> 关于
>
> 目前网络上有许多Java相关面试题的总结，但基本上都是差不多的题目，以下文章内容中的题目参考整理自互联网，其中部分是由我查阅书籍按照自己的理解修改后做出的答案，并精心挑选出来的一些面中率高、有价值的题目，非常基础的题目或是简单易回答的题目就省略了。可堪称Java面试必背系列。

为什么要使用集合?

通常，当我们需要保存`一组数据`的时候，我们会选择使用数组，需要注意的是数组保存的必须是`同一类型`的一组数据。同时，使用数组存储对象还具有一定的弊端，比如数组一旦声明之后，长度不可变；其次，声明数组时的数据类型也决定了该数组存储的数据的类型；而且，数组存储的数据是有序的、可重复的，特点单一；最后就是数组还要求在内存中地址是连续的。

因为我们在实际开发中，存储的数据的类型是多种多样的，此时，数组就不再满足我们在开发过程中的需求。于是，就出现了“集合”，集合同样也是用来存储多个数据的，但是集合提高了数据存储的灵活性，Java 集合不仅可以用来存储不同类型不同数量的对象，还可以保存具有映射关系的数据。

如何选用集合?

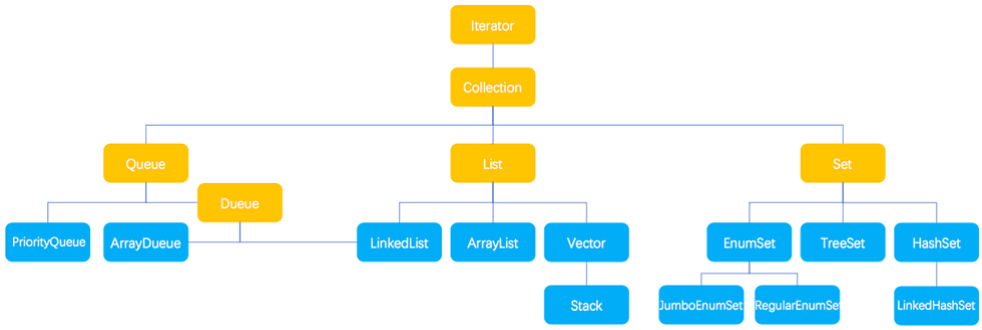
在特定需求背景下我们应主要根据集合的特点来选用合适的集合类，主要是根据对保存在集合中数据的常用操作去选择合适的数据结构对应的实现集合类。比如我们需要根据键值获取到元素值时就选用 `Map` 接口下的集合，需要排序时选择 `TreeMap`,不需要排序时就选择 `HashMap`,需要保证线程安全就选用 `ConcurrentHashMap`。

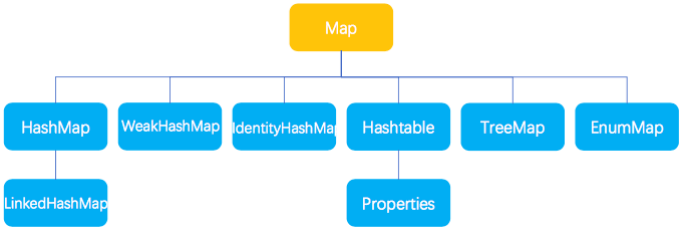
当我们只需要存放元素值时，就选择实现`Collection` 接口的集合，需要保证元素唯一时选择实现 `Set` 接口的集合比如 `TreeSet` 或 `HashSet`，不需要就选择实现 `List` 接口下的比如 `ArrayList` 或 `LinkedList`，然后再根据实现这些接口的集合的特点来选用。

简单说一下Java 集合框架，并画出大致框架结构图。

Java 集合， 也叫作容器，主要是由两大接口派生而来：一个是 `Collection`接口，主要用于存放单一元素（或是叫单列元素）；另一个是 `Map` 接口，主要用于存放键值对元素。对于 `Collection` 接口，下面又有三个主要的子接口：`List`、`Set` 和 `Queue`。

Java 集合框架如下图所示：





简单说说 List, Set, Queue, Map 四者的主要区别?

- `List`: 存储的元素是有序的、可重复的，可以存储多个NULL。
- `Set`: 存储的元素是无序的、不可重复的，仅可以存储一个NULL。
- `Queue`: 按特定的排队规则来确定先后顺序，存储的元素是有序的、可重复的。
- `Map`: 使用键值对（key-value）存储，key 是无序的、不可重复的，value 是无序的、可重复的，每个键最多映射到一个值。

大概总结说下集合框架中常用集合类的底层数据结构。

List

- **Arraylist**： `Object[] 数组`
- **Vector**： `Object[] 数组`
- **LinkedList**： `双向链表` (JDK1.6 之前为循环链表，JDK1.7 取消了循环)

Set

- **HashSet**(无序，唯一): 基于 `HashMap` 实现的，底层采用 `HashMap` 来保存元素
- **LinkedHashSet**: LinkedHashSet 是 HashSet 的子类，并且其内部是通过 `LinkedHashMap` 来实现的。
- **TreeSet**(有序，唯一): `红黑树` (自平衡的排序二叉树)

Queue

- **PriorityQueue**: `Object[] 数组来实现的二叉堆`
- **ArrayQueue**: `Object[] 数组 + 双指针`

Map

- **HashMap**： JDK1.8 之前 HashMap 由`数组+链表`组成的，数组是 HashMap 的主体，链表则是主要为了解决哈希冲突而存在的（“拉链法”解决冲突）。JDK1.8 以后在解决哈希冲突时有了较大的变化，当链表长度大于阈值（默认为 8）（将链表转换成红黑树前会判断，如果当前数组的长度小于 64，那么会选择先进行数组扩容，而不是转换为红黑树）时，将链表转化为红黑树，以减少搜索时间，所以在JDK1.8之后HashMap底层使用了`数组+链表+红黑树`这种数据结构。
- **LinkedHashMap**： LinkedHashMap 继承自 HashMap，所以它的底层仍然是基于拉链式散列结构即由`数组和链表或红黑树`组成。另外，LinkedHashMap 在上面结构的基础上，增加了一条`双向链表`，使得上面的结构可以保持键值对的插入顺序。同时通过对链表进行相应的操作，实现了访问顺序相关逻辑。
- **Hashtable**： `数组+链表`组成的，数组是 Hashtable 的主体，同样链表则是主要为了解决哈希冲突而存在的。

- **TreeMap**：`红黑树`（自平衡的排序二叉树）

Java中的容器，线程安全和线程不安全的分别有哪些?

java.util包下的集合类大部分都是线程不安全的，例如我们常用的`HashSet`、`TreeSet`、`ArrayList`、`LinkedList`、`ArrayDeque`、`HashMap`、`TreeMap`，这些都是线程不安全的集合类，但是它们的**优点是性能好。**

java.util包下也有线程安全的集合类，例如`Vector`、`Hashtable`、Stack、Enumeration。这些集合类都是比较古老的API，虽然实现了**线程安全，但是性能很差。**

> 所以即便是需要使用线程安全的集合类，也建议将线程不安全的集合类包装成线程安全集合类的方式，而不是直接使用这些古老的API。例如如果需要使用线程安全的集合类，则可以使用Collections工具类提供的synchronizedXxx()方法，将这些集合类包装成线程安全的集合类。

此外，从Java5开始，Java在`java.util.concurrent`包下提供了大量支持高效并发访问的集合类，它们既能包装良好的访问性能，又能包装线程安全。这些集合类可以分为两部分，它们的特征如下：

