目录

集合

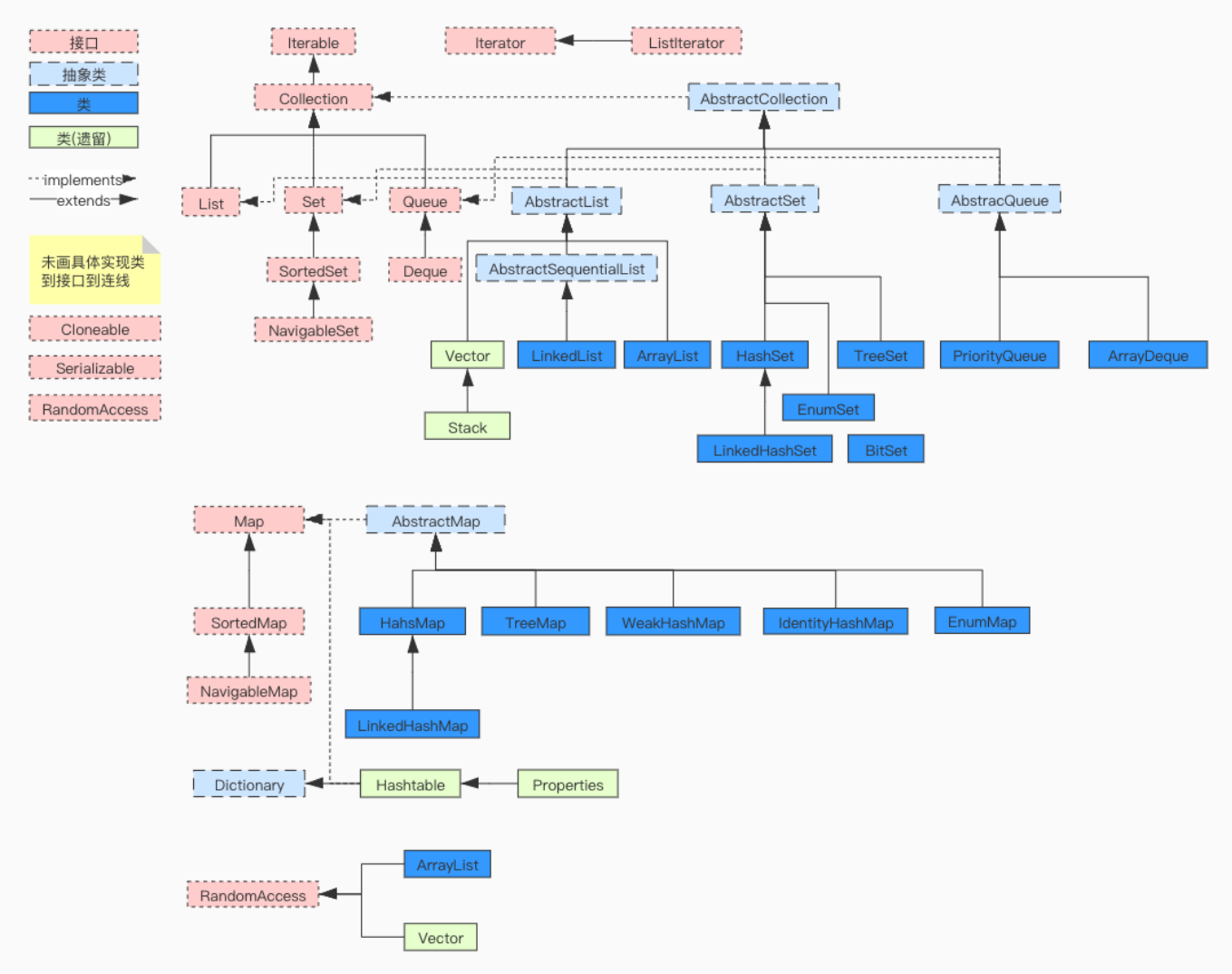
Java 语言中，数组是保存一组同类型对象的最有效的方式，但是数组是有固定大小的，如果我们在写程序时并不知道对象的确切个数时该怎么办？Java 基本类库提供容器类来解决这个问题。Java 容器类划分为两类：

- `Collection` 一个独立元素的序列，按照规则不同分为 3 类
  - `List` 按照插入顺序保存元素
  - `Set` 不能有重复元素
  - `Queue` 按照排队规则来确定对象产生的顺序
- `Map` 保存“键值对”对象，可以通过键来查找值，也称为关联数组或字典

集合实现方式：接口和实现分离

Java 集合类是通过接口（interface）和实现（implementation）相分离的方式来实现的，集合有两个基本接口：Collection 和 Map。所有元素序列对象都实现了 Collection 接口，而键值对对象则实现了 Map 接口。

通过下图，我们注意到除了接口，还有一组名字以 Abstract 开头的类，这些类是抽象类，将集合类的一些通用功能进行了封装，是为类库实现者而设计的，当我们想要实现自己的集合类时，可以直接扩展 Abstract 类，这要比实现接口要方便很多。



迭代器

集合的主要目的就是用来持有对象，而迭代器 Iterator 被设计用来遍历并选择序列中的对象。

Iterator

Iterator 接口主要包含 3 个方法：

```
interface Iterator<E> {
    // 如果存在可访问的元素，返回 true
    boolean hasNext();

    // 返回将要访问的下一个对象。如果已经达到集合的尾部，抛出 NoSuchElementException
异常
    E next();

    // 删除上次访问的对象，这个方法必须紧跟在访问一个元素之后执行。
    // 如果上次访问之后，集合已经发生变化，则抛出 IllegalStateException 异常
    void remove();
}
```

`Iterator` 迭代器是通过 `Iterable` 接口来生成，Iterable 接口定义了一个返回 Iterator 迭代器的方法：

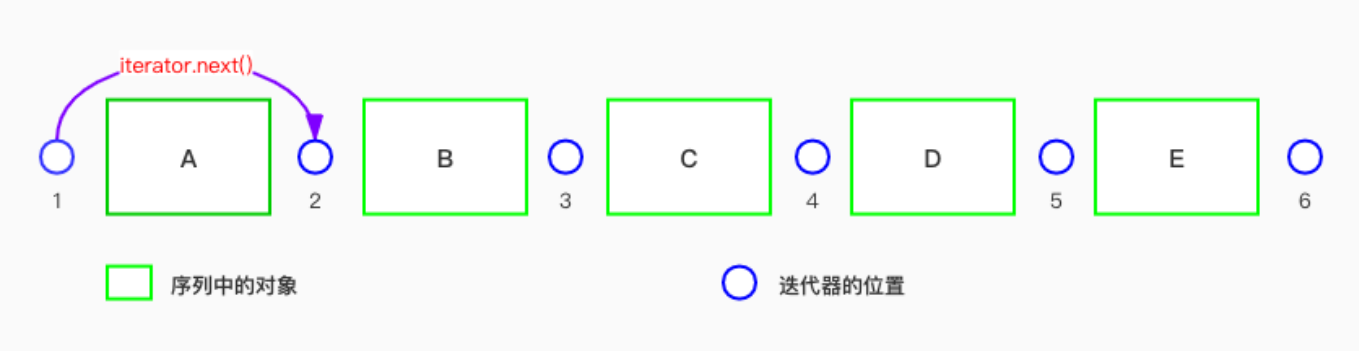
```
interface Iterable<T> {
    Iterator<T> iterator();
}
```

