泛型(Generics)

为了能够更好的学习容器，我们首先要先来学习一个概念：泛型。

泛型基本概念

泛型是JDK5.0以后增加的新特性。

泛型的本质就是“数据类型的参数化”，处理的数据类型不是固定的，而是可以作为参数传入。我们可以把“泛型”理解为数据类型的一个占位符(类似：形式参数)，即告诉编译器，在调用泛型时必须传入实际类型

> 参数化类型，白话说就是：
>
> 1. 把类型当作是参数一样传递。
> 2. <数据类型> 只能是引用类型。

泛型的好处

在不使用泛型的情况下，我们可以使用Object类型来实现任意的参数类型，但是在使用时需要我们强制进行类型转换。这就要求程序员明确知道实际类型，不然可能引起类型转换错误；但是，在编译期我们无法识别这种错误，只能在运行期发现这种错误。使用泛型的好处就是可以在编译期就识别出这种错误，有了更好的安全性；同时，所有类型转换由编译器完成，在程序员看来都是自动转换的，提高了代码的可读性。

总结一下，就是使用泛型主要是两个好处：

1. 代码可读性更好【不用强制转换】
2. 程序更加安全【只要编译时期没有警告，运行时期就不会出现ClassCastException异常】

类型擦除

编码时采用泛型写的类型参数，编译器会在编译时去掉，这称之为“类型擦除”。

泛型主要用于编译阶段，编译后生成的字节码class文件不包含泛型中的类型信息，涉及类型转换仍然是普通的强制类型转换。类型参数在编译后会被替换成Object，运行时虚拟机并不知道泛型。

泛型主要是方便了程序员的代码编写，以及更好的安全性检测。

实时效果反馈

1.泛型是在JDK哪个版本中增加的新特性?

A JDK5.0

B JDK6.0

C JDK7.0

D JDK8.0

2.泛型的本质是什么?

A 数据的参数化

B 对象的参数化

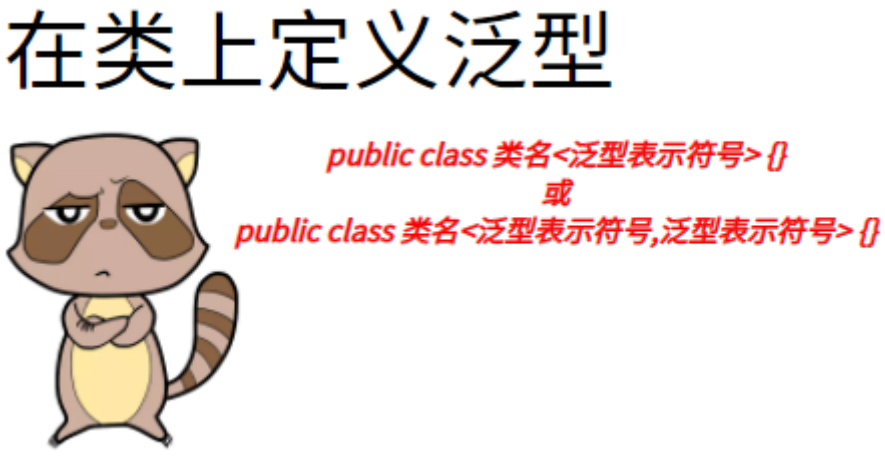
C 数据类型的参数化

D 基本类型的参数化

答案

1=>A　2=>C

泛型类



泛型标记

定义泛型时，一般采用几个标记：E、T、K、V、N、？。他们约定俗称的含义如下：

泛型标记	对应单词	说明
E	Element	在容器中使用，表示容器中的元素
T	Type	表示普通的JAVA类
K	Key	表示键，例如：Map中的键Key
V	Value	表示值
N	Number	表示数值类型
?		表示不确定的JAVA类型

泛型类的使用

语法结构

```
public class 类名<泛型标识符号> {
}

public class 类名<泛型标识符号,泛型标识符号> {
}
```

示例

```java
public class Generic<T> {
    private T  flag;

    public void setFlag(T flag){
        this.flag = flag;
    }

    public T getFlag(){
        return this.flag;
    }
}
```

```java
public class Test {
    public static void main(String[] args) {
        //创建对象时，指定泛型具体类型。
        Generic<String> generic = new Generic<>();
        generic.setFlag("admin");
        String flag = generic.getFlag();
        System.out.println(flag);

        //创建对象时，指定泛型具体类型。
        Generic<Integer> generic1 = new Generic<>();
        generic1.setFlag(100);
        Integer flag1 = generic1.getFlag();
        System.out.println(flag1);
    }
}
```

实时效果反馈

1.如下哪个选项是正确定义泛型类的语法

A public class <泛型标识符号> 类名

B public <泛型标识符号> class 类名

C <泛型标识符号> public class 类名

D public class 类名<泛型标识符号>

答案

1=>D

泛型接口

泛型接口和泛型类的声明方式一致。

泛型接口的使用

语法结构

```
public interface 接口名<泛型标识符号> {
}

public interface 接口名<泛型标识符号,泛型标识符号>
{
}
```

示例

```
public interface IGeneric<T> {
    T getName(T name);
}
```

```
//在实现接口时传递具体数据类型
public class IgenericImpl implements
Igeneric<String> {
    @Override
    public String getName(String name) {
        return name;
    }
}

//在实现接口时仍然使用泛型作为数据类型
public class IGenericImpl2<T> implements
IGeneric<T>{
    @Override
    public T getName(T name) {
        return name;
    }
```

```
15 }
```

```
1  public class Test {
2      public static void main(String[] args) {
3          IGeneric<String> igeneric= new
   IGenericImpl();
4          String name =
   igeneric.getName("oldlu");
5          System.out.println(name);
6
7          IGeneric<String> igeneric1 = new
   IGenericImpl2<>();
8          String name1 =
   igeneric1.getName("itbz");
9          System.out.println(name1);
10     }
11 }
```

实时效果反馈

1.如下哪个选项是正确定义泛型接口的语法

A public interface<泛型标识符号> 接口名

B public <泛型标识符号> interface 接口名

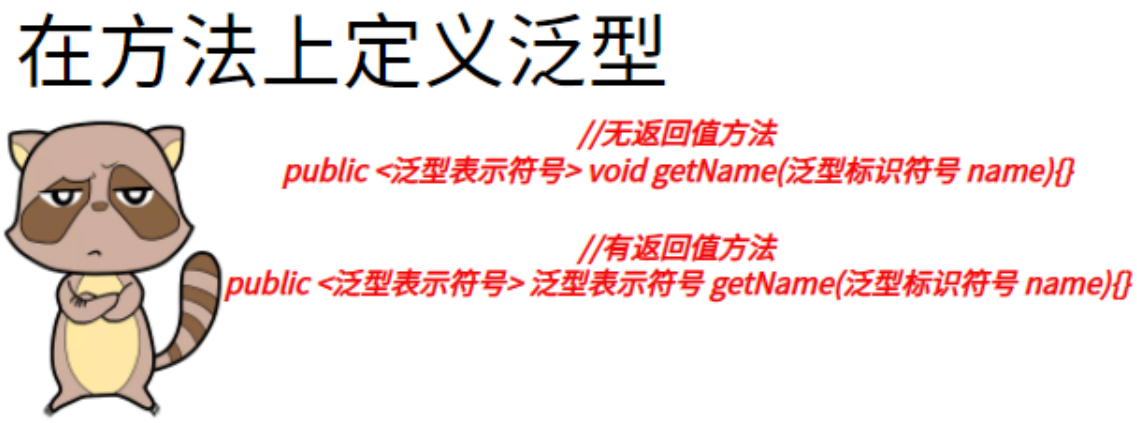
C <泛型标识符号> public interface 接口名

D public interface 接口名<泛型标识符号>

答案

1=>D

泛型方法



类上定义的泛型，在方法中也可以使用。但是，我们经常需要仅仅在某一个方法上使用泛型，这时候可以使用泛型方法。

调用泛型方法时，不需要像泛型类那样告诉编译器是什么类型，编译器可以自动推断出类型

泛型方法的使用

非静态方法

非静态方法可以使用泛型类中所定义的泛型，也可以将泛型定义在方法上。

语法结构

```
//无返回值方法
public <泛型标识符号> void getName(泛型标识符号 name){
}

//有返回值方法
public <泛型标识符号> 泛型标识符号 getName(泛型标识符号 name){
}
```

示例

```
public class MethodGeneric {
    public <T> void setName(T name){
        System.out.println(name);
    }
    public <T> T getAge(T age){
        return age;
    }
}
```

```
public class Test2 {
    public static void main(String[] args) {
        MethodGeneric methodGeneric = new
MethodGeneric();
        methodGeneric.setName("oldlu");
        Integer age =
methodGeneric.getAge(123);
        System.out.println(age);
    }
```

静态方法

静态方法中使用泛型时有一种情况需要注意一下，那就是静态方法无法访问类上定义的泛型，所以必须要将泛型定义在方法上。

语法结构

```
//无返回值静态方法
public static <泛型标识符号> void setName(泛型标识符号 name){
}

//有返回值静态方法
public static <泛型标识符号> 泛型表示符号 getName(泛型标识符号 name){
}
```

示例

```java
public class MethodGeneric {
    public static <T> void setFlag(T flag){
        System.out.println(flag);
    }

    public static <T> T getFlag(T flag){
        return flag;
    }
}
```

```java
public class Test4 {
    public static void main(String[] args) {
        MethodGeneric.setFlag("oldlu");
        Integer flag1 =
MethodGeneric.getFlag(123123);
        System.out.println(flag1);
    }
}
```

实时效果反馈

1.如下哪个选项是正确定义泛型方法的语法

A <泛型标识符号> public void getName(泛型标识符号 name)

B public void <泛型标识符号> getName(泛型标识符号 name)

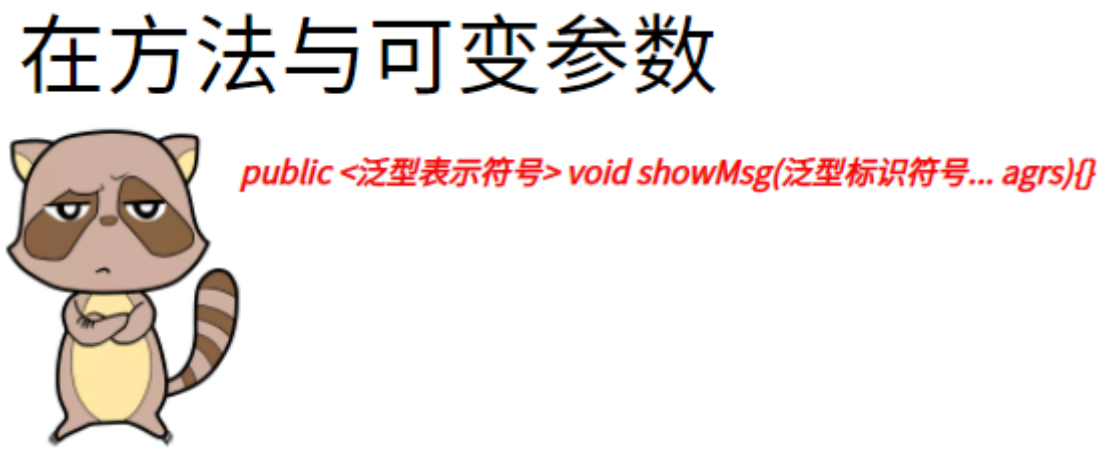
C public <泛型标识符号> void getName(泛型标识符号 name)

D public void getName <泛型标识符号>(泛型标识符号 name)

答案

1=>C

泛型方法与可变参数



在泛型方法中，泛型也可以定义可变参数类型。

语法结构

```
public <泛型标识符号> void showMsg(泛型标识符号... agrs){
}
```

示例

```
public class MethodGeneric {
    public <T> void method(T...args){
        for(T t:args){
            System.out.println(t);
        }
    }

}
```

```
public class Test5 {
    public static void main(String[] args) {
        MethodGeneric methodGeneric = new MethodGeneric();
        String[] arr = new String[]{"a","b","c"};
        Integer[] arr2 = new Integer[]{1,2,3};
        methodGeneric.method(arr);
        methodGeneric.method(arr2);
    }
}
```

实时效果反馈

1.如下哪个选项是正确的在可变参数中使用泛型

A public <泛型标识符号> void showMsg(泛型标识符号... agrs)

B public void showMsg(<泛型标识符号>... agrs)

C public <泛型标识符号> void showMsg(<泛型标识符号>... agrs)

D public <泛型标识符号> void showMsg(Object... agrs)

答案

1=>A

泛型中的通配符

无界通配符

“?”表示类型通配符，用于代替具体的类型。它只能在“<>”中使用。可以解决当具体类型不确定的问题。

语法结构

```
public void showFlag(Generic<?> generic){
}
```

示例

```
public class Generic<T> {
    private T  flag;

    public void setFlag(T flag){
        this.flag = flag;
    }

    public T getFlag(){
        return this.flag;
    }
}
```

```
public class ShowMsg {
    public void showFlag(Generic<?> generic){

 System.out.println(generic.getFlag());
    }
}
```

```
public class Test3 {
    public static void main(String[] args) {
        ShowMsg showMsg = new ShowMsg();
        Generic<Integer> generic = new
Generic<>();
        generic.setFlag(20);
        showMsg.showFlag(generic);

```

```
        Generic<Number> generic1 = new Generic<>();
        generic1.setFlag(50);
        showMsg.showFlag(generic1);

        Generic<String> generic2 = new Generic<>();
        generic2.setFlag("oldlu");
        showMsg.showFlag(generic2);
    }
}
```

实时效果反馈

1.在泛型中，无界通配符使用什么符号来表示?