- 以`Concurrent`开头的集合类：

  以Concurrent开头的集合类代表了**支持并发访问的集合**，它们可以支持多个线程并发写入访问，这些写入线程的所有操作都是线程安全的，但读取操作不必锁定。以Concurrent开头的集合类**采用了更复杂的算法来保证永远不会锁住整个集合**，因此在并发写入时有较好的性能。

- 以`CopyOnWrite`开头的集合类：

  以CopyOnWrite开头的集合类**采用复制底层数组的方式来实现写操作。**当线程对此类集合执行读取操作时，线程将会直接读取集合本身，无须加锁与阻塞。当线程对此类集合执行写入操作时，集合会在底层复制一份新的数组，接下来对新的数组执行写入操作。由于对集合的写入操作都是对数组的副本执行操作，因此它是线程安全的。

说一说 ArrayList。

> 有关ArrayList底层源码分析及实现原理看这里：【Java 集合】这次真的从0到1彻底吃透ArrayList底层实现源码！ (imyjs.cn)

ArrayList是**List接口**的常用实现类，并且是**容量可变**的**线程不安全**列表，底层使用**Object[]数组**实现，**支持对元素的快速随机访问**，但插入与删除速度很慢。集合扩容时会创建更大的数组，把**原有数组复制到新数组。**ArrayList 实现了 `RandomAccess` 标记接口，如果一个类实现了该接口，那么表示使用索引遍历比迭代器更快。

ArrayList的主要特点可以总结如下：

- ArrayList是可以动态增长和缩减的索引序列，它是基于`Object[] 数组`实现的List类。
- 该类封装了一个动态再分配的Object[] 数组，每一个类对象都有一个`capacity`属性，表示它们所封装的Object[]数组的长度，当向ArrayList中添加元素时，该属性值会自动增加。如果想ArrayList中添加大量元素，可使用ensureCapacity方法一次性增加capacity，可以减少增加重分配的次数提高性能。

- ArrayList的用法和Vector向类似，但是Vector是一个较老的集合，具有很多缺点，不建议使用。另外，ArrayList和Vector的区别是：ArrayList是线程不安全的，当多条线程访问同一个ArrayList集合时，程序需要手动保证该集合的同步性，而Vector则是线程安全的。
- ArrayList`不是线程安全`的，只能用在单线程环境下，多线程环境下可以考虑用Collections.synchronizedList(List list)函数返回一个线程安全的ArrayList类，也可以使用concurrent并发包下的CopyOnWriteArrayList类。
- ArrayList实现了`Serializable`接口，因此它支持序列化，能够通过序列化传输，实现了`RandomAccess`接口，支持快速随机访问，实际上就是通过下标序号进行快速访问，实现了`Cloneable`接口，能被克隆。

> 增删慢：每次删除元素，都需要更改数组的长度，拷贝以及移动元素的位置
>
> 查询快：由于数组在内存中是一块连续的空间，因此可以根据索引快速获取某个位置上的元素。

> 通过查看源码可以发现实际上 `RandomAccess` 接口中什么都没有定义。所以，RandomAccess 接口只是一个标识。 标识实现这个接口的类具有随机访问功能，也就是表示使用索引遍历比迭代器更快。
>
> 具体应用比如在 `binarySearch()` 方法中，它要判断传入的 list 是否 `RamdomAccess` 的实例，如果是，调用`indexedBinarySearch()`方法，如果不是，那么调用`iteratorBinarySearch()`方法。

ArrayList的扩容机制

先说下结论，一般面试时需要记住，ArrayList的初始容量为10，扩容时对是旧的容量值加上旧的容量数值进行右移一位(位运算，相当于除以2，位运算的效率更高)，所以每次扩容都是旧的容量的1.5倍。

> ArrayList是List接口的实现类，它是支持可以根据需要而动态增长的数组。java中标准数组是定长的，在数组被创建之后，它们不能被加长或缩短。这就意味着在创建数组时需要知道数组的所需长度，但有时我们需要动态程序中获取数组长度。ArrayList就是为此而生的，但是它不是线程安全的，另外ArrayList按照插入的顺序来存放数据，默认添加到尾部。
>
> - ①ArrayList扩容发生在add()方法调用的时候， 调用ensureCapacityInternal()来扩容的，通过方法calculateCapacity(elementData, minCapacity)获取需要扩容的长度:
> - ②ensureExplicitCapacity方法可以判断是否需要扩容：
> - ③ArrayList扩容的关键方法grow():获取到ArrayList中elementData数组的内存空间长度 扩容至原来的1.5倍
> - ④调用Arrays.copyOf方法将elementData数组指向新的内存空间时newCapacity的连续空间，从此方法中我们可以清晰的看出其实ArrayList扩容的本质就是计算出新的扩容数组的size后实例化，并将原有数组内容复制到新数组中去。

Array 和ArrayList有何区别?

- Array可以包含基本类型和对象类型，ArrayList只能包含对象类型。
- Array大小是固定的，ArrayList的大小是动态变化的。
- 相比于Array，ArrayList有着更多的内置方法，如addAll()、removeAll()。
- 对于基本类型数据，ArrayList使用自动装箱来减少编码工作量；而当处理固定大小的基本数据类型的时候，这种方式相对比较慢，这时候应该使用Array 。

说一说 LinkedList。

LinkedList 本质是一个双向链表。同时实现了`List`接口和`Deque`接口，也就是说它既可以看作一个顺序容器，也可以被当作堆栈、队列或双端队列进行操作。当你需要使用栈或者队列时，可以考虑用LinkedList。但是，关于栈或队列，现在首选是ArrayDeque，它有着比LinkedList（当作栈或队列使用时）更好的性能。
LinkedList 还实现了`Cloneable`接口，即覆盖了函数clone()，支持克隆。 LinkedList 也实现`Serializable`接口，这意味着LinkedList支持序列化，能通过序列化去传输。 此外，LinkedList 是非同步的，也就是线程不安全的。

与 ArrayList 相比**插入和删除速度更快，但随机访问元素很慢。**LinkedList 包含三个重要的成员：size、first 和 last。中双向链表的每个节点用内部类Node表示。size 是双向链表中节点的个数，first 和 last 分别指向首尾节点的引用。LinkedList 的优点在于可以将零散的内存单元通过附加引用的方式关联起来，形成按链路顺序查找的线性结构，内存利用率较高。

ArrayList 和 LinkedList 的区别

1. **底层数据结构**

   ArrayList 底层使用的是`Object[] 数组`，存储空间是连续的；LinkedList 底层使用的是`双向链表`数据结构（JDK1.6 之前为循环链表，JDK1.7 取消了循环。） 存储空间是不连续的。

2. **插入和删除是否受元素位置的影响**

   - ArrayList 底层采用数组存储，所以插入和删除元素的时间复杂度受元素位置的影响。比如：执行add(E e)方法的时候， ArrayList 会默认在将指定的元素追加到此列表的`末尾`，这种情况时间复杂度就是 O(1)。但是如果要在指定位置 i 插入和删除元素的话（`add(int index, E element)`）时间复杂度就为 O(n-i)。因为在进行上述操作的时候集合中第 i 和第 i 个元素之后的(n-i)个元素都要执行向后位/向前移一位的操作。
   - LinkedList 底层采用链表存储，所以，如果是在头尾插入或者删除元素不受元素位置的影响（`add(E e)`、addFirst(E e)、`addLast(E e)`、`removeFirst()`、`removeLast()`），近似 O(1)，如果是要在指定位置`i`插入和删除元素的话（`add(int index, E element)`） 时间复杂度近似为 O(n)，因为需要先移动到指定位置再插入。

