@kit.ArkTS';


const hashMap: HashMap<string, number> = new HashMap();
let result = hashMap.isEmpty();
console.log("is it empty: ", result);


hashMap.set("squirrel", 123);
let result1 = hashMap.hasValue(123);
let result2 = hashMap.hasKey("squirrel");
console.log("do we have the value: ", result1);
console.log("do we have the key: ", result2);
```

结果：

```
I     is it empty:  true
I     do we have the value:  true
I     do we have the key:  true
```

这里用到了：

- set: 添加 key-value 键值对

- hasKey: 判断是否有键 key

- hasValue: 判断是否有值 value

- isEmpty: 判断这个 HashMap 是否为空

再添加代码看 get 函数：

```
hashMap.set("sparrow", 356);

let result3 = hashMap.get("sparrow");

console.log("sparrow's value is: ", result3);
```

结果：

```
I     sparrow's value is:  356
```

再添加代码看 forEach 和 setAll（将一个 HashMap 中的所有元素组添加到另一个 hashMap 中）

```
console.log("before setAll:")

hashMap.forEach((value?: number, key?: string) => {

  console.log("value:" + value, "key:" + key);

});

let newHashMap: HashMap<string, number> = new HashMap();

newHashMap.set("newMap", 99);

hashMap.setAll(newHashMap);

console.log("after setAll:")

hashMap.forEach((value?: number, key?: string) => {

  console.log("value:" + value, "key:" + key);

});
```

结果：

```
before setAll:
value:123 key:squirrel
value:356 key:sparrow
after setAll:
value:123 key:squirrel
value:356 key:sparrow
value:99 key:newMap
```

再添加代码看 replace:

```
console.log("after replace:")

let result4 = hashMap.replace("sparrow", 357);

hashMap.forEach((value?: number, key?: string) => {

  console.log("value:" + value, "key:" + key);

});
```

结果：

```
after setAll:
value:123 key:squirrel
value:356 key:sparrow
value:99 key:newMap
after replace:
value:123 key:squirrel
value:357 key:sparrow
value:99 key:newMap
```

HashSet 可用来存储一系列值的集合，存储元素中 **value 是唯一**的。

HashSet 依据泛型定义，集合中通过 value 的 hash 值确定其存储位置，从而快速找到该值。HashSet 初始容量大小为 16，支持动态扩容，每次扩容大小为原始容量的 2 倍。value 的类型满足 ECMA 标准中要求的类型。**HashSet 基于 HashMap 实现，只对 value 对象进行处理**。底层数据结构与 HashMap 一致。

HashSet 和 TreeSet 相比，**HashSet 中的数据无序存放，即不能由用户指定排序方式**，而 TreeSet 是有序存放，能够依照用户给定的排序函数对元素进行排序。它们集合中的元素都不允许重复，但 HashSet 允许放入 null 值，TreeSet 不建议存放 null 值，可能会对排序结果产生影响。

可以利用 **HashSet 不重复**的特性，当需要不重复的集合或需要去重某个集合的时候使用。

创建一个新的文件 HashSetDemo.ets，创建基本的页面 Component，然后在@Entry 前面加入代码：

```
import { HashSet } from '@kit.ArkTS';

let hashSet: HashSet<number> = new HashSet();
hashSet.add(1);
hashSet.add(2);
hashSet.add(3);
hashSet.add(4);
hashSet.add(5);
let res = hashSet.length;
console.log("res:", res);

hashSet.forEach((value?: number, key?: number): void => {
  console.log("value:" + value, "key:" + key);
});

let hashSet1: HashSet<string> = new HashSet();
hashSet1.add("sparrow");
```

```
hashSet1.add("squirrel");
hashSet1.forEach((value?: string, key?: string): void => {
  console.log("value:" + value, "key:" + key);
});
```

结果：

```
res: 5
value:1 key:1
value:2 key:2
value:3 key:3
value:4 key:4
value:5 key:5
value:squirrel key:squirrel
value:sparrow key:sparrow
```

因为 value 是唯一的，key 直接和 value 一样？

再添加一条语句：

```
hashSet.add(4);
hashSet.add(5);
hashSet.add(5);
let res = hashSet.lengt
```

也就是 add(5)执行两次，可以发现：

