Java 基础

1. java 面向对象有哪些特征

封装：把一个对象的状态信息（也就是属性）隐藏在对象内部，不允许外部对象直接访问对象的内部信息。但是可以提供一些可以被外界访问的方法来操作属性。

继承：让某个类型的对象获得另一个类型的对象的属性的方法。继承就是子类继承父类的特征和行为，使得子类对象（实例）具有父类的属性和方法，子类也可以对父类进行扩展，增加了代码的复用性。

多态：对于同一个行为，不同的子类对象具有不同的表现形式。多态存在的 3 个条件：1）继承；2）重写；3）父类引用指向子类对象。增加了代码的可移植性、健壮性、灵活性.

2. 访问修饰符 public、private、protected，以及不写时的区别

访问修饰符，用于控制方法和属性（成员变量）的访问权限

修饰符	同类	同包	子类	不同包/其它包
public	√	√	√	√
protected	√	√	√	×
不写（不是 default）	√	√	×	×
private	√	×	×	×

扩展：被 private 修饰的成员，只能在本类进行访问，针对 private 修饰的成员变量，如果需要被其他类使用，提供相应的操作

- 提供“get 变量名()”方法，用于获取成员变量的值，方法用 public 修饰

- 提供“set 变量名(参数)”方法，用于设置成员变量的值，方法用 public 修饰/

3. final 关键字用法

- final 修饰的类叫最终类，该类不能被继承。
- final 修饰的方法不能被重写
- final 修饰的变量叫常量，常量必须初始化，初始化之后值就不能被修改。

4. this、super 关键字

this 代表所在类的当前对象的引用（地址值），即代表当前对象。

super 代表的是父类对象的引用，this 代表的是当前对象的引用。

- super()和 this()类似,区别是，super()在子类中调用父类的构造方法，this()在本类内调用本类的其它构造方法。
- this()和 super()都指的是对象，所以，均不可以在 static 环境中使用。包括：static 变量,static 方法，static 语句块。
- super()和 this()均需放在构造方法内第一行。

5. static 关键字

1. 可以用来修饰成员变量和成员方法，该变量称为**静态变量**，该方法称为**静态方法。**
2. 无 static 修饰的成员变量或者成员方法，称为**实例变量，实例方法**
3. static 修饰的成员属于类，会存储在静态区，是随着类的加载而加载的，且只加载一次，所以只有一份，节省内存。
4. 静态代码块：只会在类加载的时候执行一次

注：被 static 修饰的变量或者方法**不属于任何一个实例对象，而是被类的实例对象所共享。**

6. 重写、重载

两者区别：

重载（Overload）：在一个类中，同名的方法有不同的参数列表（参数个数、顺序、类型不同），则视为重载，与返回类型无关。

重写（**Override**）：发生在父类子类中，若子类方法想要和父类方法构成重写关系，则它的方法名、参数列表必须与父类方法相同。另外，返回值要小于等于父类方法，抛出的异常要小于等于父类方法，访问修饰符则要大于等于父类方法。

7. ==和 equals()有什么区别?

==运算符：

- 作用于基本数据类型时，是比较两个数值是否相等；
- 作用于引用数据类型时，是比较两个对象的内存地址是否相同（引用地址）；

equals()方法：

- 没有重写时，Object 默认以 == 来实现，即比较两个对象的内存地址是否相同；
- 进行重写后，一般会按照对象的内容来进行比较，若两个对象内容相同则认为对象相等，否则认为对象不等。

equals 本质上就是 ==，只不过 String 和 Integer 等重写了 equals 方法，把引用比较改成了值比较。

8. String 创建字符串对象两种方式的区别：

- 通过构造方法创建（public String()）

  通过 new 创建的字符串对象，每一次 new 都会申请一个内存空间，虽然内容相同，但是地址值不同

- 直接赋值方式创建（String s = “abc”;）

以" "方式给出的字符串，只要字符序列相同(顺序和大小写)，无论在程序代码中出现几次，JVM 都只会建立一个 String 对象，并在字符串池中维护

9. String 常用方法

- char charAt(int index)：返回指定索引处的字符；
- String substring(int beginIndex, int endIndex)：截取字符串
- String trim()：删除字符串两端的空格；
- int indexOf(String str)：返回子串在此字符串首次出现的索引；
- equals()：字符串比较。
- length()：返回字符串长度。
- indexOf()：返回指定字符的索引。

注：String 类由 final 修饰，所以不能被继承。

10. 字符串常量池的作用了解吗?

字符串常量池 是 JVM 为了提升性能和减少内存消耗针对字符串（String 类）专门开辟的一块区域，主要目的是为了避免字符串的重复创建

11. String str="i"与 String str=new String(“i”)一样吗?

不一样，因为内存的分配方式不一样。String str="i"的方式，Java 虚拟机会将其分配到常量池中；而 String str=new String(“i”) 则会被分到堆内存中。

12. String、StringBuffer 和 StringBuilder 的异同?

1. String 对象一旦创建，其值是不能修改的（被 final 关键字修饰），如果要修改，会重新开辟内存空间来存储修改之后的对象；而 StringBuffer 和 StringBuilder 对象的值是可以被修改的；

2. StringBuffer 几乎所有的方法都使用 synchronized 实现了同步，线程比较安全，在多线程系统中可以保证数据同步，但是效率比较低；而 StringBuilder 没有实现同步，线程不安全，在多线程系统中不能使用 StringBuilder，但是效率比较高。

3. 开发过程中如果频繁修改，不要使用 String，否则会造成内存空间的浪费；当需要考虑线程安全的场景下使用 StringBuffer，如果不需要考虑线程安全，追求效率的场景下可以使用 StringBuilder。

13. JDK 和 JRE 有什么区别?

- JDK：Java Development Kit 的简称，Java 开发工具包，提供了 Java 的开发环境和运行环境。
- JRE：Java Runtime Environment 的简称，Java 运行环境，为 Java 的运行提供了所需环境。
- JVM(Java Virtual Machine)，Java 虚拟机，是 JRE 的一部分，它是整个 java 实现跨平台的最核心的部分，负责运行字节码文件

具体来说 JDK 其实包含了 JRE，同时还包含了编译 Java 源码的编译器 Javac（编译成字节码文件），还包含了很多 Java 程序调试和分析的工具。简单来说：如果你需要运行 Java 程序，只需安装 JRE 就可以了，如果你需要编写 Java 程序，需要安装 JDK。

JDK 中包含了 JRE，JRE 中包含了 JVM。

14. 抽象类相关

1. 抽象类不一定非要有抽象方法，但是有抽象方法的类必定是抽象类
2. 抽象类中不能使用 final 修饰
3. 抽象类不能创建对象(抽象方法没有方法体,无法执行)
4. 抽象类存在的意义是为了被继承，否则抽象类将失去意义

抽象类能使用 final 修饰吗?

不能，定义抽象类就是让其他类继承的，如果定义为 final 该类就不能被继承，这样彼此就会产生矛盾，所以 final 不能修饰抽象类。

15. 接口相关

1. 接口中的抽象方法默认会加上 public abstract 修饰，成员变量默认会加上 public static final 修饰
2. 使用 interface 修饰，不能实例化，类可以实现多个接口
3. 接口可以继承接口，并可多继承接口，但类只能单继承

16. 抽象类和接口的区别

相同点：抽象类和接口都不能直接实例化（创建对象）

不同点：抽象类只能单继承和实现多个接口，接口可以多继承接口

抽象类中可以有构造方法，接口中不能有构造方法

关键字不同：抽象类子类使用 extends 关键字来继承抽象类，接口实现类使用关键字 implements 来实现接口；

17. 两个对象的 hashCode() 相同，则 equals() 也一定为 true，对吗?

不对，两个对象的 hashCode() 相同，equals() 不一定 true。

- 如果两个对象的 hashCode 不相同，那么这两个对象肯定是不同的两个对象
- 如果两个对象的 hashCode 相同，不代表这两个对象一定是同一个对象，也可能是两个对象
- 如果两个对象相等，那么他们的 hashCode 就一定相同

在 Java 的一些集合类的实现中，比较两个对象是否相等时，会根据上面的原则，会先调用对象 hashCode()方法得到 hashCode 进行比较，如果 hashCode 不相同，就可以直接认为这两个对象不相同，如果 hashCode 相同，那么就会进一步调用 equals()方法进行比较。而 equals()方法，就是用来最终确定两个对象是不是相等的。

18. 为什么重写 equals() 时必须重写 hashCode() 方法?

