JavaSE 八股

一、Java 语言

1. 什么是 Java?

1. Java 是一门面向对象的编程语言，由 Sun 公司的詹姆斯高斯林团队于 1995 年推出。吸收了 C++语言中大量的优点，但又抛弃了 C++ 中容易出错的地方，如垃圾回收。
2. 同时，Java 又是一门平台无关的编程语言，即一次编译，到处运行。只需要安装一个 JVM，就可以运行编译过的字节码文件。

2. 说一下 Java 的特点

主要有以下的特点：

1. 平台无关性：“一次编写，到处运行” java语言编写的 程序具有很好的可移植性
2. 面向对象：主要是封装、继承和多态 等面向对象的基本概念。面向对象编程使代码更具模块化、可重用性和可维护性。
3. 支持多线程：C++没设计，java 调用操作系统的多线程设计，为 java 语言提供了多线程支持。

3. Java 为什么是跨平台的?

1. 所谓跨平台性，是指 Java 语言编写的程序，一次编译后，可以在多个系统平台上运行。
2. 原理是增加了一个中间件 JVM，JVM 负责将 Java 字节码转换为特定平台的机器码，并执行。

4. JVM、JDK、JRE 三者关系?

它们之间的关系如下：JDK 包含 JRE，JRE 包含 JVM

1. JVM ： Java 虚拟机， Java 程序运行的环境。是 Java 实现跨平台的关键所在，针对不同的操作系统，有不同的 JVM 实现。JVM 负责将 Java 字节码转换为特定平台的机器码，并执行。
2. JRE ： Java 运行时环境。包含了运行 Java 程序所必需的库（如：java.lang，java.util），以及 Java 虚拟机
3. JDK： 一套完整的 Java SDK（软件开发工具包），包括了 JRE 以及编译器（javac）、Java 文档生成工具（Javadoc）、Java 调试器等开发工具。为开发者提供了开发、编译、调试 Java 程序的一整套环境。

5. 什么是字节码？采用字节码的好处是什么?

字节码，就是 Java 程序经过编译之后产生的.class 文件。字节码能够被虚拟机识别，从而实现 Java 程序的跨平台性。

java 程序从源代码到执行有三步：

- 编译：将我们的代码（.java）编译成虚拟机可以识别理解的字节码（.class)
- 解释：虚拟机JVM执行 Java 字节码，将字节码翻译成机器能识别的机器码
- 执行：对应的机器执行二进制机器码

6. 为什么 Java 解释和编译并存?

1. 编译性：编译型语言是指编译器针对特定的操作系统将源代码一次性翻译成可被该平台执行的机器码
2. 解释性：解释型语言是指解释器对源程序逐行解释成特定平台的机器码并立即执行。

Java 程序需要先将 Java 源代码文件编译字节码文件，再解释执行。

7. 编译型语言和解释型语言的区别?

1. 编译型语言：在程序执行之前，整个源代码会被编译成机器码或者字节码，生成可执行文件。执行时直接运行编译后的代码，速度快，但跨平台性较差。
2. 解释型语言：在程序执行时，逐行解释执行源代码，不生成独立的可执行文件。通常由解释器动态解释并执行代码，跨平台性好，但执行速度相对较慢。
3. 典型的编译型语言如 C、C++，典型的解释型语言如 Python、JavaScript。

8. Python 和 Java 区别是什么?

1. Java 是一种已编译的编程语言，Java 编译器将 Java 代码编译 class 字节码，而字节码则由 JVM 执行。
2. python 是一种解释语言，程序运行时，逐行解释执行代码，不生成独立的可执行文件。
3.

二、数据类型相关

1. 八种基本的数据类型有哪些

基本数据类型共有 8 种，可以分为三类：

数值型：整数类型（byte、short、int、long）和浮点类型（float、double）\ 字符型：char \布尔型：boolean

注意一下几点：

- (byte) 1字节、(short、char) 2字节、（int、float）4 字节、（long、double）8 字节、 boolean（1/4 字节）

1. boolean 的字节数依靠 JVM 来决定（不同的 JVM）boolean 的大小可能不相同
    - 如果 boolean 是 “单独使用”：boolean 被编译为 int 类型，占 **4** 个字节
    - 如果 boolean 是以 “boolean 数组” 的形式使用：boolean 占 **1** 个字节
2. 浮点数的默认类型为 double（如果需要声明一个常量为 float 型，则必须要在末尾加上 f 或 F）
3. 整数的默认类型为 int（声明 Long 型在末尾加上 l 或者 L）
4. float 和 double 都遵循 IEEE 754 标准。
5. 八种基本数据类型的包装类：除了 char 的是 Character、int 类型的是 Integer，其他都是首字母大写
6. char 类型是无符号的，不能为负，所以是 0 开始的

2. 引用数据类型有哪些?

类、接口、数组

3. Integer最大值+1，是什么结果?

溢出。从 Integer.MAX_VALUE 变成 Integer.MIN_VALUE

4. ❤️java 是值传递还是引用传递?

java 是值传递。

- 基本数据类型：对于基本数据类型：传递的是具体的值的副本。函数内部对参数的任何修改都不会影响原始变量。
- 引用数据类型：传递的是实参引用的对象在堆中地址值的副本。例如对象、数组等。函数内部可以通过引用修改对象的内容，但是不能改变引用本身指向的对象。（与 final 修饰成员变量类似）

5. long 和 int 可以互转吗？

可以，Java 中的 long 和 int 可以相互转换。由于 long 范围>int 范围，因此将 int 转换为 long 是安全的，而将 long 转换为 int 可能会导致数据丢失或溢出。

6. &和&&有什么区别?

& 逻辑与：一假且为假；&& 短路与：左边为 false，右边不再执行。

逻辑或和短路或也一样

7. switch 是否能作用在哪些基本数据类型上?

Switch（expr）-所有版本都不支持 long

- jdk5 前：expr 是 byte、short、int、char
- jdk5：引入枚举（enum）
- jdk7：引入 String

8. break ，continue ，return 的区别及作用?

- break ：跳出整个循环，不再执行循环（**结束当前的循环体**）
- continue： 跳出本次循环，继续执行下次循环（**结束本次循环 进入下一次循环**）
- return：程序返回，不再执行下面的代码（**结束当前的方法 直接返回**）

9. 用最有效率的方法计算 2 乘以 8?

2 << 3. 位运算，数字的二进制位左移三位相当于乘以 2 的三次方。

10. 说说自增自减运算?

符号在前就先+-，再赋值；符号在后就先赋值，后+-

- 符号在前：++1/--1，先+-，再赋值
- 符号在后：1++/1--，先赋值，再+-

```
int count = 0;
for(int i = 0;i < 100;i++)
{
    //先返回当前值0，再+1
    count = count++;
}
System.out.println("count = "+count);
//100次无用赋值，每次01 01循环
```

11. 数据类型转换方式你知道哪些?

自动、强转、字符转整形、整形转字符串

1. 自动类型转换（隐式转换）：当目标类型的范围>源类型时，Java 会自动将源类型转换为目标类型，不需要显式的类型转换。int->long、float->double
2. 强制类型转换（显式转换）：当目标类型的范围小于源类型时，需要使用强制类型转换将源类型转换为目标类型。这可能导致数据丢失或溢出。
3. 字符串转为整形：String->int,用 Integer.parseInt()方法;String->double,用 Double.parseDouble()
4. 整型转换为字符串：Integer.toString() / String.valueOf()

12. 类型互转会出现什么问题吗?

数据丢失、数据溢出、精度损失、类型不匹配出错

1. 数据丢失：范围大的数据转换范围小的数据时，可能会发生数据丢失。long->int
2. 数据溢出：范围小的数据转换范围大的数据时，可能会发生数据溢出。int->long
3. 精度损失：在进行浮点数类型的转换，可能会发生精度损失。foloat->double
4. 类型不匹配导致的错误：转换时确保，源类型和目标类型是兼容的。如果两者不兼容，会导致编译错误或运行时错误。

13. 为什么用 bigDecimal 不用 double？（数据准确性如何保证？）

Java 中进行浮点数运算的时候，会出现丢失精度的问题。double 会出现精度丢失的问题。double 执行的是二进制浮点运算，无法准确地表示，例如 `0.1` 。

```
1   System.out.println(0.05 + 0.01); //输出0.06000000000000005
2   System.out.println(1.0 - 0.42);   // 输出: 0.3580000000000001
```

bigDecimal 是精确计算，涉及金钱一般都用 bigDecimal

```
1    BigDecimal bigA = new BigDecimal("0.1");
2    BigDecimal bigB = new BigDecimal("0.2");
3    BigDecimal bigResult = bigA.add(bigB);
```

```
4    System.out.println(bigResult);    // 输出: 0.3
```

14. 装箱和拆箱是什么。

装箱和拆箱将基本数据类型和对应的包装类之间进行转换的过程。主要发生在赋值时和方法调用时。底层是基于valueOf()方法，维护了一个缓存池，范围在[-128,127]。传入的值在此范围内，会返回一个创建好的对象；传入的值>127,则会new一个对象。

1. 自动装箱底层是：包装类.valueOf（）方法。传入的值在[-128,127]，会返回数组中已经创建好的对象；如果传入的值>127，则会隐式 new 一个对象。
2. 自动装箱的弊端：因为自动装箱会隐式创建对象，占用内存，引起 GC 压力，降低程序的性能。如果在循环中使用自动装箱，会加重垃圾回收的工作量。
3. 在编程时，需要注意到这一点，正确地声明变量类型，避免因为自动装箱引起的性能问题。
4. 自动装箱主要发生在两种情况。①赋值时 ②方法调用时

15. Java 为什么要有 Integer?

1. Java是面向对象的，而int缺乏对象的特性，像在集合中，int就无法直接使用。
2. 对象具有封装性，可以把属性和方法相结合，比如parsrInt()方法就可以处理int型的数据。
3. Integer同时是一个类，还扩展了一些特有的属性。

16. 包装类的好处是什么?

1. 与对象兼容：Java 是面向对象语言，很多地方都需要使用对象而不是基本数据类型。
2. 处理 null 值：基本数据类型不能表示 null 值，而包装类可以。比如：前端忘记传参数给后端，如果使用基本数据类型，直接报错；使用包装类，就会避免这种情况。
3. 提供额外的方法和功能：包装类是一个类，提供了许多实用的方法和功能。比如：Integer 的 parseInt()、valueOf()、compareTo()。
4. 支持泛型：泛型在 Java 中只支持对象类型，基本数据类型无法直接应用于泛型。使用包装类，就可以方便的在泛型中使用基本数据类型对应的包装类，增强代码的复用性和可维护性。
5. 实现自动装箱和拆箱：Java 提供自动装箱和拆箱的功能，使得基本数据类型和包装类之间可以相互转换。提高了开发效率。

17. 那为什么还要保存 int 呢?

不管是读写效率，还是存储效率，基本类型都比包装类高效。

1. 存储效率方面：包装类是引用类型，对象的引用和对象本身是分开存储的；对于基本类型数据，变量对应的内存块直接存储数据本身。
2. 读写效率方面：基本类型数据占 4 字节，在栈内分配空间，读写直接访问内存；包装类对象存在堆，除数据还有对象头信息，通过栈中的引用间接访问，增加内存访问步骤和开销。
3. 在 64 位 JVM 上，在开启引用压缩的情况下，一个 Integer 对象占用 16 个字节的内存空间，而一个 int 类型数据只占用 4 字节的内存空间。

18. 说一下 integer 的缓存?

Integer 类内部实现了一个静态缓存池，存储特定范围内的整数值对应的 Integer 对象。

1. 默认情况下，这个范围是 -128 至 127。当通过 Integer.valueOf（int）方法创建一个在这个范围内的整数对象时，并不会每次都生成新的对象实例，而是复用缓存中的现有对象，会直接从内存中取出，不需要新建一个对象。
2. 超过这个范围，则会 new 一个 Integer。

三、面向对象

1. 面向对象和面向过程怎么区别?

面向过程以过程为核心，通过函数完成任务。代码使用一些函数和if-else。

面向对象以对象为核心，通过对象的交互完成程序功能。代码可以通过封装继承多态等方式复用。

2. 怎么理解面向对象？简单说说封装继承多态?

