语言基础

Java篇

基础知识

1.final 有什么用?

用于修饰类、属性和方法；

- 被final修饰的类不可以被继承
- 被final修饰的方法不可以被重写
- 被final修饰的变量不可以被改变，被final修饰不可变的是变量的引用，而不是引用指向的内容，引用指向的内容是可以改变的

2.final finally finalize区别

final可以修饰类、变量、方法，修饰类表示该类不能被继承、修饰方法表示该方法不能被重写、修饰变量表示该变量是一个常量不能被重新赋值。

finally一般作用在try-catch代码块中，在处理异常的时候，通常我们将一定要执行的代码方法finally代码块中，表示不管是否出现异常，该代码块都会执行，一般用来存放一些关闭资源的代码。

finalize是一个方法，属于Object类的一个方法，而Object类是所有类的父类，该方法一般由垃圾回收器来调用，当我们调用System.gc() 方法的时候，由垃圾回收器调用finalize()，回收垃圾，一个对象是否可回收的最后判断。

3.this关键字的用法

this是自身的一个对象，代表对象本身，可以理解为：指向对象本身的一个指针。

this的用法在java中大体可以分为3种：

1.普通的直接引用，this相当于是指向当前对象本身。

2.形参与成员名字重名，用this来区分：

4.super关键字的用法

super可以理解为是指向自己超（父）类对象的一个指针，而这个超类指的是离自己最近的一个父类。

super也有三种用法：

1.普通的直接引用

与this类似，super相当于是指向当前对象的父类的引用，这样就可以用super.xxx来引用父类的成员。

2.子类中的成员变量或方法与父类中的成员变量或方法同名时，用super进行区分

3、引用父类构造函数

- super（参数）：调用父类中的某一个构造函数（应该为构造函数中的第一条语句）。
- this（参数）：调用本类中另一种形式的构造函数（应该为构造函数中的第一条语句）。

this与super的区别

- super： 它引用当前对象的直接父类中的成员（用来访问直接父类中被隐藏的父类中成员数据或函数，基类与派生类中有相同成员定义时如：super.变量名 super.成员函数据名（实参）
- this：它代表当前对象名（在程序中易产生二义性之处，应使用this来指明当前对象；如果函数的形参与类中的成员数据同名，这时需用this来指明成员变量名）
- 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。
- super()和this()均需放在构造方法内第一行。
- 尽管可以用this调用一个构造器，但却不能调用两个。
- this和super不能同时出现在一个构造函数里面，因为this必然会调用其它的构造函数，其它的构造函数必然也会有super语句的存在，所以在同一个构造函数里面有相同的语句，就失去了语句的意义，编译器也不会通过。
- 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。
- 从本质上讲，this是一个指向本对象的指针, 然而super是一个Java关键字。

5.static存在的主要意义

static的主要意义是在于创建独立于具体对象的域变量或者方法。以致于即使没有创建对象，也能使用属性和调用方法！
static关键字还有一个比较关键的作用就是 用来形成静态代码块以优化程序性能。static块可以置于类中的任何地方，类中可以有多个static块。在类初次被加载的时候，会按照static块的顺序来执行每个static块，并且只会执行一次。
为什么说static块可以用来优化程序性能，是因为它的特性:只会在类加载的时候执行一次。因此，很多时候会将一些只需要进行一次的初始化操作都放在static代码块中进行。

6.什么是多态机制？Java语言是如何实现多态的?

所谓多态就是指程序中定义的引用变量所指向的具体类型和通过该引用变量发出的方法调用在编程时并不确定，而是在程序运行期间才确定，即一个引用变量倒底会指向哪个类的实例对象，该引用变量发出的方法调用到底是哪个类中实现的方法，必须在由程序运行期间才能决定。因为在程序运行时才确定具体的类，这样，不用修改源程序代码，就可以让引用变量绑定到各种不同的类实现上，从而导致该引用调用的具体方法随之改变，即不修改程序代码就可以改变程序运行时所绑定的具体代码，让程序可以选择多个运行状态，这就是多态性。
多态分为编译时多态和运行时多态。其中编辑时多态是静态的，主要是指方法的重载，它是根据参数列表的不同来区分不同的函数，通过编辑之后会变成两个不同的函数，在运行时谈不上多态。而运行时多态是动态的，它是通过动态绑定来实现的，也就是我们所说的多态性。
多态的实现
Java实现多态有三个必要条件：继承、重写、向上转型。

- 继承：在多态中必须存在有继承关系的子类和父类。
- 重写：子类对父类中某些方法进行重新定义，在调用这些方法时就会调用子类的方法。
- 向上转型：在多态中需要将子类的引用赋给父类对象，只有这样该引用才能够具备技能调用父类的方法和子类的方法。

只有满足了上述三个条件，我们才能够在同一个继承结构中使用统一的逻辑实现代码处理不同的对象，从而达到执行不同的行为。

对于Java而言，它多态的实现机制遵循一个原则：当超类对象引用变量引用子类对象时，被引用对象的类型而不是引用变量的类型决定了调用谁的成员方法，但是这个被调用的方法必须是在超类中定义过的，也就是说被子类覆盖的方法。

7.抽象类和接口的对比

抽象类是用来捕捉子类的通用特性的。接口是抽象方法的集合。

从设计层面来说，抽象类是对类的抽象，是一种模板设计，接口是行为的抽象，是一种行为的规范。

相同点

- 接口和抽象类都不能实例化
- 都位于继承的顶端，用于被其他实现或继承
- 都包含抽象方法，其子类都必须覆写这些抽象方法

不同点

参数	抽象类	接口
声明	抽象类使用abstract关键字声明	接口使用interface关键字声明
实现	子类使用extends关键字来继承抽象类。如果子类不是抽象类的话，它需要提供抽象类中所有声明的方法的实现	子类使用implements关键字来实现接口。它需要提供接口中所有声明的方法的实现
构造器	抽象类可以有构造器	接口不能有构造器
访问修饰符	抽象类中的方法可以是任意访问修饰符	接口方法默认修饰符是public。并且不允许定义为 private 或者 protected
多继承	一个类最多只能继承一个抽象类	一个类可以实现多个接口
字段声明	抽象类的字段声明可以是任意的	接口的字段默认都是 static 和 final 的

8.普通类和抽象类有哪些区别?

- 普通类不能包含抽象方法，抽象类可以包含抽象方法。
- 抽象类不能直接实例化，普通类可以直接实例化。

9.BIO,NIO,AIO 有什么区别?

简答

- BIO：Block IO 同步阻塞式 IO，就是我们平常使用的传统 IO，它的特点是模式简单使用方便，并发处理能力低。
- NIO：Non IO 同步非阻塞 IO，是传统 IO 的升级，客户端和服务器端通过 Channel（通道）通讯，实现了多路复用。
- AIO：Asynchronous IO 是 NIO 的升级，也叫 NIO2，实现了异步非堵塞 IO ，异步 IO 的操作基于事件和回调机制。

详细回答

BIO (Blocking I/O): 同步阻塞I/O模式，数据的读取写入必须阻塞在一个线程内等待其完成。在活动连接数不是特别高（小于单机1000）的情况下，这种模型是比较不错的，可以让每一个连接专注于自己的 I/O 并且编程模型简单，也不用过多考虑系统的过

载、限流等问题。线程池本身就是一个天然的漏斗，可以缓冲一些系统处理不了的连接或请求。但是，当面对十万甚至百万级连接的时候，传统的 BIO 模型是无能为力的。因此，我们需要一种更高效的 I/O 处理模型来应对更高的并发量。