因为两个相等的对象的 `hashCode` 值必须是相等。也就是说如果 `equals` 方法判断两个对象是相等的，那这两个对象的 `hashCode` 值也要相等。

如果重写 `equals()` 时没有重写 `hashCode()` 方法的话就可能会导致 `equals` 方法判断是相等的两个对象，`hashCode` 值却不相等。

注：如果重写了 equals()方法，那么就要注意 hashCode()方法，一定要保证能遵守上述规则。

19. Java 数据类型及所占字节

基本数据类型有 8 种:

- 整数类型 byte、short、int、long
- 浮点类型 float、double
- 字符型 char
- 布尔型 boolean(true、false)

引用数据类型（类、接口、数组）

1 字节=8 位

数据类型	大小
1byte	8bit
1short	2byte

1int	4byte
1long	8byte
1float	4byte
1double	8byte
1char	2byte

一个中文字符占 2 个字节

注意：char 类型、一个中文字符在 java 中占 2byte，在其它语言中占 1byte，因为 Java 是 Unicode 编码，在 Unicode 编码中 1char == 2byte，在 GBK /gb2312 编码中占 2 字节，但是 utf-8 编码中占 3 字节，ASSII 编码，一个字符占 1 个字节

20. continue、break 和 return 的区别是什么?

1. `continue` ：指跳出当前的这一次循环，继续下一次循环。
2. `break` ：指跳出整个循环体，继续执行循环下面的语句。
3. `return` 用于跳出所在方法，结束该方法的运行。

21. 包装类相关、自动装拆箱

装箱：将基本类型用它们对应的引用类型包装起来；

拆箱：将包装类型转换为基本数据类型；

为了能够将基本数据类型当成对象操作，Java 为每一个基本数据类型都引入了对应的包装类型（wrapper class），int 的包装类就是 Integer，从 Java 5 开始引入了自动装箱/拆箱机制，使得二者可以相互转换。

Java 为每个原始类型提供了包装类型：

原始类型: boolean，char，byte，short，int，long，float，double

包装类型：Boolean，Character，Byte，Short，Integer，Long，Float，Double

包装类型的缓存机制

装箱拆箱：基本类型要转换成包装类型，需要调用包装类型的静态 valueOf()函数；

而包装类型要转换成基本类型，需要调用以基本类型开头的 Value()，比如

intValue() 。

```java
//Java 会自动帮你实现拆箱装箱
Integer x = 2;     // 装箱 相当于是 Integer x = Integer.valueOf(2)
int y = x;         // 拆箱 实际是 int y = intValue(x)
```

使用 valueOf()方法时，系统将会判断数值是否存在于缓存池中。如果在就直接返回缓存池中的对象，如果不存在会直接返回一个 new 出来的新的对象。

22. java 中 IO 流分为几种?

按照流的流向分，可以分为输入流和输出流；

按照操作单元划分，可以划分为字节流和字符流；

按照流的角色划分为节点流和处理流。

Java Io 流共涉及 40 多个类，这些类看上去很杂乱，但实际上很有规则，而且彼此之间存在非常紧密的联系，Java I0 流的 40 多个类都是从如下 4 个抽象类基类中派生出来的。

InputStream/Reader: 所有的输入流的基类，前者是字节输入流，后者是字符输入流。

OutputStream/Writer: 所有输出流的基类，前者是字节输出流，后者是字符输出流。

23. Files 的常用方法都有哪些?

Files. exists()：检测文件路径是否存在。

Files. createFile()：创建文件。

Files. createDirectory()：创建文件夹。

Files. delete()：删除一个文件或目录。

Files. copy()：复制文件。

Files. move()：移动文件。

Files. size()：查看文件个数。

Files. read()：读取文件。

Files. write()：写入文件。

24. 面向对象和面向过程的区别

解决问题的方式不同

- 面向过程把解决问题的过程拆成一个个方法，通过一个个方法的执行解决问题。
- 面向对象会先抽象出对象，然后用对象执行方法的方式解决问题。

25. 深拷贝和浅拷贝区别了解吗？（clone）

浅拷贝：拷贝基本数据类型的值以及引用数据类型的引用地址，不会复制一份引用地址所指向的对象，内部的属性指向的是同一个对象

深拷贝：拷贝基本数据类型的值，会复制一份引用地址所指向的对象，深拷贝出来的对象，内部的属性指向的不是同一个对象。

26. Java 异常体系

Java 中，所有的异常都来自顶级父类 Throwable 类，Throwable 类有两个子类 **Exception** 和 **Error**

- **Exception** :程序本身可以处理的异常，可以通过 catch 来进行捕获

  运行时异常：可以正常通过编译

1) NullPointerException 空指针异常

2) ArithmeticException 数学运算异常

3) ArrayIndexOutOfBoundsException 数组下标越界异常

4) ClassCastException 类型转换异常

5) NumberFormatException 数字格式不正确异常[]

编译异常：在编译过程，就必须处理的异常，否则代码不能通过编译

SQLException //操作数据库时，查询表可能发生异常

IOException //操作文件时，发生的异常异常

FileNotFoundException //当操作一个不存在的文件时，发生异常

ClassNotFoundException//加载类,而该类不存在时，异常

EOFException//操作文件，到文件末尾，发生异常

FileNotFoundException/当操作一个不存在的文件时，发生异常

lllegalArguementException//参数异常

- **Error** ： Error 属于程序无法处理的错误，严重错误，例如 Java 虚拟机运行错误（Virtual MachineError）、虚拟机内存不够错误(OutOfMemoryError)、类定义错误（NoClassDefFoundError）等

27. JAVA 的异常捕获机制

Java 的异常捕获机制可以用 try-catch-finally 语句块来实现。通常情况下，当程序运行过程中出现异常时，程序将停止执行并输出异常信息。为了让程序能够在出现异常的情况下继续执行，可以使用 try-catch-finally 语句块来捕获异常并处理它们。

```
try {
    // 可能会产生异常的代码
} catch (ExceptionType e) {
```

```
    // 处理异常的代码
} finally {
    // 不管有没有异常都会执行的代码
}
```

Java

try 块中包含可能会产生异常的代码，如果在执行 try 块中的代码时出现了异常，就会立即跳转到 catch 块中处理异常。catch 块中的代码将根据异常类型进行处理，例如输出异常信息、记录日志、抛出新的异常等等。finally 块中的代码不管 try 块中是否发生异常都会执行，通常用于释放资源或清理工作。

如果 try 块中发生了多个异常，可以使用多个 catch 块来分别处理不同类型的异常，例如：

```
try {
    // 可能会产生异常的代码
} catch (IOException e) {
    // 处理 IO 异常的代码
} catch (SQLException e) {
    // 处理数据库异常的代码
} catch (Exception e) {
    // 处理其他类型异常的代码
} finally {
    // 不管有没有异常都会执行的代码
}
```

Java

28. 什么是反射机制?

JAVA 反射机制是在运行状态中，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一个方法和属性；这种动态获取的信息以及动态调用对象的方法的功能称为 java 语言的反射机制。

29. 反射机制优缺点

优点： 运行期类型的判断，动态加载类，提高代码灵活度。

缺点： 性能瓶颈：反射相当于一系列解释操作，通知 JVM 要做的事情，性能比直接的 java 代码要慢很多。

30. 反射的应用场景?