     > - 对于随机访问get 和 set，ArrayList 优于 LinkedList，因为 LinkedList 要移动指针。
     > - 对于新增和删除操作 add 和 remove，LinedList 比较占优势，因为 ArrayList 要移动数据。

3. **是否支持快速随机访问**

LinkedList 不支持高效的随机元素访问，而 ArrayList 支持。快速随机访问就是通过元素的序号快速获取元素对象(对应于get(int index)方法)。

4. **是否保证线程安全**

   ArrayList 和 LinkedList 都是不同步的，也就是不保证线程安全；

5. **内存空间占用**

   ArrayList 的空间浪费主要体现在在List列表的结尾会预留一定的容量空间，而 LinkedList 的空间花费则体现在它的每一个元素都需要消耗比 ArrayList 更多的空间（因为要存放直接后继和直接前驱以及数据）。

> 同样的数据量 LinkedList 所占用空间可能会更小，因为 ArrayList 需要**预留空间**便于后续数据增加，而 LinkedList 增加数据只需要增加一个节点。

Arraylist 和 Vector 的区别

相同点

- 1.都实现了List接口
- 2.底层数据结构都是Object[]数组

不同点

- 1.`线程安全`: vector使用了`synchronized`来实现线程同步，所以是线程安全的，而ArrayList是线程不安全的。
- 2.`性能：`由于vector使用了synchronized进行加锁，所以性能不如ArrayList。
- 3.`扩容`: ArrayList和vector都会根据需要动态的调整容量，但是ArrayList每次扩容为旧容量的1.5倍，而vector每次扩容为旧容量的2倍。

元素排序Comparable和Comparator有什么区别?

在 Java 语言中，Comparable 和 Comparator 都是用来进行元素排序的，但二者有着本质的区别。

1.字面含义不同

Comparable 翻译为中文是“比较”的意思，而 Comparator 是“比较器”的意思。Comparable 是以 -able 结尾的，表示它自身具备着某种能力，而 Comparator 是以 -or 结尾，表示自身是比较的参与者，这是从字面含义先来理解二者的不同。

2.用法不同

二者都是顶级的接口，但拥有的方法和用法是不同的，下面我们分别来看。

- `Comparable` 接口只有一个方法 `compareTo(Object obj)`，实现 Comparable 接口并重写 compareTo 方法就可以实现某个类的排序，它支持 Collections.sort 和 Arrays.sort 的排序。Comparable 的使用是在自定义对象的类中实现 Comparable 接口，并重写 compareTo 方法来实现自定义排序规则的

```
class Person implements Comparable<Person> {
    private int id;
    private int age;
```

```
    private String name;

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

    @Override
    public int compareTo(Person p) {
        return p.getAge() - this.getAge();
    }
}
```

compareTo 方法接收的参数 p 是要对比的对象，排序规则是用当前对象和要对比的对象进行比较，然后返回一个 int 类型的值。正序从小到大的排序规则是：使用当前的对象值减去要对比对象的值；而倒序从大到小的排序规则刚好相反：是用对比对象的值减去当前对象的值。

> 注意事项：如果自定义对象没有实现 Comparable 接口，那么它是不能使用 Collections.sort 方法进行排序的，编译器会报错。

- `Comparator`排序的方法是`compare(Object obj1, Object obj2)`，具体实现代码如下：

```
import lombok.Getter;
import lombok.Setter;

import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

public class ComparatorExample {
    public static void main(String[] args) {
        // 创建对象
        Person p1 = new Person(1, 18, "Java");
        Person p2 = new Person(2, 22, "MySQL");
        Person p3 = new Person(3, 6, "Redis");
        // 添加对象到集合
        List<Person> list = new ArrayList<>();
        list.add(p1);
        list.add(p2);
        list.add(p3);
        // 进行排序操作(根据 PersonComparator 中定义的排序规则)
        Collections.sort(list, new PersonComparator());
        // 输出集合中的顺序
        list.forEach(p -> System.out.println(p.getName() +
                ": " + p.getAge()));
    }
}
```

```java
/**
  * 用于 Person 类的比较器
  */
class PersonComparator implements Comparator<Person> {
    @Override
    public int compare(Person p1, Person p2) {
        return p2.getAge() - p1.getAge();
    }
}
@Getter
@Setter
class Person {
    private int id;
    private int age;
    private String name;

    public Person(int id, int age, String name) {
        this.id = id;
        this.age = age;
    }
}
```

Comparator 除了可以通过创建自定义比较器外，还可以通过匿名类的方式，更快速、便捷的完成自定义比较器的功能，具体的代码实现如下：

```java
// 使用 Comparator 匿名类的方式进行排序
list.sort(new Comparator<Person>() {
    @Override
    public int compare(Person p1, Person p2) {
        return p2.getAge() - p1.getAge();
    }
});
```

3.使用的场景不同

通过上面示例的实现代码我们可以看出，使用 Comparable 必须要修改原有的类，也就是你要排序那个类，就要在那个中实现 Comparable 接口并重写 compareTo 方法，所以 Comparable 更像是“对内”进行排序的接口。

而 Comparator 的使用则不相同，Comparator 无需修改原有类。也就是在最极端情况下，即使 Person 类是第三方提供的，我们依然可以通过创建新的自定义比较器 Comparator，来实现对第三方类 Person 的排序功能。也就是说通过 Comparator 接口可以实现和原有类的解耦，在不修改原有类的情况下实现排序功能，所以 Comparator 可以看作是“对外”提供排序的接口。

> Comparable 和 Comparator 都是用来实现元素排序的，它们二者的区别如下：
>
> - Comparable 是“比较”的意思，而 Comparator 是“比较器”的意思；
> - Comparable 是通过重写 compareTo 方法实现排序的，而 Comparator 是通过重写 compare 方法实现排序的；

> - Comparable 必须由自定义类内部实现排序方法，而 Comparator 是外部定义并实现排序的。
>
> 所以用一句话总结二者的区别：Comparable 可以看作是“对内”进行排序接口，而 Comparator 是“对外”进行排序的接口。

Collection和Collections有什么区别?

1、**java.util.Collection 是一个集合接口。**它提供了对集合对象进行基本操作的通用接口方法。Collection接口在Java 类库中有很多具体的实现。Collection接口的意义是为各种具体的集合提供了最大化的统一操作方式。

List，Set，Queue接口都继承Collection。 直接实现该接口的类只有AbstractCollection类，该类也只是一个抽象类，提供了对集合类操作的一些基本实现。List和Set的具体实现类基本上都直接或间接的继承了该类。

2、**java.util.Collections 是一个包装类。** 它包含有各种有关集合操作的静态方法（对集合的搜索、排序、线程安全化等），大多数方法都是用来处理线性表的。此类不能实例化，**就像一个工具类，服务于Java的Collection框架。**

遍历一个List有哪些不同的方式?