NIO (New I/O): NIO是一种同步非阻塞的I/O模型，在Java 1.4 中引入了NIO框架，对应 java.nio 包，提供了 Channel , Selector，Buffer等抽象。NIO中的N可以理解为Non-blocking，不单纯是New。它支持面向缓冲的，基于通道的I/O操作方法。NIO提供了与传统BIO模型中的 Socket 和 ServerSocket 相对应的 SocketChannel 和 ServerSocketChannel 两种不同的套接字通道实现,两种通道都支持阻塞和非阻塞两种模式。阻塞模式使用就像传统中的支持一样，比较简单，但是性能和可靠性都不好；非阻塞模式正好与之相反。对于低负载、低并发的应用程序，可以使用同步阻塞I/O来提升开发速率和更好的维护性；对于高负载、高并发的（网络）应用，应使用 NIO 的非阻塞模式来开发

AIO (Asynchronous I/O): AIO 也就是 NIO 2。在 Java 7 中引入了 NIO 的改进版 NIO 2,它是异步非阻塞的IO模型。异步 IO 是基于事件和回调机制实现的，也就是应用操作之后会直接返回，不会堵塞在那里，当后台处理完成，操作系统会通知相应的线程进行后续的操作。AIO 是异步IO的缩写，虽然 NIO 在网络操作中，提供了非阻塞的方法，但是 NIO 的 IO 行为还是同步的。对于 NIO 来说，我们的业务线程是在 IO 操作准备好时，得到通知，接着就由这个线程自行进行 IO 操作，IO操作本身是同步的。查阅网上相关资料，我发现就目前来说 AIO 的应用还不是很广泛，Netty 之前也尝试使用过 AIO，不过又放弃了。

10.String和StringBuffer、StringBuilder的区别是什么？String为什么是不可变的

可变性

String类中使用字符数组保存字符串，private final char value[]，所以string对象是不可变的。StringBuilder与StringBuffer都继承自AbstractStringBuilder类，在AbstractStringBuilder中也是使用字符数组保存字符串，char[] value，这两种对象都是可变的。

线程安全性

String中的对象是不可变的，也就可以理解为常量，线程安全。AbstractStringBuilder是StringBuilder与StringBuffer的公共父类，定义了一些字符串的基本操作，如expandCapacity、append、insert、indexOf等公共方法。StringBuffer对方法加了同步锁或者对调用的方法加了同步锁，所以是线程安全的。StringBuilder并没有对方法进行加同步锁，所以是非线程安全的。

性能

每次对String 类型进行改变的时候，都会生成一个新的String对象，然后将指针指向新的String 对象。StringBuffer每次都会对StringBuffer对象本身进行操作，而不是生成新的对象并改变对象引用。相同情况下使用StirngBuilder 相比使用StringBuffer 仅能获得10%~15% 左右的性能提升，但却要冒多线程不安全的风险。

对于三者使用的总结

如果要操作少量的数据用 = String

单线程操作字符串缓冲区 下操作大量数据 = StringBuilder

多线程操作字符串缓冲区 下操作大量数据 = StringBuffer

11.为什么要使用线程池?

线程池做的工作主要是控制运行的线程的数量，处理过程中将任务放入队列，然后再线程创建后启动这些任务如果线程的数量超过最大数量，超过数量的线程将排队等候，等其他线程执行完毕，再从队列中取出任务来执行

特点：线程复用，控制最大并发数，管理线程

一、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

二、提高响应速度，当任务到达时，任务可以不需要的等到线程创建就能够立刻执行

三、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。

12.线程池七大参数介绍

（1）corePoolSize：线程池中常驻核心线程数

（2）maximumPoolSize：线程池能够容纳同时执行的最大线程数，此值必须大于等于1

（3）keepAliveTime：多余的空闲线程存活时间。当前线程池数量超过corePoolSize时，当空闲时间到达keepAliveTime值时，多余空闲线程会被销毁直到只剩下corePoolSize个线程为止。

（4）unit：keepAliveTime的时间单位

（5）workQueue：任务队列，被提交但尚未执行的任务

（6）threadFactory：表示生成线程池中的工作线程的线程工厂，用于创建线程，一般为默认线程工厂即可

（7）handler：拒绝策略，表示当队列满了并且工作线程大于等于线程池的最大线程数（maximumPoolSize）时如何来拒绝来请求的Runnable的策略

集合

1.ArrayList 的优缺点

ArrayList的优点如下：

- ArrayList 底层以数组实现，是一种随机访问模式。ArrayList 实现了 RandomAccess 接口，因此查找的时候非常快。
- ArrayList 在顺序添加一个元素的时候非常方便。

ArrayList 的缺点如下：

- 删除元素的时候，需要做一次元素复制操作。如果要复制的元素很多，那么就会比较耗费性能。
- 插入元素的时候，也需要做一次元素复制操作，缺点同上。
- ArrayList 比较适合顺序添加、随机访问的场景。

2.ArrayList 和 LinkedList 的区别是什么?

数据结构实现：ArrayList 是动态数组的数据结构实现，而 LinkedList 是双向链表的数据结构实现。

随机访问效率：ArrayList 比 LinkedList 在随机访问的时候效率要高，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。

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。

内存空间占用：LinkedList 比 ArrayList 更占内存，因为 LinkedList 的节点除了存储数据，还存储了两个引用，一个指向前一个元素，一个指向后一个元素。

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

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。

补充：数据结构基础之双向链表

双向链表也叫双链表，是链表的一种，它的每个数据结点中都有两个指针，分别指向直接后继和直接前驱。所以，从双向链表中的任意一个结点开始，都可以很方便地访问它的前驱结点和后继结点。

3.HashSet如何检查重复？HashSet是如何保证数据不可重复的?

向HashSet 中add ()元素时，判断元素是否存在的依据，不仅要比较hash值，同时还要结合equles 方法比较。

HashSet 中的add ()方法会使用HashMap 的put()方法。

HashMap 的 key 是唯一的，由源码可以看出 HashSet 添加进去的值就是作为HashMap 的key，并且在HashMap中如果K/V相同时，会用新的V覆盖掉旧的V，然后返回旧的V。所以不会重复 （ HashMap 比较key是否相等是先比较hashcode 再比较 equals ）。

hashCode（）与equals（）的相关规定：

如果两个对象相等，则hashcode一定也是相同的

两个对象相等,对两个equals方法返回true

两个对象有相同的hashcode值，它们也不一定是相等的

综上，equals方法被覆盖过，则hashCode方法也必须被覆盖

hashCode()的默认行为是对堆上的对象产生独特值。如果没有重写hashCode()，则该class的两个对象无论如何都不会相等（即使这两个对象指向相同的数据）。

==与equals的区别

==是判断两个变量或实例是不是指向同一个内存空间 equals是判断两个变量或实例所指向的内存空间的值是不是相同

==是指对内存地址进行比较 equals()是对字符串的内容进行比较3.==指引用是否相同 equals()指的是值是否相同

4.说一下 HashMap 的实现原理?