Spring/Spring Boot、MyBatis 等框架设计、动态代理设计模式也采用了反射机制

31. Java 中创建对象的方式

1. new
2. clone
3. 通过反射

   使用 Class 对象的 newInstance()方法来创建该 Class 对象对应类的实例，而执行 newInstance()方法时实际上是利用默认构造器来创建该类的实例。

   通过调用构造器 Constructor 再去创建对象 Constructor.newInstance，可以选择使用指定构造器来创建实例
4. 反序列化

   序列化：指把 **Java** 对象转换为字节序列的过程；

   反序列化：指把字节序列恢复为 **Java** 对象的过程；

32. java 中 # 和 $ 的区别

#{} 表示一个占位符，可以防止 SQL 注入

${} 表示 SQL 的拼接

33. 数组和链表的区别：

（1）存储形式：数组在内存中是连续的一段空间，声明时就要确定长度。链表是一块可不连续的动态空间

（2）插入和删除元素：在数组中插入和删除元素会涉及到移动其它元素的操作；链表的节点之间通过指针相连，操作比数组效率更高。

（3）修改查找元素：数组可以通过下标直接访问，链表需要从头开始遍历查找。数组便于查询，链表便于插入删除。

34. 数组增删查的复杂度，链表增删查的复杂度

- 数组：

  查、改，通过下标检索，O(1)

  增：最好（末尾增加）：O(1)、最坏（头部增加）：O(n)、平均复杂度：O(n)

  删：和增加一样

- 链表：

  查、改：平均复杂度 O(n)，遍历找到元素的位置，然后更改节点存放的值。

  增：平均复杂度 O(n)

  删：和增一样

35. java8 新特性：

Lamdba 表达式、Stream API

36. 内存溢出和内存泄露的区别

- 内存泄露是指程序未能正确释放不再使用的内存资源，导致内存逐渐耗尽，影响系统性能。
- 内存溢出是指程序试图分配超过可用内存大小的内存空间，导致分配失败或崩溃。

37. 栈和队列的区别

- 栈适用于需要"后进先出"的场景，如递归、回溯等。
- 队列适用于需要"先进先出"的场景，如任务调度、缓冲区等。

Java 集合 容器

1. java 集合（容器）概述

(1) Collection (单列集合)

- Set: HashSet、TreeSet(无序)
- List: ArrayList、LinkedList、Vector(有序)

(2) Map (双列集合) K-V

- HashMap、HashTable、TreeMap

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

2. List, Set, Queue, Map 区别?

- Set 代表无序的，元素不可重复的集合；
- List 代表有序的，元素可以重复的集合；
- Queue 代表先进先出（FIFO）的队列，存储的元素是有序的、可重复的。
- Map 代表具有映射关系（key-value）的集合；key 是无序的、不可重复的，value 是无序的、可重复的，每个键最多映射到一个值。

3. List 接口介绍

1. List 集合类中元素**有序**(即添加顺序和取出顺序一致)、且**可重复**
2. List 集合中的每个元素都有其对应的顺序索引，即支持索引

4. ArrayList 扩容机制

（1）ArrayList 中维护了一个 Object 类型的数组 elementData，transient Object[] elementData

（2）扩容机制：当创建 ArrayList 对象，如果使用无参构造器，初始 elementData 容量为 0，当第一次添加元素时，则扩容 elementData 容量为 10，如需再次扩容，则扩容 elementData 为 1.5 倍

（3）如果使用指定大小的构造器，则初始 elementData 容量为指定大小，如果需扩容，则直接扩容 elementData 为 1.5 倍

5. ArrayList 和 Vector 区别

	线程安全（同步）效率	扩容倍数
ArrayList	不安全，效率高	无参构造：第一次 10，第二次 1.5 倍；有参构造 1.5 倍
Vector	安全，效率不高	无参构造：默认 10，满后 2 倍扩容；有参构造 2 倍扩容

（1）线程安全方面：ArrayList 线程不安全，效率高；Vector 线程安全，效率不高；

（2）扩容方面：ArrayList 和 Vector 都会根据实际的需要动态的调整容量，只不过在 Vector 扩容每次会增加 1 倍，而 ArrayList 只会增加 50%。

6. ArrayList 和 LinkedList 区别

ArrayList 底层是一个动态数组，查改效率较高；

LinkedList 底层是一个双向链表，增删效率较高；

（1）数据结构实现：ArrayList 底层是一个动态数组，而 LinkedList 底层是一个双向链表。

（2）增删效率：LinkedList 要比 ArrayList 效率要高，因为 ArrayList 增删操作要影响数组内的其他数据的下标。

（3）改查效率： ArrayList 比 LinkedList 效率高，因为 LinkedList 是线性的数据存储方式，所以需要移动指针从前往后依次查找。

7. Set 接口

- **无序(添加和取出的顺序不一致)，没有索引**
- **不允许重复**元素，所以最多包含一个 null

注意：取出的顺序的顺序虽然不是添加的顺序，但是他的固定.

8. HashSet

(1) 基本介绍

- HashSet 实现了 Set 接口
- HashSet 底层基于 HashMap 实现的
- 可以存放 null 值，但是只能有一个
- 不能有重复对象

(2) HashSet 添加过程

HashSet 底层是 HashMap, HashMap 底层是(数组+ 链表+红黑树)

1. 首次扩容:

   先判断数组是否为空，若数组为空则进行第一次扩容（resize）；

2. 计算索引:

   通过 hash 算法，计算键值对在数组中的索引;

3. 插入数据:

   如果当前位置元素为空，则直接插入数据；

   如果当前位置元素非空，且 key 已存在，则直接覆盖其 value；

如果当前位置元素非空，且 key 不存在，则将数据链到链表末端；

若链表长度达到 8，则将链表转换成红黑树，并将数据插入树中；

4. 再次扩容

如果数组中元素个数（size）超过 threshold，则再次进行扩容操作。

9. HashMap 和 HashTable 区别

（1）HashMap 线程不安全，效率高；Hashtable 线程安全，效率低。

（2）HashMap 可以存储 null 的 key 和 value，但 null 作为键只能有一个，null 作为值可以有多个；Hashtable 不允许有 null 键和 null 值，否则会抛出 NullPointerException。

10. HashMap 和 HashSet 区别

HashSet 底层就是基于 HashMap 实现的，HashSet 调用 add()方法向 Set 中添加元素，其实就是调用 HashMap 的 put()方法向 map 中添加元素

11. HashMap 和 TreeMap 区别

相比于 HashMap 来说 TreeMap 主要多了对集合中的元素根据键排序的能力以及对集合内元素的搜索的能力。TreeMap 适合对一个有序的 key 集合进行遍历。

12. HashMap (HashSet)底层实现（HashMap 的 Put 方法）

HashMap 基于 Hash 算法实现的

1. 往 HashMap 中 put 元素时，根据 key 通过 hash 算法与与运算，计算当前对象的元素在数组中的下标;

2. 存储元素时，

- 如果当前下标位置元素为空，则直接插入数据；

- 如果当前下标位置元素非空，首先判断当前下标位置上的 node 类型，看是红黑树还是链表

  如果 key 已存在，则直接覆盖原始值 value；

  如果 key 不存在，则将数据添加到红黑树中或者链到链表末端；

3. 在添加元素的过程中，若链表长度达到 8，并且 table 的大小>=64，则将链表转换成红黑树，并将数据插入树中；

13 为什么要右移 16 位?

HashMap 方法 put 元素时，通过传入的值获取 hashcode 与 hashcode 的无符号右移 16 位，再做异或运算，为了减少 hash 碰撞，让输入值分布的更加均匀。

14. HashSet 如何检查重复?

不仅要比较 hash 值，同时还要结合 equals 方法比较。

当加入对象时，HashSet 会先计算对象的 hashcode 值来判断对象加入的位置，同时也会与其他加入的对象的 hash 值作比较，如果没有相符的 hash，HashSet 会假设对象没有重复出现。但是如果发现有相同 hash 值的对象，这时会调用 equals()方法来检查 hash 值 相等的对象是否真的相同。如果两者相同，HashSet 就放弃添加，不相同就添加到最后。

15. HashSet 的扩容机制

（1）HashSet 底层是 HashMap,第一次添加时，table 数组扩容到 16，临界值(threshold)是 16*加载因子, (loadFactor)是 0.75 = 12

（2）如果 table 数组使用到了临界值 12,就会扩容到 16* 2=32,新的临界值就是 32 * 0.75 =24,依次类推

（3）在 Java8 中,如果一条链表的元素个数到达 TREEIFY_THRESHOLD(默认是 8).并且 table 的大小>=MIN TREEIFY CAPACITY(默认 64),就会进行树化(红黑树)，否则仍然采用数组扩容机制