面向对象是把现实世界中的事物抽象为对象，对象具有属性和行为。通过对象之间的交互来完成程序的功能。

Java 面向对象的三大特性包括：封装、继承、多态

1. 封装：利用抽象将对象的属性（数据）和行为（方法）封装成一个不可分割的实体，隐藏内部细节，对外提供部分操作。
2. 继承：继承是允许子类继承父类的属性和方法。通过继承可以建立类与类之间的层次关系，提高代码复用性。
3. 多态：指同一个接口或方法在不同的类中有不同的实现。它使得程序具有良好的灵活性和扩展性。

【多态的前置条件有三个：子类继承父类、子类重写父类的方法、父类引用指向子类的对象】

3. 多态体现在哪几个方面?

方法重载、方法重写、接口多态、向上转型和向下转型

1. 方法重载：指同一类中可以有多个同名方法，它们具有不同的参数列表（参数类型、数量或顺序不同）。根据传入的参数不同，编译器会在编译时确定调用哪个方法。
2. 方法重写：指子类能够提供对父类中同名方法的具体实现。在运行时，JVM 会根据对象的实际类型确定调用哪个方法。这是实现多态的主要方式。
3. 接口多态：多个类可以实现同一个接口，接口的引用可以指向不同的实现类。
4. 向上转型和向下转型：父类类型的引用指向子类对象，这是向上转型。通过这种方式，可以在运行时期采用不同的子类实现。/向下转型是将父类引用转回其子类类型，可以访问子类独有的属性。在执行前需要确认引用实际指向的对象类型以避免 ClassCastException。
5. 多态是对象、行为的多态，属性不谈多态

4. 多态解决了什么问题?

1. 多态指同一个接口或方法在不同的类中有不同的实现，比如说动态绑定，父类引用指向子类对象，方法的具体调用会在运行时决定。
2. 在运行时根据对象的类型进行动态绑定，编译器在编译阶段并不知道对象的类型，但是 Java 的方法调用机制能找到正确的方法体，然后执行，得到正确的结果

5. 面向对象的设计原则你知道有哪些吗

面向对象编程有 6 种原则：

1. 单一职责原则（SRP）：一个类应该只有一个引起它变化的原因，即一个类只负责一项职责。这样做的目的是使类更加清晰，更容易理解和维护。
2. 开放原则（OCP）：指软件实体应该对扩展开放，对修改关闭。这意味着一个类应该通过扩展来实现新的功能，而不是通过修改已有的代码来实现。
3. 里氏替换原则（LSP）：任何父类可以出现的地方，子类也一定可以出现。例子：一个正方形是一个矩形，但如果修改一个矩形的高度和宽度时，正方形的行为应该如何改变就是一个违反里氏替换原则的例子。
4. 接口隔离原则（ISP）：指客户端不应该依赖它不需要的接口。这意味着设计接口时应该尽量精简，不应该设计臃肿庞大的接口。

5. 依赖倒置原则（DIP）：指高层模块不应该依赖低层模块，二者都应该依赖其抽象；抽象不应该依赖细节，细节应该依赖抽象。这意味着设计时应该尽量依赖接口或抽象类，而不是实现类。

6. 重载与重写有什么区别?

1. 重载（Overloading）：指在同一个类中，可以有多个同名方法，它们具有不同的参数列表（参数类型、参数个数或参数顺序不同），编译器根据调用时的参数类型来决定调用哪个方法。
2. 重写（Overriding）：指子类可以重新定义父类中的方法，方法名、参数列表和返回类型必须与父类中的方法一致，通过@override 注解来明确表示这是对父类方法的重写。
3. 重载是指在同一个类中定义多个同名方法，而重写是指子类重新定义父类中的方法。

7. 访问修饰符 public、private、protected、以及默认时的区别?

Java 中，可以使用访问控制符来保护对类、变量、方法和构造方法的访问。

- default （即默认，什么也不写）: 在同一包内可见，不使用任何修饰符。可以修饰在类、接口、变量、方法。
- private : 在同一类内可见。可以修饰变量、方法。注意：不能修饰类（外部类）
- public : 对所有类可见。可以修饰类、接口、变量、方法
- protected : 对同一包内的类和所有子类可见。可以修饰变量、方法。注意：不能修饰类（外部类）。

8. this 关键字的作用?

1. this 代表当前对象本身
2. 形参和成员变量重名，可以用 this 区分
3. 引用本类的构造器

9. 抽象类和普通类区别?

1. 实例化：普通类可以直接实例化对象，而抽象类不能被实例化，只能被继承。
2. 方法实现：普通类中的方法可以有具体的实现；而抽象类中的方法只有方法签名。
3. 实现限制：普通类可以被其他类继承和使用，而抽象类一般用于作为基类，被其他类继承和扩展使用。

10. ❤️抽象类和接口的区别是什么?

1. 一个类只能继承一个抽象类；但一个类可以实现多个接口。故：新建线程类一般使用实现 Runnable 接口，这样线程类还可以继承其他类，而不单单是 Thread 类。
2. 一个接口也可以继承多个其他接口
3. 抽象类符合 is-a 的关系，而接口更像是 has-a 的关系，比如说一个类可以序列化的时候，它只需要实现 Serializable 接口就可以了，不需要去继承一个序列化类。
4. 抽象类更多地是用来为多个相关的类提供一个共同的基础框架，包括状态的初始化；接口是定义一套行为标准，让不同的类可以实现同一接口，提供行为的多样化实现。
5. 他们都不能实例化。他们提供的是一个规范（接口）或模版（抽象类），供子类实现或继承。

11. ❤️抽象类和接口的设计目的?

- 抽象类：代码的复用。
- 接口：主要用于指定规范。

12. 抽象类可以有构造方法吗?

可以。构造器在子类实例化时会被调用，以便进行必要的初始化工作。

13. 抽象类能加 final 修饰吗?

不能，Java 中的抽象类是用来被继承的，而 final 修饰符用于禁止类被继承或方法被重写，因此，抽象类和 final 修饰符是互斥的，不能同时使用。

14. 继承和抽象的区别?

1. 继承是一种运行子类继承父类属性和方法的机制。通过继承，子类可以重用父类的代码。
2. 抽象是一种隐藏复杂性只展必要部分的技术。如封装：利用抽象将对象的属性和行为封装成一个不可分割的实体。隐藏内部细节，只对外提供部分操作。

15. 接口里面可以定义哪些方法?

1. 抽象方法：抽象方法是接口的核心部分，所有实现接口的类都必须实现这些方法。抽象方法默认是 public 和 abstract，这些修饰符可以省略。
2. 默认方法：默认方法是在 Java 8 中引入的，允许接口提供具体实现。实现类可以选择重写默认方法
3. 静态方法：静态方法也是在 Java 8 中引入的，它们属于接口本身，可以通过接口名直接调用，而不需要实现类的对象。
4. 私有方法：私有方法是在 Java 9 中引入的，用于在接口中为默认方法或其他私有方法提供辅助功能。这些方法不能被实现类访问，只能在接口内部使用。

16. 接口可以包含构造函数吗?

1. 不能。接口主要定义一组方法规范，没有具体的细节。
2. 构造函数就是初始化 class 的属性或者方法，在 new 的一瞬间自动调用。接口不能 new

17. 成员变量和局部变量有哪些区别?

1. 语法：成员变量属于类，局部变量属于方法。成员变量可以用 public、private、Static 等修饰符修饰，局部变量不能被修饰符修饰。 但：成员变量和局部变量都能被 final 修饰。
2. 变量在内存中存储方式：成员变量 Static 修饰，属于类；无 Static，属于实例。 对象存在堆内存。局部变量为基本数据类型，存在栈中；局部变量为引用类型，栈中存放指向堆内存对象的引用或指向常量池的地址
3. 变量在内存中的生命周期：成员变量随对象的创建而存在，局部变量随方法的调用而自动消失
4. 初值：成员变量没有赋初值默认赋值（final 修饰成员变量必须显式赋值），局部变量不会自动赋值

18. Static 了解吗?

1. Static 可以修饰变量、方法、代码块、内部类，及导入包（可以直接访问静态成员，无需类名引用，简化书写）。

19. 静态变量和实例变量的区别?

- 静态变量：Static 修饰，称为类变量，属于类。一个类无论创建多少个对象，内存中仅存在一个类变量的副本。可以实现让多个对象共享内存。
- 实例变量：必须依赖于某一实例，需要先创建对象，通过对象才能访问。

20. 静态方法和实例方法有什么不同?

- 静态方法：Static 修饰的方法，也称为类方法。调用时可以用**类名。方法名/对象名。方法名**

  静态方法内不能访问类的非静态成员和方法。
- 实例方法：对象名。方法名调用。实例方法可以访问类所有成员和方法

21. ❤️final 关键字的作用?

1. final 修饰类，表示类不能被继承。如：String、Integer 等
2. final 修饰方法，表示方法不能被重写。
3. final 修饰变量，表示这个变量是常量，被初始化后不能修改。
   - 基本数据类型：初始化后不能修改；
   - 引用类型：初始化后不能指向另一个对象；但是引用对象的内容可以改变

22. final、finally、finalize 的区别?

1. final 是修饰符，可以修饰类、变量、方法....（参照上一条）
2. finally 是异常处理中的一部分，无论 try 中是否抛出异常，finally 都会执行。通常用来释放资源，如关闭文件，数据库连接等。
3. finalize 是 Object 类的方法，用于垃圾回收器将对象从内存中清理出去前做一些必要的操作。不能显式调用，他由垃圾回收器在适当的时间自动调用。

23. 非静态内部类和静态内部类的区别?

1. 非静态内部类依赖于外部类的实例，而静态内部类不依赖于外部类的实例。
2. 非静态内部类可以访问外部类的实例变量和方法，而静态内部类只能访问外部类的静态成员。
3. 非静态内部类不能定义静态成员，而静态内部类可以定义静态成员。
4. 非静态内部类在外部类实例化后才能实例化，而静态内部类可以独立实例化。
5. 非静态内部类可以访问外部类的私有成员，而静态内部类不能直接访问外部类的私有成员，需要通过实例化外部类来访问。

24. 非静态内部类可以直接访问外部方法，编译器是怎么做到的?

1. 非静态内部类可以直接访问外部方法是因为编译器在生成字节码时会为非静态内部类维护一个指向外部类实例的引用。
2. 这个引用使得非静态内部类能够访问外部类的实例变量和方法。编译器会在生成非静态内部类的构造方法时，将外部类实例作为参数传入，并在内部类的实例化过程中建立外部类实例与内部类实例之间的联系，从而实现直接访问外部方法的功能。

25. 有一个父类和子类，都有静态的成员变量、静态构造方法和静态方法，在 new 一个子类对象的时候，加载顺序是怎么样的?

当实例化一个子类对象时，静态成员变量、静态构造方法和静态方法的加载顺序遵循以下步骤：

1. 在创建子类对象之前，首先会加载父类的静态成员变量和静态代码块（构造方法无法被 static 修饰，因此这里是静态代码块）。这个加载是在类首次被加载时进行的，且只会发生一次。
2. 接下来，加载子类的静态成员变量和静态代码块。这一过程也只发生一次，即当首次使用子类的相关代码时。
3. 之后，执行实例化子类对象的过程。这时会呼叫父类构造方法，然后是子类的构造方法。

具体加载顺序可以简要总结为：

- 父类静态成员变量、静态代码块（如果有）
- 子类静态成员变量、静态代码块（如果有）
- 父类构造方法（实例化对象时）
- 子类构造方法（实例化对象时）

四、深拷贝和浅拷贝

1.深拷贝和浅拷贝的区别?

1. 浅拷贝会创建一个新的对象，新对象的属性和原对象的属性完全相同。属性如果是基本数据类型，拷贝的是基本数据类型的值；如果属性是引用类型，拷贝的是引用地址。新旧对象共享一个引用对象。
    a. 浅拷贝的实现方式为：实现 Cloneable 接口并重写 `clone()` 方法。
2. 深拷贝也会创建一个对象，但会递归复制所有引用对象，与新对象完全保持独立。新对象与原对象的任何更改都不会相互影响。

2.实现深拷贝的三种方法是什么?