A !

B ?

C #

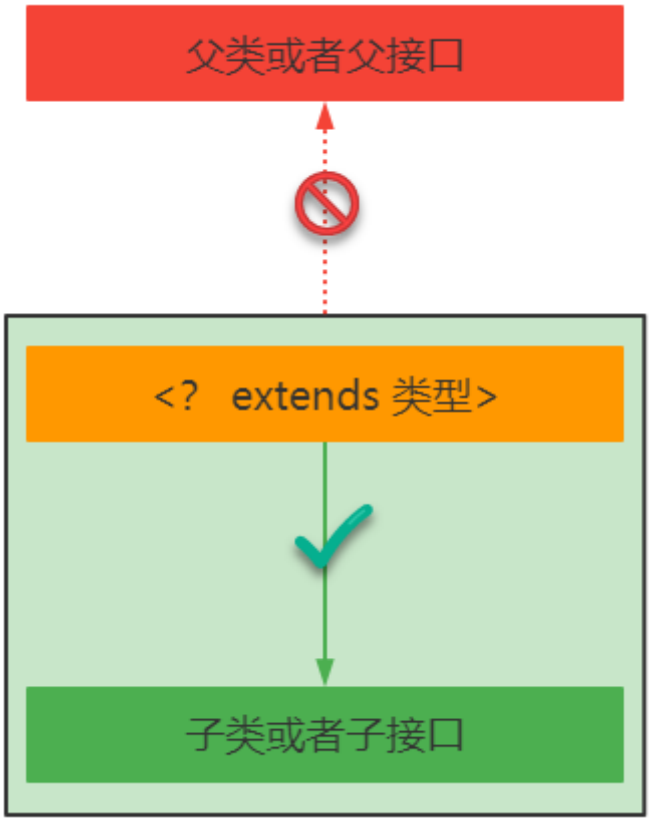
D *

答案

1=>B

统配符的上下限定

统配符的上限限定



对通配符的上限的限定：<? extends 类型>

？实际类型可以是上限限定中所约定的类型，也可以是约定类型的子类型；

语法结构

```
public void showFlag(Generic<? extends Number> generic){
}
```

示例

```
public class ShowMsg {
    public void showFlag(Generic<? extends Number> generic){

     System.out.println(generic.getFlag());
    }
}
```

```
public class Test4 {
    public static void main(String[] args) {
        ShowMsg showMsg = new ShowMsg();
        Generic<Integer> generic = new Generic<>();
        generic.setFlag(20);
        showMsg.showFlag(generic);

        Generic<Number> generic1 = new Generic<>();
        generic1.setFlag(50);
        showMsg.showFlag(generic1);
    }
}
```

实时效果反馈

1.对通配符的上限的限定是指

A 实际类型只能是上限限定中所约定的类型;

B 实际类型只能是上限限定中所约定类型的子类型;

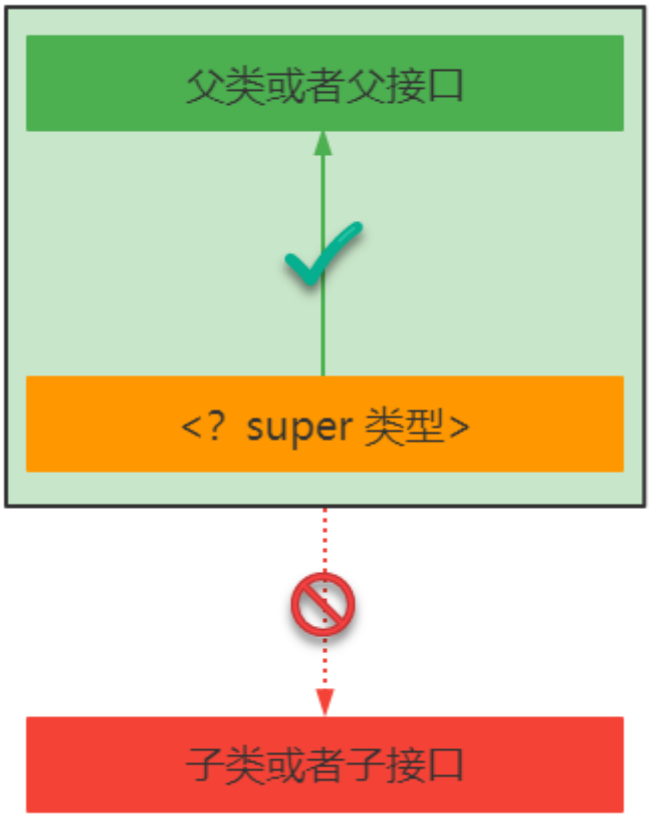
C 实际类型可以是上限限定中所约定的类型，也可以是约定类型的子类型；

D 实际类型可以是上限限定中所约定的类型，也可以是约定类型的父类型；

答案

1=>C

通配符的下限限定



对通配符的下限的限定：<? super 类型>

？实际类型可以是下限限定中所约定的类型，也可以是约定类型的父类型；

语法结构

```
public void showFlag(Generic<? super Integer> generic){
}
```

示例

```
public class ShowMsg {
    public void showFlag(Generic<? super Integer> generic){

 System.out.println(generic.getFlag());
    }
}
```

```
public class Test6 {
    public static void main(String[] args) {
        ShowMsg showMsg = new ShowMsg();
        Generic<Integer> generic = new
Generic<>();
        generic.setFlag(20);
        showMsg.showFlag(generic);

        Generic<Number> generic1 = new
Generic<>();
        generic1.setFlag(50);
        showMsg.showFlag(generic1);
    }
}
```

实时效果反馈

1.对通配符的下限的限定是指

A 实际类型只能是下限限定中所约定的类型；

B 实际类型只能是下限限定中所约定类型的子类型；

C 实际类型可以是下限限定中所约定的类型，也可以是约定类型的子类型；

D 实际类型可以是下限限定中所约定的类型，也可以是约定类型的父类型；

答案

1=>D

泛型局限性和常见错误

泛型主要用于编译阶段，编译后生成的字节码class文件不包含泛型中的类型信息。 类型参数在编译后会被替换成Object，运行时虚拟机并不知道泛型。因此，使用泛型时，如下几种情况是错误的：

1. 基本类型不能用于泛型

   `Test<int> t;` 这样写法是错误，我们可以使用对应的包装类

   `Test<Integer> t ;`

2. 不能通过类型参数创建对象

   `T elm = new T();` 运行时类型参数`T`会被替换成`Object`，无法创建T类型的对象，容易引起误解，java干脆禁止这种写法。

实时效果反馈

1.如下哪个选项是错误的使用泛型?

A Generic

B Generic

C Generic

D Generic

答案

1=>D

容器介绍

容器简介

容器，是用来容纳物体、管理物体。生活中,我们会用到各种各样的容器。如锅碗瓢盆、箱子和包等。

程序中的“容器”也有类似的功能，用来容纳和管理数据。比如，如下新闻网站的新闻列表、教育网站的课程列表就是用“容器”来管理：

视频课程信息也是使用“容器”来管理：

开发和学习中需要时刻和数据打交道，如何组织这些数据是我们编程中重要的内容。 我们一般通过“容器”来容纳和管理数据。事实上，我们前面所学的数组就是一种容器，可以在其中放置对象或基本类型数据。

数组的优势：是一种简单的线性序列，可以快速地访问数组元素，效率高。如果从查询效率和类型检查的角度讲，数组是最好的。

数组的劣势：不灵活。容量需要事先定义好，不能随着需求的变化而扩容。比如：我们在一个用户管理系统中，要把今天注册的所有用户取出来，那么这样的用户有多少个？我们在写程序时是无法确定的。因此，在这里就不能使用数组。

基于数组并不能满足我们对于“管理和组织数据的需求”，所以我们需要一种更强大、更灵活、容量随时可扩的容器来装载我们的对象。 这就是我们今天要学习的容器，也叫集合(Collection)。

实时效果反馈

1.Java中容器的作用是什么?