而集合类的基本接口 Collection 又实现了 Iterable 接口：

```
interface Collection<E> extends Iterable<E> {
    Iterator<E> iterator();
}
```

因此所以集合类对象，我们都可以使用迭代器来进行遍历（注：从严格意义上讲，Map 并不是一个集合类，但我们通常把它和集合类放在一起来讲）。从 JDK 5 开始，也可以使用 for each 循环来进行遍历操作，编译器将 for each 循环翻译为带有迭代器的循环。因此任何实现了 Iterable 接口的对象都可用于 for each 循环。

我们可以把迭代器的位置认为是在序列中两个对象之间，当调用 next 方法时，迭代器就会越过下一个对象，并返回刚刚越过的这个对象。而 remove 方法将会删除上一次调用 next 方法时返回的对象。



如上图所示：

1. 如果序列有 n 个对象，就有 n + 1 个位置可以添加新对象，这些位置与迭代器的 n + 1 个可能的位置相对应。

2. 迭代器初始状态时，在第一个对象（A）之前，如果直接调用 remove 方法，会抛 IllegalStateException 异常，因为 remove 方法依赖 next 方法，它删除的是上一次调用 next 方法越过的对象。
3. 调用 `iterator.next()` 方法时，迭代器位置从 1 跳到 2，位于第一个对象（A）和第二个对象（B）之间，并且越过第一个对象（A），因此 next 方法返回它越过的第一个对象（A）。
4. 此时再调用 `iterator.remove()` 方法，会删除上一次调用 next 方法返回的第一个对象（A）。
5. 如果再次调用 `iterator.remove()` 方法，因为上一次调用 next 方法返回的第一个对象（A）已经被删除，因此重复删除时会抛 IllegalStateException 异常。

代码示例：

```
Collection<String> collection = new ArrayList<>();
collection.add("A");
collection.add("B");
collection.add("C");
collection.add("D");
collection.add("E");

System.out.println(collection);

Iterator<String> iterator = collection.iterator();
// 从未调用 next 方法直接调用 remove 方法，因为不存在上次调用 next 返回的对象，
// 因此抛 IllegalStateException 异常。
// iterator.remove();

// 删除上次调用 next 返回的对象已经被删除
iterator.next();
iterator.remove();
System.out.println(collection);

// 上次调用 next 返回的对象已经被删除
// 重复删除抛 IllegalStateException 异常
// iterator.remove();
```

上述代码我们修改为使用迭代器遍历并删除集合中所有的对象：

```
Collection<String> collection = new ArrayList<>();
collection.add("A");
collection.add("B");
collection.add("C");
collection.add("D");
collection.add("E");

System.out.println(collection);

Iterator<String> iterator = collection.iterator();
while (iterator.hasNext()) {
    System.out.print("删除：" + iterator.next());
    iterator.remove();
    System.out.println(", 结果" + collection);
}
```

```
System.out.println(collection);

System.out.println(iterator.hasNext());
// 已经迭代器已经到达尾部，hasNext 方法返回 false
// 因此调用 next 方法取下一个对象，抛 NoSuchElementException 异常
// iterator.next();
```

注意：在获得序列的迭代器，通过迭代器对序列进行操作时，不能直接对序列进行结构修改（新增/删除），否则会抛 ConcurrentModificationException 异常；对序列进行结构修改后，必须重新获取迭代器。如下代码示例：

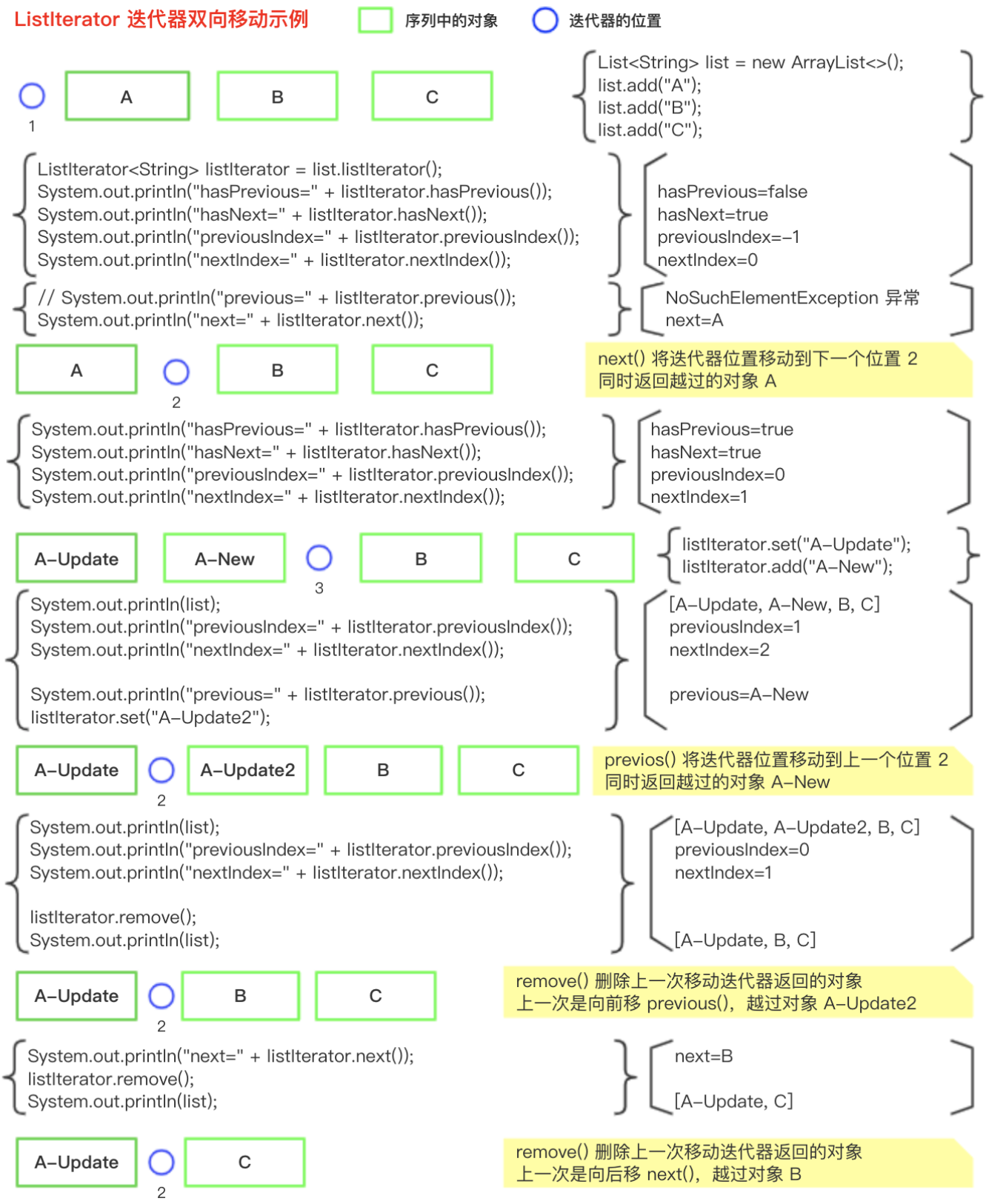
```
Collection<Node> collection = new ArrayList<>();
collection.add(new Node(1, "A"));
collection.add(new Node(2, "B"));
collection.add(new Node(3, "C"));
System.out.println(collection);

Iterator<Node> iterator = collection.iterator();

Node nodeA = iterator.next();

// 直接从序列中删除对象，属于结构修改
// collection.remove(nodeA);

// 直接向序列中添加对象，属于结构修改
// collection.add(new Node(4, "D"));

// 修改序列中已存在对象的状态，不属于结构修改
nodeA.name = "字母" + nodeA.name;

System.out.println(iterator.next());
System.out.println(collection);

// 序列发生结构修改后，需要重新获取迭代器
collection.add(new Node(5, "E"));
Iterator<Node> iterator2 = collection.iterator();
while (iterator2.hasNext()) {
    System.out.println(iterator2.next());
}
```