1. 实现 Cloneable 接口并重写 clone（） 方法：在 clone（） 方法中，通过递归克隆引用类型字段来实现深拷贝。
2. 使用序列化和反序列化：通过将对象序列化为字节流，再从字节流反序列化为对象来实现深拷贝。要求对象及其所有引用类型字段都实现 Serializable 接口。
3. 手动递归复制：针对特定对象结构，手动递归复制对象及其引用类型字段。适用于对象结构复杂度不高的情况。

3.什么是引用拷贝?

引用拷贝就是两个不同的引用指向同一个对象。

五、泛型

1. 什么是泛型?

泛型就是把类型给参数化。 它允许类、接口和方法在定义时使用一个或多个类型参数。

主要目的是在编译时提供更强的类型检查，并且在编译后能够保留类型信息，避免了在运行时出现类型转换异常。

2. 什么是泛型擦除?

Java 中，泛型只存在于源代码和编译阶段。在运行时，所有的泛型信息都会被擦除掉。所以，在运行时，所有的泛型都会被替换为他们的上限类;如果没指定上限，默认就会替换为 Object。

主要是为了向下兼容，因为 JDK5 之前是没有泛型的，为了让 JVM 保持向下兼容，就出了类型擦除这个策略。

3. 泛型擦除的影响?

- 无法在运行时，获取泛型类型的信息。例：不能使用 instanceOf 检查泛型类型。
- 不能创建泛型类型的数组。因为在运行时无法确定泛型类型
- 不能实例化泛型类型，因为在运行时泛型类型已经被擦除

4. 为什么需要泛型?

1. 适用于多种数据类型执行相同的代码。例：没有泛型则需要重新定义多个参数类型不同的方法。
2. 泛型中的类型在使用时指定，不需要强制类型转换（类型安全，编译器会检查类型）。
   a. List<String> list = new ArrayList<String>();只能放 String 类型的

六、对象

1. java 创建对象有哪些方式?

1. new 关键字：通过 new 关键字直接调用类的构造方法来创建对象。
2. 反射机制创建：反射机制允许在运行时创建对象，并且可以访问类的成员、方法、构造器
3. 使用 clone（）方法：类实现 cloneable 接口并重写 clone（）方法。
4. 使用序列化机制：通过序列化将对象转换为字节流，在通过反序列化从字节流中恢复对象。需要实现 Serializable 接口。

2. new 出的对象什么时候回收?

1. 通过过关键字 new 创建的对象，由 Java 的垃圾回收器（Garbage Collector）负责回收。垃圾回收器的工作是在程序运行过程中自动进行的，它会周期性地检测不再被引用的对象，并将其回收释放内存。

具体情况，垃圾回收器根据一些算法决定：

- 引用计数法：某个对象的引用计数为 0 时，表示该对象不再被引用，可以被回收。
- 可达性分析算法：从根对象（如方法区中的类静态属性、方法中的局部变量等）出发，通过对象之间的引用链进行遍历，如果存在一条引用链到达某个对象，则说明该对象是可达的，反之不可达，不可达的对象将被回收。
- 终结器（Finalizer）：如果对象重写了 finalize（）方法，垃圾回收器会在回收该对象之前调用 finalize（）方法，对象可以在 finalize（）方法中进行一些清理操作。然而，终结器机制的使用不被推荐，因为它的执行时间是不确定的，可能会导致不可预测的性能问题。

七、反射

1.什么是反射?

Java 反射机制是在运行状态中，对于任意一个类，都能够知道这个类中的所有属性和方法。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。

有以下特性：

1. 运行时类信息访问：反射机制允许程序在运行时获取类的完整结构信息，包括类名、包名、父类、实现的接口、构造函数、方法和字段等。

2. 动态对象创建：可以使用反射 API 动态地创建对象实例，即使在编译时不知道具体的类名。这是通过 Class 类的 newInstance（）方法或 Constructor 对象的 newInstance（）方法实现的。
3. 动态方法调用：可以在运行时动态地调用对象的方法，包括私有方法。这通过 Method 类的 invoke（）方法实现，允许你传入对象实例和参数值来执行方法。
4. 访问和修改字段值：反射还允许程序在运行时访问和修改对象的字段值，即使是私有的。这是通过 Field 类的 get（）和 set（）方法完成的。

2.反射在你平时写代码或者框架中的应用场景有哪些?

1. 加载数据库驱动：需要动态地根据实际情况加载驱动类 Class.forName（"com.mysql.cj.jdbc.Driver"）；
2. Spring 框架就大量使用了反射来动态加载和管理 Bean
3. Java 的动态代理（Dynamic Proxy）机制就使用了反射来创建代理类。代理类可以在运行时动态处理方法调用，这在实现 AOP 和拦截器时非常有用。
4. JUnit 和 TestNG 等测试框架使用反射机制来发现和执行测试方法。反射允许框架扫描类，查找带有特定注解（如 @Test）的方法，并在运行时调用它们。
5. 使用反射机制，根据这个字符串获得某个类的 Class 实例

3.Java 反射创建对象效率高还是通过 new 创建对象的效率高?

通过 new 创建对象的效率比较高。通过反射时，先查找类资源，使用类加载器创建，过程比较繁琐，所以效率较低，new 对象的方式在**编译期即可确定要加载的对象路径以及是否合法**，而通过反射的话在运行时才能确定，那这部分**安全合法性的检查就会影响类加载的效率**。

4.java 反射的作用

反射机制是在运行时，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意个对象，都能够调用它的任意一个方法。在 java 中，只要给定类的名字，就可以通过反射机制来获得类的所有信息。

八、注解

1.Java 注解的原理?

Java 的一种特殊的注释，让其他程序根据注解决定怎么执行程序，可视为特殊的注解，通过一系列方法对其进行解析。注解本质是一个继承了 Annotation 的特殊接口，其具体实现类是 Java 运行时生成的动态代理类。

我们通过反射获取注解时，返回的是 Java 运行时生成的动态代理对象。通过代理对象调用自定义注解的方法，会最终调用 AnnotationInvocationHandler 的 invoke 方法。

2.Java 注解的作用域?

Java 注解的作用域可以分为三种：

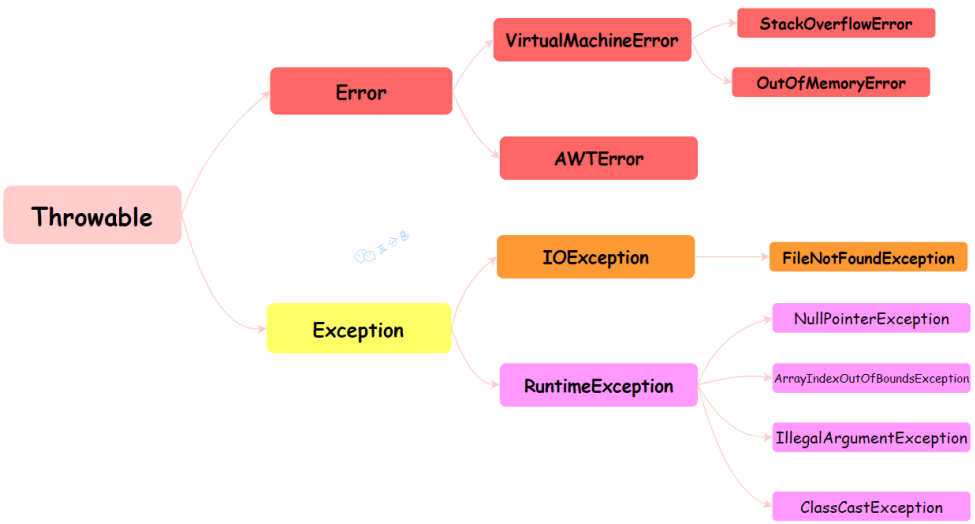
类级别作用域：用于描述类的注解，通常放置在类定义的上面，可以用来指定类的一些属性，如类的访问级别、继承关系、注释等。

方法级别作用域：用于描述方法的注解，通常放置在方法定义的上面，可以用来指定方法的一些属性，如方法的访问级别、返回值类型、异常类型、注释等。

字段级别作用域：用于描述字段的注解，通常放置在字段定义的上面，可以用来指定字段的一些属性，如字段的访问级别、默认值、注释等。

九、异常

1. 介绍一下异常



Java 的异常体系主要基于两大类：Throwable 类及其子类。Throwable 有两个重要的子类：Error 和 Exception。

1. Error（错误）：程序无法处理的严重问题。StackOverflowError 表示栈溢出。通常与 JVM 的运行状态有关，一旦发生，应用程序通常无法恢复。
2. Exception（异常）：表示程序本身可以处理的异常条件。异常分为两大类
    a. 编译时异常：这类异常在编译时必须被显式处理（捕获或声明抛出）。程序员必须处理这些可能出现的问题，增强程序的健壮性。
    b. 运行时异常：运行时错误导致，如空指针异常(NullPointerException)、数组越界(ArrayIndexOutOfBoundsException）等。运行时异常是不需要在编译时强制捕获或声明的。

2. Java 异常处理有哪些?

1. 遇到异常时可以不处理，直接通过throw 和 throws 抛出异常，交给上层调用者处理。throws用于声明可能会抛出的异常，throw 关键字用于抛出异常。
2. 使用 try-catch 捕获异常，处理异常。

3. 为什么不能在 Finally 块中使用 return?

try 块中的 return 语句执行成功后，并不马上返回，而是继续执行 finally 块中的语句，如果此处存在 return 语句，则在此直接返回，无情丢弃掉 try 块中的返回点。

4. return 先执行还是 finally 先执行?

举例：try块return 1；catch块return 2；finally块return 3；

没捕获到异常，finally块总是要运行，故控制台打印3，覆盖try中暂存的1；

如果try块中捕获到了异常，控制台就会打印3，覆盖暂存的2

5. 抛出异常为什么不用 throws?

如果异常是未检查异常或者在方法内部被捕获和处理了，那么就不需要使用 throws。

- 不可查异常（Unchecked Exceptions）：不可查异常，即 Error 和 RuntimeException 及其子类，可以不管，在运行时会处理，对于这些异常，不需要在方法签名中使用 throws 来声明。示例包括 NullPointerException、ArrayIndexOutOfBoundsException 等。
- 可查异常（checked Exceptions）：可查异常，必须 try-catch 捕获或 Throws 抛出，否则会编译异常。在方法内部捕获了可能抛出的异常，并在方法内部处理它们，而不是通过 throws 子句将它们传递到调用者。这种情况下，方法可以处理异常而无需在方法签名中使用 throws。

6. try catch 中的语句运行情况?

try 块中的代码将按顺序执行，如果抛出异常，将在 catch 块中进行匹配和处理，然后程序将继续执行 catch 块之后的代码。如果没有匹配的 catch 块，异常将被传递给上一层调用的方法

7. try{return “a”} fianlly{return “b”}这条语句返回啥

直接返回 b，丢弃 try 块中的返回点。（同 3）

十、线程

1. 线程安全的理解?

当多个线程并发共享互斥资源时，读写互斥资源的代码逻辑能正常处理，获得正确结果，不会相互干扰。

2. 线程的创建的三种方式?

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 和 Future 创建线程：通过 Call（）方法返回具体的结果

3. Thread 和 Runnable 的区别?

1. 定义： runnable 是接口，thread 是类，这个类也是实现了该接口的一个扩展而已
2. 用法：继承 thread 以后可直接启动；实现 Runnable 以后还是需要 new thread 才能启动
3. 限制：类只能单继承，接口可以多实现，如果已有继承关系，就只能实现接口

4. 线程安全的集合类?

Vector、Stack、HashTable、ConcurrentHashMap 是线程安全的

十一、Objec 类

1. == 与 equals 有什么区别?

，== 操作符和 equals() 方法用于比较两个对象

1. ==既可以判断基本数据类型，又可以判断引用数据类型
    - 对于基本数据类型：==判断值是否相等
    - 对于引用数据类型：==判断两个对象的地址值是否相等，即是否为同一对象
2. equals 是 Object 类的方法，只能判断引用类型
    - 默认判断的是地址值是否相等（对象）
    - 子类重写 equals 方法，判断内容是否相等

2. 为什么重写 equals 时必须重写 hashcode 方法?

