Basic Introduction Java

2025-02-10

1、String , StringBuilder, StringBuffer 区别 , 各自适用场景, 优缺点？以及String 为什么是不可变的?

总结：

- String：不可变的字符序列，适用于少量字符串的场景，比如常量定义、枚举值等。
- StringBuilder：可变的字符序列，适用于单线程下字符串的构建，适合于需要动态构建字符串的场景。
- StringBuffer：可变的字符序列，适用于多线程下字符串的构建，适合于需要动态构建字符串的场景。

1. String：
   底层是由字符数组实现的，且是由private final char[] value来保存字符串的字符序列, 没有提供修改字符串的方法, 因此是不可变的。
2. StringBuilder：
   底层是由字符数组实现的，且是由char[] value来保存字符串的字符序列, 提供了修改字符串的方法, 因此是可变的。适用于单线程下字符串的构建。
3. StringBuffer：
   底层是由字符数组实现的，且是由char[] value来保存字符串的字符序列, 提供了修改字符串的方法, 因此是可变的。适用于多线程下字符串的构建。

stringbuilder 和 stringbuffer 都继承AbstractStringBuilder,其中内部维护了一个初始容量为16的 可以变化的字符数组，通过append()方法添加字符

stringbuffer 提供的所有方法都由关键字synchronized修饰，因此是线程安全的。

2、java基本数据类型

整型：byte、short、int、long 分别占位8、16、32、64位, 分别占用字节 1、2、4、8。
浮点型：float、double 分别占位32、64位, 分别占用字节 4、8。
字符型：char 占位16位, 占用字节 2。
布尔型：boolean 占位1位, 占用字节 1。

3、java == 和 equals() 的区别?

对于基本数据类型 == 与equals 一致,都是比较值是否相等。
对于引用数据类型 == 比较的是引用是否相等, 而equals()方法是用来判断两个对象是否相等的。
对于equals()方法，它是Object类中的方法，它是用来判断两个对象是否相等的。
而对于引用数据类型，如果没有重写equals()方法，则默认比较的是引用是否相等，即 == 操作符与equals()方法的作用相同。

4、jdk、jre、jvm 之间的关系?

jdk：java development kit，java开发工具包，包含java编译器、类库、工具等。
jre：java runtime environment，java运行环境，包含java虚拟机。
jvm：java virtual machine，java虚拟机，是java程序的运行环境，负责执行java字节码。

jdk包含jre，jre包含jvm。

2025-02-11

5、java基础数据类型和包装类之间的区别?

基本数据类型：byte、short、int、long、float、double、char、boolean。
包装类：Byte、Short、Integer、Long、Float、Double、Character、Boolean。

- 包装类可以用于泛型，基本数据类型不可以。
- 包装类可以用于列表、集合、数组等容器类，基本数据类型不可以。
- 包装类未赋值时，默认为null ，基本数据类型有默认值。
- 包装类可以调用包装类方法，基本数据类型不可以。

6、java 基本数据类型变量存放位置?

基本数据类型的**局部变量**存放在**栈**中，基本数据类型的成员变量（未被static修饰的变量） 存放在java虚拟机的堆/方法区中/原空间中
大多数对象实例都存放在**堆**内存中,为什么说是大多数？ 由于java hotspot虚拟机 引入jit优化后，会对对象进行逃逸分析，如果发现某个对象没有逃逸到方法外部，
则会通过标量替换优化, 存放在栈中，节省内存，而非堆内存。

引用类型变量存放在**栈**中，存储的是 对象实例在堆内存中的地址。

例如：
```
int i = 10;
Integer integer = new Integer(10);
其中 i是基本数据类型 存放在栈中；
interger是引用类型 存放在栈中，存储的是Integer对象实例在堆内存中的地址。

对于Integer a= new Integer(10);
引用变量 vs 对象
引用变量：
a 是一个引用变量，存储的是 Integer 对象在堆内存中的地址。
你可以把 a 理解为一个“指针”，指向堆内存中的某个对象。

对象：
new Integer(3) 创建了一个 Integer 对象，存储在堆内存中。
这个对象包含数据（值为 3）和方法（如 toString()）。

2. 为什么说 a 是一个 Integer 对象？
严格来说，a 是一个 引用变量，而不是对象本身。

但当我们说“a 是一个 Integer 对象”时，实际上是一种简化的说法，意思是：
a 引用了一个 Integer 对象，因此可以通过 a 访问该对象的属性和方法。

3. 方法调用的机制
在 Java 中，方法调用是通过对象（或引用变量）进行的。具体过程如下：

找到对象：
```

