

**Collection :** 单列集合的根接口

**List :** 有序(存和取的顺序一致),有索引,可以存储重复

**ArrayList :** 数组实现，查找快，但是增删慢，线程*不安全*

**LinkedList :** 链表实现，增删快，但是查找慢，线程*不安全*

**Vector(向量) :** 数组实现，线程*安全*

**Set :** 无序(存和取的顺序不一致),无索引,不可以存储重复

**HashSet :** 哈希算法，

**TreeSet :** 二叉树算法

**格式 :**

类名称<泛型> 对象名 = new 类名称<>();

**java.util.Collection :**

常用方法：

\* public boolean add(E element)：注意E代表泛型，添加一个元素进入到集合中。

\* public boolean remove(Object obj)：从集合当中删除指定的元素。

\* public void clear()：清空集合当中所有的元素。

\* public boolean isEmpty()：判断集合是否为空

\* public boolean contains(Object obj)：判断指定的元素在集合当中是否存在(包含)

\* public int size()：获取集合的长度

\* public Object[] toArray()：将集合转换成为对象数组

**迭代器 :**

通常使用的get(index)的方法只能适用于XxxList，不能适用于所有的集合。

有一种统一的、适用于所有集合的遍历方法：迭代器。

迭代器的接口就是：java.util.Iterator<E>

所有的集合都是Collection接口的实现类，所以在Collection当中提供了一个方法用来**获取迭代器：**

\* public Iterator<E> iterator()：调用该方法获取一个集合对应的迭代器。

**Iterator接口**当中提供**的方法**：

\* public boolean hasNext()：判断对应的集合当中有没有下一个元素

\* public E next()：获取下一个元素，并且移动光标到下一个位置。

\* public void remove()：删除当前next()所指向的元素

如何使用迭代器遍历集合？步骤：

1. 调用集合当中的iterator()方法，获取一个迭代器。

Iterator<E> iter = 集合名.iterator();

// 2. 调用迭代器的hasNext()方法，判断有没有下一个元素。

while (iter.hasNext()) {

// 3. 如果有下一个元素，那么调用迭代器的next()方法，获取该元素。

String str = iter.next();

System.out.println(str);

}

一定要**避免**在使用**迭代器遍历集合**的过程当中，**改变（添加或删除）**集合**元素**的**个数**。

如果需要删除当前元素，不能通过集合直接删除，必须通过Iterator当中的remove()方法删除当前元素。

否则就会并发修改异常：ConcurrentModificationException

**增强for循环：**

从JDK 1.5开始，引入了一个特性，叫做增强for循环（for-each）。

格式：

for (数据类型 变量名称 : 数组或集合名称) {

// ...

}

for-each格式**只**能**支持数组或**者是java.lang.**Iterable接口的实现类**。

**Iterable接口的实现类多为集合**

\* 好处：格式简单一些，省去了index索引值的操作。

\* 弊端：凡是需要用到索引的地方，因为没有索引，所以用不了了。这时候，还要使用标准的for循环。

**泛型：**

好处：

\* 1. 不符合类型的数据，保证进不来。

\* 2. 从集合中取出来数据的时候，不需要进行强转，直接就是泛型类型。

\* 3. 意义：将可能发生的错误，从运行时，提前到编译时。

【重要概念】在Java语言中，泛型其实是**伪泛型**。

\* 泛型只在**编译**的时候**存在**，一旦进入**运行时**，就没有**泛型**了。

\* 泛型只是给编译器使用的一种类型安全手段，真正运行程序的时候，没有泛型。

可以在一个类的名称定义后面，写上一个“<泛型>”，那么这就是一个泛型类。

在本类当中，凡是用到泛型的地方，就都能写成T。

如果希望缩小泛型的作用域，从类缩小到一个方法上，也就是定义一个方法自己专用的泛型，那么：

就需要使用**泛型方法**的格式：

修饰符 <E>返回值类型 方法名(参数) {

// ...

}

使用泛型方法：

\* 1. 根据参数能够自动匹配泛型到底是谁。

\* 2. 对象名**.<类型>**方法名();