因为 HashMap 需要通过这两个方法来正确存储和查找对象。

1. HashMap 通过对象的哈希码将其存储在不同的“bucket”中，当查找对象时，它需要使用 key 的哈希码来确定对象在哪个桶中，然后再通过 `equals()` 方法找到对应的对象。
2. 如果重写了 `equals()` 方法而没有重写 `hashCode()` 方法，那么被认为相等的对象可能会有不同的哈希码，从而导致无法在 HashMap 中正确处理这些对象

3. Object 类常见方法?

getClass( ) 、finalize(java9弃用) 、hashcode( ) 、equals()、clone( ) 、toString( ) 、notify( ) 、notifyAll( ) 、wait()、

十二、java1.8 新特性

1.stream 的 API 介绍一下?

Java 8 引入了 Stream API，它提供了一种高效且易于使用的数据处理方式，特别适合集合对象的操作，如过滤、映射 map、排序 sorted、遍历等。真正的实现了函数式编程。Stream API 不仅可以提高代码的可读性和简洁性，还能利用多核处理器的优势进行并行处理。

1. 过滤操作：先创建一个集合，调用。stream（）方法创建了一个流，使用。filter（）中间操作筛选出符合条件的字符串，最后使用。collect（Collectors.toList（））终端操作将结果收集到一个新的集合中。代码更加简洁明了，逻辑一目了然。
2. 计算集合中所有数字的和：先创建一个集合，先使用。mapToInt（）将 Integer 流转换为 IntStream（这是为了高效处理基本类型），然后直接调用。sum（）方法来计算总和，极大地简化了代码。

2.Stream 流用过吗?

Stream 流，简单来说，使用 java.util.Stream 对一个包含一个或多个元素的集合做各种操作。这些操作可能是 中间操作 亦或是 终端操作。 终端操作会返回一个结果，而中间操作会返回一个 Stream 流。

Stream 流一般用于集合，我们对一个集合做几个常见操作：

代码块

```
List<String> stringCollection = new ArrayList<>();
stringCollection.add("ddd2");
stringCollection.add("aaa2");
stringCollection.add("bbb1");
stringCollection.add("aaa1");
```

- Filter 过滤

代码块

```
stringCollection
    .stream()
    .filter((s) -> s.startsWith("a"))
    .forEach(System.out::println);

// "aaa2", "aaa1"
```

- Map 转换

代码块

```
    stringCollection
    .stream()
    .map(String::toUpperCase)
    .sorted((a, b) -> b.compareTo(a))
    .forEach(System.out::println);
```

```
6
7   // "DDD2", "DDD1", "CCC", "BBB3", "BBB2", "AAA2", "AAA1"
```

❤️3.completableFuture 怎么用的?

- CompletableFuture 是 Java 8 引入的一个用于异步编程的工具类。它是对 Java 早期的 Future 接口的增强和扩展，目的是为了更方便地处理异步计算任务和异步操作的结果。在处理一些耗时的操作（如 I/O 操作、网络请求等）时，使用 CompletableFuture 可以让程序在等待这些操作完成的同时去执行其他任务，从而提高程序的性能和资源利用率。

十三、序列化

1. 对象序列化机制是什么?

- 对象序列化机制（Object serializable）是 java 中的对象持久化机制。
- 通过序列化，可将对象的状态保存为字节流；在有需要时，可以将字节数组反序列化的方式，再转换为对象。对象序列化可以很容易在 JVM 的活动对象和字节流之间转换

2. Serializable 接口有什么用?

Serializable接口用于标记一个类可以被序列化。

3. 什么是序列化和反序列化?

- 序列化：把 java 对象转换成成字节流的过程（使用 objectOutputStream 流实现，将内存中的 Java 对象保存到文件中或通过网络传输出去）
- 反序列化：把字节数流转换成 java 对象的过程（使用 ObjectInputStream 流实现，将文件中的数据或网络传输过来的数据还原为内存中的 java 对象）

4. 为什么需要序列化?

- JVM 中的对象，只有在 JVM 运行状态的时候才会存在，一旦 JVM 停止，这些对象也就随之消失了。在真实的情况下 ，我们需要 将对象持久化下来。并且需要在使用时，重新读取出来？此时就会用到序列化机制。

5. serialVersionUID 是干嘛的?

- 用来唯一标识当前的类。通过 SerialVersionUID 来确保反序列化时的对象与序列化时的对象是同一个。如果不声明全局常量 SerialVersionUID，系统会自动生成一个针对当前类的 SerialVersionUID。如果修改此类的话，会导致 SerialVersionUID 变化，进而导致反序列化时，出现 InvalidClassException 异常。
- Java 虚拟机是否允许反序列化，不仅取决于类路径和功能代码是否一致，还有一个非常重要的因素就是序列化 ID 是否一致。

6. 说说有几种序列化方式?

常见的有三种

- Java 对象序列化： Java 原生序列化方法即通过 Java 原生流(InputStream 和 OutputStream 之间的转化)的方式进行转化，一般是对象输出流 ObjectOutputStream和对象输入流 ObjectInputStream。
- Json 序列化：这个可能是我们最常用的序列化方式，Json 序列化的选择很多，一般会使用 jackson 包，通过 ObjectMapper 类来进行一些操作，比如将对象转化为 byte 数组或者将 json 串转化为对象。
- ProtoBuff 序列化：ProtocolBuffer 是一种轻便高效的结构化数据存储格式，ProtoBuff 序列化对象可以很大程度上将其压缩，可以大大减少数据传输大小，提高系统性能。

7. 有些变量不想序列化怎么办?

使用transient关键字修饰。

8. 怎么把一个对象从一个 jvm 转移到另一个 jvm?

1. 使用序列化和反序列化：将对象序列化为字节流，并将其发送到另一个 JVM，然后在另一个 JVM 中反序列化字节流恢复对象。这可以通过 Java 的 ObjectOutputStream 和 ObjectInputStream 来实现。
2. 使用消息传递机制：利用消息传递机制，比如使用消息队列（如 RabbitMQ、Kafka）或者通过网络套接字进行通信，将对象从一个 JVM 发送到另一个。这需要自定义协议来序列化对象并在另一个 JVM 中反序列化。
3. 使用远程方法调用（RPC）：可以使用远程方法调用框架，如 gRPC，来实现对象在不同 JVM 之间的传输。远程方法调用可以让你在分布式系统中调用远程 JVM 上的对象的方法。
4. 使用共享数据库或缓存：将对象存储在共享数据库（如 MySQL、PostgreSQL）或共享缓存（如 Redis）中，让不同的 JVM 可以访问这些共享数据。这种方法适用于需要共享数据但不需要直接传输对象的场景。

9. 序列化和反序列化让你自己实现你会怎么做?

Java 默认的序列化虽然实现方便，但却存在安全漏洞、不跨语言以及性能差等缺陷。

1. 无法跨语言： Java 序列化目前只适用基于 Java 语言实现的框架，其它语言大部分都没有使用 Java 的序列化框架，也没有实现 Java 序列化这套协议。因此，如果是两个基于不同语言编写的应用程序相互通信，则无法实现两个应用服务之间传输对象的序列化与反序列化。
2. 容易被攻击：Java 序列化是不安全的，我们知道对象是通过在 ObjectInputStream 上调用 readObject（） 方法进行反序列化的，这个方法其实是一个神奇的构造器，它可以将类路径上几乎所有实现了 Serializable 接口的对象都实例化。这也就意味着，在反序列化字节流的过程中，该方法可以执行任意类型的代码，这是非常危险的。
3. 序列化后的流太大：序列化后的二进制流大小能体现序列化的性能。序列化后的二进制数组越大，占用的存储空间就越多，存储硬件的成本就越高。如果我们是进行网络传输，则占用的带宽就更多，这时就会影响到系统的吞吐量。

我会考虑用主流序列化框架，比如 FastJson、Protobuff 来替代 Java 序列化。

- 如果追求性能的话，Protobuf 序列化框架会比较合适，Protobuff 的这种数据存储格式，不仅压缩存储数据的效果好， 在编码和解码的性能方面也很高效。Protobuff 的编码和解码过程结合。proto 文件格式，加上 Protocol Buffer 独特的编码格式，只需要简单的数据运算以及位移等操作就可以完成编码与解码。可以说 Protobuff 的整体性能非常优秀。

10. 将对象转为二进制字节流具体怎么实现?

在 Java 中通过序列化对象流来完成序列化和反序列化：

1. ObjectOutputStream：通过 writeObject（）方法做序列化操作。
2. ObjectInputStrean：通过 readObject（）方法做反序列化操作。

只有实现了 Serializable 或 Externalizable 接口的类的对象才能被序列化，否则抛出异常！

首先实现对象序列化，让类实现 Serializable 接口；接着创建输出流并写入对象。接着实现对象反序列化，创建输出流并读取对象。通过以上步骤，对象 j 会被序列化并写入到文件中，然后通过反序列化操作，从文件中读取字节流并恢复为对象。这种方式可以方便地将对象转换为字节流用于持久化存储、网络传输等操作。需要注意的是，要确保类实现了 Serializable 接口，并且所有成员变量都是 Serializable 的才能被正确序列化。

十四、设计模式

1.volatile 和 sychronized 如何实现单例模式?

```java
public class SingleTon {

    // volatile 关键字修饰变量 防止指令重排序
    private static volatile SingleTon instance = null;
    private SingleTon(){}

    public static  SingleTon getInstance(){
        if(instance == null){
          //同步代码块 只有在第一次获取对象的时候会执行到 ，
          //第二次及以后访问时 instance变量均非null故不会往下执行了 直接返回啦
            synchronized(SingleTon.class){
                if(instance == null){
                    instance = new SingleTon();
                }
            }
        }
        return instance;
    }
}
```

正确的双重检查锁定模式需要需要使用 volatile。volatile 主要包含两个功能。

1. 保证可见性。使用 volatile 定义的变量，将会保证对所有线程的可见性。
2. 禁止指令重排序优化。

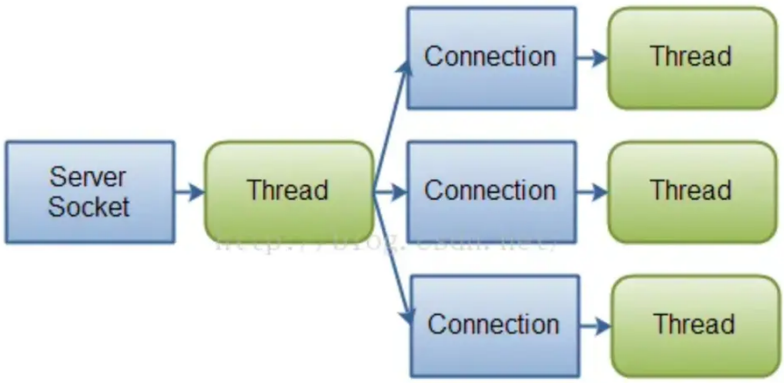
由于 volatile 禁止对象创建时指令之间重排序，所以其他线程不会访问到一个未初始化的对象，从而保证安全性。

十五、I/O

1. Java 怎么实现网络 IO 高并发编程?

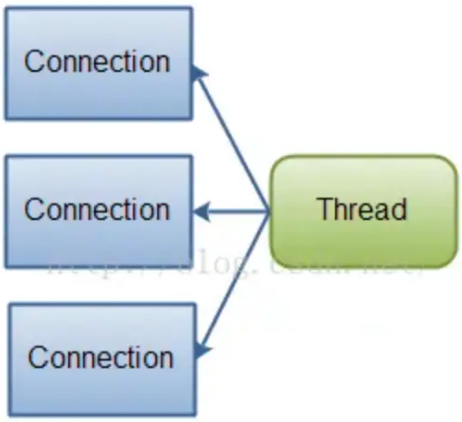
可以用 Java NIO ，是一种同步非阻塞的 I/O 模型，也是 I/O 多路复用的基础。

- 传统的 BIO 里面 socket.read（），如果 TCP RecvBuffer 里没有数据，函数会一直阻塞，直到收到数据，返回读到的数据， 如果使用 BIO 要想要并发处理多个客户端的 i/o，那么会使用多线程模式，一个线程专门处理一个客户端 io，这种模式随着客户端越来越多，所需要创建的线程也越来越多，会急剧消耗系统的性能。



Java IO: A classic IO server design - one connection handled by one thread.