先说一下常见的元素在内存中的存储方式，主要有两种:
1.**顺序存储**：相邻的数据元素在内存中的位置也是相邻的，可以根据元素的位置(如ArrayList中的下标）读取元素。
2.**链式存储**：每个数据元素包含它下一个元素的内存地址，在内存中不要求相邻。例如LinkedList。
主要的遍历方式主要有三种:

- 1.**for循环遍历**:遍历者自己在集合外部维护一个计数器，依次读取每一个位置的元素。
- 2.**Iterator遍历**:基于顺序存储集合的Iterator可以直接按位置访问数据。基于链式存储集合的Iterator，需要保存当前遍历的位置，然后根据当前位置来向前或者向后移动指针。
- 3.`foreach遍历`: foreach内部也是采用了Iterator的方式实现，但使用时不需要显示地声明Iterator。

> 在Java集合框架中，提供了一个RandomAccess接口，该接口没有方法，只是一个标记。通常用来标记List的实现是否支持RandomAccess。所以在遍历时，可以先判断是否支持RandomAccess (listinstanceof RandomAccess)，如果支持可用for循环遍历，否则建议用Iterator或 foreach遍历。

Set 有什么特点，有哪些实现类?

Set 不允许元素重复且无序，常用实现有 `HashSet`、`LinkedHashSet` 和 `TreeSet`。

- `HashSet` 通过 HashMap 实现，HashMap 的 Key 即 HashSet 存储的元素，所有 Key 都使用相同的 Value ，一个名为 `PRESENT` 的 `Object 类型常量`。使用 Key 保证元素唯一性，但不保证有序性。由于 HashSet 是 HashMap 实现的，因此**线程不安全。**HashSet 判断元素是否相同时，对于包装类型直接按值比较。对于其他引用类型先比较 hashCode 是否相同，不同则代表不是同一个对象，相同则继续比较 equals，都相同才是同一个对象。

- `LinkedHashSet` 继承自 HashSet，通过 LinkedHashMap 实现，使用`双向链表维护元素插入顺序`。
- `TreeSet` 通过 TreeMap 实现的，添加元素到集合时按照比较规则将其插入合适的位置，保证插入后的集合仍然有序。

说一说HashSet。

> 有关HashSet底层源码分析及实现原理看这里:【Java 集合】这次真的从0到1彻底吃透HashSet底层实现源码！ (imyjs.cn)

- HashSet 是基于 HashMap 实现的，底层采用 `HashMap` 来保存元素。
- HashMap 的 Key 即 HashSet 存储的元素，所有 Key 都使用相同的 Value ，一个名为 PRESENT 的 Object 类型常量。使用 Key 保证元素唯一性，但不保证有序性。
- 由于 HashSet 是 HashMap 实现的，因此`线程不安全`。
- HashSet 判断元素是否相同时，对于包装类型直接按值比较。对于其他引用类型先比较 hashCode 是否相同，元素的哈希值是通过元素的 hashCode 方法 来获取的, HashSet 首先判断两个元素的哈希值，如果哈希值一样，接着会比较 equals 方法 如果 equls 结果为 true ，HashSet 就视为同一个元素。如果 equals 为 false 就不是同一个元素。

  > 首先比较hashCode 值不同则代表不是同一个对象，相同则继续比较 equals，都相同才是同一个对象。

比较 HashSet、LinkedHashSet 和 TreeSet 三者的异同

- HashSet、LinkedHashSet 和 TreeSet 都是 `Set` 接口的实现类，都能保证元素唯一，并且都不是线程安全的。
- HashSet、LinkedHashSet 和 TreeSet 的主要区别在于底层数据结构不同。HashSet 的底层数据结构是`哈希表`（基于 `HashMap` 实现）。LinkedHashSet 的底层数据结构是`链表和哈希表`，元素的插入和取出顺序满足 FIFO。TreeSet 底层数据结构是`红黑树`，元素是有序的，排序的方式有自然排序和定制排序。
- 底层数据结构不同又导致这三者的应用场景不同。**HashSet 用于不需要保证元素插入和取出顺序的场景，LinkedHashSet 用于保证元素的插入和取出顺序满足 FIFO 的场景，TreeSet 用于支持对元素自定义排序规则的场景。**

Queue 与 Deque 的区别

Queue 是`单端队列`，只能从一端插入元素，另一端删除元素，实现上一般遵循 **先进先出（FIFO）** 规则。

Queue 扩展了 Collection 的接口，根据 **因为容量问题而导致操作失败后处理方式的不同** 可以分为两类方法: 一种在操作失败后会抛出异常，另一种则会返回特殊值。

`Queue` 接口	抛出异常	返回特殊值
插入队尾	add(E e)	offer(E e)
删除队首	remove()	poll()

`Queue` 接口	抛出异常	返回特殊值
查询队首元素	element()	peek()

Deque 是`双端队列`，在队列的两端均可以插入或删除元素。

Deque 扩展了 Queue 的接口, 增加了在队首和队尾进行插入和删除的方法，同样根据失败后处理方式的不同分为两类：

`Deque` 接口	抛出异常	返回特殊值
插入队首	addFirst(E e)	offerFirst(E e)
插入队尾	addLast(E e)	offerLast(E e)
删除队首	removeFirst()	pollFirst()
删除队尾	removeLast()	pollLast()
查询队首元素	getFirst()	peekFirst()
查询队尾元素	getLast()	peekLast()

事实上，`Deque` 还提供有 `push()` 和 `pop()` 等其他方法，可用于模拟栈。

ArrayDeque 与 LinkedList 的区别

ArrayDeque 和 LinkedList 都实现了 Deque 接口，两者都具有队列的功能，两者区别主要如下：

- ArrayDeque 是基于可变长的`数组和双指针`来实现，而 LinkedList 则通过`链表`来实现。
- ArrayDeque 不支持存储 `NULL 数据`，但 LinkedList 支持。
- ArrayDeque 是在 JDK1.6 才被引入的，而LinkedList 早在 JDK1.2 时就已经存在。
- ArrayDeque 插入时可能存在扩容过程, 不过均摊后的插入操作依然为 O(1)。虽然 LinkedList 不需要扩容，但是每次插入数据时均需要申请新的堆空间，均摊性能相比更慢。

从性能的角度上，选用 ArrayDeque 来实现队列要比 LinkedList 更好。此外，ArrayDeque 也可以用于实现栈。

说一说 PriorityQueue

PriorityQueue 是在 JDK1.5 中被引入的, 其与 Queue 的区别在于元素出队顺序是与优先级相关的，即**总是优先级最高的元素先出队。**

这里列举其相关的一些要点：

- PriorityQueue 利用了`二叉堆`的数据结构来实现的，底层使用`可变长的数组`来存储数据
- PriorityQueue 通过堆元素的上浮和下沉，实现了在 `O(logn)` 的时间复杂度内插入元素和删除堆顶元素。
- PriorityQueue 是`非线程安全`的，且`不支持存储 NULL 和 non-comparable 的对象`。
- PriorityQueue `默认是小顶堆`，但可以接收一个 Comparator 作为构造参数，从而来自定义元素优先级的先后。

PriorityQueue 在面试中可能更多的会出现在手撕算法的时候，典型例题包括堆排序、求第K大的数、带权图的遍历等，所以需要会熟练使用才行。

能否使用任何类作为Map的key?

可以，但要注意以下两点:

- 如果类重写了equals()方法，也应该重写hashcode()方法。
- 最好定义key类是不可变的，这样key对应的hashcode()值可以被缓存起来，性能更好，这也是为什么string特别适合作为HashMap的key

为什么HashMap中String、Integer这样的包装类适合作为Key?

- 这些包装类都是fina1修饰，是不可变性的，保证了key的不可更改性，不会出现放入和获取时哈希值不同的情况。
- 它们内部已经重写过hashcode() , equals()等方法。

HashMap 1.7 和 HashMap 1.8 的区别?