16. Iterator 迭代器

(1) Iterator 接口提供遍历任何 Collection 的接口。我们可以从一个 Collection 中使用迭代器方法来获取迭代器实例

(2) Iterator 对象称为迭代器,主要用于遍历 Collection 中的元素

(3) 所有实现了 Collection 接口的集合类都有一个 iterator()方法，用以返回一个实现了 Iterator 接口的对象，即可以返回一个迭代器。

```java
List<String> list = new ArrayList<>();
Iterator<String> it = list.iterator(); //得到一个集合的迭代器
while(it. hasNext()){ //判断是否还有下一个元素
  String obj = it. next(); //下移,将下移以后集合位置上的元素返回
  System. out. println(obj);
}
```

注：Iterator 的特点是只能单向遍历，但是更加安全，因为它可以确保，在当前遍历的集合元素被更改的时候，就会抛出 ConcurrentModificationException 异常。

17. Iterator 和 ListIterator 区别?

（1）Iterator 可以遍历 Set 和 List 集合，而 ListIterator 只能遍历 List。

（2）Iterator 只能单向遍历，而 ListIterator 可以双向遍历（向前/后遍历）。

（3）ListIterator 从 Iterator 接口继承，然后添加了一些额外的功能，比如添加一个元素、替换一个元素、获取前面或后面元素的索引位置。

18. HashMap 查询时间复杂度?

（1）对于指定下标的查找，理想状态下是 O(1)

（2）通过给定值进行查找，单条链表查询时，时间复杂度为 O(n)；当链表长度大于 8 时，红黑树查询时间复杂度为 O(logN)，与二分查找类似。

19. LinkedList 和 ArrayList 的插入时间复杂度?

（1）LinkedList 仅仅在头尾插入或者删除元素的时候时间复杂度近似 O(1)，其他情况增删元素的时间复杂度都是 O(n)，因为 LinkedList 在插入或删除操作前，需要通过 for 循环找到该处索引的元素。

（2）ArrayList 添加元素时，直接在尾部添加，复杂度为 O(1)；指定位置添加元素，复杂度 O(n)。

Java 多线程

多线程基础（重点）

1. 线程和进程的区别

进程：指系统正在运行的一个应用程序；程序一旦运行就是进程；进程是系统资源分配的最小单位。

线程：**线程是操作系统调度的最小单元。**进程是系统执行的最小单位。一个进程在其执行的过程中可以产生多个线程。

线程是进程的一个实体，比进程更小的独立运行基本单位。

一个程序下至少有一个进程，一个进程下至少有一个线程，一个进程下也可以有多个线程来增加程序的执行速度。

进程和进程之间的资源无法共享，线程和线程之间的资源可以共享。

2. 并行和并发有什么区别?

并行：同一时刻，多个任务同时执行；多个处理器或多核处理器同时处理多个任务

并发：同一时刻，多个任务交替执行；多个任务在同一个 CPU 核上，按细分的时间片轮流(交替)执行，从逻辑上来看那些任务是同时执行。

3. 用户线程与守护线程是什么?

1. 用户线程: 也叫工作线程，运行在前台，执行具体的任务。

2. 守护线程: 在后台运行，一般是为工作线程服务的，当所有的用户线程结束，守护线程自动结束

3. 常见的守护线程: 垃圾回收机制

4. 创建线程有哪几种方式?

创建线程有三种方式：

（1）继承 Thread 类， 重写 run 方法；

（2）实现 Runnable 接口，重写 run 方法；

（3）实现 Callable 接口，需要返回值类型。

5. 线程的 run() 和 start() 有什么区别?

调用 start() 方法，会启动一个线程并使线程进入了就绪状态，然后自动执行 run() 方法的内容，这是真正的多线程工作。而直接执行 run() 方法，会把 run 方法当成一个 main 线程下的普通方法去执行，并不会在某个线程中执行它，所以这不是多线程工作。

start() 方法用于启动线程，run() 方法用于执行线程的运行时代码。run() 可以重复调用，而 start()只能调用一次。

总结： 调用 `start()` 方法方可启动线程并使线程进入就绪状态，直接执行 `run()` 方法的话不会以多线程的方式执行。

6. 线程的状态

（1）NEW 新建状态，线程被创建出来但没有被调用 `start()`。

（2）RUNNABLE 就绪(可运行状态)，线程被调用了 `start()`，等待运行的状态。

（3）running :运行状态，就绪状态的线程被 cpu 调度后，执行程序代码

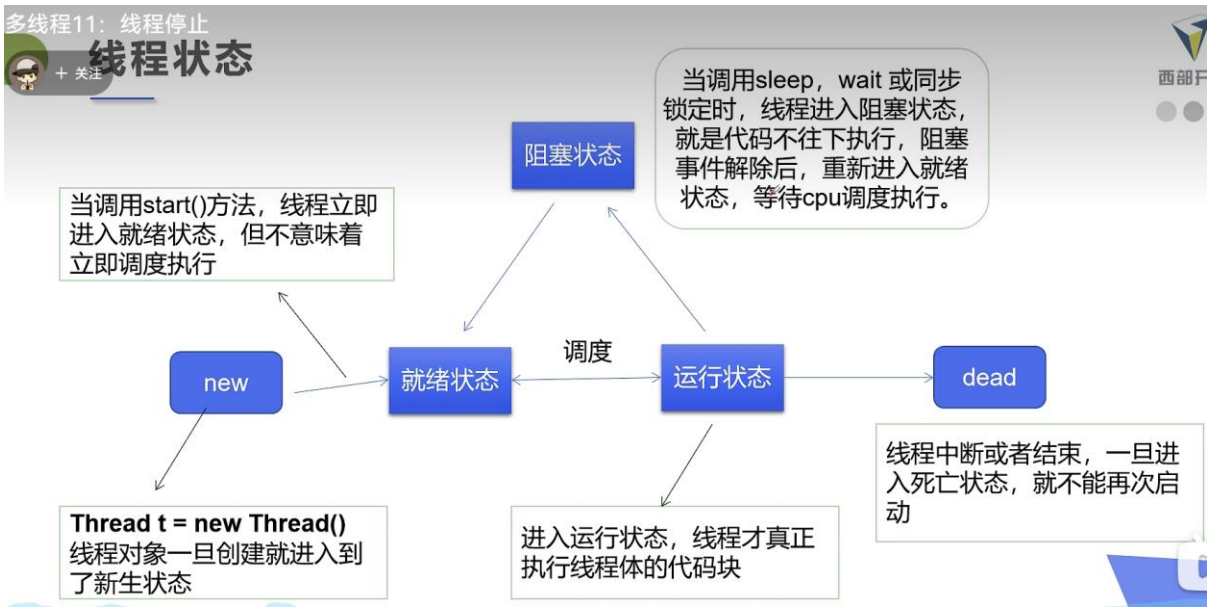
（4）BLOCKED 阻塞状态，需要等待锁释放。

（5）WAITING 持续等待状态，表示该线程需要等待其他线程做出一些特定动作（通知或中断）。

如通过 wait()方法进行等待的线程等待一个 notify()或者 notifyAll()方法，通过 join() 方法进行等待的线程等待目标线程运行结束而唤醒，