通过引用变量 a 找到堆内存中的 Integer 对象。

调用方法：
调用该对象的 toString() 方法。

4. 具体到 System.out.println(a)
System.out.println(a) 的执行过程：

参数传递：
将 a（引用变量）传递给 println 方法。

方法调用：
println 方法内部会调用 a.toString()。

找到对象：
通过 a 找到堆内存中的 Integer 对象。

执行方法：
调用该对象的 toString() 方法，返回字符串 "3"。

输出结果：
将 "3" 打印到控制台。

5. 为什么不是调用引用变量的方法？
引用变量本身只是一个地址，没有方法。

方法属于对象，因此需要通过引用变量找到对象，然后调用对象的方法。

7、java 包装类型的缓存机制?

java 包装类缓存机制：

- 缓存机制：为了提高性能，Java 包装类提供了缓存机制，当一个包装类对象被创建后，会被缓存起来，以便后续使用。
- Byte、Short、Integer、Long 包装类缓存区大小为 -128 到 127 之间的所有 Integer 值，缓存区大小为 2 的 31 次方。
- Character 包装类缓存区大小为 0 到 127 之间的缓存值
- Boolean 包装类缓存区大小为 true 和 false 两个值，缓存区大小为 2。
- Flaot、Double 包装类缓存区大小未设置，因此不会缓存。

8、java 重写 重载 区别?

重写：子类继承父类，可以对父类的方法进行重新定义，称为方法的覆盖（override）。

- 子父类之间存在继承关系，子类重写父类的方法，使得子类的方法具有父类的方法相同的功能，但又有自己的实现。
- 子类返回值小于等于父类
- 子类抛出异常小于等于父类
- 子类访问权限大于等于父类
  重载：同一个类中，可以有多个同名的方法，但是方法的参数列表必须不同，称为方法的重载（overload）。
- 同一个类中,多个同名方法，传入不同参数，实现不同的功能。

9、java 接口和抽象类的区别?

接口：接口是抽象方法的集合，不能创建对象，只能被实现。接口不能包含属性，只能包含方法。
抽象类：抽象类可以包含属性和方法，可以创建对象，也可以被继承。抽象类可以包含抽象方法和非抽象方法。

10、面向对象三大特性?

封装、继承、多态。

- 封装：是指将数据和操作数据的方法封装在一起，对外提供接口，隐藏内部的实现细节。
- 继承：是指一个类可以从另一个类继承其属性和方法，并可以增加新的属性和方法。
- 多态：是指一个类实例可以赋给一个基类变量，使得该变量调用基类的方法，实际运行的是子类的方法。

11、java 深拷贝和浅拷贝、引用拷贝

- 引用拷贝：
  引用拷贝是指将一个变量的地址赋值给另一个变量，因此，两个变量指向同一个对象。
- 浅拷贝：
  浅拷贝是指创建一个新的对象，并复制其基本数据类型的值，但是，如果该对象中还有对象，则只是复制了对象的引用，而不是对象本身。（新对象和原对象不等，但是对象的属性值是一样的）
- 深拷贝：
  深拷贝是指创建一个新的对象，并复制其基本数据类型的值，如果该对象中还有对象，则也会递归地复制所有对象的引用。

2025-02-12

12、object 类有哪些方法?