- NIO 是基于 I/O 多路复用实现的，它可以只用一个线程处理多个客户端 I/O，如果你需要同时管理成千上万的连接，但是每个连接只发送少量数据，例如一个聊天服务器，用 NIO 实现会更好一些。



Java NIO: A single thread managing multiple connections.

2.BIO、NIO、AIO 区别是什么?

1. BIO：采用阻塞式 I/O 模型，线程在执行 I/O 操作时被阻塞，无法处理其他任务，适用于连接数较少的场景。
2. NIO：采用非阻塞 I/O 模型，线程在等待 I/O 时可执行其他任务，通过 Selector 监控多个 Channel 上的事件，适用于连接数多但连接时间短的场景。
3. AIO：使用异步 I/O 模型，线程发起 I/O 请求后立即返回，当 I/O 操作完成时通过回调函数通知线程，适用于连接数多且连接时间长的场景。

白话解析：

BIO （Blocking I/O）：同步阻塞 I/O 模式。

NIO （New I/O）：同步非阻塞模式。

AIO （Asynchronous I/O）：异步非阻塞 I/O 模型。

1. 同步阻塞：这种模式下，我们的工作模式是先来到厨房，开始烧水，并坐在水壶面前一直等着水烧开。
2. 同步非阻塞模式：这种模式下，我们的工作模式是先来到厨房，开始烧水，但是我们不一直坐在水壶前面等，而是回到客厅看电视，然后每隔几分钟到厨房看一下水有没有烧开。
3. 异步非阻塞 I/O 模型：这种模式下，我们的工作模式是先来到厨房，开始烧水，我们不一直坐在水壶前面等，也不隔一段时间去看一下，而是在客厅看电视，水壶上面有个开关，水烧开之后他会通知我。

- 阻塞 VS 非阻塞：人是否坐在水壶面前一直等
- 同步 VS 非同步：水壶烧开是否主动通知人过去

适用场景：

BIO 方式适用于连接数目比较小且固定的架构，这种方式对服务器资源要求比较高，并发局限于应用中，JDK1.4 以前的唯一选择，但程序直观简单易理解。

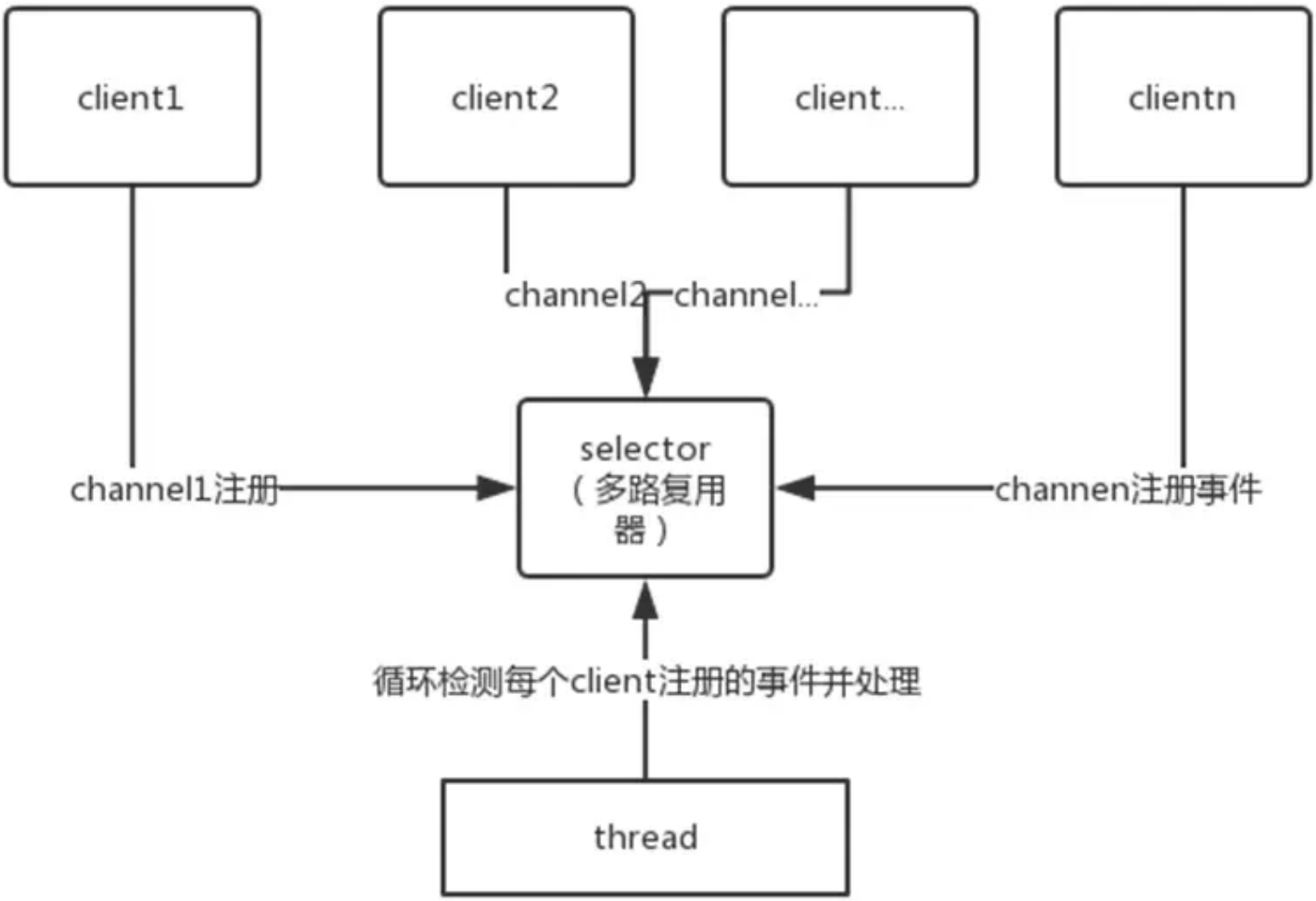
NIO 方式适用于连接数目多且连接比较短（轻操作）的架构，比如聊天服务器，并发局限于应用中，编程比较复杂，JDK1.4 开始支持。

AIO 方式适用于连接数目多且连接比较长（重操作）的架构，比如相册服务器，充分调用 OS 参与并发操作，编程比较复杂，JDK7 开始支持。

3.NIO 是怎么实现的?

- NIO 是一种同步非阻塞的 IO 模型，所以也可以叫 NON-BLOCKINGIO。同步是指线程不断轮询 IO 事件是否就绪，非阻塞是指线程在等待 IO 的时候，可以同时做其他任务。
- 同步的核心就 Selector（I/O 多路复用），Selector 代替了线程本身轮询 IO 事件，避免了阻塞同时减少了不必要的线程消耗；非阻塞的核心就是通道和缓冲区，当 IO 事件就绪时，可以通过写到缓冲区，保证 IO 的成功，而无需线程阻塞式地等待。
- NIO 由一个专门的线程处理所有 IO 事件，并负责分发。事件驱动机制，事件到来的时候触发操作，不需要阻塞的监视事件。线程之间通过 wait，notify 通信，减少线程切换。
- NIO 主要有三大核心部分：Channel（通道），Buffer（缓冲区）， Selector。传统 IO 基于字节流和字符流进行操作，而 NIO 基于 Channel 和 Buffer（缓冲区）进行操作，数据总是从通道读取到缓冲区中，或者从缓冲区写入到通道中。

- Selector（选择区）用于监听多个通道的事件（比如：连接打开，数据到达）。因此，单个线程可以监听多个数据通道。



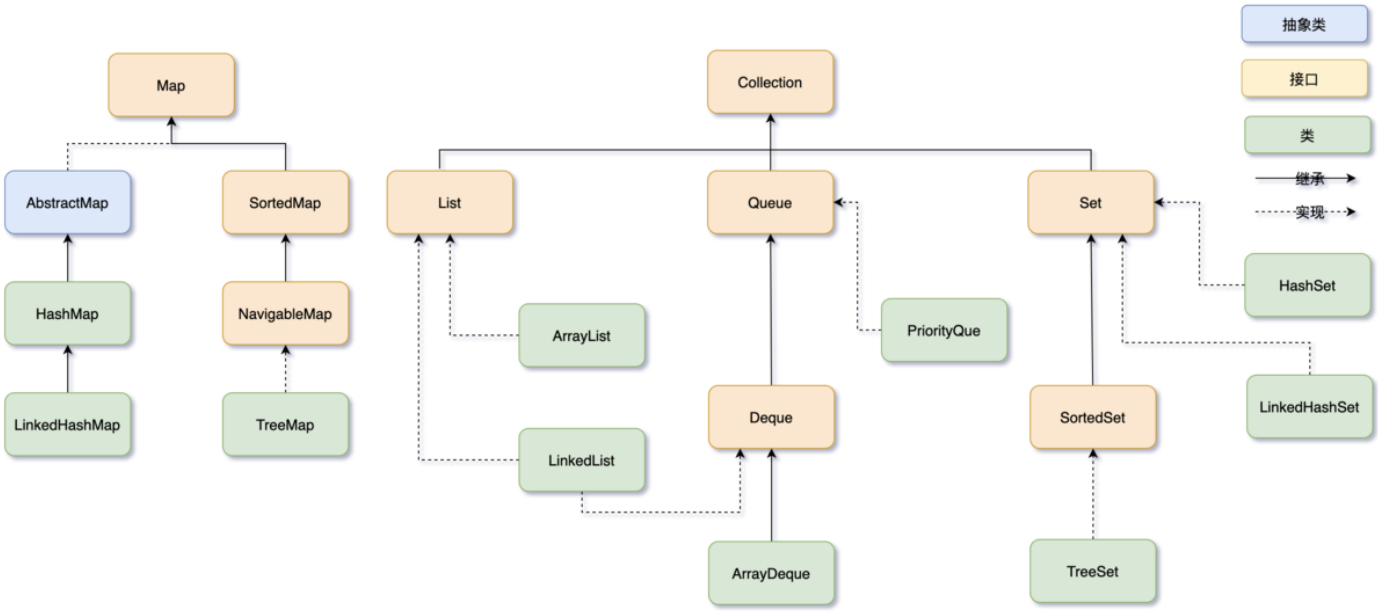
3.你知道有哪个框架用到 NIO 了吗?

Netty.

Netty 的 I/O 模型是基于非阻塞 I/O 实现的，底层依赖的是 NIO 框架的多路复用器 Selector。采用 epoll 模式后，只需要一个线程负责 Selector 的轮询。当有数据处于就绪状态后，需要一个事件分发器（Event Dispather），它负责将读写事件分发给对应的读写事件处理器（Event Handler）。事件分发器有两种设计模式：Reactor 和 Proactor，Reactor 采用同步 I/O， Proactor 采用异步 I/O。

十六、集合

1. Java 容器（集合）都有哪些?



Java 容器分为Map和 Collection两大类，其下又有很多子类。

单列集合 List 和 Set。他们继承了 Collection 接口。

1. List是有序的集合，可以包含重复的元素；List 的实现类是 ArrayList、LinkedList、Vector；
2. Set 代表无序、不可重复的集合；Set 的实现类是 HashSet、LinkedHashSet、TreeSet。
3. Queue 用于保持元素队列的集合；实现类包括双端队列 ArrayDeque，以及优先级队列 PriorityQueue。

Map 是双列结合，没有继承 Collection 接口。它的实现类有 HashMap、LinkedHashMap、TreeMap、ConcurrentHashMap、HashTable

2. Collection 和 Collections 有什么区别?

- Collection 是一个集合接口，它提供了对集合对象进行基本操作的通用接口方法，所有集合都是它的子类，比如 List、Set 等。
- Collections 是一个包装类，包含了很多静态方法，不能被实例化，就像一个工具类，比如提供的排序方法： Collections.sort（list）

List

3. ArrayList 和 LinkedList 的区别是什么?

1. 数据结构：ArrayList 是基于可变数组， LinkedList 基于双向链表。
2. 随机访问效率：ArrayList 比 LinkedList 随机访问的时候效率要高。ArrayList 可以根据索引；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。

