

# 参考

JavaGuide: <https://javaguide.cn/home.html>

Java 进阶指南: <https://tobebetterjavaer.com/>

# Java

## Java 内存管理

Java 内存管理是指 Java 虚拟机如何对 Java 程序的内存进行分配、使用和回收的过程。由于 Java 是一种自动内存管理的编程语言，程序员不需要手动分配和释放内存，而是由 JVM 负责管理内存的生命周期。

主要的内存区域包括：

### 1. 堆内存

堆是 Java 程序运行存储对象实例的区域。所有通过“new”关键字创建的对象都存储在堆上。堆的大小可以通过 JVM 启动参数进行配置，它负责存储和管理大多数 Java 对象。堆内存从 JVM 启动时创建，并在程序结束时才释放。垃圾回收器负责在堆上进行垃圾回收，回收不再使用的对象以便释放内存。

### 2. 栈内存

栈用于存储方法的调用和局部变量。每个线程都有自己的栈，栈内存会随着方法调用和返回动态地分配和释放。方法的参数和局部变量存储在栈上，并在方法执行结束后立即释放。

### 3. 方法区

方法区用于存储类的信息、静态变量、常量和字节码等。它是所有线程共享的区域。在 Java8 以上的版本中，方法区被替代为元空间，用于存储类的元数据。

堆内存、栈内存、方法区都是 JVM 定义的规范，其实现方式可能会随着版本改变。比如 JDK8 之前版本的 HotSpot 方法区使用永久代实现方法区，而 JDK8 之后的 HotSpot 方法区使用元空间实现。

内存管理的核心是垃圾回收，垃圾回收器会周期性地运行，查找并回收不再使用的对象，释放堆内存空间。Java 虚拟机提供了不同类型的垃圾回收器，可以根据应用程序的需求选择合适的回收策略。

为了优化 Java 内存管理，开发者可以遵循以下几点：

### 1. 尽量避免创建不必要的对象，减少垃圾回收的负担

2. 及时释放不再使用的对象引用，避免内存泄漏。
3. 避免过度使用静态变量和全局变量，限制其生命周期，避免内存持续占用。
4. 根据应用程序的特点，调整堆内存大小和垃圾回收器的配置。

Java 中，内存泄露是指程序中的对象不再被使用时被没有并正确释放，从而造成内存的持续增加，最终导致内存资源耗尽或变得不足。内存泄露是一种常见的软件缺陷，特别在长时间运行的程序中可能导致严重的性能问题和程序崩溃。Java 是一种自动内存管理的编程语言，它通过垃圾回收机制来自动处理对象的内存分配和释放，但在编写 Java 代码时，仍然有可能发生内存泄露。

以下是一些可能导致 Java 内存泄露的常见情况：

### 1. 长生命周期的对象持有短生命周期对象的引用

如果一个长生命周期的对象持有一个短声明周期对象的引用，并长生命周期对象在后续的使用中没有释放这个引用，那么短声明周期对象就无法被垃圾回收器回收，从而导致内存泄露

```
1 package leak;
2
3 public class LongLivedObjectHasShortLivedObjectLeak {
4
5     private LivedObject longLivedObject;
6
7     public LongLivedObjectHasShortLivedObjectLeak() {
8         longLivedObject = new LivedObject();
9     }
10
11    public void doSomething() {
12        final LivedObject shortLivedObject = new LivedObject();
13        longLivedObject.setLivedObject(shortLivedObject);
14
15        // 其他操作
16
17        // 此时，longLivedObject 持有 shortLivedObject 的引用
18        // 即使在 doSomething 方法执行结束后，shortLivedObject 可能不
再被使用
19        // 由于 shortLivedObject 还在被 LongLivedObject 引用着，垃圾
回收器无法回收 shortLivedObject，造成内存泄漏
20    }
21
22
23    private static class LivedObject {
24        private LivedObject livedObject;
25
26        public void setLivedObject(LivedObject livedObject) {
27            this.livedObject = livedObject;
28        }
29    }
}
```

```
30 }
31
```

## 2. 静态集合类未正确清理

当使用静态集合类（如HashMap、ArrayList等）时，如果对象在集合中注册了监听器或回调，但未正确地从集合中移除，这些对象将无法被垃圾回收器回收，导致内存泄漏。

```
1 package leak;
2
3 import java.util.ArrayList;
4 import java.util.List;
5
6 public class StaticCollectionNotClearLeak {
7
8     private static List<String> sharedList = new ArrayList<>();
9
10
11    public static void main(String[] args) {
12        for (int i = 0 ; i < 1_000_000 ; i ++ ) {
13            addItemToSharedList("Item " + i);
14        }
15        System.out.println("Items added to the sharedList.");
16
17        // 在这里没有对sharedList进行清理或重置为null
18        // 因此，在main方法结束后，sharedList依然引用着大量的String对象
19    }
20
21    public static void addItemToSharedList(String value) {
22        sharedList.add(value);
23    }
24 }
```

## 3. 未关闭资源

Java中的一些资源，如文件流、数据库连接、网络连接等，在使用完毕后需要手动关闭，如果程序没有正确关闭这些资源，会造成资源泄漏，进而导致内存泄漏

```
1 package leak;
2
3 import util.FileUtils;
4
5 import java.io.File;
6 import java.io.FileNotFoundException;
7 import java.io.FileReader;
8 import java.io.IOException;
```

```

9 import java.util.Scanner;
10
11 public class FileResourceNotCloseLeak {
12
13     public static void main(String[] args) {
14         File file = null;
15         try {
16             file = FileUtils.createFileIfNecessary("sample.txt");
17         } catch (IOException e) {
18             e.printStackTrace();
19         }
20         try {
21             final Scanner scanner = new Scanner(new
22 FileReader(file));
23             while (scanner.hasNext()) {
24                 System.out.println(scanner.nextLine());
25             }
26             // 没有关闭 FileReader 和 scanner, 会造成文件资源泄漏
27             // reader.close() 或者 使用 try-with-resource 来确保资源
28             // 关闭
29         } catch (FileNotFoundException e) {
30             e.printStackTrace();
31         }
32     }
33 }

```

#### 4. 匿名内部类的使用

在匿名内部类中引用外部类的实例时，由于匿名类内部持有了外部类的引用，导致外部类无法被垃圾回收器回收，从而引发内存泄漏

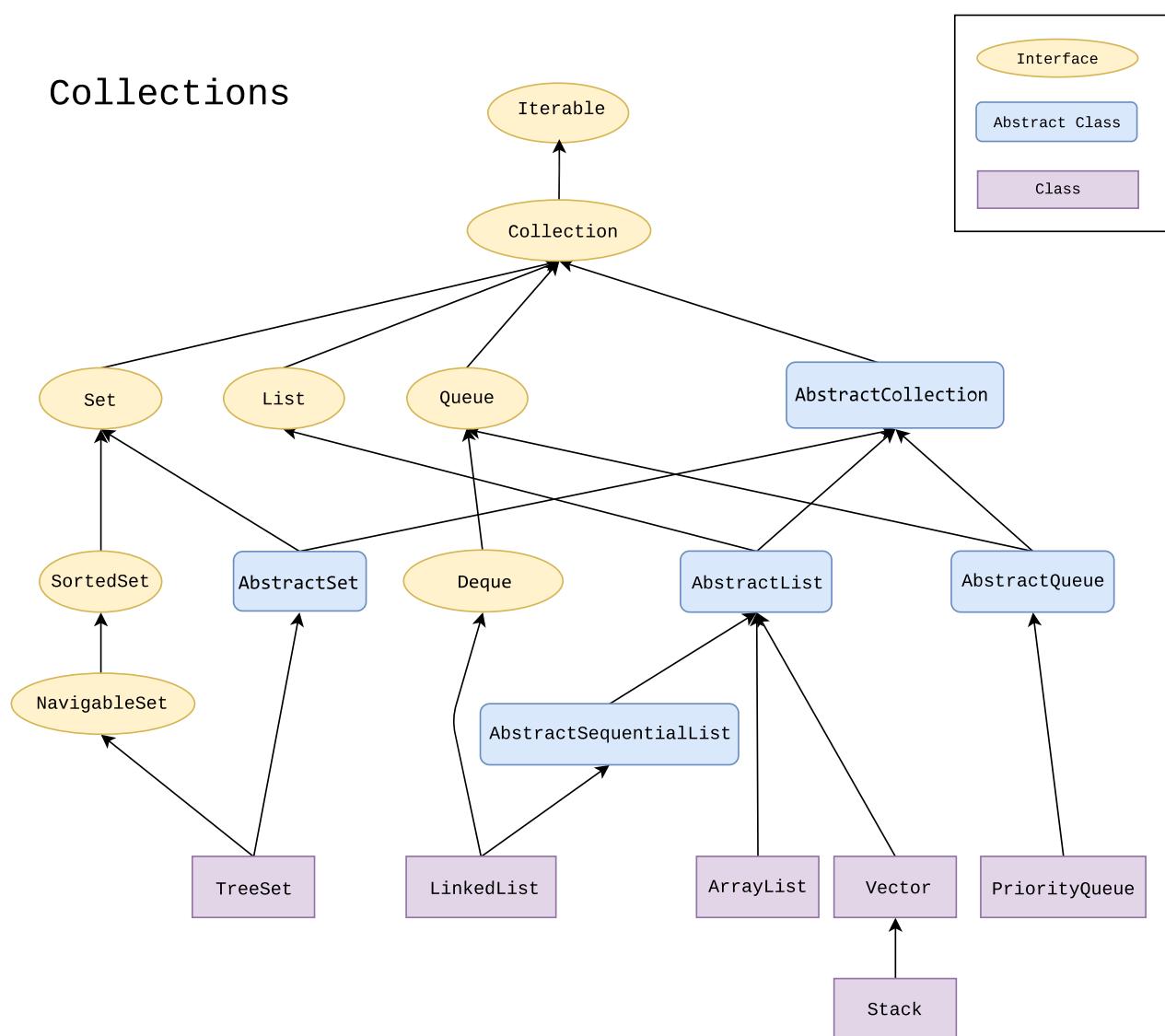
```

1 package leak;
2
3 import java.util.ArrayList;
4 import java.util.List;
5
6 public class AnonymousInnerClassLeak {
7
8     public static void main(String[] args) {
9         new AnonymousInnerClassLeak().doSomething();
10    }
11
12    public void doSomething() {
13        // 假设从这里获取了数据
14        final List<String> data = fetchData();
15    }

```

```
16 // 创建匿名内部类, 持有对外部 data 的引用
17 // 匿名内部类使用外部 data
18 final Runnable processDataRunnable = () ->
19     processData(data);
20
21     // 模拟将匿名内部类对象传给其他线程或异步任务执行
22     final Thread thread = new Thread(processDataRunnable);
23     thread.start();
24
25     // 假设这里不再需要 processDataRunnable 对象,
26     // 但由于 processDataRunnable 持有堆对 data 的引用
27     // data 无法被垃圾回收器回收, 造成内存泄漏
28 }
29
30 private void processData(List<String> data) {
31     data.stream().forEach(System.out::println);
32 }
33
34 private List<String> fetchData() {
35     final ArrayList<String> data = new ArrayList<>();
36     for (int i = 0 ; i < 1_000 ; i ++ ) {
37         data.add("data " + i);
38     }
39     return data;
40 }
41 }
42 }
```

# 集合框架



Java 集合框架是 Java 标准库中提供的一组用于存储和操作数据集合的类和接口，提供了许多实用的数据结构和算法，使得数据的管理和处理更加高效和便捷。Java 集合框架包括以下部分：

## 1. 接口

Java 集合框架定义了一些核心接口，如 **Collection**, **List**, **Set**, **Map** 等，这些接口规定了不同类型的集合的共同行为和方法。

## 2. 实现类

Java 集合框架提供了实现上述接口的具体类，开发者可以根据需求选择合适的实现类，开发者可以根据需求选择合适的实现类。例如，**ArrayList**, **LinkedList** 实现了 **List** 接口；**HashSet**, **TreeSet** 实现了 **Set** 接口，**HashMap**, **TreeMap** 实现了 **Map** 接口。

## 3. 工具类

Java 集合框架还提供了一些实用的工具类，如 `Collections` 类，它包含了许多静态方法，用于对集合进行排序、查找、填充等操作。

#### 4. 迭代器

Java 集合框架提供了迭代器 `Iterator` 接口，它允许开发者遍历集合中的元素，而不需要了解集合的底层结构。

#### 5. 泛型

Java 集合框架使用泛型来确保类型安全，使得集合可以存储特定类型的元素，避免了强制类型转换和运行时错误。

Java 集合框架可以根据数据的需求选择不同的集合类，例如：

- 使用 `ArrayList` 来存储有序的元素列表，因为它提供了快速的随机访问能力
- 使用 `HashSet` 来存储唯一元素的集合，因为它不允许重复元素
- 使用 `HashMap` 来实现键值对的存储，可以根据键查找

**Set List Queue** 三个接口都继承自 **Collection**，它们三者有什么区别

## Set

1. 表示一组不允许重复元素的集合。无序的，不能通过索引访问元素。
2. 添加到 `Set` 中的元素不会有重复的值，因为 `Set` 实现类会使用元素的哈希码来保证唯一性。
3. 常用的 `Set` 实现类有 `HashSet` (基于哈希表) `TreeSet` (基于红黑树) 等。
4. 适用于需要保持元素唯一性的场景

```
1 public class SetExample implements Giver<Set<Object>> {
2
3     @Override
4     public Set<Object> give() {
5         // 创建一个 HashSet 来存储 Object 对象
6         final HashSet<Object> set = new HashSet<>();
7
8         // 添加一些元素
9         set.add(11);
10        set.add(11); // 重复的元素不会被添加
11
12        // Set 根据 hashCode() 和 equals(Object) 来判别两个对象是否相等
13        // SameHashCodeEqualObject 类的 hashCode() 只返回 1, equals 只返回 true
14        // 那么 Set 在执行 add() 时，会将所有其他 SameHashCodeEqualObject
15        // 视作相同的对象
16        // 所以即使创建的 SameHashCodeEqualObject 对象的内容不同，但是只会被
17        // 添加一次
18        for (int i = 0 ; i < 3000 ; i ++ ) {
```

```

17         set.add(new SameHashCodeEqualObject("item " + i));
18     }
19     return set;
20 }
21
22 public static class SameHashCodeEqualObject {
23
24     private String content;
25
26     public SameHashCodeEqualObject(String content) {
27         this.content = content;
28     }
29
30     @Override
31     public int hashCode() {
32         return 1;
33     }
34
35     @Override
36     public boolean equals(Object o) {
37         return true;
38     }
39
40     @Override
41     public String toString() {
42         return "SameHashCodeObject{" +
43                 "content='" + content + '\'' +
44                 '}';
45     }
46 }
47 }
48

```

## List

1. 表示一组有序的元素列表。每个元素在列表中都有一个索引，可以根据索引来访问元素。
2. List 允许重复元素，即可以包含相同的值
3. 常用的 List 实现类有 ArrayList (动态数组) LinkedList (双向链表) 等
4. 适用于有序列表元素的场景

```

1 public class ListExample implements ExampleGiver<List<Object>> {
2     @Override
3     public List<Object> example() {
4
5         final ArrayList<Object> list = new ArrayList<>();
6

```

```
7  // 添加元素到 list
8  list.add("Alice");
9  list.add("Bob");
10 list.add("Alice");           // list 允许重复元素
11
12 list.add(114514L);
13
14 return list;
15 }
16 }
```

## Queue

1. 表示一种特殊的集合，用于存储和管理元素的顺序，通常使用 FIFO 顺序或者优先级顺序
2. Queue 主要用于任务调度、消息传递等场景
3. 常用的 Queue 实现类有 LinkedList (双向链表，可以模拟队列和双端队列)  
PriorityQueue (基于优先堆的优先队列) 等
4. Queue 接口适用于用于管理元素顺序的 (FIFO或优先级) 的场景

```
1 public class QueueExample implements ExampleGiver<Queue<Object>> {
2     @Override
3     public Queue<Object> example() {
4
5         // 创建一个 LinkedList 来模拟队列
6         final Queue<Object> queue = new LinkedList<>();
7
8         // 添加元素到队列
9         queue.offer("Alice");
10        queue.offer("Bob");
11        queue.offer("Carol");
12
13        // 使用 poll() 方法按照 先进先出FIFO 顺序获取元素并删除
14        while (! queue.isEmpty()) {
15            System.out.print(queue.poll() + "\t");
16        }
17
18        return null;
19    }
20
21 }
22 }
```

## 迭代器

在 Java 中，迭代器是一种设计模式，用于遍历集合类（如 List, Set, Map 等）中元素的对象，提供了一种统一的方式来访问集合中的元素，而无需了解底层集合的结构。

Java 中的迭代器是通过 Iterator 接口来定义的。该接口位于 java.util 包中，它定义了一组方法，允许我们在不直接操作集合底层数据结构的情况下，安全地遍历集合中的元素。

以下是 Iterator 接口常用的方法：

1. boolean hasNext() 判断是否还有下一个元素可以遍历
2. E next() 返回集合中的下一个元素
3. void remove() 从集合中移除通过 next() 方法返回的最后一个元素

在使用迭代器时，需要注意以下要点：

1. 在调用 next() 前使用 hasNext() 判断是否还有下一个元素可以遍历，并使用 next() 方法获取当前元素。通过循环遍历整个集合：
2. 避免在遍历时修改集合：当在遍历时，集合结构发生变化（例如删除和添加），迭代器会抛出此异常。
3. 迭代器提供了 remove() 方法，用于在遍历过程中移除集合中的元素。需要注意，该方法应该在调用 next() 之后、在修改集合结构之前

```
1 package iterator;
2
3 import java.util.HashSet;
4 import java.util.Iterator;
5 import java.util.List;
6
7 public class IteratorDemo {
8
9     public static void main(String[] args) {
10         String[] strings = new String[10];
11         for (int i = 0 ; i < 10 ; i ++ ) {
12             strings[i] = "Item " + i;
13         }
14
15         // Set 是无序集合，它不保持元素的插入顺序，元素之间没有明确的顺序。
16         // 因此，通过 Set 的迭代器遍历时，元素的顺序可能是不确定的。
17         System.out.println("===== Set 迭代器 =====");
18         final HashSet<String> set = new HashSet<>(List.of(strings));
19         Iterator<String> setIterator = set.iterator();
20         printIterator(setIterator);
21
22         // List 是有序集合，它保持了元素的插入顺序。
23     }
24 }
```

```

23     // 因此，通过 List 的迭代器遍历时，元素的顺序将按照插入的顺序依次遍历。
24     System.out.println("===== List 迭代器 =====");
25     final List<String> list = List.of(strings);
26     Iterator<String> listIterator = list.iterator();
27     printIterator(listIterator);
28
29     // 使用迭代器删除特定元素
30     setIterator = set.iterator();
31     removeElement(setIterator, "Item 1");
32
33
34 }
35
36 private static void printIterator(Iterator<?> iterator) {
37     // 遍历前检查是否有下一个元素
38     while (iterator.hasNext()) {
39         // 遍历过程中不允许修改集合元素
40         System.out.println(iterator.next());
41     }
42 }
43
44 private static void removeElement(Iterator<?> iterator, Object
value) {
45     // 遍历前检查是否有下一个元素
46     while (iterator.hasNext()) {
47         // 在确保 next() 获取非空元素的情况下，每次只使用一次 remove()
48         // 使用迭代器的 remove() 方法删除元素时，需要确保删除的是上一次调用
next() 方法获取的元素。
49         // 如果在调用 remove() 方法之前没有调用 next() 方法或在调用
next() 后又调用了 remove() 方法，可能会引发 IllegalStateException 异常。
50         final Object toRemove = iterator.next();
51         if (toRemove.equals(value)) {
52             iterator.remove();
53             System.out.println(toRemove);
54         }
55     }
56 }
57 }
58

```

## 集成框架底层数据结构总结

先来看一下 Collection 接口下的集合

### List

- ArrayList: Object[] 数组
- Vector: Object[] 数组
- LinkedList: 双向链表

## Map

- **HashMap**: JDK1.8 之前 **HashMap** 由数组 + 链表（也被称为桶数组 Bucket Array）组成的，数组是 **HashMap** 的主体，链表则是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）时，会将链表转换为红黑树。以减少搜索时间。

JDK11, **HashMap** 的 Bucket Array 和“阈值”

```
1  /**
2  * The table, initialized on first use, and resized as
3  * necessary. When allocated, length is always a power of two.
4  * (We also tolerate length zero in some operations to allow
5  * bootstrapping mechanics that are currently not needed.)
6  */
7 transient Node<K, V>[] table;
8
9 // 树化阈值
10 static final int TREEIFY_THRESHOLD = 8;
```

- **LinkedHashMap**: **LinkedMap** 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者红黑树组成。另外，**LinkedHashMap** 在上面的基础上，增加了一条双向链表，使得上面的结构可以保持键值对插入的顺序。同时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，数组是 **HashTable** 的主体，链表则是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）

## Set

- **HashSet**: 基于 **HashMap** 实现，底层采用 **HashMap** 来存储元素
- **LinkedHashSet**: **LinkedHashSet** 是 **HashSet** 的子类，并且其内部是通过 **LinkedHashMap** 实现的。
- **TreeSet**: 有序且唯一的红黑树，底层使用 **NavigableMap**（默认情况下是 **TreeMap**）存储数据

## List

### ArrayList 和 普通数组 的区别

**ArrayList** 内部基于动态数组实现，比普通数组使用起来更加灵活：

- **ArrayList** 会根据实际存储的元素动态地调整容量，而普通数组被创建后就不能改变它的长度了。
- **ArrayList** 允许使用泛型来确保类型安全，普通数组不可以

- `ArrayList` 中只能存储对象。对于基本类型数据，需要使用其对应的包装类（`Integer`、`Double` ...）。`ArrayList` 支持插入、删除、遍历等常见操作。并且提供了丰富的 API 方法，比如 `add`、`remove` 等。普通数组只能按照下标访问其中的元素，不具备动态添加、删除元素的功能。

```

1  /**
2   * The array buffer into which the elements of the ArrayList are
3   * stored.
4   * The capacity of the ArrayList is the length of this array
5   * buffer. Any
6   * empty ArrayList with elementData ==
7   * DEFAULTCAPACITY_EMPTY_ELEMENTDATA
8   * will be expanded to DEFAULT_CAPACITY when the first element is
9   * added.
10  */
11 transient Object[] elementData; // non-private to simplify nested
12 class access

```

- `ArrayList` 创建时不需要指定大小（但推荐指定初始大小），而普通数组创建时必须指定大小。
- `ArrayList` 是一种更灵活、更易于使用的数据结构，适用于动态管理元素集合；普通数组则适用于已知大小且不需要频繁插入、删除操作的场景

```

1 package com.congee02.list;
2
3 import java.util.ArrayList;
4 import java.util.Arrays;
5 import java.util.List;
6
7 /**
8  * 普通数组 和 ArrayList 增删改查 操作对比
9  */
10 public class ArrayListVSArray {
11
12     private static void arrayCRUD() {
13         String[] strArrays = {"hello", "world", "!"};
14         // 通过下标访问并修改
15         strArrays[0] = "goodbye";
16         System.out.println(Arrays.toString(strArrays)); // 需要借助
17         // Arrays.toString(array) 打印
18         // 删除第一个元素
19         for (int i = 0 ; i < strArrays.length - 1 ; i++) {
20             strArrays[i] = strArrays[i + 1];
21         }
22         strArrays[strArrays.length - 1] = null;
23         System.out.println(Arrays.toString(strArrays));
24     }

```

```

24
25     private static void arrayListCRUD() {
26         ArrayList<String> strList = new ArrayList<>(
27             (Arrays.asList("hello", "world", "!")));
28         // 向 strList 添加元素
29         strList.add(0, "halo");
30         strList.add("SanbornCalvin44");
31         System.out.println(strList);
32         strList.set(1, "fox");
33         System.out.println(strList);
34         strList.remove(0);
35         System.out.println(strList);
36     }
37
38     public static void main(String[] args) {
39         System.out.println("===== arrayCRUD =====");
40         arrayCRUD();
41         System.out.println("===== arrayListCRUD =====");
42         arrayListCRUD();
43     }
44 }
45

```

## ArrayList 插入和删除元素的时间复杂度

插入：

- 头部插入：由于需要将所有元素都依次向后移动一个位置，因此时间复杂度是  $O(n)$ 。
- 尾部插入：当 `ArrayList` 的容量未达到极限时，往列表末尾插入元素的时间复杂度是  $O(1)$ ，因为它只需要在数组末尾添加一个元素即可；当容量已达到极限并且需要扩容时，则需要执行一次  $O(n)$  的操作将原数组复制到新的更大的数组中，然后再执行  $O(1)$  的操作添加元素。
- 指定位置插入：需要将目标位置之后的所有元素向后移动一个位置，然后把新元素放入指定位置，时间复杂度为  $O(n)$

删除：

- 头部删除：由于需要将所有元素依次向前移动一个位置，因此时间复杂度是  $O(n)$ 。
- 尾部删除：当删除的元素位于列表末尾时，时间复杂度为  $O(1)$ 。
- 指定位置删除：需要将目标元素之后的所有元素向前移动一个位置以填补被删除的空白位置，因此需要移动平均  $n/2$  个元素，时间复杂度为  $O(n)$ 。

## LinkedList 插入和删除元素的时间复杂度

头部插入/删除：只需要修改头结点的指针即可完成插入/删除操作，因此时间复杂度为  $O(1)$ 。

尾部插入/删除：只需要修改尾结点的指针即可完成插入/删除操作，因此时间复杂度为  $O(1)$ 。

指定位置插入/删除：需要先移动到指定位置，再修改指定节点的指针完成插入/删除，因此需要移动平均  $n/2$  个元素，时间复杂度为  $O(n)$ 。

## LinkedList 为什么不能实现 RandomAccess 接口

RandomAccess 是一个标记接口，用来表明实现该接口的类支持随机访问（通过索引快速访问元素）。由于 LinkedList 底层数据结构是链表，内存地址不连续，只能通过指针来定位，而不能像数组一样使用  $Base + Offset$  随机访问，不支持随机访问。

## LinkedList 和 ArrayList 的区别

方面	ArrayList	LinkedList
线程安全	不同步，非线程安全	不同步，非线程安全
底层数据结构	Object[] 数组	双向链表
插入和删除是否受元素位置的影响	ArrayList 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	LinkedList 采用链表存储，所以在头尾插入或者删除元素不受元素位置的影响；如果是要在指定位置 $i$ 插入和删除元素的话 ( <code>add(int index, E element)</code> , <code>remove(Object o)</code> , <code>remove(int index)</code> )，时间复杂度为 $O(n)$ ，因为需要先移动到指定位置再插入和删除。
是否支持快速随机访问	支持（实现 RandomAccess 标记接口）	不支持
内存空间占用	空间浪费主要体现在 ArrayList 结尾会预留一定的空间	而 LinkedList 的空间浪费主要体现在需要额外存储后继和前驱节点

需要补充说明，需要用到 LinkedList 的场景几乎都可以使用 ArrayList 来代替，并且性能通常会更好。此外，LinkedList 不一定适合元素增删的场景，LinkedList 仅仅在头尾插入和头尾删除的情形下时间复杂度近似  $O(1)$ ，其他情况下增删的平均时间为  $O(n)$ 。

## RandomAccess 标识接口

```
1 public interface RandomAccess {  
2 }
```

RandomAccess 标识一个 Collection 接口的实现类具有随机访问功能。比如 Collections 的 binarySearch 方法，需要先判断传入的 List 是否实现了 RandomAccess，如果是调用 indexedBinarySearch，否则调用 iteratorBinarySearch 方法。

## ArrayList 扩容机制

ArrayList 扩容机制分为三个部分：

### 1. 初始容量

在创建 ArrayList 对象时，可以指定初始容量，即 Object[] 数组的初始大小，如果没有指定，默认初始容量为 10

Object[] 数组

```
1 /**
2  * The array buffer into which the elements of the ArrayList are
3  * stored.
4  * The capacity of the ArrayList is the length of this array
5  * buffer. Any
6  * empty ArrayList with elementData ==
7  * DEFAULTCAPACITY_EMPTY_ELEMENTDATA
8  * will be expanded to DEFAULT_CAPACITY when the first element is
9  * added.
10 */
11 transient Object[] elementData; // non-private to simplify nested
12 class access
```

默认初始容量

```
1 /**
2  * Default initial capacity.
3  */
4 private static final int DEFAULT_CAPACITY = 10;
```

指定初始容量

```
1 /**
2  * Constructs an empty list with the specified initial capacity.
3  *
4  * @param initialCapacity the initial capacity of the list
5  * @throws IllegalArgumentException if the specified initial
6  * capacity
7  *           is negative
8  */
9
```

```
8  public ArrayList(int initialCapacity) {
9      if (initialCapacity > 0) {
10         this.elementData = new Object[initialCapacity];
11     } else if (initialCapacity == 0) {
12         this.elementData = EMPTY_ELEMENTDATA;
13     } else {
14         throw new IllegalArgumentException("Illegal Capacity: "+
15                                         initialCapacity);
16     }
17 }
```

## 2. 添加元素

在某个 `ArrayList` 对象添加元素前，会检查元素添加后元素个数是否会达到 `Object[] elementData` 数组容量上限。如果没有，则直接插入，否则触发扩容，然后插入。

```
1 /**
2  * Inserts the specified element at the specified position in
3  * this
4  * list. Shifts the element currently at that position (if any)
5  * and
6  * any subsequent elements to the right (adds one to their
7  * indices).
8  *
9  * @param index index at which the specified element is to be
10 * inserted
11 * @param element element to be inserted
12 * @throws IndexOutOfBoundsException {@inheritDoc}
13 */
14 public void add(int index, E element) {
15     // 检查插入的索引是否正确
16     rangeCheckForAdd(index);
17     // ArrayList 修改次数（不重要）
18     modCount++;
19     final int s;
20     Object[] elementData;
21     // 如果添加元素后元素个数达到 Object[] elementData 数组上限
22     if ((s = size) == (elementData = this.elementData).length)
23         // 触发扩容
24         elementData = grow();
25     // add 逻辑，将 index 后的所有元素向后移动一格
26     System.arraycopy(elementData, index,
27                      elementData, index + 1,
28                      s - index);
29     // 将新增的 element 置于 index
30     elementData[index] = element;
31     // size 加 1
32     size = s + 1;
```

### 3. 触发扩容

当一个元素添加后达到了 Object[] 数组容量上限，则触发扩容。扩容的步骤如下：

- 创建一个新的更大的容量的 Object[] 数组（默认情况下，新数组的大小通常会是原数组大小的1.5倍，为了尽量减少频繁的扩容操作，从而提高性能）
- 将原来数组中的所有元素逐个拷贝到新数组中
- 新的数组取代原来的数组，成为当前 ArrayList 对象新的内存存储

```

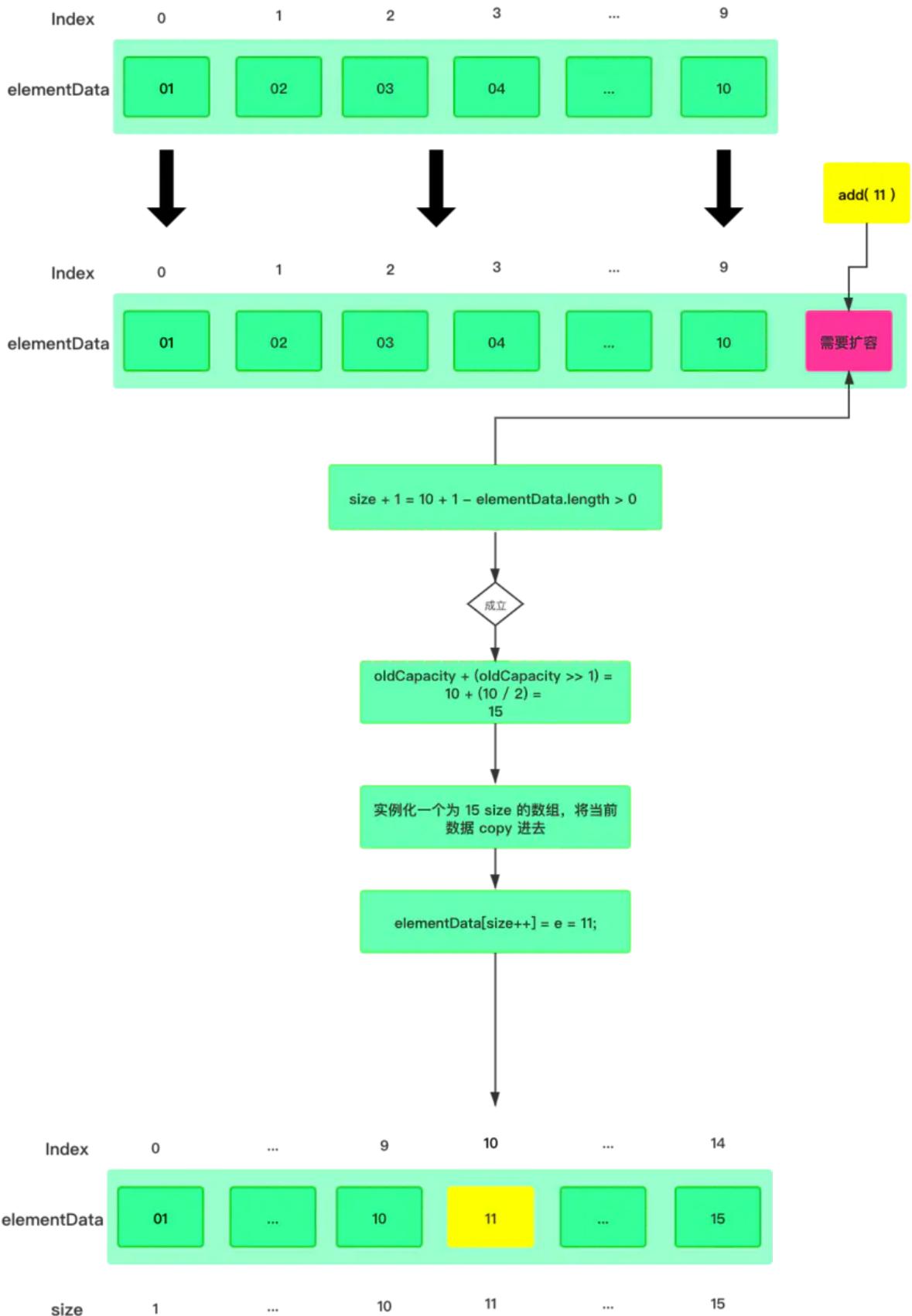
1 private Object[] grow() {
2     // 传入最小需要的容量
3     return grow(size + 1);
4 }
```

```

1 // 创建一个新的更大容量的 Object[] 数组
2 private Object[] grow(int minCapacity) {
3     return elementData = Arrays.copyOf(elementData,
4                                         newCapacity(minCapacity));
5 }
```

```

1 // 返回扩容后的容量
2 private int newCapacity(int minCapacity) {
3     // overflow-conscious code
4     int oldCapacity = elementData.length;
5     // 扩容 50%
6     int newCapacity = oldCapacity + (oldCapacity >> 1);
7
8     // 处理溢出问题
9     if (newCapacity - minCapacity <= 0) {
10         if (elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA)
11             return Math.max(DEFAULT_CAPACITY, minCapacity);
12         if (minCapacity < 0) // overflow
13             throw new OutOfMemoryError();
14         return minCapacity;
15     }
16     // 得到新扩容的容量大小
17     return (newCapacity - MAX_ARRAY_SIZE <= 0)
18         ? newCapacity
19         : hugeCapacity(minCapacity);
20 }
```



## Set

### Comparable 和 Comparator 的区别

Comparable 和 Comparator 接口都是 Java 中用于排序的接口，它们在实现类对象之间比较大小、排序等方面发挥重要作用。

- Comparable 用一个参数比较大小，方法为 compareTo(Object obj)
- Comparator 用两个参数比较大小，方法为 compare(Object obj1, Object obj2)

一般我们需要对一个集合使用自定义排序时，我们就需要重写 compareTo 方法或者 compare 方法。

### Comparator 自定义排序

```
1 package com.congee02.compare;
2
3 import java.util.ArrayList;
4 import java.util.HashSet;
5 import java.util.List;
6 import java.util.Objects;
7
8 public class UserIdSort {
9
10    private static class User {
11        private Integer id;
12        private String name;
13
14        public User(Integer id, String name) {
15            this.id = id;
16            this.name = name;
17        }
18
19        @Override
20        public String toString() {
21            return "User{" +
22                    "id=" + id +
23                    ", name='" + name + '\'' +
24                    '}';
25        }
26
27        @Override
28        public boolean equals(Object o) {
29            if (this == o) return true;
30            if (o == null || getClass() != o.getClass()) return false;
31            User user = (User) o;
32            return Objects.equals(id, user.id) && Objects.equals(name,
33            user.name);
34        }
35    }
36}
```

```

34
35     @Override
36     public int hashCode() {
37         return Objects.hash(id, name);
38     }
39 }
40
41     private static List<User> unsortedUserList() {
42         HashSet<User> userHashSet = new HashSet<>();
43         for (int i = 0 ; i < 100 ; i ++ ) {
44             userHashSet.add(new User(i, "user " + i));
45         }
46         ArrayList<User> userArrayList = new ArrayList<>(userHashSet);
47         return userArrayList;
48     }
49
50     public static void main(String[] args) {
51         List<User> userList = unsortedUserList();
52         System.out.println("===== unsorted =====");
53         userList.forEach(System.out::println);
54         userList.sort((o1, o2) -> o1.id - o2.id);
55         System.out.println("===== sorted =====");
56         userList.forEach(System.out::println);
57     }
58
59 }
60

```

## Comparable 排序

```

1 package com.congee02.compare;
2
3 import java.util.Objects;
4 import java.util.TreeMap;
5 import java.util.TreeSet;
6
7 public class UserIdComparableSort {
8     private static class ComparableUser implements
9             Comparable<ComparableUser> {
10         private Integer id;
11         private String name;
12
13         public ComparableUser(Integer id, String name) {
14             this.id = id;
15             this.name = name;
16         }
17
18         @Override
19         public String toString() {

```

```

19         return "User{" +
20                 "id=" + id +
21                 ", name='" + name + '\'' +
22                 '}';
23     }
24
25     @Override
26     public boolean equals(Object o) {
27         if (this == o) return true;
28         if (o == null || getClass() != o.getClass()) return false;
29         UserIdComparableSort.ComparableUser user =
30             (UserIdComparableSort.ComparableUser) o;
31         return Objects.equals(id, user.id) && Objects.equals(name,
32             user.name);
33     }
34
35     @Override
36     public int hashCode() {
37         return Objects.hash(id, name);
38     }
39
40     @Override
41     public int compareTo(ComparableUser o) {
42         return this.id - o.id;
43     }
44
45     public static void main(String[] args) {
46         TreeSet<ComparableUser> comparableUserTreeSet = new TreeSet<>
47             ();
48         for (int i = 0 ; i < 100 ; i ++ ) {
49             comparableUserTreeSet.add(new ComparableUser(i, "user " +
50                 i));
51         }
52     }
53 }
54 }
```

## 比较 HashSet、LinkedHashSet 和 TreeSet

- HashSet、LinkedHashSet 和 TreeSet 都是 Set 接口的实现，都能保证数据唯一性，并不都是线程安全的。
- 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）。

```
1  public class HashSet<E>
2      extends AbstractSet<E>
3      implements Set<E>, Cloneable, java.io.Serializable
4  {
5      static final long serialVersionUID = -5024744406713321676L;
6
7      // 基于 HashMap 实现
8      private transient HashMap<E, Object> map;
9
10     // 充当 Map.Entry<K, V> 中的 V, 即 Value
11     private static final Object PRESENT = new Object();
12
13     // 其他代码...
14
15     // PRESENT 的使用, 填充 Value
16     public boolean add(E e) {
17         return map.put(e, PRESENT) == null;
18     }
19
20 }
```

LinkedHashSet 的底层数据结构是链表和哈希表，元素的插入和取出满足 FIFO。

TreeSet 底层数据结构是红黑树 (TreeMap) ，元素是有序的，排序的方式有自然排序和定制排序。

```
1  public class TreeSet<E> extends AbstractSet<E>
2      implements NavigableSet<E>, Cloneable, java.io.Serializable
3  {
4      // 基于 NavigableMap 实现, 默认是 TreeMap (红黑树实现的 Map)
5      private transient NavigableMap<E, Object> m;
6
7      // 充当 Map.Entry<K, V> 中的 V, 即 Value
8      private static final Object PRESENT = new Object();
9
10     // 自行指定 NavigableMap
11     TreeSet(NavigableMap<E, Object> m) {
12         this.m = m;
13     }
14
15     // NavigableMap 默认为 TreeMap
16     public TreeSet() {
17         this(new TreeMap<>());
18     }
19 }
```

# Queue

## Queue 和 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，实现上遵循先进先出 (FIFO) 原则。Queue 扩展了 Collection 的接口，根据操作失败后处理结果的不同可以分为两类方法：一种操作失败可能会抛出异常（取决于具体实现），另一种则会返回特殊值。

Queue 接口	可能抛出异常	返回特殊值
插入队尾	add(E e)	offer(E e)
删除队首	remove()	poll()
查询队首元素	element()	peek()

### LinkedList 的 remove() poll()

```
1  /**
2  * Retrieves and removes the head (first element) of this list.
3  *
4  * @return the head of this list, or {@code null} if this list is
5  * empty
6  * @since 1.5
7  */
8  public E poll() {
9      final Node<E> f = first;
10     return (f == null) ? null : unlinkFirst(f);
11 }
12 /**
13 * Retrieves and removes the head (first element) of this list.
14 *
15 * @return the head of this list
16 * @throws NoSuchElementException if this list is empty
17 * @since 1.5
18 */
19 public E remove() {
20     return removeFirst();
21 }
22 /**
23 * Removes and returns the first element from this list.
24 *
25 * @return the first element from this list
26 * @throws NoSuchElementException if this list is empty
27 */
28 */
```

```

29  public E removeFirst() {
30      final Node<E> f = first;
31      if (f == null)
32          throw new NoSuchElementException();
33      return unlinkFirst(f);
34  }

```

Deque 是双端队列，在队列两端均可以插入或者删除元素。

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，根据上述的分类方法分为两类：

Deque 接口	可能抛出异常	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 提供 push 和 pop 方法，可以用来模拟栈。

LinkedList 的 getFisrt 和 offerFirst 源码

```

1  /**
2  * Returns the first element in this list.
3  *
4  * @return the first element in this list
5  * @throws NoSuchElementException if this list is empty
6  */
7  public E getFirst() {
8      final Node<E> f = first;
9      if (f == null)
10         throw new NoSuchElementException();
11      return f.item;
12  }
13
14 /**
15 * Retrieves, but does not remove, the first element of this list,
16 * or returns {@code null} if this list is empty.
17 *

```

```

18 * @return the first element of this list, or {@code null}
19 *         if this list is empty
20 * @since 1.6
21 */
22 public E peekFirst() {
23     final Node<E> f = first;
24     return (f == null) ? null : f.item;
25 }

```

LinkedList的addLast (不抛出异常的实现)

```

1 public void addLast(E e) {
2     linkLast(e);
3 }
4
5
6 void linkLast(E e) {
7     final Node<E> l = last;
8     final Node<E> newNode = new Node<>(l, e, null);
9     last = newNode;
10    if (l == null)
11        first = newNode;
12    else
13        l.next = newNode;
14    size++;
15    modCount++;
16 }

```

## ArrayDeque 和 LinkedList 的区别

ArrayDeque 和 LinkedList 都实现了 Deque 接口，两者都具有队列功能，但两者有什么区别呢？

- 底层结构：ArrayDeque 是基于可变长的数组和双指针来实现，而 LinkedList 则通过链表实现

ArrayDeque 底层数据结构

```

1 public class ArrayDeque<E> extends AbstractCollection<E>
2                                     implements Deque<E>, Cloneable,
3                                     Serializable
4 {
5     // 可变长数组，由扩容机制提供支持
6     transient Object[] elements;
7
8     // 头指针
9     transient int head;

```

```
10     // 尾指针
11     transient int tail;
12
13     // 其他代码...
14 }
```

## LinkedList 底层数据结构

```
1 public class LinkedList<E>
2     extends AbstractSequentialList<E>
3     implements List<E>, Deque<E>, Cloneable, java.io.Serializable
4 {
5     transient int size = 0;
6
7     // 链表头节点
8     transient Node<E> first;
9
10    // 链表尾节点
11    transient Node<E> last;
12
13    // 双链表节点
14    private static class Node<E> {
15        E item;
16        Node<E> next;
17        Node<E> prev;
18
19        Node(Node<E> prev, E element, Node<E> next) {
20            this.item = element;
21            this.next = next;
22            this.prev = prev;
23        }
24    }
25 }
```

- null 支持: ArrayDeque 不支持 null 数据, 但 LinkedList 支持  
ArrayDeque#add(E), 插入的元素为 null 时, 抛出空指针异常

```

1  public boolean add(E e) {
2      addLast(e);
3      return true;
4  }
5
6  public void addLast(E e) {
7      if (e == null)
8          throw new NullPointerException();
9      final Object[] es = elements;
10     es[tail] = e;
11     if (head == (tail = inc(tail, es.length)))
12         grow(1);
13 }

```

LinkedList#add(E)，插入的元素为 null 时，允许正常插入

```

1  public boolean add(E e) {
2      linkLast(e);
3      return true;
4  }
5
6 // 无 null 值检查，允许 null 值插入
7 void linkLast(E e) {
8     final Node<E> l = last;
9     final Node<E> newNode = new Node<>(l, e, null);
10    last = newNode;
11    if (l == null)
12        first = newNode;
13    else
14        l.next = newNode;
15    size++;
16    modCount++;
17 }

```

- 插入均摊性能：ArrayDeque 插入时可能存在扩容过程，但是均摊后插入操作时间复杂度依旧为  $O(1)$ 。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，均摊性能相对较差

ArrayDeque 触发扩容

```

1  public void addLast(E e) {
2      if (e == null)
3          throw new NullPointerException();
4      final Object[] es = elements;
5      es[tail] = e;
6      // 数组满，触发扩容
7      if (head == (tail = inc(tail, es.length)))
8          grow(1);
9 }

```

## LinkedList 插入数据时申请新的堆空间

```
1 public boolean add(E e) {
2     linkLast(e);
3     return true;
4 }
5
6 void linkLast(E e) {
7     final Node<E> l = last;
8     // new Node(l, e, null) 在堆中申请新空间以存储 Node 对象
9     final Node<E> newNode = new Node<>(l, e, null);
10    last = newNode;
11    if (l == null)
12        first = newNode;
13    else
14        l.next = newNode;
15    size++;
16    modCount++;
17 }
```

从性能角度，ArrayDeque 实现队列的性能好于 LinkedList。此外 ArrayDeque 也可以用于实现栈。

## Queue 和 Deque 对比示例

```
1 package com.congee02.queue;
2
3 import java.util.ArrayDeque;
4 import java.util.Deque;
5 import java.util.LinkedList;
6 import java.util.Queue;
7
8 public class QueueVsDeque {
9
10     private final static Queue<Object> queue = new LinkedList<>();
11     private final static Deque<Object> deque = new ArrayDeque<>();
12
13     private static void queueDemo() {
14         queue.offer(1);
15         queue.offer("World");
16         queue.offer("Hello");
17         System.out.println("Queue: ");
18         while (!queue.isEmpty()) {
19             System.out.println(queue.poll() + " ");
20         }
21     }
22
23     private static void dequeDemo() {
24         deque.offerFirst("Hello");
25         deque.offer("World");
26         deque.offer("World");
27         System.out.println("Deque: ");
28         while (!deque.isEmpty()) {
29             System.out.println(deque.pollFirst() + " ");
30         }
31     }
32 }
```

```

25     deque.offerLast("World");
26     deque.offer("!");
27     System.out.println("Deque: ");
28     while (! deque.isEmpty()) {
29         System.out.println(deque.removeFirst());
30     }
31 }
32
33 public static void main(String[] args) {
34     System.out.println("==== Queue ====");
35     queueDemo();
36     System.out.println("==== Deque ====");
37     dequeDemo();
38 }
39
40 }
41

```

## PriorityQueue

PriorityQueue (优先队列) 是 Java 中的一个队列实现，它根据元素的优先级来决定元素的顺序。与普通的队列不同，优先队列不是严格按照元素插入的先后顺序进行操作，而是根据每个元素的优先级来决定下一个要处理的元素。

PriorityQueue 使用了小顶堆数据结构实现，但可以接受一个 Comparator 作为构造器参数，从而自定义元素优先级的先后。

```

1  @SuppressWarnings("unchecked")
2  public class PriorityQueue<E> extends AbstractQueue<E>
3      implements java.io.Serializable {
4
5      // 默认堆大小
6      private static final int DEFAULT_INITIAL_CAPACITY = 11;
7
8      // 小顶堆。该数组是个动态数组，基于动态扩容机制
9      transient Object[] queue; // non-private to simplify nested class
access
10
11      // 堆的大小
12      int size;
13
14      // 自定义比较器，用于自定义排序。若为 null，则按照自然顺序排序
15      private final Comparator<? super E> comparator;
16
17      // 自动扩容
18      private void grow(int minCapacity) {

```

```

20     int oldCapacity = queue.length;
21     // Double size if small; else grow by 50%
22     int newCapacity = oldCapacity + ((oldCapacity < 64) ?
23                                         (oldCapacity + 2) :
24                                         (oldCapacity >> 1));
25     // overflow-conscious code
26     if (newCapacity - MAX_ARRAY_SIZE > 0)
27         newCapacity = hugeCapacity(minCapacity);
28     queue = Arrays.copyOf(queue, newCapacity);
29 }
30
31 // 其他代码 . . .
32 }