Object 类是所有类的父类，所有类都继承了Object类
Object 类中有一下方法

- equals(Object obj)：判断两个对象是否相等，默认比较两个对象的内存地址是否相同。
- hashCode()：返回对象的哈希码值，默认返回对象的内存地址。
- getClass()：返回对象的类对象。
- toString()：返回对象的字符串表示。
- clone()：创建并返回对象的一个副本。
- wait()：暂停当前线程的执行，让其他线程运行。
- notify()：唤醒一个正在等待当前对象的线程。
- notifyAll()：唤醒所有正在等待当前对象的线程。

13、java hashCode 和 equals 方法的作用?

- equals()：判断两个对象是否相等，默认比较两个对象的内存地址是否相同。
- hashCode()：返回对象的哈希码值，默认返回对象的内存地址。
  为什么要有 hashCode() 方法?
- 为了提高哈希表的效率，HashMap、Hashtable、HashSet 等容器类使用 hashCode() 方法来确定对象的存储位置。
- 为了保证 equals() 方法的正确性，当两个对象相等时，hashCode() 方法必须返回相同的值。

14、java 异常处理机制?

java 异常处理机制：

- try-catch-finally：异常处理的基本结构。
- throws：方法声明抛出异常。
- throw：抛出一个异常。
- Throwable：java.lang 包中 , 所有异常的父类。

java 异常层次结构
Throwable

- Error：表示 JVM 或者系统错误，如 StackOverflowError、OutOfMemoryError。
- Exception：表示程序运行过程中出现的一般性异常，如 NullPointerException、IndexOutOfBoundsException。

exception

- checked exception ： 必须处理的异常，如 IOException、SQLException。 不处理，编译器编译不通过
- unchecked exception ： 不需要处理的异常，如 RuntimeException及其子类 npe、数组越界、数字转化异常等。

2025-02-13

15、Throwable 类常用方法有哪些?

- getMessage()：返回异常的详细信息。
- getCause()：返回异常的原因。
- printStackTrace()：打印异常的堆栈信息。

16、finally 中的代码一定会执行吗?

不一定，如果实在try-catch中，直接手动停止线程，那么finally中的代码就不会执行。

- 程序所在的线程死亡。
- 关闭 CPU。

17、try-catch-finally 结构中，如果 catch 中抛出异常，finally 中的代码还会执行吗?

会

18、java 异常处理机制中，try-catch-finally 结构中，try中return语句后，finally 中的return代码还会执行吗?

会，try中return语句后 return的内容会被保存在一个本地变量中，finally 中的return代码会修改这个本地变量的值，然后再返回。

19、java 泛型？泛型的作用？ 泛型的原理？ 泛型的应用?

泛型：参数化的类型，根据使用时参数的传入正真的确认类型，提高代码灵活性和可读性
使用方式：泛型类，泛型接口，泛型方法
原理：编译器在编译时，会检查泛型类型，并将泛型类型擦除，然后再生成新的字节码文件，运行时，JVM 根据擦除后的类型信息，动态生成新的对象。
应用：泛型可以提高代码的复用性，减少代码的冗余，提高代码的可读性。

20、java 反射机制?

Java反射机制：
在程序运行时可以通过反射机制动态的获取到类的信息，同时还可以调用类中的一些方法和属性。
反射机制是指在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意方法和属性。

获取Class对象：

- Class.forName("类名")
- 对象.getClass()
- 类名.class
- ClassLoader.loadClass("类名")

21、java 注解?

注解：是Java 5.0 引入的新特性，它是附加在代码上的元数据，用于一些工具在编译、运行时进行解析和处理。
注解的作用：

- 编译时进行检查：如 @Override 注解可以检查是否重写了父类的方法。
- 代码分析：如 @Test 注解可以提高代码的可读性和可维护性。
- 运行时处理：如@Autowired 注解可以自动装配bean

都实现了Annotation接口

解析方式：

- 编译器直接扫描，如 @Override 注解，编译器会对注解进行解析，并生成相关的代码。
- 反射解析，如通过反射获取注解信息。 Spring的相关注解，通过反射获取。

22、java序列化和反序列化

java序列化和反序列化：

- 序列化：将对象转换为字节流的过程。
- 反序列化：将字节流转换为对象的过程。
- 序列化的作用：
    - 将对象保存到文件中，或者在网络中传输对象。
    - 将对象进行持久化存储。
    - 将对象进行远程调用。
- 反序列化的作用：
    - 将对象从文件中读取出来，或者从网络中接收对象。
    - 将对象进行远程调用。