指定泛型接口的泛型参数呢？

\* 1. 来一个实现类实现接口，并且在implements后面直接指定具体的泛型。

\* 2. 实现类仍然不指定泛型参数，而是继续携带泛型，但是在真正new实现类的时候，再外部指定。

泛型之间没有继承关系。

\* 如果希望一个方法的参数，既能接收ArrayList<A>，又能接收ArrayList<B>。

\* 此时必须使用“通配符”：<?>

\* 含义：可以匹配任何泛型。

\* 注意事项：<?>只能用于被动匹配，不能用于泛型定义。

通配符可以设置上限、下限。

格式：

\* <? extends A>：可以匹配A类型，或者A类型的所有子类型。上限。

\* <? super B>：可以匹配B类型，或者B类型的所有父类型。下限。

**java.util.LinkedList**

这个类内部使用的不是数组，而是双向链表。

LinkedList当中特有的方法：

\* addFirst：在第一个位置添加一个元素

\* addLast：在最后一个位置添加一个元素

\* getFirst：获取第一个位置的元素

\* getLast：获取最后一个位置的元素

\* removeFirst：删除第一位置的元素

\* removeLast：删除最后一个位置的元素

\* push：压入，在第一个位置添加一个元素

\* pop：弹出，在第一个位置获取并删除一个元素

**java.util.List**

接口的特点：

\* 1. 允许重复元素。

\* 2. 有序。顺序和插入的先后顺序一样。

\* 3. 有索引值。索引从0号开始。

List有常用的三个实现类：

\* ArrayList、Vector、LinkedList

List接口当中常用的方法有（也包含Collection继承下来的）：

\* 添加元素：add

\* 删除元素：remove

\* 获取长度：size

\* 包含元素：contains

\* 清空集合：clear

\* 判断为空：isEmpty

\* 获取元素：get(int index)

**java.util.Set**

是Collection的子接口，特点：

\* 1. 不允许重复元素出现。

\* 2. **不保证先后顺序**。(父类set接口没顺序,子类可以扩展为有顺序)

\* 3. 没有索引值。

常用的实现类：

\* HashSet、LinkedHashSet、TreeSet

\* HashSet作为Set接口的实现类，特点：快。

\* 一定会满足Set接口的要求，但是性能非常快：因为底层有一个叫做“哈希表”的数据结构。

**HashSet**

**注意**：对于**HashSet**来说，**没有get**(int index)方法，**只能**使**用迭代器**或者**增强for循环**进行**遍历**。

“**哈希值**”：Hash Code，其实就是一个【int数字】，**哈希值不是内存地址值**。

\* 内存地址值的种类，比哈希值的种类多得多。所以其实哈希值并不能表示内存地址值。

\* **==运算符**对于引用类型来说，确实是地址值比较，不是哈希值比较。

\* 只不过是JVM底层在计算哈希值的时候，通常都会使用内存地址值，参与运算而已。

\* JVM规范对于hashCode()如何进行计算没有要求，不一样的JVM会采用不一样的算法，但是通常都和内存相关。

**哈希表**：其实就是就是一个数组,**默认长度为16**

桶 : 就是哈希表格当中的每一个小格

哈希冲突 : 两个对象被放到同一个桶当中

**加载因子** : 是一个**百分比,默认0.75**,一旦哈希表当中的桶被使用的比例超过了加载因子,那么将会进行"再哈希"

**再哈希** : rehash,哈希表当中**翻倍扩容,重新分类**,分类将更加细致

HashSet如何判断两个元素一样，还是不一样？

\* 1. 先看hashCode，如果哈希值不一样，那么肯定不同，所以直接放进去。

\* 2. 如果hashCode一样，继续看equals结果为false，那么仍然不一样，放进去。

\* 3. 如果hashCode一样，继续看equals结果为true，那么就认为已经存在，不能放进去。

覆盖重写hashCode方法的时候，三项原则：

\* 1. 运行期一致性：在程序运行期间，如果对象内容不变，那么hashCode不能变。重新启动了程序之后变化无所谓。

\* 2. 对于相同的对象，必须返回相同的hashCode。

\* 3. 对于不同的对象，hashCode可以不同，也可以相同。

覆盖重写equals方法的时候，五项原则：

\* 1. 自反性：自己和自己比较，一定是true。

\* 2. 对称性：a.equals(b)结果必须和b.equals(a)一样。

\* 3. 传递性：如果a和b相同，b和c相同，那么a和c也必须相同。

\* 4. 一致性：对于不变的对象，多次调用equals()方法结果必须不能变。

\* 5. 非空性：任何对象调用x.equals(null)必须得到false值。

万分**注意**：**必须同时覆盖hashCode和equals方法**。

**java.util.LinkedHashSet**

是HashSet的子类，功能更加丰富一些。

\* 区别：LinkedHashSet底层在哈希表的基础之上，又加入了一个双向链表。

\* 所以，LinkedHashSet有顺序。

\* **Set**接口**不保证有顺序**，但是**并不意味着所有的实现类都绝对不能有顺序**。

**ArrayList 集合**

常用方法：

\* 添加元素：add

\* 获取元素：get

\* 获取长度：size

\* 删除元素：

\* remove(Object obj)：按照内容进行删除（现阶段不推荐）

\* remove(int index)：按照索引进行删除（现阶段推荐）

\* 替换元素：set(int index, E element)

**list1.retainAll(list2)**

取两个交集,如果**调用的集合改变**就返回**true**,如果调用的集合**不变**就返回**false**

(如果"传入的集合"包含"调用的集合",就是false)

(如果"传入的集合"不包含"调用的集合",就是true,并将两个集合相同部分(包括空)赋给调用的集合)