（6）TIMED_WAITING 超时等待状态，可以在指定的时间后自行返回而不是像 WAITING 那样一直等待。如 sleep(3000)方法。

（7）TERMINATED 终止状态，表示该线程已经运行完毕。



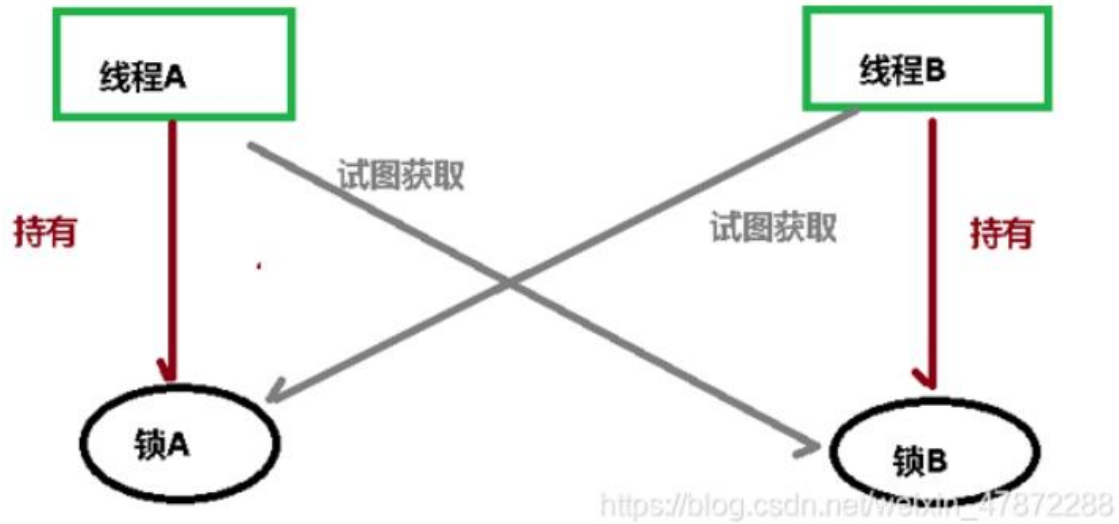
7. 什么上下文切换

线程在执行过程中会有自己的运行条件和状态（也称上下文），线程切换就是需要保存当前线程的上下文，等线程下次占用 CPU 的时候恢复，并加载下一个将要占用 CPU 的线程的上下文。（cpu 控制权从一个正在运行的线程切换到另一个就绪并等待获取 cpu 执行权的线程，但切换之前会保存自己的状态，等下次再切换时，可以再加载这个任务的状态）

8. 什么是死锁?

两个或两个以上的进程在执行过程中，因争夺资源而造成一种互相等待的现象称为死锁，若无外力干涉，它们都无法再执行下去。

举例：线程 A 持有锁 A，试图获取锁 B，线程 B 持有锁 B，试图获取锁 A，就会发生 AB 两个线程由于互相持有对方需要的锁，而发生的阻塞现象，我们称为死锁。



9. 产生死锁的条件：

（1）线程互斥：该资源任意一个时刻只由一个线程占用。

（2）不可抢占：线程已获得的资源在未使用完之前不能被其他线程强行剥夺，只有自己使用完毕后才释放资源。

（3）保持请求：一个线程因请求资源而阻塞时，对已获得的资源保持不放。

（4）循环等待：若干线程之间形成一种头尾相接的循环等待资源关系。

10. 如何预防和避免线程死锁?

（1）破坏保持请求条件 ： 一次性申请所有的资源。

（2）破坏不可抢占条件 ： 占用部分资源的线程进一步申请其他资源时，如果申请不到，可以主动释放它占有的资源。

（3）破坏循环等待条件 ： 靠按序申请资源来预防。按某一顺序申请资源，释放资源则反序释放。破坏循环等待条件。

11. sleep() 和 wait() 有什么区别?

共同点：两者都可以暂停线程的执行

不同点：

（1）类的不同：sleep()是 Thread 的静态方法，wait() 是 Object 的方法，任何对象实例都能调用

（2）是否释放锁：sleep() 不释放锁；wait() 会释放锁。

（3）用途不同：Wait 通常被用于线程间交互/通信，sleep 通常被用于暂停执行。

（4）用法不同：sleep() 方法被调用后，线程会自动苏醒；wait()方法，线程不会自动苏醒，需要别的线程使用 notify()/notifyAll()直接唤醒。或者 wait(longtimeout)，wait 方法参数设置一个超时时间，超时后线程会自动苏醒。

12. 如何使用 synchronized?

1、修饰实例方法 （锁当前对象实例）

给当前对象实例加锁，进入同步代码前要获得 **当前对象实例的锁** 。

2、修饰静态方法 （锁当前类）

给当前类加锁，会作用于类的所有对象实例 ，进入同步代码前要获得 **当前 class 的锁**。这是因为静态成员不属于任何一个实例对象，归整个类所有，不属于类的特定实例，被类的所有实例共享。

3、修饰代码块 （锁指定对象/类）

- synchronized(object) 表示进入同步代码库前要获得 **给定对象的锁。**
- synchronized(类.class) 表示进入同步代码前要获得 **给定 Class 的锁**

13. synchronized 和 volatile 有什么区别?

synchronized 关键字和 volatile 关键字是两个互补的存在，而不是对立的存在!

（1）volatile 关键字是线程同步的轻量级实现，所以 volatile 性能肯定比 synchronized 关键字要好 。但是 volatile 关键字只能用于变量，而 synchronized 关键字可以修饰方法以及代码块 。

（2）volatile 关键字不能保证数据的原子性。synchronized 关键字可以保证。

（3）volatile 关键字主要用于解决变量在多个线程之间的可见性，而 synchronized 关键字解决的是多个线程之间访问资源的同步性。

14. synchronized 和 ReentrantLock 有什么区别?

共同点：两者都是可重入锁

重入锁：自己可以再次获取自己的内部锁。比如一个线程获得了某个对象的锁，此时这个对象锁还没有释放，当其再次想要获取这个对象的锁的时候还是可以获取的，如果是不可重入锁的话，就会造成死锁。底层原理维护一个计数器，同一个线程每次获取锁，锁的计数器都自增 1，所以要等到锁的计数器下降为 0 时才能释放锁。

不同点：

1. sychronized 是一个关键字，ReentrantLock 是一个类
2. sychronized 会自动的加锁与释放锁，ReentrantLock 需要程序员手动加锁与释放锁
3. sychronized 的底层是 JVM 层面的锁，ReentrantLock 是 API 层面的锁
4. sychronized 是非公平锁，ReentrantLock 可以选择公平锁或非公平锁
5. sychronized 锁的是对象，锁信息保存在对象头中，ReentrantLock 通过代码中 int 类型的 state 标识来标识锁的状态

15. ThreadLocal

让每个线程绑定自己的值，实现每一个线程都有自己的专属本地变量。可以使用 `get()` 和 `set()` 方法来获取默认值或将其值更改为当前线程所存的副本的值，从而避免了线程安全问题。

1. ThreadLocal 是 Java 中所提供的线程本地存储机制，可以利用该机制将数据缓存在某个线程内部，该线程可以在任意时刻、任意方法中获取缓存的数据

2. ThreadLocal 底层是通过 ThreadLocalMap 来实现的，每个 Thread 对象（注意不是 ThreadLocal 对象）中都存在一个 ThreadLocalMap，Map 的 key 为 ThreadLocal 对象，Map 的 value 为需要缓存的值

3. 如果在线程池中使用 ThreadLocal 会造成内存泄漏，因为当 ThreadLocal 对象使用完之后，应该要把设置的 key，value，也就是 Entry 对象进行回收，但线程池中的线程不会回收，而线程对象是通过强引用指向 ThreadLocalMap，ThreadLocalMap 也是通过强引用指向 Entry 对象，线程不被回收，Entry 对象也就不会被回收，从而出现内存泄漏，解决办法是，在使用了 ThreadLocal 对象之后，手动调用 ThreadLocal 的 remove 方法，手动清除 Entry 对象

4. ThreadLocal 经典的应用场景就是连接管理（一个线程持有一个连接，该连接对象可以在不同的方法之间进行传递，线程之间不共享同一个连接）

16. 公平锁和非公平锁

- **公平锁**：锁被释放之后，先申请的线程/进程先得到锁。
- **非公平锁**：锁被释放之后，后申请的线程/进程可能会先获取到锁，是随机或者按照其他优先级排序的。

17. Synchronized 和 Lock 的区别

（1）synchronized 是一个关键字，底层是 JVM 层面的锁；lock 是一个接口，是 API 层面的锁

（2）synchronized 是非公平锁，lock 可以选择公平锁或非公平锁

（3）sychronized 会自动的加锁与释放锁，ReentrantLock 需要程序员手动加锁与释放锁

（4）sychronized 获取锁的时候，假设 A 线程获得锁，B 线程等待，若 A 阻塞，B 会一直等待；而 lock 在这种情况下 B 会尝试去获取锁

（5）lock 可以设置超时时间，支持响应中断；sychronized 不行，竞争不到锁会一直阻塞，容易造成死锁

多线程锁

1. ReentrantLock 有哪些方法

lock(); //获得锁

lockInterruptibly（）//获得锁，但是优先响应中断。

boolean tryLock()　//尝试获得锁，成功返回 true

tryLock（long time,TimeUnit unit）//在给定时间内尝试获得锁。

unlock(); //释放锁

2. ReentrantLock 中的公平锁和非公平锁的底层实现

首先不管是公平锁和非公平锁，它们的底层实现都会使用 AQS 来进行排队，它们的区别在于：线程在使用 lock()方法加锁时，如果是公平锁，会先检查 AQS 队列中是否存在线程在排队，如果有线程在排队，则当前线程也进行排队，如果是非公平锁，则不会去检查是否有线程在排队，而是直接竞争锁。