不同点	hashMap 1.7	hashMap 1.8
数据结构	数组+链表	数组+链表+红黑树
插入数据的方式	头插法	尾插法
hash 值计算方式	9次扰动处理(4次位运算+5次异或)	2次扰动处理(1次位运算+1次异或)
扩容策略	插入前扩容	插入后扩容

底层使用的数据结构结构不同，插入数据的方式、hash 值计算方式不同以及扩容策略不同。

> hashMap 1.7中，采用数组+链表数据结构作为底层实现，插入数据的方式为头插法，hash 值计算方式包含9次扰动处理(4次位运算+5次异或)，在插入前扩容。
>
> hashMap 1.8中，采用数组+链表+红黑树数据结构作为底层实现，插入数据的方式为尾插法，hash 值计算方式包含2次扰动处理(1次位运算+1次异或)，在插入后扩容。

HashMap 和 Hashtable 的区别

1. **线程是否安全：** `HashMap` 是非线程安全的，`Hashtable` 是线程安全的,因为 `Hashtable` 内部的方法基本都经过`synchronized` 修饰。（如果要保证线程安全的话就使用`ConcurrentHashMap`！）；
2. **效率：** 因为线程安全的问题，`HashMap` 要比 `Hashtable` 效率高一点。另外，`Hashtable` 基本被淘汰，不要在代码中使用它；
3. **对 Null key 和 Null value 的支持：** `HashMap` 可以存储 null 的 key 和 value，但 null 作为键只能有一个，null 作为值可以有多个；Hashtable 不允许有 null 键和 null 值，否则会抛出 `NullPointerException`。

4. **初始容量大小和每次扩充容量大小的不同：** ① 创建时如果不指定容量初始值，`Hashtable` 默认的初始大小为 11，之后每次扩充，容量变为原来的 2n+1。`HashMap` 默认的初始化大小为 16。之后每次扩充，容量变为原来的 2 倍。② 创建时如果给定了容量初始值，那么 Hashtable 会直接使用你给定的大小，而 `HashMap` 会将其扩充为 2 的幂次方大小。也就是说 `HashMap` 总是使用 2 的幂作为哈希表的大小。
5. **底层数据结构：** JDK1.8 以后的 `HashMap` 在解决哈希冲突时有了较大的变化，当链表长度大于阈值（默认为 8）（将链表转换成红黑树前会判断，如果当前数组的长度小于 64，那么会选择先进行数组扩容，而不是转换为红黑树）时，将链表转化为红黑树，以减少搜索时间。Hashtable 没有这样的机制。

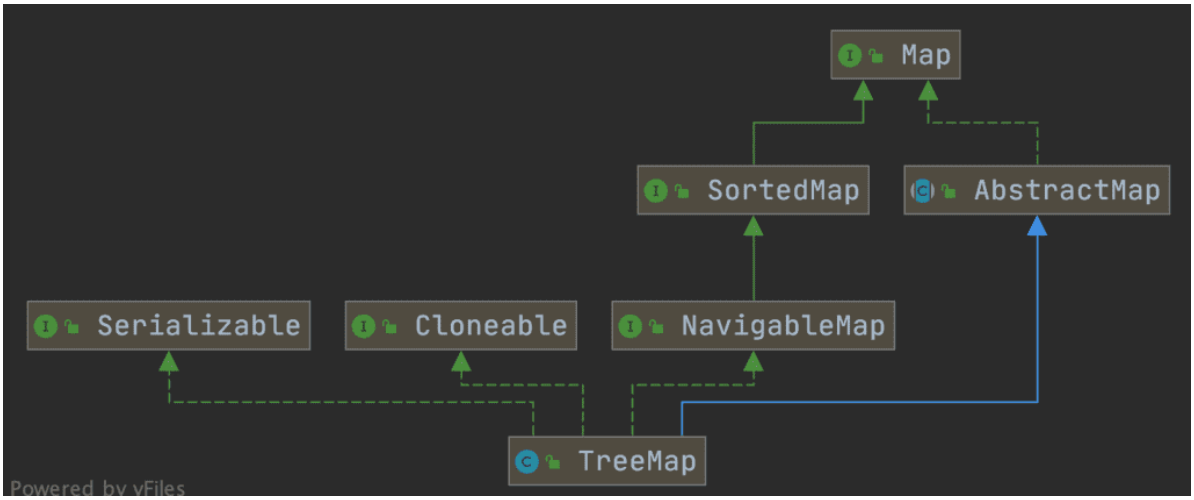
HashMap 和 HashSet 区别

HashSet 底层就是基于 HashMap 实现的。`HashSet` 的源码非常非常少，因为除了 `clone()`、`writeObject()`、`readObject()` 是 HashSet 自己不得不实现之外，其他方法都是直接调用 HashMap 中的方法。

HashMap	HashSet
实现了 `Map` 接口	实现 `Set` 接口
存储键值对	仅存储对象
调用 `put()` 向 map 中添加元素	调用 `add()` 方法向 Set 中添加元素
`HashMap` 使用键（Key）计算 `hashcode`	HashSet 使用成员对象来计算 `hashcode` 值，对于两个对象来说 `hashcode` 可能相同，所以 `equals()` 方法用来判断对象的相等性

HashMap 和 TreeMap 区别

`TreeMap` 和 `HashMap` 都继承自 `AbstractMap` ，但是需要注意的是 `TreeMap` 它还实现了 `NavigableMap` 接口和 `SortedMap` 接口。



实现 `NavigableMap` 接口让 TreeMap 有了对集合内元素的搜索的能力。

实现`SortedMap`接口让 TreeMap 有了对集合中的元素根据键排序的能力。默认是按 key 的升序排序，不过我们也可以指定排序的比较器。

综上，相比于HashMap来说 TreeMap 主要多了`对集合中的元素根据键排序的能力以及对集合内元素的搜索的能力`**。**

HashMap 的底层实现

JDK1.8 之前

JDK1.8 之前 `HashMap` 底层是 `数组和链表` 结合在一起使用也就是 **链表散列。**其中数组是 HashMap 的主体，链表则是主要为了解决哈希冲突而存在的。

HashMap 通过 key 的 hashCode 经过`扰动函数`处理过后得到 hash 值，然后通过 `(n - 1) & hash` 判断当前元素存放的位置（这里的 n 指的是数组的长度），如果当前位置存在元素的话，就判断该元素与要存入的元素的 hash 值以及 key 是否相同，如果相同的话，直接覆盖，不相同就通过拉链法解决冲突。

> 所谓扰动函数指的就是 HashMap 的 hash 方法。使用 hash 方法也就是扰动函数是为了防止一些实现比较差的 hashCode() 方法 换句话说使用扰动函数之后可以减少碰撞。
>
> 所谓 **“拉链法”** 就是：将链表和数组相结合。也就是说创建一个链表数组，数组中每一格就是一个链表。若遇到哈希冲突，则将冲突的值加到链表中即可。

JDK1.8 之后

JDK1.8 之后在解决哈希冲突时有了较大的变化，相比于之前的版本，链表容易过长，最坏情况下会成为单向链表，会严重影响 HashMap 的性能， 而红黑树搜索的时间复杂度是 O(logn)，而链表是糟糕的 O(n)，于是就引入了红黑树，链表和红黑树在达到一定条件会进行转换：