**java.util.Map<K, V> : 双列集合根接口**

常用方法：

\* public V put(K key, V value)：添加一对儿数据到集合当中。

如果即将添加的键在集合当中不存在，那么put将会返回null值

如果即将添加的键在集合当中已经存在，那么put将会返回被替换的值。

\* public V get(Object key)：根据键找到对应的值。

\* public V remove(Object key)：根据键删除一对儿数据。

\* 获取尺寸长度：size

\* 清空所有数据：clear

\* 是否包含键：containsKey

\* 是否包含值：containsValue

\* 由于Map接口没有继承java.lang.Iterable接口，所以不能“**直接**”使用迭代器，或者是增强for循环。

\* Map接口当中提供一个方法，可以获取所有的键（Key）集合：public Set<K> keySet()。

\* **遍历Map集合**的步骤：

\* 1. 调用**keySet**方法获取所有键的Set集合。

\* 2. 调用Set集合的iterator方法获取迭代器。

\* 3. 使用while、hasNext和next写法来获取每一个键（Key）。

\* 4. 从Map集合当中，根据键（Key）取出值（Value）。

如果一个接口当中包含了一个内部接口，如何实现它（们）？

\* 1. 是否可以只实现外部接口，不管内部接口？可以。

\* 2. 是否可以只实现内部接口，不管外部接口？可以。（要么导包，要么使用格式：外部接口**.**内部接口。）

\* 3. 是否可以同时实现外部接口和内部接口？可以。例如implements Outer**,** Outer**.**Inner

**遍历Map集合**还可以使用**另外一种方式**，步骤：

\* 1. 调用**entrySet**方法，获取所有的Map.Entry<K, V>键值对，得到一个Set集合。

\* 2. 对Set集合调用iterator方法，获取迭代器。

\* 3. 使用hasNext、next来拿到一个单独的键值对：Map.Entry<K, V>

\* 4. 从Entry当中分别取出key和value进行使用。

\* Map接口当中有一个方法：entryKey，获取所有的键值对集合

\* Entry内部接口当中有两个方法，可以分别获取key和value：getKey、getValue

**java.util.HashMap**

Map的常用实现类。特点：

\* 1. 线程不安全。

\* 2. 键唯一，值可以不唯一。

\* 3. 允许null值存入。

\* 4. 不保证顺序。

关于null值的存储：

\* 1. 如果key为null，value不为null：可以。因为key不能重复，所以只能放最多一个null值作为key。

\* 2. 如果key不为null，value为null：可以。

\* 3. 如果key和value全都是null：也可以。但是一般没有用。

**java.util.LinkedHashMap**

是HashMap的子类，底层其实也是哈希表，但是额外添加了双向链表，从而可以保证顺序。

**Hashtable**

基本特点：

\* 1. 从JDK 1.0开始就有，目前已经被淘汰。

\* 2. Hashtable底层也是哈希表原理，但是默认尺寸为11，加载因子也是0.75。

\* 3. 线程安全，所以性能稍微慢一些。

\* 4. 不允许null值存入。

\* 5. 子类Properties仍然比较常用。

**Properties**

是Hashtable的子类，特点：

\* 1. 同样也是线程安全的。

\* 2. Properties泛型固定为<Object, Object>，所以不需要进行指定。

\* 3. Properties当中的两个泛型，从效果上会自动处理为<String, String>。当做是两个String类型。

\* 4. Properties也是Map接口的实现类，但是一般都是只用特有方法：

特有方法 :

\* setProperty(String key, String value)：设置一对儿数据

\* getProperty(String key)：根据key获取value

\* stringPropertyNames()：获取所有的key字符串，相当于是keySet的效果

重要方法 :

**load(输入流)：从IO流当中加载数据到集合里**。

**注意** : 这里文件中的**数据内容**必须**是键值对**形式的

**store**(输出流, String 描述信息)： 集合内容存储到文件中

public void **store**(输出流**,"**属性列表的描述**"**)

out - 输出流。

comments - 属性列表的描述。

Properties props = new Properties();

BufferedReader reader = new BufferedReader(new FileReader("**config.properties**"));

props.load(reader);

props.store(new FileWriter("name.txt"), "hello");

**config:配置**

**.properties**(Eclipse对此类型文件产生支持,文件内容会和.java文件一样用彩色显示)

**.properties**文件中不显示中文,中文在此类型文件中会乱码,但是可以存入中文和读取中文

**.properties**文件中**#为单行注释**

**java.util.Collections**

是一个**工具类**，提供与集合相关的常用工具方法。

\* shuffle(List<?> list)：打乱集合当中元素的顺序。

\* sort(List<?> list)：按照一定的规则进行排序。

\* public static <T> boolean **addAll**(Collection<? super T> c**,** T... elements)

**将所有指定元素添加到指定 collection 中**

**java.util.Arrays**

数组工具类常用方法：

\* public static String toString(数组)：将参数数组转换成为字符串，格式：[a, b, c]

\* public static void sort(数组)：对数组当中的元素进行排序。

\* public static int binarySearch(Xxx[] a,Xxx key) //查找元素索引

\* public static <T> List <T> **asList**(T... a)

返回一个受指定数组支持的固定大小的列表