HashMap概述： HashMap是基于哈希表的Map接口的非同步实现。此实现提供所有可选的映射操作，并允许使用null值和null键。此类不保证映射的顺序，特别是它不保证该顺序恒久不变。

HashMap的数据结构： 在Java编程语言中，最基本的结构就是两种，一个是数组，另外一个是模拟指针（引用），所有的数据结构都可以用这两个基本结构来构造的，HashMap也不例外。HashMap实际上是一个“链表散列”的数据结构，即数组和链表的结合体。

HashMap 基于 Hash 算法实现的

当我们往Hashmap中put元素时，利用key的hashCode重新hash计算出当前对象的元素在数组中的下标

存储时，如果出现hash值相同的key，此时有两种情况。(1)如果key相同，则覆盖原始值；(2)如果key不同（出现冲突），则将当前的key-value放入链表中

获取时，直接找到hash值对应的下标，在进一步判断key是否相同，从而找到对应值。

理解了以上过程就不难明白HashMap是如何解决hash冲突的问题，核心就是使用了数组的存储方式，然后将冲突的key的对象放入链表中，一旦发现冲突就在链表中做进一步的对比。

需要注意Jdk 1.8中对HashMap的实现做了优化，当链表中的节点数据超过八个之后，该链表会转为红黑树来提高查询效率，从原来的O(n)到O(logn)

5.HashMap在JDK1.7和JDK1.8中有哪些不同？HashMap的底层实现

在Java中，保存数据有两种比较简单的数据结构：数组和链表。数组的特点是：寻址容易，插入和删除困难；链表的特点是：寻址困难，但插入和删除容易；所以我们将数组和链表结合在一起，发挥两者各自的优势，使用一种叫做拉链法的方式可以解决哈希冲突。

JDK1.8之前

JDK1.8之前采用的是拉链法。拉链法：将链表和数组相结合。也就是说创建一个链表数组，数组中每一格就是一个链表。若遇到哈希冲突，则将冲突的值加到链表中即可。

JDK1.8之后

相比于之前的版本，jdk1.8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，将链表转化为红黑树，以减少搜索时间

JDK1.7 VS JDK1.8 比较

JDK1.8主要解决或优化了一下问题：

1. resize 扩容优化
2. 引入了红黑树，目的是避免单条链表过长而影响查询效率，红黑树算法请参考
3. 解决了多线程死循环问题，但仍是非线程安全的，多线程时可能会造成数据丢失问题。

6.HashMap的扩容操作是怎么实现的?

①.在jdk1.8中，resize方法是在hashmap中的键值对大于阀值时或者初始化时，就调用resize方法进行扩容；

②.每次扩展的时候，都是扩展2倍；

③.扩展后Node对象的位置要么在原位置，要么移动到原偏移量两倍的位置。

在putVal()中，我们看到在这个函数里面使用到了2次resize()方法，resize()方法表示的在进行第一次初始化时会对其进行扩容，或者当该数组的实际大小大于其临界值值(第一次为12),这个时候在扩容的同时也会伴随的桶上面的元素进行重新分发，这也是JDK1.8版本的一个优化的地方，在1.7中，扩容之后需要重新去计算其Hash值，根据Hash值对其进行分发，但在1.8版本中，则是根据在同一个桶的位置中进行判断(e.hash & oldCap)是否为0，重新进行hash分配后，该元素的位置要么停留在原始位置，要么移动到原始位置+增加的数组大小这个位置上

7.HashMap 的长度为什么是2的幂次方

为了能让 HashMap 存取高效，尽量较少碰撞，也就是要尽量把数据分配均匀，每个链表/红黑树长度大致相同。这个实现就是把数据存到哪个链表/红黑树中的算法。

这个算法应该如何设计呢?

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

那为什么是两次扰动呢?

答：这样就是加大哈希值低位的随机性，使得分布更均匀，从而提高对应数组存储下标位置的随机性&均匀性，最终减少Hash冲突，两次就够了，已经达到了高位低位同时参与运算的目的；

8.HashMap 与 HashTable 有什么区别

线程安全： HashMap 是非线程安全的，HashTable 是线程安全的；HashTable 内部的方法基本都经过 synchronized 修饰。（如果你要保证线程安全的话就使用 ConcurrentHashMap 吧！）；

效率： 因为线程安全的问题，HashMap 要比 HashTable 效率高一点。另外，HashTable 基本被淘汰，不要在代码中使用它；

对Null key 和Null value的支持： HashMap 中，null 可以作为键，这样的键只有一个，可以有一个或多个键所对应的值为 null。但是在 HashTable 中 put 进的键值只要有一个 null，直接抛NullPointerException。

**初始容量大小和每次扩充容量大小的不同 **： ①创建时如果不指定容量初始值，Hashtable 默认的初始大小为11，之后每次扩充，容量变为原来的2n+1。HashMap 默认的初始化大小为16。之后每次扩充，容量变为原来的2倍。②创建时如果给定了容量初始值，那么 Hashtable 会直接使用你给定的大小，而 HashMap 会将其扩充为2的幂次方大小。也就是说 HashMap 总是使用2的幂作为哈希表的大小，后面会介绍到为什么是2的幂次方。

底层数据结构： JDK1.8 以后的 HashMap 在解决哈希冲突时有了较大的变化，当链表长度大于阈值（默认为8）时，将链表转化为红黑树，以减少搜索时间。Hashtable 没有这样的机制。

推荐使用：在 Hashtable 的类注释可以看到，Hashtable 是保留类不建议使用，推荐在单线程环境下使用 HashMap 替代，如果需要多线程使用则用 ConcurrentHashMap 替代。

9.HashMap 和 ConcurrentHashMap 的区别

ConcurrentHashMap对整个桶数组进行了分割分段(Segment)，然后在每一个分段上都用lock锁进行保护，相对于HashTable的synchronized锁的粒度更精细了一些，并发性能更好，而HashMap没有锁机制，不是线程安全的。（JDK1.8之后ConcurrentHashMap启用了一种全新的方式实现,利用CAS算法。）

HashMap的键值对允许有null，但是ConCurrentHashMap都不允许。

10.ConcurrentHashMap 和 Hashtable 的区别

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

底层数据结构： JDK1.7的 ConcurrentHashMap 底层采用 分段的数组+链表 实现，JDK1.8 采用的数据结构跟HashMap1.8的结构一样，数组+链表/红黑二叉树。Hashtable 和 JDK1.8 之前的 HashMap 的底层数据结构类似都是采用 数组+链表 的形式，数组是 HashMap 的主体，链表则是主要为了解决哈希冲突而存在的；

实现线程安全的方式（重要）： ① 在JDK1.7的时候，ConcurrentHashMap（分段锁） 对整个桶数组进行了分割分段(Segment)，每一把锁只锁容器其中一部分数据，多线程访问容器里不同数据段的数据，就不会存在锁竞争，提高并发访问率。（默认分配16个Segment，比Hashtable效率提高16倍。） 到了 JDK1.8 的时候已经摒弃了Segment的概念，而是直接用 Node

数组+链表+红黑树的数据结构来实现，并发控制使用 synchronized 和 CAS 来操作。（JDK1.6以后 对 synchronized锁做了很多优化） 整个看起来就像是优化过且线程安全的 HashMap，虽然在JDK1.8中还能看到 Segment 的数据结构，但是已经简化了属性，只是为了兼容旧版本；② Hashtable(同一把锁) :使用 synchronized 来保证线程安全，效率非常低下。当一个线程访问同步方法时，其他线程也访问同步方法，可能会进入阻塞或轮询状态，如使用 put 添加元素，另一个线程不能使用 put 添加元素，也不能使用 get，竞争会越来越激烈效率越低。