- 当链表超过 8 且数据总量超过 64 时会转红黑树。
- 将链表转换成红黑树前会判断，如果当前数组的长度小于 64，那么会选择先进行数组扩容，而不是转换为红黑树，以减少搜索时间。

> TreeMap、TreeSet 以及 JDK1.8 之后的 HashMap 底层都用到了红黑树。红黑树就是为了解决二叉查找树的缺陷，因为二叉查找树在某些情况下会退化成一个线性结构。

HashMap的扩容机制

为了方便说明，这里明确几个名词：

- capacity 即容量，默认16。
- loadFactor 加载因子，默认是0.75
- threshold 阈值。阈值=容量*加载因子。默认12。当元素数量超过阈值时便会触发扩容。

一般情况下，**当元素数量超过阈值时**便会触发扩容。每次扩容的容量都是之前容量的2倍。HashMap的容量是有上限的，必须小于**1<<30**，即1073741824。

JDK8的HashMap扩容通常存在以下几种情况：

1. 空参数的构造函数：实例化的HashMap默认内部数组是null，即没有实例化。第一次调用put方法时，则会开始第一次初始化扩容，长度为16。

2. 有参构造函数：用于指定容量。会根据指定的正整数找到**不小于指定容量的2的幂数**，将这个数设置赋值给**阈值**（threshold）。第一次调用put方法时，会将阈值赋值给容量，然后让阈值=容量×负载因子 。（因此并不是我们手动指定了容量就一定不会触发扩容，超过阈值后一样会扩容！！）
3. 如果不是第一次扩容，则容量变为原来的2倍，阈值也变为原来的2倍。 *（容量和阈值都变为原来的2倍时，负载因子还是不变）*

此外还有几个细节需要注意：

- 首次put时，先会触发扩容（算是初始化），然后存入数据，然后判断是否需要扩容；
- 不是首次put，则不再初始化，直接存入数据，然后判断是否需要扩容；

HashMap 中为什么链表改为红黑树的阈值是 8?

因为和哈希码碰撞次数的**泊松分布**有关。通过泊松分布算出，在负载因子0.75（HashMap默认）的情况下，当桶中结点个数为8时，出现的几率是亿分之6的，**因此常见的情况是桶中个数小于8的情况，此时链表的查询性能和红黑树相差不多，因为转化为树还需要时间和空间，所以此时没有转化成树的必要。**

通过看作者在源码中的注释，翻译过来大概的意思是：理想情况下使用随机的哈希码，容器中节点分布在 hash 桶中的频率遵循泊松分布，按照泊松分布的计算公式计算出了桶中元素个数和概率的对照表，可以看到链表中元素个数为 8 时的概率已经非常小，再多的就更少了，所以原作者在选择链表元素个数时选择了 8，是根据概率统计而选择的。所以是大于8时转为红黑树，小于等于6时转为链表。

解决hash冲突的办法有哪些？HashMap用的哪种?

解决Hash冲突方法有：

- 开放定址法：也称为再散列法，基本思想就是，如果p=H(key)出现冲突时，则以p为基础，再次hash，p1=H(p),如果p1再次出现冲突，则以p1为基础，以此类推，直到找到一个不冲突的哈希地址pi。因此开放定址法所需要的hash表的长度要大于等于所需要存放的元素，而且因为存在再次hash，所以只能在删除的节点上做标记，而不能真正删除节点。
- 再哈希法：双重散列，多重散列，提供多个不同的hash函数，当R1=H1(key1)发生冲突时，再计算R2=H2(key1)，直到没有冲突为止。这样做虽然不易产生堆集，但增加了计算的时间。
- 链地址法：拉链法，将哈希值相同的元素构成一个同义词的单链表，并将单链表的头指针存放在哈希表的第i个单元中，查找、插入和删除主要在同义词链表中进行。链表法适用于经常进行插入和删除的情况。
- 建立公共溢出区：将哈希表分为公共表和溢出表，当溢出发生时，将所有溢出数据统一放到溢出区。

HashMap中采用的是`链地址法`。

HashMap是如何解决Hash冲突的?

哈希冲突: hashMap在存储元素时会先计算key的hash值来确定存储位置，因为key 的hash值计算最后有个对数组长度取余的操作，所以即使不同的key也可能计算出相同的hash值，这样就引起了hash冲突。

HashMap中的哈希冲突解决方式可以主要从三方面考虑(以JDK1.8为背景)

- 拉链法
  HasMap中的数据结构为数组+链表/红黑树，当不同的key计算出的hash值相同时，就用链表的形式将Node结点（冲突的key及key对应的value)挂在数组后面。
- hash函数
  key的hash值经过两次扰动，key的哈希值与key的哈希值右移16位的值进行异或hash ^（hash >> 16），然后对数组的长度取余（实际为了提高性能用的是位运算，但目的和取余一样)，这样做可以让哈希取值出的高位也参与运算，进一步降低hash冲突的概率，使得数据分布更平均。
- 红黑树
  在拉链法中，如果hash冲突特别严重，则会导致数组上挂的链表长度过长，性能变差，因此在链表长度大于8时，将链表转化为红黑树，可以提高遍历链表的速度。

为什么在解决 hash 冲突的时候，不直接用红黑树？而选择先用链表，再转红黑树?

因为红黑树需要进行左旋，右旋，变色这些操作来保持平衡，而单链表不需要。

当元素小于 8 个的时候，此时做查询操作，链表结构已经能保证查询性能。当元素大于 8 个的时候， 红黑树搜索时间复杂度是 O(logn)，而链表是 O(n)，此时需要红黑树来加快查询速度，但是新增节点的效率变慢了。

因此，如果一开始就用红黑树结构，元素太少，新增效率又比较慢，无疑这是浪费性能的。

> 最开始使用链表的时候，空间占用是比较少的，而且由于链表短，所以查询时间也没有太大的问题。可是当链表越来越长，需要用红黑树的形式来保证查询的效率。**默认是链表长度达到 8 就转成红黑树，而当长度降到 6 就转换回去，这体现了时间和空间平衡的思想。**

HashMap默认加载因子是多少？为什么是 0.75，不是 0.6 或者 0.8？

HashMap中除了哈希算法之外，有两个参数影响了性能：初始容量和加载因子。初始容量是哈希表在创建时的容量，加载因子是哈希表在其容量自动扩容之前可以达到多满的一种度量。

在维基百科来描述加载因子：

> 对于开放定址法，加载因子是特别重要因素，应严格限制在0.7-0.8以下。超过0.8，查表时的CPU缓存不命中（cache missing）按照指数曲线上升。因此，一些采用开放定址法的hash库，如Java的系统库限制了加载因子为0.75，超过此值将resize散列表。

在设置初始容量时应该考虑到映射中所需的条目数及其加载因子，以便最大限度地减少扩容rehash操作次数，所以，一般在使用HashMap时建议根据预估值设置初始容量，以便减少扩容操作。