ListIterator

ListIterator 是 Iterator 的子类，提供了更强大的功能，但只能用于各种 List 类的访问。Iterator 只能向前移动，而 ListIterator 可以双向移动，还可以产生相对于迭代器在列表中指向的当前位置的前一个和后一个元素的索引，并且可以使用 set 方法替换它访问过的最后一个对象。

```
interface ListIterator<E> extends Iterator<E> {
    // 用新对象取代 next 或 previous 上一次访问的对象。
    // 如果在 next 或 previous 上一次调用之后，列表结构被修改了，则抛出
```

```
IllegalStateException 异常
    void set(E e);

    // 在当前位置前添加一个对象
    void add(E e);

    // 当反向迭代列表时，还有可访问的对象，返回 true
    boolean hasPrevious();

    // 返回前一个对象。如果已经达到列表的头部，抛出 NoSuchElementException 异常
    E previous();

    // 返回下一次调用 next 方法时将返回的对象索引
    int nextIndex();

    // 返回下一次调用 previous 方法时将返回的对象索引
    int previousIndex();
}
```



上图代码如下：

```
List<String> list = new ArrayList<>();
list.add("A");
list.add("B");
list.add("C");
System.out.println(list);
```

```
ListIterator<String> listIterator = list.listIterator();
System.out.println("hasPrevious=" + listIterator.hasPrevious());
System.out.println("hasNext=" + listIterator.hasNext());
System.out.println("previousIndex=" + listIterator.previousIndex());
System.out.println("nextIndex=" + listIterator.nextIndex());

// System.out.println("previous=" + listIterator.previous());
System.out.println("next=" + listIterator.next());

System.out.println("hasPrevious=" + listIterator.hasPrevious());
System.out.println("hasNext=" + listIterator.hasNext());
System.out.println("previousIndex=" + listIterator.previousIndex());
System.out.println("nextIndex=" + listIterator.nextIndex());

listIterator.set("A-Update");
listIterator.add("A-New");

System.out.println(list);
System.out.println("previousIndex=" + listIterator.previousIndex());
System.out.println("nextIndex=" + listIterator.nextIndex());

System.out.println("previous=" + listIterator.previous());
listIterator.set("A-Update2");

System.out.println(list);
System.out.println("previousIndex=" + listIterator.previousIndex());
System.out.println("nextIndex=" + listIterator.nextIndex());

listIterator.remove();
System.out.println(list);

System.out.println("next=" + listIterator.next());
listIterator.remove();
System.out.println(list);
```

综上所述，迭代器需要注意的一些特性：

- List 是有序的，而 Set 是无序的，它们都实现了 Iterable 接口，而依赖于位置的 add 方法由迭代器负责。因此，在 Iterator 接口中没有 add 方法，而专门用于有序列表 List 的 ListIterator 接口有 add 方法。
- add 方法只依赖于迭代器的位置，在当前迭代器位置之前添加一个对象
- remove 方法依赖于迭代器的状态，删除上一次移动迭代器位置返回的对象（上一次移动迭代器越过的对象，next/previous方法）
- set 用一个新元素取代上一次移动迭代器位置返回的对象（上一次移动迭代器越过的对象，next/previous 方法）

具体的集合

本节内容我们开始详细介绍集合类中每一个具体实现类的用途。以 Map 结尾的类实现了 Map 接口，除此之外其他所有类都实现了 Collection 接口。

集合类型	描述

集合类型	描述
ArrayList	可以动态增长和缩减的索引序列
LinkedList	可以在任意位置进行高效地插入和删除操作的有序序列
HashSet	没有重复元素的无序集
TreeSet	有序集
EnumSet	包含枚举类型值的集
LinkedHashSet	可以记住元素插入顺序的集
PriorityQueue	允许高效删除最小元素的集合
ArrayDeque	用循环数组实现的双端队列
HashMap	存储键/值关联的映射表
TreeMap	根据键有序排列的键/值映射表
EnumMap	键值属于枚举类型的映射表
LinkedHashMap	可以记住键/值对添加顺序的映射表
WeakHashMap	一种其值不再被使用时可以被垃圾回收器回收的映射表
IdentityHashMap	一种使用 == 而不是用 equals 比较键值是否相等的映射表

`java.util.Collection` 接口主要方法：

- `Iterator<E> iterator()` 返回一个用于访问集合中每个元素的迭代器
- `int size()` 返回当前存储在集合中的元素个数
- `boolean isEmpty()` 如果集合中没有元素，返回 true
- `boolean contains(Object o)` 如果集合中包含了一个与 o 相等的对象，返回 true
- `boolean containsAll(Collection<?> c)` 如果这个集合包含 c 集合中的所有元素，返回 true
- `boolean add(E e)` 将一个元素添加到集合中
- `boolean addAll(Collection<? extends E> c)` 将 c 集合中的所有元素添加到这个集合
- `boolean remove(Object o)` 从这个集合中删除等于 o 的对象，如果有匹配的对象被删除，返回 true
- `boolean removeAll(Collection<?> c)` 从这个集合中删除 c 集合中存在的所有元素。如果有匹配的对象被删除，返回 true
- `void clear()` 从这个集合中删除所有的元素
- `boolean retainAll(Collection<?> c)` 从这个集合中删除所有与 c 集合中的元素不同的元素，如果有对象被删除，返回 true
- `Object[] toArray()` 返回这个集合的对象数组
- `<T> T[] toArray(T[] a)` 返回这个集合的对象数组。如果 a 足够大，就将集合中的元素填入这个数组，剩余空间填补 null；否则，分配一个类型为 T 的新数组，长度等于集合的大小，并添入集合中的元素