不管是公平锁还是非公平锁，一旦没竞争到锁，都会进行排队，当锁释放时，都是唤醒排在最前面的线程，所以非公平锁只是体现在了线程加锁阶段，而没有体现在线程被唤醒阶段。

另外，ReentrantLock 是可重入锁，不管是公平锁还是非公平锁都是可重入的。

AQS

AbstractQueuedSynchronizer ，抽象队列同步器，一个抽象类，主要用来构建锁和同步器。

原理：通过内置的 CLH(FIFO)队列的变种——一个虚拟的双向队列，来完成资源获取，线程的排队工作，将每条将要去抢占资源的线程封装成一个 Node 节点来实现锁的分配，有一个 int 类型的变量 state 表示持有锁的状态，通过 CAS 完成对 state 值的修改

CAS-(compare and swap),比较并交换

```java
// 共享变量，使用 volatile 修饰保证线程可见性
private volatile int state;
```

Java

AQS 是 JUC 内容中最重要的基石，主要用于解决锁分配给谁的问题

JUC 常见的同步工具类（了解）

CountDownLatch 有什么用？（减少计数）

`CountDownLatch` 允许 `count` 个线程阻塞在一个地方，直至所有线程的任务都执行完毕。

举例场景：

我们要读取处理 6 个文件，这 6 个任务都是没有执行顺序依赖的任务，但是我们需要返回给用户的时候将这几个文件的处理的结果进行统计整理。

CountDownLatch 表示计数器，可以给 CountDownLatch 设置一个数字，一个线程调用 CountDownLatch 的 await()将会阻塞，其他线程可以调用 CountDownLatch 的 countDown()方法来对 CountDownLatch 中的数字减一，当数字被减成 0 后，所有 await 的线程都将被唤醒。对应的底层原理就是，调用 await()方法的线程会利用 AQS 排队，一旦数字被减为 O，则会将 AQS 中排队的线程依次唤醒。

CyclicBarrier 有什么用？（可循环的屏障）循环栅栏

CyclicBarrier 内部通过一个 count 变量作为计数器，count 的初始值为 parties 属性的初始化值，每当一个线程到了栅栏（屏障）这里了，调用 await()方法，那么就将计数器减 1。如果 count 值为 0 了，表示这是最后一个线程到达栅栏，栅栏才会打开，线程才得以通过执行。

Semaphore 有什么用？（信号量）

Semaphore 表示信号量，可以设置许可的个数，表示同时允许最多多少个线程使用该信号量，通过 acquire()来获取许可，如果没有许可可用则线程阻塞，并通过 AQS 来排队，可以通过 release()方法来释放许可，当某个线程释放了某个许可后，会从 AQS 中正在排队的第一个线程开始依次唤醒，直到没有空闲许可。

JMM（Java 内存模型)

1. 什么是 JMM

JMM(Java 内存模型 Java Memory Model，简称 JMM)，是一种抽象的概念，描述的是一组规范，通过这组规范，定义了程序中(尤其是多线程)各个变量的读写访问方式，并决定一个线程对共享变量的写入时，能对另一个线程可见。JMM 的关键技术点都是围绕**多线程的原子性、可见性和有序性**展开的。

能干嘛?

（1）通过 JMM 定义了线程和主内存之间的抽象关系。

线程之间的共享变量存储在主内存（main memory）中，每个线程都有一个私有的本地内存（local memory），本地内存中存储了该线程 以读/写 共享变量的副本。

（2）屏蔽各种硬件和操作系统的内存访问差异，以实现让 Java 程序在各种平台下都能达到一致的内存访问效果。

2. Java 内存区域和 JMM 有何区别?

（1）JVM 内存结构和 Java 虚拟机的运行时区域相关，定义了 JVM 在运行时如何分区存储程序数据，就比如说堆主要用于存放对象实例。

（2）Java 内存模型和 Java 的并发编程相关，抽象了线程和主内存之间的关系，就比如说线程之间的共享变量必须存储在主内存中，规定了从 Java 源代码到 CPU 可执行指令的这个转化过程要遵守哪些和并发相关的原则和规范，其主要目的是为了简化多线程编程，增强程序可移植性的。

3. 并发编程三个重要特性

（1）原子性：指一次操作或者多次操作是不可中断的，即多线程环境下，操作不能被其他线程干扰，要么全部都执行，要么都不执行

（2）可见性：当一个线程对共享变量进行了修改，另外的线程可以立刻看到（synchronized 、volatile）

（3）有序性：程序执行的顺序按照代码的先后顺序执行。但是由于**指令重排序**问题，代码的执行顺序未必就是编写代码时候的顺序。

重排序：为了提供性能，编译器和处理器通常会对指令序列进行**重新排序。**Java 规范规定 JVM 线程内部保证**串行化语义**，即只要程序的最终结果与它顺序化执行的结果**相等**，那么指令的执行顺序可以与代码顺序**不一致**，此过程叫指令的**重排序。**

指令重排序可以保证串行语义一致，但是没有义务保证多线程间的语义也一致 ，所以在多线程下，指令重排序可能会导致一些问题。

在 java 中，**volatile** 关键字可以禁止指令进行重排序优化

volatile 关键字

volatile 是 Java 虚拟机提供的轻量级的同步机制

volatile 关键字能保证数据的可见性，但不能保证数据的原子性

synchronized 关键字两者都能保证。

1. volatile 可以保证变量的可见性

保证不同线程对某个变量完成操作后结果及时可见，即该共享变量一旦改变所有线程立即可见。被 volatile 修改的变量有以下特点：

1. 线程中**读取**的时候，每次读取都会去**主内存中**读取共享变量最新的值 ，然后将其复制到工作内存
2. 线程中修改了工作内存中变量的副本，修改之后会**立即刷新到主内存**

2. volatile 不能保证原子性

volatile 关键字能保证变量的可见性，不能保证对变量的操作是原子性的。

JVM 只是保证从主内存加载到线程工作内存的值是最新的，也仅是数据加载时是最新的。但是在多线程环境下，对于一个变量，如果第二个线程在第一个线程读取旧值和写回新值的期间，也读取操作了这个变量，就会出现写丢失问题。

举例子：比如说，对于一个共享变量，两个线程在计算的时候，一个线程计算后还没提交，第二个线程已经计算后提交了，那么第一个线程的操作计算将会作废去读主内存最新的值，操作出现写丢失问题。

3. 底层：内存屏障

内存屏障其实就是一种 jvm 指令，java 内存模型的重排规则会要求 Java 编译器在生成 JVM 指令时插入特定的内存屏障指令 ，通过这些内存屏障指令，volatile 实现了 Java 内存模型中的可见性和有序性，*但 volatile 无法保证原子性* 。

内存屏障之前**的所有**写**操作都要回写到主内存，**内存屏障之后**的所有**读**操作都能获得内存屏障之前的所有写操作的最新结果(实现了可见性)。**

写屏障(Store Memory Barrier) ：告诉处理器在写屏障之前将所有存储在缓存(store bufferes) 中的数据刷新到主内存中。也就是说当看到 Store 屏障指令， 就必须把该指令之前所有写入指令执行完毕才能继续往下执行。

读屏障(Load Memory Barrier) ：在读指令之前插入读屏障，让工作内存或 CPU 高速缓存当中的缓存数据失效，重新回到主内存中获取最新数据。

4. 双重检验锁实现对象单例（线程安全） ：

什么是双重检验锁?