选择0.75作为默认的加载因子，完全是时间和空间成本上寻求的一种折衷选择。

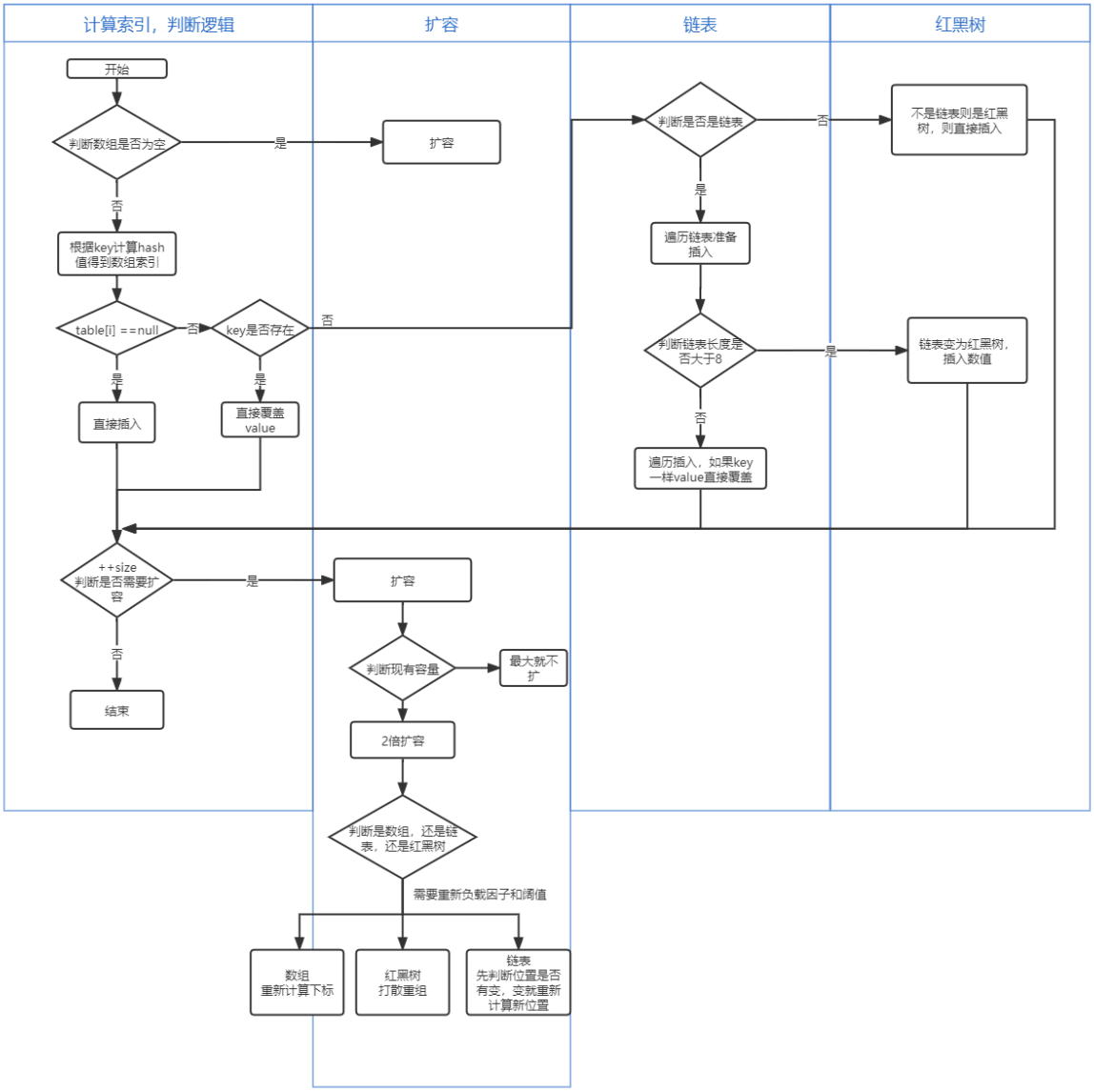
Map接口有哪些实现类，各自的使用场景是什么?

Map接口有很多实现类，其中比较常用的有`HashMap`、`LinkedHashMap`、`TreeMap`、`ConcurrentHashMap`。

- 对于不需要排序的场景，优先考虑使用HashMap，因为它是性能最好的Map实现。
- 如果需要保证线程安全，则可以使用ConcurrentHashMap。它的性能好于Hashtable，因为它在put时采用分段锁/CAS的加锁机制，而不是像Hashtable那样，无论是put还是get都做同步处理。
- 对于需要排序的场景，如果需要按插入顺序排序则可以使用LinkedHashMap，如果需要将key按自然顺序排列甚至是自定义顺序排列，则可以选择TreeMap。同时如果需要保证线程安全，则可以使用Collections工具类将上述实现类包装成线程安全的Map。

描述一下Map put的过程

hashmap的put过程



首先判断数组是否为空，若数组为空则进行第一次扩容（resize），不为空则通过hash算法，计算Key在数组中的索引；

然后进行插入元素，

- 如果当前位置元素为空，则直接插入数据；
- 如果当前位置元素非空，判断key是否存在，
    - 若key已存在，则直接覆盖其value；
    - 若key不存在，则判断是否有链表，
        - 若不是链表则是红黑树直接插入
        - 若是链表则判断链表长度是否大于8
            - 若链表长度大于8，则转换为红黑树直接插入
            - 若链表长度不大于8，则将数据链到链表末端；

如果数组中元素个数（size）超过threshold，则再次进行扩容操作。

HashMap的遍历方式?

HashMap **遍历从大的方向来说，可分为以下 4 类**：

1. 迭代器（Iterator）方式遍历；
2. For Each 方式遍历；
3. Lambda 表达式遍历（JDK 1.8+）；
4. Streams API 遍历（JDK 1.8+）。

但每种类型下又有不同的实现方式，因此具体的遍历方式又可以分为以下 7 种：

1. 使用迭代器（Iterator）EntrySet 的方式进行遍历；
2. 使用迭代器（Iterator）KeySet 的方式进行遍历；
3. 使用 For Each EntrySet 的方式进行遍历；
4. 使用 For Each KeySet 的方式进行遍历；
5. 使用 Lambda 表达式的方式进行遍历；
6. 使用 Streams API 单线程的方式进行遍历；
7. 使用 Streams API 多线程的方式进行遍历。

> 我们不能在遍历中使用集合 `map.remove()` 来删除数据，这是非安全的操作方式，但我们可以使用迭代器的 `iterator.remove()` 的方法来删除数据，这是安全的删除集合的方式。同样的我们也可以使用 Lambda 中的 `removeIf` 来提前删除数据，或者是使用 Stream 中的 `filter` 过滤掉要删除的数据进行循环，这样都是安全的，当然我们也可以在 `for` 循环前删除数据在遍历也是线程安全的。
>
> 综合性能和安全性来看，**我们应该尽量使用迭代器（Iterator）来遍历 `EntrySet` 的遍历方式来操作 Map 集合**，这样就会既安全又高效。

在HashMap遍历中，为什么EntrySet 比 KeySet的性能高?

EntrySet 之所以比 KeySet 的性能高是因为，KeySet 在循环时使用了 `map.get(key)`，而 `map.get(key)` 相当于又遍历了一遍 Map 集合去查询 `key` 所对应的值。为什么要用“又”这个词？那是因为**在使用迭代器或者 for 循环时，其实已经遍历了一遍 Map 集合了，因此再使用 `map.get(key)` 查询时，相当于遍历了两遍。**

而 EntrySet 只遍历了一遍 Map 集合，之后通过代码“Entry<Integer, String> entry = iterator.next()”把对象的 `key` 和 `value` 值都放入到了 `Entry` 对象中，因此再获取 `key` 和 `value` 值时就无需再遍历 Map 集合，只需要从 `Entry` 对象中取值就可以了。

所以，**`EntrySet` 的性能比 `KeySet` 的性能高出了一倍，因为 `KeySet` 相当于循环了两遍 Map 集合，而 `EntrySet` 只循环了一遍。**

HashMap 为什么线程不安全，体现在哪里?