`java.util.Map` 接口主要方法：

- `V get(Object key)` 获取与键对应的值；如果为找到返回 null，键可以为 null

- `V put(K key, V value)` 将键与对应的值关系插入映射表中。如果键已经存在，新的对象取代这个键对应的旧对象。这个方法返回键对应的旧值，如果没有旧值则返回 null
- `void putAll(Map<? extends K, ? extends V> m)` 将给定的映射表中的所有条目添加到这个映射表中
- `boolean containsKey(Object key)` 如果映射表中存在这个键，返回 true
- `boolean containsValue(Object value)` 如果映射表中存在这个值，返回 true
- `Set<Map.Entry<K, V>> entrySet()` 返回 Map.Entry 对象的集视图，即映射表中的键/值。可以从这个集中删除元素，同时也将自动从映射表中删除它们。但是，不能添加任何元素
- `Set<K> keySet()` 返回映射表中所有键的集视图。可以从这个集中删除元素，同时也将自动从映射表中删除了它们。但是，不能添加任何元素
- `Collection<V> values()` 返回映射表中所有值的集合视图。可以从这个集中删除元素，同时也将自动从映射表中删除了它们。但是，不能添加任何元素

`java.util.Map.Entry` 接口主要方法：

- `K getKey()` 返回这个条目的键
- `V getValue()` 返回这个条目的值
- `V setValue(V value)` 设置该键在映射表中对应的新值，并返回旧值

List

List 是一个有序序列，有两种类型的 List：

- `ArrayList` 内部使用数组实现，长于随机访问元素，但是在中间位置插入和删除元素比较慢。
- `LinkedList` 内部使用双向链表实现，随机访问速度慢，但是在中间位置插入和删除元素代价较低。

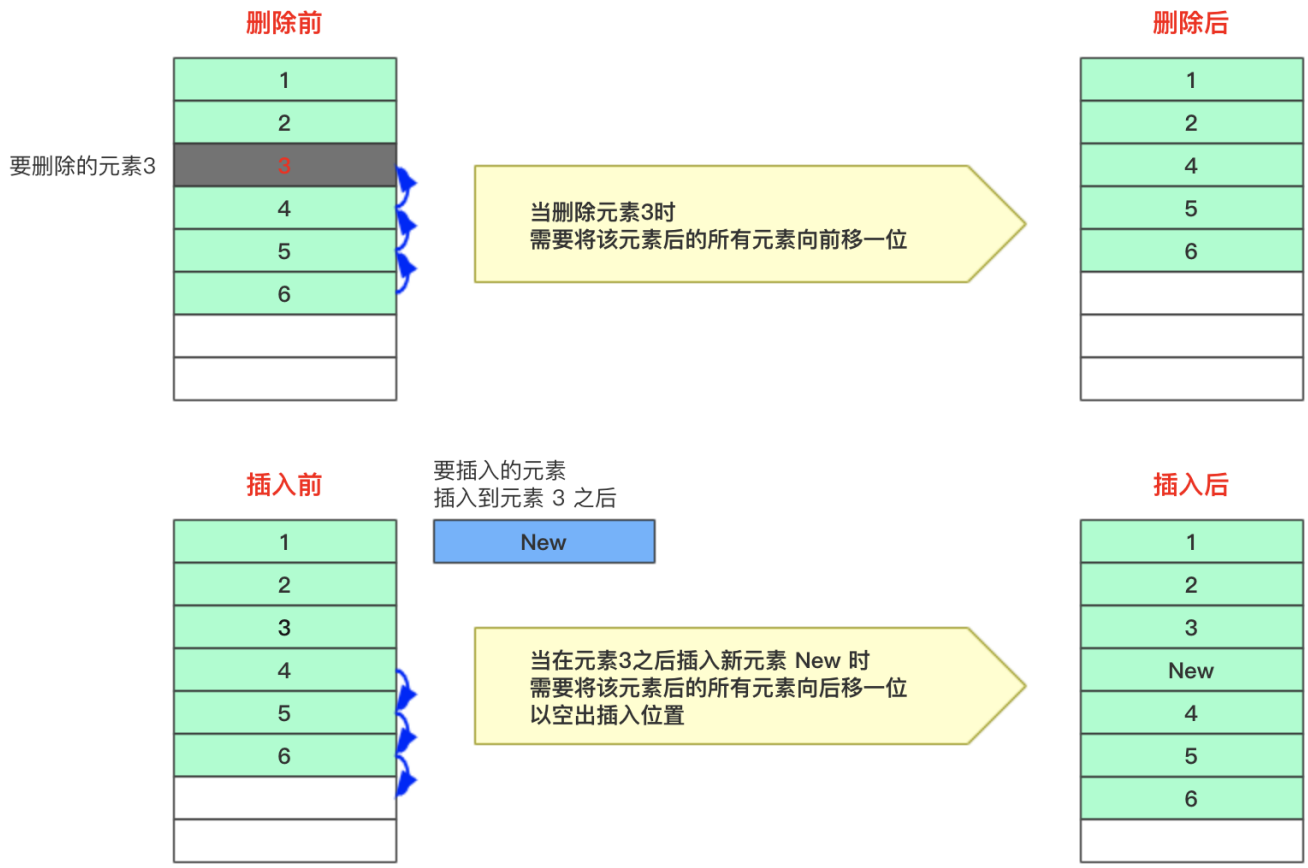
`java.util.List` 接口中主要的方法：

- `ListIterator<E> listIterator()` 返回列表迭代器，以便用来访问列表中的元素
- `ListIterator<E> listIterator(int index)` 返回列表迭代器，以便用来访问列表中的元素；第一次调用该迭代器 next 方法返回元素的索引是参数 `index`。
- `void add(int index, E element)` 在指定位置添加一个元素
- `boolean addAll(int index, Collection<? extends E> c)` 将某个集合中的所有元素添加到指定位置
- `E remove(int index)` 删除指定位置的元素并返回这个元素
- `boolean remove(Object o)` 删除和指定元素相等的元素（equals比较返回true），存在且删除成功返回 true
- `E get(int index)` 获取指定位置的元素
- `E set(int index, E element)` 用新元素取代指定位置的元素，并返回原来那个元素
- `int indexOf(Object o)` 返回与指定元素相等的元素在列表中第一次出现的位置，没如果没有则返回 -1
- `int lastIndexOf(Object o)` 返回与指定元素相等的元素在列表中最后一次出现的位置，没如果没有则返回 -1