在实现单例模式时，多线程下可能会引起线程不安全的问题，

第一种解决方法是：使用 synchronized 关键字加锁，但是性能消耗比较大。

第二种解决方法，可以使用双重检验，synchronized 配合 volatile 使用：先判断对象是否已经被初始化，再决定要不要加锁。**初始化的时候会出现加锁，后续的所有调用都不会加锁而直接返回**，同时也可以解决指令重排引起的问题。

```
public class SafeDoubleCheckSingleton
{   //-----这里没加 volatile，实现线程安全的延迟初始化
```

```
    private static volatile SafeDoubleCheckSingleton singleton;
    //私有化构造方法
    private SafeDoubleCheckSingleton(){
    }
    //双重锁设计
    public static SafeDoubleCheckSingleton getInstance(){
        if (singleton == null){
            //1.多线程并发创建对象时，会通过加锁保证只有一个线程能创建对象
            synchronized (SafeDoubleCheckSingleton.class){
                if (singleton == null){
                    //隐患：多线程环境下，由于重排序，该对象可能还未完成初始化就被其他线程读取
                    //由于指令重排，会将第二步和第三步进行重排序，某个线程可能会获得一个未完全初始化的实例
                    singleton = new SafeDoubleCheckSingleton();
                    //实例化分为三步
                    //1.分配对象的内存空间
                    //2.初始化对象
                    //3.设置对象指向分配的内存地址
                }
            }
        }
        //2.对象创建完毕，执行 getInstance()将不需要获取锁，直接返回创建对象
        return singleton;
    }
}
```

Java 常见并发容器总结

1. CopyOnWriteArrayList

线程安全的 List，在读多写少的场合性能非常好，远远好于 Vector。用到**写时复制技术**

1. 首先 CopyOnWriteArrayList 内部也是用数组来实现的，在向 CopyOnWriteArrayList 添加元素时，会复制一个新的数组，写操作在新数组上进行，读操作在原数组上进行
2. 并且，写操作会加锁，防止出现并发写入丢失数据的问题，保证了同步，避免了多线程写的时候会 copy 出多个副本出来。
3. 写操作结束之后会把原数组指向新数组
4. CopyOnWriteArrayLlist 允许在写操作时来读取数据，大大提高了读的性能，因此适合读多写少的应用场景，但是 CopyOnWriteArraylist 会比较占内存，同时可能读到的数据不是实时最新的数据，所以不适合实时性要求很高的场景

2. CopyOnWriteArraySet

底层是 CopyOnWriteArrayList

3. ConcurrentHashMap

HashMap 不是线程安全的，HasTable 的内部方法都被 synchronized 修饰了，所以是线程安全的。

ConcurrentHashMap 内部使用粒度更细的分段锁机制，分段锁简单来说就是将数据进行分段，每一段锁用于锁容器中的一部分数据，那么当多线程访问容器里的不同数据段的数据时，线程间就不会存在锁竞争，从而可以有效地提高并发访问效率。

4. BlockingQueue

阻塞队列（BlockingQueue）被广泛使用在“生产者-消费者”问题中，其原因是 BlockingQueue 提供了可阻塞的插入和移除的方法。当队列容器已满，生产者线程会被阻塞，直到队列未满；当队列容器为空时，消费者线程会被阻塞，直至队列非空时为止。

线程池（重点）

1. 为什么要用线程池?

池化技术的思想主要是为了减少每次获取资源的消耗，提高对资源的利用率。线程池的好处：

- **降低资源消耗。**通过重复利用已创建的线程降低线程创建和销毁造成的消耗。
- **提高响应速度。**当任务到达时，任务可以不需要等到线程创建就能立即执行。
- **提高线程的可管理性。**线程是稀缺资源，如果无限制的创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一的分配，调优和监控。

2. ThreadPoolExecutor 参数（线程池参数）

```
/**
     * 用给定的初始参数创建一个新的 ThreadPoolExecutor。
     */
    public ThreadPoolExecutor(int corePoolSize,//线程池的核心线程数量
                              int maximumPoolSize,//线程池的最大线程数
```

```
                            long keepAliveTime,//当线程数大于核心线程数时，多余的空闲线程存活的最长时间
                            TimeUnit unit,//时间单位
                            BlockingQueue<Runnable> workQueue,//任务队列，用来储存等待执行任务的队列
                            ThreadFactory threadFactory,//线程工厂，用来创建线程，一般默认即可
                            RejectedExecutionHandler handler//拒绝(饱和)策略，当提交的任务过多而不能及时处理时，我们可以定制策略来处理任务
                            ) {
        if (corePoolSize < 0 ||
            maximumPoolSize <= 0 ||
            maximumPoolSize < corePoolSize ||
            keepAliveTime < 0)
            throw new IllegalArgumentException();
        if (workQueue == null || threadFactory == null || handler == null)
            throw new NullPointerException();
        this.corePoolSize = corePoolSize;
        this.maximumPoolSize = maximumPoolSize;
        this.workQueue = workQueue;
        this.keepAliveTime = unit.toNanos(keepAliveTime);
        this.threadFactory = threadFactory;
        this.handler = handler;
    }
```

Java

3. 线程池的实现原理

执行 execute 方法，创建新的线程，分四种情况

（1）如果当前运行的线程少于 corePoolSize，则创建新线程来执行任务（注意，执行这一步骤需要获取全局锁）。

（2）如果运行的线程等于或多于 corePoolSize，则将任务加入 BlockingQueue。

（3）如果无法将任务加入 BlockingQueue（队列已满），则创建新的线程来处理任务）。

（4）如果创建新线程将使当前运行的线程超出 maximumPoolSize，任务将被拒绝，并调用 RejectedExecutionHandler.rejectedExecution()方法。

4. 线程池中 submit() 和 execute() 方法有什么区别?

execute()：只能执行 Runnable 类型的任务，没有返回值

submit()：可以执行 Runnable 和 Callable 类型的任务。

Callable 类型的任务可以获取执行的返回值，而 Runnable 执行无返回值。

5. ThreadPoolExecutor 饱和策略有哪些?

- AbortPolicy：直接抛出异常。
- CallerRunsPolicy：只用调用者所在线程来运行任务。
- DiscardOldestPolicy：丢弃队列里最近的一个任务，并执行当前任务。
- DiscardPolicy：不处理，丢弃掉。

其它

同步容器，并发容器

同步容器：并发访问 ArrayList、LinkedList、HashMap 等容器时，会出现问题。所以 java 提供同步容器供用户使用，主要包括：Vector、HashTable、

Collections.synchronized 方法。可以简单地理解为通过 synchronized 来实现同步的容器，同步容器会导致多个线程中对容器方法调用的串行执行，降低并发性，效率低。

并发容器：并发容器是针对多个线程并发访问而设计的，在 jdk5.0 引入了 concurrent 包,其中提供了很多并发容器,如 ConcurrentHashMap、CopyOnWriteArrayList 等。

- **CopyOnWrite 容器**：、CopyOnWriteArrayList、CopyOnWriteArraySet

  CopyOnWriteArrayList：写时复制技术，写时复制一份新的，在新的上面修改，然后把引用指向新的，适合读多写少的场合。由于写操作的时候会进行复制，会消耗内存。

  CopyOnWriteArraySet ， java.util.ArrayList 的线程安全版本，基于 CopyOnWriteArrayList

- **CocurrentMap 的实现类**：ConcurrentHashMap、ConcurrentSkipListMap

  HashMap 不是线程安全的，HasTable 的内部方法都被 synchronized 修饰了，所以是线程安全的。ConcurrentHashMap 内部使用粒度更细的分段锁机制，分段锁简单来说就是将数据进行分段，每一段锁用于锁容器中的一部分数据，那么当多线程访问容器里的不同数据段的数据时，线程间就不会存在锁竞争，从而可以有效地提高并发访问效率。

  ConcurrentSkipListMap：使用 SkipList(跳表)实现排序，Skip list（跳表）是一种可以代替平衡树的数据结构，默认是按照 Key 值升序的。Skip list 让已排序的数据分布在多层链表中，通过"空间来换取时间"的一个算法。

JVM

1. Java 内存区域（运行时数据区）（重点）

（1）程序计数器：线程私有。记录当前线程执行的位置，**保存 JVM 中下一条所要执行的字节码指令的地址**！

（2）java 虚拟机栈：线程私有，生命周期与线程相同，每个方法在执行的同时都会创建一个栈帧用于存储局部变量表、操作数栈、动态链接、方法返回地址等信息。每一个方法从被调用直至执行完成的过程，就对应着一个栈帧在虚拟机栈中入栈到出栈的过程。（栈内存用来存储局部变量和方法调用。）