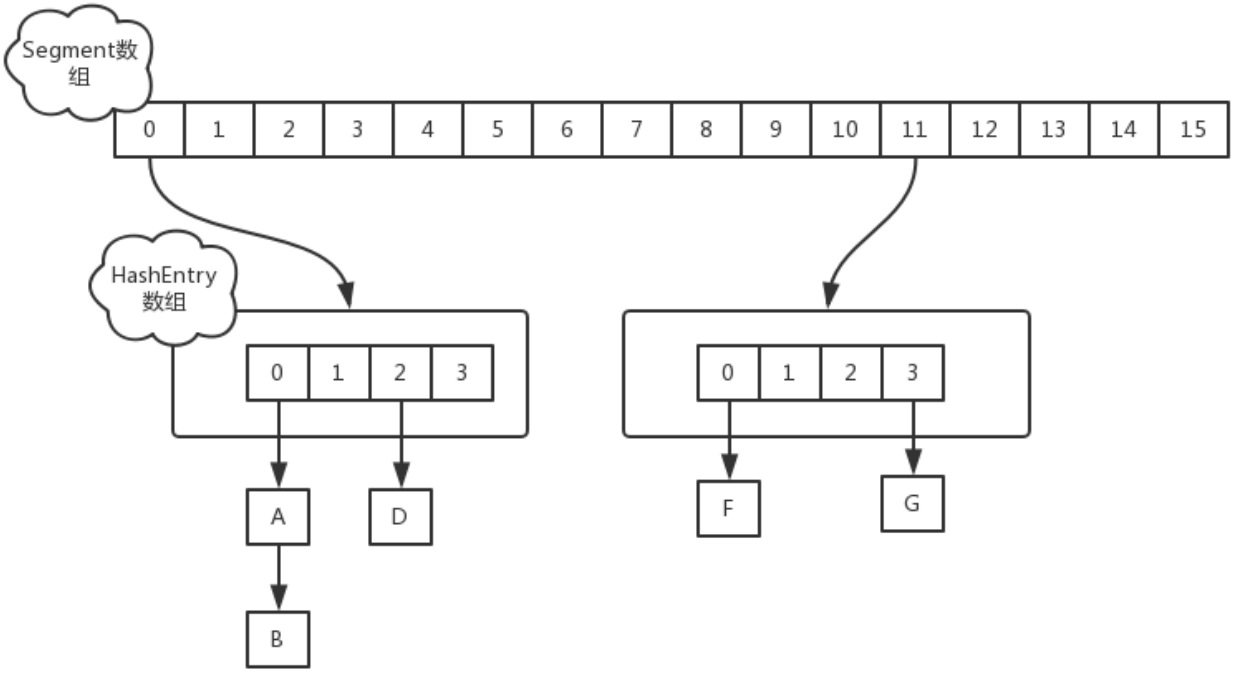
11.ConcurrentHashMap 底层具体实现知道吗？实现原理是什么？

JDK1.7

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

在JDK1.7中，ConcurrentHashMap采用Segment + HashEntry的方式进行实现，结构如下：

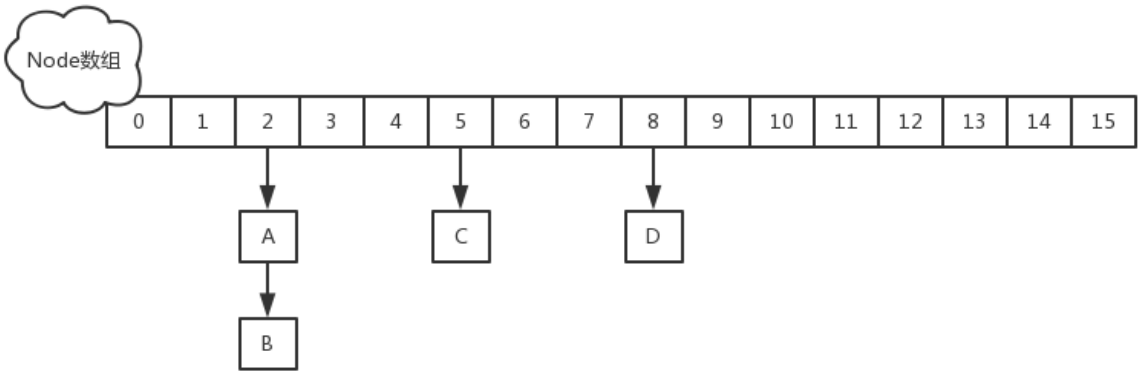
一个 ConcurrentHashMap 里包含一个 Segment 数组。Segment 的结构和HashMap类似，是一种数组和链表结构，一个 Segment 包含一个 HashEntry 数组，每个 HashEntry 是一个链表结构的元素，每个 Segment 守护着一个HashEntry数组里的元素，当对 HashEntry 数组的数据进行修改时，必须首先获得对应的 Segment的锁。



- 该类包含两个静态内部类 HashEntry 和 Segment ；前者用来封装映射表的键值对，后者用来充当锁的角色；
- Segment 是一种可重入的锁 ReentrantLock，每个 Segment 守护一个HashEntry 数组里得元素，当对 HashEntry 数组的数据进行修改时，必须首先获得对应的 Segment 锁。

JDK1.8

在JDK1.8中，放弃了Segment臃肿的设计，取而代之的是采用Node + CAS + Synchronized来保证并发安全进行实现，synchronized只锁定当前链表或红黑二叉树的首节点，这样只要hash不冲突，就不会产生并发，效率又提升N倍。



JVM

1.JVM 的主要组成部分及其作用

JVM包含两个子系统和两个组件，两个子系统为Class loader(类装载)、Execution engine(执行引擎)；两个组件为Runtime data area(运行时数据区)、Native Interface(本地接口)。

- Class loader(类装载)：根据给定的全限定名类名(如：java.lang.Object)来装载class文件到Runtime data area中的method area。
- Execution engine（执行引擎）：执行classes中的指令。
- Native Interface(本地接口)：与native libraries交互，是其它编程语言交互的接口。
- Runtime data area(运行时数据区域)：这就是我们常说的JVM的内存。

作用： 首先通过编译器把 Java 代码转换成字节码，类加载器（ClassLoader）再把字节码加载到内存中，将其放在运行时数据区（Runtime data area）的方法区内，而字节码文件只是 JVM 的一套指令集规范，并不能直接交给底层操作系统去执行，因此需要特定的命令解析器执行引擎（Execution Engine），将字节码翻译成底层系统指令，再交由 CPU 去执行，而这个过程中需要调用其他语言的本地库接口（Native Interface）来实现整个程序的功能。

2.说一下 JVM 运行时数据区

Java 虚拟机在执行 Java 程序的过程中会把它所管理的内存区域划分为若干个不同的数据区域。这些区域都有各自的用途，以及创建和销毁的时间，有些区域随着虚拟机进程的启动而存在，有些区域则是依赖线程的启动和结束而建立和销毁。Java 虚拟机所管理的内存被划分为如下几个区域：

- 程序计数器（Program Counter Register）：当前线程所执行的字节码的行号指示器，字节码解析器的工作是通过改变这个计数器的值，来选取下一条需要执行的字节码指令，分支、循环、跳转、异常处理、线程恢复等基础功能，都需要依赖这个计数器来完成；
- Java 虚拟机栈（Java Virtual Machine Stacks）：用于存储局部变量表、操作数栈、动态链接、方法出口等信息；
- 本地方法栈（Native Method Stack）：与虚拟机栈的作用是一样的，只不过虚拟机栈是服务 Java 方法的，而本地方法栈是为虚拟机调用 Native 方法服务的；
- Java 堆（Java Heap）：Java 虚拟机中内存最大的一块，是被所有线程共享的，几乎所有的对象实例都在这里分配内存；

- 方法区（Methed Area）：用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译后的代码等数据。

3.说一下堆栈的区别?