ArrayList

数组列表 ArrayList 内部使用数组实现，而数组使用下标进行访问时速度很快；而在中间位置插入和删除元素时，因为要移动插入/删除位置以后的元素，因此速度比较慢。

ArrayList 插入/删除操作



ArrayList 中的元素，它有两种访问方式：

- 使用迭代器 Iterator
- 使用 get 和 set 方法随机快速的访问每个元素

扩容机制

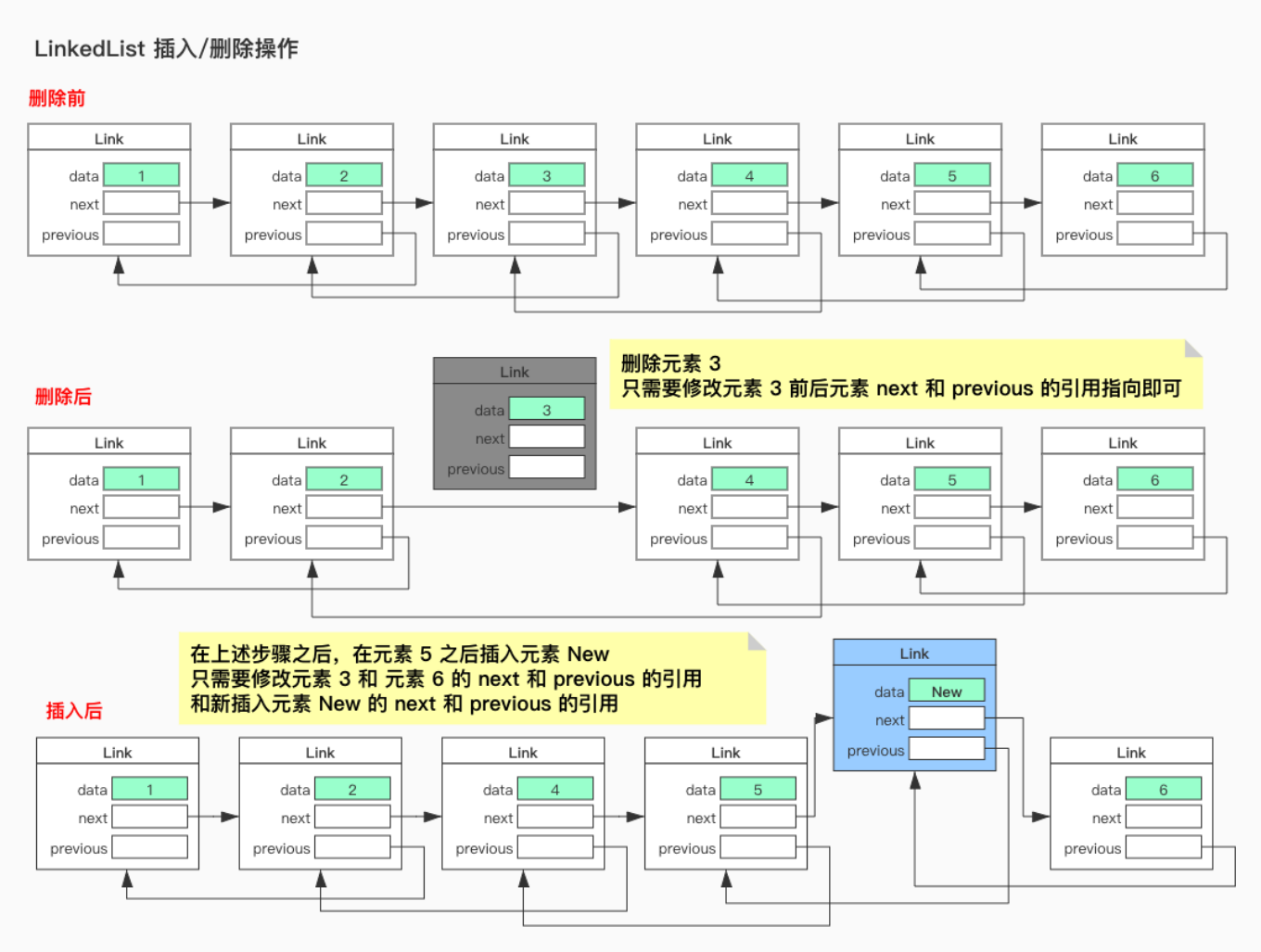
- 创建 ArrayList 时可以指定初始容量大小，如果未指定则创建一个空数组，长度为0
- 如果存储长度大于容量大小时，自动进行扩容，每次扩容到原来的 1.5 倍。相关代码如下：

  int newCapacity = oldCapacity + (oldCapacity >> 1); elementData = Arrays.copyOf(elementData, newCapacity);

ArrayList 中的方法不是同步的，如果有线程安全需要可以使用 Vector。Vector 每次扩容到原来的 2 倍。

LinkedList

链表 LinkedList 内部使用双向链表实现（Java 语言中，所有链表都是双向链接（doubly linked）），从链表中间删除或插入一个元素代价很低：删除元素只需要修改位于删除元素前后的元素的 next 和 previous 引用指向即可；插入元素只需要修改新插入元素和插入位置前后的元素的 next 和 previous 引用指向。



和 ArrayList 一样，除了使用迭代器对链表进行遍历，也可以使用 for 循环方式，如下所示：

```
for (int i = 0; i < list.size();  i++) {
    do something with list.get(i);
}
```

链表虽然也实现了随机访问，但是不支持快速随机访问，上述代码每次查找元素都会从列表的头部开始重新开始搜索（查看第 n 个元素时，需要从头开始越过 n - 1 个元素）。LinkedList 对象根本不做任何缓存；仅到索引大于 size() / 2 就从列表尾端开始搜索元素。

我们可以通过是否实现了 `RandomAccess` 接口来判断对象是否支持快速随机访问。

使用链表唯一的理由是尽可能地减少在列表中间插入或删除元素所付出的代价。

ArrayList 长于随机快速访问，而 LinkedList 长于在序列中间插入/删除元素，下边我们通过示例来验证一下。

题目：分别创建一个ArrayList/LinkedList列表，并向列表里插入10000个元素 A，然后进行如下验证：

1. 在索引为 10 的位置处插入10000个 B，并计算耗时。
2. 分别使用迭代器和for循环随机访问方式遍历整个列表，并计算耗时。

代码如下：

```
private static void testListOpTime(List list, int times) {
    for (int i = 0; i < times; i++) {
        list.add("A");
    }

    // 在索引为 10 的位置处插入10000个 B，并计算耗时。
    long start = System.currentTimeMillis();
    for (int i = 0; i < times; i++) {
        list.add(10, "B");
    }

    System.out.printf("%s：在索引为 10 的位置处插入%d个 B，耗时%dms\n",
            list.getClass().getSimpleName(),
            times,
            System.currentTimeMillis() - start);

    // 遍历 - for循环随机访问方式
    start = System.currentTimeMillis();
    for (int i = 0; i < list.size(); i++) {
        list.get(i);
    }
    System.out.printf("%s：for循环随机访问方式遍历，耗时%dms\n",
            list.getClass().getSimpleName(),
            System.currentTimeMillis() - start);

    // 遍历 - 迭代器方式
    start = System.currentTimeMillis();
    ListIterator listIterator = list.listIterator();
    while (!listIterator.hasNext()) {
        listIterator.next();
    }
    System.out.printf("%s：迭代器方式遍历，耗时%dms\n",
            list.getClass().getSimpleName(),
            System.currentTimeMillis() - start);
}

public static void main(String[] args) {
    testListOpTime(new ArrayList(), 10000);
    testListOpTime(new LinkedList(), 10000);
}
```