23、序列化协议对应于 TCP/IP 4 层模型的哪一层?

```
应用层 √
传输层
网络层
网络接口层
```

在应用层，服务于应用程序，直接负责数据的解析和处理，而不涉及到底层数据的传输。

23.1 如果有些字段不想被序列化怎么办?
transient关键字的作用：阻止实例中那些用此关键字修饰的的变量序列化；当对象被反序列化时，被transient 修饰的变量值不会被持久化和恢复。
关于 transient 还有几点注意：

- transient 只能修饰变量，不能修饰类和方法。
- transient 修饰的变量，在反序列化后变量值将会被置成类型的默认值。例如，如果是修饰 int 类型，那么反序列后结果就是 0。
- static变量因为不属于任何对象(Object)，所以无论有没有 transient 关键字修饰，均不会被序列化

24、常用的序列化协议有哪些?

- Java 自带的序列化协议：Java 自带的序列化协议是 Java 语言自带的序列化协议，它是 Java 语言特有的序列化协议 。
- Hadoop 序列化协议：Hadoop 序列化协议是 Hadoop 框架自带的序列化协议 。
- Protobuf 序列化协议：Protobuf 序列化协议是 Google 开源的序列化协议 。
- Avro 序列化协议：Avro 序列化协议是 Apache 开源的序列化协议 。
- Thrift 序列化协议：Thrift 序列化协议是 Facebook 开源的序列化协议 。
- Kryo 序列化协议：Kryo 序列化协议是 Twitter 开源的序列化协议 。
- Fastjson 序列化协议：Fastjson 序列化协议是 Alibaba 开源的序列化协议 。
- Hessian 序列化协议：Hessian 序列化协议是 Hessian 开源的序列化协议 。

csdn 手动实现RPC中的序列化问题

25、代理模式

使用代理对象来代替对真实对象(real object)的访问，这样就可以在不修改原目标对象的前提下，提供额外的功能操作，扩展目标对象的功能，对方法进行增强。

26、代理模式的分类

- 静态代理：静态代理在使用时,需要定义接口或者父类,被代理对象与代理对象一起实现相同的接口或者是继承相同父类。
- JVM角度的话：在**编译**的时候就将接口类、实现类、代理类都生成了class字节码文件。代理类和目标类的关系在编译期就确定了。
    - 对目标对象的每个方法的增强都是手动完成的（后面会具体演示代码），非常不灵活（比如接口一旦新增加方法，目标对象和代理对象都要进行修改）且麻烦(需要对每个目标类都单独写一个代理类）。 实际应用场景非常非常少，日常开发几乎看不到使用静态代理的场景
    - 静态代理实现步骤:
        - 定义一个接口及其实现类；
        - 创建一个代理类同样实现这个接口将目标对象注入进代理类，然后在代理类的对应方法调用目标类中的对应方法。
        - 这样的话，我们就可以通过代理类屏蔽对目标对象的访问，并且可以在目标方法执行前后做一些自己想做的事情。

- 动态代理：动态代理是在实现阶段不用关心代理谁,而在运行阶段才指定代理哪一个对象。
- JVM角度的话：代理类的class文件是在**程序运行**时动态生成并加载至JVM中。

27、动态代理

- JDK 动态代理：
  - 实现方式：
    - 定义一个接口及其实现类；
    - 自定义 InvocationHandler 并重写invoke方法，在 invoke 方法中,以调用处理器类型为参数，利用反射机制得到动态代理类的构造函数；我们会调用原生方法（被代理类的方法）并自定义一些处理逻辑；
    - 通过 Proxy.newProxyInstance(ClassLoader loader,Class<?>[] interfaces,InvocationHandler h) 方法创建代理对象；
- CGlib 动态代理：
  - 实现方式：
    - 定义一个类；
    - 自定义 MethodInterceptor 并重写 intercept 方法，intercept 方法就是回调方法（自定义处理逻辑）；
    - 通过 Enhancer 类的 create() 创建代理对象；