（3）本地方法栈：线程私有，本地方法栈与虚拟机栈所发挥的作用是非常相似的，它们之间的区别不过是虚拟机栈为虚拟机执行 Java 方法（也就是字节码）服务，而本地方法栈则为虚拟机使用到的 *Native* 方法服务。

（4）堆：线程共享，java 内存最大的一块，所有对象的实例和数组都存放在堆中，是垃圾收集器管理的主要区域

（5）方法区：线程共享，存放已被虚拟机加载的信息、常量、静态变量，编译后的代码等数据。

方法区的**垃圾收集**主要回收两部分内容：废弃的常量和不再使用的类型。

2. java 堆相关

由于 java 堆是垃圾收集器管理的主要区域，因此也被称作 **GC 堆（Garbage Collected Heap）**。从垃圾回收的角度，由于现在收集器基本都采用分代垃圾收集算法，所以 Java 堆还可以细分为：新生代和老年代；再细致一点有：Eden、Survivor、Old 等空间。进一步划分的目的是更好地回收内存，或者更快地分配内存。

新生代：

大部分情况，对象都会首先在 Eden 区域分配，在一次新生代垃圾回收后，如果对象还存活，eden 区域的对象则会进入 survivor 区，并且对象的年龄会加 1，直到 15 岁后进入老年代。

空间担保：

当创建的对象大于 eden，先进行一次新生代垃圾回收，如果还是无法创建新的对象，会通过空间担保策略提前进入进入老年代。当 Surivivor 区满了的时候，对象也会分配到老年代。

动态年龄：

什么是动态年龄勒，这是堆中的另一个，担保策略了，它会去判断我们 Surivivor 的区中，相同年龄的对象大于 Surivivor 区一半的时候，那么他就会判定此时这些对象已经能够很好的存活了，所以他们就集体被丢到老年代了。

老年代：

老年代存放的都是一些存活时间较长的对象，很少产生垃圾回收，一旦产生，耗时是新生代垃圾回收的 10 倍，而我们老年代快要存满时进入了一个对象，这时会产生一次老年代垃圾回收，如果 GC 结束后，还是无法存放对象的话此时就会报 OOM 异常。

3. 堆与栈的区别

最主要的区别就是栈内存用来存储局部变量和方法调用。

而堆内存用来存储 Java 中的对象。无论是成员变量，局部变量，还是类变量，它们指向的对象都存储在堆内存中。

4. 虚拟机栈内存溢出的情况

（1）虚拟机栈中，**栈帧过多**（方法无限递归）导致栈内存溢出，这种情况比较**常见。**当线程请求栈的深度超过当前 Java 虚拟机栈的最大深度的时候，就抛出 StackOverFlowError 错误;

（2）每个栈帧**所占用内存过大**(某个/某几个栈帧内存直接超过虚拟机栈最大内存)，这种情况比较**少见**!

5. GC（Garbage Collected）的两种判定方法（如何判断一个对象是否存活）

（1）引用计数法：指的是如果某个地方引用了这个对象就+1，如果失效了就-1，当为 0 就会回收但是 JVM 没有用这种方式，因为无法判定相互循环引用（A 引用 B,B 引用 A） 的情况。

（2）可达性分析算法（引用链法）： 通过一种 GC ROOT 的对象（方法区中静态属性、常量引用的对象，虚拟机栈中引用的对象等）来判断，如果有一条链能够到达 GC ROOT 就说明不可以回收，不能到达 GC ROOT 就说明可以回收。

6. 强引用、软应用、弱引用、虚引用的区别

强引用：如果一个对象具有强引用，那么垃圾回收期绝对不会回收它，当内存空间不足时，垃圾回收器宁愿抛出 OutOfMemoryError，也不会回收具有强引用的对象。

软引用(SoftReference)：当一个对象只有软引用时，只有当内存不足时，才会回收它；可以和引用队列（ReferenceQueue）联合使用，如果软引用所引用的对象被垃圾回收器所回收了，虚拟机会把这个软引用加入到与之对应的引用队列中。

弱引用(WeakReference)：在垃圾回收时，无论内存是否充足，都会回收弱引用对象；可以和引用队列（ReferenceQueue）联合使用。

虚引用：必须和引用队列联合使用；在进行垃圾回收的时候，如果发现一个对象只有虚引用，那么就会将这个对象的引用加入到与之关联的引用队列中，程序可以通过判断一个引用队列中是否已经加入了虚引用，来了解被引用的对象是否要被进行垃圾回收；虚引用主要用来跟踪对象被垃圾回收器回收的活动。

7. 垃圾收集算法（重点）

分代垃圾回收算法：

首先说一下分代垃圾回收算法，因为收集器基本上都采用分代垃圾收集算法。其实也就是将我们堆空间划分为了一个个不同的区域，新生代，老年代，划分出不同的区域后，垃圾收集器可以只回收某一个区域，新生代垃圾回收、老年代垃圾回收、整个 java 堆和方法区的垃圾回收。这样我们就可以根据各个年代的特点选择合适的垃圾收集算法。比如在新生代中，每次收集都会有大量对象死去，所以可以选择"标记-复制"算法，只需要付出少量对象的复制成本就可以完成每次垃圾收集。而老年代的对象存活几率是比较高的，而且没有额外的空间对它进行分配担保，所以我们必须选择"标记-清除"或"标记-整理"算法进行垃圾收集。

标记-清除算法：直接释放，将标记的区域中的内存释放，简单高效，但是容易产生大量不连续的碎片

标记-复制算法

为了解决效率问题，复制算法将可用内存按容量划分为相等的两部分，每次只使用其中的一块，这一块的内存使用完后，就将还存活的对象复制到另一块去，然后再把使用的空间一次清理掉。这样就使每次的内存回收都是对内存区间的一半进行回收。

不存在内存碎片，浪费内存，空间换时间

标记-整理算法

在清除对象的时候，让存活的对象向一端移动，然后清除掉端边界以外的对象；

8. 类加载过程

类加载的过程包括：**加载、验证、准备、解析、初始化。**其中验证、准备、解析统称为**连接。**

（1）加载：通过一个类的全限定名来获取定义此类的二进制字节流，在内存中生成一个代表这个类的 java.lang.Class 对象。（通过全限定名来加载生成 class 对象到内存中）

（2）验证：确保 Class 文件的字节流中包含的信息符合当前虚拟机的要求，并且不会危害虚拟机自身的安全。验证 class 文件，包括文件格式校验、元数据验证，字节码校验等。

（3）准备：为类变量分配内存并设置类变量初始值，这里所说的初始值"通常情况"下是数据类型默认的零值。（为这个对象分配内存）

类变量，即静态变量，而不包括实例变量。实例变量会在对象实例化时随着对象一块分配在 Java 堆中。

（4）解析：虚拟机将常量池内的符号引用替换为直接引用。

（5）初始化：到了初始化阶段，才真正开始执行类中定义的 Java 初始化程序代码。初始静态成员并且执行静态代码块。（开始执行构造器的代码，初始静态成员并且执行静态代码块）

9. 常见的类加载器

实现通过类的权限定名，获取该类的二进制字节流的代码块叫做类加载器。加载的作用就是将 .class 文件加载到内存。

主要有一下四种类加载器:

启动类加载器(Bootstrap ClassLoader)，最顶层的加载类，用来加载 java 核心类库，无法被 java 程序直接引用。

扩展类加载器(extensions class loader):它用来加载 Java 的扩展库。Java 虚拟机的实现会提供一个扩展库目录。该类加载器在此目录里面查找并加载 Java 类。

系统类加载器（system class loader）:面向我们用户的加载器，负责加载当前应用类路径 classpath 下的所有 jar 包和类。可以通过 ClassLoader.getSystemClassLoader()来获取它。

用户自定义类加载器，通过继承 java.lang.ClassLoader 类的方式实现

10. 双亲委派模型介绍（了解）

如果一个类加载器收到了类加载的请求，它首先不会自己去尝试加载这个类，而是把这个请求委派给父类加载器去完成，每一个层次的类加载器都是如此，因此所有的加载请求最终都应该传送到顶层的启动类加载器中，只有当父加载器反馈自己无法完成这个加载请求（它的搜索范围中没有找到所需的类）时，子加载器才会尝试自己去加载。