物理地址

- 堆的物理地址分配对对象是不连续的。因此性能慢些。在GC的时候也要考虑到不连续的分配，所以有各种算法。比如，标记-消除，复制，标记-压缩，分代（即新生代使用复制算法，老年代使用标记——压缩）
- 栈使用的是数据结构中的栈，先进后出的原则，物理地址分配是连续的。所以性能快。

内存区别

- 堆因为是不连续的，所以分配的内存是在运行期确认的，因此大小不固定。一般堆大小远远大于栈。
- 栈是连续的，所以分配的内存大小要在编译期就确认，大小是固定的。

存放的内容

- 堆存放的是对象的实例和数组。因此该区更关注的是数据的存储
- 栈存放：局部变量，操作数栈，返回结果。该区更关注的是程序方法的执行。

PS：

静态变量放在方法区

静态的对象还是放在堆。

程序的可见度

堆对于整个应用程序都是共享、可见的。

栈只对于线程是可见的。所以也是线程私有。他的生命周期和线程相同

4.简述Java垃圾回收机制

在java中，程序员是不需要显示的去释放一个对象的内存的，而是由虚拟机自行执行。在JVM中，有一个垃圾回收线程，它是低优先级的，在正常情况下是不会执行的，只有在虚拟机空闲或者当前堆内存不足时，才会触发执行，扫面那些没有被任何引用的对象，并将它们添加到要回收的集合中，进行回收。

5.GC是什么？为什么要GC

GC 是垃圾收集的意思（Gabage Collection）,内存处理是编程人员容易出现问题的地方，忘记或者错误的内存回收会导致程序或系统的不稳定甚至崩溃，Java 提供的 GC 功能可以自动监测对象是否超过作用域从而达到自动回收内存的目的，Java 语言没有提供释放已分配内存的显示操作方法。

6.垃圾回收的优点和原理。并考虑2种回收机制

java语言最显著的特点就是引入了垃圾回收机制，它使java程序员在编写程序时不再考虑内存管理的问题。由于有这个垃圾回收机制，java中的对象不再有“作用域”的概念，只有引用的对象才有“作用域”。

垃圾回收机制有效的防止了内存泄露，可以有效的使用可使用的内存。

垃圾回收器通常作为一个单独的低级别的线程运行，在不可预知的情况下对内存堆中已经死亡的或很长时间没有用过的对象进行清除和回收。

程序员不能实时的对某个对象或所有对象调用垃圾回收器进行垃圾回收。

垃圾回收有**分代复制垃圾回收、标记垃圾回收、增量垃圾回收**。

7.垃圾回收器的基本原理是什么？垃圾回收器可以马上回收内存吗？有什么办法主动通知虚拟机进行垃圾回收？

对于GC来说，当程序员创建对象时，GC就开始监控这个对象的地址、大小以及使用情况。

通常，GC采用有向图的方式记录和管理堆(heap)中的所有对象。通过这种方式确定哪些对象是"可达的"，哪些对象是"不可达的"。当GC确定一些对象为"不可达"时，GC就有责任回收这些内存空间。

可以。程序员可以手动执行System.gc()，通知GC运行，但是Java语言规范并不保证GC一定会执行。

8.Java 中都有哪些引用类型?

- 强引用：发生 gc 的时候不会被回收。
- 软引用：有用但不是必须的对象，在发生内存溢出之前会被回收。
- 弱引用：有用但不是必须的对象，在下一次GC时会被回收。
- 虚引用（幽灵引用/幻影引用）：无法通过虚引用获得对象，用 PhantomReference 实现虚引用，虚引用的用途是在 gc 时返回一个通知。

9.怎么判断对象是否可以被回收?

垃圾收集器在做垃圾回收的时候，首先需要判定的就是哪些内存是需要被回收的，哪些对象是「存活」的，是不可以被回收的；哪些对象已经「死掉」了，需要被回收。

一般有两种方法来判断：

引用计数器法：为每个对象创建一个引用计数，有对象引用时计数器 +1，引用被释放时计数 -1，当计数器为 0 时就可以被回收。它有一个缺点不能解决循环引用的问题；

可达性分析算法：从 GC Roots 开始向下搜索，搜索所走过的路径称为引用链。当一个对象到 GC Roots 没有任何引用链相连时，则证明此对象是可以被回收的。

10.在Java中，对象什么时候可以被垃圾回收

当对象对当前使用这个对象的应用程序变得不可触及的时候，这个对象就可以被回收了。

垃圾回收不会发生在永久代，如果永久代满了或者是超过了临界值，会触发完全垃圾回收(Full GC)。如果你仔细查看垃圾收集器的输出信息，就会发现永久代也是被回收的。这就是为什么正确的永久代大小对避免Full GC是非常重要的原因。

11.JVM中的永久代中会发生垃圾回收吗

垃圾回收不会发生在永久代，如果永久代满了或者是超过了临界值，会触发完全垃圾回收(Full GC)。如果你仔细查看垃圾收集器的输出信息，就会发现永久代也是被回收的。这就是为什么正确的永久代大小对避免Full GC是非常重要的原因。请参考下Java8：从永久代到元数据区

(Java8中已经移除了永久代，新加了一个叫做元数据区的native内存区)

12.说一下 JVM 有哪些垃圾回收算法?

标记-清除算法：标记无用对象，然后进行清除回收。缺点：效率不高，无法清除垃圾碎片。

复制算法：按照容量划分二个大小相等的内存区域，当一块用完的时候将活着的对象复制到另一块上，然后再把已使用的内存空间一次清理掉。缺点：内存使用率不高，只有原来的一半。

标记-整理算法：标记无用对象，让所有存活的对象都向一端移动，然后直接清除掉端边界以外的内存。

分代算法：根据对象存活周期的不同将内存划分为几块，一般是新生代和老年代，新生代基本采用复制算法，老年代采用标记整理算法。

详细来说：

标记-清除算法:

标记无用对象，然后进行清除回收。

标记-清除算法（Mark-Sweep）是一种常见的基础垃圾收集算法，它将垃圾收集分为两个阶段：

- 标记阶段：标记出可以回收的对象。
- 清除阶段：回收被标记的对象所占用的空间。

标记-清除算法之所以是基础的，是因为后面讲到的垃圾收集算法都是在此算法的基础上进行改进的。

优点：实现简单，不需要对象进行移动。

缺点：标记、清除过程效率低，产生大量不连续的内存碎片，提高了垃圾回收的频率。

复制算法

为了解决标记-清除算法的效率不高的问题，产生了复制算法。它把内存空间划为两个相等的区域，每次只使用其中一个区域。垃圾收集时，遍历当前使用的区域，把存活对象复制到另外一个区域中，最后将当前使用的区域的可回收的对象进行回收。

优点：按顺序分配内存即可，实现简单、运行高效，不用考虑内存碎片。

缺点：可用的内存大小缩小为原来的一半，对象存活率高时会频繁进行复制。

标记-整理算法

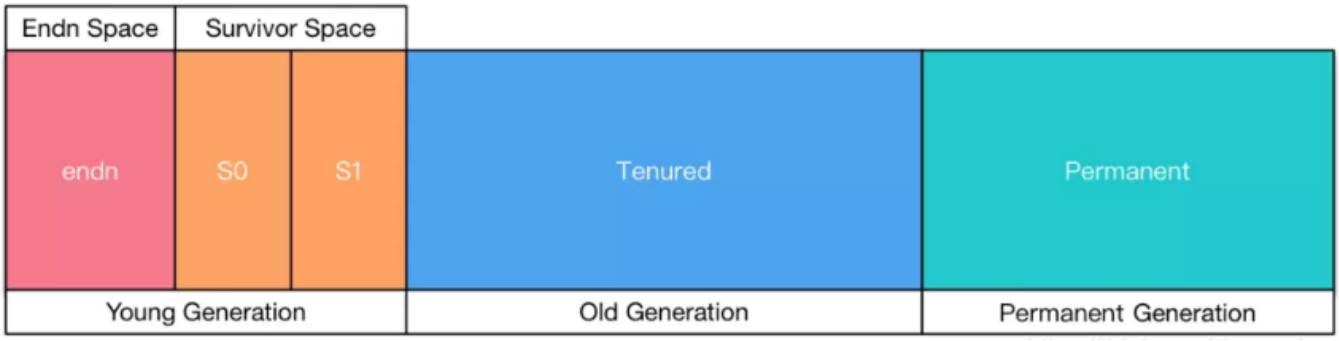
在新生代中可以使用复制算法，但是在老年代就不能选择复制算法了，因为老年代的对象存活率会较高，这样会有较多的复制操作，导致效率变低。标记-清除算法可以应用在老年代中，但是它效率不高，在内存回收后容易产生大量内存碎片。因此就出现了一种标记-整理算法（Mark-Compact）算法，与标记-整理算法不同的是，在标记可回收的对象后将所有存活的对象压缩到内存的一端，使他们紧凑的排列在一起，然后对端边界以外的内存进行回收。回收后，已用和未用的内存都各自一边。

优点：解决了标记-清理算法存在的内存碎片问题。

缺点：仍需要进行局部对象移动，一定程度上降低了效率。

分代收集算法

当前商业虚拟机都采用**分代收集**的垃圾收集算法。分代收集算法，顾名思义是根据对象的**存活周期**将内存划分为几块。一般包括**年轻代**、**老年代** 和 **永久代**，如图所示：



13.说一下 JVM 有哪些垃圾回收器?