JDK 动态代理只能代理实现了接口的类或者直接代理接口，而 CGLIB 可以代理未实现任何接口的类。 另外， CGLIB 动态代理是通过生成一个被代理类的子类来拦截被代理类的方法调用，因此不能代理声明为 final 类型的类和方法。就二者的效率来说，大部分情况都是 JDK 动态代理更优秀，随着 JDK 版本的升级，这个优势更加明显。

28、静态代理与动态代理的区别

- 静态代理：
  - 优点：
    - 可以做到在不修改目标对象的前提下扩展目标对象的功能。
    - 静态代理的实现没有侵入目标对象的代码。
  - 缺点：
    - 如果目标很多，则静态代理的实现会非常臃肿，非常不利于代码维护。
    - 一旦接口增加方法，目标对象与代理对象都要进行修改。这是非常麻烦的！
- 动态代理：
  - 优点：
    - 可以做到在不修改目标对象的前提下扩展目标对象的功能。
    - 动态代理的实现非常简洁。
  - 缺点：
    - 动态代理的实现有一个前提条件：目标对象必须实现了接口，如果没有实现接口则不能使用动态代理。
- 灵活性：动态代理更加灵活，不需要必须实现接口，可以直接代理实现类，并且可以不需要针对每个目标类都创建一个代理类。另外，静态代理中，接口一旦新增加方法，目标对象和代理对象都要进行修改，这是非常麻烦的！
- JVM 层面：静态代理在编译时就将接口、实现类、代理类这些都变成了一个个实际的 class 文件。而动态代理是在运行时动态生成类字节码，并加载到 JVM 中的。

29、BigDecimal

为了防止浮点数精度丢失，java 提供了 BigDecimal 类来进行精确计算。
BigDecimal 是不可变的、任意精度的有符号十进制数。

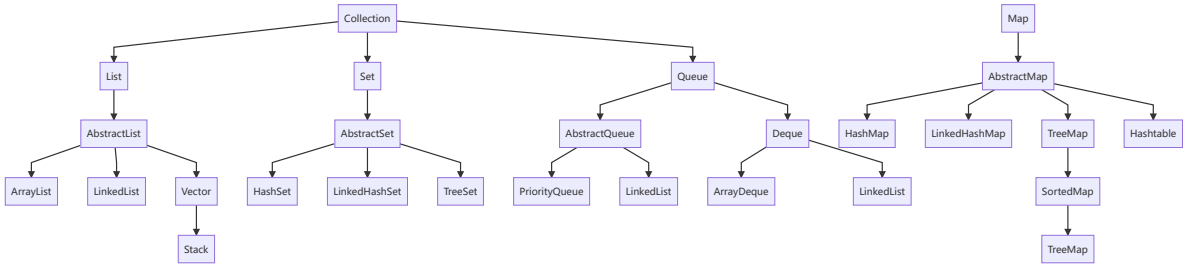
等值比较时 使用CompareTo方法 而不是equals方法。 equals方法不仅会比较值是否相等，还会比较精度（scale）。

第二章 集合相关知识点

30、集合框架

集合

- Collection 接口：
  - List 接口：
    - ArrayList：底层是数组，线程不安全，效率高。
    - LinkedList：底层是链表，线程不安全，效率高。
    - Vector：底层是数组，线程安全，效率低。
  - Set 接口：
    - HashSet：底层是哈希表，线程不安全，效率高。
    - LinkedHashSet：底层是链表和哈希表，线程不安全，效率高。
    - TreeSet：底层是红黑树，线程不安全，效率高。
  - Queue 接口：
    - PriorityQueue：底层是堆，线程不安全，效率高。
    - Deque 接口：
      - ArrayDeque：底层是数组，线程不安全，效率高。
      - LinkedList：底层是链表，线程不安全，效率高。
      - BlockingDeque 接口：
        - ArrayBlockingQueue：底层是数组，线程安全，效率高。
        - LinkedBlockingQueue：底层是链表，线程安全，效率高。
        - PriorityBlockingQueue：底层是堆，线程安全，效率高。
- Map 接口：
  - HashMap：底层是哈希表，线程不安全，效率高。
  - LinkedHashMap：底层是链表和哈希表，线程不安全，效率高。
  - TreeMap：底层是红黑树，线程不安全，效率高。
  - Hashtable：底层是哈希表，线程安全，效率低。
  - ConcurrentHashMap：底层是分段锁，线程安全，效率高。