A 容纳数据

B 处理数据

C 生产数据

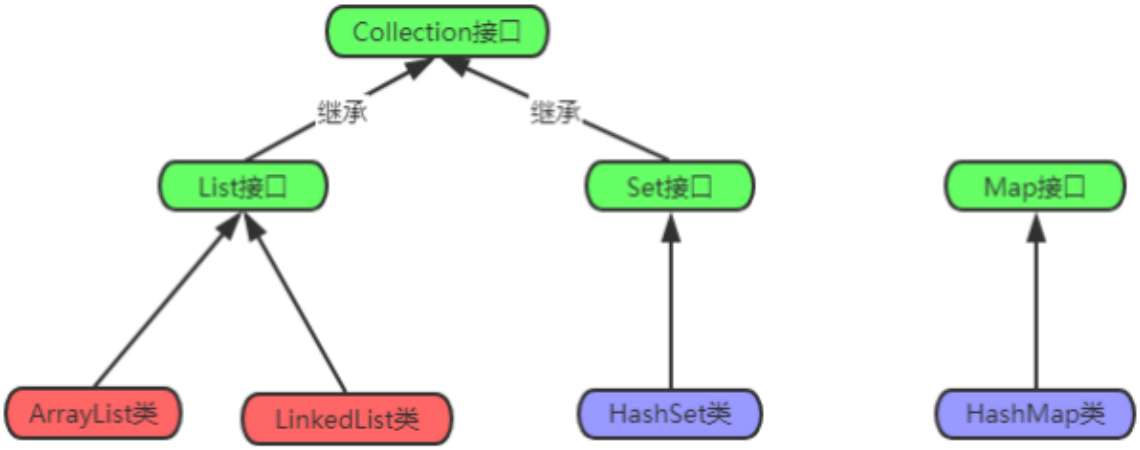
D 销毁数据

答案

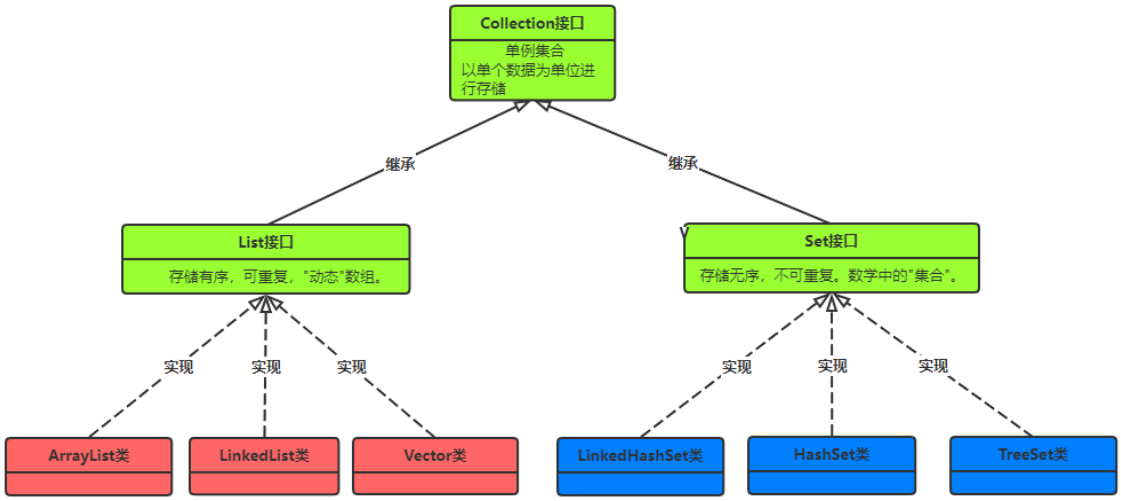
1=>A

容器的结构

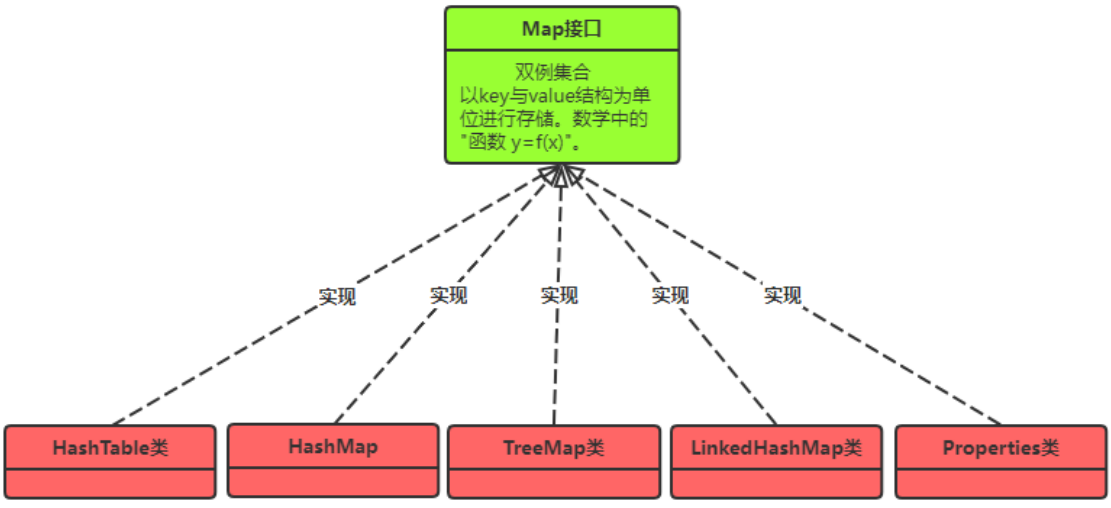
结构图


Collection接口
继承
继承
List接口
Set接口
Map接口
ArrayList类
LinkedList类
HashSet类
HashMap类

单例集合


Collection接口
单例集合
以单个数据为单位进行存储
继承
继承
List接口
存储有序，可重复，"动态"数组。
Set接口
存储无序，不可重复。数学中的"集合"。
实现
实现
实现
实现
实现
实现
ArrayList类
LinkedList类
Vector类
LinkedHashSet类
HashSet类
TreeSet类

双例集合


Map接口
双例集合
以key与value结构为单位进行存储。数学中的"函数 y=f(x)"。
实现
实现
实现
实现
实现
HashTable类
HashMap
TreeMap类
LinkedHashMap类
Properties类

实时效果反馈

1.如下哪个接口不是容器接口?

A List

B Set

C Map

D Comparable

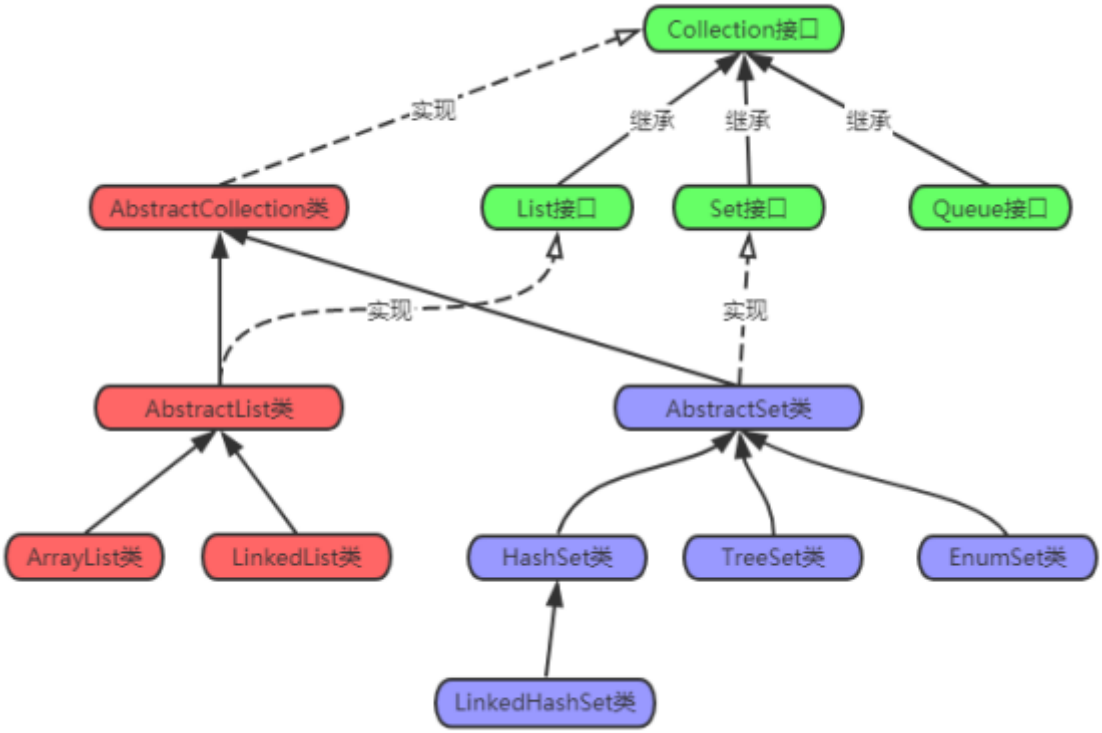
答案

1=>D

单例集合

Collection接口介绍

Collection 表示一组对象，它是集中、收集的意思。Collection接口的两个子接口是List、Set接口。



Collection接口中定义的方法

方法	说明
boolean add(Object element)	增加元素到容器中
boolean remove(Object element)	从容器中移除元素
boolean contains(Object element)	容器中是否包含该元素
int size()	容器中元素的数量
boolean isEmpty()	容器是否为空
void clear()	清空容器中所有元素
Iterator iterator()	获得迭代器，用于遍历所有元素
boolean containsAll(Collection c)	本容器是否包含c容器中的所有元素
boolean addAll(Collection c)	将容器c中所有元素增加到本容器
boolean removeAll(Collection c)	移除本容器和容器c中都包含的元素
boolean retainAll(Collection c)	取本容器和容器c中都包含的元素，移除非交集元素
Object[] toArray()	转化成Object数组

> 由于List、Set是Collection的子接口，意味着所有List、Set的实现类都有上面的方法。

JDK8之后，Collection接口新增的方法（将在JDK新特性和函数式编程中介绍）：

新增方法	说明
removeIf	作用是删除容器中所有满足filter指定条件的元素
stream parallelStream	stream和parallelStream 分别返回该容器的Stream视图表示，不同之处在于parallelStream()返回并行的Stream，Stream是Java函数式编程的核心类。
spliterator	可分割的迭代器，不同以往的iterator需要顺序迭代，Spliterator可以分割为若干个小的迭代器进行并行操作，可以实现多线程操作提高效率

实时效果反馈

1.如下哪个接口不是Collection接口的子接口?

A List

B Set

C Map

D Queue

答案

1=>C

List接口介绍

List接口特点

List是有序、可重复的容器。

有序：有序(元素存入集合的顺序和取出的顺序一致)。List中每个元素都有索引标记。可以根据元素的索引标记（在List中的位置）访问元素，从而精确控制这些元素。

可重复：List允许加入重复的元素。更确切地讲，List通常允许满足e1.equals(e2) 的元素重复加入容器。

List接口中的常用方法

除了Collection接口中的方法，List多了一些跟顺序(索引)有关的方法，参见下表：

方法	说明
void add (int index, Object element)	在指定位置插入元素，以前元素全部后移一位
Object set (int index,Object element)	修改指定位置的元素
Object get (int index)	返回指定位置的元素
Object remove (int index)	删除指定位置的元素，后面元素全部前移一位
int indexOf (Object o)	返回第一个匹配元素的索引，如果没有该元素，返回-1.
int lastIndexOf (Object o)	返回最后一个匹配元素的索引，如果没有该元素，返回-1

实时效果反馈

1.如下哪个选项是List接口的特点?

A 有序，可重复

B 有序，不可重复

C 无序，可重复

D 无序，不可重复

答案

1=>A

ArrayList容器的基本使用

ArrayList是List接口的实现类。是List存储特征的具体实现。

ArrayList底层是用数组实现的存储。 特点：查询效率高，增删效率低，线程不安全。

```
public class ArrayListTest {
    public static void main(String[] args) {
        //实例化ArrayList容器
        List<String> list  = new ArrayList<>();


        //添加元素
        boolean flag1 = list.add("oldlu");
        boolean flag2 = list.add("itbz");
        boolean flag3 = list.add("sxt");
        boolean flag4 = list.add("sxt");

 System.out.println(flag1+"\t"+flag2+"\t"+flag3+"\t"+flag4);

```

```
14
15          //删除元素
16          boolean flag4 =
    list.remove("oldlu");
17          System.out.println(flag4);
18
19          //获取容器中元素的个数
20          int size = list.size();
21          System.out.println(size);
22
23          //判断容器是否为空
24          boolean empty = list.isEmpty();
25          System.out.println(empty);
26
27          //容器中是否包含指定的元素
28          boolean value =
    list.contains("itbz");
29          System.out.println(value);
30
31
32          //清空容器
33          list.clear();
34          Object[] objects1 = list.toArray();
35
     System.out.println(Arrays.toString(objects1
    ));
36      }
37  }
```

实时效果反馈

1.ArrayList容器底层是用什么存储元素的?

A 链表

B 数组

C 队列

D 树

答案

1=>B

ArrayList容器的索引操作

```java
public class ArrayListTest2 {
    public static void main(String[] args) {
        //实例化容器
        List<String> list = new ArrayList<>();
        //添加元素
        list.add("oldlu");
        list.add("itbz");

        //向指定位置添加元素
        list.add(0,"sxt");

        System.out.println("获取元素");
        String value1 = list.get(0);
        System.out.println(value1);

        System.out.println("获取所有元素方式一");
        //使用普通for循环
        for(int i=0;i<list.size();i++){
            System.out.println(list.get(i));
        }

        System.out.println("获取所有元素方式二");
        //使用Foreach循环
        for(String str:list){
            System.out.println(str);
        }

```

```
28
29          System.out.println("元素替换");
30          list.set(1,"kevin");
31          for(String str:list){
32              System.out.println(str);
33          }
34
35
36          System.out.println("根据索引位置删除元
    素");
37          String value2 = list.remove(1);
38          System.out.println(value2);
39          System.out.println("----------------
    ");
40          for(String str:list){
41              System.out.println(str);
42          }
43
44          System.out.println("查找元素第一次出现的
    位置");
45          int value3 = list.indexOf("sxt");
46          System.out.println(value3);
47
48          System.out.println("查找元素最后一次出现
    的位置");
49          list.add("sxt");
50          for(String str:list){
51              System.out.println(str);
52          }
53          int value4 =
    list.lastIndexOf("sxt");
54          System.out.println(value4);
```

```
55
56          }
57  }
```

实时效果反馈

1.在ArrayList中的索引起始数是多少?

A -1

B 1

C 0

D 2

答案

1=>C

ArrayList的并集、交集、差集

并集

```
1           //并集操作：将另一个容器中的元素添加到当前容器中
2           List<String> a  = new ArrayList<>();
```

```
        a.add("a");
        a.add("b");
        a.add("c");

        List<String> b = new ArrayList<>();
        b.add("a");
        b.add("b");
        b.add("c");

        //a并集b
        a.addAll(b);
        for(String str :a){
            System.out.println(str);
        }
```

交集

```
 //交集操作：保留相同的，删除不同的
        List<String> a1  = new ArrayList<>
();
        a1.add("a");
        a1.add("b");
        a1.add("c");

        List<String> b1 = new ArrayList<>();
        b1.add("a");
        b1.add("d");
        b1.add("e");
        //交集操作
        a1.retainAll(b1);
        for(String str :a1){
```

```
            System.out.println(str);
        }
```

差集

```
//差集操作：保留不同的，删除相同的
        List<String> a2  = new ArrayList<>();
        a2.add("a");
        a2.add("b");
        a2.add("c");

        List<String> b2= new ArrayList<>();
        b2.add("b");
        b2.add("c");
        b2.add("d");
        a2.removeAll(b2);
        for(String str :a2){
            System.out.println(str);
        }
```

实时效果反馈

1.ArrayList容器中实现差集操作的方法是?