3. 增加和删除效率：在非首尾的增加和删除操作，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。

综合来说，在需要频繁读取集合中的元素时，更推荐使用 ArrayList，而在插入和删除操作较多时，更推荐使用 LinkedList。

4. ArrayList 的扩容原理?

了解。当往 ArrayList 中添加元素时，会先检查是否需要扩容，如果当前容量+1 超过数组长度，就会进行扩容。

扩容后的新数组长度是原来的 1.5 倍，然后再把原数组的值拷贝到新数组中。

1. 利用无参构造创建集合，底层默认创建长度为 0 的数组；添加第一个元素时，长度会变为 10；存满时，扩容至 1.5 倍。
2. 若使用指定大小的构造器，则初始容量为指定大小，再扩容直接变为 1.5 倍
3. transient Object[] elementData; 该属性不能被序列化

5. ArrayList 怎么序列化的知道吗?

在 ArrayList 中，writeObject 方法被重写了，用于自定义序列化逻辑：只序列化有效数据，因为 elementData 数组的容量一般大于实际的元素数量，声明的时候也加了 transient 关键字。

6. 快速失败fail-fast了解吗?

fail—fast 是 Java 集合的一种错误检测机制。

- 在用迭代器遍历集合对象时，如果线程 A 遍历过程中，线程 B 对集合对象的内容进行了修改，就会抛出 Concurrent Modification Exception
- 迭代器在遍历时直接访问集合中的内容，并且在遍历过程中使用一个 modCount 变量。集合在被遍历期间如果内容发生变化，就会改变modCount的值。每当迭代器使用 hashNext()/next()遍历下一个元素之前，都会检测 modCount 变量是否为 expectedmodCount 值，是的话就返回遍历；否则抛出异常，终止遍历。
- 异常的抛出条件是检测到 modCount！=expectedmodCount 这个条件。如果集合发生变化时修改 modCount 值刚好又设置为了 expectedmodCount 值，则异常不会抛出。因此，不能依赖于这个异常是否抛出而进行并发操作的编程，这个异常只建议用于检测并发修改的 bug。
- java.util 包下的集合类都是快速失败的，不能在多线程下发生并发修改（迭代过程中被修改），比如 ArrayList 类。

7. 有哪几种实现 ArrayList 线程安全的方法?

常用的有两种。

1. 可以使用 Collections.synchronizedList() 方法，它可以返回一个线程安全的 List。

   SynchronizedList list = Collections.synchronizedList(new ArrayList());

   内部是通过 synchronized 关键字加锁来实现的。

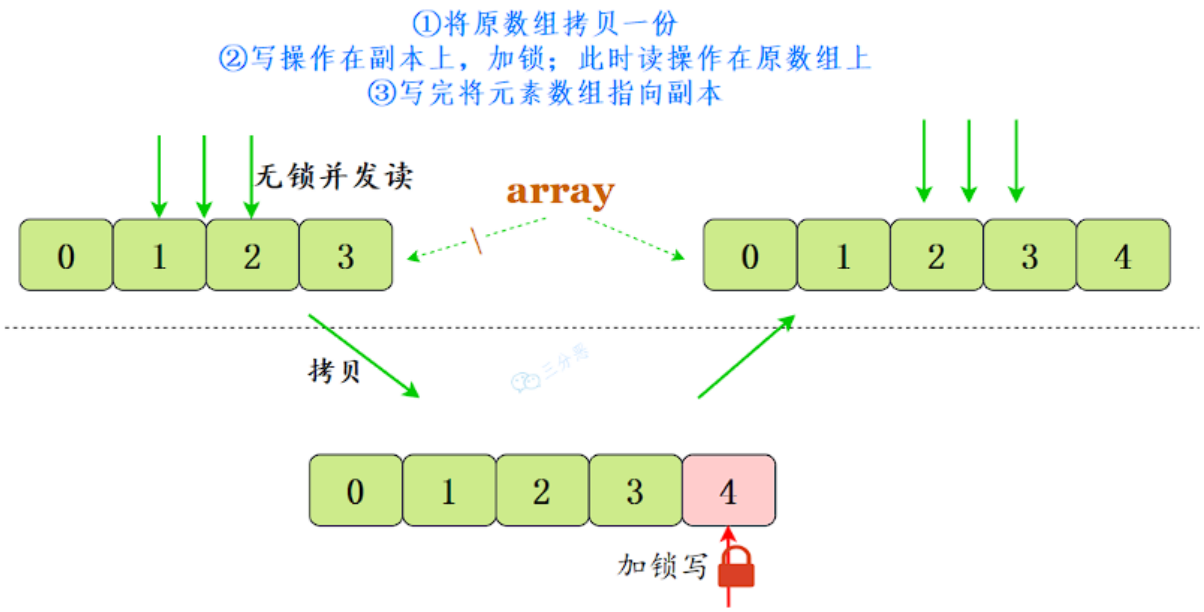
2. 可以直接使用 CopyOnWriteArrayList，它是线程安全的 ArrayList，遵循写时复制的原则。CopyOnWrite 就是当我们往一个容器添加元素的时候，不直接往容器中添加，而是先复制出一个新的容器，然后在新的容器里添加元素，添加完之后，再将原容器的引用指向新的容器。多个线程在读的时候，不需要加锁，因为当前容器不会添加任何元素。这样就实现了线程安全。

   CopyOnWriteArrayList list = new CopyOnWriteArrayList();

8. CopyOnWriteArrayList 了解多少?

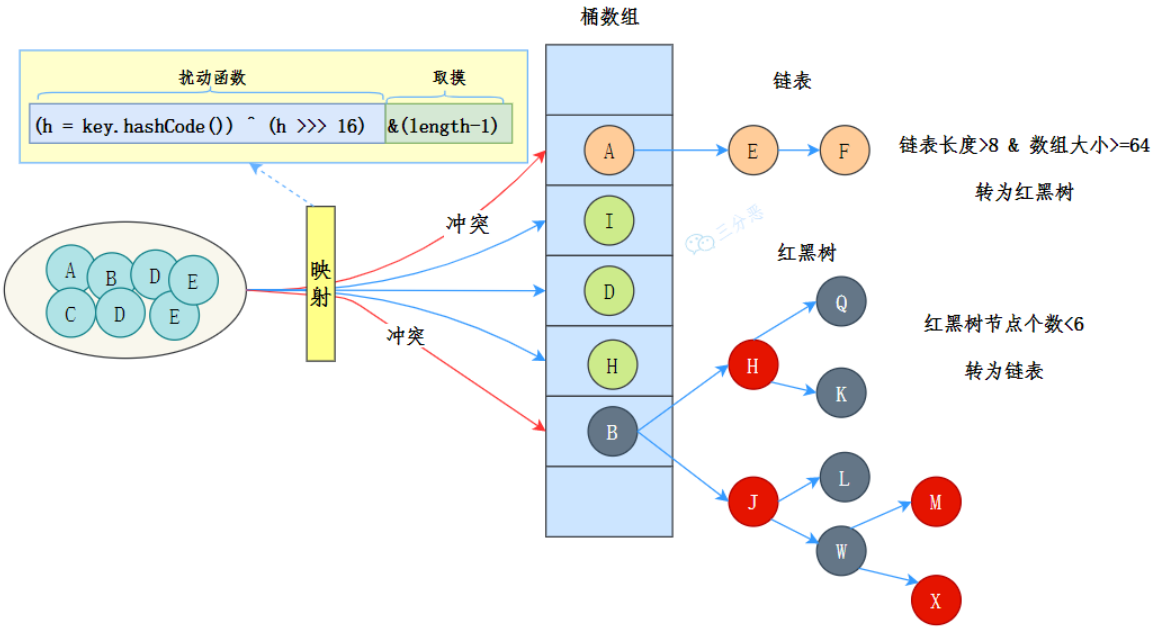
CopyOnWriteArrayList 就是线程安全版本的 ArrayList。CopyOnWrite——写时复制。

CopyOnWriteArrayList 采用了一种读写分离的并发策略。CopyOnWriteArrayList 容器允许并发读，读操作是无锁的。至于写操作，比如说向容器中添加一个元素，首先将当前容器复制一份，然后在新副本上执行写操作，结束之后再将原容器的引用指向新容器。



Map

9. 能说一下 HashMap 的底层数据结构吗?



HashMap 底层基于哈希表：

- 1.JDK8 以前是：数组+链表
- 2.JDK8 以后是：数组+链表+红黑树

1. 数组用来存储键值对，每个键值对可以通过索引直接拿到，索引是通过对键的哈希值进行进一步的 hash() 处理得的。
2. 当多个键经过哈希处理后得到相同的索引时，需要通过链表来解决哈希冲突——将具有相同索引的键值对通过链表存储起来。
3. 不过，链表过长时，查询效率会比较低，于是当链表的长度超过 8 时（且数组的长度大于 64），链表就会转换为红黑树。红黑树的查询效率是 O(logn)，比链表的 O(n) 要快。
4. hash() 方法的目标是尽量减少哈希冲突，保证元素能够均匀地分布在数组的每个位置上。

代码块

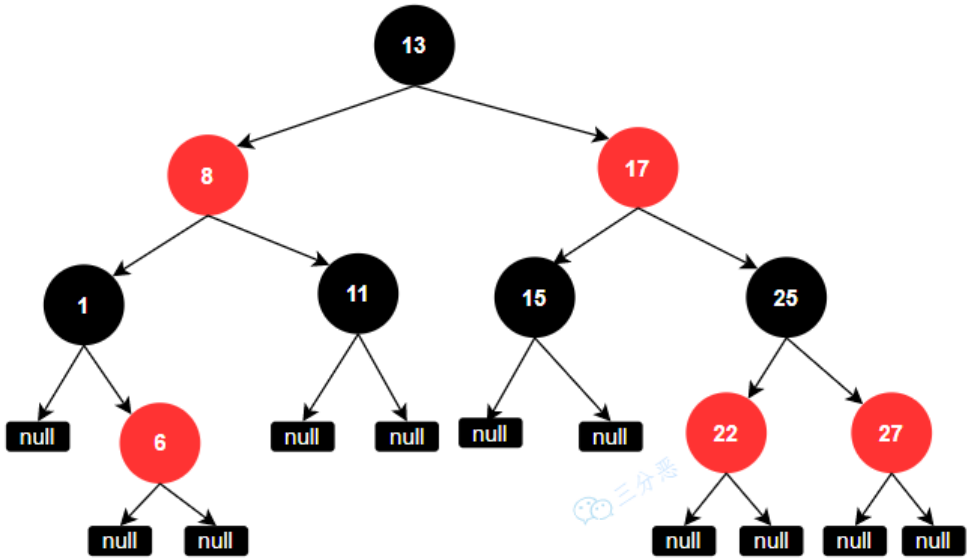
```
1   static final int hash(Object key) {
2   int h;
3   return (key == null)? 0 : (h = key.hashCode()) ^ (h >>> 16);
4   }
5    # 扰动函数
```

5. 如果键的哈希值已经在数组中存在，其对应的值将被新值覆盖。
6. HashMap 的初始容量是 16，随着元素的不断添加，HashMap 就需要进行扩容，阈值是capacity * loadFactor，capacity 为容量，loadFactor 为负载因子，默认为 0.75。
7. 扩容后的数组大小是原来的 2 倍，然后把原来的元素重新计算哈希值，放到新的数组中。

10. 你对红黑树了解多少？

红黑树是一种自平衡的二叉查找树：

1. 每个节点要么是红色，要么是黑色；
2. 根节点永远是黑色；
3. 所有的叶子节点都是是黑色的（下图中的 NULL 节点）；
4. 红色节点的子节点一定是黑色的；
5. 从任一节点到其每个叶子的所有简单路径都包含相同数目的黑色节点。