以ArrayList为例子，ArrayList继承了AbstractList，实现了List接口，List接口继承了Collection接口，Collection接口继承了Iterable接口。

31、ArrayList 和 Array 的区别?

- ArrayList是一个泛型类,而Array是一个数组。 对于基本的数据类型，ArrayList要求存储类型转化为包装类进行存储，而Array可以直接存储基本数据类型。
- ArrayList是一个动态数组,而Array是一个静态数组。 支持自动扩容,而Array不支持自动扩容。
- ArrayList是一个可变长度的数组,而Array是一个固定长度的数组。
  提供add,remove等方法，进行动态添加和删除元素。而Array不提供这些方法，只有进行下标遍历进行添加和删除元素。

31、ArrayList 和 LinkedList 的区别?

- ArrayList是一个动态数组,而LinkedList是一个双向链表。
- ArrayList，LinkedList 是线程不安全的集合。
- ArrayList的查找速度快,而LinkedList的查找速度慢。
- ArrayList 实现了 RandomAccess 接口 支持随机访问,而LinkedList不支持随机访问。

32、ArrayList 和 Vector 的区别?

- ArrayList是List的实现类，vector也是List的实现类，但是Vector是比较古老的实现类，现在已经很少使用了。
- ArrayList是线程不安全的，而Vector是线程安全的。

33、ArrayList linkedList 插入元素时间复杂度?

- ArrayList:
  -
  当Arraylist插入该元素后的容量正好大于当前数组的长度时，ArrayList会进行自动扩容，需要将原有数组中的内容复制到新数组中，时间复杂度为O(
  n),然后将插入的元素放到新数组的最后，时间复杂度为O(1)
    - 当ArrayList插入元素后的容量小于当前数组的长度时，只需要再最后插入该元素，时间复杂度为O(1)
    - 当数组中间某个位置插入元素，或者数组头部插入某个元素时，需要将后面的元素统一向后挪动一位，时间复杂度为O(n)
- LinkedList:
    - 当LinkedList插入元素时，只需要将该元素放到链表的最后，时间复杂度为O(1)
    - 当链表中间某个位置插入元素，时间复杂度为O(n)

34、RandomAccess 接口的作用及使用?

RandomAccess 接口是List的子接口，他表示List能够通过下标直接访问对应下标位置的元素，是个标记接口，只用做标记，接口中无内容；实现这个接口的list类，表示可以被随机访问元素

使用：
在使用Collections类的二分查找binarySearch方法时，会根据传入的list是否实现了RandomAccess接口来决定使用二分查找还是线性查找。
如果传入的list实现了RandomAccess接口，则使用二分查找；如果没有实现，则使用线性查找。

源码：

```
public static<T>
int binarySearch(List<?extends Comparable<? super T>> list,T key){
    if(list instanceof RandomAccess||list.size()<BINARYSEARCH_THRESHOLD)
    return Collections.indexedBinarySearch(list,key);
    else
    return Collections.iteratorBinarySearch(list,key);
    }
```

ArrayList实现了RandomAccess接口，而LinkedList没有实现。
至于LinkedList为什么不实现RandomAccess接口，我猜测还是因为底层是链表的关系，底层是链表，不支持随机访问，所以没有实现RandomAccess接口。