执行结果如下（每次执行结果会存在细微差异）：

```
ArrayList：在索引为 10 的位置处插入10000个 B，耗时16ms
ArrayList：for循环随机访问方式遍历，耗时1ms
ArrayList：迭代器方式遍历，耗时0ms
LinkedList：在索引为 10 的位置处插入10000个 B，耗时2ms
LinkedList：for循环随机访问方式遍历，耗时160ms
LinkedList：迭代器方式遍历，耗时0ms
```

在不知道 List 确切类型的前提下如果需要遍历，优先使用迭代器，也可以使用 `instanceof RandomAccess` 来判断 list 实际类型是否支持快速随机访问，如下代码所示：

```
if (list instanceof RandomAccess) {
    for (int i = 0; i < times; i++) {
        list.get(i);
    }
} else {
    Iterator iterator = list.iterator();
    while (!iterator.hasNext()) {
        iterator.next();
    }
}
```

Set

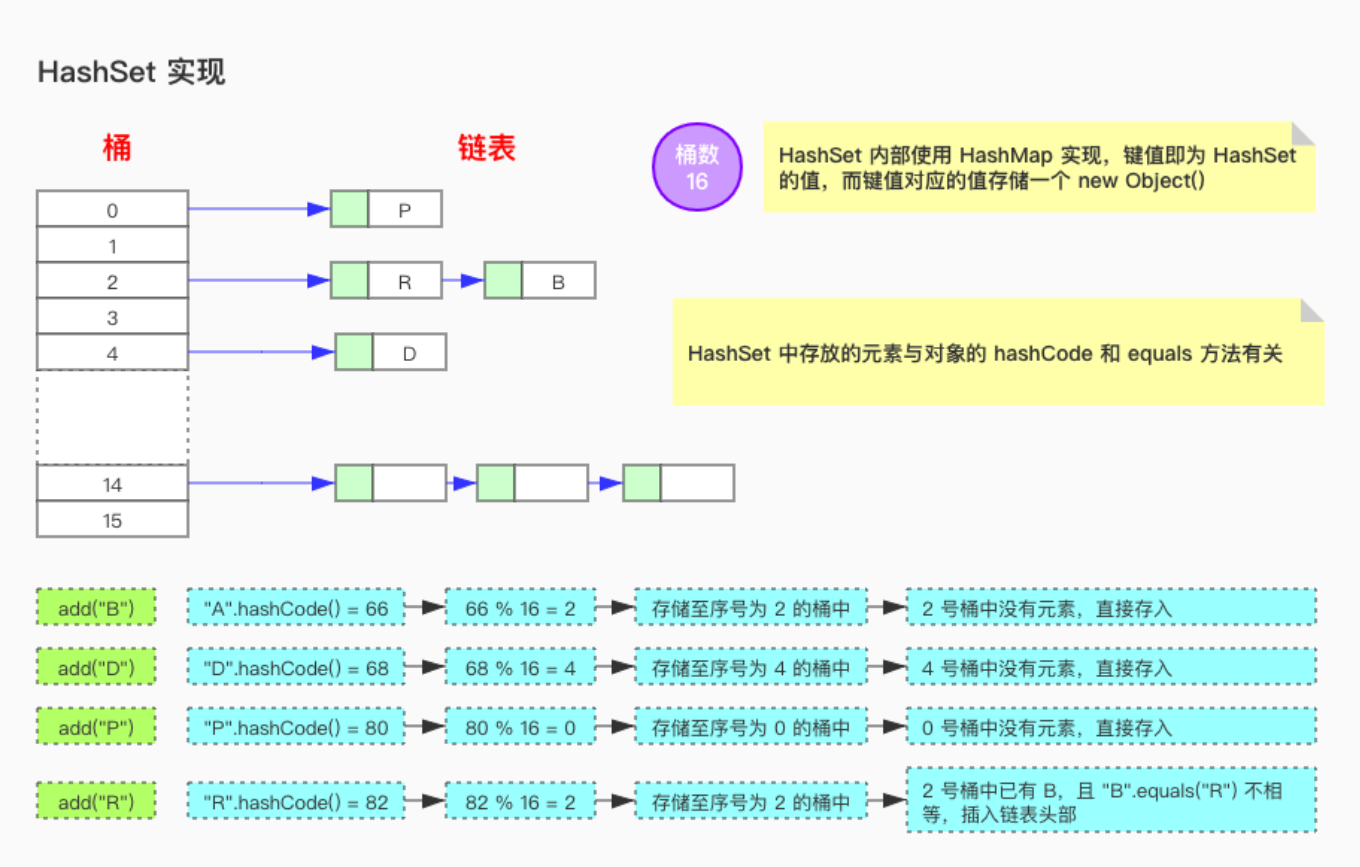
集 Set 不保存重复的元素，它最常用的就是测试归属，因此查找是 Set 最重要的操作。Set 基于对象的值来确定归属性。

Set 具有和 Collection 完全一样的接口，没有任何额外的功能。

HashSet

HashSet 内部使用 HashMap 来实现，HashMap 的键值存储的即是 HashSet 的值，而键值对应的值存放了一个静态对象 `PRESENT`。主要代码如下：

```
public class HashSet<E> extends AbstractSet<E> implements Set<E> {
    // 用于存放数据，键值即为 Set 集的元素
    private transient HashMap<E,Object> map;

    private static final Object PRESENT = new Object();

    public HashSet() {
        map = new HashMap<>();
    }

    public boolean add(E e) {
        return map.put(e, PRESENT)==null;
    }

    public Iterator<E> iterator() {
        return map.keySet().iterator();
    }
}
```



TreeSet

树集 TreeSet 是一个有序集合（sorted collection）。可以以任意顺序将元素插入到集合中，在遍历时，每个元素自动按照排序后的顺序呈现。

将一个元素添加到 TreeSet 中比添加到 HashSet 中慢，但是要比将元素添加到数组或链表的正确位置上要快很多。TreeSet 可以自动的对元素进行排序。