```

在最小堆中，父结点的值小于等于其子节点的值，这意味着优先队列中最高优先级的元素位于队列的前面。通过这种方式，可以确保每次从队列取出的元素是当前优先级最高的，主要依靠堆操作的上滤和下滤操作：

上滤操作：

```

1 // k: 填入的位置
2 // x: 填入的内容
3 private void siftUp(int k, E x) {
4     // 使用比较器对比
5     if (comparator != null)
6         siftUpUsingComparator(k, x, queue, comparator);
7     // 调用 Comparable 对比
8     else
9         siftUpComparable(k, x, queue);
10 }
11
12 private static <T> void siftUpComparable(int k, T x, Object[] es) {
13     Comparable<? super T> key = (Comparable<? super T>) x;
14     while (k > 0) {
15         int parent = (k - 1) >>> 1;
16         Object e = es[parent];
17         if (key.compareTo((T) e) >= 0)
18             break;
19         es[k] = e;
20         k = parent;
21     }
22     es[k] = key;
23 }
24
25 private static <T> void siftUpUsingComparator(
26     int k, T x, Object[] es, Comparator<? super T> cmp) {
27     while (k > 0) {
28         int parent = (k - 1) >>> 1;
29         Object e = es[parent];

```

```

30     if (cmp.compare(x, (T) e) >= 0)
31         break;
32     es[k] = e;
33     k = parent;
34 }
35 es[k] = x;
36 }

```

下滤操作：

```

1  private void siftDown(int k, E x) {
2      if (comparator != null)
3          siftDownUsingComparator(k, x, queue, size, comparator);
4      else
5          siftDownComparable(k, x, queue, size);
6  }
7
8  private static <T> void siftDownComparable(int k, T x, Object[] es,
9      int n) {
10     // assert n > 0;
11     Comparable<? super T> key = (Comparable<? super T>)x;
12     int half = n >>> 1;           // loop while a non-leaf
13     while (k < half) {
14         int child = (k << 1) + 1; // assume left child is least
15         Object c = es[child];
16         int right = child + 1;
17         if (right < n &&
18             ((Comparable<? super T>) c).compareTo((T) es[right]) > 0)
19             c = es[child = right];
20         if (key.compareTo((T) c) <= 0)
21             break;
22         es[k] = c;
23         k = child;
24     }
25     es[k] = key;
26 }
27
28 private static <T> void siftDownUsingComparator(
29     int k, T x, Object[] es, int n, Comparator<? super T> cmp) {
30     // assert n > 0;
31     int half = n >>> 1;
32     while (k < half) {
33         int child = (k << 1) + 1;
34         Object c = es[child];
35         int right = child + 1;
36         if (right < n && cmp.compare((T) c, (T) es[right]) > 0)
37             c = es[child = right];
38         if (cmp.compare(x, (T) c) <= 0)
39             break;

```

```
39     es[k] = c;
40     k = child;
41 }
42     es[k] = x;
43 }
```

其中堆数据结构是基于动态数组的，自然支持动态扩容。

## PriorityQueue 常用 API

方法	描述
add(E e) / offer(E e)	将元素插入队列，根据优先级进行排序。
remove() / poll()	移除并返回队列中的头元素，如果队列为空则返回 null。
peek()	返回队列中的头元素，但不移除它。
size()	返回队列中的元素数量。
isEmpty()	检查队列是否为空。
clear()	清空队列中的所有元素。
comparator()	返回用于排序的比较器，如果没有指定则返回 null。
toArray()	将队列转换为数组。
addAll(Collection<? extends E> c)	将一个集合中的所有元素添加到队列中。

## PriorityQueue 练习

### 找出数组中第 k 大的元素

```
1 package com.congee02.queue.practice;
2
3 import java.util.PriorityQueue;
4
5 /**
6  * 找到数据中第 k 大的元素
7 */
8 public class FindKthLargest {
9
10     private static int findKthLargest(int[] num, int k) {
11         if (k < 1) {
```

```
12         throw new IllegalArgumentException("k < 1.");
13     }
14     if (num.length < 1) {
15         throw new IllegalArgumentException("num is empty.");
16     }
17     if (num.length < k) {
18         throw new IllegalArgumentException("num.length < k");
19     }
20     if (num.length == 1) {
21         return num[0];
22     }
23     PriorityQueue<Integer> priorityQueue
24         = new PriorityQueue<>(num.length, ((o1, o2) -> o2 - o1));
25     for (int e : num) {
26         priorityQueue.add(e);
27     }
28     for (int i = 0 ; i < k - 1 ; i++) {
29         priorityQueue.poll();
30     }
31     return priorityQueue.poll();
32 }
33
34 public static void main(String[] args) {
35     System.out.println(findKthLargest(new int[]{1}, 1));
36 }
37
38 }
```

## 合并k个有序数组

```
1 package com.congee02.queue.practice;
2
3 import com.congee02.queue.practice.utils.InputGenerateUtils;
4
5 import java.util.*;
6
7 /**
8 * 合并 k 个有序数组
9 */
10 public class CombineSortedLists {
11
12     private static List<List<Integer>> randomSortedListList() {
13         List<List<Integer>> listList =
14             InputGenerateUtils.randomIntegerListList();
15         for (List<Integer> list : listList) {
16             Collections.sort(list);
17             System.out.println(list);
18         }
19     }
20 }
```

```
17
18     }
19
20
21     public static List<Integer> combineSortedLists(List<List<Integer>>
22 listList) {
23         int size = listList.size();
24         int internalSizeTotal = 0;
25         int internalSizeMax = -1;
26         int[] sizeCache = new int[size];
27         for (int i = 0 ; i < size ; i ++ ) {
28             sizeCache[i] = listList.get(i).size();
29             internalSizeMax = Math.max(internalSizeMax, sizeCache[i]);
30             internalSizeTotal += sizeCache[i];
31         }
32         PriorityQueue<Integer> queue = new
33 PriorityQueue(internalSizeTotal);
34         for (int j = 0 ; j < internalSizeMax ; j ++ ) {
35             for (int i = 0 ; i < size ; i ++ ) {
36                 if (j >= sizeCache[i]) {
37                     continue;
38                 }
39                 queue.add(listList.get(i).get(j));
40             }
41         }
42         ArrayList<Integer> result = new ArrayList<>();
43         while (! queue.isEmpty()) {
44             result.add(queue.poll());
45         }
46         return result;
47     }
48
49     public static void main(String[] args) {
50         List<List<Integer>> listList = randomSortedListList();
51         System.out.println(combineSortedLists(listList));
52     }
53 }
```

## 前K个高频元素

```
1 package com.congee02.queue;  
2  
3 import com.congee02.queue.practice.utils.InputGenerateUtils;  
4  
5 import java.util.*;  
6  
7 /**
```

```
8  * 找出 k 个频率最高的元素
9  */
10 public class FindKthFrequencyElements {
11
12     private static List<Integer> randomElementList() {
13         return InputGenerateUtils.randomIntegerList(20, 10);
14     }
15
16     private static class ElementFrequency<E> {
17         private E element;
18         private int count;
19
20         public ElementFrequency(E element) {
21             this.element = element;
22             this.count = 0;
23         }
24
25         public ElementFrequency(E element, int count) {
26             this.element = element;
27             this.count = count;
28         }
29
30         @Override
31         public String toString() {
32             return "ElementFrequency{" +
33                 "element=" + element +
34                 ", count=" + count +
35                 '}';
36         }
37     }
38
39     private static List<ElementFrequency>
40     findKthFrequencyElements(List<Integer> list, int k) {
41         if (k < 1) {
42             throw new IllegalArgumentException("k < 1.");
43         }
44         if (list.isEmpty()) {
45             throw new IllegalArgumentException("num is empty.");
46         }
47         if (list.size() == 1) {
48             return List.of(new ElementFrequency(list.get(0), 1));
49         }
50         Map<Integer, Integer> elementCount = new HashMap<>() {
51             @Override
52             public Integer put(Integer key, Integer value) {
53                 if (!this.containsKey(key)) {
54                     return super.put(key, 1);
55                 }
56                 Integer oldValue = super.get(key);
57                 if (oldValue < value) {
58                     return super.put(key, value);
59                 }
60                 return oldValue;
61             }
62         };
63         Map<Integer, Integer> sortedElementCount = elementCount.entrySet()
64             .stream()
65             .sorted(Comparator.comparingInt(Map.Entry::getValue).reversed())
66             .collect(Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue));
67         return sortedElementCount.entrySet()
68             .stream()
69             .take(k)
70             .map(Map.Entry::getKey)
71             .map(key -> list.get(key))
72             .collect(Collectors.toList());
73     }
74 }
```

```

56             return super.put(key, oldValue + 1);
57         }
58     };
59     list.forEach(e -> {
60         elementCount.put(e, null);
61     });
62     PriorityQueue<Map.Entry<Integer, Integer>> queue
63         = new PriorityQueue<>(elementCount.size(), (o1, o2) ->
64             o2.getValue() - o1.getValue());
65     Set<Map.Entry<Integer, Integer>> entries =
66     elementCount.entrySet();
67     queue.addAll(entries);
68     ArrayList<ElementFrequency> result = new ArrayList<>
69     (Math.min(k, entries.size()));
70     for (int i = 0 ; i < k && ! queue.isEmpty() ; i++) {
71         Map.Entry<Integer, Integer> entry = queue.poll();
72         result.add(new ElementFrequency(entry.getKey(),
73             entry.getValue()));
74     }
75     return result;
76 }
77
78
79 }
80

```

## 距离原点最近的K个点 (欧几里德距离和曼哈顿距离)

```

1 package com.congee02.queue.practice;
2
3 import java.util.*;
4 import java.util.function.Function;
5
6 /**
7  * 寻找前 k 个最接近原点的二维点
8 */
9 public class FindFirstKthClosestPointsToOrigin {
10
11     // 二维点类
12     private static class TwoDimensionDot {
13         private double x;
14         private double y;
15
16         public TwoDimensionDot(double x, double y) {

```

```
17         this.x = x;
18         this.y = y;
19     }
20
21     // 计算曼哈顿距离到原点的方法
22     public double manhattanDistanceToOrigin() {
23         return Math.abs(x) + Math.abs(y);
24     }
25
26     // 计算欧几里德距离到原点的方法
27     public double euclideanDistanceToOrigin() {
28         return Math.sqrt(x * x + y * y);
29     }
30
31     @Override
32     public String toString() {
33         return "TwoDimensionDot{" +
34             "x=" + x +
35             ", y=" + y +
36             '}';
37     }
38 }
39
40     private static final Random random = new Random();
41
42     // 生成随机的二维点
43     private static TwoDimensionDot randomTwoDimensionDot(double
44 xBound, double yBound) {
45         return new TwoDimensionDot((random.nextDouble() * xBound) *
46 (random.nextBoolean() ? -1 : 1), random.nextDouble() * yBound *
47 (random.nextBoolean() ? -1 : 1));
48     }
49
50     // 生成随机的二维点列表
51     private static List<TwoDimensionDot>
52 randomTwoDimensionDotList(double xBound, double yBound, int size) {
53         List<TwoDimensionDot> list = new ArrayList<>(size);
54         for (int i = 0 ; i < size ; i ++ ) {
55             list.add(randomTwoDimensionDot(xBound, yBound));
56         }
57         return list;
58     }
59
60     // 寻找前 k 个最接近原点的点
61     private static List<TwoDimensionDot>
62 firstKthClosestPointsToOrigin(List<TwoDimensionDot> dots, int k,
63 Function<TwoDimensionDot, Double> distanceCalculator) {
64         int size = dots.size();
65         if (k < 1) {
66             return dots;
67         }
68         List<TwoDimensionDot> result = new ArrayList<>(k);
69         result.add(dots.get(0));
70         for (int i = 1 ; i < size ; i ++ ) {
71             TwoDimensionDot currentDot = dots.get(i);
72             if (distanceCalculator.apply(currentDot) < distanceCalculator
73                 .apply(result.get(0))) {
74                 result.set(0, currentDot);
75             }
76         }
77         return result;
78     }
79
80     // 寻找前 k 个最远的点
81     private static List<TwoDimensionDot>
82 firstKthFurthestPointsToOrigin(List<TwoDimensionDot> dots, int k,
83 Function<TwoDimensionDot, Double> distanceCalculator) {
84         int size = dots.size();
85         if (k < 1) {
86             return dots;
87         }
88         List<TwoDimensionDot> result = new ArrayList<>(k);
89         result.add(dots.get(0));
90         for (int i = 1 ; i < size ; i ++ ) {
91             TwoDimensionDot currentDot = dots.get(i);
92             if (distanceCalculator.apply(currentDot) > distanceCalculator
93                 .apply(result.get(0))) {
94                 result.set(0, currentDot);
95             }
96         }
97         return result;
98     }
99 }
```

```
60             throw new IllegalArgumentException("k < 1.");
61         }
62         if (size < 1) {
63             throw new IllegalArgumentException("num is empty.");
64         }
65         if (size < k) {
66             throw new IllegalArgumentException("size < k");
67         }
68         if (size == 1) {
69             return List.of(dots.get(0));
70         }
71         PriorityQueue<TwoDimensionDot> queue = new PriorityQueue<>
72             (size, (o1, o2) -> {
73                 final double o1d, o2d;
74                 if ((o1d = distanceCalculator.apply(o1)) == (o2d =
75                     distanceCalculator.apply(o2))) {
76                     return 0;
77                 } else if (o1d - o2d < 0) {
78                     return -1;
79                 }
80                 return 1;
81             });
82         queue.addAll(dots);
83         ArrayList<TwoDimensionDot> result = new ArrayList<>();
84         for (int i = 0; i < k; i++) {
85             result.add(queue.poll());
86         }
87         return result;
88     }
89
90     // 寻找前 k 个最接近原点的点（使用曼哈顿距离）
91     private static List<TwoDimensionDot>
92         firstManhattanKthClosestPointsToOrigin(List<TwoDimensionDot> dots, int
93         k) {
94         return firstKthClosestPointsToOrigin(dots, k,
95             TwoDimensionDot::manhattanDistanceToOrigin);
96     }
97
98     // 寻找前 k 个最接近原点的点（使用欧几里德距离）
99     private static List<TwoDimensionDot>
100        firstEuclideanKthClosestPointsToOrigin(List<TwoDimensionDot> dots, int
101        k) {
102         return firstKthClosestPointsToOrigin(dots, k,
103             TwoDimensionDot::euclideanDistanceToOrigin);
104     }
105
106     public static void main(String[] args) {
107         // 生成随机二维点列表
108     }
109 }
```

```

100 |         List<TwoDimensionDot> dots = randomTwoDimensionDotList(20,
101 |             20, 20);
102 |             dots.forEach(System.out::println);
103 |             // 寻找前 K 个最接近原点的点 (曼哈顿距离)
104 |             System.out.println("===== Manhattan Distance =====");
105 |             firstManhattanKthClosestPointsToOrigin(dots,
106 |                 3).forEach(System.out::println);
107 |             // 寻找前 K 个最接近原点的点 (欧几里德距离)
108 |             System.out.println("===== Euclidean Distance =====");
109 |             firstEuclideanKthClosestPointsToOrigin(dots,
110 |                 3).forEach(System.out::println);
111 |
112 }
113

```

## 阻塞队列 BlockingQueue

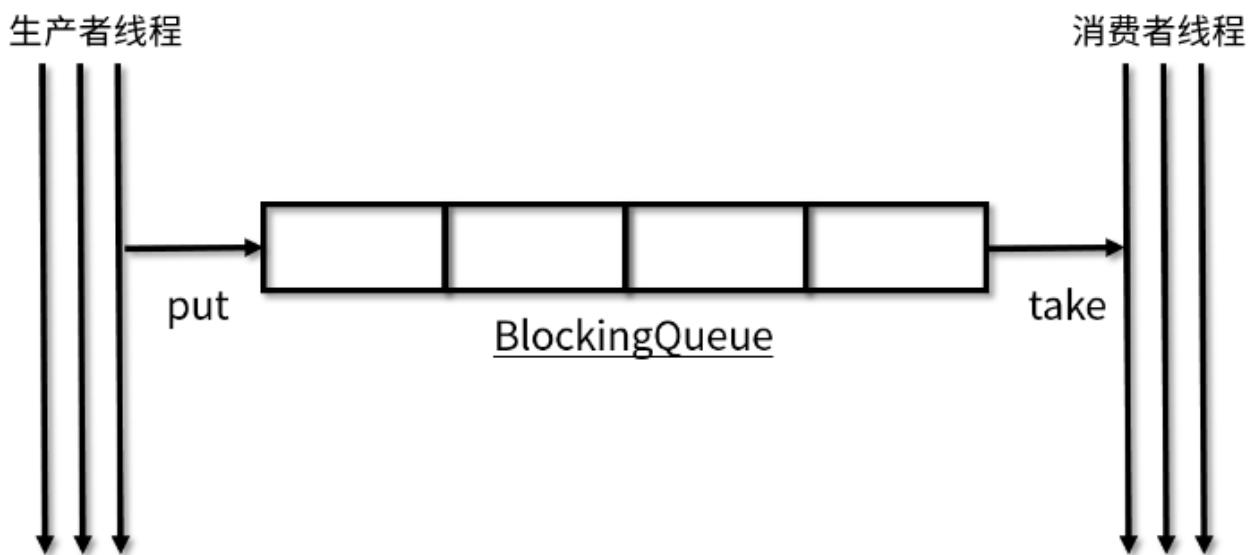
BlockingQueue 是 Java 中的一个接口，规范一个具有阻塞特性的队列。BlockingQueue 继承自 Queue 接口，提供了一组用于多线程编程的方法。

方法	描述
boolean add(E e)	将元素添加到队列，如果队列已满则抛出异常。
boolean offer(E e, long timeout, TimeUnit unit)	尝试将元素添加到队列，如果队列已满，则等待指定的时间，返回是否添加成功。
void put(E e)	将元素添加到队列，如果队列已满，则阻塞等待直到队列有空闲位置。
E poll(long timeout, TimeUnit unit)	尝试从队列中取出元素，如果队列为空，则等待指定的时间，返回取出的元素。
E take()	从队列中取出元素，如果队列为空，则阻塞等待直到队列中有可取出的元素。
int remainingCapacity()	返回队列中剩余的可用空间。
int size()	返回队列中当前的元素个数。
boolean isEmpty()	判断队列是否为空。

方法	描述
boolean contains (Object o)	判断队列是否包含指定元素。
boolean remove (Object o)	从队列中移除指定元素。
int drainTo (Collection<? super E> c)	将队列中所有元素移动到给定集合，并返回移动的元素数量。
int drainTo (Collection<? super E> c, int maxElements)	将队列中指定数量的元素移动到给定集合，并返回移动的元素数量。

当 BlockingQueue 队列已满或者为空时，调用该方法的线程会被阻塞，直到有空闲位置（尝试加入）或者队列中有可取出的元素（尝试取出）。BlockingQueue 使得线程能够在适当的时候进行等待和唤醒，从而实现了线程之间的协调和同步。

BlockingQueue 常用于 Producer-Consumer 模型中



BlockingQueue 的实现类有：PriorityBlockingQueue, LinkedBlockingQueue, ArrayBlockingQueue, DelayQueue, SynchronousQueue。

具体实现类的区别在这暂且不表，在多线程部分中详细阐述。

## Map ★

### HashMap VS HashTable

- 线程安全

HashMap 是非线程安全的（在多线程环境下使用 ConcurrentHashMap）；

Hashtable 是线程安全的，其内部方法基本经过 synchronized 修饰。

```
1 | public synchronized V put (K key, V value) {
```

```

2     // Make sure the value is not null
3     if (value == null) {
4         throw new NullPointerException();
5     }
6
7     // Makes sure the key is not already in the hashtable.
8     Entry<?, ?> tab[] = table;
9     int hash = key.hashCode();
10    int index = (hash & 0x7FFFFFFF) % tab.length;
11    @SuppressWarnings("unchecked")
12    Entry<K, V> entry = (Entry<K, V>) tab[index];
13    for (; entry != null; entry = entry.next) {
14        if ((entry.hash == hash) && entry.key.equals(key)) {
15            V old = entry.value;
16            entry.value = value;
17            return old;
18        }
19    }
20
21    addEntry(hash, key, value, index);
22    return null;
23 }

```

- 效率

Hashtable 需要保证线程安全，所以效率逊于 HashMap。

需要阐明，HashTable 基本被淘汰，不要在代码中使用。

- Null Key 和 Null Value

HashMap 支持 Null Key 和 Null Value，和其他 Key 一样，Null Key 唯一，Null Value 不唯一；

```

1 package com.congee02.map;
2
3 import java.util.HashMap;
4 import java.util.Hashtable;
5
6 public class HashTableVSHashMap {
7
8     private final static HashMap<String, String> map = new
9     HashMap<>();
10    private final static Hashtable<String, String> table = new
11    Hashtable<>();
12
13    private static void nullKeyNullValueSupport() {
14        System.out.println("===== HashMap Supports Null Key and
15        Null Value =====");
16        map.put(null, "NullKey->Value");

```

```

14         map.put("Key->NullValue", null);
15         System.out.println(map);
16         System.out.println("===== HashTable doesn't Supports Null
17 Key and Null Value =====");
18         try {
19             table.put(null, "NullKey->Value");
20             System.out.println(table);
21         } catch (NullPointerException e) {
22             System.err.println("HashTable 不支持 NullKey");
23         }
24         try {
25             table.put("Key->NullValue", null);
26             System.out.println(table);
27         } catch (NullPointerException e) {
28             System.err.println("HashTable 不支持 NullValue");
29         }
30         map.clear();
31         table.clear();
32     }
33
34     public static void main(String[] args) {
35         nullKeyNullValueSupport();
36     }
37
38 }
39

```

运行结果：

```

1 ===== HashMap Supports Null Key and Null Value =====
2 {null=NullKey->Value, Key->NullValue=null}
3 ===== HashTable doesn't Supports Null Key and Null Value =====
4 HashTable 不支持 NullKey
5 HashTable 不支持 NullValue

```

而 HashTable 不支持 Null Key 和 Null Value。注释：NullPointerException if the key or value is {@code null}。

如果键或者值是 null，则抛出 NullPointerException

```

1 /**
2  * Maps the specified {@code key} to the specified
3  * {@code value} in this hashtable. Neither the key nor the
4  * value can be {@code null}. <p>
5  *
6  * The value can be retrieved by calling the {@code get} method
7  * with a key that is equal to the original key.
8  *
9  * @param      key      the hashtable key

```

```

10  * @param      value   the value
11  * @return     the previous value of the specified key in this
12  *             hashtable,
13  *             or {@code null} if it did not have one
14  * @exception  NullPointerException if the key or value is
15  *             {@code null}
16  * @see       Object#equals(Object)
17  * @see       #get(Object)
18  */
19  public synchronized V put(K key, V value) {
20      // Make sure the value is not null
21      if (value == null) {
22          throw new NullPointerException();
23      }
24
25      // Makes sure the key is not already in the hashtable.
26      Entry<?, ?> tab[] = table;
27      int hash = key.hashCode();
28      int index = (hash & 0x7FFFFFFF) % tab.length;
29      @SuppressWarnings("unchecked")
30      Entry<K, V> entry = (Entry<K, V>) tab[index];
31      for (; entry != null; entry = entry.next) {
32          if ((entry.hash == hash) && entry.key.equals(key)) {
33              V old = entry.value;
34              entry.value = value;
35              return old;
36          }
37      }
38      addEntry(hash, key, value, index);
39      return null;
40  }

```

- 初始容量和扩充容量

HashTable 默认的初始容量为 11，之后每次扩充，容量变为原来的 2 倍加一。指定初始容量时，Hashtable 直接使用给定的初始容量。

```

1  // Hashtable 构造函数
2  public Hashtable(int initialCapacity, float loadFactor) {
3      if (initialCapacity < 0)
4          throw new IllegalArgumentException("Illegal Capacity: "+
5                                         initialCapacity);
6      if (loadFactor <= 0 || Float.isNaN(loadFactor))
7          throw new IllegalArgumentException("Illegal Load:
8                                         "+loadFactor);
9      if (initialCapacity==0)
10         initialCapacity = 1;
11         this.loadFactor = loadFactor;

```

```

12     // 直接使用给定的初始容量
13     table = new Entry<?, ?>[initialCapacity];
14     threshold = (int) Math.min(initialCapacity * loadFactor,
15         MAX_ARRAY_SIZE + 1);

```

HashMap 默认的初始容量为 16，之后每次扩充，容量变为原来的 2 倍。指定初始容量时，HashMap 会向上取大于等于给定容量最小的  $2^n$  的数作为初始容量。

```

1 // HashMap 构造函数
2 public HashMap(int initialCapacity, float loadFactor) {
3     if (initialCapacity < 0)
4         throw new IllegalArgumentException("Illegal initial
5                                         capacity: " +
6                                         initialCapacity);
7     if (initialCapacity > MAXIMUM_CAPACITY)
8         initialCapacity = MAXIMUM_CAPACITY;
9     if (loadFactor <= 0 || Float.isNaN(loadFactor))
10        throw new IllegalArgumentException("Illegal load factor:
11                                         " +
12                                         loadFactor);
13     this.loadFactor = loadFactor;
14     this.threshold = tableSizeFor(initialCapacity);
15 }
16
17 // 向上取大于等于给定容量最小的  $2^n$  的数作为初始容量。
18 static final int tableSizeFor(int cap) {
19     int n = -1 >>> Integer.numberOfLeadingZeros(cap - 1);
20     return (n < 0) ? 1 : (n >= MAXIMUM_CAPACITY) ?
21         MAXIMUM_CAPACITY : n + 1;
22 }

```

- 底层数据结构

Hashtable 使用传统的数组和链表。

```

1 public class Hashtable<K, V>
2     extends Dictionary<K, V>
3     implements Map<K, V>, Cloneable, java.io.Serializable {
4
5     /**
6      * 数组
7      */
8     private transient Entry<?, ?>[] table;
9
10    /**
11     * 链表
12     */
13    private static class Entry<K, V> implements Map.Entry<K, V> {
14        final int hash;

```

```

15     final K key;
16     V value;
17     Entry<K,V> next;
18
19     protected Entry(int hash, K key, V value, Entry<K,V>
20 next) {
21         this.hash = hash;
22         this.key = key;
23         this.value = value;
24         this.next = next;
25     }
26 }

```

在JDK1.8后，当 HashMap 长度某个链表长度大于树化阈值

(TREEIFY\_THRESHOLD, 默认为8) 时，首先在treeifyBin方法中检查当前数组的长度是否大于等于最小树化容量 (MIN\_TREEIFY\_CAPACITY, 默认为 64) ，如果否，对数组进行扩容来减少哈希冲突；如果是，将链表转化为红黑树。当一个红黑树的节点数量小于等于退化阈值时 (UNTREEIFY\_THRESHOLD, 默认为 6) ，将红黑树退化为链表。

树化阈值、退化阈值、最小树化容量

```

1 /**
2  * The bin count threshold for using a tree rather than list for
3  * a
4  * bin. Bins are converted to trees when adding an element to a
5  * bin with at least this many nodes. The value must be greater
6  * than 2 and should be at least 8 to mesh with assumptions in
7  * tree removal about conversion back to plain bins upon
8  * shrinkage.
9 */
10 static final int TREEIFY_THRESHOLD = 8;
11 /**
12  * The bin count threshold for untreeifying a (split) bin during
13  * a
14  * resize operation. Should be less than TREEIFY_THRESHOLD, and
15  * at
16  * most 6 to mesh with shrinkage detection under removal.
17 */
18 static final int UNTREEIFY_THRESHOLD = 6;
19 /**
20  * The smallest table capacity for which bins may be treeified.
21  * (Otherwise the table is resized if too many nodes in a bin.)
22  * Should be at least 4 * TREEIFY_THRESHOLD to avoid conflicts
23  * between resizing and treeification thresholds.
24 */

```

```
24 static final int MIN_TREEIFY_CAPACITY = 64;
```

尝试向 HashMap 添加一个 Entry

```
1 public V put(K key, V value) {
2     return putVal(hash(key), key, value, false, true);
3 }
4
5 // putVal 方法用于将键值对插入 HashMap
6 // hash: 键的哈希码
7 // key: 要插入的键
8 // value: 要插入的值
9 // onlyIfAbsent: 是否仅在键不存在的情况下插入
10 // evict: 是否在进行扩容时尝试删除最老的条目来释放空间
11 final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
12                 boolean evict) {
13     Node<K,V>[] tab; Node<K,V> p; int n, i;
14     // 获取当前 table 和 table 长度
15     // 如果当前 table 为 null 或者 table 为空
16     // 则初始化此 table, 并重新获取 table 和 table 的长度
17     if ((tab = table) == null || (n = tab.length) == 0)
18         n = (tab = resize()).length;
19     // 取得哈希后的值, 赋值给 i, 然后将哈希后得到的首个节点赋值给 p。
20     // 当前链表没有节点, 不存在哈希冲突
21     if ((p = tab[i = (n - 1) & hash]) == null)
22         // 直接创建新节点, 并直接插入该链表
23         tab[i] = newNode(hash, key, value, null);
24     else {
25         // 存在哈希冲突
26         Node<K,V> e; K k;
27         // 链表首个节点Key的相等 (哈希相等或者(== 或者 equals 等))
28         if (p.hash == hash &&
29             ((k = p.key) == key || (key != null &&
30             key.equals(k))))
31             // 当前 Key 的 Entry 已经存在, 将旧的 Entry 保存到 e 中
32             e = p;
33
34         // 如果当前节点为红黑树节点
35         else if (p instanceof TreeNode)
36             // 使用红黑树特有的插入方式
37             e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash,
38             key, value);
39             // 当前节点不为红黑树节点, 则向下继续寻找有没有相同的 Key
40             else {
41                 for (int binCount = 0; ; ++binCount) {
42                     // 找不到相同的 Key, 插入一个节点, 插入后,
43                     // 检查当前链表是否超过树化阈值; 若是, 则尝试将链表转化为
44                     // 红黑树
45                     if ((e = p.next) == null) {
```

```

43             p.next = newNode(hash, key, value, null);
44             if (binCount >= TREEIFY_THRESHOLD - 1) // -1
45             for 1st
46                 // 尝试将当前链表转为红黑树
47                 treeifyBin(tab, hash);
48                 break;
49             }
50             // 找到了相同的 Key, 停止寻找, 并赋值 e 给 p
51             if (e.hash == hash &&
52                 ((k = e.key) == key || (key != null &&
53                 key.equals(k))))
54                 break;
55             p = e;
56         }
57         // 已经存在该 Key 的 Entry
58         if (e != null) { // existing mapping for key
59             // 得到旧值
60             oldValue = e.value;
61             // 当 onlyIfAbsent 为 false 时, 无论键是否已存在, 都会执行
62             // 插入操作。
63             if (!onlyIfAbsent || oldValue == null)
64                 // 更新值
65                 e.value = value;
66             // 该方法暂时为空方法
67             afterNodeAccess(e);
68             // 返回更新前的旧值
69             return oldValue;
70         }
71         ++modCount;
72         // 如果当前数组大小超过阈值, 则扩展数组
73         if (++size > threshold)
74             resize();
75         // 该方法暂时为空方法
76         afterNodeInsertion(evict);
77         return null;
78     }
79
80     // 将链表树化为红黑树
81     final void treeifyBin(Node<K,V>[] tab, int hash) {
82         int n, index; Node<K,V> e;
83         // 当数组为空, 或者数组的长度小于最小树化容量时, 尝试扩展数组容量后直接
84         // 结束
85         if (tab == null || (n = tab.length) < MIN_TREEIFY_CAPACITY)
86             resize();
87         // 如果桶中存在节点
88         else if ((e = tab[index = (n - 1) & hash]) != null) {

```

```

88     TreeNode<K, V> hd = null, tl = null;
89     do {
90         // 将链表中的节点转换为红黑树节点
91         TreeNode<K, V> p = replacementTreeNode(e, null);
92         if (tl == null)
93             hd = p;
94         else {
95             p.prev = tl;
96             tl.next = p;
97         }
98         tl = p;
99     } while ((e = e.next) != null);
100    // 将转换后的红黑树设置到对应桶中
101    if ((tab[index] = hd) != null)
102        hd.treeify(tab);
103    }
104 }

```

## HashMap 的容量为什么必须是 2 的正整数次幂

当哈希码分布均匀且HashMap容量为2的正整数次幂时，以下几点是如何协同作用来减少碰撞、提高分布均匀性以及优化索引计算的：

### 1. 位运算索引计算

哈希表容量为 2 的正整数次幂，即  $2^k$ 。使得  $capacity - 1$  在二进制表示中有如下形式 111...11 (共 k 个 1)。因此通过对哈希码进行位与运算

$hashcode \& (capacity - 1)$ ，可以仅保留哈希码的低 k 位，而高位将被忽略。上述位与等价于  $hashcode \% capacity$ ，但是由于哈希表容量为 2 的正整数次幂，可以避免使用代价高昂的取模运算（取模运算本质上是一个除法操作，它需要计算被除数除以除数，然后得到余数，需要多个周期计算），而使用高效的位运算来实现取模。

### 2. 均匀分布

假设哈希码在二进制下是均匀分布的，意味着每一位的值都是随机的。当容量为 2 的正整数次幂时，取低 k 位的操作即使微小的变化也能够在索引计算中产生显著影响 ( $capacity - 1$  全为 1，可以察觉到所有低位的变化)。使得即使哈希码的高位相同，不同键在低位的微小差异也能够将其映射到不同的位置。

### 3. 碰撞减少

上述设计决策使得即使在哈希码的低位上发生微小变化，所映射的索引也会发生显著变化。因此，即使有微小的键冲突，也有很高的概率不会映射到同一个桶。这有助于减少碰撞的发生，因为键在哈希表中更均匀地分布。

# HashMap 的 loadFactor 为什么是 0.75F

## loadFactor 简述

loadFactor 是 HashMap 中的一个参数，称为加载因子，用于控制在何时触发哈希表的扩容操作，表示哈希表中当前存储的元素数量与哈希表容量的最大允许比值。

当哈希表中元素数量达到  $capacity * loadFactor$ ，会触发 HashMap 的扩容操作。loadFactor 默认为 0.75F。

```
1  /**
2  * The load factor used when none specified in constructor.
3  */
4  static final float DEFAULT_LOAD_FACTOR = 0.75f;
```

## 为什么 HashMap 的 loadFactor 默认为 0.75F

简言之，HashMap 的 loadFactor 默认为 0.75F 是出于对时间复杂度和空间复杂度的权衡而得出的经验值，具体如下：

### 1. 减少碰撞（时间复杂度）

较小的 loadFactor 值使得哈希表容量更大，减少了碰撞的可能性，从而导致链表或者红黑树的长度变长，从而影响查找效率。通过设置较小的 loadFactor，可以确保哈希表在一定程度上保持较低的碰撞率。

### 2. 空间利用（空间复杂度）

较大的 loadFactor 值会使哈希值的容量较小，节省了空间。然而过小的容量会导致碰撞变得更加频繁，从而降低了查找效率。因此，选择一个适度的 loadFactor 可以在一定程度上平衡占用空间和查找效率。

loadFactor 的默认值作为经验值，在多数情况下可以发挥好的作用。然而，有些时候，需要通过分析具体业务场景来指定具体的 loadFactor 以适应业务需求。

```
1  public HashMap(int initialCapacity, float loadFactor) {
2      if (initialCapacity < 0)
3          throw new IllegalArgumentException("Illegal initial capacity:
" +
4                                         initialCapacity);
5      if (initialCapacity > MAXIMUM_CAPACITY)
6          initialCapacity = MAXIMUM_CAPACITY;
7      if (loadFactor <= 0 || Float.isNaN(loadFactor))
8          throw new IllegalArgumentException("Illegal load factor: " +
9                                         loadFactor);
10     this.loadFactor = loadFactor;
11     this.threshold = tableSizeFor(initialCapacity);
12 }
```

## 时间换空间，空间换时间

HashMap 中 loadFactor 为 0.75F 体现了时间复杂度和空间复杂度的权衡。在一些算法中，也有类似的思想，比如时间换空间，空间换时间，以下是一些例子

### 1. 时间换空间

- 延迟线存储器：

将数据在时间上分布存储，而不是在空间上。

以小丑扔球的比喻为例，假设小丑每隔一段时间才需要某个球，而不是一直携带所有的球。这样，他可以将球分布在不同的时间点，从而减少携带的球的数量。类似地，算法中可以将数据按需计算，而不是提前全部计算，从而减少内存占用。

### 2. 空间换时间

- LongCache、IntegerCache、字符串常量池：

IntegerCache 和类似的机制在Java中是空间换时间的例子。它们将一些常用的数据（如整数、字符串等）提前计算并存储，以便在需要时能够快速访问，避免了重复计算，但占用了额外的内存空间。

- 动态规划（DP）：

在 DP 中，通过建立一个辅助的动态规划表来存储中间计算结果，避免重复计算，以此减少时间复杂度。虽然占用了一定内存空间，但能够显著减少计算量。

- 前缀和：

在处理连续子数组和类似问题时，使用前缀和可以大大降低时间复杂度。虽然需要额外空间来存储前缀和数组，但可以将求和操作的时间复杂度从  $O(n^2)$  降低到  $O(n)$

- 记忆化算法（例如斐波那契数列的计算）：

记忆化算法是一种通过存储已计算的中间结果来避免重复计算的方法，特别适用于递归算法，如斐波那契数列。这样做会占用一定的内存空间，但显著减少了计算时间。

### 3. 在现代计算机中，会更加倾向于空间换时间的策略（存储设备价格越来越低廉，容量越来越大）。在早期计算机中，会更加倾向于时间换时间的策略（存储设备昂贵且容量小，只能牺牲时间来获取降低内存占用）

## HashMap VS HashSet 的区别

HashSet 的较底层数据结构是 HashMap。HashSet 将其内容置于 HashMap 的 Key 中，利用 Map 的 Key 唯一的性质，用一个统一的 Object 对象（一个被称为PRESENT的静态常量）充当 Value，并以此实现 Set 操作。

```
1 public class HashSet<E>
2     extends AbstractSet<E>
3     implements Set<E>, Cloneable, java.io.Serializable
4 {
5     static final long serialVersionUID = -5024744406713321676L;
```

```

6
7 // 较底层数据结构为 HashMap
8 private transient HashMap<E, Object> map;
9
10 // 统一的 Object对象
11 private static final Object PRESENT = new Object();
12
13 // 基于 HashMap 添加元素
14 public boolean add(E e) {
15     return map.put(e, PRESENT) == null;
16 }
17
18 // 其他代码 ...
19
20 }

```

## HashMap 和 HashSet 的简要对比

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 VS TreeMap

HashMap 和 TreeMap 都继承自 AbstractMap，都实现了 Map 的基本功能，但是需要注意，TreeMap 在此基础上还实现了 SortedMap 接口的内容。

SortedMap 接口是对 NavigableMap 的增强。

NavigableMap 提供了一组方法，允许在 Map 中执行导航操作，比如可以获取指定范围内的子映射、查找最接近给定键的键值对等等。以下是 NavigableMap 提供的方法：

1. `ceilingEntry(K key)`：返回映射中大于等于给定键的最小键值对，如果不存在则返回 `null`。
2. `ceilingKey(K key)`：返回映射中大于等于给定键的最小键，如果不存在则返回 `null`。

3. `floorEntry(K key)`：返回映射中小于等于给定键的最大键值对，如果不存在则返回 `null`。

4. `floorKey(K key)`：返回映射中小于等于给定键的最大键，如果不存在则返回 `null`。

5. `higherEntry(K key)`：返回映射中严格大于给定键的最小键值对，如果不存在则返回 `null`。

6. `higherKey(K key)`：返回映射中严格大于给定键的最小键，如果不存在则返回 `null`。

7. `lowerEntry(K key)`：返回映射中严格小于给定键的最大键值对，如果不存在则返回 `null`。

8. `lowerKey(K key)`：返回映射中严格小于给定键的最大键，如果不存在则返回 `null`。

9. `pollFirstEntry()`：移除并返回映射中的第一个键值对，如果映射为空，则返回 `null`。

10. `pollLastEntry()`：移除并返回映射中的最后一个键值对，如果映射为空，则返回 `null`。

11. `subMap(K fromKey, boolean fromInclusive, K toKey, boolean toInclusive)`：返回指定键范围内的子映射。

12. `headMap(K toKey, boolean inclusive)`：返回键小于给定键的部分映射。

13. `tailMap(K fromKey, boolean inclusive)`：返回键大于等于给定键的部分映射。

SortedMap 在 NavigableMap 的基础上添加了键排序的相关方法：

1. `comparator()`：返回用于对键进行排序的比较器，如果使用自然顺序排序则返回 `null`。

2. `firstKey()`：返回映射中的第一个键。

3. `lastKey()`：返回映射中的最后一个键。

4. `headMap(K toKey)`：返回键小于给定键的部分映射。

5. `tailMap(K fromKey)`：返回键大于等于给定键的部分映射。

6. `subMap(K fromKey, K toKey)`：返回指定键范围内的子映射。

TreeMap 就是 SortedMap 典型的实现类，使得 TreeMap 拥有对集合中键元素排序的能力。默认情况下 TreeMap 的比较器 Comparator 为 `null`，使用自然排序（使用 Comparable 的 `compareTo(T)`），当指定 TreeMap 的 Comparator 时，使用 Comparator 进行排序。

```

1 // 指定 Comparator 来进行自定义排序
2 public TreeMap(Comparator<? super K> comparator) {
3     this.comparator = comparator;
4 }
5
6 // 默认 comparator 为 null。使用自然排序 (Comparable)
7 public TreeMap() {
8     comparator = null;
9 }

```

指定键比较器来操作 TreeMap

```

1 package com.congee02.map.tree;
2
3 import java.util.*;
4
5 public class TreeMapCustomComparator {
6
7     private static class CustomKey {
8         private String name;
9         private Integer id;
10
11         public CustomKey(String name, Integer id) {
12             this.name = name;
13             this.id = id;
14         }
15
16         @Override
17         public String toString() {
18             return "CustomKey{" +
19                     "name='" + name + '\'' +
20                     ", id=" + id +
21                     '}';
22         }
23     }
24
25     private static List<CustomKey> randomUnsortedCustomKey(int size) {
26         HashSet<CustomKey> set = new HashSet<>(size);
27         for (int i = 0 ; i < size ; i ++ ) {
28             set.add(new CustomKey("customKey " + i, i));
29         }
30         return new ArrayList<>(set);
31     }
32
33     public static void main(String[] args) {
34         int size = 5;
35         List<CustomKey> keys = randomUnsortedCustomKey(size);
36         TreeMap<CustomKey, String> treeMap = new TreeMap<>
37             (Comparator.comparingInt(k -> k.id));

```

```
37     for (int i = 0 ; i < size ; i ++ ) {
38         treeMap.put(keys.get(i), "value " + i);
39     }
40     System.out.println(treeMap);
41 }
42
43 }
44
```

打印结果：

```
1 {CustomKey{name='customKey 0', id=0}=value 0, CustomKey{name='customKey
1', id=1}=value 2, CustomKey{name='customKey 2', id=2}=value 1,
CustomKey{name='customKey 3', id=3}=value 3, CustomKey{name='customKey
4', id=4}=value 4}
```

简言之，TreeMap 相较于 HashMap 多了对集合中的元素根据键排序（SortedMap）的能力和对集合内元素搜索（NavigableMap）的能力。

## HashSet 如何检查重复

上面提到 HashSet 依靠 HashMap 实现，HashMap 如何检查重复键，与 HashSet 检查重复的机制几乎雷同。

需要复述的是，HashSet（当然 HashMap 也是）判断两个对象相同，先要检查哈希值相同，如果哈希值相同，再检查 equals 是否为 true（如果当前 Key 为 null 则用 == 判断是否相等）。

```
1 // 先判断哈希值是否相同，若是，则进一步使用 equals 判断（如果当前 Key 为 null 使用 == 判断相等）
2 if (p.hash == hash &&
3     ((k = p.key) == key || (key != null && key.equals(k)))) {
4     // ...
5 }
```

这样做的原因是：对象之间的哈希也可能存在哈希碰撞，哈希值相同并不一定保证两对象相同，还需要使用 equals 进一步判断（如果当前 Key 为 null 则用 == 判断是否相等）。所以在写实体类时，在重写 hashCode() 的同时还需要重写 equals(Object)。

## HashMap 多线程操作导致死循环

JDK1.7 及之前版本的 HashMap 在多线程环境下扩容操作可能存在死循环问题，这是由于一个桶位中有多个元素需要扩容时，多个线程对链表进行操作，头插法可能会导致链表中的节点执行错误的位置，从而形成一个环形链表，进而使查询元素的操作陷入死循环，无法结束。

为了解决这个问题，JDK1.8 的 HashMap 使用尾插法而非头插法来避免链表中的环形结构。

在实际开发中，推荐使用 ConcurrentHashMap 而非 HashMap 或者 Hashtable

## HashMap 为什么线程不安全

JDK1.7 及其之前版本，在多线情况下可能会引发死循环或者数据丢失问题。

上面已经讲过死循环问题，这里讲讲 JDK1.8 及其以前版本出现的数据丢失问题：

- 两个线程 1,2 同时进行 put 操作，并且发生了哈希冲突（hash 函数计算出的插入下标是相同的）。
- 不同的线程可能在不同的时间片获得 CPU 执行的机会，当前线程 1 执行完哈希冲突判断后，但是未完成插入操作，由于时间片耗尽挂起。线程 2 先完成了插入操作。
- 随后，线程 1 获得时间片，由于之前已经进行过 hash 碰撞的判断，所有此时会直接进行插入，这就导致线程 2 插入的数据被线程 1 覆盖了。

## HashMap 常见的遍历方式

- KeySet 迭代器

优点：适用于只需遍历键的场景，对键值对的访问效率较高

缺点：如果需要获取对应的值，需要再次通过键进行查找，此时性能较差

- EntrySet 迭代器

优点：同时获取键和值，遍历过程中无需其他额外操作

缺点：相较于仅遍历键的方式，稍微多一些代码，但效率更高

- KeySet For 循环

优点：同时获取键和值，性能和可读性都比较好。

缺点：相对于其他简单遍历方式，代码量稍微多一些。

- EntrySet For 循环

优点：同时获取键和值，性能和可读性都比较好。

缺点：相对于其他简单遍历方式，代码量稍微多一些。

- Lambda

优点：使用 Lambda 表达式，代码更为简洁，适合简单遍历和处理操作。

缺点：相对于其他方式，可能在性能上稍有损失，尤其在大规模数据集上。

- 单线程 Stream

优点：适用于数据处理和转换，可以结合 Lambda 表达式进行操作。

缺点：在处理大数据集时，可能没有明显的性能提升，而且需要考虑线程安全。

- 多线程 Stream

优点：适用于大规模数据集的并行处理，可以充分利用多核处理器提高性能。

缺点：需要考虑线程安全、并发性和遍历顺序不确定性的问题，不适合有状态操作。

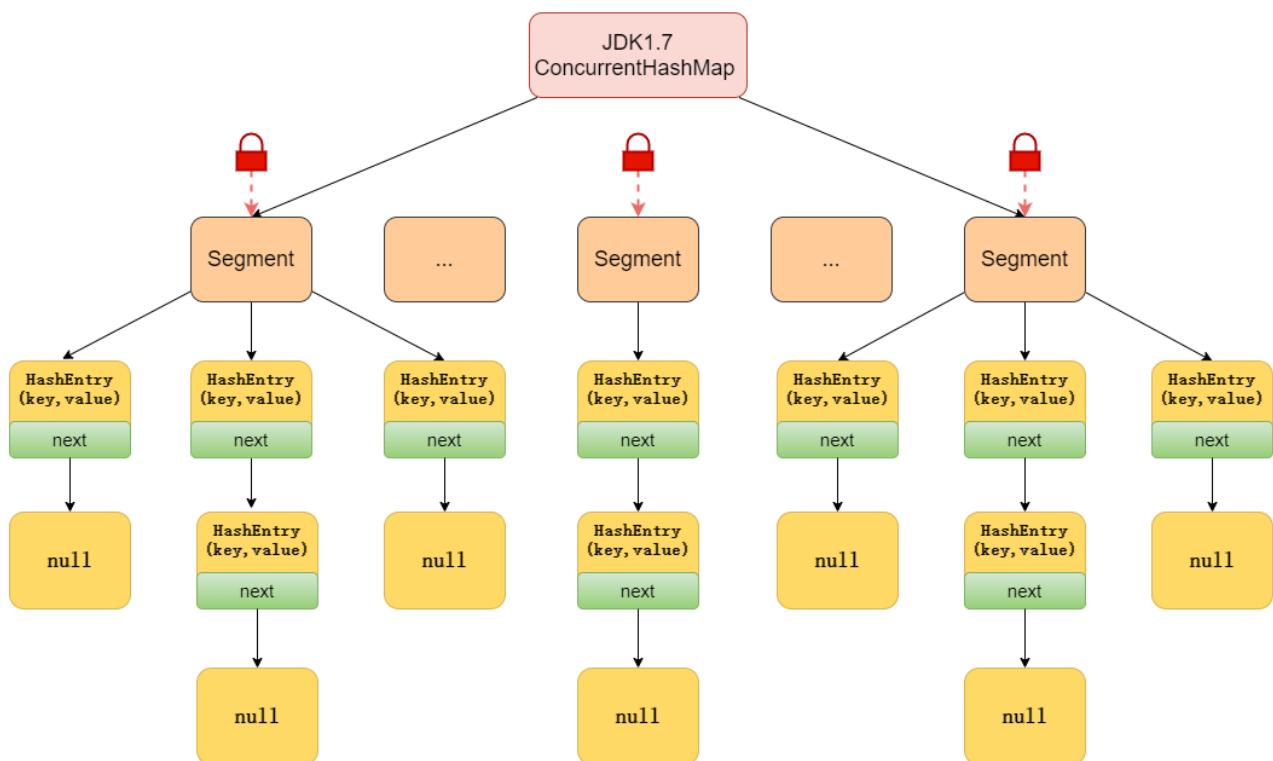
- 使用场景

- 适用场景：

- 如果只关心键而不需要值，可以使用 KeySet 迭代器或 KeySet For 循环。
- 如果需要同时获取键和值，EntrySet 迭代器和 EntrySet For 循环是更好的选择。
- 使用 Lambda 表达式进行简单的遍历和操作，可以考虑使用 forEach 方法配合 Lambda。
- 对于大规模数据集的数据处理，可以考虑使用单线程 Stream 或多线程 Stream 来提高性能。
- 在并行处理时，考虑数据安全和遍历顺序不确定性，避免有状态操作。另外并行操作特别适合阻塞场景。