```
res: 5
value:1 key:1
value:2 key:2
value:3 key:3
value:4 key:4
value:5 key:5
value:squirrel key:squirrel
value:sparrow key:sparrow
```

结果依然是一样的。所以如果添加了相同的值，只保留一个。

其他的如 remove， clear 等函数可以自行尝试。

- TreeMap

TreeMap 可用来存储具有关联关系的 key-value 键值对集合，存储元素中 key 是唯一的，每个 key 会对应一个 value 值。

TreeMap 依据泛型定义，集合中的 **key 值是有序**的，TreeMap 的**底层是一棵二叉树**，可以通过树的二叉查找快速的找到键值对。key 的类型满足 ECMA 标准中要求的类型。**TreeMap 中的键值是有序存储的**。TreeMap 底层基于红黑树实现，可以进行快速的插入和删除。

TreeMap 和 HashMap 相比，HashMap 依据键的 hashCode 存取数据，访问速度较快。而 TreeMap 是有序存取，效率较低。

一般需要存储有序键值对的场景，可以使用 TreeMap。

创建一个新的文件 TreeMapDemo.ets，创建基本的页面 Component，然后在@Entry 前面加入代码：

```
import { TreeMap } from '@kit.ArkTS';


//使用 comparator firstValue < secondValue，表示期望结果为升序排序。反之 firstValue > secondValue，表示为降序排序。
let treeMap : TreeMap<string,string> = new TreeMap<string,string>((firstValue: string, secondValue: string) : boolean => {return firstValue > secondValue});
treeMap.set("aa","3");
treeMap.set("dd","1");
treeMap.set("cc","2");
treeMap.set("bb","4");
let numbers = Array.from(treeMap.keys())
for (let item of numbers) {
```

```
    console.log("treeMap:" + item);
}
```

注意，顺序是需要你自己去定义的。上面的例子中用的是降序，所以结果：

```
treeMap:dd
treeMap:cc
treeMap:bb
treeMap:aa
```

如果我们把代码改为：

```
let treeMap : TreeMap<string,string> = new TreeMap<string,string>((firstValue: string,
secondValue: string) : boolean => {return firstValue < secondValue});
```

结果：

```
treeMap:aa
treeMap:bb
treeMap:cc
treeMap:dd
```

对于自定义的类型：

```
//当插入自定义类型时，则必须要提供比较函数。
class TestEntry{
    public id: number = 0;
    public name: string = '';
}
let ts1: TreeMap<TestEntry, string> = new TreeMap<TestEntry, string>((t1: TestEntry, t2:
TestEntry): boolean => {return t1.id < t2.id;});
let entry1: TestEntry = {
```

```
  id: 0,
  name: 'zhangshan'
};
let entry2: TestEntry = {
  id: 1,
  name: 'lisi'
}
ts1.set(entry1, "0");
ts1.set(entry2, "1");
console.log("treeMap: ", ts1.length);
let objects = Array.from(ts1.keys())
for (let item of objects) {
  console.log("treeMap:" + JSON.stringify(item));
}
```

结果：