A addAll

B retainAll

C remove

D removeAll

答案

1=>D

ArrayList源码分析

ArrayList底层是用数组实现的存储。

成员变量

```
/**
 * Default initial capacity.
 */
private static final int DEFAULT_CAPACITY =
10;

/**
 * The array buffer into which the elements
of the ArrayList are stored.
/**
 * The array buffer into which the elements
of the ArrayList are stored.
 * The capacity of the ArrayList is the
length of this array buffer. Any
 * empty ArrayList with elementData ==
DEFAULTCAPACITY_EMPTY_ELEMENTDATA
```

```
 * will be expanded to DEFAULT_CAPACITY when the first element is added.
 */
transient Object[] elementData; // non-private to simplify nested class access

/**
 * The size of the ArrayList (the number of elements it contains).
 *
 * @serial
 */
private int size;
```

数组初始大小

```
/**
 * Default initial capacity.
 */
private static final int DEFAULT_CAPACITY = 10;
```

添加元素

```java
/**
 * Appends the specified element to the end of this list.
 *
 * @param e element to be appended to this list
 * @return <tt>true</tt> (as specified by {@link Collection#add})
 */
public boolean add(E e) {
    ensureCapacityInternal(size + 1);  // Increments modCount!!
    elementData[size++] = e;
    return true;
}
```

判断数组是否扩容

```
//容量检查
private void ensureCapacityInternal(int minCapacity) {

 ensureExplicitCapacity(calculateCapacity(elementData, minCapacity));
}
//容量确认
private void ensureExplicitCapacity(int minCapacity) {
    modCount++;

    //判断是否需要扩容，数组中的元素个数-数组长度，如果大于0表明需要扩容
    if (minCapacity - elementData.length > 0)
        grow(minCapacity);
}
```

数组扩容

```
/**
 * Increases the capacity to ensure that it can hold at least the
 * number of elements specified by the minimum capacity argument.
 *
 * @param minCapacity the desired minimum capacity
 */
private void grow(int minCapacity) {
```

```
    // overflow-conscious code
    int oldCapacity = elementData.length;
//扩容1.5倍
    int newCapacity = oldCapacity +
(oldCapacity >> 1);
    if (newCapacity - minCapacity < 0)
        newCapacity = minCapacity;
    if (newCapacity - MAX_ARRAY_SIZE > 0)
        newCapacity =
hugeCapacity(minCapacity);
    // minCapacity is usually close to size,
so this is a win:
    elementData = Arrays.copyOf(elementData,
newCapacity);
}
```

Vector容器的基本使用

Vector底层是用数组实现的，相关的方法都加了同步检查，因此"线程安全,效率低"。 比如，indexOf方法就增加了synchronized同步标记。

```
public synchronized int indexOf(Object o, int index) {
                    //代码省略
}
```

Vector的使用

Vector的使用与ArrayList是相同的，因为他们都实现了List接口，对List接口中的抽象方法做了具体实现。

```java
public class VectorTest {
    public static void main(String[] args) {

        //实例化Vector
        List<String> v = new Vector<>();
        v.add("a");
        v.add("b");
        v.add("a");

        for(int i=0;i<v.size();i++){
            System.out.println(v.get(i));
        }
        System.out.println("----------------
------");
        for(String str:v){
            System.out.println(str);
        }
    }
}
```

实时效果反馈

1.Vector容器底层是用什么结构存储元素的?

A 链表

B 数组

C 队列

D 树

答案

1=>B

Vector源码分析

成员变量

```
/**
 * The array buffer into which the components of the vector are
 * stored. The capacity of the vector is the length of this array buffer,
 * and is at least large enough to contain all the vector's elements.
 *
 * <p>Any array elements following the last element in the Vector are null.
 *
 * @serial
 */
protected Object[] elementData;

    /**
     * The number of valid components in this {@code Vector} object.
     * Components {@code elementData[0]} through
```

```
15      * {@code elementData[elementCount-1]}
are the actual items.
16      *
17      * @serial
18      */
19     protected int elementCount;
20
21     /**
22      * The amount by which the capacity of
the vector is automatically
23      * incremented when its size becomes
greater than its capacity.  If
24      * the capacity increment is less than
or equal to zero, the capacity
25      * of the vector is doubled each time it
needs to grow.
26      *
27      * @serial
28      */
29     protected int capacityIncrement;
```

构造方法

```
1 public Vector() {
2     this(10);
3 }
```

添加元素

```
/**
 * Appends the specified element to the end of this Vector.
 *
 * @param e element to be appended to this Vector
 * @return {@code true} (as specified by {@link Collection#add})
 * @since 1.2
 */
public synchronized boolean add(E e) {
    modCount++;
    ensureCapacityHelper(elementCount + 1);
    elementData[elementCount++] = e;
    return true;
}
```

数组扩容

```
/**
 * This implements the unsynchronized semantics of ensureCapacity.
 * Synchronized methods in this class can internally call this
 * method for ensuring capacity without incurring the cost of an
 * extra synchronization.
 *
 * @see #ensureCapacity(int)
 */
```

```
9  private void ensureCapacityHelper(int
   minCapacity) {
10     // overflow-conscious code
11 //判断是否需要扩容，数组中的元素个数-数组长度，如果大
   于0表明需要扩容
12     if (minCapacity - elementData.length >
   0)
13         grow(minCapacity);
14 }
```

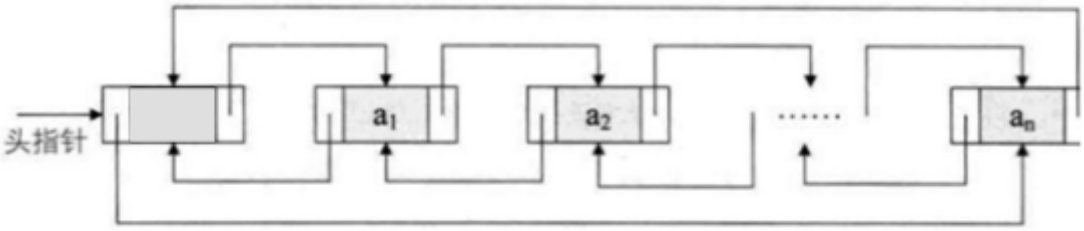
```
1  private void grow(int minCapacity) {
2      // overflow-conscious code
3      int oldCapacity = elementData.length;
4  //扩容2倍
5      int newCapacity = oldCapacity +
   ((capacityIncrement > 0) ?
6
   capacityIncrement : oldCapacity);
7      if (newCapacity - minCapacity < 0)
8          newCapacity = minCapacity;
9      if (newCapacity - MAX_ARRAY_SIZE > 0)
10         newCapacity =
   hugeCapacity(minCapacity);
11     elementData = Arrays.copyOf(elementData,
   newCapacity);
12 }
```

LinkedList容器介绍

LinkedList底层用双向链表实现的存储。特点：查询效率低，增删效率高，线程不安全。

双向链表也叫双链表，是链表的一种，它的每个数据节点中都有两个指针，分别指向前一个节点和后一个节点。 所以，从双向链表中的任意一个节点开始，都可以很方便地找到所有节点。

LinkedList的存储结构图



每个节点都应该有3部分内容：

```
class  Node<E> {
        Node<E>  previous;     //前一个节点
        E  element;            //本节点保存的数据
        Node<E> next;          //后一个节点
}
```

List实现类的选用规则

> 如何选用ArrayList、LinkedList、Vector?
>
> 1. 需要线程安全时，用Vector。
> 2. 不存在线程安全问题时，并且查找较多用ArrayList（一般使用它）
> 3. 不存在线程安全问题时，增加或删除元素较多用LinkedList

实时效果反馈

1.LinkedList容器底层是用什么结构存储元素的?

A 双向链表

B 数组

C 队列

D 树

答案

1=>A

LinkedList容器的使用（List标准）

LinkedList实现了List接口，所以LinkedList是具备List的存储特征的(有序，元素有重复)。

```
public class LinkedListTest {
    public static void main(String[] args) {
        //实例化LinkedList容器
        List<String> list = new LinkedList<>();

        //添加元素
        boolean a = list.add("a");
        boolean b = list.add("b");
        boolean c = list.add("c");
        list.add(3,"a");
        System.out.println(a+"\t"+b+"\t"+c);
```

```
12
13          for(int i=0;i<list.size();i++){
14              System.out.println(list.get(i));
15          }
16      }
17  }
```

LinkedList容器的使用（非List标准）

方法	说明
void addFirst(E e)	将指定元素插入到开头
void addLast(E e)	将指定元素插入到结尾
getFirst()	返回此链表的第一个元素
getLast()	返回此链表的最后一个元素
removeFirst()	移除此链表中的第一个元素，并返回这个元素
removeLast()	移除此链表中的最后一个元素，并返回这个元素
E pop()	从此链表所表示的堆栈处弹出一个元素,等效于removeFirst
void push(E e)	将元素推入此链表所表示的堆栈 这个等效于addFisrt(E e)

```
1  public class LinkedListTest2 {
2      public static void main(String[] args) {
3          System.out.println("-------
   LinkedList-------------");
4          //将指定元素插入到链表开头
5          LinkedList<String> linkedList1 = new
   LinkedList<>();
6          linkedList1.addFirst("a");
```

```
        linkedList1.addFirst("b");
        linkedList1.addFirst("c");
        for (String str:linkedList1){
            System.out.println(str);
        }
        System.out.println("----------------
------");
        //将指定元素插入到链表结尾
        LinkedList<String> linkedList = new
LinkedList<>();
        linkedList.addLast("a");
        linkedList.addLast("b");
        linkedList.addLast("c");
        for (String str:linkedList){
            System.out.println(str);
        }

        System.out.println("----------------
-----------");
        //返回此链表的第一个元素

 System.out.println(linkedList.getFirst());
        //返回此链表的最后一个元素

 System.out.println(linkedList.getLast());

        System.out.println("----------------
-------");
        //移除此链表中的第一个元素，并返回这个元素
        linkedList.removeFirst();
        //移除此链表中的最后一个元素，并返回这个元
素
```

```
        linkedList.removeLast();
        for (String str:linkedList){
            System.out.println(str);
        }

        System.out.println("----------------------");
        linkedList.addLast("c");

        //从此链表所表示的堆栈处弹出一个元素,等效于removeFirst
        linkedList.pop();
        for (String str:linkedList){
            System.out.println(str);
        }
        System.out.println("----------------------");
        //将元素推入此链表所表示的堆栈   这个等效于addFisrt(E e)
        linkedList.push("h");
        for (String str:linkedList){
            System.out.println(str);
        }
    }
}
```

LinkedList的源码分析

添加元素

节点类

```
private static class Node<E> {
    E item;
    Node<E> next;
    Node<E> prev;

    Node(Node<E> prev, E element, Node<E> next) {
        this.item = element;
        this.next = next;
        this.prev = prev;
    }
}
```

成员变量

```
transient int size = 0;

/**
 * Pointer to first node.
 * Invariant: (first == null && last == null) ||
 *            (first.prev == null && first.item != null)
 */
transient Node<E> first;

/**
 * Pointer to last node.
 * Invariant: (first == null && last == null) ||
```

```
 *              (last.next == null &&
last.item != null)
 */
transient Node<E> last;
```

添加元素

```
/**
 * Appends the specified element to the end
of this list.
 *
 * <p>This method is equivalent to {@link
#addLast}.
 *
 * @param e element to be appended to this
list
 * @return {@code true} (as specified by
{@link Collection#add})
 */
public boolean add(E e) {
    linkLast(e);
    return true;
}

/**
 * Links e as last element.
 */
void linkLast(E e) {
    final Node<E> l = last;
    final Node<E> newNode = new Node<>(l, e,
null);
```

```
20      last = newNode;
21      if (l == null)
22          first = newNode;
23      else
24          l.next = newNode;
25      size++;
26      modCount++;
27  }
```

头尾添加元素

addFirst

```
1   /**
2    * Inserts the specified element at the
    beginning of this list.
3    *
4    * @param e the element to add
5    */
6   public void addFirst(E e) {
7       linkFirst(e);
8   }
9
10  /**
11   * Links e as first element.
12   */
13  private void linkFirst(E e) {
14      final Node<E> f = first;
15      final Node<E> newNode = new Node<>(null,
    e, f);
16      first = newNode;
17      if (f == null)
```

```
            last = newNode;
        else
            f.prev = newNode;
        size++;
        modCount++;
    }
```

addLast

```
    /**
     * Appends the specified element to the end
    of this list.
     *
     * <p>This method is equivalent to {@link
    #add}.
     *
     * @param e the element to add
     */
    public void addLast(E e) {
        linkLast(e);
    }

    /**
     * Links e as last element.
     */
    void linkLast(E e) {
        final Node<E> l = last;
        final Node<E> newNode = new Node<>(l, e,
    null);
        last = newNode;
        if (l == null)
```

```
            first = newNode;
        else
            l.next = newNode;
        size++;
        modCount++;
    }
```

获取元素

```
    /**
     * Returns the element at the specified position in this list.
     *
     * @param index index of the element to return
     * @return the element at the specified position in this list
     * @throws IndexOutOfBoundsException {@inheritDoc}
     */
    public E get(int index) {
        checkElementIndex(index);
        return node(index).item;
    }
```

```
private void checkElementIndex(int index) {
    if (!isElementIndex(index))
        throw new
IndexOutOfBoundsException(outOfBoundsMsg(inde
x));
}
```

```
/**
 * Tells if the argument is the index of an
existing element.
 */
private boolean isElementIndex(int index) {
    return index >= 0 && index < size;
}
```

```
/**
 * Returns the (non-null) Node at the
specified element index.
 */
Node<E> node(int index) {
    // assert isElementIndex(index);

    if (index < (size >> 1)) {
        Node<E> x = first;
        for (int i = 0; i < index; i++)
            x = x.next;
        return x;
    } else {
        Node<E> x = last;
        for (int i = size - 1; i > index; i-
-)
            x = x.prev;
```

```
16            return x;
17        }
18    }
```

Set接口介绍

Set接口继承自Collection接口，Set接口中没有新增方法，它和Collection接口保持完全一致。我们在前面学习List接口的使用方式，在Set中仍然适用。因此，学习Set的使用将没有任何难度。

Set接口特点

Set特点：无序、不可重复。无序指Set中的元素没有索引，我们只能遍历查找；不可重复指不允许加入重复的元素。更确切地讲，新元素如果和Set中某个元素通过equals()方法对比为true，则只能保留一个。

Set常用的实现类有：HashSet、TreeSet等，我们一般使用HashSet。

实时效果反馈

1.如下哪个选项是Set接口的特点?