## ConcurrentHashMap 如何实现线程安全

当谈及 ConcurrentHashMap 如何实现线程安全时，需要分别考虑 JDK1.7 和 JDK1.8 的实现。



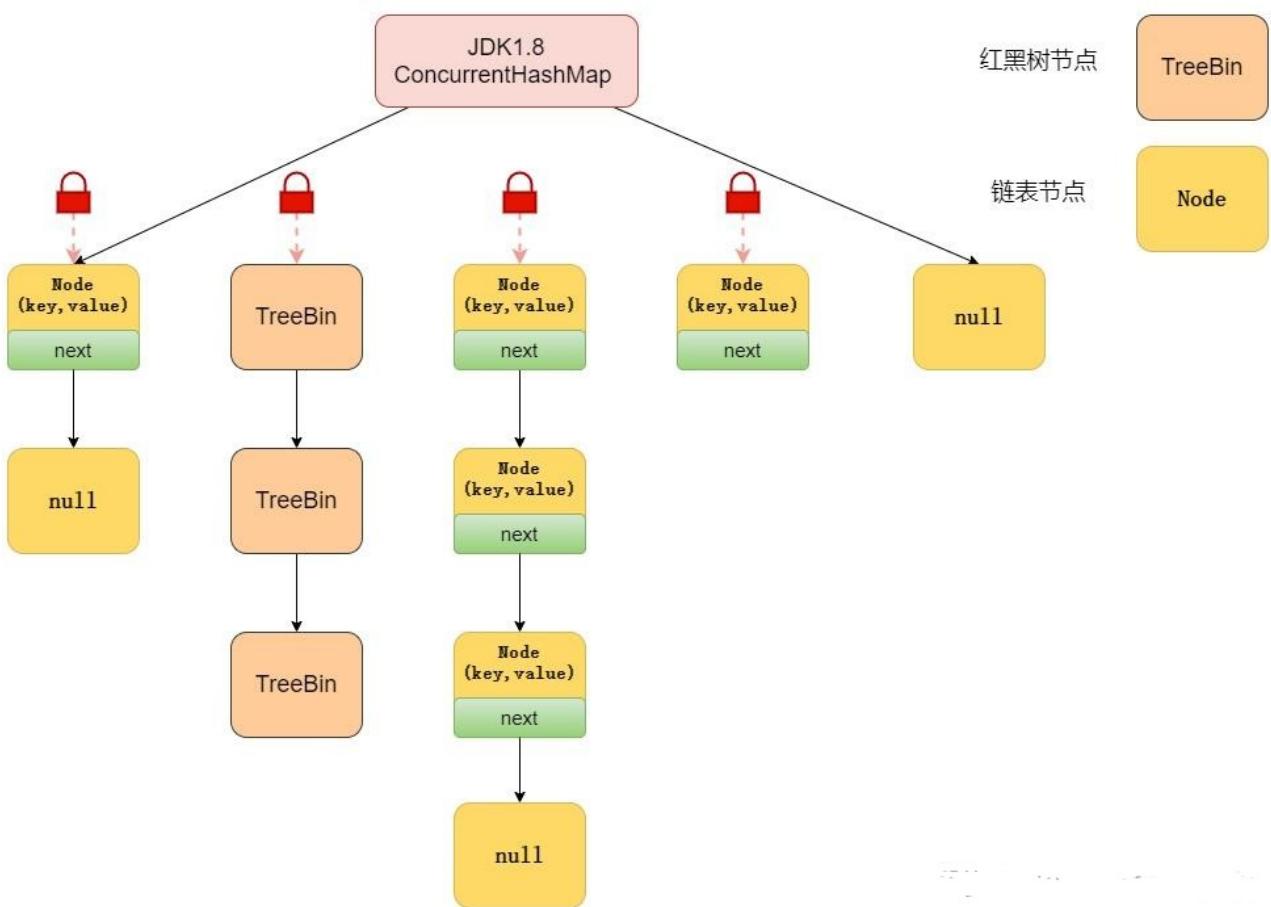
如上图所示，JDK1.7 中，ConcurrentHashMap 主要由 Segment 段数组组成，而 Segment 由 HashEntry 条目数组组成。ConcurrentHashMap 将整个哈希桶数组分成若干个 Segment 段，并且为每个 Segment 段上锁。如此，就允许多个线程分别同时访问不同的 Segment 段，只有多个线程访问同一个 Segment 段时，才发生锁的竞争，以此提高 ConcurrentHashMap 的并发度。并发度默认为 16。其锁的级别为 Segment 段。

### JDK1.7 ConcurrentHashMap 部分源码

```
1  public class ConcurrentHashMap<K, V> extends AbstractMap<K, V>
2      implements ConcurrentMap<K, V>, Serializable {
3
4      /**
5       * The number of segments in this map.  Higher values allow
6       * for more parallelism in iteration and in parallel
7       * computation.  The default value is 16.
8       */
9      private static final int Segments = 16;
```

```
5     * 默认并发度
6     */
7     static final int DEFAULT_CONCURRENCY_LEVEL = 16;
8
9     /**
10      * JDK1.7 ConcurrentHashMap 中的 Segment 数组
11      */
12     final Segment<K,V>[] segments;
13
14     // 链表条目
15     static final class HashEntry<K,V> {
16         final int hash;
17         final K key;
18
19         // volatile 保证多线程可见性
20         volatile V value;
21         volatile HashEntry<K,V> next;
22
23         HashEntry(int hash, K key, V value, HashEntry<K,V> next) {
24             this.hash = hash;
25             this.key = key;
26             this.value = value;
27             this.next = next;
28         }
29
30         // ...
31     }
32
33     // Segment 继承 ReentrantLock 重入锁, 扮演锁的作用
34     static final class Segment<K,V> extends ReentrantLock implements
35     Serializable {
36
37     /**
38      * The maximum number of times to tryLock in a prescan before
39      * possibly blocking on acquire in preparation for a locked
40      * segment operation. On multiprocessors, using a bounded
41      * number of retries maintains cache acquired while locating
42      * nodes.
43      */
44     static final int MAX_SCAN_RETRIES =
45         Runtime.getRuntime().availableProcessors() > 1 ? 64 : 1;
46
47     // 存储的 HashEntry
48     transient volatile HashEntry<K,V>[] table;
49
50     // 元素的个数
51     transient int count;
52
53     // ...
54 }
```

53 | }  
54  
55 } }



如上图所示，JDK1.8 中，ConcurrentHashMap 采用了 Node 链表 或者红黑树 的结构；在锁的实现上，弃用 Segment 分段锁，采用 CAS + synchronized 实现更加细粒度的锁。

JDK1.8 putVal 使用 CAS + synchronized 实现插入一条 Entry

```
1 static final int MOVED = -1;
2
3 /** Implementation for put and putIfAbsent */
4 final V putVal(K key, V value, boolean onlyIfAbsent) {
5
6     if (key == null || value == null) throw new
7 NullPointerException();
8
9     // 1. 计算 hash 值
10    int hash = spread(key.hashCode());
11
12    int binCount = 0;
13
14    for (Node<K,V>[] tab = table;;) {
15        Node<K,V> f; int n, i, fh;
16
17        // 2. 判断是否需要初始化
18
19        if (tab == null || (n = tab.length) == 0 ||
20            (fh = f.floorHash(hash)) < 0 ||
21            (i = f.findSlotFor(hash)) < 0) {
22
23            if (tab == null || n == 0) {
24                if (n == 0)
25                    n = MOVED;
26
27                if (n > 0)
28                    tab = createTab(n);
29
30                if (tab == null)
31                    throw new OutOfMemoryError("Allocating " +
32                        "table with capacity " + n);
33
34                if (n == MOVED)
35                    return null;
36
37                if (fh < 0)
38                    fh = tab.length;
39
40                if (fh < 0)
41                    fh = tab.length;
42
43                if (fh < 0)
44                    fh = tab.length;
45
46                if (fh < 0)
47                    fh = tab.length;
48
49                if (fh < 0)
50                    fh = tab.length;
51
52                if (fh < 0)
53                    fh = tab.length;
54
55                if (fh < 0)
56                    fh = tab.length;
57
58                if (fh < 0)
59                    fh = tab.length;
60
61                if (fh < 0)
62                    fh = tab.length;
63
64                if (fh < 0)
65                    fh = tab.length;
66
67                if (fh < 0)
68                    fh = tab.length;
69
70                if (fh < 0)
71                    fh = tab.length;
72
73                if (fh < 0)
74                    fh = tab.length;
75
76                if (fh < 0)
77                    fh = tab.length;
78
79                if (fh < 0)
80                    fh = tab.length;
81
82                if (fh < 0)
83                    fh = tab.length;
84
85                if (fh < 0)
86                    fh = tab.length;
87
88                if (fh < 0)
89                    fh = tab.length;
90
91                if (fh < 0)
92                    fh = tab.length;
93
94                if (fh < 0)
95                    fh = tab.length;
96
97                if (fh < 0)
98                    fh = tab.length;
99
100               if (fh < 0)
101                   fh = tab.length;
102
103               if (fh < 0)
104                   fh = tab.length;
105
106               if (fh < 0)
107                   fh = tab.length;
108
109               if (fh < 0)
110                   fh = tab.length;
111
112               if (fh < 0)
113                   fh = tab.length;
114
115               if (fh < 0)
116                   fh = tab.length;
117
118               if (fh < 0)
119                   fh = tab.length;
120
121               if (fh < 0)
122                   fh = tab.length;
123
124               if (fh < 0)
125                   fh = tab.length;
126
127               if (fh < 0)
128                   fh = tab.length;
129
130               if (fh < 0)
131                   fh = tab.length;
132
133               if (fh < 0)
134                   fh = tab.length;
135
136               if (fh < 0)
137                   fh = tab.length;
138
139               if (fh < 0)
140                   fh = tab.length;
141
142               if (fh < 0)
143                   fh = tab.length;
144
145               if (fh < 0)
146                   fh = tab.length;
147
148               if (fh < 0)
149                   fh = tab.length;
150
151               if (fh < 0)
152                   fh = tab.length;
153
154               if (fh < 0)
155                   fh = tab.length;
156
157               if (fh < 0)
158                   fh = tab.length;
159
160               if (fh < 0)
161                   fh = tab.length;
162
163               if (fh < 0)
164                   fh = tab.length;
165
166               if (fh < 0)
167                   fh = tab.length;
168
169               if (fh < 0)
170                   fh = tab.length;
171
172               if (fh < 0)
173                   fh = tab.length;
174
175               if (fh < 0)
176                   fh = tab.length;
177
178               if (fh < 0)
179                   fh = tab.length;
180
181               if (fh < 0)
182                   fh = tab.length;
183
184               if (fh < 0)
185                   fh = tab.length;
186
187               if (fh < 0)
188                   fh = tab.length;
189
190               if (fh < 0)
191                   fh = tab.length;
192
193               if (fh < 0)
194                   fh = tab.length;
195
196               if (fh < 0)
197                   fh = tab.length;
198
199               if (fh < 0)
200                   fh = tab.length;
201
202               if (fh < 0)
203                   fh = tab.length;
204
205               if (fh < 0)
206                   fh = tab.length;
207
208               if (fh < 0)
209                   fh = tab.length;
210
211               if (fh < 0)
212                   fh = tab.length;
213
214               if (fh < 0)
215                   fh = tab.length;
216
217               if (fh < 0)
218                   fh = tab.length;
219
220               if (fh < 0)
221                   fh = tab.length;
222
223               if (fh < 0)
224                   fh = tab.length;
225
226               if (fh < 0)
227                   fh = tab.length;
228
229               if (fh < 0)
230                   fh = tab.length;
231
232               if (fh < 0)
233                   fh = tab.length;
234
235               if (fh < 0)
236                   fh = tab.length;
237
238               if (fh < 0)
239                   fh = tab.length;
240
241               if (fh < 0)
242                   fh = tab.length;
243
244               if (fh < 0)
245                   fh = tab.length;
246
247               if (fh < 0)
248                   fh = tab.length;
249
250               if (fh < 0)
251                   fh = tab.length;
252
253               if (fh < 0)
254                   fh = tab.length;
255
256               if (fh < 0)
257                   fh = tab.length;
258
259               if (fh < 0)
260                   fh = tab.length;
261
262               if (fh < 0)
263                   fh = tab.length;
264
265               if (fh < 0)
266                   fh = tab.length;
267
268               if (fh < 0)
269                   fh = tab.length;
270
271               if (fh < 0)
272                   fh = tab.length;
273
274               if (fh < 0)
275                   fh = tab.length;
276
277               if (fh < 0)
278                   fh = tab.length;
279
280               if (fh < 0)
281                   fh = tab.length;
282
283               if (fh < 0)
284                   fh = tab.length;
285
286               if (fh < 0)
287                   fh = tab.length;
288
289               if (fh < 0)
290                   fh = tab.length;
291
292               if (fh < 0)
293                   fh = tab.length;
294
295               if (fh < 0)
296                   fh = tab.length;
297
298               if (fh < 0)
299                   fh = tab.length;
300
301               if (fh < 0)
302                   fh = tab.length;
303
304               if (fh < 0)
305                   fh = tab.length;
306
307               if (fh < 0)
308                   fh = tab.length;
309
310               if (fh < 0)
311                   fh = tab.length;
312
313               if (fh < 0)
314                   fh = tab.length;
315
316               if (fh < 0)
317                   fh = tab.length;
318
319               if (fh < 0)
320                   fh = tab.length;
321
322               if (fh < 0)
323                   fh = tab.length;
324
325               if (fh < 0)
326                   fh = tab.length;
327
328               if (fh < 0)
329                   fh = tab.length;
330
331               if (fh < 0)
332                   fh = tab.length;
333
334               if (fh < 0)
335                   fh = tab.length;
336
337               if (fh < 0)
338                   fh = tab.length;
339
340               if (fh < 0)
341                   fh = tab.length;
342
343               if (fh < 0)
344                   fh = tab.length;
345
346               if (fh < 0)
347                   fh = tab.length;
348
349               if (fh < 0)
350                   fh = tab.length;
351
352               if (fh < 0)
353                   fh = tab.length;
354
355               if (fh < 0)
356                   fh = tab.length;
357
358               if (fh < 0)
359                   fh = tab.length;
360
361               if (fh < 0)
362                   fh = tab.length;
363
364               if (fh < 0)
365                   fh = tab.length;
366
367               if (fh < 0)
368                   fh = tab.length;
369
370               if (fh < 0)
371                   fh = tab.length;
372
373               if (fh < 0)
374                   fh = tab.length;
375
376               if (fh < 0)
377                   fh = tab.length;
378
379               if (fh < 0)
380                   fh = tab.length;
381
382               if (fh < 0)
383                   fh = tab.length;
384
385               if (fh < 0)
386                   fh = tab.length;
387
388               if (fh < 0)
389                   fh = tab.length;
390
391               if (fh < 0)
392                   fh = tab.length;
393
394               if (fh < 0)
395                   fh = tab.length;
396
397               if (fh < 0)
398                   fh = tab.length;
399
400               if (fh < 0)
401                   fh = tab.length;
402
403               if (fh < 0)
404                   fh = tab.length;
405
406               if (fh < 0)
407                   fh = tab.length;
408
409               if (fh < 0)
410                   fh = tab.length;
411
412               if (fh < 0)
413                   fh = tab.length;
414
415               if (fh < 0)
416                   fh = tab.length;
417
418               if (fh < 0)
419                   fh = tab.length;
420
421               if (fh < 0)
422                   fh = tab.length;
423
424               if (fh < 0)
425                   fh = tab.length;
426
427               if (fh < 0)
428                   fh = tab.length;
429
430               if (fh < 0)
431                   fh = tab.length;
432
433               if (fh < 0)
434                   fh = tab.length;
435
436               if (fh < 0)
437                   fh = tab.length;
438
439               if (fh < 0)
440                   fh = tab.length;
441
442               if (fh < 0)
443                   fh = tab.length;
444
445               if (fh < 0)
446                   fh = tab.length;
447
448               if (fh < 0)
449                   fh = tab.length;
450
451               if (fh < 0)
452                   fh = tab.length;
453
454               if (fh < 0)
455                   fh = tab.length;
456
457               if (fh < 0)
458                   fh = tab.length;
459
460               if (fh < 0)
461                   fh = tab.length;
462
463               if (fh < 0)
464                   fh = tab.length;
465
466               if (fh < 0)
467                   fh = tab.length;
468
469               if (fh < 0)
470                   fh = tab.length;
471
472               if (fh < 0)
473                   fh = tab.length;
474
475               if (fh < 0)
476                   fh = tab.length;
477
478               if (fh < 0)
479                   fh = tab.length;
480
481               if (fh < 0)
482                   fh = tab.length;
483
484               if (fh < 0)
485                   fh = tab.length;
486
487               if (fh < 0)
488                   fh = tab.length;
489
490               if (fh < 0)
491                   fh = tab.length;
492
493               if (fh < 0)
494                   fh = tab.length;
495
496               if (fh < 0)
497                   fh = tab.length;
498
499               if (fh < 0)
500                   fh = tab.length;
501
502               if (fh < 0)
503                   fh = tab.length;
504
505               if (fh < 0)
506                   fh = tab.length;
507
508               if (fh < 0)
509                   fh = tab.length;
510
511               if (fh < 0)
512                   fh = tab.length;
513
514               if (fh < 0)
515                   fh = tab.length;
516
517               if (fh < 0)
518                   fh = tab.length;
519
520               if (fh < 0)
521                   fh = tab.length;
522
523               if (fh < 0)
524                   fh = tab.length;
525
526               if (fh < 0)
527                   fh = tab.length;
528
529               if (fh < 0)
530                   fh = tab.length;
531
532               if (fh < 0)
533                   fh = tab.length;
534
535               if (fh < 0)
536                   fh = tab.length;
537
538               if (fh < 0)
539                   fh = tab.length;
540
541               if (fh < 0)
542                   fh = tab.length;
543
544               if (fh < 0)
545                   fh = tab.length;
546
547               if (fh < 0)
548                   fh = tab.length;
549
550               if (fh < 0)
551                   fh = tab.length;
552
553               if (fh < 0)
554                   fh = tab.length;
555
556               if (fh < 0)
557                   fh = tab.length;
558
559               if (fh < 0)
560                   fh = tab.length;
561
562               if (fh < 0)
563                   fh = tab.length;
564
565               if (fh < 0)
566                   fh = tab.length;
567
568               if (fh < 0)
569                   fh = tab.length;
570
571               if (fh < 0)
572                   fh = tab.length;
573
574               if (fh < 0)
575                   fh = tab.length;
576
577               if (fh < 0)
578                   fh = tab.length;
579
580               if (fh < 0)
581                   fh = tab.length;
582
583               if (fh < 0)
584                   fh = tab.length;
585
586               if (fh < 0)
587                   fh = tab.length;
588
589               if (fh < 0)
590                   fh = tab.length;
591
592               if (fh < 0)
593                   fh = tab.length;
594
595               if (fh < 0)
596                   fh = tab.length;
597
598               if (fh < 0)
599                   fh = tab.length;
600
601               if (fh < 0)
602                   fh = tab.length;
603
604               if (fh < 0)
605                   fh = tab.length;
606
607               if (fh < 0)
608                   fh = tab.length;
609
610               if (fh < 0)
611                   fh = tab.length;
612
613               if (fh < 0)
614                   fh = tab.length;
615
616               if (fh < 0)
617                   fh = tab.length;
618
619               if (fh < 0)
620                   fh = tab.length;
621
622               if (fh < 0)
623                   fh = tab.length;
624
625               if (fh < 0)
626                   fh = tab.length;
627
628               if (fh < 0)
629                   fh = tab.length;
630
631               if (fh < 0)
632                   fh = tab.length;
633
634               if (fh < 0)
635                   fh = tab.length;
636
637               if (fh < 0)
638                   fh = tab.length;
639
640               if (fh < 0)
641                   fh = tab.length;
642
643               if (fh < 0)
644                   fh = tab.length;
645
646               if (fh < 0)
647                   fh = tab.length;
648
649               if (fh < 0)
650                   fh = tab.length;
651
652               if (fh < 0)
653                   fh = tab.length;
654
655               if (fh < 0)
656                   fh = tab.length;
657
658               if (fh < 0)
659                   fh = tab.length;
660
661               if (fh < 0)
662                   fh = tab.length;
663
664               if (fh < 0)
665                   fh = tab.length;
666
667               if (fh < 0)
668                   fh = tab.length;
669
670               if (fh < 0)
671                   fh = tab.length;
672
673               if (fh < 0)
674                   fh = tab.length;
675
676               if (fh < 0)
677                   fh = tab.length;
678
679               if (fh < 0)
680                   fh = tab.length;
681
682               if (fh < 0)
683                   fh = tab.length;
684
685               if (fh < 0)
686                   fh = tab.length;
687
688               if (fh < 0)
689                   fh = tab.length;
690
691               if (fh < 0)
692                   fh = tab.length;
693
694               if (fh < 0)
695                   fh = tab.length;
696
697               if (fh < 0)
698                   fh = tab.length;
699
700               if (fh < 0)
701                   fh = tab.length;
702
703               if (fh < 0)
704                   fh = tab.length;
705
706               if (fh < 0)
707                   fh = tab.length;
708
709               if (fh < 0)
710                   fh = tab.length;
711
712               if (fh < 0)
713                   fh = tab.length;
714
715               if (fh < 0)
716                   fh = tab.length;
717
718               if (fh < 0)
719                   fh = tab.length;
720
721               if (fh < 0)
722                   fh = tab.length;
723
724               if (fh < 0)
725                   fh = tab.length;
726
727               if (fh < 0)
728                   fh = tab.length;
729
730               if (fh < 0)
731                   fh = tab.length;
732
733               if (fh < 0)
734                   fh = tab.length;
735
736               if (fh < 0)
737                   fh = tab.length;
738
739               if (fh < 0)
740                   fh = tab.length;
741
742               if (fh < 0)
743                   fh = tab.length;
744
745               if (fh < 0)
746                   fh = tab.length;
747
748               if (fh < 0)
749                   fh = tab.length;
750
751               if (fh < 0)
752                   fh = tab.length;
753
754               if (fh < 0)
755                   fh = tab.length;
756
757               if (fh < 0)
758                   fh = tab.length;
759
760               if (fh < 0)
761                   fh = tab.length;
762
763               if (fh < 0)
764                   fh = tab.length;
765
766               if (fh < 0)
767                   fh = tab.length;
768
769               if (fh < 0)
770                   fh = tab.length;
771
772               if (fh < 0)
773                   fh = tab.length;
774
775               if (fh < 0)
776                   fh = tab.length;
777
778               if (fh < 0)
779                   fh = tab.length;
780
781               if (fh < 0)
782                   fh = tab.length;
783
784               if (fh < 0)
785                   fh = tab.length;
786
787               if (fh < 0)
788                   fh = tab.length;
789
790               if (fh < 0)
791                   fh = tab.length;
792
793               if (fh < 0)
794                   fh = tab.length;
795
796               if (fh < 0)
797                   fh = tab.length;
798
799               if (fh < 0)
800                   fh = tab.length;
801
802               if (fh < 0)
803                   fh = tab.length;
804
805               if (fh < 0)
806                   fh = tab.length;
807
808               if (fh < 0)
809                   fh = tab.length;
810
811               if (fh < 0)
812                   fh = tab.length;
813
814               if (fh < 0)
815                   fh = tab.length;
816
817               if (fh < 0)
818                   fh = tab.length;
819
820               if (fh < 0)
821                   fh = tab.length;
822
823               if (fh < 0)
824                   fh = tab.length;
825
826               if (fh < 0)
827                   fh = tab.length;
828
829               if (fh < 0)
830                   fh = tab.length;
831
832               if (fh < 0)
833                   fh = tab.length;
834
835               if (fh < 0)
836                   fh = tab.length;
837
838               if (fh < 0)
839                   fh = tab.length;
840
841               if (fh < 0)
842                   fh = tab.length;
843
844               if (fh < 0)
845                   fh = tab.length;
846
847               if (fh < 0)
848                   fh = tab.length;
849
850               if (fh < 0)
851                   fh = tab.length;
852
853               if (fh < 0)
854                   fh = tab.length;
855
856               if (fh < 0)
857                   fh = tab.length;
858
859               if (fh < 0)
860                   fh = tab.length;
861
862               if (fh < 0)
863                   fh = tab.length;
864
865               if (fh < 0)
866                   fh = tab.length;
867
868               if (fh < 0)
869                   fh = tab.length;
870
871               if (fh < 0)
872                   fh = tab.length;
873
874               if (fh < 0)
875                   fh = tab.length;
876
877               if (fh < 0)
878                   fh = tab.length;
879
880               if (fh < 0)
881                   fh = tab.length;
882
883               if (fh < 0)
884                   fh = tab.length;
885
886               if (fh < 0)
887                   fh = tab.length;
888
889               if (fh < 0)
890                   fh = tab.length;
891
892               if (fh < 0)
893                   fh = tab.length;
894
895               if (fh < 0)
896                   fh = tab.length;
897
898               if (fh < 0)
899                   fh = tab.length;
900
901               if (fh < 0)
902                   fh = tab.length;
903
904               if (fh < 0)
905                   fh = tab.length;
906
907               if (fh < 0)
908                   fh = tab.length;
909
910               if (fh < 0)
911                   fh = tab.length;
912
913               if (fh < 0)
914                   fh = tab.length;
915
916               if (fh < 0)
917                   fh = tab.length;
918
919               if (fh < 0)
920                   fh = tab.length;
921
922               if (fh < 0)
923                   fh = tab.length;
924
925               if (fh < 0)
926                   fh = tab.length;
927
928               if (fh < 0)
929                   fh = tab.length;
930
931               if (fh < 0)
932                   fh = tab.length;
933
934               if (fh < 0)
935                   fh = tab.length;
936
937               if (fh < 0)
938                   fh = tab.length;
939
940               if (fh < 0)
941                   fh = tab.length;
942
943               if (fh < 0)
944                   fh = tab.length;
945
946               if (fh < 0)
947                   fh = tab.length;
948
949               if (fh < 0)
950                   fh = tab.length;
951
952               if (fh < 0)
953                   fh = tab.length;
954
955               if (fh < 0)
956                   fh = tab.length;
957
958               if (fh < 0)
959                   fh = tab.length;
960
961               if (fh < 0)
962                   fh = tab.length;
963
964               if (fh < 0)
965                   fh = tab.length;
966
967               if (fh < 0)
968                   fh = tab.length;
969
970               if (fh < 0)
971                   fh = tab.length;
972
973               if (fh < 0)
974                   fh = tab.length;
975
976               if (fh < 0)
977                   fh = tab.length;
978
979               if (fh < 0)
980                   fh = tab.length;
981
982               if (fh < 0)
983                   fh = tab.length;
984
985               if (fh < 0)
986                   fh = tab.length;
987
988               if (fh < 0)
989                   fh = tab.length;
990
991               if (fh < 0)
992                   fh = tab.length;
993
994               if (fh < 0)
995                   fh = tab.length;
996
997               if (fh < 0)
998                   fh = tab.length;
999
1000              if (fh < 0)
1001                  fh = tab.length;
1002
1003              if (fh < 0)
1004                  fh = tab.length;
1005
1006              if (fh < 0)
1007                  fh = tab.length;
1008
1009              if (fh < 0)
1010                  fh = tab.length;
1011
1012              if (fh < 0)
1013                  fh = tab.length;
1014
1015              if (fh < 0)
1016                  fh = tab.length;
1017
1018              if (fh < 0)
1019                  fh = tab.length;
1020
1021              if (fh < 0)
1022                  fh = tab.length;
1023
1024              if (fh < 0)
1025                  fh = tab.length;
1026
1027              if (fh < 0)
1028                  fh = tab.length;
1029
1030              if (fh < 0)
1031                  fh = tab.length;
1032
1033              if (fh < 0)
1034                  fh = tab.length;
1035
1036              if (fh < 0)
1037                  fh = tab.length;
1038
1039              if (fh < 0)
1040                  fh = tab.length;
1041
1042              if (fh < 0)
1043                  fh = tab.length;
1044
1045              if (fh < 0)
1046                  fh = tab.length;
1047
1048              if (fh < 0)
1049                  fh = tab.length;
1050
1051              if (fh < 0)
1052                  fh = tab.length;
1053
1054              if (fh < 0)
1055                  fh = tab.length;
1056
1057              if (fh < 0)
1058                  fh = tab.length;
1059
1060              if (fh < 0)
1061                  fh = tab.length;
1062
1063              if (fh < 0)
1064                  fh = tab.length;
1065
1066              if (fh < 0)
1067                  fh = tab.length;
1068
1069              if (fh < 0)
1070                  fh = tab.length;
1071
1072              if (fh < 0)
1073                  fh = tab.length;
1074
1075              if (fh < 0)
1076                  fh = tab.length;
1077
1078              if (fh < 0)
1079                  fh = tab.length;
1080
1081              if (fh < 0)
1082                  fh = tab.length;
1083
1084              if (fh < 0)
1085                  fh = tab.length;
1086
1087              if (fh < 0)
1088                  fh = tab.length;
1089
1090              if (fh < 0)
1091                  fh = tab.length;
1092
1093              if (fh < 0)
1094                  fh = tab.length;
1095
1096              if (fh < 0)
1097                  fh = tab.length;
1098
1099              if (fh < 0)
1100                  fh = tab.length;
1101
1102              if (fh < 0)
1103                  fh = tab.length;
1104
1105              if (fh < 0)
1106                  fh = tab.length;
1107
1108              if (fh < 0)
1109                  fh = tab.length;
1110
1111              if (fh < 0)
1112                  fh = tab.length;
1113
1114              if (fh < 0)
1115                  fh = tab.length;
1116
1117              if (fh < 0)
1118                  fh = tab.length;
1119
1120              if (fh < 0)
1121                  fh = tab.length;
1122
1123              if (fh < 0)
1124                  fh = tab.length;
1125
1126              if (fh < 0)
1127                  fh = tab.length;
1128
1129              if (fh < 0)
1130                  fh = tab.length;
1131
1132              if (fh < 0)
1133                  fh = tab.length;
1134
1135              if (fh < 0)
1136                  fh = tab.length;
1137
1138              if (fh < 0)
1139                  fh = tab.length;
1140
1141              if (fh < 0)
1142                  fh = tab.length;
1143
1144              if (fh < 0)
1145                  fh = tab.length;
1146
1147              if (fh < 0)
1148                  fh = tab.length;
1149
1150              if (fh < 0)
1151                  fh = tab.length;
1152
1153              if (fh < 0)
1154                  fh = tab.length;
1155
1156              if (fh < 0)
1157                  fh = tab.length;
1158
1159              if (fh < 0)
1160                  fh = tab.length;
1161
1162              if (fh < 0)
1163                  fh = tab.length;
1164
1165              if (fh < 0)
1166                  fh = tab.length;
1167
1168              if (fh < 0)
1169                  fh = tab.length;
1170
1171              if (fh < 0)
1172                  fh = tab.length;
1173
1174              if (fh < 0)
1175                  fh = tab.length;
1176
1177              if (fh < 0)
1178                  fh = tab.length;
1179
1180              if (fh < 0)
1181                  fh = tab.length;
1182
1183              if (fh < 0)
1184                  fh = tab.length;
1185
1186              if (fh < 0)
1187                  fh = tab.length;
1188
1189              if (fh < 0)
1190                  fh = tab.length;
1191
1192              if (fh < 0)
1193                  fh = tab.length;
1194
1195              if (fh < 0)
1196                  fh = tab.length;
1197
1198              if (fh < 0)
1199                  fh = tab.length;
1200
1201              if (fh < 0)
1202                  fh = tab.length;
1203
1204              if (fh < 0)
1205                  fh = tab.length;
1206
1207              if (fh < 0)
1208                  fh = tab.length;
1209
1210              if (fh < 0)
1211                  fh = tab.length;
1212
1213              if (fh < 0)
1214                  fh = tab.length;
1215
1216              if (fh < 0)
1217                  fh = tab.length;
1218
1219              if (fh < 0)
1220                  fh = tab.length;
1221
1222              if (fh < 0)
1223                  fh = tab.length;
1224
1225              if (fh < 0)
1226                  fh = tab.length;
1227
1228              if (fh < 0)
1229                  fh = tab.length;
1230
1231              if (fh < 0)
1232                  fh = tab.length;
1233
1234              if (fh < 0)
1235                  fh = tab.length;
1236
1237              if (fh < 0)
1238                  fh = tab.length;
1239
1240              if (fh < 0)
1241                  fh = tab.length;
1242
1243              if (fh < 0)
1244                  fh = tab.length;
1245
1246              if (fh < 0)
1247                  fh = tab.length;
1248
1249              if (fh < 0)
1250                  fh = tab.length;
1251
1252              if (fh < 0)
1253                  fh = tab.length;
1254
1255              if (fh < 0)
1256                  fh = tab.length;
1257
1258              if (fh < 0)
1259                  fh = tab.length;
1260
1261              if (fh < 0)
1262                  fh = tab.length;
1263
1264              if (fh < 0)
1265                  fh = tab.length;
1266
1267              if (fh < 0)
1268                  fh = tab.length;
1269
1270              if (fh < 0)
1271                  fh = tab.length;
1272
1273              if (fh < 0)
1274                  fh = tab.length;
1275
1276              if (fh < 0)
1277                  fh = tab.length;
1278
1279              if (fh < 0)
1280                  fh = tab.length;
1281
1282              if (fh < 0)
1283                  fh = tab.length;
1284
1285              if (fh < 0)
1286                  fh = tab.length;
1287
1288              if (fh < 0)
1289                  fh = tab.length;
1290
1291              if (fh < 0)
1292                  fh = tab.length;
1293
1294              if (fh < 0)
1295                  fh = tab.length;
1296
1297              if (fh < 0)
1298                  fh = tab.length;
1299
1300              if (fh < 0)
1301                  fh = tab.length;
1302
1303              if (fh < 0)
1304                  fh = tab.length;
1305
1306              if (fh < 0)
1307                  fh = tab.length;
1308
1309              if (fh < 0)
1310                  fh = tab.length;
1311
1312              if (fh < 0)
1313                  fh = tab.length;
1314
1315              if (fh < 0)
1316                  fh = tab.length;
1317
1318              if (fh < 0)
1319                  fh = tab.length;
1320
1321              if (fh < 0)
1322                  fh = tab.length;
1323
1324              if (fh < 0)
1325                  fh = tab.length;
1326
1327              if (fh < 0)
1328                  fh = tab.length;
1329
1330              if (fh < 0)
1331                  fh = tab.length;
1332
1333              if (fh < 0)
1334                  fh = tab.length;
1335
1336              if (fh < 0)
1337                  fh = tab.length;
1338
1339              if (fh < 0)
1340                  fh = tab.length;
1341
1342              if (fh < 0)
1343                  fh = tab.length;
1344
1345              if (fh < 0)
1346                  fh = tab.length;
1347
1348              if (fh < 0)
1349                  fh = tab.length;
1350
1351              if (fh < 0)
1352                  fh = tab.length;
1353
1354              if (fh < 0)
1355                  fh = tab.length;
1356
1357              if (fh < 0)
1358                  fh = tab.length;
1359
1360              if (fh < 0)
1361                  fh = tab.length;
1362
1363              if (fh < 0)
1364                  fh = tab.length;
1365
1366              if (fh < 0)
1367                  fh = tab.length;
1368
1369              if (fh < 0)
1370                  fh = tab.length;
1371
1372              if (fh < 0)
1373                  fh = tab.length;
1374
1375              if (fh < 0)
1376                  fh = tab.length;
1377
1378              if (fh < 0)
1379                  fh = tab.length;
1380
1381              if (fh < 0)
1382                  fh = tab.length;
1383
1384              if (fh < 0)
1385                  fh = tab.length;
1386
138
```

```

17     if (tab == null || (n = tab.length) == 0)
18         tab = initTable();           // 第一次 put 时 table 未初始化,
初始化之
19     else if ((f = tabAt(tab, i = (n - 1) & hash)) == null) {
20         // 3.1 如果当前位置为 null, 尝试使用 CAS 添加
21         if (casTabAt(tab, i, null,
22                         new Node<K,V>(hash, key, value, null))) {
23             break;                  // no lock when adding to
empty bin
24         }
25         // 3.2 如果 f.hash == MOVED 成立, 说明其他线程正在扩容, 则一起参与扩容
26         else if ((fh = f.hash) == MOVED)
27             tab = helpTransfer(tab, f);
28         // 3.3 如果都不满足, synchronized 锁住 f 节点, 判断是链表还是红黑树,
后遍历插入
29     else {
30         V oldVal = null;
31         synchronized (f) {
32             if (tabAt(tab, i) == f) {
33                 if (fh >= 0) {
34                     binCount = 1;
35                     for (Node<K,V> e = f;; ++binCount) {
36                         K ek;
37                         if (e.hash == hash &&
38                             ((ek = e.key) == key ||
39                             (ek != null && key.equals(ek)))) {
40                             oldVal = e.val;
41                             if (!onlyIfAbsent)
42                                 e.val = value;
43                             break;
44                         }
45                         Node<K,V> pred = e;
46                         if ((e = e.next) == null) {
47                             pred.next = new Node<K,V>(hash, key,
48                                                 value,
49                                                 null);
50                             break;
51                         }
52                     }
53                 else if (f instanceof TreeBin) {
54                     Node<K,V> p;
55                     binCount = 2;
56                     if ((p = ((TreeBin<K,V>) f).putTreeVal(hash,
57                                         key,
58                                         value)) !=
59                                         null) {
59                         oldVal = p.val;
60                         if (!onlyIfAbsent)
61                             e.val = value;
62                         break;
63                     }
64                 }
65             }
66         }
67     }
68 }

```

```

60             p.val = value;
61         }
62     }
63 }
64
65 if (binCount != 0) {
66     if (binCount >= TREEIFY_THRESHOLD)
67         // 4. 当链表长度达到树化阈值后,
68         // 则扩容或者转换为红黑树(取决于容量是否达到最小树化容量)
69         treeifyBin(tab, i);
70     if (oldVal != null)
71         return oldVal;
72     break;
73 }
74 }
75 }
76 addCount(1L, binCount);
77 return null;
78 }

```

将锁的级别控制在了更细粒度的哈希桶数组元素级别，只需要对当前链表头节点或者当前红黑树上锁，其他线程就不会影响当前整个链表或者红黑树。

此外 JDK1.8 ConcurrentHashMap 使用 synchronized 来代替 Segment 段的 ReentrantLock 重入锁有两点原因：

1. synchronized 锁动态升级：JDK1.6 版本后，synchronized 会根据竞争条件有需要地从无锁 -> 偏向锁 -> 轻量级锁 -> 重量级锁一步步转换，而不是一开始就是重量级锁。
2. 减小内存开销：JDK1.7 中使用继承于 ReentrantLock 的 Segment 来实现同步，而 Segment 由涵盖整个链表，所以说一个链表的所有节点都支持同步，但是，实际上只有链表的头节点（或者红黑树的根节点）需要同步，这带来了大量不必要的内存浪费。所以，JDK1.8 开始，使用 synchronized 只对链表的头节点或者红黑树的根节点上锁即可实现同步。

## 迭代器一致性

迭代器的一致性通常指的是在多线程或多进程环境中使用迭代器进行遍历时，保证遍历的结果是正确且可预测的。

简言之，弱一致性迭代器允许在遍历过程中修改原数据，强一致性则不允许在遍历过程中修改原数据。

支持迭代器的 Java 集合通常会有 modCount (modification count, 修改计数) 字段，用于维护迭代器的一致性。这个字段用于记录对集合结构修改操作的次数，以便在迭代过程中能够检测到并发修改，从而避免可能的数据不一致问题。

在使用迭代器遍历集合时，迭代器会记录创建时的 modCount 值，作为期望的 modCount 值 (expectedModCount)，然后在每次迭代时都会检查当前的 modCount 值是否与创建时的相同，如果发现 modCount 不等于 expectedModCount，则说明在遍历过程中集合结构发生了改变，可能导致不一致，此时会抛出 ConcurrentModificationException 异常，以警示在迭代过程中集合结构发生了改变。

## ConcurrentHashMap 和 Hashtable 都是线程安全的，两者有什么区别

ConcurrentHashMap 和 Hashtable 都是用于多线程环境下进行并发访问的数据结构，都提供了一定程度的线程安全性。然而，两者在实现和性能方面存在一些重要区别。

### 1. 同步粒度

- Hashtable 使用同步方法保护，也就是锁是当前对象，这就意味着每次只能由一个线程访问当前 Hashtable 对象。锁粒度较大，可能导致高并发环境下的性能瓶颈，因为多个线程可能被迫等待 Hashtable 对象锁的释放。

```
1 public synchronized boolean containsKey(Object key) {  
2     Entry<?, ?> tab[] = table;  
3     int hash = key.hashCode();  
4     int index = (hash & 0x7FFFFFFF) % tab.length;  
5     for (Entry<?, ?> e = tab[index] ; e != null ; e = e.next)  
6     {  
7         if ((e.hash == hash) && e.key.equals(key)) {  
8             return true;  
9         }  
10    }  
11    return false;  
12 }
```

- ConcurrentHashMap (JDK1.7) 将整个哈希表分割成一系列段，每个段都有独立的锁。这意味着多个线程可以各自访问不同的段，提高了并发性能。

### 2. 性能

- Hashtable 的性能在高并发的环境下相对较差，因为锁的粒度大，可能导致多个线程在等待锁上浪费时间
- ConcurrentHashMap 采用了分段锁的思想，允许多个线程分别访问不同的段，从而减少了竞争和等待时间。

### 3. Null Key 和 Null Value

- Hashtable 不允许存储 Null Key 和 Null Value，如果尝试插入 Null Key 或者 Null Value，会抛出 NullPointerException 异常
- JDK8以前，ConcurrentHashMap 不支持 Null Key 和 Null Value；JDK8及其以后的版本中，ConcurrentHashMap 引入了一种特殊的标记值，用以处理 Null Key 和 Null Value。

### 4. 迭代器一致性

- Hashtable 没有提供弱一致性的迭代行，迭代器是强一致性的。所以创建迭代器时，会将当前 Hashtable 锁住以创建一个稳定的快照。
- ConcurrentHashMap 提供弱一致性的迭代器，允许在迭代过程中看到部分更新。

总的来说，ConcurrentHashMap 是在高并发环境下更为推荐的选择，因为它在性能和并发性方面具有优势。



## 基本介绍

Java 泛型是 Java 编程语言中引入的一种特性，它允许在定义类、接口和方法时使用类型参数，以实现代码的重用和类型安全。泛型提供了编译时类型检查，并使得代码更加灵活和通用。

使用泛型时可以将类型作为参数传递，而不是在代码中直接指定具体的类型。这样可以增加代码的灵活性，使得一个类、接口或者方法可以用于多种类型的数据，而不规定于特定的数据类型。

泛型的优点包括：

### 1. 类型安全

编译器会在编译时进行类型检查，确保代码不会发生类型转换异常。

### 2. 代码重用

可以编写通用的代码，可以用于处理多种不同类型的数据

### 3. 简化代码

避免了手动进行类型转化，使代码更加简洁

泛型的语法使用尖括号 (<>) 来定义类型参数，通常使用单个大写字母表示，比如 <T>，<K>，<V> 等。在类、接口、方法的定义中，使用这些类型参数来代表具体的类型。

定义一个简单的泛型类

```
1 public class GenericCalculator<T extends Number> {  
2  
3     public double add(T x, T y) {  
4         return x.doubleValue() + y.doubleValue();  
5     }  
6  
7     public double sub(T x, T y) {  
8         return x.doubleValue() - y.doubleValue();  
9     }  
10  
11    public double mul(T x, T y) {  
12        return x.doubleValue() * y.doubleValue();  
13    }  
14}
```

```
14
15     public double div(T x, T y) {
16         return x.doubleValue() / y.doubleValue();
17     }
18
19 }
20
```

```
1 import java.util.Scanner;
2
3 public class GenericCalculatorTest {
4
5     public static void main(String[] args) {
6         GenericCalculator<Number> calculator = new GenericCalculator<>
7             ();
8         Scanner scanner = new Scanner(System.in);
9         int x = scanner.nextInt();
10        byte y = scanner.nextByte();
11        System.out.println(calculator.add(x, y));
12    }
13 }
```

## 要点

使用泛型时，需要注意以下要点：

### 1. 类型参数

泛型使用类型参数 (Type Parameter) 来表示参数化类型。类型参数通常用大写字母表示，如T、E、K等。类型参数在定义类、接口或方法时使用，并在使用时由具体的类型替代。

### 2. 类型安全

泛型在编译时提供类型检查机制，确保代码中的类型转换是安全的。这样可以避免在运行时出现类型转换异常，提高代码的稳定性和可靠性。

### 3. 泛型类

使用泛型类可以创建通用的数据结构和容器，例如列表、栈、队列等。泛型类的定义在类名后面使用尖括号括起来的类型参数列表。

### 4. 泛型接口

泛型接口与泛型类类似，允许在接口中使用类型参数。这使得实现泛型接口的类也能够具有通用的特性。

### 5. 泛型方法

可以在普通类中定义泛型方法，这样可以在方法级别上使用类型参数。泛型方法可以是静态的或非静态的，并且可以在调用时指定具体的类型。

## 6. 通配符

通配符是指用 ? 代替具体的类型参数。通配符通常用于在泛型类或方法中表示未知类型或者限制类型的范围。

```
1 package wild;
2
3 import java.util.ArrayList;
4 import java.util.List;
5 import java.util.Random;
6
7 /**
8  * 上界限定 (extends):
9  * 使用上界限定可以确保泛型类型参数必须是指定类或其子类。这允许你在泛型类或方法中操作这个范围内的对象。在泛型参数后使用 extends 关键字，然后跟着上界类的名称。
10 *
11 * T extends Number
12 * T 必须是 Number 或其子类，所以你可以传递 Integer、Double、Float。
13 *
14 * 下界限定 (super):
15 * 使用下界限定可以确保泛型类型参数必须是指定类或其父类。这允许你在泛型类或方法中操作这个范围内的对象。在泛型参数后使用 super 关键字，然后跟着下界类的名称。
16 *
17 * ? super T
18 * ? super T 表示可以是 T 或其父类，所以你可以传递 List<Object>、
List<Number>、List<Integer>。
19 *
20 *
21 */
22 public class GenericWildcardTest {
23
24     public static void main(String[] args) {
25         // 无限制通配符
26         List<?> wildcardList = new ArrayList<>();
27         wildcardList = new ArrayList<Integer>();
28         wildcardList = new ArrayList<Double>();
29         // 无法添加元素到无限制通配符集合
30         //     wildcardList.add(10);      // Error
31
32         // 上界通配符
33         List<? extends Number> upperBoundList = new ArrayList<>();
34         upperBoundList = new ArrayList<Integer>();
35         upperBoundList = new ArrayList<Double>();
36         // 无法添加元素到上界通配符集合
37         //     upperBoundList.add(10); // Error
38
39         // 下界通配符
```

```

40         List<? super Integer> lowerBoundList = new ArrayList<>();
41         lowerBoundList = new ArrayList<Number>();
42         lowerBoundList = new ArrayList<Object>();
43         lowerBoundList.add(10); //可以添加到下界通配符集合
44
45         processWildcardList(new ArrayList<Integer>());
46         processUpperBoundList(new ArrayList<Double>());
47         processLowerBoundList(new ArrayList<Object>());
48
49     }
50
51     // 无限制通配符作为方法参数
52     public static void processWildcardList(List<?> list) {
53         // 可以访问元素，但不能添加元素到通配符集合
54         list.forEach(System.out::println);
55     }
56
57     // 上界通配符作为方法参数
58     // '?' 代表的类型作为 Number 的 子类 或者 实现类
59     public static void processUpperBoundList(List<? extends
Number> list) {
60         // 可以访问元素，但不能添加元素到通配符集合
61         list.forEach(System.out::println);
62     }
63
64     // 下界通配符作为方法参数
65     // '?' 代表的类型作为 Number 的 父类
66     public static void processLowerBoundList(List<? super
Integer> list) {
67         Random random = new Random(System.currentTimeMillis());
68         for (int i = 0 ; i < 5 ; i ++ ) {
69             list.add(random.nextInt());
70         }
71     }
72
73
74 }
75

```

## 7. 限定类型

泛型可以使用限定类型 (Bounded Type) 来约束类型参数。通过**extends**关键字，可以限制类型参数必须是指定类或其子类。

## 8. 类型擦除

Java中的泛型是通过类型擦除实现的。在编译时，泛型信息被擦除，并替换为**Object**类型。这意味着在运行时，泛型的类型参数信息是不可用的。

```

1 package erase;
2

```

```
3 import java.util.ArrayList;
4
5 public class TypeErase {
6
7     public static void main(String[] args) {
8         ArrayList<String> stringList = new ArrayList<>();
9         stringList.add("abc");
10
11         ArrayList<Integer> integerList = new ArrayList<>();
12         integerList.add(123);
13
14         // 运行时，泛型的类型参数信息是不可用的。
15         // 结果为 true
16         System.out.println(stringList.getClass() ==
17             integerList.getClass());
18     }
19
20 }
21
```