35、ArrayList的扩容机制

以JDK1.8为例子进行分析：

- 与构造函数的初始容量有关
    - ArrayList()： 无参构造函数，初始化的ArrayList容量为0，当第一次添加元素时，容量扩充至**10**.
    - ArrayList(int initialCapacity)： 有参构造函数，初始化的ArrayList容量为initialCapacity。
    - ArrayList(Collection<? extends E> c)： 构造函数，初始化的ArrayList容量为c的大小。
- add方法
    - 在add方法中时，先调用ensureCapacityInternal,进入ensureExplicitCapacity方法中，
    - 比较minCapacity - elementData.length > 0，如果大于0，则需要进行扩容。
    - 调用grow方法，扩容后的容量为oldCapacity + (oldCapacity >> 1)，oldCapacity 右移一位，其效果相当于oldCapacity

/2，oldCapacity 为偶数就是 1.5 倍，否则是 1.5 倍左右。
    - 调用Arrays.copyOf方法，将原来的数组复制到新的数组中。
    - 如果新容量>
      MAX_ARRAY_SIZE,则调用hugeCapacity方法，进行容量的判断。如果要添加的元素数量大于MAX_ARRAY_SIZE，则新容量为Integer.MAX_VALUE，否则新容量为MAX_ARRAY_SIZE（Integer.MaxValue-8）。

36、集合中的fail-fast 和 fail-safe 区别?

java.util包下的集合类大部分都不支持线程安全，因此在集合操作的时候提供了modCount变量进行修改次数的记录，在迭代期间通过与expectedModCount进行比较，比较是否一致来判断是否进行了并发操作

- fail-fast: 快速失败，在集合在迭代的过程中，集合结构发生变化的时候，对针对可能发生的异常进行提前表明故障并停止运行，会抛出ConcurrentModificationException异常。
- fail-safe: 安全失败，在集合迭代的过程中，集合结构发生变化的时候，会复制一份新的集合，对新的集合进行迭代，不会抛出ConcurrentModificationException异常。
    - fail-safe 优先进行复制，在复制的数组上进行操作，完成后在通过指针指向新的数组。

37、set

set 是一个接口，继承自Collection接口，

- set 中不允许有重复的元素，是指元素内容相同，在调用equals方法时返回false, 需要重写equals和hashCode方法
- set 中元素的顺序是无序的,无序表示的是底层中数据存储不是按照数组一样顺序存储的，是按照hash值计算后进行存储的

38、hashset、linkedHashSet、TreeSet 区别?

- 都实现了Set接口
- 都线程不安全
- hashset底层是hashmap实现的
- linkedHashSet底层是链表+hashMap实现的
- TreeSet底层是红黑树实现的

39、hashmap的实现原理?

- JDK1.8 之前:
    - hashmap 底层数据存储结构在JDK1.8之前数是数组+链表的结构
    - hashmap的key通过hashCode方式进行扰动函数处理散列之后，得到hash值，然后通过（n-1）& hash 的计算方式，计算出数组下标，如果当前下标数据存在，
      比较该位置元素的key和hash是否相等，如果相等就进行覆盖，如果不相等，就通过链表的方式进行存储。
- JDK1.8 :
    - hashmap底层数据存储结构变化比较明显，在JDK1.8之后，hashmap底层数据存储结构是数组+链表+红黑树的结构
    - **链表长度>8 且 数组长度超过64**，链表会转化为红黑树。如果数组长度小于64，即使链表长度>8,链表不会转化为红黑树，而是会触发数组扩容。
    - **链表长度<6时**，红黑树会转化为链表。

    - **数组长度默认是16**，当使用率达到0.75时，数组会进行扩容，创建一个新的数组，数组长度为原来数组长度的两倍
        - 进行重新计算各个元素的hash值，然后通过（n-1）& hash 的计算方式，计算出数组下标，然后重新进行存储。