A 有序，可重复

B 有序，不可重复

C 无序，可重复

D 无序，不可重复

答案

1=>D

HashSet容器的使用

HashSet是Set接口的实现类。是Set存储特征的具体实现。

```
public class HashSetTest {
    public static void main(String[] args) {
        //实例化HashSet
        Set<String> set = new HashSet<>();
        //添加元素
        set.add("a");
        set.add("b1");
        set.add("c2");
        set.add("d");
```

```
        set.add("a");

        //获取元素,在Set容器中没有索引，所以没有对应的get(int index)方法
        for(String str: set){
            System.out.println(str);
        }
        System.out.println("----------------");
        //删除元素
        boolean flag = set.remove("c2");
        System.out.println(flag);
        for(String str: set){
            System.out.println(str);
        }
        System.out.println("----------------");
        int size = set.size();
        System.out.println(size);
    }
}
```

实时效果反馈

1.在HashSet容器中，获取指定元素的方法是?

A get();

B get(int index);

C getSet();

D HashSet容器中，无获取指定元素的方法;

答案

1=>D

HashSet存储特征分析

HashSet 是一个不保证元素的顺序且没有重复元素的集合，是线程不安全的。HashSet允许有null 元素。

无序：

在HashSet中底层是使用HashMap存储元素的。HashMap底层使用的是数组与链表实现元素的存储。元素在数组中存放时，并不是有序存放的也不是随机存放的，而是对元素的哈希值进行运算决定元素在数组中的位置。

不重复：

当两个元素的哈希值进行运算后得到相同的在数组中的位置时，会调用元素的equals()方法判断两个元素是否相同。如果元素相同则不会添加该元素，如果不相同则会使用单向链表保存该元素。

实时效果反馈

1.在HashSet容器中，底层是使用存储元素的?

A ArrayList

B Vector

C HashMap

D LinkedList

答案

1=>C

通过HashSet存储自定义对象

创建Users对象

```
public class Users {
    private String username;
    private int userage;

    public Users(String username, int
userage) {
```

```
6           this.username = username;
7           this.userage = userage;
8       }
9
10      public Users() {
11      }
12
13      @Override
14      public boolean equals(Object o) {
15          if (this == o) return true;
16          if (o == null || getClass() !=
o.getClass()) return false;
17
18          Users users = (Users) o;
19
20          if (userage != users.userage) return
false;
21          return username != null ?
username.equals(users.username) :
users.username == null;
22      }
23
24      @Override
25      public int hashCode() {
26          int result = username != null ?
username.hashCode() : 0;
27          result = 31 * result + userage;
28          return result;
29      }
30
31      public String getUsername() {
32          return username;
```

```
    }

    public void setUsername(String username)
{
        this.username = username;
    }

    public int getUserage() {
        return userage;
    }

    public void setUserage(int userage) {
        this.userage = userage;
    }

    @Override
    public String toString() {
        return "Users{" +
                "username='" + username +
'\'' +
                ", userage=" + userage +
                '}';
    }
}
```

在HashSet中存储Users对象

```
public class HashSetTest2 {
    public static void main(String[] args) {
        //实例化HashSet
        Set<Users> set = new HashSet<>();
```

```
        Users u = new Users("oldlu",18);
        Users u1 = new Users("oldlu",18);
        set.add(u);
        set.add(u1);
        System.out.println(u.hashCode());
        System.out.println(u1.hashCode());
        for(Users users:set){
            System.out.println(users);
        }
    }
}
```

HashSet底层源码分析

成员变量

```
private transient HashMap<E,Object> map;

// Dummy value to associate with an Object in the backing Map
private static final Object PRESENT = new Object();
```

添加元素

```
/**
 * Adds the specified element to this set if it is not already present.
```

```
 * More formally, adds the specified element
<tt>e</tt> to this set if
 * this set contains no element <tt>e2</tt>
such that
 * <tt>(e==null&amp;nbsp;?
&amp;nbsp;e2==null&amp;nbsp;:&amp;nbsp;e.equals(e2))
</tt>.
 * If this set already contains the element,
the call leaves the set
 * unchanged and returns <tt>false</tt>.
 *
 * @param e element to be added to this set
 * @return <tt>true</tt> if this set did not
already contain the specified
 * element
 */
public boolean add(E e) {
    return map.put(e, PRESENT)==null;
}
```

TreeSet容器的使用

TreeSet
底层是使用TreeMap存储元素的

TreeSet实现了Set接口，它是一个可以对元素进行排序的容器。底层实际是用TreeMap实现的，内部维持了一个简化版的TreeMap，通过key来存储元素。 TreeSet内部需要对存储的元素进行排序，因此，我们需要给定排序规则。

排序规则实现方式：

- 通过元素自身实现比较规则。
- 通过比较器指定比较规则。

```
public class TreeSetTest {
    public static void main(String[] args) {
        //实例化TreeSet
        Set<String> set = new TreeSet<>();
        //添加元素
        set.add("c");
        set.add("a");
        set.add("d");
        set.add("b");
        set.add("a");

        //获取元素
```

```
13          for(String str :set){
14              System.out.println(str);
15          }
16      }
17  }
```

实时效果反馈

1.在TreeSet容器中，底层是使用存储元素的?

A ArrayList

B TreeMap

C HashMap

D LinkedList

答案

1=>B

通过元素自身实现比较规则

在元素自身实现比较规则时，需要实现Comparable接口中的compareTo方法，该方法中用来定义比较规则。TreeSet通过调用该方法来完成对元素的排序处理。

创建Users类

```
1  public class Users implements
   Comparable<Users>{
2      private String username;
3      private int userage;
4
5      public Users(String username, int
   userage) {
6          this.username = username;
7          this.userage = userage;
8      }
9
```

```
10      public Users() {
11      }
12
13      @Override
14      public boolean equals(Object o) {
15          System.out.println("equals...");
16          if (this == o) return true;
17          if (o == null || getClass() !=
o.getClass()) return false;
18
19          Users users = (Users) o;
20
21          if (userage != users.userage) return
false;
22          return username != null ?
username.equals(users.username) :
users.username == null;
23      }
24
25      @Override
26      public int hashCode() {
27          int result = username != null ?
username.hashCode() : 0;
28          result = 31 * result + userage;
29          return result;
30      }
31
32      public String getUsername() {
33          return username;
34      }
35
```

```
36      public void setUsername(String username)
{
37          this.username = username;
38      }
39
40      public int getUserage() {
41          return userage;
42      }
43
44      public void setUserage(int userage) {
45          this.userage = userage;
46      }
47
48      @Override
49      public String toString() {
50          return "Users{" +
51                  "username='" + username +
'\'' +
52                  ", userage=" + userage +
53                  '}';
54      }
55
56      //定义比较规则
57      //正数：大，负数：小，0：相等
58      @Override
59      public int compareTo(Users o) {
60          if(this.userage > o.getUserage()){
61              return 1;
62          }
63          if(this.userage == o.getUserage()){
64              return
this.username.compareTo(o.getUsername());
```

```
65          }
66          return -1;
67      }
68  }
```

```
1  Set<Users> set1 = new TreeSet<>();
2  Users u = new Users("oldlu",18);
3  Users u1 = new Users("admin",22);
4  Users u2 = new Users("sxt",22);
5  set1.add(u);
6  set1.add(u1);
7  set1.add(u2);
8  for(Users users:set1){
9      System.out.println(users);
10 }
```

实时效果反馈

1.在TreeSet中，当元素自身实现比较规则时需要实现哪个接口?

A Container接口

B Collection接口

C Comparator接口

D Comparable接口

答案

1=>D

通过比较器实现比较规则

通过比较器定义比较规则时，我们需要单独创建一个比较器，比较器需要实现Comparator接口中的compare方法来定义比较规则。在实例化TreeSet时将比较器对象交给TreeSet来完成元素的排序处理。此时元素自身就不需要实现比较规则了。

创建Student类

```
1 public class Student {
2     private String name;
3     private int age;
4
5     public Student(String name, int age) {
```

```
6           this.name = name;
7           this.age = age;
8       }
9
10      public Student() {
11      }
12
13      @Override
14      public String toString() {
15          return "Student{" +
16                  "name='" + name + '\'' +
17                  ", age=" + age +
18                  '}';
19      }
20
21      public String getName() {
22          return name;
23      }
24
25      public void setName(String name) {
26          this.name = name;
27      }
28
29      public int getAge() {
30          return age;
31      }
32
33      public void setAge(int age) {
34          this.age = age;
35      }
36
37      @Override
```

```
38      public boolean equals(Object o) {
39          if (this == o) return true;
40          if (o == null || getClass() !=
    o.getClass()) return false;
41
42          Student student = (Student) o;
43
44          if (age != student.age) return
    false;
45          return name != null ?
    name.equals(student.name) : student.name ==
    null;
46      }
47
48      @Override
49      public int hashCode() {
50          int result = name != null ?
    name.hashCode() : 0;
51          result = 31 * result + age;
52          return result;
53      }
54  }
```

创建比较器

```
1   public class StudentComparator implements
    Comparator<Student> {
2
3       //定义比较规则
4       @Override
```

```
    public int compare(Student o1, Student o2) {
        if(o1.getAge() > o2.getAge()){
            return 1;
        }
        if(o1.getAge() == o2.getAge()){
            return o1.getName().compareTo(o2.getName());
        }
        return -1;
    }
}
```

```
public class TreeSetTest3 {
    public static void main(String[] args) {
        //创建TreeSet容器，并给定比较器对象
        Set<Student> set = new TreeSet<>(new StudentComparator());
        Student s = new Student("oldlu",18);
        Student s1 = new Student("admin",22);
        Student s2 = new Student("sxt",22);
        set.add(s);
        set.add(s1);
        set.add(s2);
        for(Student student:set){
            System.out.println(student);
        }
    }
}
```

实时效果反馈

1.在TreeSet中，定义外部排序比较器时需要实现哪个接口?

A Container接口

B Collection接口

C Comparator接口

D Comparable接口

答案

1=>C

TreeSet底层源码分析

成员变量

```
/**
 * The backing map.
 */
private transient NavigableMap<E,Object> m;

// Dummy value to associate with an Object in the backing Map
private static final Object PRESENT = new Object();
```

构造方法

```
public TreeSet() {
    this(new TreeMap<E,Object>());
}
```

添加元素

```
/**
 * Adds the specified element to this set if it is not already present.
 * More formally, adds the specified element <tt>e</tt> to this set if
 * this set contains no element <tt>e2</tt> such that
 * <tt>(e==null ? e2==null : e.equals(e2))</tt>.
 * If this set already contains the element, the call leaves the set
 * unchanged and returns <tt>false</tt>.
 *
 * @param e element to be added to this set
 * @return <tt>true</tt> if this set did not already contain the specified
 * element
 */
public boolean add(E e) {
    return map.put(e, PRESENT)==null;
}
```

单例集合使用案例

需求:

产生1-10之间的随机数([1,10]闭区间)，将不重复的10个随机数放到容器中。

使用List类型容器实现

```
public class ListDemo {
    public static void main(String[] args) {
        List<Integer> list = new ArrayList<>();
       while(true){
            //产生随机数
            int num = (int)(Math.random()*10+1);
             //判断当前元素在容器中是否存在
            if(!list.contains(num)){
                list.add(num);
            }
            //结束循环
            if(list.size() == 10){
                break;
            }
       }
       for(Integer i:list){
            System.out.println(i);
       }
    }
}
```

使用Set类型容器实现

```
public class SetDemo {
    public static void main(String[] args) {
        Set<Integer> set = new HashSet<>();
        while(true){
            int num = (int)(Math.random()*10+1);
            //将元素添加容器中，由于Set类型容器是不允许有重复元素的，所以不需要判断。
            set.add(num);
            //结束循环
            if(set.size() == 10){
                break;
            }
        }
        for(Integer i:set){
            System.out.println(i);
        }
    }
}
```

双例集合

Map接口介绍

Map接口定义了双例集合的存储特征，它并不是Collection接口的子接口。双例集合的存储特征是以key与value结构为单位进行存储。体现的是数学中的函数 y=f(x)感念。

Map与Collecton的区别：

- Collection中的容器，元素是孤立存在的（理解为单身），向集合中存储元素采用一个个元素的方式存储。
- Map中的容器，元素是成对存在的(理解为现代社会的夫妻)。每个元素由键与值两部分组成，通过键可以找对所对应的值。
- Collection中的容器称为单列集合，Map中的容器称为双列集合。
- Map中的集合不能包含重复的键，值可以重复；每个键只能对应一个值。
- Map中常用的容器为HashMap，TreeMap等。

Map接口中常用的方法表

方法	说明
V put (K key,V value)	把key与value添加到Map集合中
void putAll(Map m)	从指定Map中将所有映射关系复制到此Map中
V remove (Object key)	删除key对应的value
V get(Object key)	根据指定的key，获取对应的value
boolean containsKey(Object key)	判断容器中是否包含指定的key
boolean containsValue(Object value)	判断容器中是否包含指定的value
Set keySet()	获取Map集合中所有的key，存储到Set集合中
Set<Map.Entry<K,V>> entrySet()	返回一个Set基于Map.Entry类型包含Map中所有映射。
void clear()	删除Map中所有的映射

实时效果反馈

1.如下对Map描述错误的是?