## 9. 自动装箱与拆箱

泛型在自动装箱和拆箱的过程中发挥重要作用。自动装箱允许在基本类型和包装类型之间自动进行转换。

# 类型擦除

## 类型擦除机制的体现

### 原始类型相等

```
1 package erase.demo;
2
3 import java.util.ArrayList;
4
5 /**
6  * 原始类型相等
7  */
8 public class TypeEraseDemoByEqual {
9
10
11     /**
12      * 在这个例子中，我们定义了两个ArrayList数组，不过一个是ArrayList<String>
13      * 泛型类型的，只能存储字符串；
14      * 一个是ArrayList<Integer>泛型类型的，只能存储整数，最后，我们通过list1对
15      * 象和list2对象的getClass()方法获取他们的类的信息，最后发现结果为true。
16
17 }
```

```

14     * 说明泛型类型String和Integer都被擦除掉了，只剩下原始类型。
15     * @param args
16     */
17     public static void main(String[] args) {
18         ArrayList<String> stringList = new ArrayList<>();
19         stringList.add("abc");
20
21         ArrayList<Integer> integerList = new ArrayList<>();
22         integerList.add(123);
23
24         // 运行时，泛型的类型参数信息是不可用的。
25         // 结果为 true
26         System.out.println(stringList.getClass() ==
27             integerList.getClass());
28     }
29
30 }
31

```

## 通过反射添加其他元素

```

1 package erase.demo;
2
3 import java.lang.reflect.InvocationTargetException;
4 import java.lang.reflect.Method;
5 import java.util.ArrayList;
6 import java.util.List;
7
8 /**
9  * 通过反射添加其它类型元素
10 */
11 public class TypeEraseDemoByReflection {
12
13     /**
14      *
15      * 在程序中定义了一个ArrayList泛型类型实例化为Integer对象，如果直接调用
16      * add()方法，
17      * 那么只能存储整数数据，不过当我们利用反射调用add()方法的时候，却可以存储字符
18      * 串，
19      * 这说明了Integer泛型实例在编译之后被擦除掉了，只保留了原始类型。
20      *
21      * @param args
22      * @throws NoSuchMethodException
23      * @throws InvocationTargetException
24      * @throws IllegalAccessException
25      */
26

```

```
24     public static void main(String[] args) throws
25         NoSuchMethodException, InvocationTargetException,
26         IllegalAccessException {
27
28     List<Integer> integerList = new ArrayList<>();
29     integerList.add(1024);
30
31     Method addObjectMethod =
32         integerList.getClass().getMethod("add", Object.class);
33
34     addObjectMethod.invoke(integerList, "Hello");
35     addObjectMethod.invoke(integerList, new Object() {
36         @Override
37         public String toString() {
38             return "SAMPLE";
39         }
40     });
41
42
43 }
44
45 }
46
```

## 类型擦除后保留的原始类型

原始类型就是擦去了泛型信息，最后在字节码中的类型变量的真正类型，无论何时定义一个泛型，相应的原始类型对会被自动提供，类型变量擦除，并使用其限定类型（无限定的变量用 Object）替换。

ObjectPair

```
1 package erase.origin;
2
3 public class ObjectPair<T> {
4     private T value;
5
6     public ObjectPair(T value) {
7         this.value = value;
8     }
9
10    public T getValue() {
11        return value;
12    }
13}
```

ObjectPair 的原始类型为

```
1 package erase.origin;
2
3 public class ObjectPair {
4     private Object value;
5
6     public ObjectPair(Object value) {
7         this.value = value;
8     }
9
10    public Object getValue() {
11        return value;
12    }
13}
```

因为在Pair<T>中，T是一个无限定的类型变量，所以用Object替换，其结果就是一个普通的类，如同泛型加入Java语言之前的已经实现的样子。在程序中可以包含不同类型的Pair，如Pair<String>或Pair<Integer>，但是擦除类型后他们的就成为原始的Pair类型了，原始类型都是Object。

ComparablePair

```
1 package erase.origin;
2
3 public class ComparablePair<T extends Comparable> {
4     private T value;
5
6     public ComparablePair(T value) {
7         this.value = value;
8     }
9
10    public T getValue() {
11        return value;
12    }
13 }
```

原始类型就是Comparable

ComparablePair 的原始类型

```
1 package erase.origin;
2
3 public class ComparablePair {
4     private Comparable value;
5
6     public ComparablePair(Comparable value) {
7         this.value = value;
8     }
9
10    public Comparable getValue() {
11        return value;
12    }
13 }
```

## 调用时指定泛型和不指定泛型

- 在不指定泛型的情况下，泛型变量的类型为该方法中的几种类型的同一父级的最小级，直  
到 Object

```
1 public class ComparablePairTest {
2
3     private static class NormalClass {
4     }
5
6     private static class ComparableImplementationClass implements
7         Comparable {
8
9         @Override
10        public int compareTo(Object o) {
11            return 0;
12        }
13 }
```

```

12     }
13
14     private static class ComparableImplementationClassSon extends
ComparableImplementationClass {
15     }
16
17     public static void main(String[] args) {
18         // ComparablePair pair = new ComparablePair(new
NormalClass()); // 泛型变量的类型不为该方法中的几种类型的同一父级的最上级,
ERROR
19         ComparablePair<ComparableImplementationClass> pair =
20             new ComparablePair<>(new
ComparableImplementationClass()); // 泛型变量的类型为该方法中的几种类型的
同一父级的最上级
21         ComparablePair<ComparableImplementationClassSon> sonPair =
22             new ComparablePair<>(new
ComparableImplementationClassSon()); // 泛型变量的类型为该方法中的几种类型的
同一父级的最上级
23     }
24
25 }

```

- 在指定泛型的情况下，该方法的几种类型必须是该泛型的实例的类型或者其子类

```

1 package invoke;
2
3 import java.io.Serializable;
4
5 public class InvokeGenericExplicitOrImplicit {
6
7     public static void main(String[] args) {
8         /** 不指定泛型 */
9         Integer integer = add(1, 2);           // 这两个参数都是
Integer, 所以 T 为 Integer
10        Number f = add(1, 1.2);             // 一个是 Integer, 一个
是 Float, 取同一父类的最上级 Number
11        Serializable s = add(1, "add");     // 一个是 Integer, 一个
是 String, 取同一父类的最上级 Serializable
12
13        /** 指定泛型 */
14        Integer integer1 = InvokeGenericExplicitOrImplicit.
<Integer>add(1, 2); // 指定了 Integer, 所以只能为 Integer 或者其子类
15        // Integer f = InvokeGenericExplicitOrImplicit.
<Integer>add(1, 2.2);           // 编译错误, 指定了 Integer, 不能为
Float
16        Number number = InvokeGenericExplicitOrImplicit.
<Number>add(1, 2.2); // 指定为 Number, 所以可以为 Float 或者
Integer

```

```
17    }
18
19    public static <T> T add(T x, T y) {
20        return y;
21    }
22
23 }
24
```

## 类型擦除引起的问题以及解决方案

Java 的泛型是伪泛型，通过类型擦除来实现，以避免了类型膨胀问题，但是也引发了许多问题，所以JDK对这些问题做出了种种限制，避免错误发生。

### 泛型检查后再编译以及以及编译的对象和引用传递问题

Java 编译器通过先检查代码中泛型的类型，然后再进行类型擦除，再进行编译。也就是类型检查是在编译时完成的。

所以当往 `ArrayList<Integer>` 中添加 "abc" 时，Java 编译器检查出泛型类型不相符，报错。但是可以通过反射等方式逃脱Java编译器的检查。

需要注意，类型检查是由引用完成而非引用的对象，见下例

```
1 package erase.check;
2
3 import java.util.ArrayList;
4
5 /**
6  * 因为类型检查就是编译时完成的,
7  * new ArrayList() 只是在内存中开辟了一个存储空间, 可以存储任何类型对象, 而真正设
8  * 计类型检查的是它的引用
9  *
10 * 类型检查就是针对引用的, 谁是一个引用, 用这个引用调用泛型方法, 就会对这个引用调用
11 * 的方法进行类型检测, 而无关它真正引用的对象
12 *
13 */
14 public class CompileTypeCheck {
15
16     public static void main(String[] args) {
17
18         //list1引用能完成泛型类型的检查。
19         ArrayList<String> list1 = new ArrayList();
20         list1.add("1");
21         // list1.add(1);      // ERROR
22         // 引用指定类型, 检查类型, 返回 String
23         String stringElement = list1.get(0);
24     }
25 }
```

```
23 // 而引用list2没有使用泛型，所以不行。
24 ArrayList list2 = new ArrayList<String>();
25 list2.add("abc");
26 list2.add(1);
27 // 引用未指定类型，不检查类型，返回 Object
28 Object o = list2.get(1);
29
30
31 }
32
33 }
34
```

还需要注意，泛型中类型不考虑继承关系

```
1 ArrayList<String> list1 = new ArrayList<Object>(); // 编译错误
2 ArrayList<Object> list2 = new ArrayList<String>(); // 编译错误
```

## 自动类型转换

因为类型擦除机制，所以所有泛型的类型变量最后都会被替换为原始类型。

既然都被替换为原始类型，那么我们在获取的时候为什么不用强制转换呢？

在 `ArrayList#get(int)` 中寻找答案

```
1 /**
2  * Returns the element at the specified position in this list.
3  *
4  * @param index index of the element to return
5  * @return the element at the specified position in this list
6  * @throws IndexOutOfBoundsException {@inheritDoc}
7  */
8 public E get(int index) {
9     Objects.checkIndex(index, size);
10    return elementData(index);
11 }
12
13 E elementData(int index) {
14     return (E) elementData[index];
15 }
```

可以看到，在 `return` 之前，假设泛型类型变量为 `String`，虽然泛型信息会被擦除掉，但是会将 `(E) elementData[index]` 编译为 `(String) elementData[index]`，所以我们不用自行强转。

## 类型擦除与多态的冲突和解决方法

现有泛型类 Box<T>

```
1 public class Box<T> {  
2  
3     private T value;  
4  
5     public T getValue() {  
6         return value;  
7     }  
8  
9     public void setValue(T value) {  
10        this.value = value;  
11    }  
12}
```

有子类 StringBox 继承于 Box，并重写 setValue

```
1 package erase.bridge;  
2  
3 public class StringBox extends Box<String> {  
4  
5     @Override  
6     public void setValue(String value) {  
7         super.setValue(value);  
8     }  
9 }  
10}
```

然而，类型擦除后，父类

```
1 public class Box {  
2  
3     private Object value;  
4  
5     public Object getValue() {  
6         return value;  
7     }  
8  
9     public void setValue(Object value) {  
10        this.value = value;  
11    }  
12}
```

而回看子类的 setValue(String)

```
1 | @Override
2 | public void setValue(String value) {
3 |     super.setValue(value);
4 | }
```

两者的参数列表不同，直觉上应该是重载而不是重写。这样类型擦除和多态就有了冲突。

于是 JVM 使用桥方法来解决这个冲突。

可以使用 `javap -c className` 的方式反编译 `StringBox`

```
1 | Compiled from "StringBox.java"
2 | public class erase.bridge.StringBox extends
|   erase.bridge.Box<java.lang.String> {
3 |     public erase.bridge.StringBox() {
4 |       Code:
5 |         0: aload_0
6 |         1: invokespecial #1           // Method
|           erase/bridge/Box."<init>":()V
7 |         4: return
8 |
9 |     public void setValue(java.lang.String); // 重写的 setValue 方法
10 |       Code:
11 |         0: aload_0
12 |         1: aload_1
13 |         2: invokespecial #2           // Method
|           erase/bridge/Box.setValue:(Ljava/lang/Object;)V
14 |         5: return
15 |
16 |     public void setValue(java.lang.Object); // 编译器生成的桥方法
17 |       Code:
18 |         0: aload_0
19 |         1: aload_1
20 |         2: checkcast      #3           // class java/lang/String
21 |         5: invokevirtual #4           // Method setValue:
|           (Ljava/lang/String;)V 调用我们重写的 setValue 方法
22 |         8: return
23 |   }
24 | }
```

其中 `public void setValue(java.lang.Object)` 时编译器生成的桥方法，而桥方法的内部实现，就只是去调用我们自己重写的方法。通过反射也可以看到桥方法。

```
1 | public class BridgeDemo {
2 |     public static void main(String[] args) throws
|       NoSuchMethodException, InvocationTargetException,
|       IllegalAccessException {
3 |         StringBox stringBox = new StringBox();
4 |         Class<StringBox> clazz = StringBox.class;
```

```
5     Method bridgeSetValueMethod = clazz.getMethod("setValue",
6         Object.class);
7     System.out.println(bridgeSetValueMethod.isBridge());
8     // bridgeSetValueMethod.invoke(stringBox, 123); // ClassCastException
9
10    Method overrideSetValueMethod = clazz.getMethod("setValue",
11        String.class);
12    System.out.println(overrideSetValueMethod.isBridge());
13    overrideSetValueMethod.invoke(stringBox, "123");
14 }
```

## 泛型类型不能是基本数据类型

不能用类型参数替换基本类型。就比如，没有ArrayList<double>，只有ArrayList<Double>。因为当类型擦除后，ArrayList的原始类型变为Object，但是Object类型不能存储double值，只能引用Double的值。

## 泛型信息对 instanceof 无效

```
1 | ArrayList<String> arrayList = new ArrayList<String>();
```

因为类型擦除之后，ArrayList<String>只剩下原始类型，泛型信息String不存在了。

那么，编译时进行类型查询的时候使用下面的方法是错误的

```
1 | if (arrayList instanceof ArrayList<String>)
```

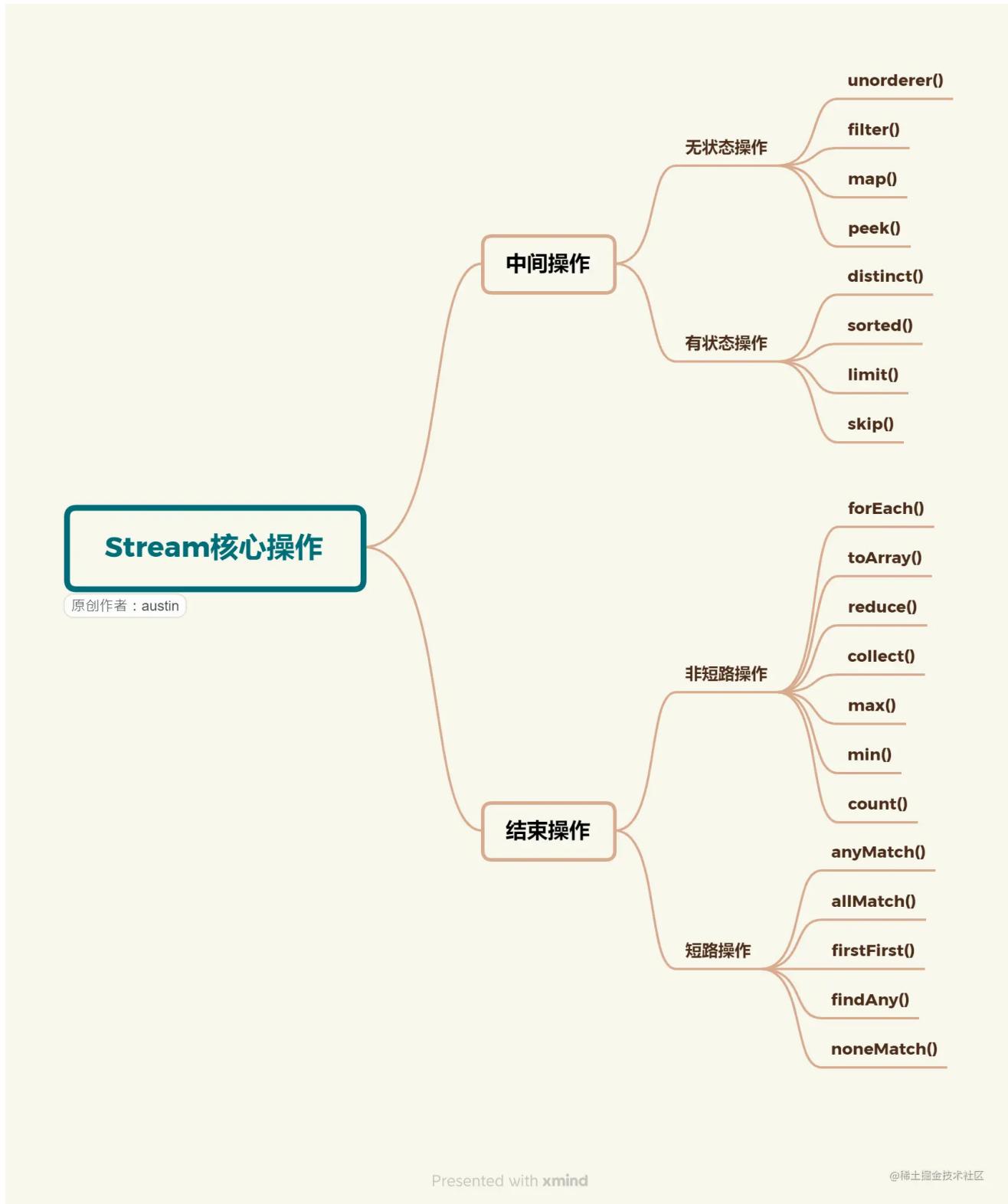
## 泛型不可以作为静态变量，也不可以作为静态方法的返回值

因为泛型类中的泛型参数的实例化是在定义对象的时候指定的，而静态变量和静态方法不需要使用对象来调用。对象都没有创建，如何确定这个泛型参数是何种类型，所以当然是错误的。

```
1 | public class Test2<T> {
2 |     public static T one;      //编译错误
3 |     public static T show(T one) { //编译错误
4 |         return null;
5 |     }
6 | }
```

但是允许静态的泛型方法

```
1 public class Test2<T> {  
2  
3     public static <T>T show(T one) { //这是正确的  
4         return null;  
5     }  
6 }
```



# 介绍

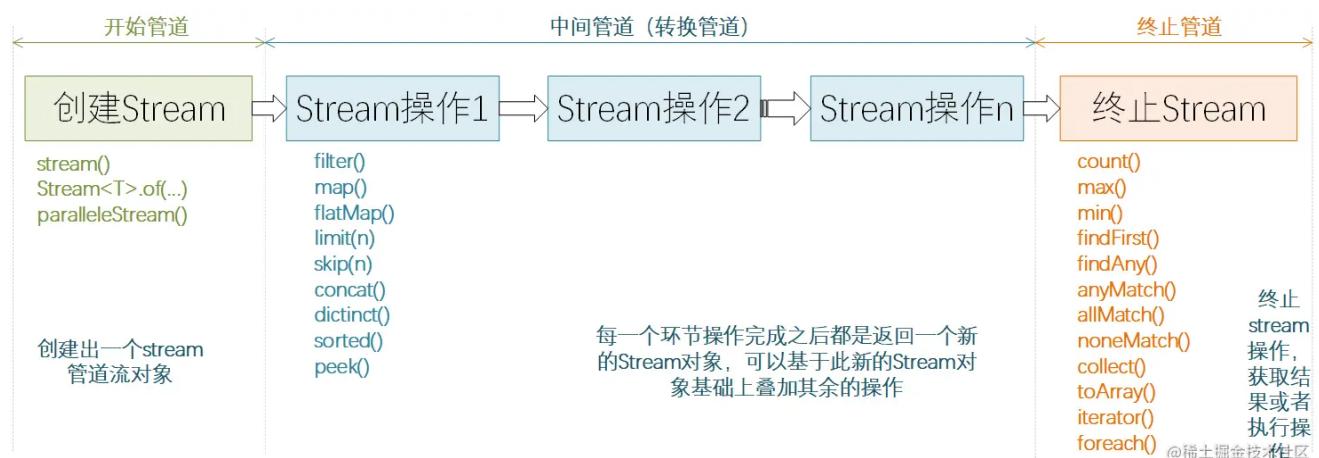
Java 中的流 (Streams) 是一种用于处理数据序列的抽象概念。它们提供了一种高效、灵活和统一的方式来处理输入和输出操作，包括文件、网络连接、数组等。

要想操作流，首先需要有一个数据源，可以是数组或者集合。每次操作都会返回一个新的流对象，方便进行链式操作，但原有的流对象会保持不变。

流的操作可以分为两种类型：

- 1) 中间操作，可以有多个，每次返回一个新的流，可进行链式操作。
- 2) 终端操作，只能有一个，每次执行完，这个流也就用光光了，无法执行下一个操作，因此只能放在最后。

可以将流的使用步骤分为三步



字符串列表去重计数

```
1 package example;
2
3 import java.util.ArrayList;
4 import java.util.List;
5 import java.util.stream.Stream;
6
7 public class StringDistinctExample {
8
9     public static void main(String[] args) {
10         List<String> list = new ArrayList<>();
11
12         list.add("武汉加油");
13         list.add("中国加油");
14         list.add("世界加油");
15         list.add("世界加油");
16
17         long distinctCount = -1;
18         try (Stream<String> stream = list.stream()) {
19             // 中间操作 去重
20             Stream<String> distinctStream = stream.distinct();
```

```
21         // 终端操作 终端方法
22         distinctCount = distinctStream.count();
23     }
24
25     System.out.println(distinctCount);
26 }
27
28 }
29 }
```

## 名字過濾器

```
1 package example;
2
3 import java.util.ArrayList;
4 import java.util.List;
5
6 public class NameFilterExample {
7
8     private static final String[] NAME_DATA = {
9         "张伟", "王芳", "李明", "赵敏", "钱建国", "孙雅婷", "周宇", "吴
10    雪", "郑阳", "王洋",
11    "陈琳", "杨健", "刘慧", "黄强", "胡莉", "林宇", "程芳", "朱强",
12    "徐静", "何婷",
13    "孔浩", "曹磊", "曾倩", "吕波", "丁雪", "江波", "何文", "董慧",
14    "洪霞", "叶峰",
15    "武鹏", "尹萍", "黄华", "罗刚", "梁洁", "郑涛", "谢静", "韩明",
16    "何秀英", "李建",
17    "张霞", "吴斌", "刘雪", "王强", "李敏", "赵辉", "孙倩", "周鹏",
18    "吴洁", "郑阳",
19    "林峰", "杨柳", "梁伟", "朱丽", "刘鹏", "陈婷", "张波", "王倩",
20    "李宇", "赵萌",
21    "孙涛", "周霞", "吴健", "郑文", "王婷", "冯刚", "陈洁", "杨涛",
22    "徐丽", "曹伟",
23    "曾琳", "吕明", "丁倩", "江强", "何欣", "董勇", "洪萍", "叶磊",
24    "武秀英", "尹峰",
25    "黄雪", "罗杰", "梁萍", "郑刚", "谢倩", "韩涛", "何芳", "李伟",
26    "张丽", "吴磊",
27    };
28
29     // 将 NAME_DATA 中, 以周开头的元素放到新数组中
30     public static void main(String[] args) {
31
32         ArrayList<String> nameList = new ArrayList<>
33         (List.of(NAME_DATA));
34
35         ArrayList<String> list = new ArrayList<>();
36
37         // 常规方法
38
39         for (String name : NAME_DATA) {
40             if (name.startsWith("周")) {
41                 list.add(name);
42             }
43         }
44
45         System.out.println(list);
46     }
47 }
```

```

28     for (String name : NAME_DATA) {
29         if (name.startsWith("周")) {
30             list.add(name);
31         }
32     }
33     System.out.println(list);
34
35     list.clear();
36
37     // 流
38     nameList.stream().filter(name ->
39         name.startsWith("周")).forEach(list::add);
40     System.out.println(list);
41
42
43
44 }
45

```

中间操作不会立即执行，只有等到终端操作的时候，流才开始真正地遍历，用于映射、过滤等。通俗点说，就是一次遍历执行多个操作，性能就大大提高了。

## 创建流

API	功能说明
stream()	创建出一个新的stream串行流对象
parallelStream()	创建出一个可并行执行的stream流对象
Stream.of()	通过给定的一系列元素创建一个新的Stream串行流对象

如果是数组，可以使用 `Arrays.stream()` 或者 `Stream.of()` 创建流

```

1 package create;
2
3 import java.util.Arrays;
4 import java.util.stream.IntStream;
5 import java.util.stream.Stream;
6
7 public class ArrayStreamCreate {
8
9     private final static int STREAM_ARRAY_SIZE = 1000;

```

```

10     private final static int[] STREAM_ARRAY = new
11     int[STREAM_ARRAY_SIZE];
12
13     static {
14         for (int i = 0 ; i < 1000 ; i ++ ) {
15             STREAM_ARRAY[i] = i;
16         }
17     }
18
19     /**
20      *
21      * 处理原始整数数据 (int 类型) 的流。它专门用于处理整数数据，因此在处理数值型
22      * 数据时更加高效。
23      *
24      * IntStream 提供了一些特定于整数数据的操作，例如求和、平均值、最大值、最小值
25      * 等。这些操作在处理数值数据时非常有用。
26      *
27      * 在 IntStream 中，整数数据是原始类型，不需要进行装箱 (boxing) 操作。这可
28      * 以提高性能，尤其在大量数据处理时。
29
30     *
31     * @return
32     */
33
34     private static IntStream createStreamWithArraysStream() {
35         return Arrays.stream(STREAM_ARRAY);
36     }
37
38
39     private static Stream<Integer> createStreamWithStreamOf() {
40         return Stream.of(1, 2, 3, 4, 5, 6);
41     }
42
43
44     public static void main(String[] args) {
45         System.out.println("===== Arrays.stream(array) CREATE =====");
46         try (IntStream stream = createStreamWithArraysStream()) {
47             stream.forEach(System.out::println);
48         }
49
50         System.out.println("===== Stream.of(array) CREATE =====");
51         try (Stream<Integer> stream = createStreamWithStreamOf()) {
52             stream.forEach(System.out::println);
53         }
54     }
55
56
57 }

```

如果是集合，可以直接调用 `Collection#stream()` 方法来创建流

```

1 /**
2  * Returns a sequential {@code Stream} with this collection as its
3  * source.

```

```

3  *
4  * <p>This method should be overridden when the {@link #spliterator()} }
5  * method cannot return a spliterator that is {@code IMMUTABLE},
6  * {@code CONCURRENT}, or <em>late-binding</em>. (See {@link
7  #spliterator()} }
8  * for details.)
9  *
10 * {@implSpec
11 * The default implementation creates a sequential {@code Stream} from
12 * the
13 * collection's {@code Spliterator}.
14 *
15 * @return a sequential {@code Stream} over the elements in this
16 * collection
17 * @since 1.8
18 */
19 default Stream<E> stream() {
20     return StreamSupport.stream(spliterator(), false);
21 }

```

```

1 package create;
2
3 import java.util.*;
4 import java.util.stream.Stream;
5
6 public class CollectionStreamCreate {
7
8     private static final int COLLECTION_DATA_SIZE = 1000;
9     private static final Set<String> STRING_SET = new HashSet<>(
10         COLLECTION_DATA_SIZE);
11     private static final Queue<String> STRING_QUEUE = new LinkedList<>(
12         ());
13
14     static {
15         for (int i = 0 ; i < COLLECTION_DATA_SIZE ; i ++ ) {
16             STRING_SET.add("Set Item " + i);
17         }
18         for (int i = 0 ; i < COLLECTION_DATA_SIZE ; i ++ ) {
19             STRING_QUEUE.add("Queue Item " + i);
20         }
21     }
22
23     public static void main(String[] args) {
24
25         Collection<String> setCollection = STRING_SET;
26         Collection<String> queueCollection = STRING_QUEUE;
27
28         try (

```

```

27         final Stream<String> setStream =
setCollection.stream();
28         final Stream<String> queueStream =
queueCollection.stream()
29     ) {
30         System.out.println("SET STREAM COUNT: " +
setStream.count());
31         System.out.println("QUEUE STREAM COUNT: " +
queueStream.count());
32     }
33
34
35 }
36
37 }
38

```

如果在多线程环境下，可能需要调用 `Collectiton#parallelStream()` 或者 `Stream#parallel()` 创建并发流。可以使用 `Stream#sequential()` 将并发流转换为普通流使其强制保持顺序，或者使用 `Stream#forEachOrdered()`

## 操作流

`Stream` 提供了许多有用的操作流的方法

### 过滤

使用 `filter(Predicate<? super T>)` 从流中筛选出我们需要的元素。

```

1 public class NameFilter {
2
3     public static void main(String[] args) {
4         final ArrayList<String> list = new ArrayList<>();
5         list.addAll(List.of("王力宏", "王一博", "王晨", "邵贵龙", "李芙蓉",
6             "王媛", "大王叫我来巡山"));
7         try (Stream<String> stream = list.stream()) {
8             stream.filter(element ->
9                 element.contains("王")).forEach(System.out::println);
10        }
11    }

```

`filter(Predicate<? super T>)` 接收一个 `Predicate`。

```

1 /**
2  * Represents a predicate (boolean-valued function) of one argument.

```

```

3  *
4  * <p>This is a <a href="package-summary.html">functional
5  interface</a>
6  *
7  * @param <T> the type of the input to the predicate
8  *
9  * @since 1.8
10 */
11 @FunctionalInterface
12 public interface Predicate<T> {
13
14     /**
15      * Evaluates this predicate on the given argument.
16      *
17      * @param t the input argument
18      * @return {@code true} if the input argument matches the
19      * predicate,
20      * otherwise {@code false}
21      */
22     boolean test(T t);
23
24     /** 其他逻辑运算方法 */
25 }

```

Predicate 是 Java8 提供的一个函数式接口，接受一个参数返回一个布尔值，我们可以使用 Lambda 表达式传递给 filter 方法。

```

1 | <a unclosed stream instance>.filter(element -> element.contains("王"));
  // 筛选出名字中带 "王" 的字符串

```

forEach(Consumer) 接受一个 Consumer。Consumer 是 Java8 提供的一个函数式接口，接受一个单输入参数且无返回值的方法。类名/对象名::方法名是 Java8 引入的新语法。

```

1 // 表示对流中每个元素作为参数调用 System.out.println
2 // 其中 System.out 是一个对象
3 <a unclosed stream instance>.forEach(System.out::println);

```

## 映射

通过某些操作，将一个流中的元素转化成新的流中的元素。当前后元素是一对一关系时，使用 map(Function)。

```

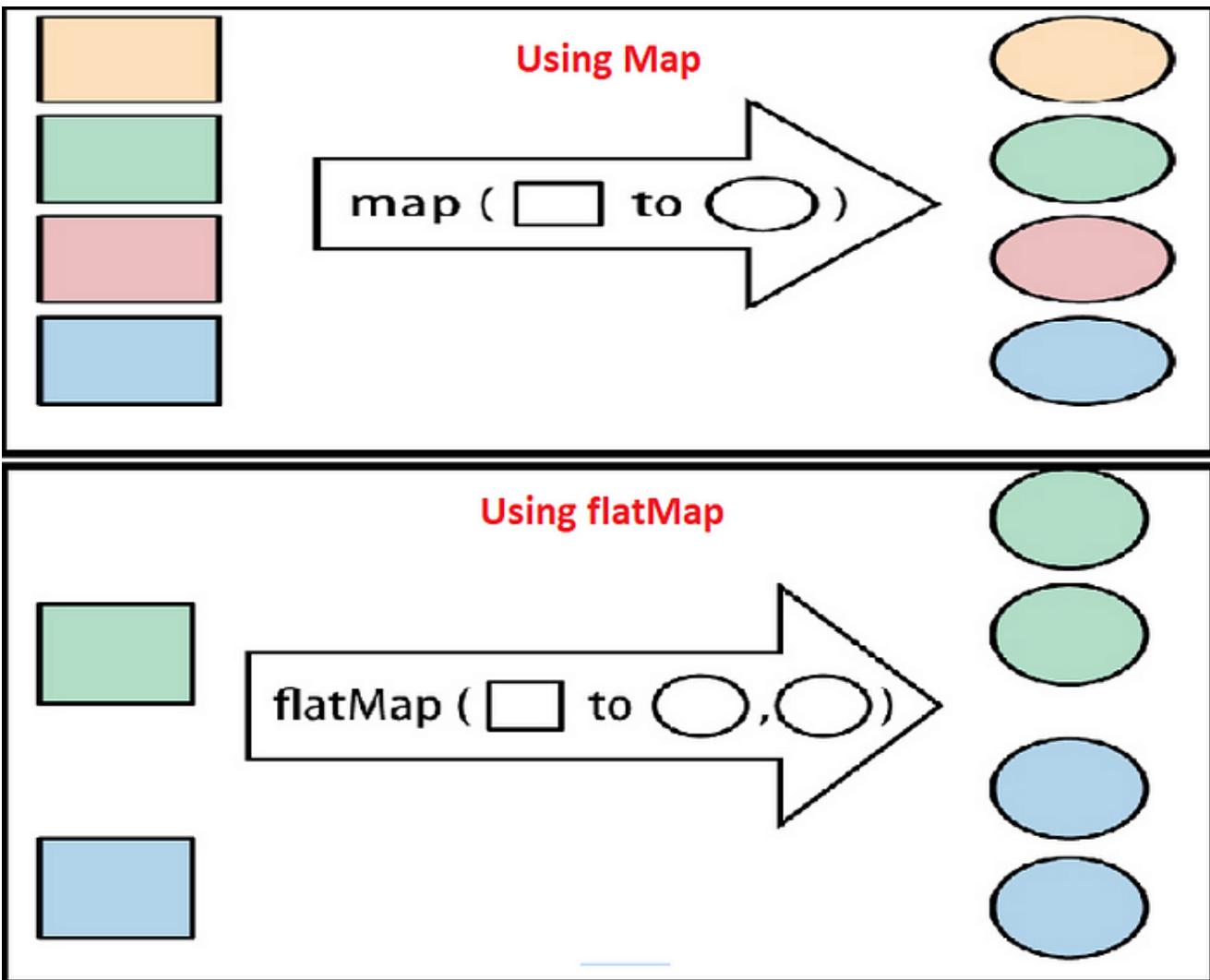
1 package map;
2
3 import entity.Person;
4
5 import java.util.HashSet;

```

```
6 import java.util.List;
7 import java.util.Set;
8 import java.util.stream.Collectors;
9 import java.util.stream.Stream;
10
11 public class UpdatePersonAgeMap {
12
13     public static void main(String[] args) {
14         HashSet<Person> set = new HashSet<>(List.of(
15             new Person("Alice", 21),
16             new Person("Bob", 30),
17             new Person("Seaborn", 20)
18         ));
19         System.out.println("=====");
20         System.out.println("Data to be updated =====");
21         System.out.println(set);
22         Set<Person> updatedSet = null;
23         try (final Stream<Person> stream = set.stream()) {
24             updatedSet = stream.map(p -> new Person(p.getName(),
25                 p.getAge() + 5)).collect(Collectors.toSet());
26         }
27         System.out.println("=====");
28         System.out.println("Updated data (Age plus 5) =====");
29     }
30 }
```

Function 是 Java8 提供的一个函数式接口，接受一个参数并返回一个对象，我们可以使用 Lambda 表达式传递给 map 方法。

当前后元素关系是一对多关系时，使用 flatMap(Function) 方法， flat(Function) 和 flatMap(Function) 对比如下



将若干句子拆分成一个单词列表

```

1 package map;
2
3 import java.util.Arrays;
4 import java.util.List;
5 import java.util.stream.Collectors;
6 import java.util.stream.Stream;
7
8 public class SentenceSplitflatMap {
9
10    public static void main(String[] args) {
11
12        List<String> sentences = List.of(
13            "Hello world",
14            "Java Stream API",
15            "flatMap example"
16        );
17
18        try (Stream<String> stream = sentences.stream()) {
19            final List<String> words = stream.map(sentence ->
20                sentence.split(" "))
21                    .flatMap(Arrays::stream)
22
23        }
24    }
25
26    public static void main() {
27        System.out.println(words);
28    }
29
30 }
```

```

21         .collect(Collectors.toList());
22         System.out.println(words);
23     }
24 }
25
26 }
```

将一个二维线性表扁平化为一维线性表

```

1 package map;
2
3 import java.util.Collection;
4 import java.util.List;
5 import java.util.stream.Collectors;
6
7 public class FlatArrayFlatMap {
8
9     public static void main(String[] args) {
10         List<List<Integer>> listOfLists = List.of(
11             List.of(1, 2, 3),
12             List.of(4, 5, 6),
13             List.of(7, 8, 9)
14         );
15         List<Integer> flatList =
16             listOfLists.stream().flatMap(Collection::stream).collect(Collectors.to
17             List());
18         System.out.println(flatList);
19     }
20 }
```

## 匹配

Stream 接口提供了三个方法可供进行元素匹配，分别是

- anyMatch (Predicate)，只要有一个元素满足匹配条件就返回 true
- allMatch (Predicate)，只要有一个元素不满足匹配条件就返回 false
- noneMatch (Predicate)，只要有一个元素满足匹配条件就返回 false

姓名匹配

```

1 package match;
2
3 import java.util.ArrayList;
4 import java.util.List;
5
6 public class NameMatch {
```

```
7
8     public static void main(String[] args) {
9         ArrayList<String> nameList = new ArrayList<>(List.of(
10            "周杰伦",
11            "王力宏",
12            "陶喆",
13            "林俊杰"
14        ));
15        // 查看列表中是否有任何一个姓王的姓名
16        boolean anyWangSurname = nameList.stream().anyMatch(name ->
17            name.startsWith("王"));
18        System.out.println(anyWangSurname);
19
20        // 查看列表是不是所有名字都是双字名
21        boolean allDoubleName = nameList.stream().allMatch(name ->
22            name.length() == 2);
23        System.out.println(allDoubleName);
24
25        // 查看列表中是否没有姓伍的姓名
26        boolean noneWuSurname = nameList.stream().noneMatch(name ->
27            name.startsWith("队伍"));
28        System.out.println(noneWuSurname);
29    }
30}
```

## 偶数匹配

```
1 package match;
2
3 import java.util.ArrayList;
4 import java.util.List;
5
6 public class NameMatch {
7
8     public static void main(String[] args) {
9         ArrayList<String> nameList = new ArrayList<>(List.of(
10            "周杰伦",
11            "王力宏",
12            "陶喆",
13            "林俊杰"
14        ));
15        // 查看列表中是否有任何一个姓王的姓名
16        boolean anyWangSurname = nameList.stream().anyMatch(name ->
17            name.startsWith("王"));
18        System.out.println(anyWangSurname);
```

```

19     // 查看列表是不是所有名字都是双字名
20     boolean allDoubleName = nameList.stream().allMatch(name ->
21         name.length() == 2);
22     System.out.println(allDoubleName);
23
24     // 查看列表中是否没有姓伍的姓名
25     boolean noneWuSurname = nameList.stream().noneMatch(name ->
26         name.startsWith("队伍"));
27     System.out.println(noneWuSurname);
28
29 }
30

```

## 组合

reduce 的主要作用就是将 Stream 的元素组合起来，有两种用法。

Optional<T> reduce(BinaryOperator)

没有起始值，只有BinaryOperator 参数，表示一个二元操作，返回 Optional。

下面是一个例子，此时 reduce 返回空的 Optional 对象

```

1 package reduce;
2
3 import java.util.List;
4 import java.util.Optional;
5
6 public class ReduceNullExample {
7
8     public static void main(String[] args) {
9         List<Integer> list = List.of();
10        Optional<Integer> sum = list.stream().reduce(
11            (x, y) -> x + y
12        );
13        if (sum.isPresent()) {
14            System.out.println("SUMMATION: " + sum); // 不会执行
15        } else {
16            System.out.println("列表为空，没有计算结果");
17        }
18    }
19
20 }

```

T reduce(T identity , BinaryOperator)

有起始值有 BinaryOperator 参数。此时返回类型和起始值类型一致。

求出列表中的最大值

```
1 package reduce;
2
3 import java.util.ArrayList;
4
5 public class MaxNumberReduce {
6
7     public static void main(String[] args) {
8         ArrayList<Integer> numbers = new ArrayList<>();
9         for (int i = 0 ; i < 10 ; i ++ ) {
10             numbers.add(i);
11         }
12         int initialValue = Integer.MIN_VALUE;
13         Integer maxValue = numbers.stream().reduce(initialValue, (x,
14 y) -> x > y ? x : y);
15         System.out.println("The max value of the list: " + maxValue);
16     }
17 }
```

0 到 100 相加

```
1 package reduce;
2
3 import java.util.ArrayList;
4 import java.util.List;
5
6 public class NumbersSummationReduce {
7
8     private static List<Integer> list() {
9         ArrayList<Integer> list = new ArrayList<>();
10        for (int i = 0 ; i <= 100 ; i ++ ) {
11            list.add(i);
12        }
13        return list;
14    }
15
16    public static void main(String[] args) {
17        List<Integer> list = list();
18        int summation = 0;
19        Integer result = list.stream().reduce(summation, (x, y) -> {
20            System.out.println("x = " + x + "; y = " + y);
21            return x + y;
22        });
23        System.out.println(result);
24    }
}
```

```
25  
26 }  
27
```

求平均值

```
1 package reduce;  
2  
3 import java.util.ArrayList;  
4 import java.util.Random;  
5  
6 public class AverageNumberReduce {  
7  
8     private static final int SAMPLE_NUM = 100;  
9  
10    public static void main(String[] args) {  
11  
12        Random random = new Random();  
13        ArrayList<Double> scores = new ArrayList<>();  
14        for (int i = 0 ; i < SAMPLE_NUM ; i ++ ) {  
15            int append = random.nextInt(20) + 81;  
16            scores.add((double) append);  
17        }  
18  
19        double initialValue = 0;  
20        System.out.println(scores.stream().reduce(initialValue, (x, y)  
-> (x + y / (double) SAMPLE_NUM)));  
21    }  
22}  
23  
24 }
```

## 结果收集

结果收集主要使用 `collect(Collector)` 方法，在Java中，`java.util.stream.Collector`接口是用于将流中元素进行收集、聚合和归约操作的关键组件。允许将流的元素累积到一个可变结果容器中，从而生成一个最终结果。

1. `Supplier<A> supplier()`
2. `BiConsumer<A, T> accumulator()`
3. `BinaryOperator<A> combiner()`
4. `Function<A, R> finisher()`
5. `Set<Characteristics> characteristics()`

```
1 package result;  
2
```

```
3 import java.util.Collections;
4 import java.util.EnumSet;
5 import java.util.Set;
6 import java.util.function.BiConsumer;
7 import java.util.function.BinaryOperator;
8 import java.util.function.Function;
9 import java.util.function.Supplier;
10 import java.util.stream.Collector;
11
12 public class StringJoinCollector implements Collector<String,
13     StringBuilder, String> {
14     @Override
15     public Supplier<StringBuilder> supplier() {
16         return StringBuilder::new;
17     }
18
19     /**
20      * accumulator() 方法用于将流中的元素逐个添加到中间结果容器中，适用于顺序流
21      * 和并行流。
22     */
23     @Override
24     public BiConsumer<StringBuilder, String> accumulator() {
25         return StringBuilder::append;
26     }
27
28     /**
29      * combiner() 方法用于在并行流操作中合并多个部分结果容器，只适用于并行流。
30     */
31     @Override
32     public BinaryOperator<StringBuilder> combiner() {
33         return StringBuilder::append;
34     }
35
36     @Override
37     public Function<StringBuilder, String> finisher() {
38         return StringBuilder::toString;
39     }
40
41     @Override
42     public Set<Characteristics> characteristics() {
43         return
44             Collections.unmodifiableSet(EnumSet.of(Characteristics.IDENTITY_FINISH
45         ));
46     }
47 }
```

## 生成集合

统一收集某簇对象的某个属性

```
1 package result;
2
3 import entity.Department;
4
5 import java.util.Arrays;
6 import java.util.List;
7 import java.util.Map;
8 import java.util.Set;
9 import java.util.stream.Collectors;
10
11 public class EntityIdCollect {
12
13     public static void main(String[] args) {
14         List<Department> list =
15             Arrays.asList(new Department(17L), new
16             Department(85L), new Department(100L), new Department(100L));
17
18         // 1. collect 成 set
19         Set<Long> idSet =
20             list.stream().map(Department::getDeptId).collect(Collectors.toSet());
21         System.out.println("idSet: " + idSet);
22
23         // 2. collect 成 list
24         List<Long> idList =
25             list.stream().map(Department::getDeptId).collect(Collectors.toList());
26         System.out.println("idList: " + idList);
27
28         // 3. collect 成 map
29         Map<Department, Long> idMap =
30             list.stream().distinct().collect(Collectors.toMap(x -> x,
31             Department::getDeptId));
32         System.out.println(idMap);
33     }
34 }
35 }
```

## 字符串

自定义拼接字符串

```
1 package result;
2
3 import java.util.Arrays;
4 import java.util.List;
5
```

```

6  public class StringJoinCollect {
7
8      public static void main(String[] args) {
9          List<String> list = Arrays.asList(
10              "你好",
11              "我好",
12              "大家好"
13          );
14          String joinString = list.stream().collect(new
15              StringJoinCollector());
16          System.out.println(joinString);
17      }
18  }
19

```

带分隔符地拼接字符串

```

1 package result;
2
3 import java.util.Arrays;
4 import java.util.List;
5 import java.util.StringJoiner;
6 import java.util.stream.Collectors;
7
8 public class StringJoinWithDelimiterCollect {
9
10     public static void main(String[] args) {
11         List<String> list = Arrays.asList(
12             "你好",
13             "我好",
14             "大家好"
15         );
16         // Collectors.joining(", ") 的底层是 StringJoiner
17         String collect = list.stream().collect(Collectors.joining(", "
18             ));
19         System.out.println(collect);
20     }
21 }
22

```

## 数据处理

Collectors.averagingInt(map) 和 summarizingInt(map)

```

1 package result;
2
3 import java.util.Arrays;

```

```
4 import java.util.IntSummaryStatistics;
5 import java.util.List;
6 import java.util.stream.Collectors;
7
8 public class DataProcessCollect {
9
10    public static void main(String[] args) {
11        List<Integer> numbers = Arrays.asList(10, 20, 30, 40, 50);
12
13        // 输出 element / 10 的平均值
14        Double average =
15        numbers.stream().collect(Collectors.averagingInt(value -> value /
16        10));
17        System.out.println(average);
18
19        // 输出 element / 2 的个数, 总和, 最小值, 平均值, 最大值
20        IntSummaryStatistics statistic =
21        numbers.stream().collect(Collectors.summarizingInt(n -> n / 2));
22        System.out.println(statistic);
23    }
24}
```

## 练习

### 计算列表中所有正整数的平均值

```
1 package practice.primary;
2
3 import practice.RandomUtils;
4
5 import java.util.Collection;
6 import java.util.HashSet;
7 import java.util.stream.Collectors;
8
9 /**
10  * 计算列表中所有正整数的平均值。
11 */
12 public class PositiveIntegerAverage {
13
14     private static Collection<Integer> collection() {
15         HashSet<Integer> set = new HashSet<>();
16         for (int i = 0 ; i < 10 ; i ++ ) {
17             set.add(RandomUtils.randomInteger(80) *
RandomUtils.randomSign());
```

```

18     }
19     return set;
20 }
21
22 public static void main(String[] args) {
23     Collection<Integer> collection = collection();
24
25     double average = collection.stream()
26         .filter(element -> element > 0) // 筛选
27         .mapToDouble(Integer::doubleValue) // 将正
28         .average() // 取其
29         .orElse(0.0) // 若
30         .OptionalDouble isEmpty 取 0.0
31     ;
32     System.out.println(average);
33
34     double averageBySummarizing =
35     collection.stream().filter(element -> element >
36     0).collect(Collectors.summarizingInt(e -> e)).getAverage();
37     System.out.println(averageBySummarizing);
38 }
```

## 将字符串列表转换为大写

```

1 package practice.primary;
2
3 import java.util.Arrays;
4 import java.util.List;
5 import java.util.stream.Collectors;
6
7 /**
8  * 将字符串列表转换为大写。
9 */
10 public class StringTransferToUpperCase {
11
12     private static List<String> words = Arrays.asList(
13         "apple", "banana", "car", "dog", "elephant",
14         "flower", "guitar", "house", "ice cream", "jacket",
15         "kite", "lamp", "moon", "notebook", "orange",
16         "pineapple", "queen", "rainbow", "sun", "tree"
17     );
18
19     public static void main(String[] args) {
```

```

20     StringTransferToUpperCase.words.stream()
21         .map(s -> s.toUpperCase())           // 一对一的全部转为大
22         .collect(Collectors.toList())       // 收集结果成集合
23     ;
24 }
25
26 }
27

```

## 找出列表中的最大值和最小值。

```

1 package practice.primary;
2
3 import practice.RandomUtils;
4
5 import java.util.Comparator;
6 import java.util.IntSummaryStatistics;
7 import java.util.List;
8 import java.util.Optional;
9 import java.util.stream.Collectors;
10
11 /**
12  * 找出列表中的最大值和最小值。
13 */
14 public class FindMinMaxInList {
15
16     public static void main(String[] args) {
17         List<Integer> list =
18             RandomUtils.randomIntegerList(20, 21, 80);
19         System.out.println(list);
20
21         /* SUMMARIZING */
22         System.out.println("/* SUMMARIZING */");
23         IntSummaryStatistics statistics =
24             list.stream().collect(Collectors.summarizingInt(element -> element));
25         System.out.println("Min: " + statistics.getMax());
26         System.out.println("Max: " + statistics.getMin());
27
28         /* MAX, MIN */
29         System.out.println("/* MAX, MIN */");
30         Optional<Integer> min =
31             list.stream().max(Comparator.comparingInt(x -> x));
32         Optional<Integer> max =
33             list.stream().min(Comparator.comparingInt(x -> x));
34         System.out.println("Min: " + min.orElse(Integer.MAX_VALUE));
35         System.out.println("Max: " + max.orElse(Integer.MIN_VALUE));
36     }
37