如果说垃圾收集算法是内存回收的方法论，那么垃圾收集器就是内存回收的具体实现。下图展示了7种作用于不同分代的收集器，其中用于回收新生代的收集器包括Serial、PraNew、Parallel Scavenge，回收老年代的收集器包括Serial Old、Parallel Old、CMS，还有用于回收整个Java堆的G1收集器。不同收集器之间的连线表示它们可以搭配使用。

- Serial收集器（复制算法): 新生代单线程收集器，标记和清理都是单线程，优点是简单高效；
- ParNew收集器 (复制算法): 新生代收并行集器，实际上是Serial收集器的多线程版本，在多核CPU环境下有着比Serial更好的表现；
- Parallel Scavenge收集器 (复制算法): 新生代并行收集器，追求高吞吐量，高效利用 CPU。吞吐量 = 用户线程时间/(用户线程时间+GC线程时间)，高吞吐量可以高效率的利用CPU时间，尽快完成程序的运算任务，适合后台应用等对交互相应要求不高的场景；
- Serial Old收集器 (标记–整理算法): 老年代单线程收集器，Serial收集器的老年代版本；
- Parallel Old收集器 (标记–整理算法)： 老年代并行收集器，吞吐量优先，Parallel Scavenge收集器的老年代版本；
- CMS(Concurrent Mark Sweep)收集器（标记–清除算法）： 老年代并行收集器，以获取最短回收停顿时间为目标的收集器，具有高并发、低停顿的特点，追求最短GC回收停顿时间。
- G1(Garbage First)收集器 (标记–整理算法)： Java堆并行收集器，G1收集器是JDK1.7提供的一个新收集器，G1收集器基于“标记–整理”算法实现，也就是说不会产生内存碎片。此外，G1收集器不同于之前的收集器的一个重要特点是：G1回收的范围是整个Java堆(包括新生代，老年代)，而前六种收集器回收的范围仅限于新生代或老年代。

14.详细介绍一下 CMS 垃圾回收器和G1垃圾收集器

CMS垃圾回收器

CMS与G1都是并发回收，多线程分阶段回收，只有某阶段会stw；2.CMS只会回收老年代和永久代（1.8开始为元数据区，需要设置CMSClassUnloadingEnabled），不会收集年轻代；年轻带只能配合Parallel New或Serial回收器； CMS是一种预处理垃圾回收器，它不能等到old内存用尽时回收，需要在内存用尽前，完成回收操作，否则会导致并发回收失败；所以CMS垃圾回收器开始执行回收操作，有一个触发阈值，默认是老年代或永久带达到92%；

CMS 处理过程有七个步骤（这也是面试常问的问题，包括哪俩个阶段会STW）：

1. 初始标记(CMS-initial-mark) ,会导致STW； 初始标记阶段就是标记老年代中的GC ROOT对象和与GC ROOT对象关联的对象给标记出来。

2. 并发标记(CMS-concurrent-mark)，与用户线程同时运行；因 为是并发运行的，在运行期间会发生新生代的对象晋升到老年代、或者是直接在老年代分配对象、或者更新老年代对象的引用关系等等，对于这些对象，都是需要进行重新标记的，否则有些对象就会被遗漏，发生漏标的情况。为了提高重新标记的效率，该阶段会把上述对象所在的Card标识为Dirty，后续只需扫描这些Dirty Card的对象，避免扫描整个老年代；

并发标记阶段只负责将引用发生改变的Card标记为Dirty状态，不负责处理；

3. 重新标记(CMS-remark) ，会导致STW； 这个阶段会导致第二次stop the word，该阶段的任务是完成标记整个年老代的所有的存活对象。

这个阶段，重新标记的内存范围是整个堆，包含_young_gen和_old_gen。为什么要扫描新生代呢，因为对于老年代中的对象，如果被新生代中的对象引用，那么就会被视为存活对象，即使新生代的对象已经不可达了，也会使用这些不可达的对象当做cms的“gc root”，来扫描老年代； 因此对于老年代来说，引用了老年代中对象的新生代的对象，也会被老年代视作“GC ROOTS”:当此阶段耗时较长的时候，可以加入参数-XX:+CMSScavengeBeforeRemark，在重新标记之前，先执行一次ygc，回收掉年轻带的对象无用的对象，并将对象放入幸存带或晋升到老年代，这样再进行年轻带扫描时，只需要扫描幸存区的对象即可，一般幸存带非常小，这大大减少了扫描时间

由于之前的预处理阶段是与用户线程并发执行的，这时候可能年轻带的对象对老年代的引用已经发生了很多改变，这个时候，remark阶段要花很多时间处理这些改变，会导致很长stop the word，所以通常CMS尽量运行Final Remark阶段在年轻代是足够干净的时候。另外，还可以开启并行收集：-XX:+CMSParallelRemarkEnabled。来提高这个阶段的效率

4. 并发清除(CMS-concurrent-sweep)，与用户线程同时运行；

通过以上4个阶段的标记，老年代所有存活的对象已经被标记并且现在要通过Garbage Collector采用清扫的方式回收那些不能用的对象了。

这个阶段主要是清除那些没有标记的对象并且回收空间；由于CMS并发清理阶段用户线程还在运行着，伴随程序运行自然就还会有新的垃圾不断产生，这一部分垃圾出现在标记过程之后，CMS无法在当次收集中处理掉它们，只好留待下一次GC时再清理掉。这一部分垃圾就称为“浮动垃圾”

CMS垃圾回收器的优化：

1.减少remark阶段停顿一般CMS的GC耗时 80%都在remark阶段，如果发现remark阶段停顿时间很长，可以尝试添加该参数：-XX:+CMSScavengeBeforeRemark

在执行remark操作之前先做一次ygc，目的在于减少ygen对oldgen的无效引用，降低remark时的开销。

2.内存碎片

CMS是基于标记-清除算法的，只会将标记为为存活的对象删除，并不会移动对象整理内存空间，会造成内存碎片，这时候我们需要用到这个参数;-XX:CMSFullGCsBeforeCompaction=n

3.promotion failed与解决方法

过早提升与提升失败

在 Minor GC 过程中，Survivor Unused 可能不足以容纳 Eden 和另一个 Survivor 中的存活对象， 那么多余的将被移到老年 代， 称为过早提升（Premature Promotion）,这会导致老年代中短期存活对象的增长， 可能会引发严重的性能问题。 再进一 步， 如果老年代满了， Minor GC 后会进行 Full GC， 这将导致遍历整个堆， 称为提升失败（Promotion Failure）。

早提升的原因

1. Survivor空间太小，容纳不下全部的运行时短生命周期的对象，如果是这个原因，可以尝试将Survivor调大，否则端生命周 期的对象提升过快，导致老年代很快就被占满，从而引起频繁的full gc；

2. 对象太大，Survivor和Eden没有足够大的空间来存放这些大象；

提升失败原因

当提升的时候，发现老年代也没有足够的连续空间来容纳该对象。为什么是没有足够的连续空间而不是空闲空间呢?