A Map接口定义的存储特征是键值对;

B Map接口是双例集合的根接口;

C Map中一个Key只能有一个Value与之对应;

D Map接口是Collection接口的子接口;

2.在Map中添加元素的方法是?

A put

B add

C insert

D addKV

答案

1=>D　2=>A

HashMap容器的使用

HashMap采用哈希算法实现，是Map接口最常用的实现类。 由于底层采用了哈希表存储数据，我们要求键不能重复，如果发生重复，新的键值对会替换旧的键值对。 HashMap在查找、删除、修改方面都有非常高的效率。

```
public class HashMapTest {
    public static void main(String[] args) {
        //实例化HashMap容器
        Map<String,String> map = new HashMap<>();

        //添加元素
        map.put("a","A");
        map.put("b","B");
        map.put("c","C");
        map.put("a","D");

        //获取容器中元素数量
        int size = map.size();
        System.out.println(size);
        System.out.println("---------------");
```

```
16
17          //获取元素
18          //方式一
19          String v = map.get("a");
20          System.out.println(v);
21          System.out.println("---------------
");
22
23          //方式二
24          Set<String> keys = map.keySet();
25          for(String key:keys){
26              String v1 = map.get(key);
27              System.out.println(key+" ----
"+v1);
28          }
29          System.out.println("----------------
---");
30
31          //方式三
32          Set<Map.Entry<String,String>>
entrySet = map.entrySet();
33          for(Map.Entry<String,String>
entry:entrySet){
34              String key = entry.getKey();
35              String v2 = entry.getValue();
36              System.out.println(key+" -------
---  "+v2);
37          }
38
39          System.out.println("----------------
----");
40          //Map容器的并集操作
```

```
41          Map<String,String> map2 = new
HashMap<>();
42          map2.put("f","F");
43          map2.put("c","CC");
44          map.putAll(map2);
45          Set<String> keys2 = map.keySet();
46          for(String key:keys2){
47              System.out.println("key: "+key+"
Value: "+map.get(key));
48          }
49
50          System.out.println("---------------
");
51          //删除元素
52          String v3 = map.remove("a");
53          System.out.println(v3);
54          Set<String> keys3 = map.keySet();
55          for(String key:keys3){
56              System.out.println("key: "+key+"
Value: "+map.get(key));
57          }
58
59          System.out.println("---------------
---");
60          //判断Key是否存在
61          boolean b = map.containsKey("b");
62          System.out.println(b);
63          //判断Value是否存在
64          boolean cc =
map.containsValue("CC");
65          System.out.println(cc);
66
```

```
67          }
68      }
```

HashTable类和HashMap用法几乎一样，底层实现几乎一样，只不过HashTable的方法添加了synchronized关键字确保线程同步检查，效率较低。

> **HashMap与HashTable的区别**
>
> 1. HashMap：线程不安全，效率高。允许key或value为null
> 2. HashTable: 线程安全，效率低。不允许key或value为null

HashMap的底层源码分析

底层存储介绍

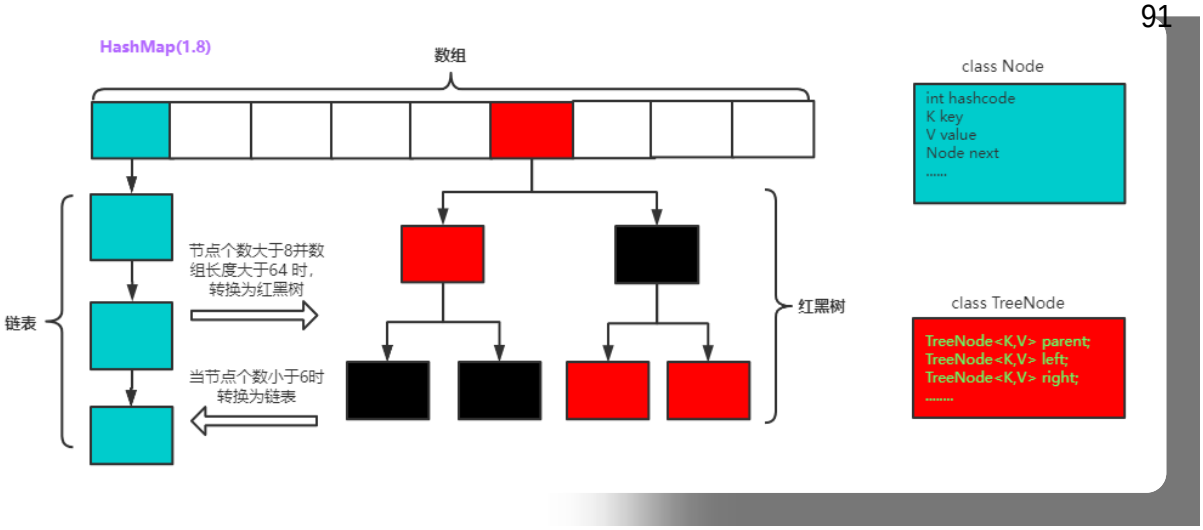
HashMap底层实现采用了哈希表，这是一种非常重要的数据结构。对于我们以后理解很多技术都非常有帮助。

数据结构中由数组和链表来实现对数据的存储，他们各有特点。

(1) 数组：占用空间连续。 寻址容易，查询速度快。但是，增加和删除效率非常低。

(2) 链表：占用空间不连续。 寻址困难，查询速度慢。但是，增加和删除效率非常高。

那么，我们能不能结合数组和链表的优点（即查询快，增删效率也高）呢？ 答案就是“哈希表”。 哈希表的本质就是“数组+链表”。



> **Oldlu建议**
>
> 对于本章中频繁出现的“底层实现”讲解，建议学有余力的童鞋将它搞通。刚入门的童鞋如果觉得有难度，可以暂时跳过。入门期间，掌握如何使用即可，底层原理是扎实内功，便于大家应对一些大型企业的笔试面试。

HashMap中的成员变量

```
/**
 * The default initial capacity - MUST be a power of two.
 */
static final int DEFAULT_INITIAL_CAPACITY = 1 << 4; // aka 16

/**
 * The maximum capacity, used if a higher value is implicitly specified
```

```
8   * by either of the constructors with
    arguments.
9   * MUST be a power of two <= 1<<30.
10  */
11  static final int MAXIMUM_CAPACITY = 1 << 30;
12
13  /**
14  * The load factor used when none specified
    in constructor.
15  */
16  static final float DEFAULT_LOAD_FACTOR =
    0.75f;
17
18  /**
19  * The bin count threshold for using a tree
    rather than list for a
20  * bin.  Bins are converted to trees when
    adding an element to a
21  * bin with at least this many nodes. The
    value must be greater
22  * than 2 and should be at least 8 to mesh
    with assumptions in
23  * tree removal about conversion back to
    plain bins upon
24  * shrinkage.
25  */
26  static final int TREEIFY_THRESHOLD = 8;
27
28  /**
29  * The bin count threshold for untreeifying
    a (split) bin during a
```

```
30  * resize operation. Should be less than
    TREEIFY_THRESHOLD, and at
31  * most 6 to mesh with shrinkage detection
    under removal.
32  */
33 static final int UNTREEIFY_THRESHOLD = 6;
34
35 /**
36  * The smallest table capacity for which
    bins may be treeified.
37  * (Otherwise the table is resized if too
    many nodes in a bin.)
38  * Should be at least 4 * TREEIFY_THRESHOLD
    to avoid conflicts
39  * between resizing and treeification
    thresholds.
40  */
41 static final int MIN_TREEIFY_CAPACITY = 64;
42 /**
43  * The number of key-value mappings
    contained in this map.
44  */
45 transient int size;
46
47 /**
48  * The table, initialized on first use, and
    resized as
49  * necessary. When allocated, length is
    always a power of two.
50  * (We also tolerate length zero in some
    operations to allow
```

```
 * bootstrapping mechanics that are currently not needed.)
 */
transient Node<K,V>[] table;
```

HashMap中存储元素的节点类型

Node类

```
static class Node<K,V> implements Map.Entry<K,V> {
    final int hash;
    final K key;
    V value;
    Node<K,V> next;

    Node(int hash, K key, V value, Node<K,V> next) {
        this.hash = hash;
        this.key = key;
        this.value = value;
        this.next = next;
    }

    public final K getKey()        { return key; }
    public final V getValue()      { return value; }
    public final String toString() { return key + "=" + value; }

    public final int hashCode() {
```

```
        return Objects.hashCode(key) ^ Objects.hashCode(value);
    }

    public final V setValue(V newValue) {
        V oldValue = value;
        value = newValue;
        return oldValue;
    }

    public final boolean equals(Object o) {
        if (o == this)
            return true;
        if (o instanceof Map.Entry) {
            Map.Entry<?,?> e = (Map.Entry<?,?>)o;
            if (Objects.equals(key, e.getKey()) &&
                Objects.equals(value, e.getValue()))
                return true;
        }
        return false;
    }
}
```

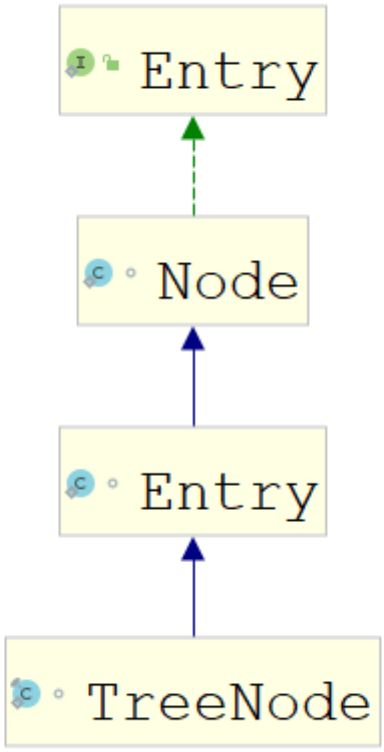
TreeNode类

```
/**
 * Entry for Tree bins. Extends LinkedHashMap.Entry (which in turn
```

```
 * extends Node) so can be used as extension of either regular or
 * linked node.
 */
static final class TreeNode<K,V> extends LinkedHashMap.Entry<K,V> {
    TreeNode<K,V> parent;  // red-black tree links
    TreeNode<K,V> left;
    TreeNode<K,V> right;
    TreeNode<K,V> prev;    // needed to unlink next upon deletion
    boolean red;
    TreeNode(int hash, K key, V val, Node<K,V> next) {
        super(hash, key, val, next);
    }

    /**
     * Returns root of tree containing this node.
     */
    final TreeNode<K,V> root() {
        for (TreeNode<K,V> r = this, p;;) {
            if ((p = r.parent) == null)
                return r;
            r = p;
        }
    }
```