```

```
35  }
36
```

## 偶数筛

```
1 package practice.primary;
2
3 import practice.RandomUtils;
4
5 import java.util.List;
6 import java.util.stream.Collectors;
7
8 /**
9  * 从列表中筛选出所有偶数。
10 */
11 public class EvenSieve {
12
13     public static void main(String[] args) {
14         List<Integer> list = RandomUtils.randomIntegerList(20, 80 + 1,
15 20);
16         List<Integer> evenList = list.stream().filter(n -> (n & 1) ==
0).collect(Collectors.toList());
17         System.out.println("evenList: " + evenList);
18     }
19 }
20
```

## 找出列表中长度大于等于5的字符串

```
1 package practice.primary;
2
3
4 import java.util.List;
5 import java.util.stream.Collectors;
6
7 /**
8  * 找出列表中长度大于等于5的字符串
9 */
10 public class LengthGreater5String {
11
12     public static void main(String[] args) {
13         List<String> list = List.of("apple", "banana", "kiwi",
14 "grape", "orange");
15         System.out.println(list.stream().filter(s -> s.length() >=
5).collect(Collectors.toList()));
16     }
17 }
```

## 将整数列表中的所有元素平方后生成新的列表

```

1 package practice.primary;
2
3 import practice.RandomUtils;
4
5 import java.util.List;
6 import java.util.stream.Collectors;
7
8 /**
9  * 将整数列表中的所有元素平方后生成新的列表
10 */
11 public class IntegerSquare {
12
13     public static void main(String[] args) {
14         List<Integer> list =
15             RandomUtils.randomIntegerList(20, 0, 40);
16
17         List<Integer> squareList = list.stream().map(e -> e *
18             e).collect(Collectors.toList());
19         System.out.println(list);
20         System.out.println(squareList);
21     }
22 }
23

```

## 将字符串列表按照长度排序

```

1 package practice.primary;
2
3 import practice.RandomUtils;
4
5 import java.util.ArrayList;
6 import java.util.Comparator;
7 import java.util.List;
8 import java.util.stream.Collectors;
9
10 /**
11  * 将字符串列表按照长度排序
12 */
13 public class StringLengthSort {
14
15     public static void main(String[] args) {
16         List<String> list = new ArrayList<>();

```

```
17     for (int i = 0 ; i < 10 ; i ++ ) {
18         list.add(RandomUtils.generateRandomString());
19     }
20     List<String> sortedList =
21         list.stream().sorted(Comparator.comparingInt(String::length)).collect(
22             Collectors.toList());
23     sortedList.forEach(System.out::println);
24 }
25 }
```

## 将两个整数列表合并为一个列表，然后去除重复元素

```
1 package practice.primary;
2
3 import practice.RandomUtils;
4
5 import java.util.Collection;
6 import java.util.Comparator;
7 import java.util.List;
8 import java.util.stream.Collectors;
9 import java.util.stream.Stream;
10
11 /**
12  * 将两个整数列表合并为一个列表，然后去除重复元素
13 */
14 public class MergeIntegerList {
15
16     public static void main(String[] args) {
17
18         List<Integer> firstList = RandomUtils.randomIntegerList(20,
19             10, 10);
20         List<Integer> secondList = RandomUtils.randomIntegerList(20,
21             10, 10);
22
23         firstList.sort(Comparator.comparingInt(x -> x));
24         secondList.sort(Comparator.comparingInt(x -> x));
25
26         System.out.println("firstList = " + firstList);
27         System.out.println("secondList = " + secondList);
28
29         List<Integer> mergeDistinctList =
30             Stream.of(firstList,
31             secondList).flatMap(Collection::stream).distinct().collect(Collectors.
32             toList());
33         System.out.println("mergeDistinctList = " +
34             mergeDistinctList);
35 }
```

```
31     }
32
33 }
34
```

## 找出列表中第一个大于10的元素。

```
1 package practice.intermediate;
2
3 import practice.RandomUtils;
4
5 import java.util.List;
6 import java.util.stream.Collectors;
7
8 /**
9  * 找出列表中第一个大于10的元素。
10 */
11 public class FirstGreater10Element {
12
13     public static void main(String[] args) {
14         // 生成一个包含 100 个随机整数的列表，范围在 0 到 20 之间
15         List<Integer> list =
16             RandomUtils.randomIntegerList(100, 0, 20);
17         System.out.println(list);
18
19         // 使用 takeWhile 方法获取满足条件的元素，直到遇到第一个不满足条件的元素，然后统计满足条件的元素个数
20         long resultIndex = list.stream().takeWhile(integer -> integer
21             <= 10).collect(Collectors.toList()).stream().count();
22
23         // 输出满足条件的元素个数和第一个大于 10 的元素的值
24         System.out.println("resultIndex: " + resultIndex + "; result:
25             " + list.get((int) resultIndex));
26     }
27 }
```

## 计算列表中所有数字的乘积。

```
1 package practice.intermediate;
2
3 import practice.RandomUtils;
4
5 import java.util.List;
6
7 /**
8  * 计算列表中所有数字的乘积。
9  */
```

```
10 public class AllNumberMultiplication {
11
12     public static void main(String[] args) {
13         List<Integer> list =
14             RandomUtils.randomIntegerList(5, 1, 10);
15         System.out.println(list);
16         System.out.println(list.stream().mapToLong(x -> x).reduce(1L,
17             (acc, x) -> acc * x));
18     }
19 }
20
```

## 将字符串列表中的元素连接成一个以逗号分隔的字符串

```
1 package practice.intermediate;
2
3 import practice.RandomUtils;
4
5 import java.util.List;
6 import java.util.stream.Collectors;
7
8 /**
9  * 将字符串列表中的元素连接成一个以逗号分隔的字符串
10 */
11 public class CommaDelimiterJoinString {
12
13     public static void main(String[] args) {
14         List<String> list =
15             RandomUtils.generateRandomStringList(10);
16         list.forEach(System.out::println);
17         String commaJoinString =
18             list.stream().collect(Collectors.joining(",\n"));
19         System.out.println(commaJoinString);
20     }
21 }
22
```

## 找出列表中长度最长的字符串。

```
1 package practice.intermediate;
2
3 import practice.RandomUtils;
4
5 import java.util.Comparator;
6 import java.util.List;
```

```
7  /**
8  * 找出列表中长度最长的字符串。
9  */
10 public class LengthMaxString {
11
12
13     public static void main(String[] args) {
14         List<String> list =
15             RandomUtils.generateRandomStringList(10);
16         String maxLengthString =
17             list.stream().max(Comparator.comparingInt(String::length)).orElseThrow(
18             () ->
19                 System.out.println(maxLengthString));
20     }
21 }
```

## 将列表中的字符串按照字母顺序排序，并去除重复项。

```
1 package practice.intermediate;
2
3 import java.util.ArrayList;
4 import java.util.List;
5 import java.util.stream.Collectors;
6
7 /**
8 * 将列表中的字符串按照字母顺序排序，并去除重复项。
9 */
10 public class StringDictionarySortDistinct {
11
12     private static List<String> list() {
13         List<String> strings = new ArrayList<>();
14         strings.add("apple");
15         strings.add("orange");
16         strings.add("banana");
17         strings.add("apple");
18         strings.add("kiwi");
19         strings.add("banana");
20         return strings;
21     }
22
23     public static void main(String[] args) {
24         List<String> list = list();
25         List<String> distinctSortedStringList =
26             list.stream().distinct().sorted().collect(Collectors.toList());
27         System.out.println(distinctSortedStringList);
28     }
29 }
```

```
29  }
30
```

## 将整数列表分组，使得奇数和偶数分开。

```
1 package practice.intermediate;
2
3 import practice.RandomUtils;
4
5 import java.util.List;
6 import java.util.Map;
7 import java.util.stream.Collectors;
8
9 /**
10  * 将整数列表分组，使得奇数和偶数分开。
11 */
12 public class EvenOddIntegerGroup {
13
14     public static void main(String[] args) {
15         List<Integer> list = RandomUtils.randomIntegerList(100, 20,
16         81);
17         Map<Boolean, List<Integer>> evenOddGroupMap =
18         list.stream().collect(Collectors.groupingBy(element -> ((element & 1)
19         == 0)));
20         evenOddGroupMap.entrySet().forEach(System.out::println);
21     }
22 }
```

## 找出列表中长度大于等于3的字符串，并将它们以逗号分隔的形式输出。

```
1 package practice.intermediate;
2
3 import practice.RandomUtils;
4
5 import java.util.List;
6 import java.util.stream.Collectors;
7
8 /**
9  * 找出列表中长度大于等于3的字符串，并将它们以逗号分隔的形式输出。
10 */
11 public class StringLengthGreaterThanOrEqualTo3CommaDelimiter {
12
13     public static void main(String[] args) {
```

```

14     List<String> list = RandomUtils.generateRandomStringList(100,
15         1, 10);
16     String result = list.stream().filter(s -> s.length() >
17         3).collect(Collectors.joining(",\n"));
18     System.out.println(result);
19 }
20

```

## 将字符串列表中的每个字符串的每个字符拆分成一个字符列表。

```

1 package practice.intermediate;
2
3 import practice.RandomUtils;
4
5 import java.util.List;
6 import java.util.stream.Collectors;
7
8 /**
9  * 将字符串列表中的每个字符串的每个字符拆分成一个字符列表。
10 */
11 public class SplitStringListIntoCharList {
12
13     public static void main(String[] args) {
14         List<String> list = RandomUtils.generateRandomStringList(20);
15         List<Character> letters = list.stream()
16             .flatMapToInt(String::chars)           // 转为 IntStream
17             .mapToObj(i -> (char) i)           // 转为
18             Stream<Character>
19             .collect(Collectors.toList());     // 收集结果
20         System.out.println(letters);
21     }
22 }
23

```

## 找出列表中所有的质数

```

1 package practice.advanced;
2
3 import practice.RandomUtils;
4
5 import java.util.List;
6 import java.util.stream.Collectors;
7
8 public class FindPrimeNumber {

```

```
9
10 public static void main(String[] args) {
11     List<Integer> list =
12         RandomUtils.randomIntegerList(100, 20, 81);
13     List<Integer> primeList = list.stream()
14         .filter(FindPrimeNumber::prime)
15         .collect(Collectors.toList());
16     System.out.println(primeList);
17 }
18
19 private static boolean prime(int n) {
20     if (n < 1) {
21         return false;
22     }
23     if (n == 1) {
24         return true;
25     }
26     for (int i = 2 ; i < Math.sqrt(n) ; i++) {
27         if (n % i == 0) {
28             return false;
29         }
30     }
31     return true;
32 }
33
34 }
```

## 统计列表中每个字符串中每个字符出现的次数

```
1 package practice.advanced;
2
3 import practice.RandomUtils;
4
5 import java.util.List;
6 import java.util.Map;
7 import java.util.stream.Collectors;
8
9 /**
10 * 统计列表中每个字符串中每个字符出现的次数
11 */
12 public class StringCharacterCounter {
13
14     public static void main(String[] args) {
15         List<String> list = RandomUtils.generateRandomStringList(20);
16         Map<Character, Long> map = list.stream().flatMap(s ->
17             s.chars().mapToObj(c -> (char) c)).collect(
18                 Collectors.groupingBy(
19                     c -> c, // 分类依据
20                     Collectors.counting()
21                 )
22             );
23
24         map.forEach((k, v) -> System.out.println(k + " : " + v));
25     }
26 }
```

```
19             Collectors.counting() // 对分完类的各组继续进行
20             )
21         );
22         System.out.println(map);
23     }
24 }
25 }
26 }
```

## 找出列表中长度最长的字符串的长度，使用reduce操作

```
1 package practice.advanced;
2
3 import practice.RandomUtils;
4
5 import java.util.List;
6
7 /**
8  * 找出列表中长度最长的字符串的长度，使用reduce操作
9 */
10 public class FindLengthMaxStringByReduce {
11
12     public static void main(String[] args) {
13         List<String> list = RandomUtils.generateRandomStringList(100);
14         String lengthMaxString = list.stream().reduce("", (acc, x) ->
15             acc.length() > x.length() ? acc : x);
16         System.out.println(lengthMaxString);
17     }
18 }
19 }
```

## 计算列表中所有数字的平方和，使用map和reduce操作

```
1 package practice.advanced;
2
3 import practice.RandomUtils;
4
5 import java.util.List;
6
7 /**
8  * 计算列表中所有数字的平方和，使用map和reduce操作
9 */
10 public class SquareSumByMapAndReduce {
11
12     public static void main(String[] args) {
13         List<Integer> list =
14             RandomUtils.randomIntegerList(5, 0, 10);
15     }
16 }
```

```

15     System.out.println(list);
16     long squareSummation = list.stream().mapToLong(x ->
17         x).reduce(0L, (acc, x) -> acc + x * x);
18     System.out.println(squareSummation);
19 }
20 }
21

```

## 通过一个实例深入了解 Stream Collect

人员信息数据

姓名	子公司	部门	年龄	工资
大壮	上海公司	研发一部	28	3000
二牛	上海公司	研发一部	24	2000
铁柱	上海公司	研发二部	34	5000
翠花	南京公司	测试一部	27	3000
玲玲	南京公司	测试二部	31	4000

现有集团内所有人员列表，需要从中筛选出上海子公司的全部人员



```

1 /**
2  * 现有集团内所有人员列表，需要从中筛选出上海子公司的全部人员
3 */
4 public static List<Employee> filterEmployeesBySubCompany() {
5     return employeeCollection().stream()
6         .filter(employee -> "上海公
7 司".equals(employee.getSubCompany()))
8         .collect(Collectors.toList());
}

```

## 现有集团内所有人员列表，需要从中筛选出上海子公司的全部人员，并按照部门进行分组

```
1  /**
2  * 现有集团内所有人员列表，需要从中筛选出上海子公司的全部人员，并按照部门进行分组
3  */
4  public static Map<String, List<Employee>>
5      filterEmployeeBySubCompanyAndDepartment() {
6      return employeeCollection().stream()
7          .filter(employee -> "上海公
司".equals(employee.getSubCompany())))
8      .collect(Collectors.groupingBy(Employee::getDepartment));
```

## collect / Collector / Collectors 区别与联系

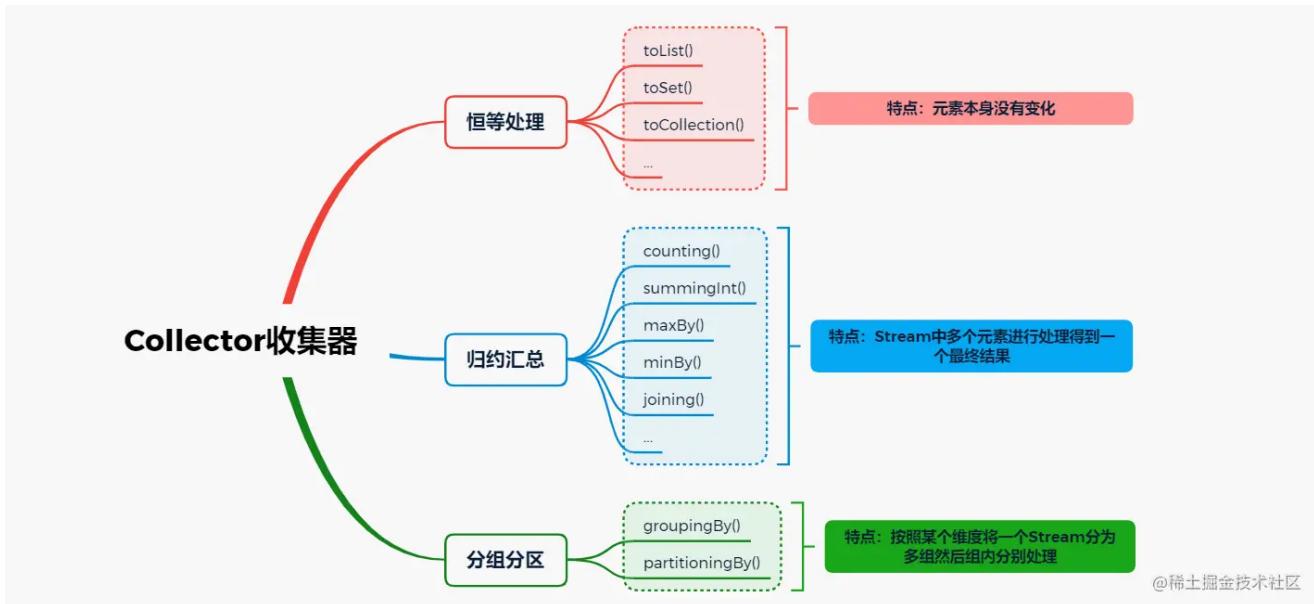
1. collect 是 Stream 流的一个终止方法，会使用传入的收集器对结果执行相关的操作，这个收集器必须是 Collector 接口的某个具体实现类
2. Collector 是一个接口，collect 方法的收集器是 Collector 接口的具体实现类
3. Collectors 是一个工具类，提供了很多静态工厂方法，为了方便开发时使用预置的较为通用的收集器（当然也可以自己实现 Collector）

Stream 结果收集的本质，就是将 Stream 中的元素通过收集器定义的函数处理逻辑进行加工吗，然后输出加工后的结果。



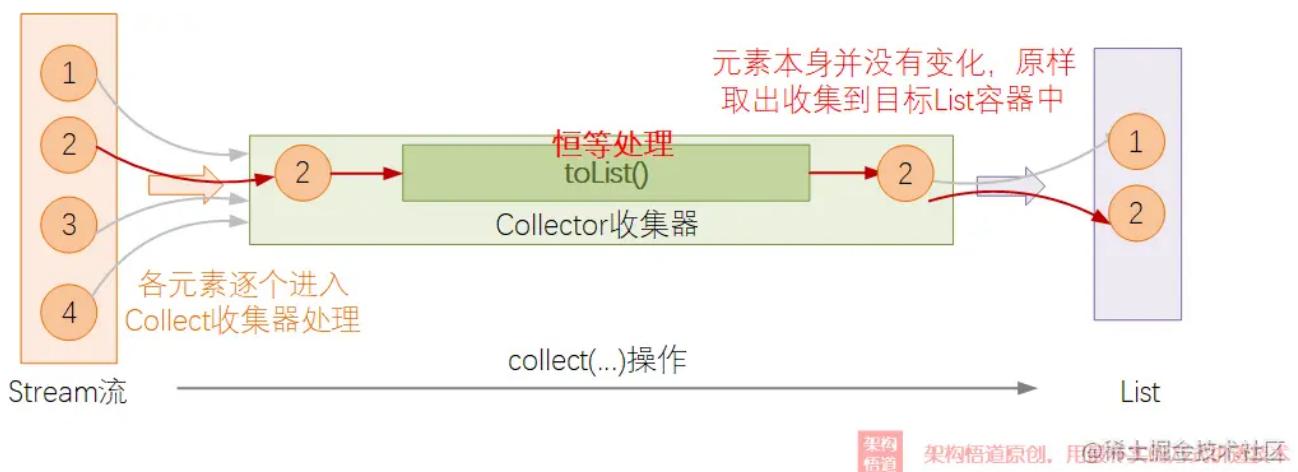
## Collector 使用剖析

根据其执行的操作类型来划分，可将收集器分为几种不同的大类：



## 恒等处理

所谓恒等处理就是指 Stream 的元素经过 Collector 函数处理前后完全不变，例如 `toList()` 操作，只是将最终结果从 Stream 取出并放入 List 对象中，并没有对元素本身进行任何处理



恒等处理类型的Collector是实际编码中最常被使用的一种。

## 归约汇总

对于规约汇总的操作，Stream 流中的元素逐个遍历，进入到 Collector 处理函数中，然后会与上一个元素的处理结果进行合并处理，并得到一个新的结果，以此类推，直到遍历完成，得到最终结果。比如 `Collectors.summingInt()`，方法逻辑如下



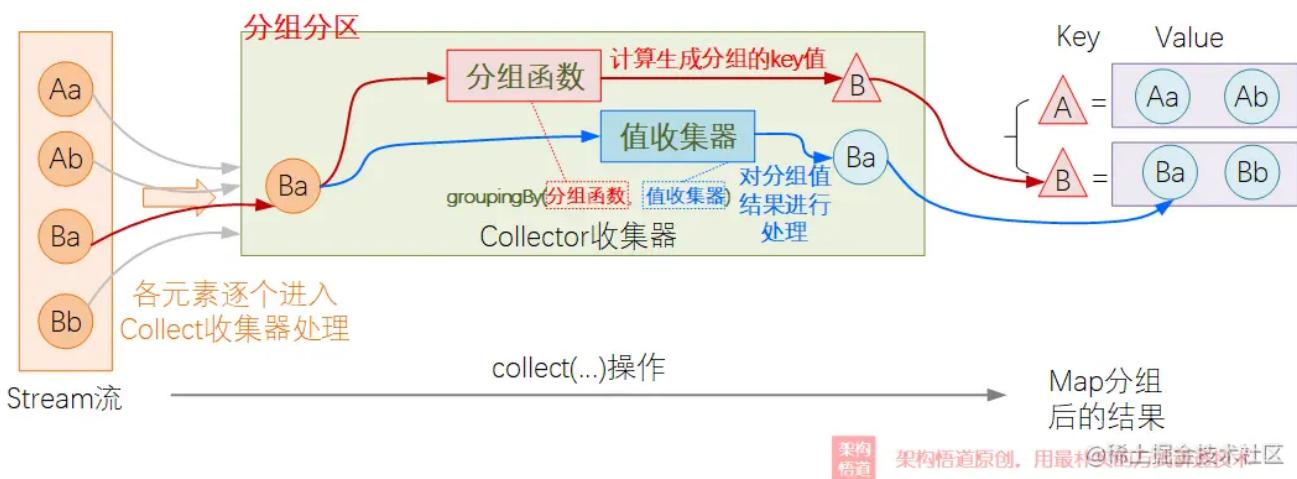
上例中，计算上海子公司每个月需要支付的员工工资

```

1 /**
2  * 计算上海子公司每个月需要支付的员工总工资
3 */
4 private static int ShanghaiSubCompanySalarySum() {
5     return employeeCollection().stream()
6         .filter(employee -> "上海公
司".equals(employee.getSubCompany()))
7         .collect(Collectors.summingInt(employee ->
employee.getSalary()));
8 }
```

## 分组分区

Collectors工具类中提供了groupingBy方法用来得到一个分组操作Collector，其内部处理逻辑可以参见下图的说明：



groupingBy() 需要传入两个参数，分组函数和值收集器

- 分组函数

一个处理函数，基于指定的元素进行处理，返回一个用于分组的值，对于经过此函数处理后返回值相同的元素，将被分配到同一个组里。

- 值收集器

对于分组后的数据元素做进一步处理转换逻辑，此处还是一个常规的 Collector 收集器于 groupingBy() 分组函数和值收集器缺一不可，其中，值收集器默认为 toList()

```
1  public static <T, K> Collector<T, ?, Map<K, List<T>>>
2  groupingBy(Function<? super T, ? extends K> classifier) {
3      return groupingBy(classifier, toList());
4  }
```

按照子公司维度将员工分组（默认值收集器）

```
1  /**
2  * 按照子公司维度将员工分组
3  */
4  private static Map<String, List<Employee>> groupBySubCompany() {
5      return employeeCollection().stream()
6          .collect(Collectors.groupingBy(Employee::getSubCompany))
7          // 值收集器默认为 toList()
8      ;
```

而如果不仅需要分组，还需要对分组后的数据进行处理的时候，则需要同时给定分组函数以及值收集器：

按照子公司分组，并统计每个子公司的员工数

```
1  /**
2  * 按照子公司分组，统计每个子公司的员工数量
3  */
4  private static Map<String, Long> groupBySubCompanyThenCount() {
5      return employeeCollection().stream()
6          .collect(
7              Collectors.groupingBy(
8                  Employee::getSubCompany,
9                  Collectors.counting()
10             )
11         );
12 }
```

## 叠加嵌套

collect 第二个参数类型是 Collector，故允许嵌套。

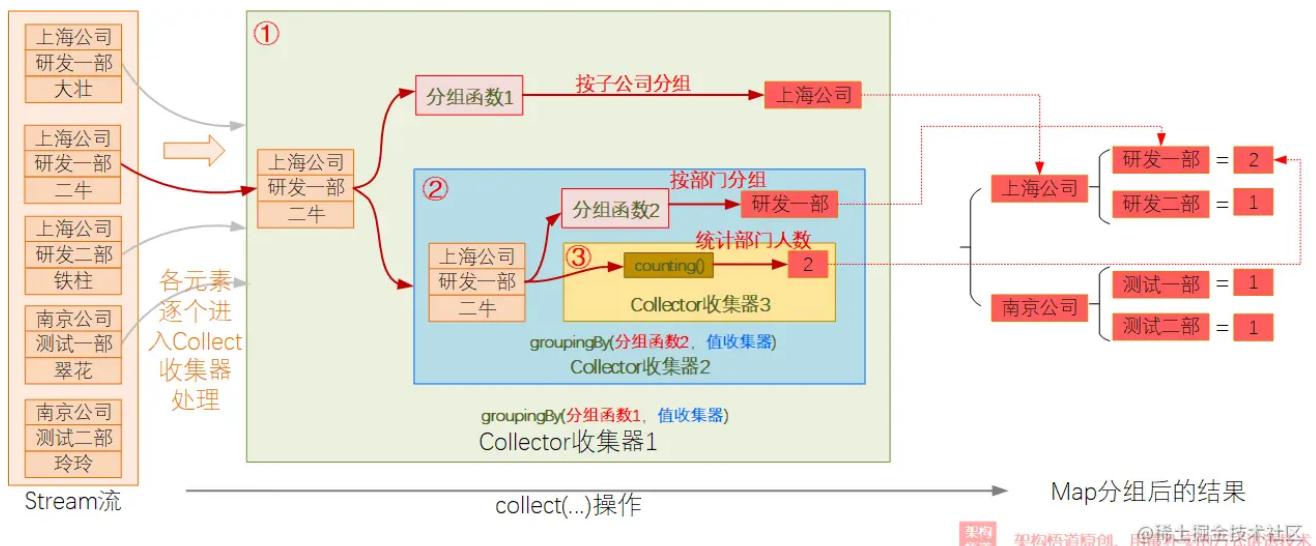
现有整个集团全体员工的列表，需要统计各子公司内各部门下的员工人数。

```

1  /**
2  * 现有整个集团全体员工的列表, 需要统计各子公司内各部门下的员工人数。
3  */
4  private static Map<String, Map<String, Long>>
5  groupBySubCompanyAndSectionThenCount() {
6      return employeeCollection().stream().collect(
7          Collectors.groupingBy(
8              Employee::getSubCompany,
9              Collectors.groupingBy(
10                 Employee::getDepartment,
11                 Collectors.counting()
12             )
13         );

```

处理逻辑如下



## Collectors 提供的值收集器

方法	含义说明
toList	将流中的元素收集到一个List中
toSet	将流中的元素收集到一个Set中
toCollection	将流中的元素收集到一个Collection中
toMap	将流中的元素映射收集到一个Map中
counting	统计流中的元素个数
summingInt	计算流中指定int字段的累加总和。针对不同类型的数字类型, 有不同的方法, 比如summingDouble等

方法	含义说明
averagingInt	计算流中指定int字段的平均值。针对不同类型的数字类型，有不同的方法，比如averagingLong等
joining	将流中所有元素（或者元素的指定字段）字符串值进行拼接，可以指定拼接连接符，或者首尾拼接字符
maxBy	根据给定的比较器，选择出值最大的元素
minBy	根据给定的比较器，选择出值最小的元素
groupingBy	根据给定的分组函数的值进行分组，输出一个Map对象
partitioningBy	根据给定的分区函数的值进行分区，输出一个Map对象，且key始终为布尔值类型
collectingAndThen	包裹另一个收集器，对其结果进行二次加工转换
reducing	从给定的初始值开始，将元素进行逐个的处理，最终将所有元素计算为最终的1个值输出

这里着重讲解 collectAndThen。collectAndThen 收集器必须先传入一个 downstream 收集器，再传入一个 finisher 方法。downstream 计算完成后，对 downstream 使用 finisher 抽取出我们真正想要的结果。



给定集团所有员工列表，找出上海公司中工资最高的员工。

```

1  /**
2  * 给定集团所有员工列表，找出上海公司中工资最高的员工。
3  */
4  private static Employee findShanghaiHighestSalaryEmployee() {
5      return employeeCollection().stream().filter(e -> "上海公
司".equals(e.getSubCompany())).collect(
6          Collectors.collectingAndThen(
7
    Collectors.maxBy(Comparator.comparingInt(Employee::getSalary)),
8
        Optional::orElseThrow
9
    )
10
);
11
}

```

## 自定义收集器

上述例子中我们使用的都是 `Collectors` 工具类提供的收集器。但是有时，我们需要定制化的场景，现有收集器无法满足我们的诉求，此时可以自己实现自定义收集器。

### Collector 接口

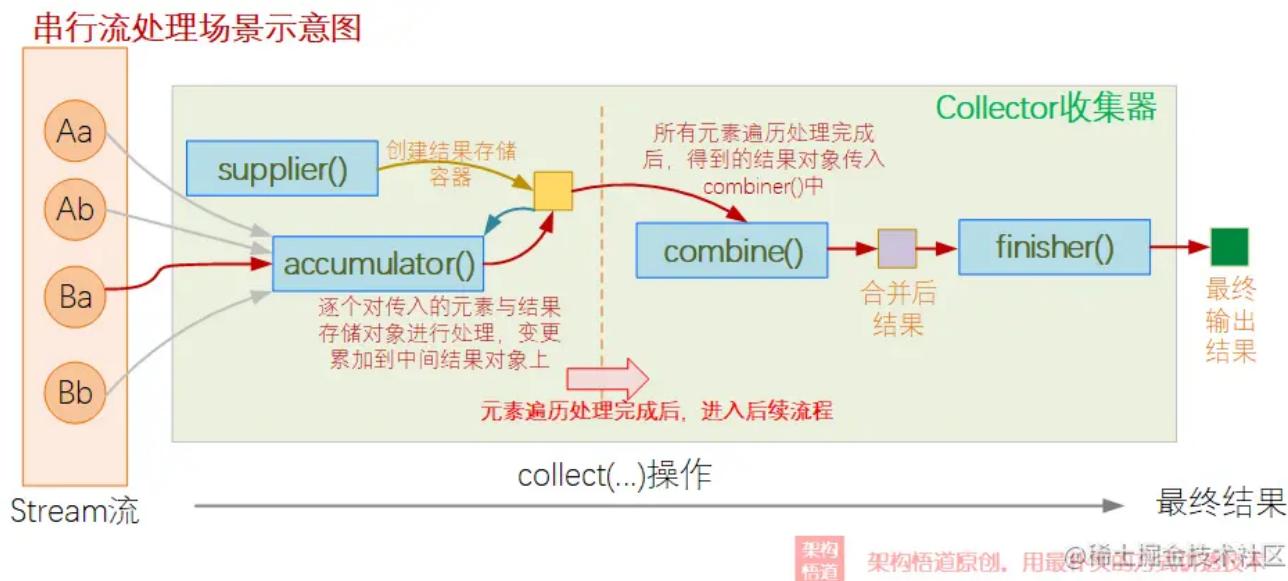
收集器是 `Collector` 接口的实现类。如果需要定制收集器，则需要先深入了解 `Collector` 接口。`Collector` 有 5 个接口：

接口名称	功能含义说明
supplier	创建新的结果容器，可以是一个容器，也可以是一个累加器实例，总之是用来存储结果数据的
accumulator	元素进入收集器中的具体处理操作
finisher	当所有元素都处理完成后，在返回结果前的对结果的最终处理操作，当然也可以选择不做任何处理，直接返回
combiner	各个子流的处理结果最终如何合并到一起去，比如并行流处理场景，元素会被切分为好多个分片进行并行处理，最终各个分片的数据需要合并为一个整体结果，即通过此方法来指定子结果的合并逻辑
characteristics	对此收集器处理行为的补充描述，比如此收集器是否允许并行流中处理，是否 <code>finisher</code> 方法必须要有等等，此处返回一个 <code>Set</code> 集合，里面的候选值是固定的几个可选项。

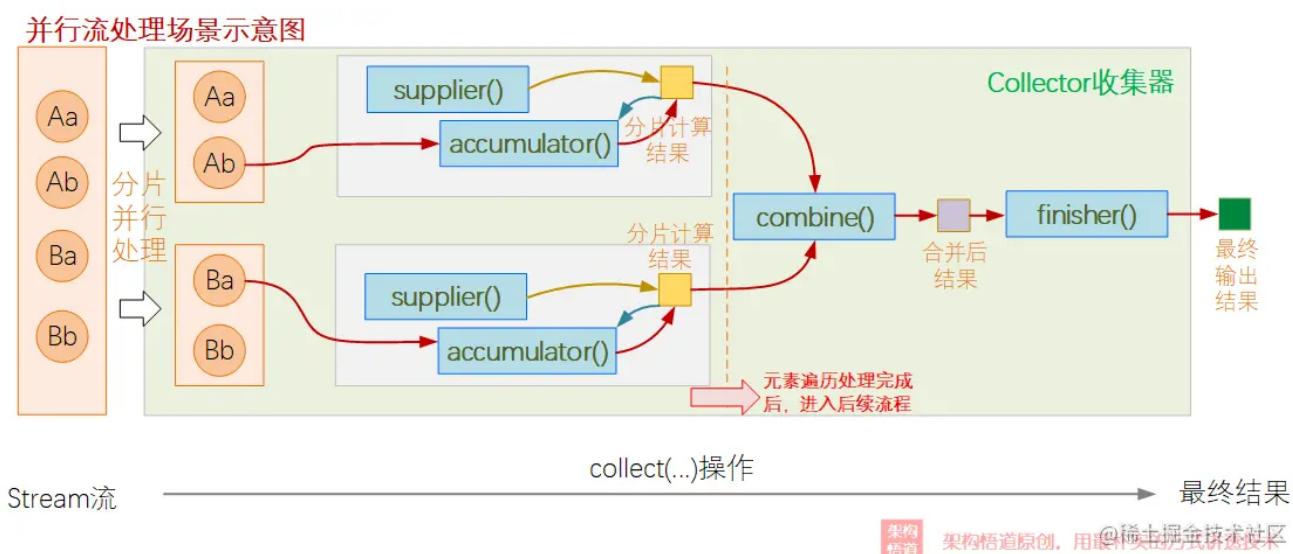
于 `characteristics` 的可选项，说明如下

取值	含义说明
UNORDERED	声明此收集器的汇总归约结果与Stream流元素遍历顺序无关，不受元素处理顺序影响
CONCURRENT	声明此收集器可以多个线程并行处理，允许并行流中进行处理
IDENTITY_FINISH	声明此收集器的finisher方法是一个恒等操作，可以跳过

那么，Collector 是如何互相配合协作的呢？



在并行场景下，先分片再多线程分片处理，最后合并



以 Collectors.toList() 为例，

```

1  /**
2   * Returns a {@code Collector} that accumulates the input elements
3   * into a
4   * new {@code List}. There are no guarantees on the type, mutability,

```

```

4  * serializability, or thread-safety of the {@code List} returned; if
5  * more
6  * control over the returned {@code List} is required, use {@link
7  #toCollection(Supplier)}.
8  *
9  * @param <T> the type of the input elements
10 * @return a {@code Collector} which collects all the input elements
11 into a
12 * {@code List}, in encounter order
13 */
14 public static <T>
15 Collector<T, ?, List<T>> toList() {
16     return new CollectorImpl<>((Supplier<List<T>>) ArrayList::new,
17         List::add,
18         (left, right) -> { left.addAll(right);
19         return left; },
20         CH_ID);
21 }

```

- supplier: ArrayList::new, 即创建一个 ArrayList 作为结果存储容器
- accumulator: List::add, 对每个流中的元素作为方法入参执行 List::add 方法添加到结果容器
- combiner: (left, right) -> { left.addAll(right); return left; }。合并并行操作生成的各个子 ArrayList, 最后通过 left.addAll(right) 合并为最终结果
- finisher: 未提供, 使用默认值恒等操作
- characteristics: 返回 IDENTITY\_FINISH, 也即最终结果直接返回, 不使用 finisher 二次加工。注意没有声明 CONCURRENT, 所以不支持并发。

## 实现 Collector 接口

现有需求: 计算流中每个元素的某个 Integer 字段值平方的总和, 支持并发, 使用收集器实现。

首先, Collector 接口是一个泛型接口

```

1 public interface Collector<T, A, R> {
2     // interface body...
3 }

```

T 表示输入的类型, A 表示存储累加容器的类型, R 表示结果类型。

输入类型为 Integer, 存储累加容器类型为 AtomicInteger (支持并发), 结果类型为 Integer。

## supplier

创建一个结果存储累加的容器。既然我们要计算多个值的累加结果，那首先创建一个线程安全的 AtomicInteger 存储一个累加结果。

```
1  @Override
2  public Supplier<AtomicInteger> supplier() {
3      return () -> new AtomicInteger(0);
4 }
```

## accumulator

实现具体的积累计算逻辑，整个 Collector 的核心业务逻辑所在。累加遍历值的平方到累加容器。

```
1  @Override
2  public BiConsumer<AtomicInteger, Integer> accumulator() {
3      return (acc, current) -> {
4          acc.addAndGet(current * current);
5      };
6 }
7 }
```

## combiner

并行流将 Stream 切片，再对分片进行合并，combiner 描述两个分片如何合并。

```
1  @Override
2  public BinaryOperator<AtomicInteger> combiner() {
3      return (firstSum, secondSum) -> {
4          firstSum.addAndGet(secondSum.get());
5          return firstSum;
6      };
7 }
```

## finisher

将存储累加容器转换为结果。使用 AtomicInteger::get 得到 Integer 类型的结果

```
1  @Override
2  public Function<AtomicInteger, Integer> finisher() {
3      return AtomicInteger::get;
4 }
```

## characteristics

说明 Collector 收集器的一些特性：

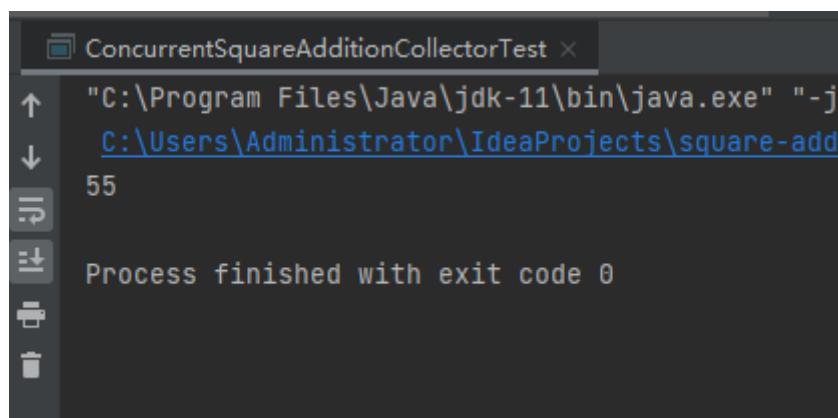
1. 允许并行使用，声明 CONCURRENT 属性
2. 元素先后计算顺序无关，声明 UNORDERED 属性

3. finisher 方法对存储累加容器做了转换处理，并非恒等处理操作，所以不能声明 IDENTITY\_FINISH

```
1  @Override
2  public Set<Characteristics> characteristics() {
3      return Collections.unmodifiableSet(
4          EnumSet.of(
5              Characteristics.CONCURRENT,
6              Characteristics.UNORDERED
7          )
8      );
9  }
```

## 测试

```
1 package com.congee02.collector;
2
3 import java.util.List;
4
5 class ConcurrentSquareAdditionCollectorTest {
6
7     public static void main(String[] args) {
8         System.out.println(List.of(1, 2, 3, 4, 5).stream().collect(new
9             ConcurrentSquareAdditionCollector()));
10    }
11 }
```



多线程

# 基本概念

## 进程

进程是对运行时程序的封装，是系统进行资源调配和分配的基本单位，实现了操作系统的并发

## 线程

是进程的子任务，是 CPU 调度和分派的基本单位，实现了进程内部的并发

## 进程和线程

1. 线程在进程中进行
2. 进程之间不会互相影响，主线程结束，则整个进程结束
3. 不同的进程数据很难共享
4. 同进程下的不同线程之间数据很容易共享
5. 进程使用内存地址可以限定使用量

## 线程的创建

### 继承 Thread 类，重写 run() 方法

```
1  private static class CounterExtendThread extends Thread {  
2  
3      public CounterExtendThread(String name) {  
4          super(name);  
5      }  
6  
7      @Override  
8      public void run() {  
9          for (int i = 0 ; i < 10; i ++ ) {  
10              System.out.println(Thread.currentThread().getName() + " :  
11                  " + i);  
12          }  
13      }  
14  }
```

## Thread 列表

```
1  private static List<Thread> threads() {  
2      CounterExtendThread firstCounter =  
3          new CounterExtendThread("firstCounter");  
4      CounterExtendThread secondCounter =  
5          new CounterExtendThread("secondCounter");  
6      CounterExtendThread thirdCounter =  
7          new CounterExtendThread("thirdCounter");  
8  }
```

```
9     return List.of(
10         firstCounter,
11         secondCounter,
12         thirdCounter
13     );
14 }
```

## 调用 Thread::run ThreadUtils#runThreads(List<Thread>)

```
1 public static void runThreads(List<Thread> threads) {
2     threads.forEach(Thread::run);
3 }
```

## 调用 Thread::start ThreadUtils#startThreads(List<Thread>)

```
1 public static void startThreads(List<Thread> threads) {
2     threads.forEach(Thread::start);
3 }
```

## 运行，观察区别

```
1 public static void main(String[] args) {
2     System.out.println("===== Thread::run =====");
3     ThreadUtils.runThreads(threads());
4     System.out.println("===== Thread::start =====");
5     ThreadUtils.startThreads(threads());
6 }
7 }
```

## 测试结果

```
1 ===== Thread::run =====
2 main : 0
3 main : 1
4 main : 2
5 main : 3
6 main : 4
7 main : 5
8 main : 6
9 main : 7
10 main : 8
11 main : 9
12 main : 0
13 main : 1
14 main : 2
15 main : 3
```

```
16 main : 4
17 main : 5
18 main : 6
19 main : 7
20 main : 8
21 main : 9
22 main : 0
23 main : 1
24 main : 2
25 main : 3
26 main : 4
27 main : 5
28 main : 6
29 main : 7
30 main : 8
31 main : 9
32 ===== Thread::start =====
33 firstCounter : 0
34 firstCounter : 1
35 firstCounter : 2
36 secondCounter : 0
37 secondCounter : 1
38 firstCounter : 3
39 secondCounter : 2
40 thirdCounter : 0
41 secondCounter : 3
42 firstCounter : 4
43 secondCounter : 4
44 thirdCounter : 1
45 secondCounter : 5
46 firstCounter : 5
47 firstCounter : 6
48 firstCounter : 7
49 firstCounter : 8
50 firstCounter : 9
51 secondCounter : 6
52 thirdCounter : 2
53 thirdCounter : 3
54 thirdCounter : 4
55 thirdCounter : 5
56 thirdCounter : 6
57 thirdCounter : 7
58 thirdCounter : 8
59 secondCounter : 7
60 thirdCounter : 9
61 secondCounter : 8
62 secondCounter : 9
```

## Thread::start 和 Thread::run 的区别

### Thread::run

1. run() 方法定义在 Runnable 接口的实现类，描述线程的执行逻辑
2. 调用 run() 不会创建新的线程，而是在当前线程中直接执行 run() 方法。这会让代码逐个执行而不会真正实现并发
3. run() 方法在当前线程执行，无法充分利用多核处理器来实现并行计算

### Thread::start

1. start() 方法定义在 Thread 类。通过继承 Thread 类并重写 run() 方法，可以创建一个新的进程，并在新线程上执行 run() 方法的代码
2. 调用 start() 方法会启动一个新的线程，并在该线程中执行 run() 方法。这样可以实现真正的并发执行，充分利用多核处理器
3. 注意，一个线程对象的 start() 方法只能调用一次，如果尝试多次调用，将会抛出 IllegalThreadStateException 异常

## 总结

1. 执行 run() 方法会在当前线程中顺序执行，没有真正的并发
2. 使用 start() 方法会创建一个新线程，在新线程中并发执行 run() 方法的代码

通常情况下，推荐使用实现 Runnable 接口的方式来创建线程，因为它更灵活，允许将同一个 Runnable 实例传递给多个线程，从而实现更好的资源利用。使用 Thread 类限制了线程的继承关系，不够灵活

## 实现 Runnable 类，实现 run() 方法

```
1  private static class CounterImplementRunnable implements Runnable {  
2  
3      @Override  
4      public void run() {  
5          for (int i = 0 ; i < 10 ; i++) {  
6              try {  
7                  Thread.sleep(20);  
8              } catch (InterruptedException e) {  
9                  e.printStackTrace();  
10             }  
11             System.out.println(Thread.currentThread().getName() + " :  
12             " + i);  
13         }  
14     }  
15 }
```

实例化 Thread 时，使用 Thread(Runnable, String) 构造器

```
1 private static List<Thread> threads() {
2     return List.of(
3             new Thread(new CounterImplementRunnable(), "firstCounter"),
4             new Thread(new CounterImplementRunnable(),
5 "secondCounter"),
6             new Thread(new CounterImplementRunnable(), "thirdCounter")
7     );
}
```

测试

```
1 public static void main(String[] args) {
2     System.out.println("===== Thread::run =====");
3     ThreadUtils.runThreads(threads());
4     System.out.println("===== Thread::start =====");
5     ThreadUtils.startThreads(threads());
6 }
```

结果

```
1 ===== Thread::run =====
2 main : 0
3 main : 1
4 main : 2
5 main : 3
6 main : 4
7 main : 5
8 main : 6
9 main : 7
10 main : 8
11 main : 9
12 main : 0
13 main : 1
14 main : 2
15 main : 3
16 main : 4
17 main : 5
18 main : 6
19 main : 7
20 main : 8
21 main : 9
22 main : 0
23 main : 1
24 main : 2
25 main : 3
26 main : 4
27 main : 5
28 main : 6
29 main : 7
```

```
30 main : 8
31 main : 9
32 ===== Thread::start =====
33 firstCounter : 0
34 secondCounter : 0
35 firstCounter : 1
36 secondCounter : 1
37 firstCounter : 2
38 firstCounter : 3
39 firstCounter : 4
40 firstCounter : 5
41 secondCounter : 2
42 secondCounter : 3
43 secondCounter : 4
44 secondCounter : 5
45 firstCounter : 6
46 secondCounter : 6
47 thirdCounter : 0
48 secondCounter : 7
49 firstCounter : 7
50 secondCounter : 8
51 thirdCounter : 1
52 secondCounter : 9
53 firstCounter : 8
54 thirdCounter : 2
55 firstCounter : 9
56 thirdCounter : 3
57 thirdCounter : 4
58 thirdCounter : 5
59 thirdCounter : 6
60 thirdCounter : 7
61 thirdCounter : 8
62 thirdCounter : 9
```

## 实现 Callable 接口，重写 run() 方法，支持 FutureTask

```
1 package com.congee02.multithread.create;
2
3 import java.util.concurrent.Callable;
4 import java.util.concurrent.ExecutionException;
5 import java.util.concurrent.FutureTask;
6
7 public class CreateSquareCalculatorThreadByImplementCallable {
8
9     private static class SquareCalculator implements Callable<Integer>
10    {
```

```
11  private int number;
12
13  public SquareCalculator(int number) {
14      this.number = number;
15  }
16
17  @Override
18  public Integer call() throws Exception {
19      Thread.sleep(1000L);
20      return number * number;
21  }
22 }
23
24 public static void main(String[] args) {
25     FutureTask<Integer> task = new FutureTask<>(new
26     SquareCalculator(20));
27     new Thread(task).start();
28     try {
29         Integer result = task.get();
30         System.out.println(result);
31     } catch (InterruptedException | ExecutionException e) {
32         e.printStackTrace();
33     }
34 }
35 }
36 }
```

## 控制当前线程

线程控制有三个常见的方法 sleep(long), join(), setDaemon(boolean)

### **sleep(long)**

当前线程暂停指定毫秒数，进入休眠状态

```
1 package com.congee02.multithread.control;
2
3 public class ThreadSleep {
4
5     public static void main(String[] args) {
6         try {
7             Thread.sleep(200);
8         } catch (InterruptedException e) {
9             e.printStackTrace();
10        }
11    }
12 }
13 }
```

## join()

执行后，当前线程执行完毕后，后续线程才能得到 CPU 的执行权。

```
1 package com.congee02.multithread.control;
2
3 import java.util.Random;
4
5 public class ThreadJoin {
6
7     public static void main(String[] args) {
8         Random random = new Random();
9         Runnable count = () -> {
10             for (int i = 0; i < 5; i++) {
11                 try {
12                     Thread.sleep(random.nextInt(10));
13                 } catch (InterruptedException e) {
14                     e.printStackTrace();
15                 }
16                 System.out.println(Thread.currentThread().getName() +
17                         " : " + i);
18             }
19         };
20
21         Thread joinThread = new Thread(count, "Join Thread");
22         joinThread.start();
23         try {
24             joinThread.join();
25         } catch (InterruptedException e) {
26             e.printStackTrace();
27         }
28         System.out.println("===== Join Thread End =====");
```

```
29
30     Thread t1 = new Thread(count, "Normal Thread " + 1);
31     Thread t2 = new Thread(count, "Normal Thread " + 2);
32     Thread t3 = new Thread(count, "Normal Thread " + 3);
33
34     t1.start();
35     t2.start();
36     t3.start();
37 }
38
39 }
40
```

执行结果

```
1 Join Thread : 0
2 Join Thread : 1
3 Join Thread : 2
4 Join Thread : 3
5 Join Thread : 4
6 ===== Join Thread End =====
7 Normal Thread 3 : 0
8 Normal Thread 1 : 0
9 Normal Thread 1 : 1
10 Normal Thread 2 : 0
11 Normal Thread 2 : 1
12 Normal Thread 3 : 1
13 Normal Thread 1 : 2
14 Normal Thread 1 : 3
15 Normal Thread 1 : 4
16 Normal Thread 2 : 2
17 Normal Thread 3 : 2
18 Normal Thread 2 : 3
19 Normal Thread 2 : 4
20 Normal Thread 3 : 3
21 Normal Thread 3 : 4
```