```
treeMap:  2
treeMap:{"id":0,"name":"zhangshan"}
treeMap:{"id":1,"name":"lisi"}
```

其他的函数如 get， isEmpty， getKey， getValue 等和 HashMap 类似，可以自行尝试。

- TreeSet

TreeSet 可用来存储一系列值的集合，**存储元素中 value 是唯一的**。

TreeSet 依据泛型定义，集合中的 value 值是**有序的**，TreeSet 的底层是一棵二叉树，可以通过树的

二叉查找快速的找到该 value 值，value 的类型满足 ECMA 标准中要求的类型。TreeSet 中的值是有序存储的。TreeSet 底层基于红黑树实现，可以进行快速的插入和删除。

TreeSet 基于 TreeMap 实现，在 TreeSet 中，只对 value 对象进行处理。TreeSet 可用于存储一系列值的集合，元素中 value 唯一，且能够依照用户给定的排序函数对元素进行排序。

TreeSet 和 HashSet 相比，HashSet 中的数据无序存放，而 TreeSet 是有序存放。它们集合中的元素都不允许重复，但 **HashSet 允许放入 null 值，TreeSet 不建议存放 null 值，可能会对排序结果产生影响**。

一般需要**存储有序集合**的场景，可以使用 TreeSet。

文档：

https://developer.huawei.com/consumer/cn/doc/harmonyos-references-V5/js-apis-treeset-V5

请自己尝试里面的例子。

- LightWeightMap

LightWeightMap 可用来存储具有关联关系的 key-value 键值对集合，存储元素中 key 是唯一的，每个 key 会对应一个 value 值。LightWeightMap 依据泛型定义，采用更加轻量级的结构，**底层标识唯一 key 通过 hash 实现，其冲突策略为线性探测法**。集合中的 key 值的查找依赖于 hash 值以及二分查找算法，通过一个数组存储 hash 值，然后映射到其他数组中的 key 值以及 value 值，key 的类型满足 ECMA 标准中要求的类型。

初始默认容量大小为 8，每次扩容大小为原始容量的 2 倍。

LightWeightMap 和 HashMap 都是用来存储键值对的集合，**LightWeightMap 占用内存更小**。

当需要存取 key-value 键值对时，推荐使用占用内存更小的 LightWeightMap。

查看文档：

https://developer.huawei.com/consumer/cn/doc/harmonyos-references-V5/js-apis-lightweightmap-V5

用法与 HashMap 比较类似。请自行尝试。

- LightWeightSet

LightWeightSet 可用来存储一系列值的集合，**存储元素中 value 是唯一的**。

LightWeightSet 依据泛型定义，采用更加轻量级的结构，初始默认容量大小为 8，每次扩容大小为原始容量的 2 倍。集合中的 value 值的查找依赖于 hash 以及二分查找算法，通过一个数组存储 hash 值，然后映射到其他数组中的 value 值，value 的类型满足 ECMA 标准中要求的类型。

LightWeightSet 底层标识唯一 value 基于 hash 实现，其冲突策略为线性探测法，查找策略基于二分查找法。

LightWeightSet 和 HashSet 都是用来存储键值的集合，LightWeightSet 的占用内存更小。

当需要存取某个集合或是对某个集合去重时，推荐使用占用内存更小的 LightWeightSet。

查看文档：

https://developer.huawei.com/consumer/cn/doc/harmonyos-references-V5/js-apis-lightweightset-V5

用法与 HashSet 比较类似。请自行尝试。

- PlainArray

PlainArray 可用来存储具有关联关系的键值对集合，**存储元素中 key 是唯一的，并且对于 PlainArray 来说，其 key 的类型为 number 类型**。每个 key 会对应一个 value 值，类型依据泛型的定义，PlainArray 采用更加轻量级的结构，集合中的 key 值的查找依赖于二分查找算法，然后映射

到其他数组中的 value 值。

初始默认容量大小为 16，每次扩容大小为原始容量的 2 倍。

PlainArray 和 LightWeightMap 都是用来存储键值对，且均采用轻量级结构，但 PlainArray 的 key 值类型只能为 number 类型。

当需要存储 key 值为 number 类型的键值对时，可以使用 PlainArray。

华为官方文档：

https://developer.huawei.com/consumer/cn/doc/harmonyos-guides/nonlinear-container#plainarray

创建一个新的文件 PlainArrayDemo.ets，创建基本的页面 Component，然后在@Entry 前面加入代码：

```
import { PlainArray } from '@kit.ArkTS';


const plainArray: PlainArray<string> = new PlainArray();
let result = plainArray.isEmpty();
console.log("is it empty: ", result);


plainArray.add(1, "squirrel");
let result1 = plainArray.has(1);
console.log("key 1 exist? ", result1);


plainArray.add(2, "sparrow");
let result2 = plainArray.get(2);
console.log("key 2 value ", result2);
```

```
let result3 = plainArray.getIndexOfKey(2);
console.log("key 2 index ", result3);

let result4 = plainArray.getIndexOfValue("squirrel");
console.log("index is: ", result4);

plainArray.forEach((value: string, index?: number) => {
  console.log("value:" + value, "index:" + index);
});
```

结果：

```
is it empty:  true
key 1 exist?  true
key 2 value  sparrow
key 2 index  1
index is:  0
value:squirrel index:1
value:sparrow index:2
```

基本上我们把 ArkTS 的容器类库中的线性容器和非线性容器给过了一遍，希望大家在看到类似的代码的时候能够看懂。至于在项目中实际使用，还需要自己多加练习，掌握好之后才能发挥这些容器的优势。

五、实验注意事项

1. 注意教师的操作演示。
2. 学生机与教师机内网连通，能接收和提交实验结果。
3. 按实验要求输入测试数据，并查看输出结果是否符合实验结果。

六、思考题

1.通过这个实验，你学到了什么？