11. HashMap底层为什么用红黑树？

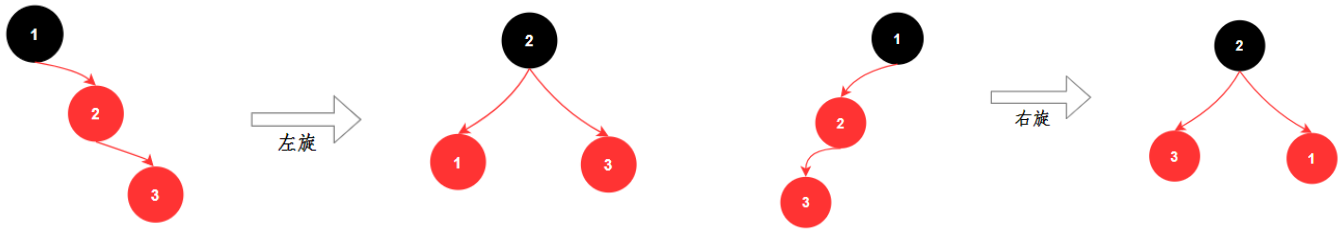
（回答逻辑从为什么不用：为什么不用二叉树？为什么不用平衡二叉树？）

- 二叉树是最基本的树结构，每个节点最多有两个子节点，但是二叉树容易出现极端情况，比如插入的数据是有序的，那么二叉树就会退化成链表，查询效率就会变成 O(n)。
- 平衡二叉树比红黑树的要求更高，每个节点的左右子树的高度最多相差 1，这种高度的平衡保证了极佳的查找效率，但在进行插入和删除操作时，可能需要频繁地进行旋转来维持树的平衡，维护成本更高。
- 链表的查找时间复杂度是 O(n)，当链表长度较长时，查找性能会下降。红黑树是一种折中的方案，查找、插入、删除的时间复杂度都是 O(log n)。

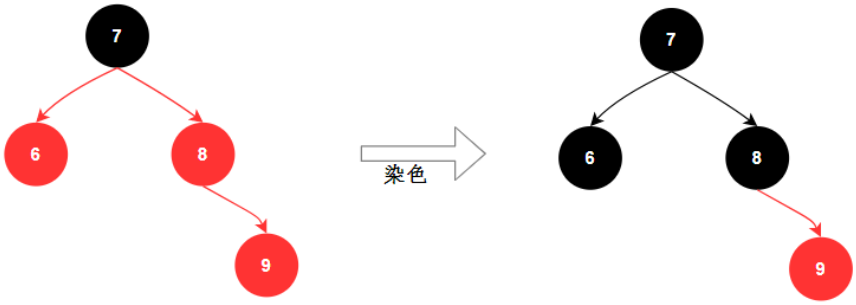
12. 红黑树怎么保持平衡的？

旋转和染色。

1. 通过左旋和右旋来调整树的结构，避免某一侧过深。



2. 染色，修复红黑规则，从而保证树的高度不会失衡。



13. 说一下 HashMap 的实现原理?

HashMap 基于 Hash 算法实现的，我们通过 put（key，value）存储，get（key）来获取。当传入 key 时，HashMap 会根据 key。 hashCode（） 计算出 hash 值，根据 hash 值将 value 保存在 bucket 里。当计算出的 hash 值相同时，我们称之为 hash 冲突，HashMap 的做法是用链表和红黑树存储相同 hash 值的 value。当 hash 冲突的个数比较少时，使用链表否则使用红黑树。

14. HashMap 的 put 流程知道吗?

哈希寻址 → 处理哈希冲突（链表还是红黑树）→ 判断是否需要扩容 → 插入/覆盖节点。

1. 第一步，通过 hash 方法进一步扰动哈希值，以减少哈希冲突。
2. 第二步，进行第一次的数组扩容；并使用哈希值和数组长度进行取模运算，确定索引位置。

如果当前位置为空，直接将键值对插入该位置；否则判断当前位置的第一个节点是否与新节点的 key 相同，如果相同直接覆盖 value，如果不同，说明发生哈希冲突。

如果是链表，将新节点添加到链表的尾部；如果链表长度大于等于 8，则将链表转换为红黑树。

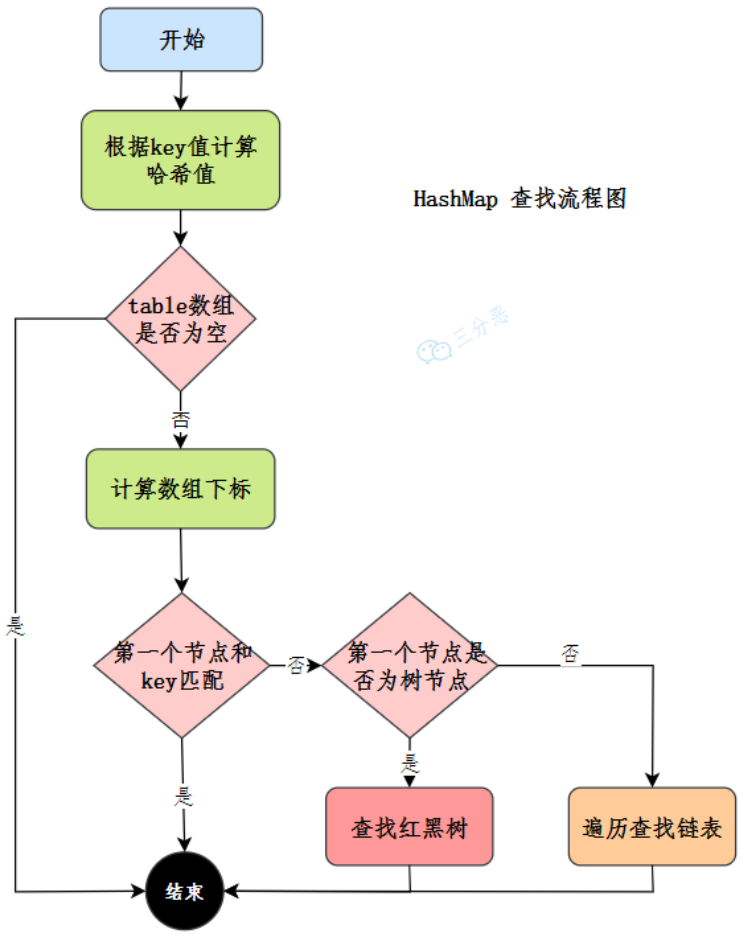
3. 每次插入新元素后，检查是否需要扩容，如果当前元素个数大于阈值（capacity * loadFactor），则进行扩容，扩容后的数组大小是原来的 2 倍；并且重新计算每个节点的索引，进行数据重新分布。

15. 只重写元素的 equals 方法没重写 hashCode，put 的时候会发生什么?

如果只重写 equals 方法，没有重写 hashCode 方法，那么会导致 equals 相等的两个对象，hashCode 不相等，这样的话，两个对象会被 put 到数组中不同的位置，导致 get 的时候，无法获取到正确的值。

16. HashMap 怎么查找元素的呢?

通过哈希值定位索引 → 定位桶 → 检查第一个节点 → 遍历链表或红黑树查找 → 返回结果。



HashMap 查找流程图

17. HashMap 的 hash 函数是怎么设计的?

先拿到 key 的哈希值，是一个 32 位的 int 类型数值，然后再让哈希值的高 16 位和低 16 位进行异或操作，这样能保证哈希分布均匀。

代码块

```
1   static final int hash(Object key) {
2       int h;
3       // 如果 key 为 null，返回 0；否则，使用 hashCode 并进行扰动
4       return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
```

```
5   }
```

18. 为什么 hash 函数能减少哈希冲突?

哈希表的索引是通过 h & (n-1) 计算的，n 是底层数组的容量；n-1 和某个哈希值做 & 运算，相当于截取了最低的四位。如果数组的容量很小，只取 h 的低位很容易导致哈希冲突。

通过异或操作将 h 的高位引入低位，可以增加哈希值的随机性，从而减少哈希冲突。

19. 如果初始化 HashMap，传一个 17 的容量，它会怎么处理?

HashMap 会将容量调整到大于等于 17 的最小的 2 的幂次方，也就是 32。这是因为哈希表的大小最好是 2 的 N 次幂，这样可以通过 (n - 1) & hash 高效计算出索引值。

20. 解决哈希冲突有哪些方法?

我知道的有 3 种，再哈希法、开放地址法和拉链法。

1. 再哈希法:准备两套哈希算法，当发生哈希冲突的时候，使用另外一种哈希算法，直到找到空槽为止。对哈希算法的设计要求比较高。
2. 开放地址法:遇到哈希冲突的时候，就去寻找下一个空的槽。具体有三种方法
    a. 线性探测：从冲突的位置开始，依次往后找，直到找到空槽。
    b. 二次探测：从冲突的位置 x 开始，第一次增加 1^2个位置，第二次增加 2^2，直到找到空槽。
    c. 双重哈希：和再哈希法类似，准备多个哈希函数，发生冲突的时候，使用另外一个哈希函数。
3. 拉链法：链地址法，当发生哈希冲突的时候，使用链表将冲突的元素串起来。HashMap 采用的正是拉链法。

21. 为什么 HashMap 链表转红黑树的阈值为 8 呢?

树化发生在 table 数组的长度大于 64，且链表的长度大于 8 的时候。

根据源码：红黑树节点的大小大概是普通节点大小的两倍，所以转红黑树，牺牲了空间换时间，更多的是一种兜底的策略，保证极端情况下的查找效率。

- 阈值为什么要选 8 呢？和统计学有关。理想情况下，使用随机哈希码，链表里的节点符合泊松分布，出现节点个数的概率是递减的，节点个数为 8 的情况，发生概率仅为0.00000006。

- 至于红黑树转回链表的阈值为什么是 6，而不是 8？是因为如果这个阈值也设置成 8，假如发生碰撞，节点增减刚好在 8 附近，会发生链表和红黑树的不断转换，导致资源浪费。

22. HashMap扩容发生在什么时候呢？

当键值对数量超过阈值，也就是容量 * 负载因子时（默认的负载因子是0.75；初始容量是16）

1 左移 4 位，0000 0001 → 0001 0000，也就是 2 的 4 次方。

代码块

```
1   static final int DEFAULT_INITIAL_CAPACITY = 1 << 4;
```

23. 为什么选择 0.75 作为 HashMap 的默认负载因子呢？

这是一个经验值。如果设置得太低，如 0.5，会浪费空间；如果设置得太高，如 0.9，会增加哈希冲突。

这是对时间成本和空间成本平衡的考虑。

1. 如果设计的比较大，如 0.9：那么就是在空位比较少的时候才扩容，发生哈希冲突的概率就高，查找的时间成本就增加了
2. 如果设计的比较小，如 0.5：那么当一半的时候就扩容，此时发生哈希冲突的概率低，但是就需要更多空间去存储元素，空间成本就增加了

24. HashMap的扩容机制了解吗？

扩容时，HashMap 会创建一个新的数组，其容量是原来的两倍。然后遍历旧哈希表中的元素，将其重新分配到新的哈希表中。

- 如果当前桶中只有一个元素，那么直接通过键的哈希值与数组大小取模锁定新的索引位置：e.hash & (newCap - 1)。
- 如果当前桶是红黑树，那么会调用 split() 方法分裂树节点，以保证树的平衡。
- 如果当前桶是链表，会通过旧键的哈希值与旧的数组大小取模 (e.hash & oldCap) == 0 来作为判断条件，如果条件为真，元素保留在原索引的位置；否则元素移动到原索引 + 旧数组大小的位置

25. JDK 8 对 HashMap 做了哪些优化呢？

①、底层数据结构由数组 + 链表改成了数组 + 链表或红黑树的结构。

如果多个键映射到了同一个哈希值，链表会变得很长，在最坏的情况下，当所有的键都映射到同一个桶中时，性能会退化到 O(n)，而红黑树的时间复杂度是 O(logn)。

②、链表的插入方式由头插法改为了尾插法。头插法在扩容后容易改变原来链表的顺序。

③、扩容的时机由插入时判断改为插入后判断，这样可以避免在每次插入时都进行不必要的扩容检查，因为有可能插入后仍然不需要扩容。

④、哈希扰动算法也进行了优化。JDK 7 是通过多次移位和异或运算来实现的。JDK 8 让 hash 值的高 16 位和低 16 位进行了异或运算，让高位的信息也能参与到低位的计算中，这样可以极大程度上减少哈希碰撞。

26. 你能自己设计实现一个 HashMap 吗？

可以，我先说一下整体的设计思路：

1. 第一步，实现一个 hash 函数，对键的 hashCode 进行扰动
2. 第二步，实现一个拉链法的方法来解决哈希冲突
3. 第三步，扩容后，重新计算哈希值，将元素放到新的数组中

27. HashMap 是线程安全的吗？

HashMap 不是线程安全的，主要有以下几个问题：

①、多线程下扩容会死循环。JDK7 中的 HashMap 使用的是头插法来处理链表，在多线程环境下扩容会出现环形链表，造成死循环。

②、多线程在进行 put 元素的时候，可能会导致元素丢失。因为计算出来的位置可能会被其他线程覆盖掉，比如说一个县城 put 3 的时候，另外一个线程 put 了 7，就把 3 给弄丢了。

③、put 和 get 并发时，可能导致 get 为 null。线程 1 执行 put 时，因为元素个数超出阈值而扩容，线程 2 此时执行 get，就有可能出现这个问题。因为线程 1 执行完 table = newTab 之后，线程 2 中的 table 已经发生了改变，比如说索引 3 的键值对移动到了索引 7 的位置，此时线程 2 去 get 索引 3 的元素就 get 不到了。

28. 怎么解决 HashMap 线程不安全的问题呢？

1. 在早期的 JDK 版本中，可以用 Hashtable 来保证线程安全。Hashtable 在方法上加了 synchronized 关键字。
2. 可以通过 Collections.synchronizedMap 方法返回一个线程安全的 Map，内部是通过 synchronized 对象锁来保证线程安全的，比在方法上直接加 synchronized 关键字更轻量级。
3. 使用并发工具包下的 ConcurrentHashMap，使用了CAS+ synchronized 关键字来保证线程安全。

29. HashMap 内部节点是有序的吗?

无序的，根据 hash 值随机插入。

30. 讲讲 LinkedHashMap 怎么实现有序的?