TreeSet 内部使用 TreeMap 来实现，TreeMap 的键值存储的即是 TreeSet 的值，而键值对应的值存放了一个静态对象 PRESENT。主要代码如下：

```
public class TreeSet<E> extends AbstractSet<E> implements NavigableSet<E>
{
    private transient NavigableMap<E,Object> m;

    private static final Object PRESENT = new Object();

    TreeSet(NavigableMap<E,Object> m) {
        this.m = m;
    }

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

    public TreeSet(Comparator<? super E> comparator) {
        this(new TreeMap<>(comparator));
    }
```

```
    public boolean add(E e) {
        return m.put(e, PRESENT)==null;
    }

    public Iterator<E> iterator() {
        return m.navigableKeySet().iterator();
    }
}
```

TreeSet 通过 `Comparable` 接口的 `compareTo` 方法来进行排序：

- a 与 b 相等时：a.compareTo(b) == 0
- 排序后 a 位于 b 之前：a.compareTo(b) < 0
- 排序后 a 位于 b 之后：a.compareTo(b) > 0

因此如果要插入自定义对象到 TreeSet，就必须实现 Comparable 接口自定义排列顺序，因为在 Object 中没有 compareTo 接口的默认实现。

然而使用 Comparable 接口定义排序也有其局限性，对于给定的类，只能实现这个接口一次，也就是说只能有一种排序。如果在一个集合中需要按照用户注册信息进行排序，而在另一个集合中却要按用户年龄排序，应该怎么办?

这种情况下，我们可以通过创建 TreeSet 时将 Comparator 对象传给它的构造器告诉它排序的比较方法，对于设置了 Comparator 对象的 TreeSet，向里添加的对象不需要实现 Comparable 接口。

```
interface Comparator<T> {
    int compare(T a, T b);
}
```

排序规则和上边 compareTo 一样，a 位于 b 之前 compare 方法返回负数；a 和 b 相等返回 0；否则返回正数。

LinkedHashSet

链接散列集 LinkedHashSet 继承自 HashSet，在使用哈希桶的方式存放元素的同时，使用链表的方式来记录插入元素的顺序。

EnumSet

枚举集 EnumSet 是一个枚举类型元素集的高效实现。由于枚举类型只有有限个实例，所以 EnumSet 内部使用位序列实现，如果对应的值在集中，则相应的位被置为 1。

Queue

队列是一种“先进先出”的数据结构，可以在队列尾部添加元素，在队列的头部删除元素。Java 6 中引入了 Deque 接口，并由 ArrayDeque 和 LinkedList 类实现，这两个类都提供了双端队列。队列主要有以下两种：

- 队列，可以在尾部添加一个元素，在头部删除一个元素。
- 双端队列，有两个端头的队列，可以在头部和尾部同时添加或删除元素

注： 队列和双端队列都不支持在队列中间添加元素。

`java.util.Queue` 接口主要方法：

- `boolean add(E e)`
- `boolean offer(E e)`

  如果队列没满，将给定元素添加到队列尾部并返回 true。如果队列满了 add 方法抛 IllegalStateException 异常，offer 方法返回 false

- `E remove()`
- `E poll()`

  如果队列不空，删除并返回这个队列头部的元素。如果队列是空 remove 方法抛 NoSuchElementException 异常，poll 方法返回 null

- `E element()`
- `E peek()`

  如果队列不空，返回队列头部的元素，但不删除。如果队列为空 element 方法抛 NoSuchElementException 异常，peek 方法返回 null

`java.util.Deque` 接口主要方法：

- `void addFirst(E e)`
- `void addLast(E e)`
- `boolean offerFirst(E e)`
- `boolean offerLast(E e)`

  将给定的对象添加到双端队列的头部或尾部。如果队列满了，addFirst/addLast 方法抛 IllegalStateException 异常，offerFirst/offerLast 方法返回 false

- `E removeFirst()`
- `E removeLast()`
- `E pollFirst()`
- `E pollLast()`

  如果队列不空，删除并返回队列头部的元素。如果队列为空，removeFirst/removeLast 方法抛 NoSuchElementException 异常，pollFirst/pollLast 方法返回 null

- `E getFirst()`
- `E getLast()`
- `E peekFirst()`

- `E peekLast()`

  如果队列非空，返回队列头部的元素，但不删除。如果队列为空，getFirst/getLast 方法抛 NoSuchElementException 异常，peekFirst/peekLast 方法返回 null

ArrayDeque

ArrayDeque 是使用循环数组实现的双端队列，可以在队列两边插入和删除元素。循环数组是有界集合，容量有限，当达到容量限制时进行自动扩容，每次扩容原来的 1 倍。

LinkedList

LinkedList 也实现了 Deque 接口，是双端队列的链表实现方式。它的性能要比 ArrayDeque 低。

PriorityQueue

优先级队列（priority queue）中的元素可以按任意的顺序插入，却总是按照排序的顺序进行检索。

- remove 方法总是会获得当前优先级队列中最小的元素
- 当使用迭代的方式处理时，则未进行排序

优先级队列使用堆（heap）这种数据结构，它是一个可以自我调整的二叉树，对树执行添加（add）和删除（remove）操作，可以让最小的元素移动到根，而不必对元素进行排序。

和 TreeSet 一样，优先级队列可以保存实现了 Comparable 接口的类对象，也可以保存在构造器中提供比较器的对象。

使用优先级队列的典型示例是任务调度。

Map

映射表（map）用来存放键/值对，根据键就能查找到值。Java 类有映射表的两个通用实现：HashMap 和 TreeMap，他们都实现了 Map 接口。

散列映射表（HashMap）对键进行散列，散列比较函数只能作用于键，与键关联的值不能进行散列比较。树映射表（TreeMap）用键的整体顺序对元素进行排序，并将其组织成搜索树。

键必须是唯一的，不能对同一个键存放两个值。如果对同一个键两次调用 put 方法，第二个的值会取代第一个值。

映射表本身并不是一个集合，但是我们可以获得映射表的视图，这是一组实现了 Collection 接口对象，或者它的子接口的视图。有 3 个视图：

1. 键集 `Set<K> keySet()`
2. 值集合（不是集） `Collection<V> values()`
3. 键/值对集 `Set<Map.Entry<K,V>> entrySet()`