它们的继承关系



HashMap中的数组初始化

在JDK1.8的HashMap中对于数组的初始化采用的是延迟初始化方式。通过resize方法实现初始化处理。resize方法既实现数组初始化，也实现数组扩容处理。

```
1 /**
2  * Initializes or doubles table size.  If
null, allocates in
3  * accord with initial capacity target held
in field threshold.
4  * Otherwise, because we are using power-of-
two expansion, the
```

```
 * elements from each bin must either stay at same index, or move
 * with a power of two offset in the new table.
 *
 * @return the table
 */
final Node<K,V>[] resize() {
    Node<K,V>[] oldTab = table;
    int oldCap = (oldTab == null) ? 0 : oldTab.length;
    int oldThr = threshold;
    int newCap, newThr = 0;
    if (oldCap > 0) {
        if (oldCap >= MAXIMUM_CAPACITY) {
            threshold = Integer.MAX_VALUE;
            return oldTab;
        }
        else if ((newCap = oldCap << 1) < MAXIMUM_CAPACITY &&
                 oldCap >= DEFAULT_INITIAL_CAPACITY)
            newThr = oldThr << 1; // double threshold
    }
    else if (oldThr > 0) // initial capacity was placed in threshold
        newCap = oldThr;
    else {               // zero initial threshold signifies using defaults
        newCap = DEFAULT_INITIAL_CAPACITY;
```

```
28          newThr = (int)(DEFAULT_LOAD_FACTOR *
DEFAULT_INITIAL_CAPACITY);
29      }
30      if (newThr == 0) {
31          float ft = (float)newCap *
loadFactor;
32          newThr = (newCap < MAXIMUM_CAPACITY
&& ft < (float)MAXIMUM_CAPACITY ?
33                    (int)ft :
Integer.MAX_VALUE);
34      }
35      threshold = newThr;
36
 @SuppressWarnings({"rawtypes","unchecked"})
37          Node<K,V>[] newTab = (Node<K,V>
[])new Node[newCap];
38      table = newTab;
39      if (oldTab != null) {
40          for (int j = 0; j < oldCap; ++j) {
41              Node<K,V> e;
42              if ((e = oldTab[j]) != null) {
43                  oldTab[j] = null;
44                  if (e.next == null)
45                      newTab[e.hash & (newCap
- 1)] = e;
46                  else if (e instanceof
TreeNode)
47
((TreeNode<K,V>)e).split(this, newTab, j,
oldCap);
48                  else { // preserve order
```

```
                    Node<K,V> loHead = null, loTail = null;
                    Node<K,V> hiHead = null, hiTail = null;
                    Node<K,V> next;
                    do {
                        next = e.next;
                        if ((e.hash & oldCap) == 0) {
                            if (loTail == null)
                                loHead = e;
                            else
                                loTail.next = e;
                            loTail = e;
                        }
                        else {
                            if (hiTail == null)
                                hiHead = e;
                            else
                                hiTail.next = e;
                            hiTail = e;
                        }
                    } while ((e = next) != null);
                    if (loTail != null) {
                        loTail.next = null;
                        newTab[j] = loHead;
                    }
```

```
73                          if (hiTail != null) {
74                              hiTail.next = null;
75                              newTab[j + oldCap] =
   hiHead;
76                          }
77                      }
78                  }
79              }
80          }
81          return newTab;
82      }
```

HashMap中计算Hash值

1. 获得key对象的hashcode
   首先调用key对象的hashcode()方法，获得key的hashcode值。

2. 根据hashcode计算出hash值(要求在[0, 数组长度-1]区间)hashcode是一个整数，我们需要将它转化成[0, 数组长度-1]的范围。我们要求转化后的hash值尽量均匀地分布在[0,数组长度-1]这个区间，减少“hash冲突”
   - 一种极端简单和低下的算法是：
     hash值 = hashcode/hashcode;
     也就是说，hash值总是1。意味着，键值对对象都会存储到数组索引1位置，这样就形成一个非常长的链表。相当于每存储一个对象都会发生“hash冲突”，HashMap也退化成了一个“链表”。

   - 一种简单和常用的算法是（相除取余算法）：
     hash值 = hashcode%数组长度;

这种算法可以让hash值均匀的分布在[0,数组长度-1]的区间。但是，这种算法由于使用了“除法”，效率低下。JDK后来改进了算法。首先约定数组长度必须为2的整数幂，这样采用位运算即可实现取余的效果：hash值 = hashcode&(数组长度-1)。

```
/**
 * Associates the specified value with the specified key in this map.
 * If the map previously contained a mapping for the key, the old
 * value is replaced.
 *
 * @param key key with which the specified value is to be associated
 * @param value value to be associated with the specified key
 * @return the previous value associated with <tt>key</tt>, or
 *         <tt>null</tt> if there was no mapping for <tt>key</tt>.
 *         (A <tt>null</tt> return can also indicate that the map
 *         previously associated <tt>null</tt> with <tt>key</tt>.)
 */
public V put(K key, V value) {
    return putVal(hash(key), key, value, false, true);
}
```

```
static final int hash(Object key) {
    int h;
    return (key == null) ? 0 : (h =
key.hashCode()) ^ (h >>> 16);
}
```

```
/**
 * Implements Map.put and related methods
 *
 * @param hash hash for key
 * @param key the key
 * @param value the value to put
 * @param onlyIfAbsent if true, don't change
existing value
 * @param evict if false, the table is in
creation mode.
 * @return previous value, or null if none
 */
final V putVal(int hash, K key, V value,
boolean onlyIfAbsent,
               boolean evict) {
    Node<K,V>[] tab; Node<K,V> p; int n, i;
    if ((tab = table) == null || (n =
tab.length) == 0)
        n = (tab = resize()).length;
    if ((p = tab[i = (n - 1) & hash]) ==
null)
        tab[i] = newNode(hash, key, value,
null);
    else {
        Node<K,V> e; K k;
        if (p.hash == hash &&
```

```
21              ((k = p.key) == key || (key != null && key.equals(k))))
22              e = p;
23          else if (p instanceof TreeNode)
24              e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);
25          else {
26              for (int binCount = 0; ; ++binCount) {
27                  if ((e = p.next) == null) {
28                      p.next = newNode(hash, key, value, null);
29                      if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for 1st
30                          treeifyBin(tab, hash);
31                      break;
32                  }
33                  if (e.hash == hash &&
34                      ((k = e.key) == key || (key != null && key.equals(k))))
35                      break;
36                  p = e;
37              }
38          }
39          if (e != null) { // existing mapping for key
40              V oldValue = e.value;
41              if (!onlyIfAbsent || oldValue == null)
42                  e.value = value;
```

```
                afterNodeAccess(e);
                return oldValue;
            }
        }
        ++modCount;
        if (++size > threshold)
            resize();
        afterNodeInsertion(evict);
        return null;
    }
```

HashMap中添加元素

```
/**
 * Associates the specified value with the specified key in this map.
 * If the map previously contained a mapping for the key, the old
 * value is replaced.
 *
 * @param key key with which the specified value is to be associated
 * @param value value to be associated with the specified key
 * @return the previous value associated with <tt>key</tt>, or
 *         <tt>null</tt> if there was no mapping for <tt>key</tt>.
 *         (A <tt>null</tt> return can also indicate that the map
```

```
 *         previously associated
<tt>null</tt> with <tt>key</tt>.)
 */
public V put(K key, V value) {
    return putVal(hash(key), key, value,
false, true);
}
```

HashMap中数组扩容

```
/**
 * Implements Map.put and related methods
 *
 * @param hash hash for key
 * @param key the key
 * @param value the value to put
 * @param onlyIfAbsent if true, don't change
existing value
 * @param evict if false, the table is in
creation mode.
 * @return previous value, or null if none
 */
final V putVal(int hash, K key, V value,
boolean onlyIfAbsent,
               boolean evict) {
    Node<K,V>[] tab; Node<K,V> p; int n, i;
    if ((tab = table) == null || (n =
tab.length) == 0)
        n = (tab = resize()).length;
    if ((p = tab[i = (n - 1) & hash]) ==
null)
```

```
17          tab[i] = newNode(hash, key, value,
null);
18      else {
19          Node<K,V> e; K k;
20          if (p.hash == hash &&
21              ((k = p.key) == key || (key !=
null && key.equals(k))))
22              e = p;
23          else if (p instanceof TreeNode)
24              e =
((TreeNode<K,V>)p).putTreeVal(this, tab,
hash, key, value);
25          else {
26              for (int binCount = 0; ;
++binCount) {
27                  if ((e = p.next) == null) {
28                      p.next = newNode(hash,
key, value, null);
29                      if (binCount >=
TREEIFY_THRESHOLD - 1) // -1 for 1st
30                          treeifyBin(tab,
hash);
31                      break;
32                  }
33                  if (e.hash == hash &&
34                      ((k = e.key) == key ||
(key != null && key.equals(k))))
35                      break;
36                  p = e;
37              }
38          }
```

```
        if (e != null) { // existing mapping for key
            V oldValue = e.value;
            if (!onlyIfAbsent || oldValue == null)
                e.value = value;
            afterNodeAccess(e);
            return oldValue;
        }
    }
    ++modCount;
    if (++size > threshold)
        resize();
    afterNodeInsertion(evict);
    return null;
}
```

```
/**
 * Initializes or doubles table size.  If null, allocates in
 * accord with initial capacity target held in field threshold.
 * Otherwise, because we are using power-of-two expansion, the
 * elements from each bin must either stay at same index, or move
 * with a power of two offset in the new table.
 *
 * @return the table
 */
final Node<K,V>[] resize() {
```

```
11      Node<K,V>[] oldTab = table;
12      int oldCap = (oldTab == null) ? 0 :
oldTab.length;
13      int oldThr = threshold;
14      int newCap, newThr = 0;
15      if (oldCap > 0) {
16          if (oldCap >= MAXIMUM_CAPACITY) {
17              threshold = Integer.MAX_VALUE;
18              return oldTab;
19          }
20          else if ((newCap = oldCap << 1) <
MAXIMUM_CAPACITY &&
21                   oldCap >=
DEFAULT_INITIAL_CAPACITY)
22              newThr = oldThr << 1; // double
threshold
23      }
24      else if (oldThr > 0) // initial capacity
was placed in threshold
25          newCap = oldThr;
26      else {                // zero initial
threshold signifies using defaults
27          newCap = DEFAULT_INITIAL_CAPACITY;
28          newThr = (int)(DEFAULT_LOAD_FACTOR *
DEFAULT_INITIAL_CAPACITY);
29      }
30      if (newThr == 0) {
31          float ft = (float)newCap *
loadFactor;
32          newThr = (newCap < MAXIMUM_CAPACITY
&& ft < (float)MAXIMUM_CAPACITY ?
```

```
                    (int)ft :
Integer.MAX_VALUE);
    }
    threshold = newThr;

 @SuppressWarnings({"rawtypes","unchecked"})
        Node<K,V>[] newTab = (Node<K,V>
[])new Node[newCap];
    table = newTab;
    if (oldTab != null) {
        for (int j = 0; j < oldCap; ++j) {
            Node<K,V> e;
            if ((e = oldTab[j]) != null) {
                oldTab[j] = null;
                if (e.next == null)
                    newTab[e.hash & (newCap
- 1)] = e;
                else if (e instanceof
TreeNode)

((TreeNode<K,V>)e).split(this, newTab, j,
oldCap);
                else { // preserve order
                    Node<K,V> loHead = null,
loTail = null;
                    Node<K,V> hiHead = null,
hiTail = null;
                    Node<K,V> next;
                    do {
                        next = e.next;
                        if ((e.hash &
oldCap) == 0) {
```

```
55                                  if (loTail ==
null)
56                                      loHead = e;
57                                  else
58                                      loTail.next
= e;
59                                      loTail = e;
60                                  }
61                                  else {
62                                      if (hiTail ==
null)
63                                          hiHead = e;
64                                      else
65                                          hiTail.next
= e;
66                                          hiTail = e;
67                                  }
68                              } while ((e = next) !=
null);
69                              if (loTail != null) {
70                                  loTail.next = null;
71                                  newTab[j] = loHead;
72                              }
73                              if (hiTail != null) {
74                                  hiTail.next = null;
75                                  newTab[j + oldCap] =
hiHead;
76                              }
77                          }
78                      }
79                  }
80              }
```

```
81        return newTab;
82    }
```

TreeMap容器的使用

TreeMap和HashMap同样实现了Map接口，所以，对于API的用法来说是没有区别的。HashMap效率高于TreeMap；TreeMap是可以对键进行排序的一种容器，在需要对键排序时可选用TreeMap。TreeMap底层是基于红黑树实现的。