老年代容纳不下提升的对象有两种情况：

1. 老年代空闲空间不够用了；

2. 老年代虽然空闲空间很多，但是碎片太多，没有连续的空闲空间存放该对象；

解决方法

1. 如果是因为内存碎片导致的大对象提升失败，cms需要进行空间整理压缩；

2. 如果是因为提升过快导致的，说明Survivor 空闲空间不足，那么可以尝试调大 Survivor；

3. 如果是因为老年代空间不够导致的，尝试将CMS触发的阈值调低；

4.增加线程数

CMS默认启动的回收线程数目是 (ParallelGCThreads + 3)/4) ，这里的ParallelGCThreads是年轻代的并行收集线程数，感觉 有 点怪怪的；

年轻代的并行收集线程数默认是(ncpus <= 8) ? ncpus : 3 + ((ncpus * 5) / 8)，可以通过-XX:ParallelGCThreads= N 来调整；

如果要直接设定CMS回收线程数，可以通过-XX:ParallelCMSThreads=n，注意这个n不能超过cpu线程数，需要注意的是增加 gc线程数，就会和应用争抢资源；

G1垃圾回收器：

G1垃圾回收器相关数据结构

1.在HotSpot的实现中，整个堆被划分成2048左右个Region。每个Region的大小在1-32MB之间，具体多大取决于堆的大小。

2.对于Region来说，它会有一个分代的类型，并且是唯一一个。即，每一个Region，它要么是young的，要么是old的。还有一类十分特殊的Humongous。所谓的Humongous，就是一个对象的大小超过了某一个阈值——HotSpot中是Region的1/2，那么它会被标记为Humongous。

每一个分配的Region，都可以分成两个部分，已分配的和未被分配的。它们之间的界限被称为top。总体上来说，把一个对象分配到Region内，只需要简单增加top的值。这个做法实际上就是bump-the-pointer。

即每一次回收都是回收N个Region。这个N是多少，主要受到G1回收的效率和用户设置的软实时目标有关。每一次的回收，G1会选择可能回收最多垃圾的Region进行回收。与此同时，G1回收器会维护一个空间Region的链表。每次回收之后的Region都会被加入到这个链表中。

每一次都只有一个Region处于被分配的状态中，被称为current region。在多线程的情况下，这会带来并发的问题。G1回收器采用和CMS一样的TLABs的手段。即为每一个线程分配一个Buffer，线程分配内存就在这个Buffer内分配。但是当线程耗尽了自己的Buffer之后，需要申请新的Buffer。这个时候依然会带来并发的问题。G1回收器采用的是CAS（Compate And Swap）操作。

3.卡片 Card

在每个分区内部又被分成了若干个大小为512 Byte卡片(Card)，标识堆内存最小可用粒度所有分区的卡片将会记录在全局卡片表(Global Card Table)中，分配的对象会占用物理上连续的若干个卡片，当查找对分区内对象的引用时便可通过记录卡片来查找该引用对象(见RSet)。每次对内存的回收，都是对指定分区的卡片进行处理。

4.RS(Remember Set)

RS(Remember Set)是一种抽象概念，用于记录从非收集部分指向收集部分的指针的集合。

在传统的分代垃圾回收算法里面，RS(Remember Set)被用来记录分代之间的指针。在G1回收器里面，RS被用来记录从其他Region指向一个Region的指针情况。因此，一个Region就会有一个RS。这种记录可以带来一个极大的好处：在回收一个Region的时候不需要执行全堆扫描，只需要检查它的RS就可以找到外部引用，而这些引用就是initial mark的根之一。

那么，如果一个线程修改了Region内部的引用，就必须要去通知RS，更改其中的记录。为了达到这种目的，G1回收器引入了一种新的结构，CT(Card Table)——卡表。每一个Region，又被分成了固定大小的若干张卡(Card)。每一张卡，都用一个Byte来记录是否修改过。卡表即这些byte的集合。实际上，如果把RS理解成一个概念模型，那么CT就可以说是RS的一种实现方式。

在RS的修改上也会遇到并发的问题。因为一个Region可能有多个线程在并发修改，因此它们也会并发修改RS。为了避免这样一种冲突，G1垃圾回收器进一步把RS划分成了多个哈希表。每一个线程都在各自的哈希表里面修改。最终，从逻辑上来说，RS就是这些哈希表的集合。哈希表是实现RS的一种通常的方式之一。它有一个极大的好处就是能够去除重复。这意味着，RS的大小将和修改的指针数量相当。而在不去重的情况下，RS的数量和写操作的数量相当。

G1垃圾回收器执行步骤：

1、初始标记；

初始标记阶段仅仅只是标记一下GC Roots能直接关联到的对象，并且修改TAMS的值，让下一个阶段用户程序并发运行时，能在正确可用的Region中创建新对象，这一阶段需要停顿线程，但是耗时很短，

2、并发标记；

并发标记阶段是从GC Root开始对堆中对象进行可达性分析，找出存活的对象，这阶段时耗时较长，但可与用户程序并发执行。

3、最终标记；

最终标记阶段则是为了修正在并发标记期间因用户程序继续运作而导致标记产生变动的那一部分标记记录，虚拟机将这段时间对象变化记录在线程Remenbered Set Logs里面，最终标记阶段需要把Remembered Set Logs的数据合并到

Remembered Set Logs里面，最终标记阶段需要把Remembered Set Logs的数据合并到Remembered Set中，这一阶段需要停顿线程，但是可并行执行。

4、筛选回收：最后在筛选回收阶段首先对各个Region的回收价值和成本进行排序，根据用户所期望的GC停顿时间来制定回收计划。

15.新生代垃圾回收器和老年代垃圾回收器都有哪些？有什么区别?

新生代回收器：Serial、ParNew、Parallel Scavenge

老年代回收器：Serial Old、Parallel Old、CMS

整堆回收器：G1

新生代垃圾回收器一般采用的是复制算法，复制算法的优点是效率高，缺点是内存利用率低；老年代回收器一般采用的是标记–整理的算法进行垃圾回收。

16.简述分代垃圾回收器是怎么工作的?

分代回收器有两个分区：老生代和新生代，新生代默认的空间占比总空间的 1/3，老生代的默认占比是 2/3。

新生代使用的是复制算法，新生代里有 3 个分区：Eden、To Survivor、From Survivor，它们的默认占比是 8:1:1，它的执行流程如下：

- 把 Eden + From Survivor 存活的对象放入 To Survivor 区；
- 清空 Eden 和 From Survivor 分区；
- From Survivor 和 To Survivor 分区交换，From Survivor 变 To Survivor，To Survivor 变 From Survivor。

每次在 From Survivor 到 To Survivor 移动时都存活的对象，年龄就 +1，当年龄到达 15（默认配置是 15）时，升级为老生代。大对象也会直接进入老生代。

老生代当空间占用到达某个值之后就会触发全局垃圾收回，一般使用标记整理的执行算法。以上这些循环往复就构成了整个分代垃圾回收的整体执行流程。

17.简述java内存分配与回收策率以及Minor GC和Major GC

所谓自动内存管理，最终要解决的也就是内存分配和内存回收两个问题。前面我们介绍了内存回收，这里我们再来聊聊内存分配。

对象的内存分配通常是在 Java 堆上分配（随着虚拟机优化技术的诞生，某些场景下也会在栈上分配，后面会详细介绍），对象主要分配在新生代的 Eden 区，如果启动了本地线程缓冲，将按照线程优先在 TLAB 上分配。少数情况下也会直接在老年代上分配。总的来说分配规则不是百分百固定的，其细节取决于哪一种垃圾收集器组合以及虚拟机相关参数有关，但是虚拟机对于内存的分配还是会遵循以下几种「普世」规则：