LinkedHashMap 在 HashMap 的基础上维护了一个双向链表，通过 before 和 after 标识前置节点和后置节点。从而实现插入的顺序或访问顺序。

31. 讲讲 TreeMap 怎么实现有序的?

TreeMap 通过 key 的比较器来决定元素的顺序，如果没有指定比较器，那么 key 必须实现 Comparable 接口。

TreeMap 的底层是红黑树，红黑树是一种自平衡的二叉查找树，每个节点都大于其左子树中的任何节点，小于其右子节点树种的任何节点。

插入或者删除元素时通过旋转和染色来保持树的平衡。

查找的时候从根节点开始，利用二叉查找树的特点，逐步向左子树或者右子树递归查找，直到找到目标元素。

32. TreeMap 和 HashMap 的区别

①、HashMap 是基于数组+链表+红黑树实现的，put 元素的时候会先计算 key 的哈希值，然后通过哈希值计算出元素在数组中的存放下标，然后将元素插入到指定的位置，如果发生哈希冲突，会使用链表来解决，如果链表长度大于 8，会转换为红黑树。

②、TreeMap 是基于红黑树实现的，put 元素的时候会先判断根节点是否为空，如果为空，直接插入到根节点，如果不为空，会通过 key 的比较器来判断元素应该插入到左子树还是右子树。

- 在没有发生哈希冲突的情况下，HashMap 的查找效率是 O(1)。适用于查找操作比较频繁的场景。
- TreeMap 的查找效率是 O(logn)。并且保证了元素的顺序，因此适用于需要大量范围查找或者有序遍历的场景。
- Map 集合常规使用选择 HashMap；Map 集合中需要实现排序选择 TreeMap

33. HashMap、Hashtable 区别?

不同点：

1. HashMap 只能有一个 k 为 null，但可以有多个 v 为 null；HashTable 的 k，v 都不为 null，否则抛出 NullPointerException
2. HashMap 不是线程安全的，HashMap 效率较高；Hashtable 是线程安全的，Hashtable 效率较低（synchronized）
3. HashMap 去掉了 HashTable 的 contains 方法，但是加上了 containsValue（）和 containsKey（）方法；

相同点：

1. 为了成功的在 HashMap 和 Hashtable 中存储和获取对象，用作 key 的对象必须实现 hashCode（）方法和 equals（）方法；
2. HashMap 和 HashTable 的数据元素是无序的；
3. HashMap 和 Hashtable 的底层实现都是数组+链表结构实现；

- HashTable 是遗留类，内部实现很多没优化和冗余。在多线程环境下，有同步的 ConcurrentHashMap 替代，没有必要用 HashTable。

Set

34. 说一下 HashSet 的实现原理?

HashSet 是基于 HashMap 实现的，HashSet 底层使用 HashMap 来保存所有元素，当传入 key 时，HashSet 会根据 key。 hashCode（） 计算出 hash 值，根据 hash 值将 value 保存在 bucket 里。当计算出的 hash 值相同时，我们称之为 hash 冲突，HashMap 的做法是用链表和红黑树存储相同 hash 值的 value。当 hash 冲突的个数比较少时，使用链表否则使用红黑树。

35. 如何实现数组和 List 之间的转换?

1. 数组转 List：
    - 使用 Arrays. asList（array） 进行转换。
    - 使用 StreamAPI，先将数组转为 Stream，然后使用 collect 方法将 Stream 收集喂 List
2. List 转数组：使用 List 自带的 toArray（） 方法。

36. ArrayList 和 Vector 的区别是什么?

- 线程安全：Vector 使用了 Synchronized 来实现线程同步，是线程安全的，而 ArrayList 是非线程安全的。
- 性能：ArrayList 在性能方面要优于 Vector。
- 扩容：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%。

37. Array 和 ArrayList 有何区别？

- Array 可以存储基本数据类型和对象，ArrayList 只能存储对象。
- Array 是指定固定大小的，而 ArrayList 大小是自动扩展的。
- Array 内置方法没有 ArrayList 多，比如 addAll、removeAll 等方法只有 ArrayList 有。

38. 哪些集合类是线程安全的？

Vector、HashTable、Stack 是线程安全的。在 JDK 1.5 之后随着 Java。 util. concurrent 并发包的出现，HashMap 也有了自己对应的线程安全类，比如 HashMap 对应的线程安全类就是 ConcurrentHashMap。

39. 迭代器 Iterator 是什么？

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，遍历集合。Iterator 它可以确保，在当前遍历的集合元素被更改的时候，抛出 ConcurrentModificationException 异常。

40. Iterator 和 ListIterator 有什么区别？

- Iterator 可以遍历 Set 和 List 集合，而 ListIterator 只能遍历 List。
- Iterator 只能单向遍历，而 ListIterator 可以双向遍历（向前/后遍历）。
- 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。

41. 怎么确保一个集合不能被修改？

可以使用 Collections. unmodifiableCollection（Collection c） 方法来创建一个只读集合，这样改变集合的任何操作都会抛出 Java。 lang. UnsupportedOperationException 异常。

42. List 和 Set 的区别?

1. 有序性：List 保证插入顺序排序；Set 存储和取出顺序不一致
2. 唯一性：List 可以重复，Set 元素唯一
3. 获取元素：List 可以根据索引，Set 不能根据索引

43. 集合和数组的区别是什么?

1. 数组长度确地，集合长度可变
2. 数组可以是基本数据类型和引用数据类型，集合只能是引用类型。
3. 数组只能存一种元素，集合可以存不同类型。一般存同一种

44. HashMap 底层为什么要用红黑树呢?

（回答的逻辑：不是讲红黑树的优点，而是为什么不用其他的树？ 如二叉搜索树，AVL）

a. 二叉搜索树容易出现极端情况，如果元素插入是有序的，那么二叉树就会退化成链表，查询效率就会变为 O（n）

b. AVL 树比红黑树的要求更高，每个节点的左右子树的高度最多相差 1，高度的平衡保证了极佳的查找效率，但在进行插入和删除操作时，可能需要频繁地进行旋转来维持树的平衡，维护成本更高。

c. 链表的查找时间复杂度是 O（n），当链表长度较长时，查找性能会下降。红黑树是一种折中的方案，查找、插入、删除的时间复杂度都是 O（log n）。

45. hashtable 允许插入 null 键和 null 值?

不允许。多线程情况下，会产生歧义：到底是本身是 null 还是 key 不存在。插入 null 键或 null 值时，HashTable 会抛出 NullPointerException 异常

46. hashMap? 数据覆盖？死循环?

HashMap 是线程不安全的。主要有三个原因

- 多线程下扩容会死循环：HashMap 发生哈希冲突，是将相同哈希值的键值对通过链表的形式存放起来。jdk7 采用头插法，当发生哈希冲突时，下一个元素会放在上一个元素的前面。当多线程情况下，两个人同时进行相同的操作，导致存放时产生了环形链表；jdk8 已修复，扩容时保持原来链表的顺序。
- 多线程的 put 会导致元素丢失：因为计算出来的位置可能会被其他线程的 put 覆盖。正常情况下发生 hash 冲突，是以链表的形式挂在下面；如果是多线程的情况下，计算出的索引位置相同，就会造成前一个 key 被后一个 key 覆盖，从而导致元素的丢失。
- Put 和 get 并发时，可能导致 get 为 null。线程 1 执行 put 时，因为元素的个数超出阈值而导致出现扩容，线程 2 此时执行 get，就有可能出现这个问题。hashmap 底层扩容机制是 new 一个新的数组（容量为原数组的两倍），然后把元素复制过去；如果此时线程 1 还未把元素复制到新数组，线程 2 发起 get 请求，就会找不到元素返回 null。

十七、String

1. ❤️String 为什么是不可变的?

- String 这个类是 final 修饰的。所以该类不能被继承，子类不能改变 String 的行为，确保了不可变性。
- String 类没有提供任何可以修改其内容的公共方法，像 concat 这些看似修改字符串的操作，实际上都是返回一个新创建的字符串对象，而原始字符串对象保持不变。
- String 内部的字符数组是 private final 修饰的。一旦 String 对象被创建，这个字符数组的引用不能被改变。虽然数组本身是 final 的，但它的内容是可以被改变的。String 类也没有提供任何方法来直接访问或修改这个内部字符数组的内容。（final 修饰变量）

2. String 的内部数组是什么类型的?

String 内部数组是 Char[]类型的，JDK9 引入 Byte[]数组存储字符串数据。char 是两个字节，byte 是一个字节。

1. **内存优化**：通过使用 byte[] ，Java 能够在存储 ASCII 符时节省内存。这对于包含大量 ASCII 字符的字符串非常有效。特别是在某些应用程序中，字符串的内存占用可能会显著减少。
2. **支持多种字符编码**：通过引入 coder 字段，Java 能够支持不同的字符编码，这使得 String 类在处理多种语言和字符集时更加灵活。

3. String 和 StringBuilder、StringBuffer 三者区别?

1. String 被声明为 final class，字符串的值实际上存储在一个 final char[]数组中。是不可变字符串。因为它的不可变性，所以拼接字符串时候会产生很多无用的中间对象，如果频繁的进行这样的操作对性能有所影响。
2. StringBuffer 就是为了解决大量拼接字符串时产生很多中间对象问题而提供的一个类。线程安全。
3. StringBuilder 是 JDK1.5 发布的，它和 StringBuffer 本质上没什么区别，就是去掉了保证线程安全的那部分，减少了开销。

速度：StringBuilder>StringBuffer>String

使用：少量数据->String 单线程大量数据->StringBuilder 多线程大量数据->StringBuffer

4. String s1 = new String（"abc"） 和 String s2 = "abc" 的区别?

- new String（“abc”）：先在堆中创建空间，里面维护了 value 属性，指向常量池的“abc”空间，如果常量池没有“abc”，重新创建；如果有，直接通过 value 指向。最终指向堆中的地址
- 直接赋值：先查看常量池是否有“abc”，有的话直接访问；没有创建并指向。s1 最终指向常量池的空间地址

5. String s = new String（"abc"）创建了几个对象?

两个。常量池创建一个“abc”；堆中创建一个 String 对象

6. String 的 intern 方法有什么作用?

用于在字符串常量池中查找与当前 `String` 对象内容相同的字符串。

- 找到，则返回字符串常量池中该字符串的引用
- 没找到，将当前 `String` 对象的引用添加到字符串常量池中，并返回该引用。

7. String 怎么转成 Integer 的？原理?

- Integer.parseInt(String s)
- Integer.valueOf(String s)

原理：都是调用Integer类的parseInt方法