## setDaemon(boolean)

将线程标记为守护线程，用来服务其他的线程。Java 中的垃圾回收线程，就是典型的守护线程。

守护线程主要用于系统服务、定期任务和周期性操作、资源管理和监控、垃圾回收、后台日志记录、时间处理、自动化任务等等。

```
1 package com.congee02.multithread.control;
2
```

```

3  public class ThreadSetDaemon {
4
5      public static void main(String[] args) {
6          Thread loggerDaemon = loggerDaemon();
7          loggerDaemon.start();
8
9          try {
10              Thread.sleep(10000);
11          } catch (InterruptedException e) {
12              e.printStackTrace();
13          }
14
15      }
16
17      private static Thread loggerDaemon() {
18          Thread thread = new Thread(logger());
19          thread.setDaemon(true);
20          return thread;
21      }
22
23      private static Runnable logger() {
24          return () -> {
25              try {
26                  int count = 1;
27                  while (true) {
28                      System.out.println("Logging entry " + count++);
29                      Thread.sleep(1000);
30                  }
31              } catch (InterruptedException e) {
32                  e.printStackTrace();
33              }
34          };
35      }
36
37  }
38

```

## 与普通线程相比，守护线程有如下特点

### 1. 终止行为

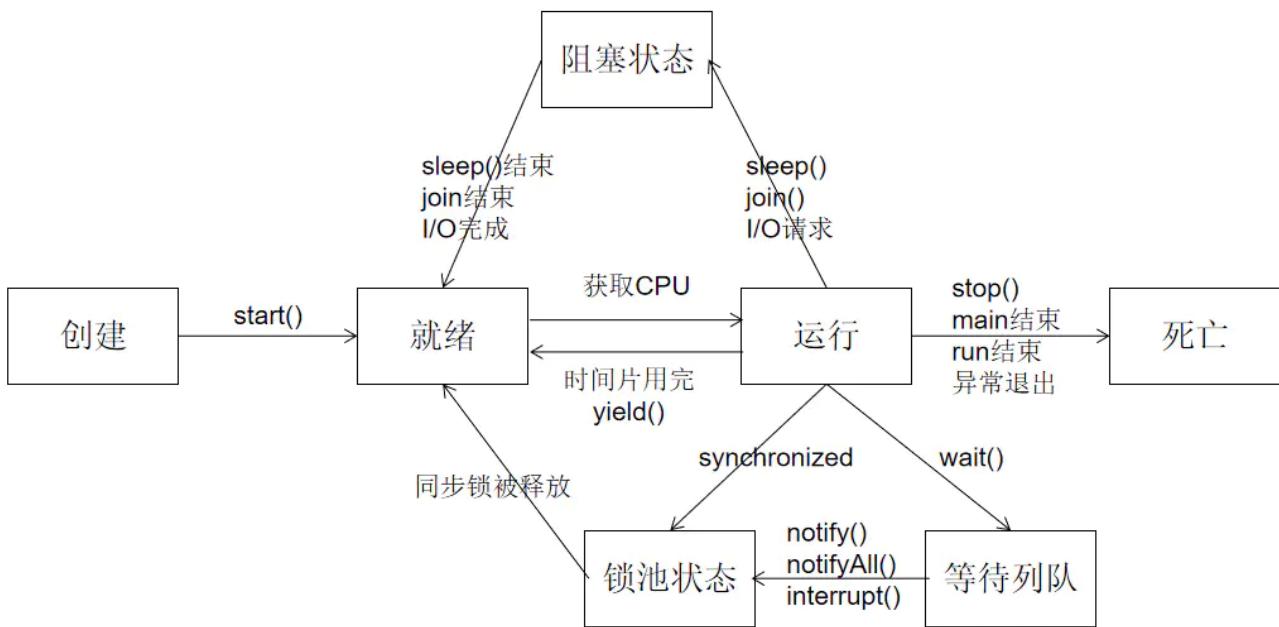
- 普通线程：普通线程的终止不会影响程序的继续执行，即使所有的普通进程已经停止，程序仍然会继续运行，知道所有线程执行完成。
- 守护线程：守护线程的终止会随着程序的终止而终止，即使守护线程的任务尚未完成。当所有的非守护线程都已经终止时，守护线程会被强制终止。

### 2. 执行任务

- 普通线程：普通线程会阻止程序的继续执行，直到它们完成任务
- 守护线程：守护线程在后台默默执行，不阻塞主程序的执行

总的来说，守护线程和普通线程之间的最大区别在于它们的终止行为和对主程序终止的影响。

## 线程的生命周期



线程生命周期简图

## 获取线程执行结果

在上述三种创建线程的方式（继承Thread、实现Runnable、实现Callable）中。前两种方式若要获取执行结果，必须通过共享变量或者线程通信的方式来达到目的，使用起来比较麻烦。

Java 提供了 Callable、Future、FutureTask，允许在任务执行后获取执行结果。

### Callable 和 Runnable

Runnable::run() 方法返回值为 void，所以执行完任务后无法返回任何结果。

```
1  @FunctionalInterface
2  public interface Runnable {
3      /**
4      * When an object implementing interface <code>Runnable</code> is
5      * used
6      * to create a thread, starting the thread causes the object's
7      * <code>run</code> method to be called in that separately
8      * executing
9      * thread.
10     * <p>
11     * The general contract of the method <code>run</code> is that it
12     * may
13     * take any action whatsoever.
14     *
```

```
12     * @see      java.lang.Thread#run()
13     */
14     public abstract void run();
15 }
16
```

Callable::call() 方法返回的是一个泛型类，有返回结果。

```
1  @FunctionalInterface
2  public interface Callable<V> {
3      /**
4      * Computes a result, or throws an exception if unable to do so.
5      *
6      * @return computed result
7      * @throws Exception if unable to compute a result
8      */
9      V call() throws Exception;
10 }
```

当我们需要使用 Callable 获取线程执行的结果时，我们需要配合 ExecutorService 和 Future 来使用。

```
1 package com.congee02.multithread.reuslt;
2
3 import java.util.ArrayList;
4 import java.util.concurrent.*;
5
6 public class CallableGetExecuteResult {
7
8     private final static int TASK_NUM = 10;
9
10    public static void main(String[] args) {
11
12        // 创建一个包含五个线程的线程池
13        ExecutorService executorService =
14            Executors.newFixedThreadPool(5);
15
16        // 创建一个 Callable 任务
17        Callable<String> task = () -> "Hello from " +
18        Thread.currentThread().getName();
19
20        // 提交任务到 ExecutorService 对象中执行，获取 Future 对象
21        ArrayList<Future<String>> futures = new ArrayList<>(TASK_NUM);
22        for (int i = 0 ; i < TASK_NUM ; i ++ ) {
23            futures.add(executorService.submit(task));
24        }
25    }
26}
```

```
24 // 通过 Future 获取任务结果
25 futures.stream().map(future -> {
26     try {
27         return future.get();
28     } catch (InterruptedException | ExecutionException e) {
29         e.printStackTrace();
30     }
31     return null;
32 }).forEach(System.out::println);
33
34 // 关闭 ExecutorService 对象，不再接受新的任务，等待所有已提交的任务完
成
35     executorService.shutdown();
36 }
37 }
38
```

## ExecutorService

ExecutorService 是 Java 并发库中的一个接口，提供了管理和控制线程池的功能，使得在多线程下可以更加方便地管理任务的执行。通过使用 ExecutorService，允许将任务提交给线程池，并由线程池管理线程的创建、执行、复用以及资源的管理。

常用的 ExecutorService 实现类

### 1. Executors.newFixedThreadPool(int nThreads)

创建一个固定大小的线程池，池中包含指定数量的线程。

### 2. Executors.newCachedThreadPool()

创建一个缓存线程池，线程数量根据需要自动调整。适用于任务数不确定且执行时间较短的场景

### 3. Executors.newSingleThreadExecutor()

创建一个单线程的线程池，适用于需要保证任务按顺序执行的情况。

### 4. Executors.newScheduledThreadPool(int corePoolSize)

## Future

Future 是 Java 并发库中的一个接口，它表示一个异步运算的结果。可以在提交任务后获取任务的执行的结果、取消任务执行、查询任务状态等，有五个方法：

### 1. boolean cancel(boolean mayInterruptIfRunning)：尝试取消任务的执行。

mayInterruptIfRunning 参数表示是否允许中断正在执行的任务。

### 2. boolean isCancelled()：检查任务是否已被取消。

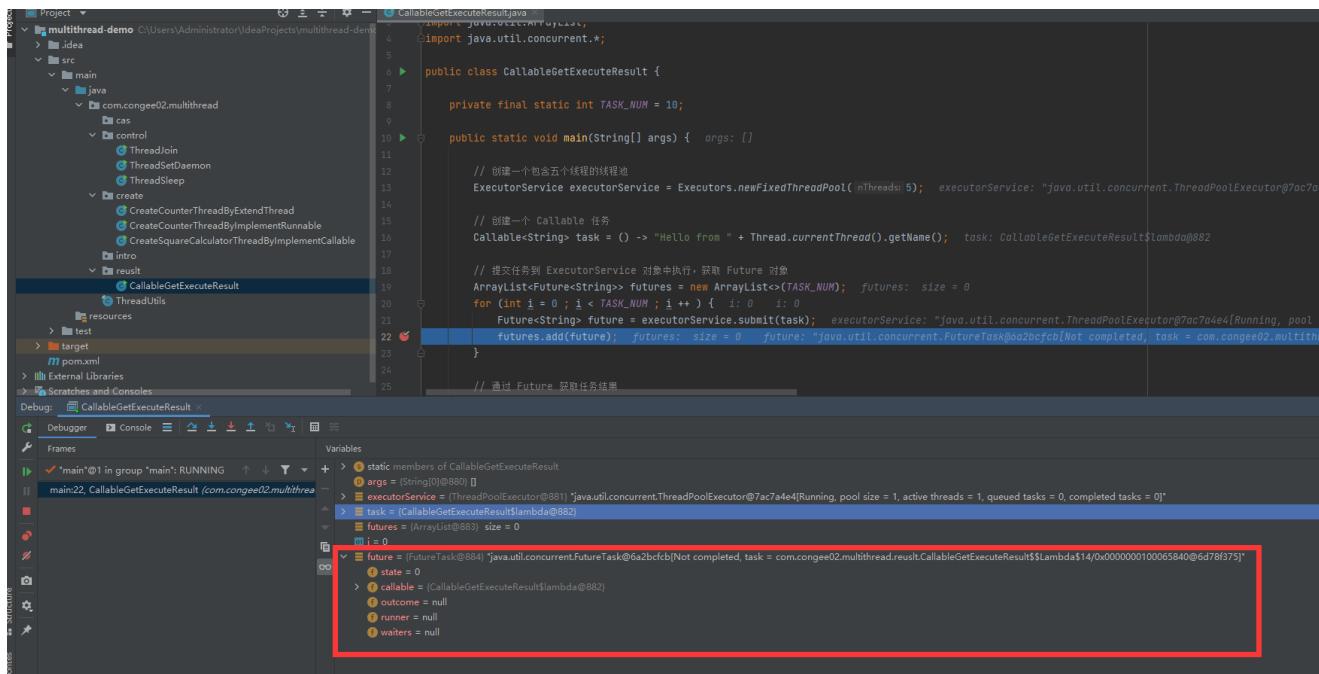
### 3. boolean isDone()：检查任务是否已经完成（不管是正常完成、取消还是抛出异常）。

4. `v get()`: 获取任务的执行结果, 如果任务未完成, 则阻塞等待直到任务完成。

5. `v get(long timeout, TimeUnit unit)`: 获取任务的执行结果, 但最多等待指定的时间, 如果在指定时间内任务仍未完成, 则抛出 `TimeoutException`。

## FutureTask

FutureTask 是 Future 接口唯一的实现类。在前面的例子中, `executorService.submit(task)` 返回的就是 FutureTask 类型的对象。



FutureTask 实现的接口

```
1 public class FutureTask<V> implements RunnableFuture<V> {  
2     // class body ...  
3 }
```

再看 RunnableFuture

```
1 public interface RunnableFuture<V> extends Runnable, Future<V> {  
2     /**  
3      * Sets this Future to the result of its computation  
4      * unless it has been cancelled.  
5      */  
6     void run();  
7 }
```

RunnableFuture 继承了 Runnable 和 Future, 所以既可以作为 Runnable 被线程执行, 又可以作为 Future 得到 Callable 的返回值

又一个示例

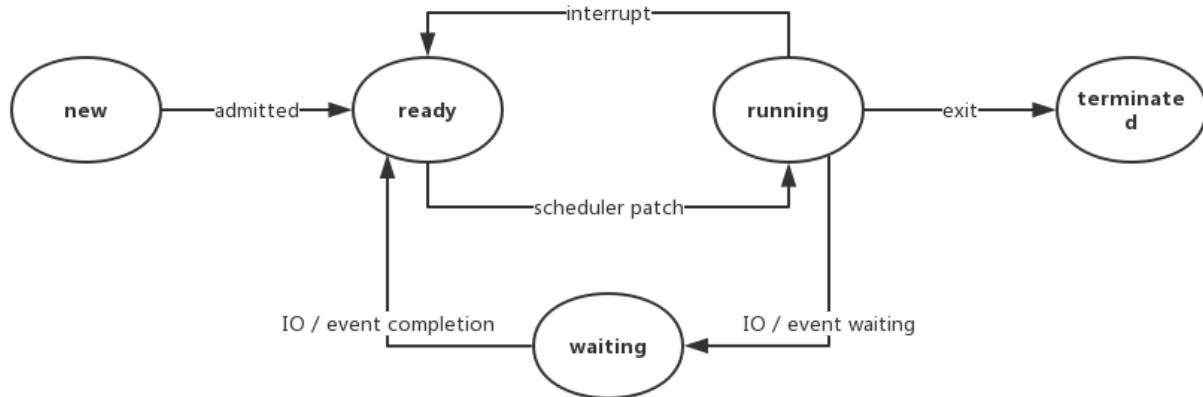
```
1 package com.congee02.multithread.reuslt;  
2
```

```
3  import java.util.ArrayList;
4  import java.util.concurrent.*;
5
6  public class CallableGetExecuteComputationResult {
7
8      private static final int CALLABLE_TASK_NUM = 5;
9
10     public static void main(String[] args) {
11
12         // 固定大小的线程池
13         ExecutorService executorService =
14             Executors.newFixedThreadPool(3);
15
16         // 创建 FutureTask 任务
17         ArrayList<FutureTask<Integer>> tasks = new ArrayList<>(
18             CALLABLE_TASK_NUM);
19         for (int i = 0 ; i < CALLABLE_TASK_NUM ; i++) {
20             int index = i;
21             tasks.add(new FutureTask<>(() -> {
22                 TimeUnit.SECONDS.sleep(index);
23                 return (index + 1) * 100;
24             }));
25         }
26
27         // 提交 FutureTask 任务到 executorService
28         tasks.forEach(executorService::submit);
29
30         // 打印任务结果
31         tasks.stream().map(task -> {
32             try {
33                 return task.get();
34             } catch (InterruptedException | ExecutionException e) {
35                 e.printStackTrace();
36             }
37             return null;
38         }).forEach(System.out::println);
39
40     }
41
42 }
43
```

# Java 线程的六种状态及其切换

## 操作系统线程

在操作系统中，线程被看作是轻量级的进程，所以操作系统的线程状态和操作系统的进程状态是一致的。



操作系统的线程主要有以下三个状态：

- 就绪状态  
线程正在等待使用 CPU，经调度程序之后进入 running 状态
- 执行状态  
线程正在使用 CPU
- 等待状态  
线程经过等待事件的调用或者等待其他资源（比如 I/O）

## Java 线程

现在来看 Java 线程的 6 个状态

```
1 public enum State {  
2     /**  
3      * Thread state for a thread which has not yet started.  
4      */  
5     NEW,  
6  
7     /**  
8      * Thread state for a runnable thread. A thread in the runnable  
9      * state is executing in the Java virtual machine but it may  
10     * be waiting for other resources from the operating system  
11     * such as processor.  
12     */  
13     RUNNABLE,  
14  
15     /**
```

```
16     * Thread state for a thread blocked waiting for a monitor lock.
17     * A thread in the blocked state is waiting for a monitor lock
18     * to enter a synchronized block/method or
19     * reenter a synchronized block/method after calling
20     * {@link Object#wait()} Object.wait}.
21     */
22 BLOCKED,
23
24 /**
25     * Thread state for a waiting thread.
26     * A thread is in the waiting state due to calling one of the
27     * following methods:
28     * <ul>
29     *   <li>{@link Object#wait()} Object.wait} with no timeout</li>
30     *   <li>{@link #join()} Thread.join} with no timeout</li>
31     *   <li>{@link LockSupport#park()} LockSupport.park}</li>
32     * </ul>
33     *
34     * <p>A thread in the waiting state is waiting for another thread
35     * to
36     * perform a particular action.
37     *
38     * For example, a thread that has called {@code Object.wait()}
39     * on an object is waiting for another thread to call
40     * {@code Object.notify()} or {@code Object.notifyAll()} on
41     * that object. A thread that has called {@code Thread.join()}
42     * is waiting for a specified thread to terminate.
43     */
44 WAITING,
45
46 /**
47     * Thread state for a waiting thread with a specified waiting
48     * time.
49     * A thread is in the timed waiting state due to calling one of
50     * the following methods with a specified positive waiting time:
51     * <ul>
52     *   <li>{@link #sleep Thread.sleep}</li>
53     *   <li>{@link Object#wait(long) Object.wait} with timeout</li>
54     *   <li>{@link #join(long) Thread.join} with timeout</li>
55     *   <li>{@link LockSupport#parkNanos LockSupport.parkNanos}</li>
56     *   <li>{@link LockSupport#parkUntil LockSupport.parkUntil}</li>
57     * </ul>
58     */
59 TIMED_WAITING,
60
61 /**
62     * Thread state for a terminated thread.
63     * The thread has completed execution.
64     */
```

```
63     TERMINATED;  
64 }
```

## NEW

处于 NEW 状态的线程此时尚未启动。这里的“尚未启动”指线程对象尚未调用 start 方法

```
1 package com.congee02.multithread.state;  
2  
3 public class ThreadNewState {  
4  
5     public static void main(String[] args) {  
6         // 创建线程, 但尚未启动 (start 方法), 此时线程处于 NEW 状态  
7         Thread thread = new Thread(() -> {});  
8         System.out.println(thread.getState());  
9     }  
10 }  
11 }  
12 }
```

上述线程创建但未启动线程，处于 NEW 状态

## 能否反复调用一个线程的 start 方法

要回答这个问题，需要查看 Thread 的 start 方法源码

```
1  /**  
2  * Causes this thread to begin execution; the Java Virtual Machine  
3  * calls the {@code run} method of this thread.  
4  * <p>  
5  * The result is that two threads are running concurrently: the  
6  * current thread (which returns from the call to the  
7  * {@code start} method) and the other thread (which executes its  
8  * {@code run} method).  
9  * <p>  
10 * It is never legal to start a thread more than once.  
11 * In particular, a thread may not be restarted once it has completed  
12 * execution.  
13 *  
14 * @throws     IllegalThreadStateException  if the thread was already  
15 * started.  
16 * @see       #run()  
17 * @see       #stop()  
18 */  
19 public synchronized void start() {  
20     /**  
21      * This method is not invoked for the main method thread or  
22      * "system"  
23      * group threads created/set up by the VM. Any new functionality  
24      * added
```

```

22     * to this method in the future may have to also be added to the
23     * VM.
24     *
25     * A zero status value corresponds to state "NEW".
26     */
27     if (threadStatus != 0)
28         throw new IllegalThreadStateException();
29
30     /* Notify the group that this thread is about to be started
31      * so that it can be added to the group's list of threads
32      * and the group's unstarted count can be decremented. */
33     group.add(this);
34
35     boolean started = false;
36     try {
37         start0();
38         started = true;
39     } finally {
40         try {
41             if (!started) {
42                 group.threadStartFailed(this);
43             }
44         } catch (Throwable ignore) {
45             /* do nothing. If start0 threw a Throwable then
46              * it will be passed up the call stack */
47         }
48     }

```

关键在于最开始的 if 条件判断

```

1 if (threadStatus != 0)
2     throw new IllegalThreadStateException();

```

其中 0 指代 Thread 的 NEW 状态。那么，如果当前线程对象的状态不是 NEW，也无法重现转变到 NEW 状态，此时尝试调用 start() 时，会抛出 IllegalThreadStateException。

当线程刚创建再第一次启动（调用 start() 方法）后，状态不再为 NEW，那么再次调用时，会抛出 IllegalThreadStateException 异常。故曰：线程只能启动一次。

## RUNNABLE

表示当前线程正在运行中。处于 RUNNABLE 状态的线程在 Java 虚拟机中运行，也有可能在等待 CPU 分配资源。

需要注意，Java 线程的 RUNNABLE 包括了操作系统线程的 ready 和 running 两个状态。

## BLOCKED

阻塞状态。处于 BLOCKED 状态的线程正等待锁的释放以进入同步区。

```
1 package com.congee02.multithread.state;
2
3 public class ThreadBlockedState {
4
5     // 锁 同步区
6     private final static Object resource = new Object();
7     private final static Runnable getLockAndRun = () -> {
8         // 尝试得到锁。若锁被占用，等待锁被释放，在此期间线程等待资源，状态为
9         // BLOCK
10        synchronized (resource) {
11            System.out.println(Thread.currentThread().getName() + ":" +
12                "Holding the lock...");
13            try {
14                Thread.sleep(10000);
15            } catch (InterruptedException e) {
16                e.printStackTrace();
17            }
18            System.out.println(Thread.currentThread().getName() + ":" +
19                "Release the lock...");
20        }
21    }
22
23    public static void main(String[] args) throws InterruptedException {
24        Thread t1 = new Thread(getLockAndRun, "Thread1");
25        t1.start();
26        Thread t2 = new Thread(getLockAndRun, "Thread2");
27        t2.start();
28        while (t2.getState() != Thread.State.TERMINATED) {
29            System.out.println("Thread2 state: " + t2.getState());
30            Thread.sleep(1000); // 增加适当的延迟
31        }
32        System.out.println(t2.getState());
33    }
34}
```

## WAITING

等待状态。处于等待状态的线程转为 RUNNABLE 状态需要其他线程唤醒。

调用三个方法会使得线程进入等待状态：

- Object#wait()

使当前线程处于等待状态，直到被另一个线程唤醒

生产者-消费者问题

```
1 package com.congee02.multithread.wait;
2
3 public class ProducerConsumer {
4
5     private static final Object lock = new Object();
6     private static Runnable produce = () -> {
7         synchronized (lock) {
8             System.out.println("Producer: Producing data...");
9             try {
10                 lock.wait();
11             } catch (InterruptedException e) {
12                 e.printStackTrace();
13             }
14             System.out.println("Producer: Resumed");
15         }
16     };
17
18     private static Runnable consume = () -> {
19         synchronized (lock) {
20             System.out.println("Consumer: Waiting for data...");
21             try {
22                 Thread.sleep(2000);
23             } catch (InterruptedException e) {
24                 e.printStackTrace();
25             }
26             lock.notify();
27         }
28     };
29
30     public static void main(String[] args) {
31         Thread producer = new Thread(produce);
32         Thread consumer = new Thread(consume);
33
34         producer.start();
35         consumer.start();
36
37         try {
38             producer.join();
39             consumer.join();
40         }
```

```
40     } catch (InterruptedException e) {
41         e.printStackTrace();
42     }
43 }
44 }
45 }
46 }
47 }
```

- Thread#join()
- 等待线程执行完毕，底层是调用 Object 实例的 wait 方法
- LockSupport#park()

## **TIMED\_WAITING**

超时等待状态。线程等待一个具体的时间，时间到后会被自动唤醒。

调用下列方法会使得线程进入超时等待状态：

- Thread#sleep(long millis)
- Object#wait(long timeout)
- Thread#join(long millis)
- LockSupport#parkNanos(long nanos)
- LockSupport#parkUntil(long deadline)

## **TIMED\_WAITING 和 WAITING 的区别**

### 1. TIMED\_WAITING (计时等待)

当线程使用 Thread#sleep(long millis) 方法、Object#wait(long timeout) 方法或者其他类似带有超时参数的等待方法且参数为正整数时，线程会进入 TIMED\_WAITING 状态；当其参数为0时，依然进入到 WAITING 状态。这表示线程在等待一段时间后会自动恢复到 RUNNABLE 状态。例如，Thread.sleep(1000) 会使当前进程进入到 TIMED\_WAITING 状态，持续 1 秒后，恢复到 RUNNABLE 状态。

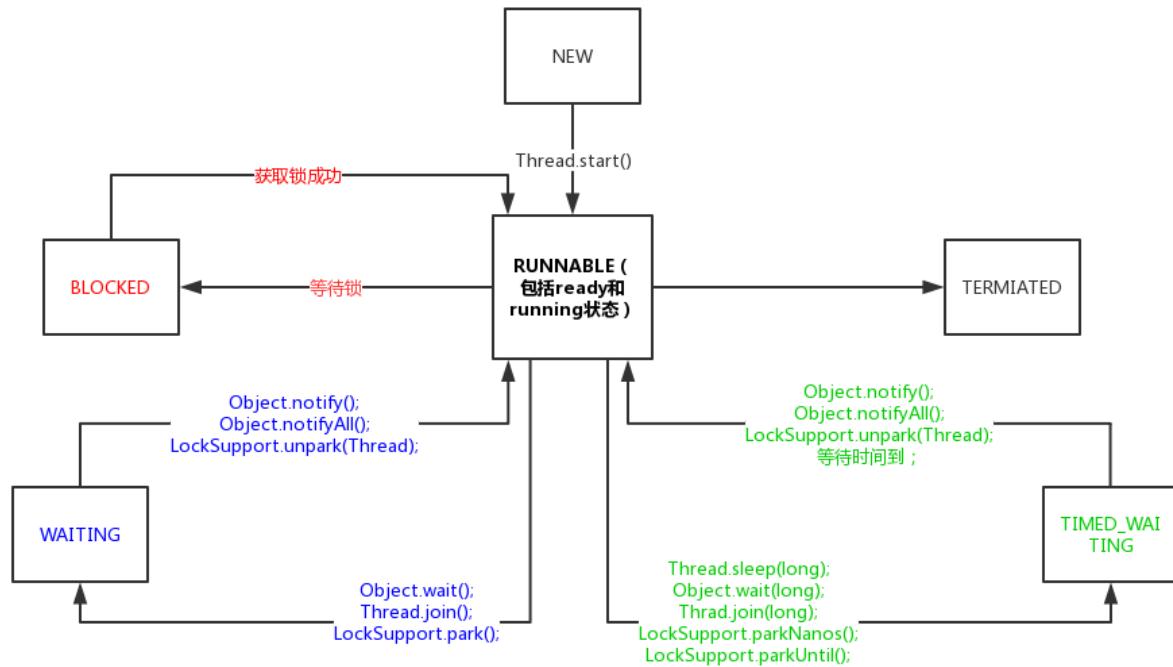
### 2. WAITING (无限等待)

当线程使用 Object#wait() 方法，Thread#join() 等方法，线程进入 WAITING 状态。在此状态下，线程会一直等待某个条件满足，直到其他线程显示地唤醒它。

## **TERMINATED**

终止状态，此时线程已经执行完毕。

# Java 线程 状态转换



## BLOCKED 与 RUNNABLE 状态的转换

处于 BLOCKED 状态的线程是因为在等待锁的释放。有两个线程 t1 和 t2, t1 提前获得了锁并且暂未释放, 此时 t2 处于 BLOCKED 状态。

```
1 package com.congee02.multithread.state;
2
3 public class ThreadBlockedState {
4
5     // 锁 同步区
6     private final static Object resource = new Object();
7     private final static Runnable getLockAndRun = () -> {
8         // 尝试得到锁。若锁被占用，等待锁被释放，在此期间线程等待资源，状态为
9         // BLOCK
10        synchronized (resource) {
11            System.out.println(Thread.currentThread().getName() + ":
12                Holding the lock...");
13            try {
14                Thread.sleep(10000);
15            } catch (InterruptedException e) {
16                e.printStackTrace();
17            }
18            System.out.println(Thread.currentThread().getName() + ":
19                Release the lock...");
20        }
21    };
22}
```

```
21     public static void main(String[] args) throws InterruptedException
22     {
23         Thread t1 = new Thread(getLockAndRun, "Thread1");
24         t1.start();
25         Thread t2 = new Thread(getLockAndRun, "Thread2");
26         t2.start();
27         while (t2.getState() != Thread.State.TERMINATED) {
28             Thread.sleep(1000); // 增加适当的延迟
29             System.out.println("Thread2 state: " + t2.getState());
30         }
31     }
32
33 }
34
```

结果

```
1 Thread1: Holding the lock...
2 Thread2 state: BLOCKED
3 Thread2 state: BLOCKED
4 Thread2 state: BLOCKED
5 Thread2 state: BLOCKED
6 Thread2 state: BLOCKED
7 Thread2 state: BLOCKED
8 Thread2 state: BLOCKED
9 Thread2 state: BLOCKED
10 Thread2 state: BLOCKED
11 Thread1: Release the lock...
12 Thread2: Holding the lock...
13 Thread2 state: TIMED_WAITING
14 Thread2 state: TIMED_WAITING
15 Thread2 state: TIMED_WAITING
16 Thread2 state: TIMED_WAITING
17 Thread2 state: TIMED_WAITING
18 Thread2 state: TIMED_WAITING
19 Thread2 state: TIMED_WAITING
20 Thread2 state: TIMED_WAITING
21 Thread2 state: TIMED_WAITING
22 Thread2 state: TIMED_WAITING
23 Thread2: Release the lock...
24 Thread2 state: TERMINATED
25 TERMINATED
```

## WAITING 与 RUNNABLE 的转换

有三种方法将线程的状态从 RUNNABLE 转换为 WAITING 状态，下面主要介绍 Object#wait() 和 Thread#join()。

### 1. Object#wait()

一个线程调用锁的 wait() 方法前，线程必须持有该对象的锁。

线程使用 wait() 方法时，会释放当前的锁，直到有其他线程调用 notify() / notifyAll() 方法唤醒。

简单的 Producer-Consumer 问题

```
1 package com.congee02.multithread.wait;
2
3
4 public class ProducerConsumer {
5
6     // 锁
7     private static final Object lock = new Object();
8
9     // 生产者逻辑
10    private static Runnable produce = () -> {
11        // 尝试获取锁
12        synchronized (lock) {
13            System.out.println("Producer: Producing data...");
14            try {
15                // 生产者已经生产数据，等待消费者消费。
16                // 暂时释放锁，等待消费者使用 lock.notify() 唤醒
17                lock.wait();
18            } catch (InterruptedException e) {
19                e.printStackTrace();
20            }
21            System.out.println("Producer: Resumed");
22        }
23    };
24
25     // 消费者逻辑
26    private static Runnable consume = () -> {
27        // 尝试获取锁。
28        // 若生产者还在生产，即生产者还持有锁，则等待生产者生产数据然后暂时
29        // 释放锁
30        synchronized (lock) {
31            System.out.println("Consumer: Waiting for data...");
32            try {
33                Thread.sleep(2000);
34            } catch (InterruptedException e) {
35                e.printStackTrace();
36            }
37            // 告知生产者可以重新获取锁并继续运行
38        }
39    };
40 }
```

```
37         lock.notify();
38     }
39 }
40
41 public static void main(String[] args) {
42     Thread producer = new Thread(produce);
43     Thread consumer = new Thread(consume);
44
45     producer.start();
46     consumer.start();
47
48     try {
49         producer.join();
50         consumer.join();
51     } catch (InterruptedException e) {
52         e.printStackTrace();
53     }
54
55 }
56
57 }
58 }
```

## 2. Thread#join()

调用某个线程对象 `join()` 方法，其他线程会一直等待这个线程执行完毕后执行自己的逻辑。那么其他线程处于 WAITING 状态。

## 线程中断

在某些情况下，线程启动后发现其不再需要执行时，需要中断线程。目前在 Java 里还没有安全直接的方法来停止线程，但是提供了线程中断机制来处理需要中断线程的情况。

线程中断是一种协作机制。需要注意，通过中断操作不能直接终止一个线程，而是通知需要被终端的线程自行处理。

简要介绍 `Thread` 类中线程中断的方法：

- `void interrupt()`

该方法用于中断线程。当调用这个方法时，设置其中断状态为 `true`。

- `boolean isInterrupt()`

查询线程的中断状态，返回当前线程的中断状态，但不会清除中断。

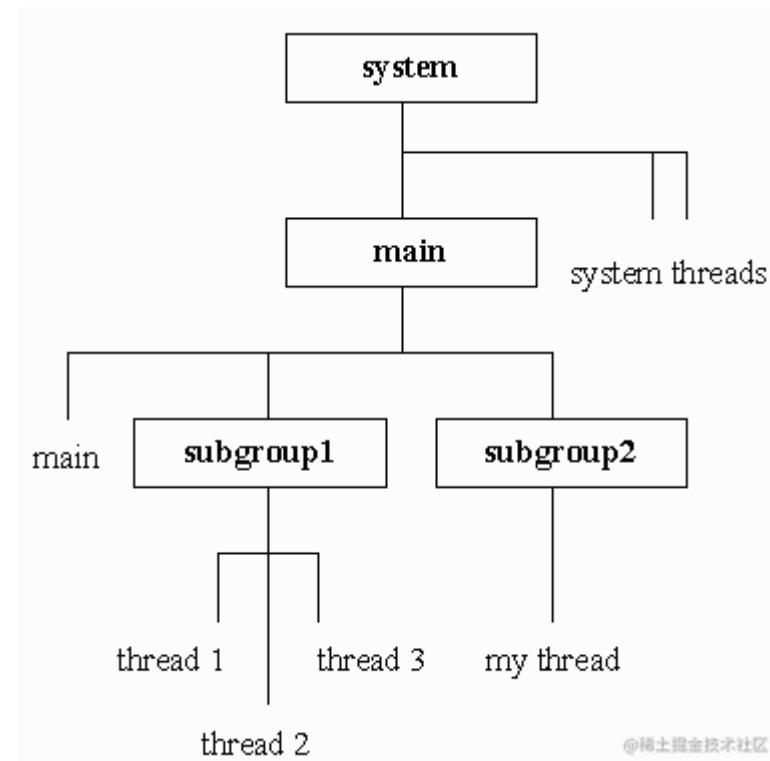
- `static boolean interrupt()`

查询当前线程的中断状态，并且清楚中断状态。如果当前中断状态为 `true`，则状态被清除并返回 `true`；若中断状态为 `false`，返回 `false`。

## 线程组 ThreadGroup

线程组简单来说就是一个线程集合和其他线程组集合。线程组的出现是为了更方便地管理线程。

线程组时父子结构的，一个线程组可以集成其他线程组，同时也可以拥有其他子线程组。从结构上来看，线程组是一个树形结构，每个线程都隶属于一个线程组，线程组又有父线程组，这样追溯下去，可以追溯到一个根线程组——System 线程组。



下面介绍一下线程组树的结构

1. JVM 创建的 system 线程组是用来处理 JVM 系统任务的线程组，比如都西昂的销毁等。
2. system 线程组的直接子线程组是 main 线程组，这个线程组至少包括一个 main 线程，用于执行 main 方法。
3. main 线程组的子线程组就是应用程序创建的线程组

线程组 ThreadLocal 部分源码：

```
1 public
2 class ThreadGroup implements Thread.UncaughtExceptionHandler {
3     // 父线程组
4     private final ThreadGroup parent;
5
6     // ThreadGroup 的名称
7     String name;
8
9     // 线程最大优先级
10    int maxPriority;
```

```
11 // 是否已经被销毁
12 boolean destroyed;
13
14 // 是否是守护进程
15 boolean daemon;
16
17 // 还未启动的进程数量
18 int nUnstartedThreads = 0;
19
20 // 线程总数
21 int nthreads;
22
23 // 存储线程
24 Thread threads[];
25
26 // 线程组数量
27 int ngroups;
28 // 存储子线程组
29 ThreadGroup groups[];
30
31 // 其他成员方法和成员属性 ... ...
32
33 }
```

可以在main方法中看到JVM创建的system线程组和main线程组:

```
1 package com.congee02.multithread.group;
2
3 public class MainThreadGroupInsight {
4
5     public static void main(String[] args) {
6         // 获取当前线程的线程组
7         ThreadGroup mainThreadGroup =
8             Thread.currentThread().getThreadGroup();
9
10        // 获取当前线程的线程组的父线程组
11        ThreadGroup mainThreadGroupParent =
12            mainThreadGroup.getParent();
13
14        // 当前线程为 main 线程, 隶属于 main 线程组
15        System.out.println("mainThreadGroup: " + mainThreadGroup);
16
17        // main 线程组的父线程组为 system
18        System.out.println("mainThreadGroupParent: " +
19            mainThreadGroupParent);
20    }
21
22 }
```

## 运行结果

```
1 mainThreadGroup: java.lang.ThreadGroup[name=main,maxpri=10]
2 mainThreadGroupParent: java.lang.ThreadGroup[name=system,maxpri=10]
```

一个线程可以访问其所属线程组的信息，但不能访问其线程组的父线程组或其他线程组的信息，也就是说线程组是一个向下引用的树状结构，这样设计是为了防止下级获取上级的引用而无法被 GC 回收。

## 线程组的构造器

ThreadGroup 提供了两个构造函数

Constructor	Description
ThreadGroup(String name)	根据线程组名称创建线程组，其父线程组为当前线程的线程组
ThreadGroup(ThreadGroup parent, String name)	根据线程组名称创建线程组，其父线程组为指定的parent线程组

下面演示这两个构造器的使用

```
1 package com.congee02.multithread.group;
2
3 public class ThreadGroupCreate {
4
5     public static void main(String[] args) {
6
7         // 若未指定，默认的父线程组为 main
8         ThreadGroup subThreadGroup1 = new
9         ThreadGroup("subThreadGroup1");
10
11         // 指定默认父线程组为 subThreadGroup1
12         ThreadGroup subThreadGroup2 = new ThreadGroup(subThreadGroup1,
13         "subThreadGroup2");
14
15         System.out.println("Parent of subThreadGroup1: " +
16         subThreadGroup1.getParent());
17         System.out.println("Parent of subThreadGroup2: " +
18         subThreadGroup2.getParent());
19
20     }
21
22 }
```

```
1 | Parent of subThreadGroup1: java.lang.ThreadGroup[name=main,maxpri=10]
2 | Parent of subThreadGroup2:
  | java.lang.ThreadGroup[name=subThreadGroup1,maxpri=10]
```

## 线程组的常用方法

### 1. 获取当前线程组名字

```
ThreadGroup#getName()
```

### 2. 从线程组中提取活动线程数组

```
1 | package com.congee02.multithread.group;
2 |
3 | import java.util.Arrays;
4 | import java.util.Random;
5 |
6 | public class ExtractThreadsFromThreadGroup {
7 |
8 |     private final static Random random = new Random();
9 |     private final static Runnable helloRunnable = () -> {
10 |         try {
11 |             Thread.sleep(random.nextInt(100));
12 |         } catch (InterruptedException e) {
13 |             e.printStackTrace();
14 |         }
15 |         System.out.println("Hello from " +
16 | Thread.currentThread().getName() + ".");
17 |     };
18 |
19 |     private final static int THREAD_NUM = 10;
20 |
21 |     public static void main(String[] args) {
22 |         ThreadGroup threadGroup = new
23 | ThreadGroup("subThreadGroup");
24 |         for (int i = 0 ; i < THREAD_NUM ; i ++ ) {
25 |             new Thread(threadGroup, helloRunnable).start();
26 |         }
27 |         Thread[] extractActiveThreads = new
28 | Thread[threadGroup.activeCount()];
29 |         threadGroup.enumerate(extractActiveThreads);
30 |
31 |         System.out.println(Arrays.toString(extractActiveThreads));
32 |     }
33 |
34 | }
```

### 3. 线程组统一异常处理

```

1 package com.congee02.multithread.group;
2
3 public class ThreadGroupUnifiedExceptionManagement {
4
5     public static void main(String[] args) {
6         ThreadGroup group
7             = new ThreadGroup("Print-Uncaught-Exception-
8 ThreadGroup");
9         @Override
10        // 重写处理异常的方法
11        public void uncaughtException(Thread t, Throwable e)
12        {
13            System.out.println(t.getName() + " throws a
14 exception: " + e.getMessage());
15        }
16    };
17
18    Thread thread = new Thread(group, () -> {
19        // 抛出一个异常
20        throw new RuntimeException("This is a testing
Exception.");
21    }, "Throw-Exception-Thread");
22
23    thread.start();
24}

```

## 线程优先级

线程的优先级级别由操作系统决定，不同的操作系统级别是不一样的，在 Java 中，提供了一个级别范围 1~10 以方便参考。Java 默认的优先级为 5，线程的执行顺序由调度程序来决定，线程的优先级会在线程调用之前设定。

Thread 中定义了三个静态方法来确定线程优先级

```

1 /**
2  * The minimum priority that a thread can have.
3 */
4 public static final int MIN_PRIORITY = 1;
5
6 /**
7  * The default priority that is assigned to a thread.
8 */
9 public static final int NORM_PRIORITY = 5;
10
11 /**
12  * The maximum priority that a thread can have.

```

```
13  */
14  public static final int MAX_PRIORITY = 10;
```

## 获取线程优先级别 Thread#getPriority()

```
1 package com.congee02.multithread.priority;
2
3 public class ThreadPriorityInsight {
4
5     public static void main(String[] args) {
6         new Thread(() -> {
7             Thread currentThread = Thread.currentThread();
8             System.out.println("Priority level of " +
9 currentThread.getName() + " is " + currentThread.getPriority());
10        }, "Default-Priority-Thread").start();
11    }
12 }
```

## 设置线程优先级别 Thread#setPriority()

```
1 package com.congee02.multithread.priority;
2
3 public class ThreadPrioritySet {
4
5     private static final Runnable getPriorityRunnable = () -> {
6         Thread thread = Thread.currentThread();
7         System.out.println("The priority of " + thread.getName() + " "
8 is " + thread.getPriority());
9     };
10
11     public static void main(String[] args) {
12         Thread highPriorityThread = new Thread(getPriorityRunnable,
13 "HighPriorityThread");
14         highPriorityThread.setPriority(Thread.NORM_PRIORITY + 3);
15
16         Thread normPriorityThread = new Thread(getPriorityRunnable,
17 "NormPriorityThread");
18         normPriorityThread.setPriority(Thread.NORM_PRIORITY);
19
20         Thread lowPriorityThread = new Thread(getPriorityRunnable,
21 "LowPriorityThread");
22         lowPriorityThread.setPriority(Thread.NORM_PRIORITY - 3);
23
24         lowPriorityThread.start();
25         normPriorityThread.start();
26         highPriorityThread.start();
27     }
28 }
```

```
25      }
26
27  }
28
```

需要注意，较高优先级的线程只是提高线程先执行的概率，真正的执行顺序由操作系统决定。

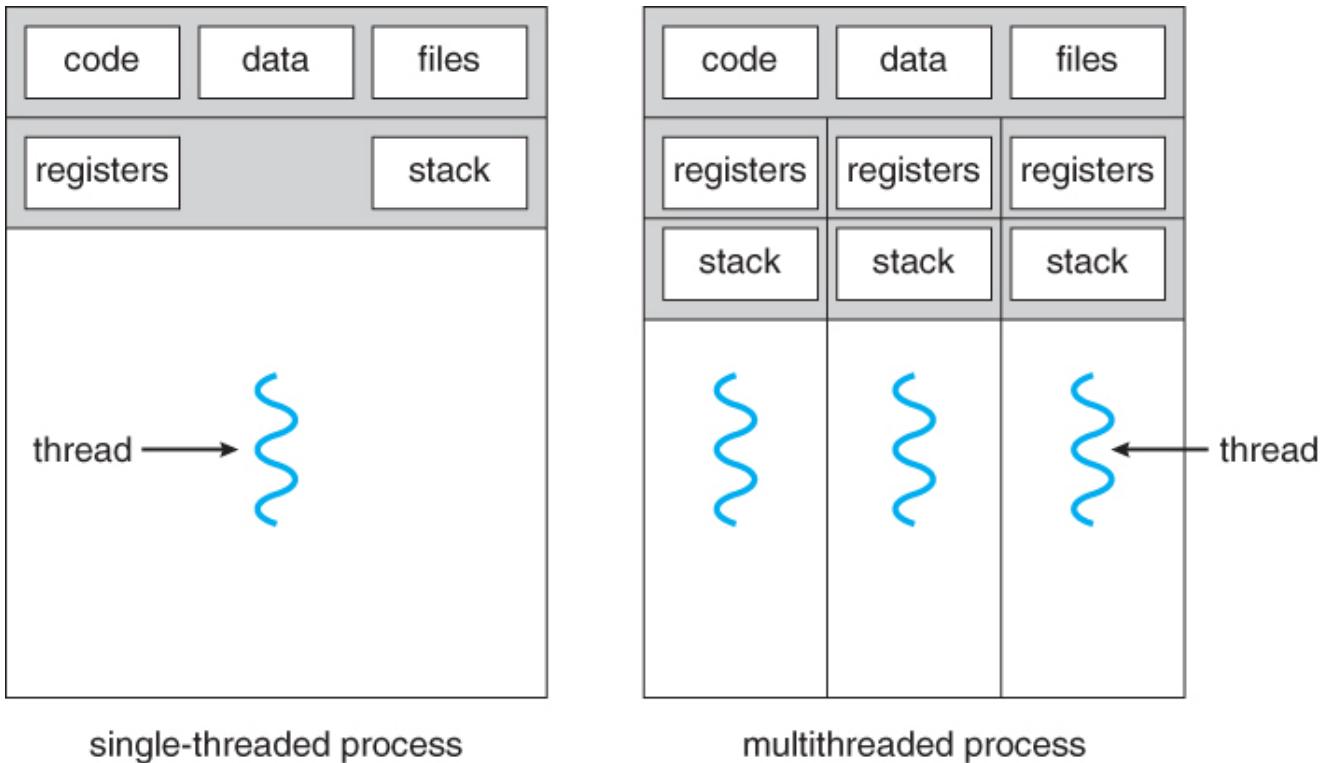
运行结果：

```
1 The priority of HighPriorityThread is 8
2 The priority of LowPriorityThread is 2
3 The priority of NormPriorityThread is 5
```

## 进程 Process VS 线程 Thread

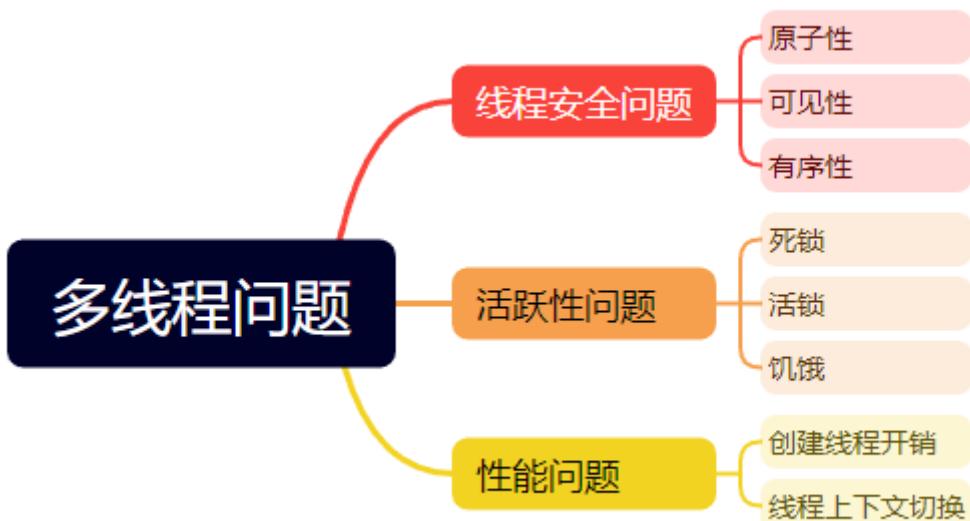
层面 Aspect	进程 Process	线程 Thread
定义 Definition	有自己独立内存和资源的程序	进程的最小单元
创建 Creation	高昂的创建和管理代价	轻量级的创建和管理
隔离 Isolation	进程之间互相隔离。一个进程崩溃不会直接影响到其他进程。	隶属于同一进程的线程共享相同的地址空间和资源，所以线程也可以影响到整个线程
通信 Communication	介于进程之间的隔离性，进程间的通信较为复杂。	线程共享相同的内存地址空间和资源，所以无需复杂的线程间通信就可以直接访问和修改共享的内存区域
上下文切换 Context Switching	进程的上下文切换代价更大，更耗时	线程的上下文切换代价更小，更块
资源额外开销 Resource Overhead	每个线程都有自己独立的内存空间和系统资源，额外开销较大	线程间共享资源，因此有更小的资源额外开销
容错度 Fault Tolerance	鉴于进程间的隔离性，有较大容错度	一个线程与其他线程共享计算机资源，且隶属于某个线程，因而容错度较小
可伸缩性 Scalability	鉴于管理进程的额外开销，可伸缩性较差	鉴于其轻量，线程有更好的可伸缩性
并行性 Parallelism	线程可以在多核 CPU 上并行运行	线程可以在线程中并行执行

层面 Aspect	进程 Process	线程 Thread
同步 Synchronization	需要更加复杂的通信机制 (Inter-Process Communication, IPC)	同步更加容易实现 (共享内 存)
例子 Example	多个浏览器程序各自独立运行	一个浏览器使用多个线程来页 面渲染和网络传输等等