注意：keySet 既不是 HashSet，也不是 TreeSet，而是一个实现了 Set 接口的其他类的对象。因此可以像使用集一样使用 keySet。

HashMap

链表和数组都是有序的元素序列，如果我们不在意元素的顺序，只关心查找元素的速度，有一种数据结构可以实现，这就是散列表。

散列表（hash table）为每一个对象计算一个整数值，这个整数值称为散列码（hash code）。散列码是由对象的实例域产生的一个整数。

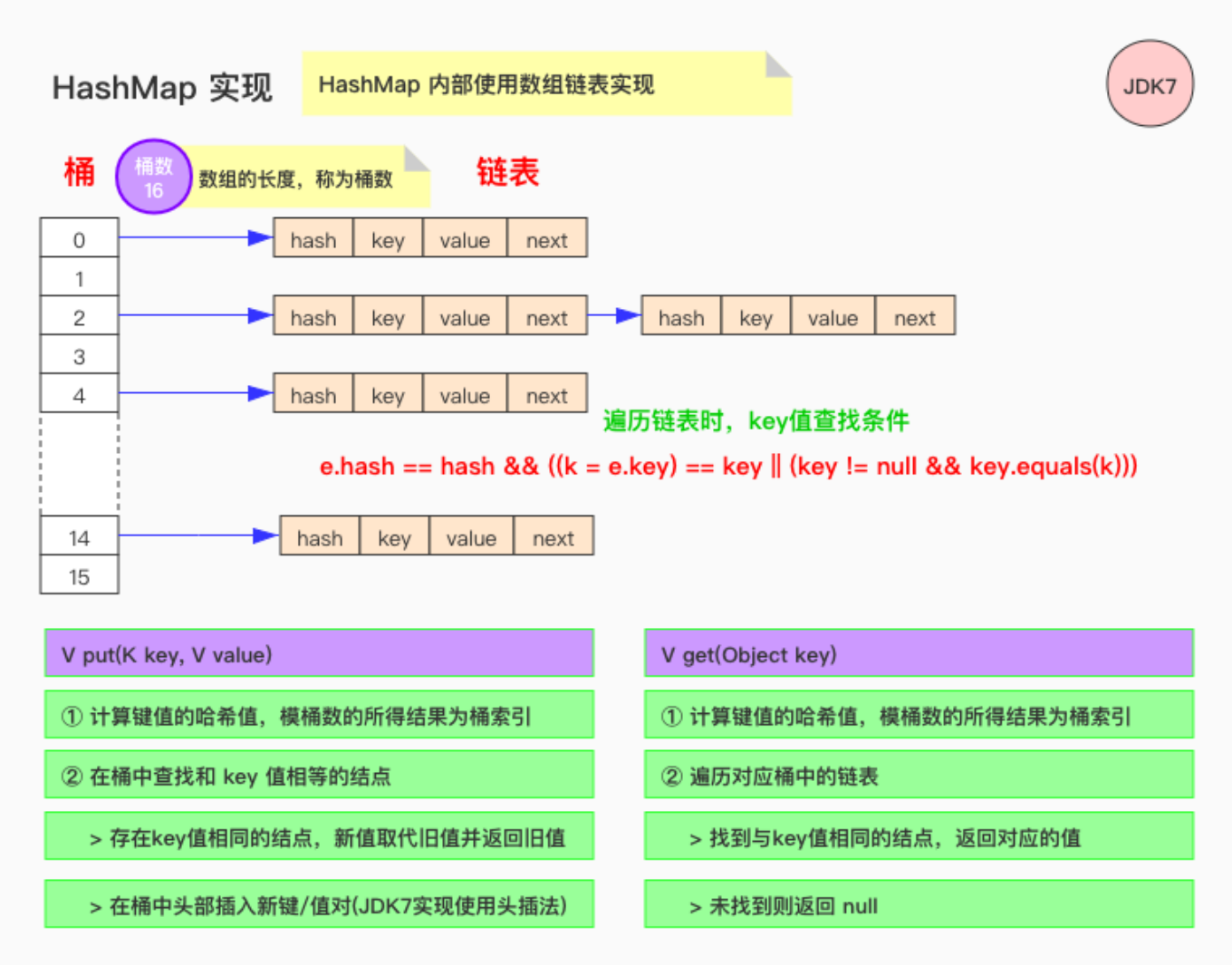
`Object` 默认的计算对象散列码的方法返回的是对象的地址。因此自定义类时，需要实现这个类的 hashCode 方法，并且 hashCode 方法应该与 equals 方法兼容，当 `a.equals(b) == true` 时，a 与 b 的散列码必须相同（`a.hashCode() == b.hashCode()`）。

在 Java 语言中，散列表用链表数组的方式实现，每个列表被称为桶（bucket），列表长度称为桶数，而桶由数组组成。要查找或存放对象时，先计算对象的散列码，然后与桶的总数取余，所得的结果就是查找或存放对象在桶中的索引。如果是查找对象，则将查找对象与桶中的所有对象进行比较（使用 `equals` 方法），如果存在则返回，否则返回 null；如果是保存对象，桶中没有其他元素时直接插入到桶中，否则用新对象与桶中的所有对象进行比较，查看对象是否已经存在。如果桶已经被占满了，这种现象称为散列冲突（hash collision）。

桶数和散列表的运行性能有关，如果知道最终要插入散列表的元素个数，就可以设置桶数，通常设置为预计元素个数的75% ~ 150%。

当散列表太满，就需要再散列（rehashed）。再散列是指创建一个桶数更多的表，并将所有元素插入到新表中，然后丢弃原来的表。

我们根据装填因子（load factor）来判断散列表是否太满，也即是何时进行再散列。例如默认装填因子为0.75，当表中超过75%的位置已填入元素，这个表就会用双倍的桶数自动的进行再散列。



注：上图为 JDK7 中 HashMap 的实现。key 值为 null 时程序做了特殊处理，key 值为 null 时哈希值为 0， 放在索引为 0 的哈希桶中。

HashMap 在 JDK8 中的实现做了如下改进：

- 插入新结点时使用尾插法，将新增结点追加到链表末尾
- 对链表做了优化，当链表较长时将链表转换为红黑树，以提高查询效率（链表长度大于 TREEIFY_THRESHOLD 时转换为红黑树）
- 链表结点的定义由 Entry 类改为 Node 类，但实现基本一致

TreeMap

WeakHashMap

弱散列映射表 WeakHashMap

LinkedHashMap

链接散列映射表使用哈希桶的方式存放元素的同时，使用链表来记录访问顺序。链接散列映射表将用访问顺序，而不是插入顺序，对映射表进行迭代。每次调用 get 或 put，受影响的元素将从前的位置删除，并放到链表的尾部。

访问顺序可以用于实现高速缓存的“最近最少使用”原则。

EnumMap

EnumMap 是一个键类型为枚举类型的映射表。

IdentityHashMap

标识散列映射表 IdentityHashMap 这个类，键的散列值使用 `System.identityHashCode` 方法计算，而不是使用自己的 hashCode 方法，并且在对两个对象进行比较时使用 `==`，而不是 `equals`。

不同的键对象即使内容相同，也会被视为不同的对象。因此可以用来实现对象遍历，例如对象序列化。

工具类

- Arrays
- Collections

. 子范围 . 不可修改的视图 . 同步视图

遗留的集合

. 哈希表 Hashtable . 枚举 Enumeration . 属性映射表 Properties . 栈 Stack . 位集 BitSet