- HashMap在JDK 7 时多线程下扩容并发执行put操作时，可能会导致形成循环链表，从而引起死循环。
- 多线程的put可能导致元素的丢失。
- put和get并发时，可能导致get为null。

在 **hashMap1.7 中扩容**的时候，因为采用的是头插法，所以会可能会有循环链表产生，导致数据有问题，在 1.8 版本已修复，改为了尾插法。

在任意版本的 hashMap 中，如果在**插入数据时多个线程命中了同一个槽**，可能会有数据覆盖的情况发生，导致线程不安全。

那么 HashMap 线程不安全怎么解决？或者说如何得到一个线程安全的Map?

- 用Collections工具类，将线程不安全的Map包装成线程安全的Map，也就是给 hashMap **直接加锁**,来保证线程安全
- 使用java.util.concurrent包下的**concurrentHashMap** , 不管是其 1.7 还是 1.8 版本,本质都是**减小了锁的粒度,减少线程竞争**来保证高效
- 使用 **hashTable**,其实就是在其方法上加了 synchronized 锁，但是不建议使用Hashtable，虽然Hashtable是线程安全的，但是性能较差。

详细说说怎样实现一个线程同步的HashMap?

- 将所有public方法都加上`synchronized`，可以使用 Collections.synchronizedMap同步方法，这是`java.util.Collections`提供的一个静态方法，用这个方法包装下`HashMap`，它就变成线程安全的了。

  `synchronizedMap`实现线程安全的原理也很简单，它首先基于当前的map对象生成一个新的map类型`synchronizedMap`, 这是`Collections`类里面的一个内部类。进入源码可以看到它的所有操作都用了`synchronized`加了一个对象锁
- 使用ConcurrentHashMap同步，在java7中，ConcurrentHashMap 是一个segment数组，segment通过继承 ReentrantLock来进行加锁，锁的颗粒度比较细，相当于每次锁住的是一个segment。这样性能更高。在JDK8里，ConcurrentHashMap的实现又有了很大变化，它在锁分离的基础上，大量利用了了CAS指令。并且底层存储有一个小优化，当链表长度太长（默认超过8）时，链表就转换为红黑树。链表太长时，增删查改的效率比较低，改为红黑树可以提高性能。实现的难度有点高，JDK8里的ConcurrentHashMap有6000多行代码，JDK7才1500多行。

> 两种方法的对比：

> 首先，从上面分析的同步原理看，synchronizedMap加锁是基于操作的，简单粗暴。而ConcurrentHashMap是`分段加锁，锁的颗粒度更细，性能自然更高`。高并发的场景下还是建议使用后者。
>
> 还有一个区别是，ConcurrentHashMap永远不会抛出ConcurrentModificationException异常。而synchronizedMap在迭代遍历时，如果某些元素被删除了，会触发fail-fast机制抛出ConcurrentModificationException异常。

HashMap 的长度为什么是 2 的幂次方

为了能让 HashMap 存取高效，尽量减少碰撞，也就是要尽量把数据分配均匀。Hash 值的范围值-2147483648 到 2147483647，前后加起来大概 40 亿的映射空间，只要哈希函数映射得比较均匀松散，一般应用是很难出现碰撞的。但问题是一个 40 亿长度的数组，内存是放不下的。所以这个散列值是不能直接拿来用的。用之前还要先做对数组的长度取模运算，得到的余数才能用来要存放的位置也就是对应的数组下标。这个数组下标的计算方法是“`(n - 1) & hash`”。（n 代表数组长度）。这也就解释了 HashMap 的长度为什么是 2 的幂次方。

这个算法应该如何设计呢?

我们首先可能会想到采用%取余的操作来实现。但是，重点来了：**“取余(%)操作中如果除数是 2 的幂次则等价于与其除数减一的与(&)操作（也就是说 hash%length==hash&(length-1)的前提是 length 是 2 的 n 次方;)。”** 并且 **采用二进制位操作 &，相对于%能够提高运算效率，这就解释了 HashMap 的长度为什么是 2 的幂次方**

ConcurrentHashMap 和 Hashtable 的区别

`ConcurrentHashMap` 和 `Hashtable` 的区别主要体现在实现线程安全的方式上不同。

- **底层数据结构：** JDK1.7 的 `ConcurrentHashMap` 底层采用 **分段的数组+链表** 实现，JDK1.8 采用的数据结构跟 `HashMap1.8` 的结构一样，数组+链表/红黑二叉树。`Hashtable` 和 JDK1.8 之前的 `HashMap` 的底层数据结构类似都是采用 **数组+链表** 的形式，数组是 HashMap 的主体，链表则是主要为了解决哈希冲突而存在的；
- **实现线程安全的方式：**
  - ① **在 JDK1.7 的时候，** `ConcurrentHashMap` **（分段锁）** 对整个桶数组进行了分割分段(`Segment`)，每一把锁只锁容器其中一部分数据，多线程访问容器里不同数据段的数据，就不会存在锁竞争，提高并发访问率。 **到了 JDK1.8 的时候已经摒弃了** `Segment` **的概念，而是直接用** `Node` **数组+链表+红黑树的数据结构来实现，并发控制使用** `synchronized` **和 CAS 来操作。 (JDK1.6 以后 对** `synchronized` **锁做了很多优化)** 整个看起来就像是优化过且线程安全的 `HashMap`，虽然在 JDK1.8 中还能看到 `Segment` 的数据结构，但是已经简化了属性，只是为了兼容旧版本；
  - ② `Hashtable`**(同一把锁)** :使用 `synchronized` 来保证线程安全，效率非常低下。当一个线程访问同步方法时，其他线程也访问同步方法，可能会进入阻塞或轮询状态，如使用 put 添加元素，另一个线程不能使用 put 添加元素，也不能使用 get，竞争会越来越激烈效率越低。

ConcurrentHashMap 线程安全的具体实现方式/底层具体实现

JDK1.7

首先将数据分为一段一段的存储，然后给每一段数据配一把锁，当一个线程占用锁访问其中一个段数据时，其他段的数据也能被其他线程访问。

`ConcurrentHashMap` 是由 `Segment` 数组结构和 `HashEntry` 数组结构组成。

Segment 实现了 `ReentrantLock`,所以 `Segment` 是一种可重入锁，扮演锁的角色。`HashEntry` 用于存储键值对数据。

```
static class Segment<K,V> extends ReentrantLock implements Serializable
{
}
```

一个 `ConcurrentHashMap` 里包含一个 `Segment` 数组。`Segment` 的结构和 `HashMap` 类似，是一种数组和链表结构，一个 `Segment` 包含一个 `HashEntry` 数组，每个 `HashEntry` 是一个链表结构的元素，每个 `Segment` 守护着一个 `HashEntry` 数组里的元素，当对 `HashEntry` 数组的数据进行修改时，必须首先获得对应的 `Segment` 的锁。

JDK1.8

`ConcurrentHashMap` 取消了 `Segment` 分段锁，采用 CAS 和 `synchronized` 来保证并发安全。数据结构跟 HashMap1.8 的结构类似，数组+链表/红黑二叉树。Java 8 在链表长度超过一定阈值（8）时将链表（寻址时间复杂度为 O(N)）转换为红黑树（寻址时间复杂度为 O(log(N))）

`synchronized` 只锁定当前链表或红黑二叉树的首节点，这样只要 hash 不冲突，就不会产生并发，效率又提升 N 倍。

微信关注