在使用TreeMap时需要给定排序规则：

- 元素自身实现比较规则
- 通过比较器实现比较规则

元素自身实现比较规则

```
1  public class Users implements
   Comparable<Users>{
2      private String username;
3      private int userage;
```

```
4
5       public Users(String username, int
    userage) {
6           this.username = username;
7           this.userage = userage;
8       }
9
10      public Users() {
11      }
12
13      @Override
14      public boolean equals(Object o) {
15          System.out.println("equals...");
16          if (this == o) return true;
17          if (o == null || getClass() !=
    o.getClass()) return false;
18
19          Users users = (Users) o;
20
21          if (userage != users.userage) return
    false;
22          return username != null ?
    username.equals(users.username) :
    users.username == null;
23      }
24
25      @Override
26      public int hashCode() {
27          int result = username != null ?
    username.hashCode() : 0;
28          result = 31 * result + userage;
29          return result;
```

```
30      }
31
32      public String getUsername() {
33          return username;
34      }
35
36      public void setUsername(String username)
{
37          this.username = username;
38      }
39
40      public int getUserage() {
41          return userage;
42      }
43
44      public void setUserage(int userage) {
45          this.userage = userage;
46      }
47
48      @Override
49      public String toString() {
50          return "Users{" +
51                  "username='" + username +
'\'' +
52                  ", userage=" + userage +
53                  '}';
54      }
55
56      //定义比较规则
57      //正数：大，负数：小，0：相等
58      @Override
59      public int compareTo(Users o) {
```

```
        if(this.userage < o.getUserage()){
            return 1;
        }
        if(this.userage == o.getUserage()){
           return
this.username.compareTo(o.getUsername());
        }
        return -1;
    }
}
```

```
public class TreeMapTest {
    public static void main(String[] args) {
        //实例化TreeMap
        Map<Users,String> map = new
TreeMap<>();
        Users u1 = new Users("oldlu",18);
        Users u2 = new Users("admin",22);
        Users u3 = new Users("sxt",22);
        map.put(u1,"oldlu");
        map.put(u2,"admin");
        map.put(u3,"sxt");
        Set<Users> keys = map.keySet();
        for(Users key :keys){
            System.out.println(key+" -------
-- "+map.get(key));
        }
    }
}
```

通过比较器实现比较规则

```java
public class Student {
    private String name;
    private int age;

    public Student(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public Student() {
    }

    @Override
    public String toString() {
        return "Student{" +
                "name='" + name + '\'' +
                ", age=" + age +
                '}';
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

```

```
    public void setAge(int age) {
        this.age = age;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;

        Student student = (Student) o;

        if (age != student.age) return false;
        return name != null ? name.equals(student.name) : student.name == null;
    }

    @Override
    public int hashCode() {
        int result = name != null ? name.hashCode() : 0;
        result = 31 * result + age;
        return result;
    }
}
```

```
public class StudentComparator implements Comparator<Student> {

    //定义比较规则
```

```
    @Override
    public int compare(Student o1, Student o2) {
        if(o1.getAge() > o2.getAge()){
            return 1;
        }
        if(o1.getAge() == o2.getAge()){
            return o1.getName().compareTo(o2.getName());
        }
        return -1;
    }
}
```

```
public class TreeMapTest {
    public static void main(String[] args) {
          Map<Student,String> treeMap = new TreeMap<>(new StudentComparator());
    Student s1 = new Student("oldlu",18);
    Student s2 = new Student("admin",22);
    Student s3 = new Student("sxt",22);
    treeMap.put(s1,"oldlu");
    treeMap.put(s2,"admin");
    treeMap.put(s3,"sxt");
    Set<Student> keys1 = treeMap.keySet();
    for(Student key :keys1){
        System.out.println(key+" ---- "+treeMap.get(key));
      }
    }
}
```

实时效果反馈

1.TreeMap可以对容器中的排序?

A 对Key排序;

B 对Value排序;

C 对Key与Value同时排序;

D 对指定的Key与Value进行排序;

答案

1=>A

TreeMap的底层源码分析

TreeMap是红黑二叉树的典型实现。我们打开TreeMap的源码，发现里面有一行核心代码：

```
private transient Entry<K,V> root = null;
```

root用来存储整个树的根节点。我们继续跟踪Entry（是TreeMap的内部类）的代码：

```
static final class Entry<K,V> implements Map.Entry<K,V> {
    K key;
    V value;
    Entry<K,V> left;
    Entry<K,V> right;
    Entry<K,V> parent;
    boolean color = BLACK;
```

可以看到里面存储了本身数据、左节点、右节点、父节点、以及节点颜色。 TreeMap的put()/remove()方法大量使用了红黑树的理论。在本节课中，不再展开。需要了解更深入的，可以参考专门的数据结构书籍。

TreeMap和HashMap实现了同样的接口Map，因此，用法对于调用者来说没有区别。HashMap效率高于TreeMap；在需要排序的Map时才选用TreeMap。

实时效果反馈

1.TreeMap容器底层是使用什么结构存储数据的?

A 数组;

B 链表;

C 哈希表;

D 红黑树;

答案

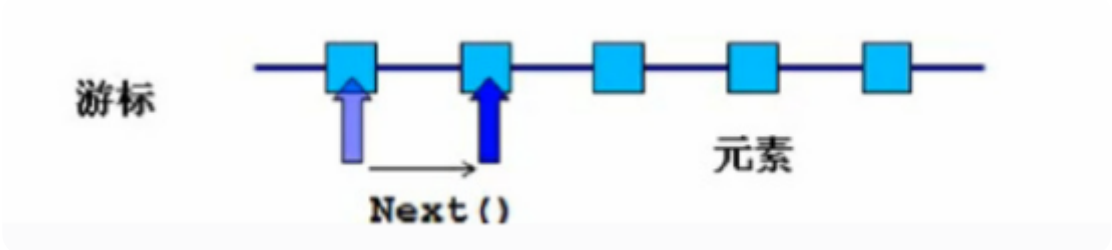
1=>D

Iterator接口

Iterator迭代器接口介绍

Collection接口继承了Iterable接口，在该接口中包含一个名为iterator的抽象方法，所有实现了Collection接口的容器类对该方法做了具体实现。iterator方法会返回一个Iterator接口类型的迭代器对象，在该对象中包含了三个方法用于实现对单例容器的迭代处理。

Iterator对象的工作原理：



Iterator接口定义了如下方法：

1. `boolean hasNext();` //判断游标当前位置的下一个位置是否还有元素没有被遍历；
2. `Object next();` //返回游标当前位置的下一个元素并将游标移动到下一个位置；
3. `void remove();` //删除游标当前位置的元素，在执行完next后该操作只能执行一次；

实时效果反馈

1.Iterator接口的作用是?

A 定义了迭代双例集合的标准；

B 定义了迭代单例集合的标准；

C 定义了迭代数组的标准；

D 定义了迭代树型结构的标准；

答案

1=>B

Iterator迭代器的使用

迭代List接口类型容器

```
public class IteratorListTest {
    public static void main(String[] args) {
        //实例化容器
        List<String> list  = new ArrayList<>();
        list.add("a");
        list.add("b");
        list.add("c");
        //获取元素
        //获取迭代器对象
        Iterator<String> iterator = list.iterator();
        //方式一:在迭代器中，通过while循环获取元素
        while(iterator.hasNext()){
```

```
            String value = iterator.next();
            System.out.println(value);
        }
        System.out.println("----------------
---------------");
        //方法二：在迭代器中，通过for循环获取元素
        for(Iterator<String> it =
list.iterator();it.hasNext();){
            String value = it.next();
            System.out.println(value);
        }

    }
}
```

迭代Set接口类型容器

```
public class IteratorSetTest {
    public static void main(String[] args) {
        //实例化Set类型的容器
        Set<String> set  = new HashSet<>();
        set.add("a");
        set.add("b");
        set.add("c");
        //方式一：通过while循环
        //获取迭代器对象
        Iterator<String> iterator =
set.iterator();
        while(iterator.hasNext()){
            String value = iterator.next();
            System.out.println(value);
```

```
        }
        System.out.println("----------------
---------");
        //方式二：通过for循环
        for(Iterator<String> it =
set.iterator();it.hasNext();){
            String value = it.next();
            System.out.println(value);
        }
    }
}
```

迭代Map接口类型容器

```
public class IteratorMapTest {
    public static void main(String[] args) {
        //实例化HashMap容器
        Map<String, String> map = new
HashMap<String, String>();

        //添加元素
        map.put("a", "A");
        map.put("b", "B");
        map.put("c", "C");

        //遍历Map容器方式一
        Set<String> keySet = map.keySet();

        for (Iterator<String> it =
keySet.iterator(); it.hasNext();){
            String key = it.next();
```

```
            String value = map.get(key);
            System.out.println(key+" ------ "+value);
        }
        System.out.println("----------------");

        //遍历Map容器方式二
        Set<Map.Entry<String, String>> entrySet = map.entrySet();
        Iterator<Map.Entry<String, String>> iterator = entrySet.iterator();
        while(iterator.hasNext()){
            Map.Entry entry = iterator.next();

         System.out.println(entry.getKey()+" ------ "+ entry.getValue());
        }
    }
}
```

在迭代器中删除元素

```
public class IteratorRemoveTest {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("a");
        list.add("b");
        list.add("c");
```

```
        list.add("d");
        Iterator<String> iterator = list.iterator();
        while(iterator.hasNext()){
            //不要在一次循环中多次调用next方法。
            String value = iterator.next();
            iterator.remove();
        }
        System.out.println("----------------");
        for(Iterator<String> it = list.iterator();it.hasNext();){
            System.out.println(it.next());
            list.add("dddd");
        }
    }
}
```

遍历集合的方法总结

遍历List方法一：普通for循环

```
for(int i=0;i<list.size();i++){//list为集合的对象名
    String temp = (String)list.get(i);
    System.out.println(temp);
}
```

遍历List方法二：增强for循环(使用泛型！)

```
for (String temp : list) {
    System.out.println(temp);
}
```

遍历List方法三：使用Iterator迭代器(1)

```
for(Iterator iter=
list.iterator();iter.hasNext();){
    String temp = (String)iter.next();
    System.out.println(temp);
}
```

遍历List方法四：使用Iterator迭代器(2)

```
Iterator  iter =list.iterator();
while(iter.hasNext()){
    Object  obj =  iter.next();
    iter.remove();//如果要遍历时，删除集合中的元
素，建议使用这种方式！
    System.out.println(obj);
}
```

遍历Set方法一：增强for循环

```
for(String temp:set){
    System.out.println(temp);
}
```

遍历Set方法二：使用Iterator迭代器

```
for(Iterator iter =
set.iterator();iter.hasNext();){
    String temp = (String)iter.next();
    System.out.println(temp);
}
```

遍历Map方法一：根据key获取value

```
Map<Integer, Man> maps = new HashMap<Integer,
Man>();
Set<Integer>  keySet =  maps.keySet();
for(Integer id : keySet){
    System.out.println(maps.get(id).name);
}
```

遍历Map方法二：使用entrySet

```
Set<Map.Entry<Integer, Man>>  ss =
maps.entrySet();
for (Iterator<Map.Entry<Integer, Man>>
iterator = ss.iterator();
iterator.hasNext();) {
    Map.Entry e =  iterator.next();
    System.out.println(e.getKey()+"--
"+e.getValue());
}
```

Collections工具类

类 java.util.Collections 提供了对Set、List、Map进行排序、填充、查找元素的辅助方法。

方法名	说明
void sort(List)	对List容器内的元素排序，排序规则是升序。
void shuffle(List)	对List容器内的元素进行随机排列
void reverse(List)	对List容器内的元素进行逆续排列
void fill(List, Object)	用一个特定的对象重写整个List容器
int binarySearch(List, Object)	对于顺序的List容器，折半查找查找特定对象

Collections工具类的常用方法

```
public class CollectionsTest {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>
();
```

```
        list.add("c");
        list.add("b");
        list.add("a");
        //对元素排序
        Collections.sort(list);
        for(String str:list){
            System.out.println(str);
        }
        System.out.println("----------------");

        List<Users> list2 = new ArrayList<>();
        Users u = new Users("oldlu",18);
        Users u2 = new Users("sxt",22);
        Users u3 = new Users("admin",22);
        list2.add(u);
        list2.add(u2);
        list2.add(u3);
        //对元素排序
        Collections.sort(list2);
        for(Users user:list2){
            System.out.println(user);
        }
        System.out.println("----------------");

        List<Student> list3 = new ArrayList<>();
        Student s = new Student("oldlu",18);
        Student s1 = new Student("sxt",20);
```

```
        Student s2 = new Student("admin",20);

        list3.add(s);
        list3.add(s1);
        list3.add(s2);

        Collections.sort(list3,new StudentComparator());
        for(Student student:list3){
            System.out.println(student);
        }
        System.out.println("----------------");

        List<String> list4 = new ArrayList<>();
        list4.add("a");
        list4.add("b");
        list4.add("c");
        list4.add("d");

        //洗牌
        Collections.shuffle(list4);
        for(String str:list4){
            System.out.println(str);
        }
    }
}
```

实时效果反馈

1.如下对Collections描述正确的是?

A 是单例集合接口;

B 是双例集合接口;

C 是操作容器的工具类;

D 是迭代器接口;

答案

1=>C