## 多线程引发的问题

多线程引发的问题被分为三大类：线程安全性问题、活跃性问题、性能问题。



## 线程安全问题

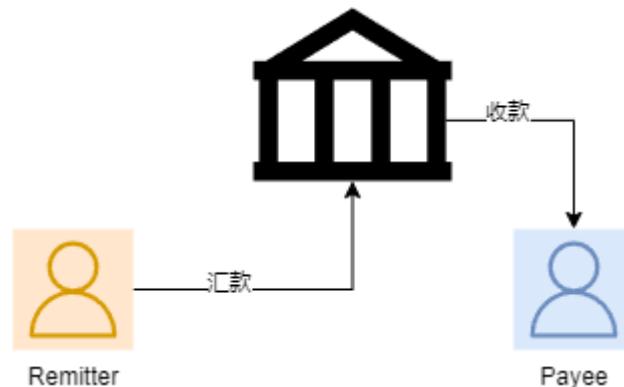
### 原子性

一个操作或者多个操作，要么全部执行并且执行的过程中不会被任何因素打断，要么不执行。

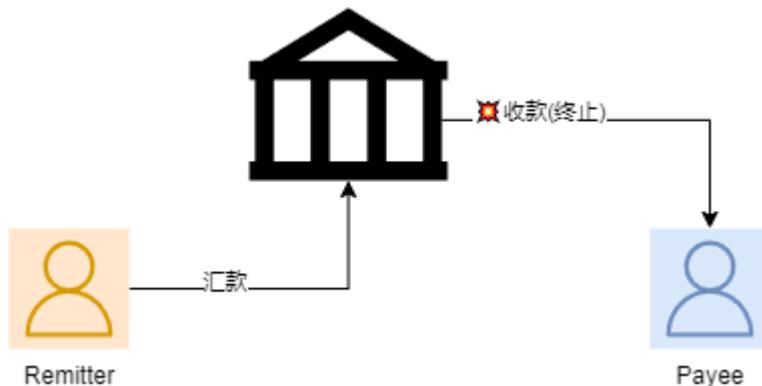
原子操作：不会被线程调度机制打断的操作，没有上下文切换

原子性典型的例子是银行转账问题。

银行转账分为两个操作：汇款和收款



如果汇款和收款两个操作不具有原子性。汇款操作成功，汇款人的账户减去金额给银行。但是此后因为一些原因，收款操作突然终止，收款人没有收到钱，这里就出现了问题。



在并发编程中，很多操作不是原子操作。

```
1 | i = 0;          // 0
2 | i ++;          // 1
3 | i = j;          // 2
4 | i = i + 1;    // 3
```

上述四个操作中，只有操作0是原子方法，其他操作都包含多个步骤：

- 操作0  
将字面量 0 直接赋值给 i
- 操作1  
读取 i 的值 -> 对 i 加 1
- 操作2

读取 j 的值 -> 将 j 的值赋值给 i

- 操作3

读取 i -> 计算 i + 1 -> 将 i + 1 复制给 i

在多线程的环境下，每个线程共享这些变量，如果不使用锁机制等手段进行控制，可能会得到意料之外的值。

在 Java 语言中，可以使用 synchronized 和 lock 来保证操作的原子性。

例子，多线程计数器：

```
1 package com.congee02.multithread.atomicity;
2
3 import java.util.Random;
4
5 public class SynchronizedAtomicityOperation {
6
7     /**
8      * 原子操作的计数器
9      */
10    private static class AtomicIncrementCounter {
11
12        // 创建一个锁对象
13        private final Object lock = new Object();
14
15        private int count = 0;
16
17        public void increment() {
18            synchronized (lock) {
19                count++;
20                System.out.println("Thread " +
21 Thread.currentThread().getName() + " executed increment()");
22            }
23        }
24
25        public int getCount() {
26            return count;
27        }
28
29        private static final Random random = new Random();
30
31        private static final AtomicIncrementCounter counter = new
32        AtomicIncrementCounter();
33        private static final Runnable incrementRunnable = () -> {
34            for (int i = 0; i < 100 ; i++) {
35                try {
36                    Thread.sleep(random.nextInt(10));
37                } catch (InterruptedException e) {
38
```

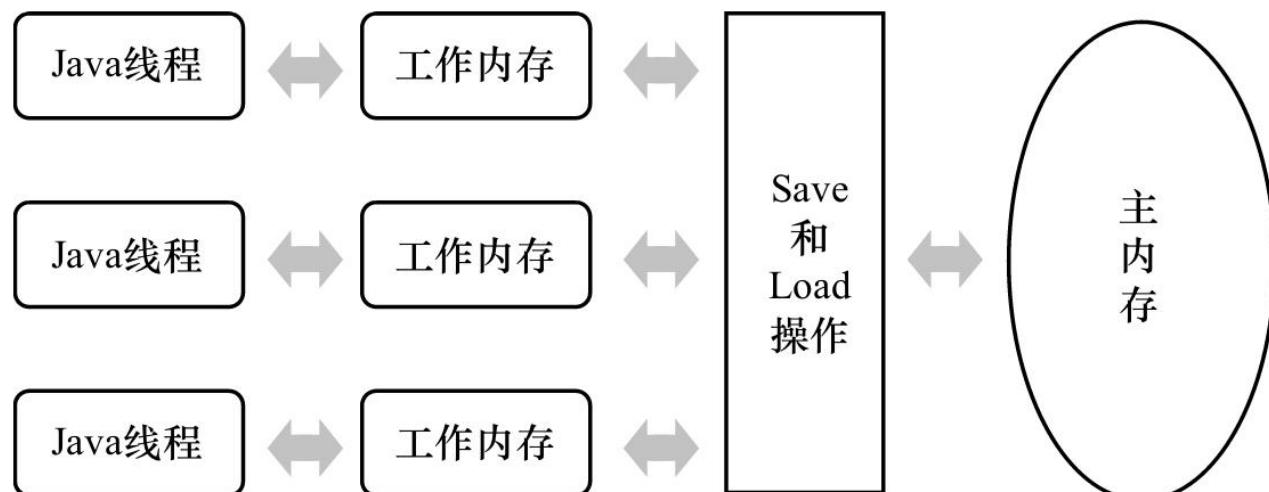
```

37         e.printStackTrace();
38     }
39     counter.increment();
40 }
41 }
42
43 public static void main(String[] args) {
44
45     Thread firstThread = new Thread(incrementRunnable,
46 "firstThread");
47     Thread secondThread = new Thread(incrementRunnable,
48 "secondThread");
49
50     firstThread.start();
51     secondThread.start();
52
53     // 等待线程执行完成后打印结果
54     try {
55         firstThread.join();
56         secondThread.join();
57     } catch (InterruptedException e) {
58         e.printStackTrace();
59     }
60
61 }
62 }
63

```

## 可见性

当多个线程访问同一变量时，一个线程改变了这个变量的值，其他线程立即能看到修改后的值。



如上图，每个线程都有自己的工作内存，工作内存和主内存的交互靠 store/load 操作。

鉴于可见性问题，Java 提供了 volatile 关键字。当一个变量被 volatile 修饰时：

- 写操作  
写入的值会立即被更新到主内存而非自己的工作内存
- 读操作  
去主内存中读取变量的值

## 有序性

有序性指程序按照代码的先后顺序执行。

在默认情况下，为了优化性能，程序中语句执行的先后顺序会被改变，称为指令重排。

例如

```
1 代码语句:  
2 a = 6;  
3 b = 7;  
4  
5 可能的乱序执行:  
6 b = 7;  
7 a = 6;
```

要具体介绍有序性，我们从设计模式中的懒加载单例模式开始。

在单线程的情况下，懒加载单例模式可以写做：

```
1 package com.congee02.multithread.ordered;  
2  
3 public final class NaiveLazyLoadSingleton {  
4  
5     private NaiveLazyLoadSingleton() {}  
6  
7     private static NaiveLazyLoadSingleton instance;  
8  
9     public static NaiveLazyLoadSingleton getInstance() {  
10         if (instance == null) {  
11             instance = new NaiveLazyLoadSingleton();  
12         }  
13         return instance;  
14     }  
15 }  
16 }
```

但是在多线程的情况下，这是不安全的。因为 "instance = new NaiveLazyLoadSingleton();" 不是一个原子操作，它分为这几个步骤：

1. 类加载

如果类没有被加载，Java类加载器加载类的字节码文件。

## 2. 分配内存

在堆内存中分配足够的内存空间来存储对象的实例变量。

## 3. 初始化实例变量

对象的实例变量会被设置为默认值，例如数字类型为 0，引用类型为 null

## 4. 执行构造函数

调用类的构造函数，初始化对象的实例变量，执行构造函数中的代码块，完成对象的初始化过程

## 5. 返回引用

构造函数执行完毕后，会返回一个指向新创建对象的引用。

在多线程情况下，上述实例化的一些步骤会被指令重排，比如：

### 1. 分配内存和初始化实例变量

在多线程情况下，可能会先分配内存，然后另一个线程就能够访问到未完全初始化的对象。这会导致其他线程可能会看到实例变量的默认值，而不是期望的初始化值。

### 2. 执行构造函数

在某些情况下，构造函数中的代码可能在对象分配内存之前就被执行

那么，首先将 `instance` 修饰为 `volatile`，当一个线程实例化 `instance` 时，另一个线程立即能在进程的主内存中观察到实例化过程中 `instance` 值的变化，以保障其可见性。若未被 `volatile` 修饰，当线程 A 实例化 `instance` 完成后，另一个线程可能没能见到完整的 `instance`，进而导致未知的错误。另外 `volatile` 禁止了对其修饰变量的相关操作的重排序：

- **写-读重排序**

当写操作在读操作之前，线程 A 的读操作在线程 B 的写操作进行期间执行，可能会导致线程 A 看到一个未完全初始化的值。

- **写-写重排序**

当两个写操作的顺序被调整后，可能导致某个线程的写操作被延迟到后面的位置。

- **读-写重排序**

当读操作在写操作之后，读操作可能会读取到旧的值。

其次，将 `getInstance()` 方法修饰为 `synchronized`，保证任意时刻至多只有一个线程调用这个方法，防止多个线程同时判断 `instance == null` 为 `true`，从而重复创建多个实例。

```
1 package com.congee02.multithread.ordered;
2
3 public final class NaiveSynchronizedLazyLoadSingleton {
4
5     private NaiveSynchronizedLazyLoadSingleton() {}
```

```

7     private static volatile NaiveSynchronizedLazyLoadSingleton
8         instance;
9
9     public synchronized static NaiveSynchronizedLazyLoadSingleton
10        getInstance() {
11         if (instance == null) {
12             instance = new NaiveSynchronizedLazyLoadSingleton();
13         }
14     return instance;
15
16 }
17

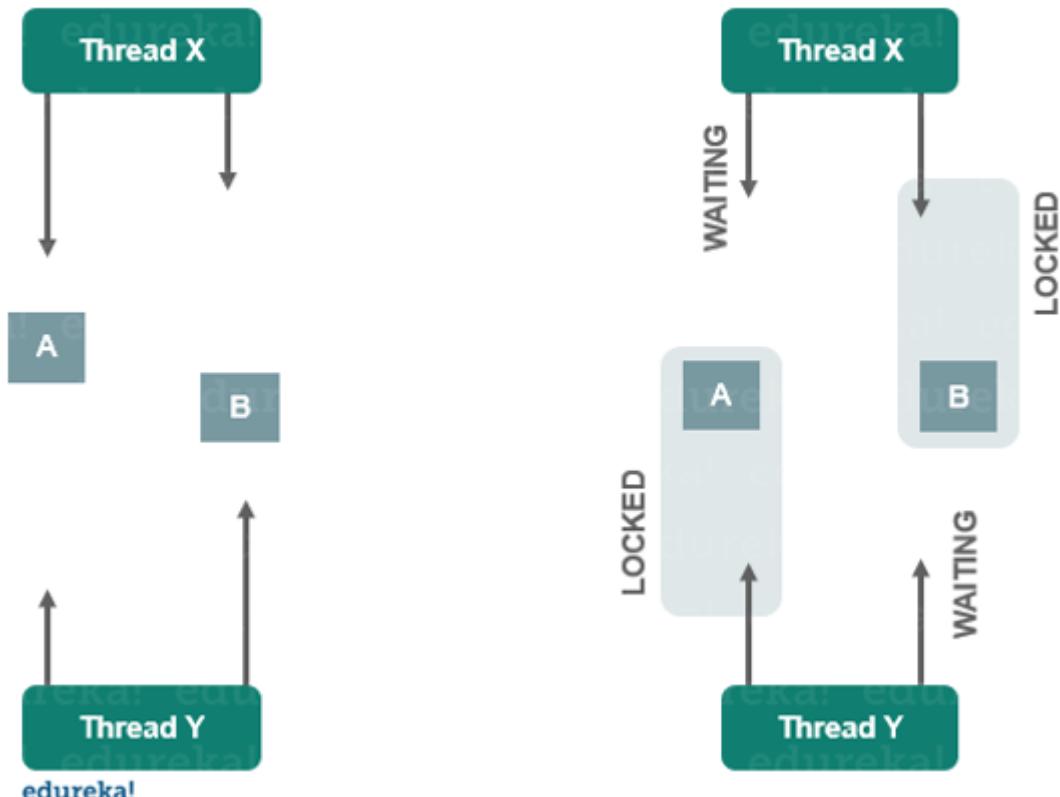
```

现在，这个类是线程安全的，但是我们发现 `synchronized` 只在 `getInstance()` 第一次被调用的时候有效，在 `getInstance()` 第一次被调用结束后，`instance` 已经被完全实例化，此时允许多个线程同时访问 `getInstance()` 方法且仍旧保持线程安全，但是其方法依然只能被至多一个线程访问，导致了性能的额外开销。可以使用双重检查锁来解决这个问题，在这里暂且不表。

## 活跃性问题

活跃性问题指的是某些线程无法继续执行的状态，导致系统无法正常前进。活跃性问题涵盖死锁、活锁和饥饿。

### 死锁



如图：

1. 线程 X 持有资源 B
2. 线程 Y 持有资源 A
3. 线程 X 想要资源 A, 但是被线程 Y 占用
4. 线程 Y 想要资源 B, 但是被线程 X 占用

在这种情况下, 线程 X 和线程 Y 互相等待对方释放资源, 形成了死锁。

线程 X 无法执行, 因为它需要资源 A, 而线程 Y 占用资源 A。

线程 Y 无法执行, 因为它需要资源 B, 而线程 X 占用资源 B。

Java 代码如下:

```
1 package com.congee02.multithread.liveness;
2
3 public class DeadLockDemo {
4
5     private final static Object resourceA = new Object() {
6         @Override
7         public String toString() {
8             return "resource A";
9         }
10    };
11    private final static Object resourceB = new Object() {
12        @Override
13        public String toString() {
14            return "resource B";
15        }
16    };
17
18    private static class TwoResourceRunnable implements Runnable {
19
20        private final Object firstResource;
21        private final Object secondResource;
22
23        public TwoResourceRunnable(Object firstResource, Object secondResource) {
24            this.firstResource = firstResource;
25            this.secondResource = secondResource;
26        }
27
28        @Override
29        public void run() {
30            String currentThreadName =
31                Thread.currentThread().getName();
32            synchronized (firstResource) {
33                System.out.println(currentThreadName + ": holding " +
34                    firstResource);
```

```

33         System.out.println(currentThreadName + ": requesting "
+ secondResource);
34         synchronized (secondResource) {
35             System.out.println(currentThreadName + ": get " +
secondResource);
36         }
37         System.out.println(currentThreadName + " finished.");
38     }
39 }
40
41 }
42
43 public static void main(String[] args) throws InterruptedException
{
44
45     Thread x = new Thread(new TwoResourceRunnable(resourceB,
resourceA), "X");
46     Thread y = new Thread(new TwoResourceRunnable(resourceA,
resourceB), "Y");
47
48     x.start();
49     y.start();
50 }
51
52
53 }
54

```

运行结果

```

1 X: holding resource B
2 Y: holding resource A
3 X: requesting resource A
4 Y: requesting resource B

```

结合上述代码，理解死锁形成的四个必需条件：

### 1. 互斥条件

每个资源只能被一个线程占用。在上述代码中，资源 resourceA 和 resourceB 是被 synchronized 块保护的临界资源，只能被一个线程占用。

### 2. 请求与保持条件

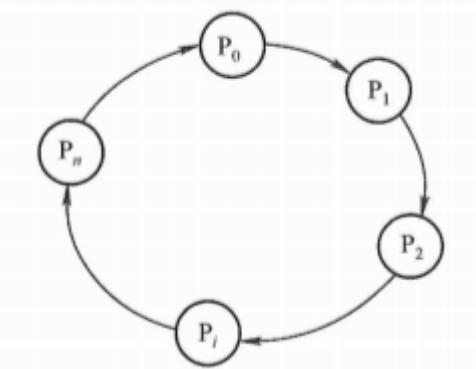
线程至少持有一个资源，并且正在请求至少一个资源，而请求的资源被其他线程占用。在上述代码中，线程X和线程Y都持有一个资源，并且尝试获取另一个资源。

### 3. 不可剥夺条件

资源只能在持有者线程释放后才能被其他线程获取，不能被其他线程强行剥夺。在上述代码中，一旦线程获取了资源的锁，其他线程无法主动剥夺这个锁。

#### 4. 循环等待条件

多个线程形成一个循环等待资源的环链。如图所示，线程  $P_1$  等待线程  $P_2$  释放资源 ... 线程  $P_n$  等待线程  $P_1$  释放资源。在上述代码中，线程 X 等待线程 Y 持有的资源，线程 Y 等待线程 X 的资源，形成了循环等待条件。



若需要解除死锁，只需要破坏上述四个条件之一即可。

## 活锁

### 饥饿

在多线程的情形下，饥饿指一个或者多个线程由于某种原因无法得到其所需的资源，从而无法继续执行。这可能导致线程被长时间阻塞，无法完成其任务。与死锁和活锁不同，饥饿是指一个或多个线程无法取得进展，而不仅仅是线程之间的相互影响。

以下是一个示例，低优先级的线程若长时间不能得到资源，则运行失败

```
1 package com.congee02.multithread.liveness;
2
3 import java.util.concurrent.TimeUnit;
4
5 public class StarvationDemo {
6
7     private static final Object resource = new Object() {
8         @Override
9         public String toString() {
10             return "resource";
11         }
12     };
13
14     // 定义一个长时间持有资源的任务
15     private final static Runnable holdResourceLongTimeRunnable = () ->
16     {
17         String name = Thread.currentThread().getName();
18         System.out.println(name + " is trying to access the
resource");
19         synchronized (resource) {
20             try {
21                 TimeUnit.SECONDS.sleep(10);
22             } catch (InterruptedException e) {
23                 e.printStackTrace();
24             }
25         }
26     };
27 }
```

```
19         System.out.println(name + " acquired the resource.");
20         try {
21             // 模拟高优先级线程长时间占用资源
22             TimeUnit.SECONDS.sleep(10);
23         } catch (InterruptedException e) {
24             System.out.println(name + " is interrupted.");
25         }
26         System.out.println(name + " released the resource.");
27     }
28 }
29
30     private static Thread setThreadPriority(Thread thread, int
31 priority) {
32         thread.setPriority(priority);
33         return thread;
34     }
35
36     public static void main(String[] args) throws InterruptedException
37     {
38         // 创建一个优先级最高的线程
39         Thread highPriorityThread =
40             setThreadPriority(new
41             Thread(holdResourceLongTimeRunnable, "High-Priority-Thread"),
42             Thread.MAX_PRIORITY);
43
44         // 创建一个优先级最低的线程
45         Thread lowPriorityThread =
46             setThreadPriority(new
47             Thread(holdResourceLongTimeRunnable, "Low-Priority-Thread"),
48             Thread.MIN_PRIORITY);
49
50         // 启动两个线程
51         highPriorityThread.start();
52         lowPriorityThread.start();
53
54         Thread.sleep(100);
55
56         int count = 0;
57         while (lowPriorityThread.getState() == Thread.State.BLOCKED) {
58             TimeUnit.SECONDS.sleep(1);
59             count++;
60             if (count == 5) {
61                 System.err.println("Time out, thread" +
62                     lowPriorityThread + "becomes invalid.");
63                 lowPriorityThread.stop();
64             }
65         }
66     }
67 }
```

代码中，高优先级的线程长时间占用资源，致使低优先级的线程因为长时间未获取到资源而被饿死。

## 性能问题

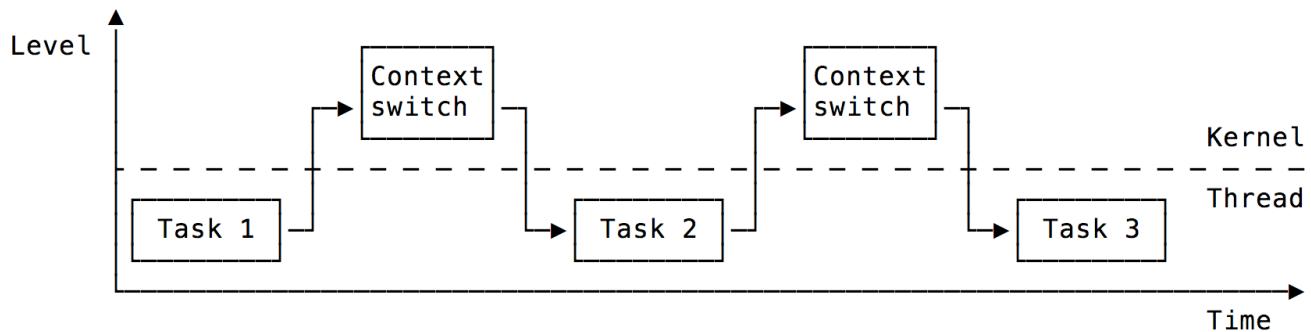
即使线程安全和活跃性问题都没有发生，多线程并发也并不一定比串行执行要快。多线程并发编程需要创建进程，而且进程之间需要上下文切换，这些操作都是有代价的，需要考虑。

### 创建线程开销

创建线程时，需要向系统申请资源。创建线程的开销十分昂贵，需要给新创建的线程分配内存，列入调度等。

### 线程上下文切换

为了实现多个线程同时运行（宏观上），CPU 为多个不同的线程分配时间片，在时间片内特定线程可以使用 CPU。当一个线程从另一个线程切换过来时，CPU 需要保存现场，并执行下一个线程，这个行为被称为上下文切换。



若要减少上下文切换带来的开销，可以：

- 减少线程数目  
过多的线程会产生频繁的上下文切换
- 无锁并发编程

较为典型的用法为 ConcurrentHashMap 分段锁的思想。ConcurrentHashMap 将数据分为多个段，每个段由一个独立的锁保护，减小整体的锁的竞争程度，以便多个线程可以并发地访问不同的段。总结来说，锁分段是一种优化策略，旨在通过将数据分成多个部分，并为每个部分提供独立的锁，以减小锁竞争的影响，从而提高并发性能。

- CAS 算法  
CAS 算法在更新数据时，允许尝试更新一个值，仅在值未被其他线程改变时才进行更新，避免了传统锁带来的线程阻塞和切换。
- 使用更轻量级的线程
- 局部性原则

将相关的数据放在一起，使得线程在执行任务时尽可能使用缓存，减少缓存失效带来的开销。这有助于减小上下文切换的开销。

## Java 内存模型

### 并发编程模型的通信和同步

线程通信解决线程间如何通信的问题；线程同步保证线程执行顺序正确，依赖于线程通信实现。

主要有两种并发模型解决该问题：

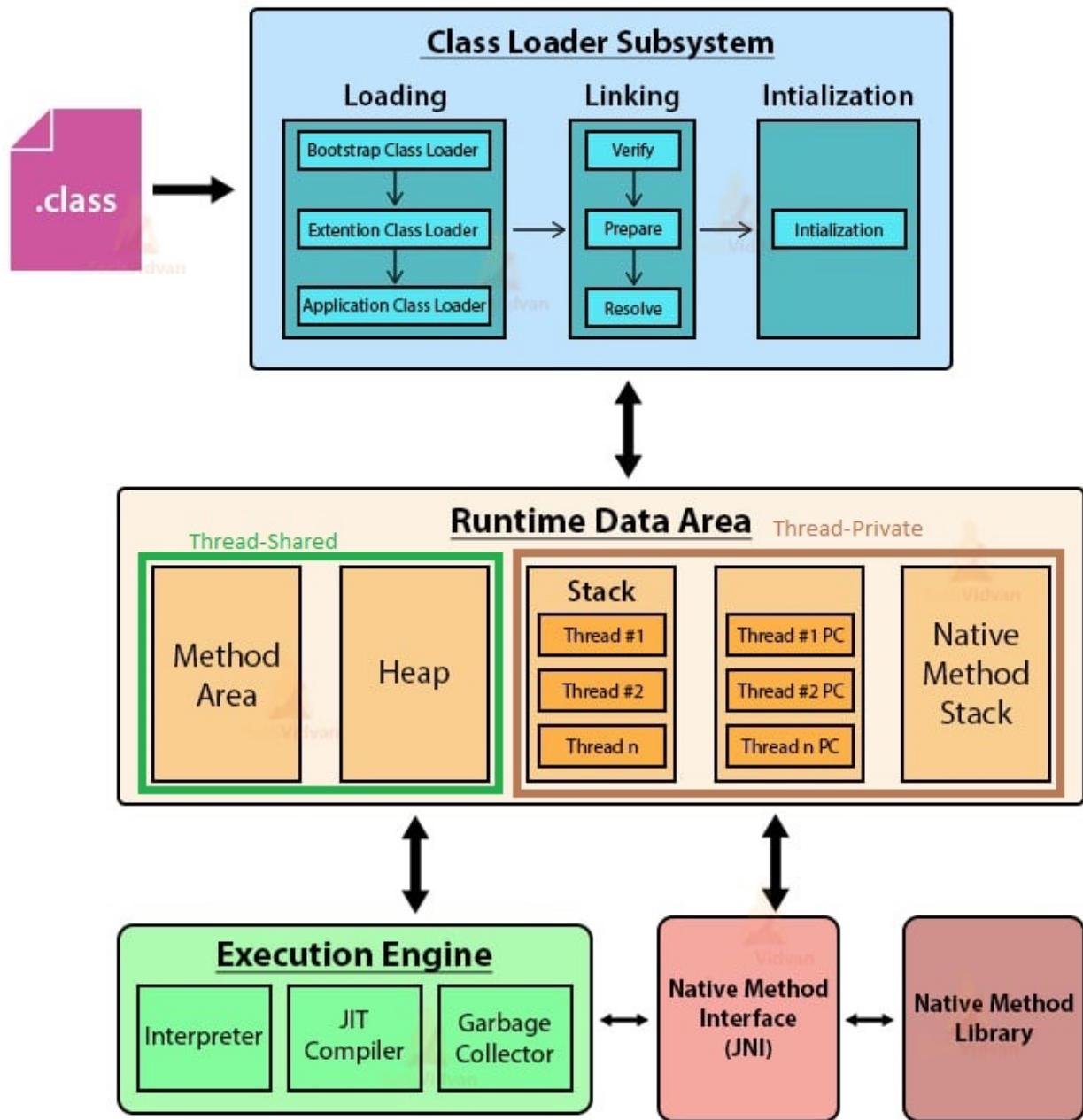
模型	线程通信	线程同步
消息传递模型	线程间无公共状态。线程间使用消息接发显式地通信。	发送消息和接受消息总是有先后顺序的，所以线程间进行隐式的同步。
共享内存模型	线程间有公共状态。通过读写线程间的公共状态隐式地通信。	必须使用 <code>synchronized</code> 关键字等手段确认某段代码或者某个变量需要互斥访问，其同步是显式的。

在 Java 中，主要使用共享内存模型来进行线程通信和线程同步。

## Java 抽象内存模型

### 运行时内存的划分

# JVM Model



如上图 JVM 模型，在运行时数据区，于任意线程，栈（虚拟机栈 Stack 和本地方法栈 Native Method Stack）和程序计数器（PC）都是私有的，而堆和方法区是共享的。接下来说讲运行时数据区的各个部分：

- 虚拟机栈（Stack，更确切地说是 VM Stack）

每个线程都有自己的虚拟机栈，用于存储方法调用的局部变量、操作数栈、方法参数和返回值。每个方法调用在虚拟机上创建一个栈帧（栈帧是用于存储方法调用信息、局部变量和操作数栈的内存区域，支持方法的执行和返回）。

- 本地方法栈（Native Method Stack）

类似于虚拟机栈，每个线程都有自己的本地方法栈，用于存储本地方法（用 native 关键字修饰的方法）的调用信息和执行状态。

- 程序计数器（Program Counter，PC）

每个线程都有一个独立的程序计数器，保存当前线程正在执行的字节码指令的行号。它是线程私有的，用于支持方法调用和线程切换后的恢复。

- 方法区 (Method Area)

方法区存储类的结构信息，包括类的成员变量、方法代码、静态变量、常量池等。方法区是所有线程共享的。

- 堆 (Heap)

堆是存储对象实例的区域，所有线程共享。堆内存被用于存储所有创建的对象，包括类的实例和数组。堆是Java内存管理的主要焦点

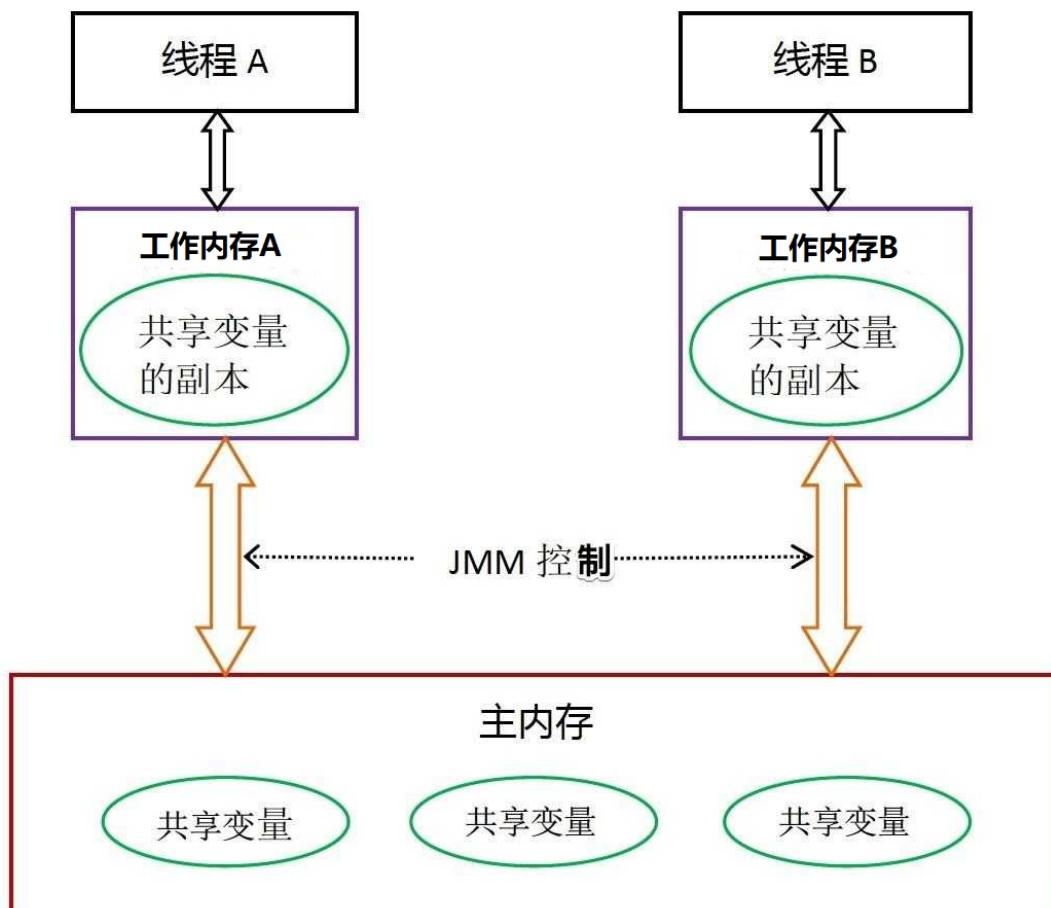
上面提到，栈和程序计数器是各个线程私有的，那么其中的内容（局部变量、异常处理参数 ... ...）不会在线程之间共享，也就不会产生可见性问题。要研究可见性问题，应该着眼于线程共享的方法区和堆区。

### 如果堆是共享的，为什么堆中可能存在不可见问题？

首先需要阐明不可见问题和堆的共享性无关，而是和多线程本身的并发访问有关。

当一个线程修改一个未被 `volatile` 修饰的变量时，修改先在自己线程独有的工作内存中应用，而不会立马在主内存中应用更改，而其他线程不能在主内存新的值。

当一个线程访问一个未被 `volatile` 修饰的变量时，该变量的修改可能已经在其他线程的工作内存应用，而没有应用到主内存，可能导致当前线程访问到旧的值甚至是不完整的对象。



需要注意，上述共享变量都没有被 `volatile` 关键字修饰

从图中可见：

1. 共享变量的实例都在主内存中，且每个线程都保持至少一份必要的共享变量的副本在自己的工作内存
2. 如果线程 A 需要向线程 B 通信，则线程 A 需要将更新后的共享变量刷新到主内存，此后，线程 B 读取更新后的共享变量。

## JMM 和 Java 运行时数据区

JMM (Java Memory Model) 和 Java 运行时数据区 (Java Runtime Data Area) 是两个不同的，容易混淆的概念，下面主要说明两者的区别以及联系。

### Java Memory Model, JMM

JMM 是一种规范，用于定义多线程程序中，线程之间共享变量的访问方式、内存可见性、指令重排等行为。JMM 定义了一系列规则，确保多线程程序在不同线程之间的操作对其他线程可见，并避免出现由于指令重排导致的意外行为。

JMM 的主要目标是提供一种形式化的规范，使得多线程程序中的线程能够正确协同工作，而不会出现数据不一致或者其他意外结果。

简言之，JMM 是一种规范，定义了多线程程序中的内存可见性和访问方式。

### Java Runtime Data Area, Java 运行时数据区

Java 运行时数据区是 JVM 运行 Java 程序所使用的内存区域的总称。运行时数据区包含了许多个不同的区域，用于存储不同类型的数据，包含堆、方法区、虚拟机栈、本地方法栈、程序计数器等，记录了类的结构信息、对象实例、线程执行的状态信息等。

简言之，Java 运行时数据区是 Java 程序在运行时所使用的内存划分，用于存储程序的数据和状态。

## 指令重排

指令重排是编译器和处理器提高程序性能的有效手段。

那么，指令重排如何提高性能？简言之，是通过尽可能地利用现代 CPU 的流水线架构，尽可能减少流水线中断发生的次数。

具体而言，分为以下几点：

### 1. 流水线

将指令的执行过程分为多个阶段，以充分利用现代处理器的流水线架构，减少指令执行的等待时间。

### 2. 减少分支判断

分支语句可能导致处理器的流水线分支中断，降低性能。通过指令重排，尽量避免分支预测错误，可以减少分支延迟对程序性能的影响。

### 3. 提高局部性

指令重排可以让同一段代码在不同的时间点多次执行，从而充分利用指令的缓存，减少缓存的冷启动时间。

指令重排可以使紧密相关的指令在空间上紧密排列，从而利用处理器的数据缓存。

### 4. 并行执行

处理器有多个执行单元，可以同时执行多条指令。指令重排可以使独立的指令并行执行，从而提高整体性能。

指令重排分为如下三类：

- 编译器重排

在单线程情形下，在不改变单线程程序语义的前提下，重新安排语句的执行顺序。

- 处理器重排

现代处理器拥有多级流水线和多核心，允许在执行过程中对指令进行重排，以最大程度地利用处理器资源

- 内存重排

处理器为了优化性能，在执行程序时可能会对内存操作进行重新排序，以最大程度地利用处理器内部资源

## **JMM 如何协调指令重排，确保多线程程序在不同的线程之间保持正确的内存操作顺序和可见性？**

JMM 定义了下述概念：

### 1. 原子性 Atomicity

JMM 保证基本的操作（如读写操作）是原子的，即它们不会被中断或交错执行。

### 2. 可见性 Visibility

JMM 保证线程对共享变量的修改对其他线程是可见的，即一个线程对共享变量的修改在另一个线程中是可以看到的。

### 3. 顺序性 Ordering

一个线程内的指令不会被重排或者乱序执行，保持了程序次序（Program Order）。然而，JMM 不保证不同线程之间的操作顺序。这是因为在多线程编程中，不同线程之间的操作可能会交错执行，导致不同线程中的操作顺序与程序次序不一致。JMM 并不保证一个线程的操作会立即对其他线程可见，也不保证不同线程间的操作顺序，除非使用了适当的同步机制。

### 4. happens-before 关系

这是 JMM 中重要的概念，用于定义内存操作之间的顺序关系。如果操作 A happens-before 操作 B，那么操作 A 对于操作 B 是可见的

## **happens-before 规则**

### 1. 程序顺序规则

在同一个线程中，按照源代码顺序执行的操作会确保 A 的结果在 B 执行时对于线程可见。

### 2. 监视器的锁的释放-获取规则

如果线程 T1 在释放互斥锁后，线程 T2 获取相同的互斥锁。那么 T1 的释放操作 happens-before T2 的获取操作，确保对于互斥锁的操作在释放和获取之间具有顺序性和可见性。

### 3. volatile 变量规则

如果线程 T1 对 volatile 变量进行写操作，然后线程 T2 对相同的 volatile 变量进行读操作，那么 T1 的写操作 happens-before T2 的读操作，确保对 volatile 变量的写入对其他线程的读是可见的。

### 4. 线程启动规则

如果线程 T1 启动线程 T2，那么线程 T1 的启动 T2 的操作 happens-before T2 的所有操作，确保新线程在启动后执行。

### 5. 线程终止规则

如果线程 T1 终止，那么 T1 的所有操作 happens-before 其他线程检测到 T1 已经终止

### 6. 中断规则

如果线程 T1 中断线程 T2，那么 T1 的中断操作 happens-before T2 检测到中断事件

## **volatile 关键字**

volatile 关键字的作用是告诉编译器和运行时环境，被修饰的变量可能会被多个线程访问，因此需要特殊处理来确保线程之间的一致性和可见性。具体来说，volatile 关键字有两个作用：

### 1. 可见性

当一个线程修改了 volatile 变量的值，这个修改立即会写回到主内存。其他线程在读取这个变量时，会从主内存中重新加载最新的值，而不是使用本地缓存（工作内存）中的旧值。这样确保所有的线程都能看到最新的值，避免了因为线程不一致而产生的问题。

### 2. 禁止重排序

编译器和处理器在优化代码时可能会对指令进行重排序，以提高性能。然而在多线程情况下，重排序可能导致意外的结果。通过使用 volatile，可以告诉编译器和处理器不要对 volatile 变量的读写操作进行重排序，从而确保正确的执行顺序。

在使用 volatile 时，需要注意以下几点：

### 1. 不保证原子性

不能保证复合操作比如 count ++ 的原子性，此时需要额外的同步手段来保证操作原子性。

### 2. 有限使用场景

`volatile` 主要适用于变量的读取操作比写入操作频繁的情况，例如开关标志。对于复杂的操作，如多步操作的原子性要求，通常需要更加强大的同步机制，如 `synchronized` 或者 `java.util.concurrent` 包中的工具类

### 3. 不保证互斥性

`volatile` 不提供线程之间的互斥性，因此不能用来代替锁。如果多个线程需要进行复合操作，仍需要使用适当的锁机制来保证线程安全。

### 4. 性能开销大

由于 `volatile` 的特性要求对内存的写入和读取都是直接的，不能经过线程的工作内存，因此会引入一些性能开销。在高并发环境下，频繁使用 `volatile` 可能会影响性能。

简言之，`volatile` 关键字适用于需要保证可见性和禁止重排序的简单场景，但在需要复杂同步和原子操作的情况下，需要结合其他同步机制来确保线程安全。

## volatile 开关示例

```
1 package com.congee02.multithread.volatilec;
2
3 public class VolatileExample {
4
5     /**
6      * volatile 开关
7      */
8     private static class VolatileFlagToggle {
9         private volatile boolean flag = false;
10
11         public synchronized void toggleFlag() {
12             flag = ! flag;
13             System.out.println(Thread.currentThread().getName() + " :
14             " + "Flag has been toggled.");
15         }
16
17         public void printFlag() {
18             System.out.println(Thread.currentThread().getName() + " :
19             " + "The flag is " + flag + ".");
20         }
21     }
22
23     /**
24      * 打开开关
25      */
26     private static final Runnable writeRunnable = () -> {
27         toggle.toggleFlag();
28         try {
29             Thread.sleep(1000);
30         } catch (InterruptedException e) {
31             e.printStackTrace();
32         }
33     }
34 }
```

```

29         Thread.sleep(1000);
30     } catch (InterruptedException e) {
31         e.printStackTrace();
32     }
33 }
34
35 /**
36 * 等待开关开启
37 */
38 private final static Runnable readRunnable = () -> {
39     while (!toggle.flag);
40     toggle.printFlag();
41 };
42
43 public static void main(String[] args) {
44
45     final Thread writeThread = new Thread(writeRunnable,
46 "writeThread");
47     final Thread readThread = new Thread(readRunnable,
48 "readThread");
49
50     writeThread.start();
51     readThread.start();
52
53 }

```

在复杂操作的情况下，可以使用 `synchronized` 来提供更强的同步机制。

## **synchronized 关键字**

同步块：同步块Java 中用于实现线程同步的一种机制，允许你在代码中指定一个特定的代码块，以确保同一时间只有一个线程可以进入该代码块执行。具体而言，`synchronized` 可以修饰实例方法、静态方法、代码块。

`volatile` 不能提供线程之间对共享资源的互斥性，而 `synchronized` 可以弥补这点。`synchronized` 可以修饰方法和代码块，使其变为同步块。

## **修饰实例方法**

可将 `synchronized` 关键字直接应用于实例方法，使整个实例方法成为一个同步块。此时，`synchronized` 加锁的对象就是当前实例方法所在实例本身，也就是说当线程 A 在访问某个对象的 `synchronized` 实例方法时，其他线程不能访问该对象，包括该对象的非 `synchronized` 方法，此时锁的粒度可能会太大，从而降低性能。

```

1 public synchronized void add() {
2     this.counter++;
3 }

```

请看异步电源开关的例子：

```
1 package com.congee02.multithread.synchronizedc;
2
3 public class SynchronizedSwitchExample {
4
5     /**
6      * 异步电源开关
7      */
8     private static class SynchronizedPowerSwitch {
9
10        /**
11         * volatile: 保证其可见性 & 防止指令重排
12         */
13        private volatile boolean flag;
14
15        /**
16         * synchronized: 只允许一个线程切换开关
17         */
18        public synchronized void toggleFlag() {
19            flag = ! flag;
20            System.out.println("toggleFlag: " + "{ NanoTime: " +
21                (System.nanoTime() - baseNanoTime) + "; Thread: " +
22                Thread.currentThread().getName() + " }");
23        }
24
25        /**
26         * 检查开关
27         */
28        public boolean check() {
29            System.out.println("check: " + "{NanoTime: " +
30                (System.nanoTime() - baseNanoTime) + "; Thread: " +
31                Thread.currentThread().getName() + " }");
32            return flag;
33        }
34
35        private static final long baseNanoTime = System.nanoTime();
36
37        private final static Runnable toggleSwitchRunnable = () -> {
38            for (int i = 0 ; i < 5 ; i ++ ) {
39                powerSwitch.toggleFlag();
40            }
41        };
42    }
```

```

43     private final static Runnable checkSwitchRunnable = () -> {
44         for (int i = 0 ; i < 5 ; i ++ ) {
45             powerSwitch.check();
46         }
47     };
48
49     public static void main(String[] args) {
50
51         Thread toggleThread1 = new Thread(toggleSwitchRunnable,
52         "▲Toggle");
53         Thread toggleThread2 = new Thread(toggleSwitchRunnable,
54         "○Toggle");
55         toggleThread1.start();
56         toggleThread2.start();
57
58         Thread checkThread1 = new Thread(checkSwitchRunnable,
59         "▲Check");
60         Thread checkThread2 = new Thread(checkSwitchRunnable,
61         "○Check");
62         checkThread1.start();
63         checkThread2.start();
64     }
65 }

```

某次运行结果：

```

1  check: {NanoTime: 3660500; Thread: ○Check}
2  check: {NanoTime: 53886700; Thread: ○Check}
3  check: {NanoTime: 53945100; Thread: ○Check}
4  check: {NanoTime: 53995300; Thread: ○Check}
5  check: {NanoTime: 54042300; Thread: ○Check}
6  toggleFlag: { NanoTime: 3234900; Thread: ▲Toggle}
7  toggleFlag: { NanoTime: 54467000; Thread: ▲Toggle}
8  toggleFlag: { NanoTime: 54525600; Thread: ▲Toggle}
9  toggleFlag: { NanoTime: 54672400; Thread: ▲Toggle}
10  toggleFlag: { NanoTime: 54901900; Thread: ▲Toggle}
11  check: {NanoTime: 3972600; Thread: ▲Check}
12  check: {NanoTime: 55672800; Thread: ▲Check}
13  check: {NanoTime: 55850800; Thread: ▲Check}
14  check: {NanoTime: 55992100; Thread: ▲Check}
15  check: {NanoTime: 56143500; Thread: ▲Check}
16  toggleFlag: { NanoTime: 57288600; Thread: ○Toggle}
17  toggleFlag: { NanoTime: 57541300; Thread: ○Toggle}
18  toggleFlag: { NanoTime: 57690600; Thread: ○Toggle}
19  toggleFlag: { NanoTime: 57810500; Thread: ○Toggle}
20  toggleFlag: { NanoTime: 57922700; Thread: ○Toggle}

```

可以观察到，没有任何两个线程可以同时访问 powerSwitch 对象，其锁的粒度为当前实例对象。

## 修饰静态方法

synchronized 修饰静态方法的使用与实例方法并无差别，在静态方法上加上synchronized关键字即可，其锁的粒度为类。这里给出的例子是懒汉式单例模式的 getInstance 方法。

```
1 public synchronized static NaiveSynchronizedLazyLoadSingleton  
2     getInstance() {  
3         if (instance == null) {  
4             instance = new NaiveSynchronizedLazyLoadSingleton();  
5         }  
6         return instance;  
7     }
```

## 修饰代码块

格式如下

```
1 synchronized (lockObject) {  
2     // block code ...  
3 }
```

其 lockObject 可以是 this，一个类的 class 对象，具体的锁。前两者，一个对实例对象本身上锁，一个对类上锁。

```
1 synchronized (this) {  
2     // block code...  
3 }
```

等同于

```
1 <其他修饰符(不包含static)> synchronized methodName() {  
2  
3 }
```

```
1 public final class ExampleLockClass {
2
3     private ExampleLockClass() { }
4
5     public static void foo() {
6         synchronized (ExampleLockClass.class) {
7             System.out.println("Foo invoked.");
8         }
9     }
10
11 }
12 }
```

等同于

```
1 package com.congee02.multithread.synchronizedc;
2
3 public final class ExampleLockClass {
4
5     private ExampleLockClass() { }
6
7     public static synchronized void foo() {
8         System.out.println("Foo invoked.");
9     }
10
11 }
```

synchronized 的 lockObject 不是 this 也不是 某个类的 Class 对象的情况，请看多线程计数器的例子：

```
1 package com.congee02.multithread.synchronizedc;
2
3 import java.util.TreeMap;
4
5 public class CorrectSynchronizedCounterExample {
6
7     private static class CorrectSynchronizedCounter {
8
9         private Integer count = 0;
10        private final Object lock = new Object();
11
12        public void increment() {
13            synchronized (lock) {
14                count++;
15            }
16        }
17
18        public Integer getCount() {
19            synchronized (lock) {
```

```

20         return count;
21     }
22 }
23 }
24 }
25
26     private final static CorrectSynchronizedCounter counter = new
27     CorrectSynchronizedCounter();
28     private final static Runnable incrementRunnable = () -> {
29         for (int i = 0 ; i < 100 ; i++) {
30             counter.increment();
31         }
32     };
33     private final static Runnable getCountRunnable = () -> {
34         for (int i = 0 ; i < 100 ; i++) {
35             System.out.println(counter.getCount());
36         }
37     };
38
39     public static void main(String[] args) {
40         Thread incrementThread = new Thread(incrementRunnable,
41 "incrementThread");
42         Thread getCountThread = new Thread(getCountRunnable,
43 "getCountThread");
44         incrementThread.start();
45         getCountThread.start();
46     }
47 }

```

这里可能会产生疑问，会为什么不把 count 作为锁对象，而是专门创建一个锁对象呢，这是因为 Integer 对象是不可变的，执行 count ++ 时，实际上是创建了一个新的 Integer 对象并赋值给 count，导致每个进程进入 synchronized (count) 时获取的其实不同的锁对象，请看错误示范。

```

1 package com.congee02.multithread.synchronizedc;
2
3 public class ErrorSynchronizedCounterExample {
4
5     private static class ErrorSynchronizedCounter {
6         private volatile Integer count = 0;
7
8         public void increment() {
9             synchronized (count) {
10                 // Integer 对象是不可变的，执行 count ++ 时，
11                 // 实际上创建了一个新的 Integer 对象并赋值给 count，
12                 // 导致每个进程进入 synchronized (count) 时获取的其实是不同
13                 // 的锁对象
14                 count++;
15             }
16         }
17     }
18 }

```

```
14     }
15 }
16
17     public int getCount() {
18         synchronized (count) {
19             return count;
20         }
21     }
22 }
23
24 }
25 }
```

## Java 中锁的分类

### 根据线程预估的线程竞争情况分类

#### 悲观锁

悲观锁的核心假设是在多线程环境中，资源的访问会发生竞争，因此默认情况下假设任何访问共享资源的操作都会发生冲突。因此，在每次访问共享资源前，悲观锁都会将资源锁定，阻止其他线程的访问，确保数据的一致性。比如前面 `synchronized` 修饰代码块或者方法就是悲观锁的体现。