40、queue 和 deque 的区别

- queue是一个接口，继承自collection接口，是一个单端队列，只允许在一端进行插入操作，为了处理长度溢出的情况，提供了两组使用的方法
    - 添加元素：
        - add:添加元素，当队列满时，抛出异常
        - offer:添加元素，当队列满时，返回false
    - 获取队首元素(不删除-只做查询)：
        - element:获取元素，当队列为空时，抛出异常 NoSuchElementException
        - peek:获取元素，当队列为空时，返回null
    - 获取队首元素(获取并出队列)：
        - remove:获取元素，当队列为空时，抛出异常 NoSuchElementException
        - poll:获取元素，当队列为空时，返回null
- deque是一个接口，继承自queue接口，拓展了queue 的单端插入 是一个双端队列，允许在两端进行插入和删除操作，提供了两组使用的方法
    - 队首插入：
        - addFirst:添加元素，当队列满时，抛出异常
        - offerFirst:添加元素，当队列满时，返回false
    - 队尾插入：
        - addLast:添加元素，当队列满时，抛出异常
        - offerLast:添加元素，当队列满时，返回false
    - 队首获取(不删除-只做查询)：
        - getFirst:获取元素，当队列为空时，抛出异常 NoSuchElementException
        - peekFirst:获取元素，当队列为空时，返回null
    - 队尾获取(不删除-只做查询)：
        - getLast:获取元素，当队列为空时，抛出异常 NoSuchElementException
        - peekLast:获取元素，当队列为空时，返回null
    - 队首获取(获取并出队列)：
        - removeFirst:获取元素，当队列为空时，抛出异常 NoSuchElementException
        - pollFirst:获取元素，当队列为空时，返回null
    - 队尾获取(获取并出队列)：
        - removeLast:获取元素，当队列为空时，抛出异常 NoSuchElementException
        - pollLast:获取元素，当队列为空时，返回null

41、ArrayDeque 和 linkedList 有什么区别?

- ArrayDeque 和 linkedList都实现了Deque接口，都可以做队列使用
- ArrayDeque 底层是动态数组和双指针来实现的，linkedList 底层是链表
- ArrayDeque 不支持null值，linkedList 支持null值
- ArrayDeque 插入时存在扩容操作，linkedList 插入时总是要申请新的堆空间
- ArrayDeque 提供了pop和push方法 可以当栈使用

42、priorityQueue 优先队列

- 优先队列是一个无界队列，底层是数组实现的，但是逻辑上时一颗满二叉树，满足堆的性质，数组下标符合满二叉树的性质：当前节点arr[i] 子二叉树的左右节点 arr[2*i+1] arr[2*i+2] ;父节点 arr[(i-1)/2]
- 优先队列的元素按照优先级进行排序，优先级高的元素排在前面，优先级低的元素排在后面。
- 优先队列的元素可以是任何类型，但是必须实现Comparable接口，或者在构造函数中传入Comparator接口的实现类。
- 优先队列的元素可以重复，但是重复的元素按照优先级进行排序。
- 优先队列的元素不可以为null
- 线程不安全，如果需要线程安全，可以使用PriorityBlockingQueue
- 扩容机制
    - 默认创建容量为**11**的数组
    - 如果当前容量小于64，每次扩容为原来的长度的2倍+2
    - 如果当前容量大于64，每次扩容为原来的1.5倍
    - 如果扩容后的容量大于MAX_ARRAY_SIZE，每次扩容为Integer.MAX_VALUE

43、blockingQueue?

- 阻塞队列是一个接口，继承自Queue接口，同样是一个双端队列
- 阻塞队列是一个线程安全的队列
- 常用的阻塞队列 实现类
    - ArrayBlockingQueue：底层是数组实现的，是一个有界队列
    - LinkedBlockingQueue：底层是链表实现的，是一个无界队列
    - PriorityBlockingQueue：底层是堆实现的，是一个无界队列
    - DelayQueue：底层是堆实现的，是一个无界队列，元素必须实现Delayed接口，元素只有在达到指定的延迟时间后才能从队列中获取

44、ArrayBlockingQueue和LinkedBlockingQueue的区别?

- ArrayBlockingQueue底层是数组，LinkedBlockingQueue底层是链表
- ArrayBlockingQueue是一个有界的队列即初始化的时候就得赋予容量大小，linkedBlockingQueue是一个无界的队列，但是如果在初始化的时候指定大小，同样会变成有界队列
- ArrayBlockingQueue在空间的占用上是一次性分配空间，往往申请的空间得大于实际使用的空间，而LinkedBlockingQueue只有在添加元素时，才会分配堆内存空间

45、map (**)

46、java 垃圾回收机制?

java