#### 乐观锁

乐观锁的核心假设是在多线程环境中，资源的访问冲突较少，因此默认情况下假定资源的访问不会发生冲突。乐观锁允许多个线程同时访问共享资源，不会阻塞其他线程的访问。然而，在更新共享资源时，乐观锁会在更新操作之前检查共享资源是否已经被其他线程修改过。如果共享资源尚未修改，更新可以成功；如果资源已被修改，更新操作失败，需要重新尝试。（CAS 算法）

### 悲观锁和乐观锁的适用场景

悲观锁通常用于写操作较多的情况下（多写场景，竞争激烈），这样可以避免频繁失败和重试影响性能，悲观锁的开销是固定的。不过，如果乐观锁解决了频繁失败和重试这个问题，可以考虑适用乐观锁。

乐观锁通常用于写操作较少的情况下（多读场景，竞争较少），这样可以避免频繁加锁影响性能。需要注意，

### 根据锁状态转换和竞争情况的不同处理方式分类

#### 偏向锁

偏向锁是乐观锁。偏向锁的理论基础在于：在多数情况下，同一线程会多次获取同一个锁的假设，认为在无竞争的条件下，同一线程多次获取锁的概率较高。其目标是，减少无竞争条件下的锁开销。核心思想：当第一个线程获取偏向锁时，记录该线程ID，使得第一个获取锁的线程不需要竞争，之后再次获取锁时可以直接获得。

## 轻量级锁

轻量级锁是乐观锁。轻量级锁是在少量线程竞争同一个锁的情况下，为了减少传统重量级锁的开销而引入的。当一个线程尝试获取锁时，JVM 会在对象头中存储锁记录（线程 ID 或者指向线程ID的指针）。如果没有竞争，获取和释放锁仅涉及少量的 CAS 操作。如果出现竞争，锁会升级为重量级锁。

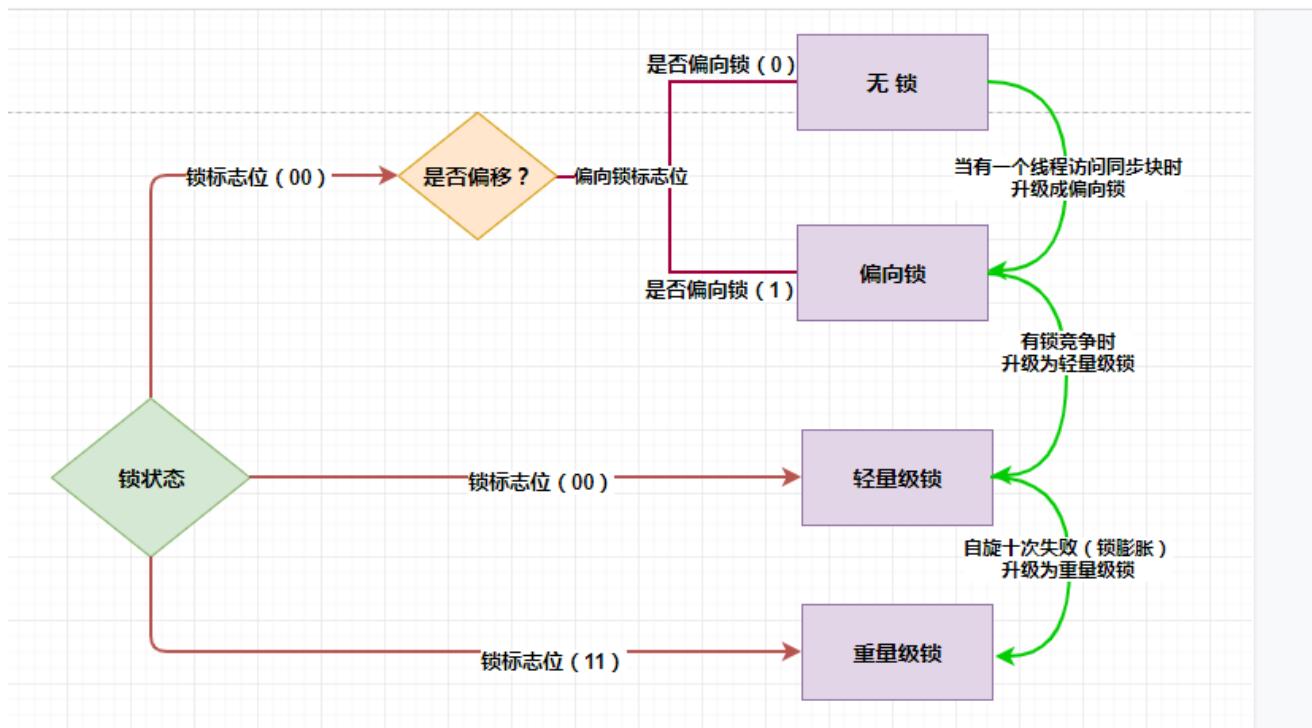
## 重量级锁

重量级锁是悲观锁。悲观锁是传统的锁机制，用于处理高度竞争的情况。它会涉及到线程的阻塞和唤醒，需要操作系统内核的支持。当多个线程竞争同一个锁时，其他线程会被阻塞，直到持有锁的线程释放锁。重量级锁的竞争情况下，会引起较高的上下文切换和性能开销。需要注意，在谈论死锁时，其谈到的锁是传统意义上的锁，也即重量级锁或者悲观锁。那么，偏向锁和轻量级锁不会引发死锁。

## 锁的对比

锁	优点	缺点	适用场景
偏向锁	加锁和解锁不需要额外的消耗，和执行非同步方法比仅存在纳秒级的差距。	如果线程间存在锁竞争，会带来额外的锁撤销的消耗。	适用于只有一个线程访问同步块场景。
轻量级锁	竞争的线程不会阻塞，提高了程序的响应速度。	如果始终得不到锁竞争的线程使用自旋会消耗 CPU。	追求响应时间。同步块执行速度非常快。
重量级锁	线程竞争不使用自旋，不会消耗 CPU。	线程阻塞，响应时间缓慢。	追求吞吐量。同步块执行时间较长。

## 锁的升级 (也称为锁的膨胀)



这三种锁状态是逐步升级的，从无锁开始，升级为偏向锁，再升级为轻量级锁，最后升级为重量级锁。

### 1. 无锁升级为偏向锁（第一次有线程获取锁）

偏向锁通常在第一次使用锁时使用，当一个线程第一次获取锁时，JVM 会将其升级为偏向锁。

### 2. 偏向锁升级为轻量级锁（无竞争条件 -> 存在竞争条件）

若偏向锁一直没有其他线程获取，则第一个线程不用获取锁直接执行代码（无竞争条件）。然而，如果有其他线程尝试获取相同的锁（此时存在竞争条件），偏向锁升级为轻量级锁。

### 3. 轻量级锁升级为重量级锁（自旋失败次数过多）

锁的自旋：当一个线程尝试获取一个已经被其他线程持有的锁时，如果该锁处于被锁定状态，那么该线程不会被立即阻塞等待，而是会尝试循环地重复尝试获取锁，这个循环的过程称为自旋。

当轻量级锁自旋多次失败时，JVM 会将当前的轻量级锁升级为重量级锁。需要注意，Java 锁自旋采用自适应自旋的策略，即：初始自旋次数较少，后随着自旋失败次数逐渐增加自旋次数，直至获取锁成功或者被升级为重量级锁。

## Compare-And-Swap, CAS

CAS 是乐观锁的一种实现机制，全称是比较并交换（Compare And Swap）。

CAS 算法有三个输入：

- V: 要更新的变量
- E: 预期值
- N: 新值

伪码如下

```
1 | def CAS (V, E, N):  
2 |     # 对比要更新的变量的值和预期值  
3 |     if (V equals E):  
4 |         # 更新变量，返回更新成功  
5 |         V = N  
6 |         return True  
7 |     else  
8 |         # 什么都不做，返回更新失败  
9 |         return
```

CAS 是一种原子操作，属于系统原语，是一条 CPU 的原子指令，在 CPU 层面保障 CAS 的原子性。那么，当多个线程同时使用 CAS 操作一个变量时，只有一个会胜出并成功更新。其余线程均会失败，但不会被挂起，只是被告知失败，并且允许再次尝试，也允许失败的线程放弃操作。

## ReentrantLock

ReentrantLock 是 Java 编程语言提供的一种可重入锁的实现，用于多线程编程中的同步控制。ReentrantLock 比传统的 synchronized 关键字提供了更多的功能和灵活性，允许开发者更精细地控制线程的同步行为。

ReentrantLock 的主要特点包括：

1. 可重入性
2. 公平性选择
3. 等待限时

## CAS 实现原子操作的三大问题

### ABA 问题

简单的 CAS 仅仅靠变量的值来识别该变量是否被其他线程改变过，这样实现存在潜在的问题：在 CAS 操作期间，变量的值从 A 变为 B，然后又从 B 变为 A，此时 CAS 无法察觉变量的值被其他线程改变过，产生不可预料的后果。

JDK 给出的解决方案是在变量面前追加版本号或者时间戳。从 JDK1.5 开始，JDK 的 atomic 包提供了一个 AtomicStampedReference 类来解决 ABA 问题。

该类的 compareAndSet 方法首先检查当前引用是否等于预期引用，并且检查当前标志是否等于预期标志，如果二者都相等，才使用 CAS 设置为新的值和标志。

```
1  public boolean compareAndSet(V    expectedReference,
2                                V    newReference,
3                                int  expectedStamp,
4                                int  newStamp) {
5
6     Pair<V> current = pair;
7
8     return
9        expectedReference == current.reference &&
10       expectedStamp == current.stamp &&
11       ((newReference == current.reference &&
12         newStamp == current.stamp) ||
13         casPair(current, Pair.of(newReference, newStamp)));
14 }
```

### 配合自旋时，CAS 循环时间开销大

CAS 多与自旋结合，如果自旋 CAS 长时间不成功，会占用大量的 CPU 资源。

解决思路是让 JVM 支持处理器提供的 pause 指令

pause 指令能让自旋失败时 CPU 睡眠一小段时间再继续自旋，从而大大降低读操作的频率，也大幅减少了 CPU 流水线断流的代价。

## CAS 只能保证一个共享变量的共享操作

1. 使用JDK 1.5开始就提供的AtomicReference类保证对象之间的原子性，把多个变量放到一个对象里面进行CAS操作；
2. 使用锁。锁内的临界区代码可以保证只有当前线程能操作。

## BlockingQueue, 阻塞队列

## AbstractQueuedSynchronizer, AQS

AQS 是 AbstractQueuedSynchronizer 的简称，翻译为 抽象队列同步器，从字面意思立即：

- 抽象：抽象类，只实现一些主要逻辑，部分方法为抽象方法，交给子类实现。
- 队列：使用先进先出 FIFO 队列来存储数据
- 同步：实现了同步的功能

# Java IO

## IO 流简介

IO 即 Input/Output，输入和输出。数据输入到计算机内存的过程称为输入，反之输出到外部存储的过程称为输出。数据传输过程类似于水流，因此被称为 IO 流。

在 IO 中流，按照数据流向分为 输入流 和 输出流，按照数据的处理方式分为 字节流 和 字符流，以此有了 Java IO 流中最基本的 4 个抽象类。

- InputStream: 字节输入流
- OutputStream: 字节输出流
- Reader: 字符输入流，InputStream 的子类
- Writer: 字符输出流，InputStream 的子类

## 字节流

### InputStream

Java 中的 InputStream 是一个字节输入流的抽象基类，用于从不同数据源中读取字节数据。它是输入流的基础接口，用于从不同数据源中读取字节数据。数据源可以是文件，可以是序列化文件，不同的数据源有不同的 InputStream 实现： FileInputStream(数据源是文件)，ObjectInputStream(数据源是对象序列化文件) ...

InputStream 常用方法：

方法	方法描述
int read()	从输入流读取下一个字节数据，并且返回读取的字节。正常读取时，返回的范围为 [0, 255]；如果读到流的末尾，则返回 -1
int read(byte[] buffer)	从输入流读取字节数据填充到给定的字节数组 <b>buffer</b> 中，返回实际读取的字节数。如果没有更多的数据可读，则返回 -1；等价于 <code>read(buffer, 0, buffer.length)</code>
int read(byte[] buffer, int offset, int length)	从偏移量 <b>offset</b> 开始，从输入流读取字节数据填充到指定的字节数组 <b>buffer</b> 中，最多读取 <b>length</b> 个字节，返回实际读取到的字节数。如果没有更多的数据可读，则返回 -1
long skip(long n)	跳过输入流中的 <b>n</b> 个字节不读，返回实际跳过的字节数
int available()	返回在没有阻塞的情况下可以从输入流读取的字节数，通常用于检查还有多少字节可读
void close()	关闭输入流并释放相关资源

从 JDK9 开始，InputStream 新增了一些加强方法：

方法	方法描述
byte[] readAllBytes()	读取输入流中所有字节，返回字节数组
int readNBytes(byte[] b, int off, int len)	阻塞直到读取 <b>len</b> 个字节到 <b>buffer</b> ，返回实际读取的字节数
long transferTo(OutputStream out)	读取当前流所有字节并写入到给定的输出流，返回写入的字节数

FileInputStream 是一个典型的字节输入流类，用于从文件中读取二进制数据。

比如，从 `input-file.txt` 文件中读取数据，跳过前缀：

`input-file.txt`

```
1 | PREFIX: DATA1
```

FileInputStreamDemo.java

```
1 | package com.congee02.bytes;
2 |
```

```
3 import java.io.FileInputStream;
4 import java.io.IOException;
5 import java.io.InputStream;
6
7 /**
8 * 以 FileInputStream 为例，演示 InputStream 的基本使用方法
9 */
10 public class FileInputStreamDemo {
11
12     private final static String PREFIX = "PREFIX: ";
13
14     public static void main(String[] args) {
15         // 使用 try-with-resource
16         try (InputStream in = new FileInputStream("input-file.txt")) {
17             // 查看流中可读的字节数
18             System.out.println("Remaining Bytes: " + in.available());
19             int readByte;
20             long prefixSkip = PREFIX.length();
21             long actualSkipBytes = in.skip(prefixSkip);
22             System.out.println("Skip the prefix, the actual skip
bytes: " + actualSkipBytes);
23             System.out.print("Reading the content: ");
24             // 读取数据
25             while ((readByte = in.read()) != -1) {
26                 System.out.print((char) readByte);
27             }
28         } catch (IOException e) {
29             e.printStackTrace();
30         }
31     }
32
33 }
34
35 }
```

运行输出：

```
1 Remaining Bytes: 13
2 Skip the prefix, the actual skip bytes: 8
3 Reading the content: DATA1
```

在上述的代码中，我们使用 `InputStream#read()` 函数逐字节的读取 `input-file.txt` 文件，可以想到，这样做会频繁调用 IO 操作，导致线程在等待 IO 的时间损耗较长，导致线程频繁阻塞。

因此，我们需要使用 `int read(byte[] buffer)` 方法，打开文件资源后，将多个字节先读取到缓冲 `buffer` 中，等到缓冲满或者是文件读完后，再将读到的字节传给程序。

下面分别使用带缓冲读取和逐字节读取来读取一个较大的 `txt` 文件，并对比性能。

## 测试性能的小工具类

```
1 package com.congee02;
2
3 public final class SimpleProfileUtils {
4
5     private SimpleProfileUtils() {}
6
7     /**
8      * 测试程序运行的纳秒时间
9      * @param runnable 运行的代码块
10     * @return 程序运行的纳秒时间
11     */
12    public static long profile(Runnable runnable) {
13        long start = System.nanoTime();
14        runnable.run();
15        long end = System.nanoTime();
16        return end - start;
17    }
18
19 }
20
```

## 对比测试

```
1 package com.congee02.bytes;
2
3 import com.congee02.SimpleProfileUtils;
4
5 import java.io.*;
6
7 public class FileInputStreamWithBufferVsWithoutBuffer {
8
9     private final static String READ_FILE_PATH = "1984.txt";
10
11     /**
12      * 带缓存地读取
13      */
14     private final static Runnable readWithBuffer = () -> {
15         byte[] buffer = new byte[4096];
16         int readByteNum;
17         try (InputStream in = new FileInputStream(READ_FILE_PATH)) {
18             while ((readByteNum = in.read(buffer)) != -1) {
19                 }
20             } catch (IOException e) {
21                 e.printStackTrace();
22             }
23     };
24 }
```

```

25  /**
26  * 不带缓存地读取
27  */
28 private final static Runnable readWithoutBuffer = () -> {
29     int readByte;
30     try (InputStream in = new FileInputStream(READ_FILE_PATH)) {
31         while ((readByte = in.read()) != -1) {
32         }
33     } catch (IOException e) {
34         e.printStackTrace();
35     }
36 };
37
38 public static void main(String[] args) {
39     System.out.println("不带缓存地读取\\t1984.txt 文件: \t" +
SimpleProfileUtils.profile(readWithoutBuffer));
40     System.out.println("带缓存地读取\\t1984.txt 文件: \t" +
SimpleProfileUtils.profile(readWithBuffer));
41 }
42
43 }
44

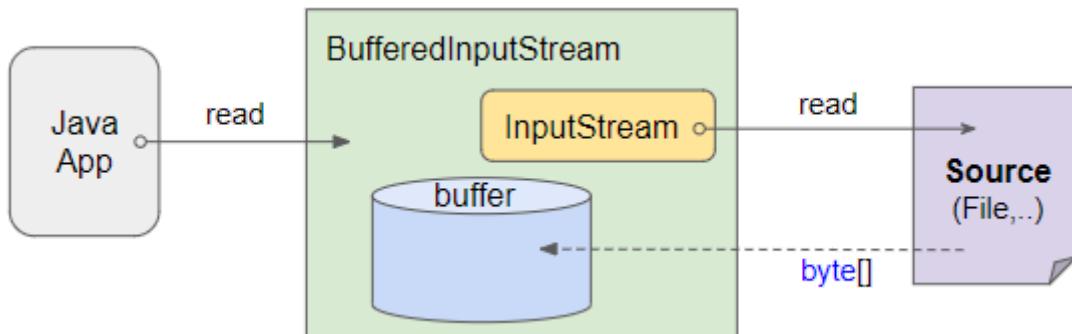
```

运行结果：

1	不带缓存地读取 1984.txt 文件:	74251900
2	带缓存地读取 1984.txt 文件:	234400

可见，带缓存读取文件时，因为一次性读取多个字节，所以无需频繁调用 IO 操作，效率远高于逐字节读取。事实上，JDK 提供了 `BufferedInputStream`，是 `InputStream` 的装饰器，用于增强 `InputStream`，提供了带缓冲读取流的功能。

## BufferedInputStream



`BufferedInputStream` 是 Java 标注库中的一个类，用于读取数据时提供缓冲功能，以提高输入操作的效率。它是 `InputStream` 的一个装饰器（继承自 `FilterInputStream`），通过在其上添加缓冲来减少直接从底层输入流读取数据的次数。

### BufferedInputStream 构造器

```

1 // in: 装饰的 in 对象
2 // size: 缓冲区大小
3 public BufferedInputStream(InputStream in, int size) {
4     // FilterInputStream
5     super(in);
6     if (size <= 0) {
7         throw new IllegalArgumentException("Buffer size <= 0");
8     }
9     buf = new byte[size];
10 }
11
12 // 使用默认的缓冲区大小 (8 KiB)
13 public BufferedInputStream(InputStream in) {
14     this(in, DEFAULT_BUFFER_SIZE);
15 }

```

对比 InputStream 和 BufferedInputStream 逐字节读取的性能

```

1 package com.congee02.bytes.buffer;
2
3 import com.congee02.utils.SimpleProfileUtils;
4
5 import java.io.*;
6
7 public class BufferedInputStreamRead {
8
9     private final static String READ_FILE_PATH = "1984.txt";
10
11     private static FileInputStream getFileInputStream() throws
12     FileNotFoundException {
13         return new FileInputStream(READ_FILE_PATH);
14     }
15
16     /**
17      * 使用 BufferedInputStream 逐字节读取
18      */
19     private static final Runnable bufferedByteByByteRead = () -> {
20         int readByte;
21         try (BufferedInputStream bufferedIn = new
22             BufferedInputStream(getFileInputStream())) {
23             while ((readByte = bufferedIn.read()) != -1) {
24                 }
25             } catch (IOException e) {
26                 e.printStackTrace();
27             }
28     };
29
30     /**
31      * 不带缓存地逐字节读取
32      */
33
34     private static final Runnable byteByByteRead = () -> {
35         int readByte;
36         try (FileInputStream fileInputStream = new
37             FileInputStream(READ_FILE_PATH)) {
38             while ((readByte = fileInputStream.read()) != -1) {
39                 }
40             } catch (IOException e) {
41                 e.printStackTrace();
42             }
43     };
44
45     public static void main(String[] args) {
46         SimpleProfileUtils.start("buffered");
47         bufferedByteByByteRead.run();
48         SimpleProfileUtils.end();
49
50         SimpleProfileUtils.start("noBuffer");
51         byteByByteRead.run();
52         SimpleProfileUtils.end();
53     }
54 }

```

```

30     */
31     private final static Runnable naiveByteByByteRead = () -> {
32         int readByte;
33         try (InputStream in = new FileInputStream(READ_FILE_PATH)) {
34             while ((readByte = in.read()) != -1) {
35                 }
36             } catch (IOException e) {
37                 e.printStackTrace();
38             }
39         };
40
41
42
43     public static void main(String[] args) {
44         System.out.println("bufferedByteByByteRead: " +
45             SimpleProfileUtils.profile(bufferedByteByByteRead));
46         System.out.println("naiveByteByByteRead: " +
47             SimpleProfileUtils.profile(naiveByteByByteRead));
48     }
49

```

运行结果：

```

1 | bufferedByteByByteRead: 2314600
2 | naiveByteByByteRead: 76154600

```

不出意外地，使用 `BufferedInputStream` 要比直接使用 `InputStream` 逐字节读取要快得多。这是因为 `BufferedInputStream` 会先读取固定长度（默认为 8192，即 8 KiB）的数据存储到 `byte` 数组缓冲区（`buf` 数组），然后逐字节读取时，先检查缓冲区是否被读完，如果未被读完，则从缓冲区中读取一个字节，并将指针前移，表示已经读过一个字节。可以在 `BufferedInputStream` 重写的 `read()` 方法中看到这个过程：

```

1 // 读取一个字节
2 public synchronized int read() throws IOException {
3     // count 指代当前缓冲区的有效字节数
4     // pos 指代当前缓冲区的位置位置索引
5     // pos >= count 表示当前缓冲区已经读完
6     if (pos >= count) {
7         // 重新从流中读取字节到缓冲区
8         fill();
9         // 如果当前缓冲区还是已经读完,
10        // 则表示整个流已经被读完, 返回 -1
11        if (pos >= count)
12            return -1;
13    }
14    // 尝试得到缓冲区, 从缓冲区取数后 pos ++。
15    // 只取其末尾 8 位 (1 个字节)

```

```

16     return getBufIfOpen() [pos++ ] & 0xff;
17 }
18
19 // 确保当前流关闭后, 缓冲区没有被置空
20 private byte[] getBufIfOpen() throws IOException {
21     byte[] buffer = buf;
22     if (buffer == null)
23         throw new IOException("Stream closed");
24     return buffer;
25 }

```

但是, 当调用 `read(byte[] b, int offset, int length)` 或者 `read(byte[])` 来读取流中字节时, `BufferedInputStream` 和 `InputStream` 的实现方式相同, 因此性能几乎没有差距, 或者说性能差距小到可以省略。

`BufferedInputStream` 的 `read(byte[] b, int offset, int length)` 实现来自于其父类 `FilterInputStream`, 直接调用被装饰对象的 `read(byte[], int, int)`。

```

1 /**
2  * 被装饰对象
3 */
4 protected volatile InputStream in;
5
6 public int read(byte b[], int off, int len) throws IOException {
7     return in.read(b, off, len);
8 }

```

我们用代码实际测试一下, 因为 IO 读取速度不稳定, 所以取其平均比值对比

```

1 package com.congee02.bytes.buffer;
2
3 import com.congee02.utils.SimpleProfileUtils;
4
5 import java.io.BufferedInputStream;
6 import java.io.FileInputStream;
7 import java.io.FileNotFoundException;
8 import java.io.IOException;
9
10 public class BufferedInputStreamBufferRead {
11
12     private final static String READ_FILE_PATH = "1984.txt";
13
14     private static FileInputStream getFileInputStream() throws
15     FileNotFoundException {
16         return new FileInputStream(READ_FILE_PATH);
17     }
18
19     private final static int READ_BYTE_BUFFER_SIZE = 8192;

```

```

20  /**
21  * BufferedInputStream 使用 buffer 读取
22  */
23  private final static Runnable bufferedInBuffer = () -> {
24      try (BufferedInputStream bufferedIn
25          = new BufferedInputStream(getFileInputStream(),
26          READ_BYTE_BUFFER_SIZE)) {
27          byte[] buffer = new byte[READ_BYTE_BUFFER_SIZE];
28          int readLength;
29          while ((readLength = bufferedIn.read(buffer)) != -1) {
30          }
31      } catch (IOException e) {
32          e.printStackTrace();
33      }
34  };
35 /**
36 * InputStream 使用 buffer 读取
37 */
38  private static final Runnable naiveInBuffer = () -> {
39      try (FileInputStream in = getFileInputStream()) {
40          byte[] buffer = new byte[READ_BYTE_BUFFER_SIZE];
41          int readLength;
42          while ((readLength = in.read(buffer)) != -1) {
43          }
44      } catch (IOException e) {
45          e.printStackTrace();
46      }
47  };
48
49  public static void main(String[] args) {
50      double avg = 0.0;
51      for (int i = 0 ; i < 100000 ; i++) {
52          final long y = SimpleProfileUtils.profile(naiveInBuffer);
53          final long x =
SimpleProfileUtils.profile(bufferedInBuffer);
54          avg += x / (double) y / (double) 100000;
55      }
56      System.out.println("bufferedInBuffer/naiveInBuffer: " + avg);
57  }
58
59 }
60

```

运行结果：

1	bufferedInBuffer/naiveInBuffer: 1.1311103060210648
---	--

可以看到 `BufferedInputStream` 稍慢于 `InputStream`，这是因为 `BufferedInputStream` 需要额外调用其构造器来申请缓存区空间 (`buf = new byte[size];`) 。

一般情况下，我们不单独使用 `InputStream`，而是使用经过 `BufferedInputStream` 装饰的 `InputStream`。再者，`BufferedInputStream` 是线程安全的，适用于并发环境。

## OutputStream

`OutputStream` 用于将内存中的字节信息写入到目的地。`OutputStream` 常用 API:

方法	方法描述
<code>void write(int b)</code>	将一个字节写入到输出流
<code>void write(byte[] b)</code>	将字节数组写入到输出流，等价于 <code>wirte(b, 0, b.length)</code>
<code>void write(byte[] b, int off, int len)</code>	从数组b 的 off 位置开始写入长度为 len 到输出流
<code>void flush()</code>	刷新输出流，确保所有缓冲的数据都被写入输入流
<code>void close()</code>	关闭输出流，释放相关资源

`FileOutputStream` 是常见的字节输出流对象，可以向指定文件写入字节数据。

创建一个 `output-file.txt` (如果没有)，并覆盖写入 "Hello, Java IO."

```
1 package com.congee02.bytes.out;
2
3 import java.io.FileOutputStream;
4 import java.io.IOException;
5
6 public class FileOutputStreamDemo {
7
8     private static final String WRITE_FILE_PATH = "output-file.txt";
9     private static final String WRITE_CONTENT = "Hello, Java IO.";
10
11     public static void main(String[] args) {
12         try (FileOutputStream out = new
13             FileOutputStream(WRITE_FILE_PATH)) {
14             byte[] bytes = WRITE_CONTENT.getBytes();
15             out.write(bytes);
16         } catch (IOException e) {
17             e.printStackTrace();
18         }
19     }
}
```

```
20 | }
21
```

## output-file.txt

```
1 | Hello, Java IO.
```

同样的，OutputStream 写入时，如果是单字节一次次写入，则会频繁使用 IO 操作，使得线程阻塞时间较长，影响整体性能。自然地，我们可以通过创建缓冲区来避免过于频繁的 IO 操作，来减少线程因为 IO 阻塞带来的时间损耗。这里不再赘述。

read(int) 和 read(byte[]) 对比：

```
1 | package com.congee02.bytes.out;
2
3 | import com.congee02.utils.SimpleProfileUtils;
4
5 | import java.io.*;
6 | import java.nio.charset.StandardCharsets;
7
8 | public class FileOutputStreamWithBufferVsWithoutBuffer {
9
10 |     private final static String longString;
11
12 |     private final static String WRITE_FILE_PATH = "1984.txt";
13
14 |     static {
15 |         StringBuilder sb = new StringBuilder();
16 |         try (final BufferedInputStream
17 |              in = new BufferedInputStream(new
18 | FileInputStream("1984.txt"))) {
19 |             byte[] buffer = new byte[8192];
20 |             int readLength;
21 |             while ((readLength = in.read(buffer)) != -1) {
22 |                 sb.append(new String(buffer, 0, readLength));
23 |             }
24 |         } catch (IOException e) {
25 |             e.printStackTrace();
26 |         }
27 |         longString = sb.toString();
28 |     }
29
30 |     private final static Runnable writeWithBuffer = () -> {
31 |         try (FileOutputStream out = new
32 | FileOutputStream(WRITE_FILE_PATH)) {
33 |             out.write(longString.getBytes(StandardCharsets.UTF_8));
34 |         } catch (IOException e) {
35 |             e.printStackTrace();
36 |         }
37 |     }
38 }
```

```

35    } ;
36
37    private final static Runnable writeWithoutBuffer = () -> {
38        try (FileOutputStream out = new
39             FileOutputStream(WRITE_FILE_PATH)) {
40            byte[] bytes =
41                longString.getBytes(StandardCharsets.UTF_8);
42            for (byte b : bytes) {
43                out.write(b);
44            }
45        } catch (IOException e) {
46            e.printStackTrace();
47        }
48    }
49
50    public static void main(String[] args) {
51        System.out.println("writeWithBuffer: " +
52            SimpleProfileUtils.profile(writeWithBuffer));
53        System.out.println("writeWithoutBuffer: " +
54            SimpleProfileUtils.profile(writeWithoutBuffer));
55    }
56
57 }

```

运行结果

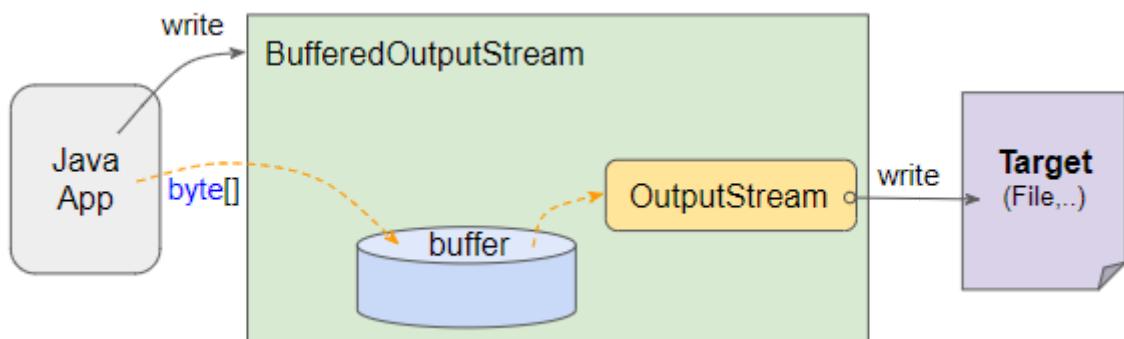
```

1 | writeWithBuffer: 552700
2 | writeWithoutBuffer: 3924600

```

同样的，既然 InputStream 有 BufferedInputStream，那么 OutputStream 可以有 BufferedOutputStream，用于创建带缓存的，线程安全增强的 OutputStream。

## BufferedOutputStream



BufferedOutputStream 是 Java 中继承于 FilterOutputStream (表明是个装饰类) 的一个类，它提供了一种将字节数据写入输出流并使用内部缓冲区的方法。使用 BufferedOutputStream 的主要目的是提高写操作的效率，特别是处理小块数据时。

类似于 `BufferedInputStream`，`BufferedOutputStream` 内部也维护了一个缓冲区，并且，这个缓存区的大小也是 **8192** 字节

同样的，`BufferedOutputStream` 对单字节写入做了优化。

```
1 package com.congee02.bytes.out.buffer;
2
3 import com.congee02.utils.SimpleProfileUtils;
4
5 import java.io.*;
6
7 public class BufferedOutputStreamByteByByteRead {
8
9     private final static String longString;
10    static {
11        StringBuilder sb = new StringBuilder();
12        try (final BufferedInputStream
13             in = new BufferedInputStream(new
14             FileInputStream("1984.txt"))) {
15            byte[] buffer = new byte[8192];
16            int readLength;
17            while ((readLength = in.read(buffer)) != -1) {
18                sb.append(new String(buffer, 0, readLength));
19            }
20        } catch (IOException e) {
21            e.printStackTrace();
22        }
23        longString = sb.toString();
24    }
25    private final static byte[] writeBytes = longString.getBytes();
26
27    private final static String WRITE_FILE_PATH = "1984-copy-1.txt";
28
29    private static FileOutputStream getFileOutputStream() throws
30    FileNotFoundException {
31        return new FileOutputStream(WRITE_FILE_PATH);
32    }
33
34    private final static Runnable bufferedByteByByteWrite = () -> {
35        try (BufferedOutputStream bos = new
36             BufferedOutputStream(getFileOutputStream())) {
37            for (byte b : writeBytes) {
38                bos.write(b);
39            }
40        } catch (IOException e) {
41            e.printStackTrace();
42        }
43    };
44}
```

```

42     private final static Runnable naiveByteByByteWrite = () -> {
43         try (FileOutputStream out = getFileOutputStream()) {
44             for (byte b : writeBytes) {
45                 out.write(b);
46             }
47         } catch (IOException e) {
48             e.printStackTrace();
49         }
50     };
51
52     public static void main(String[] args) {
53         System.out.println("bufferedByteByByteWrite: " +
54             SimpleProfileUtils.profile(bufferedByteByByteWrite));
55         System.out.println("naiveByteByByteWrite: " +
56             SimpleProfileUtils.profile(naiveByteByByteWrite));
57     }
58

```

运行结果：

```

1 | bufferedByteByByteWrite: 9579300
2 | naiveByteByByteWrite: 121322700

```

BufferedOutputStream 也重写了父类 FilterOutputStream 的 write(byte[] b, int off, int len)

开发中不直接使用 OutputStream，而先使用 BufferedOutputStream 装饰后使用。

## 字符流

字符流处理的是字符，字节流处理的是字节。但是实际上，字符流底基于字节流，只是额外提供了一层字符编码（UTF-8、UTF-16）的抽象。也可以说：字符流处理的是经过编码的字节流。

既然字符流和字节流本质都是操作字节，那为什么会有个字符流呢？

- **字符编码和国际化支持：** 文本数据在计算机内部以字节形式表示，但这些字节需要根据特定的字符编码转换为人类可读的字符。这涉及到字符的映射和编码，而不同语言和文化有不同的字符需求。字符流的一个主要优势是它们内置了字符编码和解码，使得在处理多语言文本时更加方便。例如，UTF-8、UTF-16等字符编码可以处理不同语言的字符。
- **换行符处理：** 不同操作系统使用不同的换行符表示行尾（例如，Windows使用"\r\n"，Unix使用"\n"）。字符流会自动处理这些换行符的转换，使得在不同操作系统间移植文本文件更容易。
- **高级处理：** 字节流在处理文本时需要手动处理字符编码、换行符等细节。而字符流将这些细节封装起来，提供更高级别的抽象，减少了开发人员处理细节的负担。

- **文本处理的便利性：**由于大部分应用程序处理的是文本数据，字符流提供了更适合这种场景的API。字符流允许按字符而不是字节读取数据，这对于处理文本数据更自然。

当我们尝试使用字节流读取一个带中文的文本文件，可能会因为未采取合适的字符编码，出现乱码。

chinese-info.txt

```
1 | 你好，Java IO.
```

测试代码

```
1 package com.congee02.character;
2
3 import java.io.FileInputStream;
4 import java.io.IOException;
5
6 public class GarbledRead {
7
8     public static void main(String[] args) {
9         StringBuilder sb = new StringBuilder();
10        try (FileInputStream in = new FileInputStream("chinese-
11          info.txt")) {
12            int readByte;
13            while ((readByte = in.read()) != -1) {
14                sb.append((char) readByte);
15            }
16        } catch (IOException e) {
17            e.printStackTrace();
18        }
19        System.out.println(sb);
20    }
21 }
22 }
```

运行结果：

```
1 | ä½ å¥½ï¼•Java IO.
```

## Reader

Reader 是 Java 标准库中用于读取字符数据的抽象类。它是字符流的基类，提供了一组用于读取字符数据的方法（字符输入流）。与 InputStream 大体相同，区别在于 InputStream 操纵的最小单元是字节，Reader 操纵的最小单元是字符。常见 API 如下：

方法签名	描述
int read() throws IOException	读取并返回一个字符的Unicode值，到达流末尾返回-1。
int read(char[] cbuf) throws IOException	将字符读入到字符数组cbuf中，返回读取的字符数，到达流末尾返回-1。
int read(char[] cbuf, int off, int len) throws IOException	将字符读入到字符数组cbuf的指定位置，返回读取的字符数。
long skip(long n) throws IOException	跳过指定数量的字符。
boolean ready() throws IOException	检查是否可以从流中读取数据而不阻塞。
void close() throws IOException	关闭流，释放相关资源。

我们尝试使用 `FileReader` 再次读取上述带中文的文件。

```

1 package com.congee02.character.in;
2
3 import java.io.FileReader;
4 import java.io.IOException;
5
6 public class FileReaderDemo {
7
8     private final static String READ_FILE = "chinese-info.txt";
9
10    public static void main(String[] args) {
11        try (final FileReader reader = new FileReader(READ_FILE)) {
12            int readCharacter;
13            while ((readCharacter = reader.read()) != -1) {
14                System.out.print((char) readCharacter);
15            }
16        } catch (IOException e) {
17            e.printStackTrace();
18        }
19    }
20
21 }
22

```

运行结果：

```
1 | 你好, Java IO.
```

可以看到，当使用 `FileReader` 时，自动选取了合适的编码方式，中文被正常打印了。

让我们猜想一下这背后的基本步骤：

1. 打开文件
2. 读取文件字节
3. 获取文件编码格式
4. 按照编码格式将读取到的文件字节转换为字符
5. 返回读取到的字符

查看源码，发现 `FileReader` 继承自 `InputStreamReader`，此外没有自己的方法，只是创建一个 `FileInputStream` 对象作为 `InputStreamReader` 的被装饰对象。`InputStreamReader` 使用默认编码编码来创建 `StreamDecoder` 解码器对象来解析字节为字符，也可以自己决定字符编码。

```
1  public InputStreamReader(InputStream in) {  
2      super(in);  
3      // 使用默认的字符编码格式解析  
4      sd = StreamDecoder.forInputStreamReader(in, this,  
5                          Charset.defaultCharset()); // ## check lock object  
6  }  
7  
8  
9  // 使用给定的字符编码格式解析  
10 public InputStreamReader(InputStream in, String charsetName)  
11     throws UnsupportedEncodingException  
12 {  
13     super(in);  
14     if (charsetName == null)  
15         throw new NullPointerException("charsetName");  
16     sd = StreamDecoder.forInputStreamReader(in, this, charsetName);  
17 }  
18  
19 // 使用给定的字符编码格式解析  
20 public InputStreamReader(InputStream in, Charset cs) {  
21     super(in);  
22     if (cs == null)  
23         throw new NullPointerException("charset");  
24     sd = StreamDecoder.forInputStreamReader(in, this, cs);  
25 }
```

当调用 `FileReader` 的 `read()` 方法时，实际上是调用 `InputStreamReader` 的 `StreamDecoder` 对象 `sd` 的 `read` 方法来将读取到的字节转换为字符。`read(byte[], int, int)` 也是同理。所以字符流能够正确读取字符受益于 `StreamDecoder` 对象执行的转换操作。

```

1 public int read() throws IOException {
2     return sd.read();
3 }
4 public int read(char cbuf[], int offset, int length) throws IOException
{
5     return sd.read(cbuf, offset, length);
6 }

```

## Writer

与 Reader 相对的，Writer 用于将内存中的字符信息写入到目的地，关键方法如下：

方法签名	描述
void write(int c) throws IOException	将指定的字符写入流中。
void write(char[] cbuf) throws IOException	将字符数组中的字符写入流中。
void write(char[] cbuf, int off, int len) throws IOException	将字符数组中从偏移量 off 开始的 len 个字符写入流中。
void write(String str) throws IOException	将字符串写入流中。
void write(String str, int off, int len) throws IOException	将字符串中从偏移量 off 开始的 len 个字符写入流中。
void flush() throws IOException	刷新流，将未写入的数据强制写入目标。
void close() throws IOException	关闭流，释放相关资源。

示例如下，写入当前时间

```

1 package com.congee02.character.out;
2
3 import java.io.FileWriter;
4 import java.io.IOException;
5 import java.util.Date;
6
7 public class FileWriterDemo {
8
9     private final static String WRITE_FILE_PATH = "file-writer-
output.txt";
10
11    public static void main(String[] args) {

```

```
12     try (FileWriter writer = new FileWriter(WRITE_FILE_PATH)) {
13         String writeContent = "APPEND-TIME: " + new Date();
14         writer.write(writeContent.toCharArray());
15     } catch (IOException e) {
16         e.printStackTrace();
17     }
18 }
19
20 }
21
```

我们看一下 `FileWriter` 是如何执行 `write` 方法的，所有重写的 `write` 都基于 `write(char[] cs, int off, int len)`，所以我们看它。

我们发现 `FileWriter` 没有直接写 `write(char[] cs, int off, int len)`，但是继承了 `OutputStreamWriter`，那么该方法来自 `OutputStreamWriter`。

`OutputStreamWriter#write(char cbuf[], int off, int len)`

```
1 private final StreamEncoder se;
2 public void write(char cbuf[], int off, int len) throws IOException {
3     se.write(cbuf, off, len);
4 }
```

和 `FileReader` 相似，实际上使用的是 `StreamEncoder` 编码器的 `write` 方法将读取到的字符编码为特定格式再进行写入。

## BufferedReader 和 BufferedWriter

`BufferedReader`（字符缓冲输入流）和 `BufferedWriter`（字符缓冲输出流）类似于 `BufferedInputStream`（字节缓冲输入流）和 `BufferedOutputStream`（字节缓冲输出流），内部都维护了一个字节数组作为缓冲区。不过，前者主要是用来操作字符信息。

## 打印流

打印流主要有两个类 `PrintStream` 和 `PrintWriter`。

### 1. 字符编码：

- `PrintStream` 使用底层平台默认的字符编码来处理字符数据，这可能导致在不同平台上的输出结果不一致。
- `PrintWriter` 允许指定字符编码，使得输出更加可控。这在处理多语言文本、网络通信等场景中特别有用。

### 2. 继承关系：

- `PrintStream` 是 `OutputStream` 的子类，因此它主要是基于字节的输出流，使用字节流来输出字符数据。它经常用于控制台输出。

- `PrintWriter` 是 `Writer` 的子类，它是基于字符的输出流，专门用于处理字符数据的输出。它通常用于文本文件输出和字符数据的网络传输。

### 3. 自动刷新：

- `PrintStream` 默认情况下在每次调用 `println()` 方法时会自动刷新，这意味着数据会立即写入目标。
- `PrintWriter` 默认情况下是非自动刷新的，您需要手动调用 `flush()` 方法或者关闭流来确保数据写入目标。

### 4. 异常处理：

- `PrintStream` 在抛出异常时可能会中断输出流，这可能会导致之后的输出被忽略。
- `PrintWriter` 在抛出异常时不会中断流，允许继续写入数据，但您需要检查流是否发生异常。

### 5. 可靠性：

- `PrintStream` 在处理控制台输出等简单场景时通常足够可靠。
- `PrintWriter` 在处理重要的文件输出和网络通信等场景时更加可靠，因为它提供了更多的控制和错误处理机制。

这里不作为重点解释，给出两个例子来演示 `PrintStream` 和 `PrintWriter`

```
1 package com.congee02.print.bytes;
2
3 import java.io.*;
4
5 public class PrintStreamDemo {
6
7     private final static String STREAM_WRITE_FILE_PATH = "PRINT-
8 STREAM-OUTPUT.txt";
9     private final static String WRITER_WRITE_FILE_PATH = "PRINT-
10 WRITER-OUTPUT.txt";
11
12     private final static Runnable printStreamWrite = () -> {
13         try (PrintStream out = new PrintStream(new
14             BufferedOutputStream(new FileOutputStream(STREAM_WRITE_FILE_PATH)))) {
15             out.printf("%d + %d = %d", 3, 4, 3 + 4);
16         } catch (FileNotFoundException e) {
17             e.printStackTrace();
18         }
19     };
20
21     private final static Runnable printWriterWrite = () -> {
22         try (PrintWriter writer = new PrintWriter(new
23             BufferedWriter(new FileWriter(WRITER_WRITE_FILE_PATH)))) {
24             writer.printf("%d * %d = %d", -1, -9, -1 * (-9));
25         } catch (IOException e) {
```

```

22         e.printStackTrace();
23     }
24 }
25
26 public static void main(String[] args) throws
27 FileNotFoundException {
28     printStreamWrite.run();
29     printWriterWrite.run();
30 }
31 }
32

```

需要额外说明的是，我们平时使用的 `System.out` 中的 `out` 是一个 `PrintStream` 对象，可以在 `System` 类看到，该 `PrintStream` 默认指向标准输入即命令行，也可以通过 `System.setOut(PrintStream)` 方法来设置其指向的输出（比如文件甚至是网络IO）。

## 随机访问流

随机访问流（Random Access Streams）是一种允许在文件中进行随机读写操作的流，它可以在文件中以任意顺序读取或写入数据，而不需要按顺序处理整个文件。在 Java 中，`RandomAccessFile` 是用于创建随机访问流的类，它提供了一系列方法来在文件中定位并进行读写操作。

以下是一些 `RandomAccessFile` 类的常用方法：

方法签名	描述
<code>long getFilePointer() throws IOException</code>	获取当前文件指针的位置。
<code>void seek(long pos) throws IOException</code>	设置文件指针的位置。
<code>int read() throws IOException</code>	从文件读取一个字节，并返回其整数值。
<code>int read(byte[] b) throws IOException</code>	从文件读取一组字节，存储在字节数组 <code>b</code> 中。
<code>int read(byte[] b, int off, int len) throws IOException</code>	从文件读取指定长度的字节，存储在字节数组 <code>b</code> 中的指定位置。
<code>void write(int b) throws IOException</code>	将指定字节写入文件。
<code>void write(byte[] b) throws IOException</code>	将字节数组中的所有字节写入文件。

方法签名	描述
void write(byte[] b, int off, int len) throws IOException	将字节数组中的指定范围字节写入文件。
void setLength(long newLength) throws IOException	设置文件的长度。如果指定的长度小于当前长度，则截断文件；如果大于当前长度，则在文件末尾添加零字节。
void close() throws IOException	关闭流。

在创建 RandomAccessFile 时，需要指定操作的文件和读写模式。

```

1  /**
2  * @param      file    文件对象
3  * @param      mode    读写模式
4  */
5  public RandomAccessFile(File file, String mode)
6      throws FileNotFoundException
7  {
8      this(file, mode, false);
9  }
10
11 // openAndDelete 是否在打开完毕后删除
12 private RandomAccessFile(File file, String mode, boolean
openAndDelete) {
13     // ...
14 }
```

RandomAccessFile 维护一个文件指针，表示下一个将要被写入或者读取的字节所处的位置。我们可以通过 seek(long pos) 来设置文件指针的偏移量（距离文件开头 pos 个字节处）。如需要获取文件指针当前位置，使用 getFilePointer() 方法。

```

1 package com.congee02.random;
2
3 import java.io.File;
4 import java.io.IOException;
5 import java.io.RandomAccessFile;
6 import java.nio.charset.StandardCharsets;
7
8 public class RandomAccessFileDemo {
9
10     private static final String RANDOM_ACCESS_MANIPULATION_FILE_PATH =
11         "RANDOM-ACCESS-MANIPULATION.txt";
12 }
```

```
12     private static void
13         printCurrentRandomAccessFileInfo(RandomAccessFile file) throws
14             IOException {
15             System.out.println("Offset before Reading: " +
16                 file.getFilePointer() +
17                     "; Character of Current Location: " +
18             (char) file.read() +
19                     "; Offset after Reading: " +
20             file.getFilePointer());
21         }
22
23
24     public static void main(String[] args) {
25         try (final RandomAccessFile file = new RandomAccessFile(new
26             File(RANDOM_ACCESS_MANIPULATION_FILE_PATH), "rw")) {
27
28             // 文件指针为 0, 在此处写入数据
29
30             file.write("ABCDEFGHIJKLMNOPQRSTUVWXYZ".getBytes(StandardCharsets.UTF
31             _8));
32
33             // 移动文件指针到 3 byte 的位置
34             file.seek(3);
35             printCurrentRandomAccessFileInfo(file);
36
37             // 在 4 byte 的位置写入数据
38             file.write("1234".getBytes(StandardCharsets.UTF_8));
39             printCurrentRandomAccessFileInfo(file);
40
41             // 移动文件指针到 0 byte 的位置
42             file.seek(0);
43             printCurrentRandomAccessFileInfo(file);
44
45         } catch (IOException e) {
46             e.printStackTrace();
47         }
48     }
49
50 }
```

RandomAccessFile 实现大文件续点上传 ...

## **FilterOutputStream & FilterInputStream**

### **字符流和字节流互相转换**