- **对象优先在 Eden 区分配**

  多数情况，对象都在新生代 Eden 区分配。当 Eden 区分配没有足够的空间进行分配时，虚拟机将会发起一次 Minor GC。如果本次 GC 后还是没有足够的空间，则将启用分配担保机制在老年代中分配内存。

这里我们提到 Minor GC，如果你仔细观察过 GC 日常，通常我们还能从日志中发现 Major GC/Full GC。

- Minor GC 是指发生在新生代的 GC，因为 Java 对象大多都是朝生夕死，所有 Minor GC 非常频繁，一般回收速度也非常快；
- Major GC/Full GC 是指发生在老年代的 GC，出现了 Major GC 通常会伴随至少一次 Minor GC。Major GC 的速度通常会比 Minor GC 慢 10 倍以上。

- **大对象直接进入老年代**

  所谓大对象是指需要大量连续内存空间的对象，频繁出现大对象是致命的，会导致在内存还有不少空间的情况下提前触发 GC 以获取足够的连续空间来安置新对象。

  前面我们介绍过新生代使用的是标记–清除算法来处理垃圾回收的，如果大对象直接在新生代分配就会导致 Eden 区和两个 Survivor 区之间发生大量的内存复制。因此对于大对象都会直接在老年代进行分配。

- 长期存活对象将进入老年代

  虚拟机采用分代收集的思想来管理内存，那么内存回收时就必须判断哪些对象应该放在新生代，哪些对象应该放在老年代。因此虚拟机给每个对象定义了一个对象年龄的计数器，如果对象在 Eden 区出生，并且能够被 Survivor 容纳，将被移动到 Survivor 空间中，这时设置对象年龄为 1。对象在 Survivor 区中每「熬过」一次 Minor GC 年龄就加 1，当年龄达到一定程度（默认 15） 就会被晋升到老年代。

18.简述java类加载机制?

虚拟机把描述类的数据从Class文件加载到内存，并对数据进行校验，解析和初始化，最终形成可以被虚拟机直接使用的java类型。

19.描述一下JVM加载Class文件的原理机制

Java中的所有类，都需要由类加载器装载到JVM中才能运行。类加载器本身也是一个类，而它的工作就是把class文件从硬盘读取到内存中。在写程序的时候，我们几乎不需要关心类的加载，因为这些都是隐式装载的，除非我们有特殊的用法，像是反射，就需要显式的加载所需要的类。

类装载方式，有两种 ：

1.隐式装载， 程序在运行过程中当碰到通过new 等方式生成对象时，隐式调用类装载器加载对应的类到jvm中，

2.显式装载， 通过class.forname()等方法，显式加载需要的类

Java类的加载是动态的，它并不会一次性将所有类全部加载后再运行，而是保证程序运行的基础类(像是基类)完全加载到jvm中，至于其他类，则在需要的时候才加载。这当然就是为了节省内存开销。

20.什么是类加载器，类加载器有哪些?

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。

主要有一下四种类加载器:

- 启动类加载器(Bootstrap ClassLoader)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）：它根据 Java 应用的类路径（CLASSPATH）来加载 Java 类。一般来说，Java 应用的类都是由它来完成加载的。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。

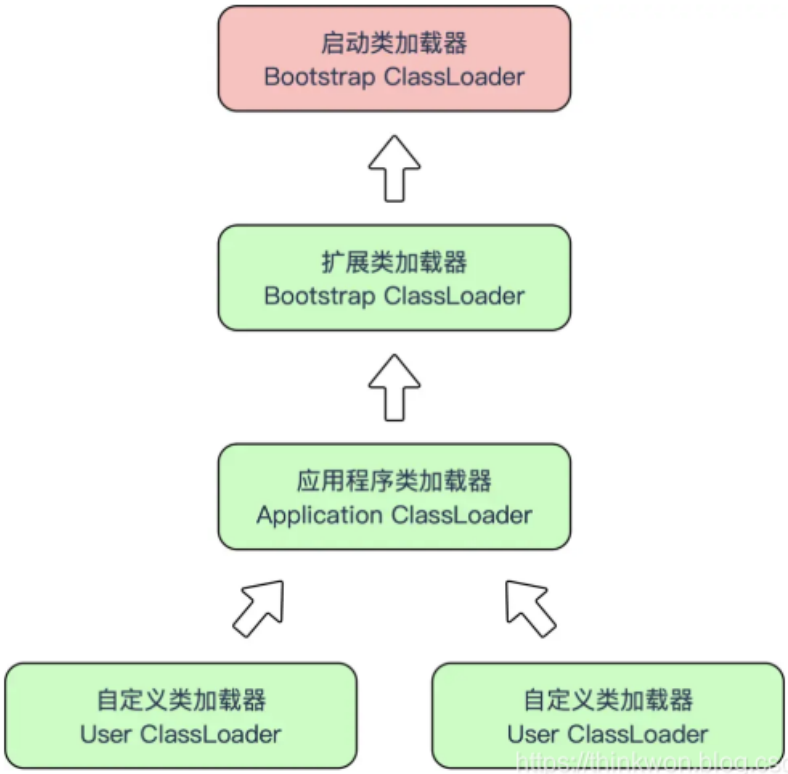
21.说一下类装载的执行过程?

类装载分为以下 5 个步骤：

- 加载：根据查找路径找到相应的 class 文件然后导入；
- 验证：检查加载的 class 文件的正确性；
- 准备：给类中的静态变量分配内存空间；
- 解析：虚拟机将常量池中的符号引用替换成直接引用的过程。符号引用就理解为一个标示，而在直接引用直接指向内存中的地址；
- 初始化：对静态变量和静态代码块执行初始化工作。

22.什么是双亲委派模型?

在介绍双亲委派模型之前先说下类加载器。对于任意一个类，都需要由加载它的类加载器和这个类本身一同确立在 JVM 中的唯一性，每一个类加载器，都有一个独立的类名称空间。类加载器就是根据指定全限定名称将 class 文件加载到 JVM 内存，然后再转化为 class 对象。



双亲委派模型：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，每一层的类加载器都是如此，这样所有的加载请求都会被传送到顶层的启动类加载器中，只有当父加载无法完成加载请求（它的搜索范围中没找到所需的类）时，子加载器才会尝试去加载类。

当一个类收到了类加载请求时，不会自己先去加载这个类，而是将其委派给父类，由父类去加载，如果此时父类不能加载，反馈给子类，由子类去完成类的加载。

23.说一下 JVM 调优的工具?

JDK 自带了很多监控工具，都位于 JDK 的 bin 目录下，其中最常用的是 jconsole 和 jvisualvm 这两款视图监控工具。

jconsole：用于对 JVM 中的内存、线程和类等进行监控；

jvisualvm：JDK 自带的全能分析工具，可以分析：内存快照、线程快照、程序死锁、监控内存的变化、gc 变化等。

常用的 JVM 调优的参数

-Xms2g：初始化推大小为 2g；

-Xmx2g：堆最大内存为 2g；

-XX:NewRatio=4：设置年轻的和老年代的内存比例为 1:4；

-XX:SurvivorRatio=8：设置新生代 Eden 和 Survivor 比例为 8:2；

—XX:+UseParNewGC：指定使用 ParNew + Serial Old 垃圾回收器组合；

-XX:+UseParallelOldGC：指定使用 ParNew + ParNew Old 垃圾回收器组合；

-XX:+UseConcMarkSweepGC：指定使用 CMS + Serial Old 垃圾回收器组合；

-XX:+PrintGC：开启打印 gc 信息；

-XX:+PrintGCDetails：打印 gc 详细信息

Scala

1.scala语言有什么特点？什么是函数式编程？有什么优点?

- scala语言集成面向对象和函数式编程
- 函数式编程是一种典范，将电脑的运算视作是函数的运算。
- 与过程化编程相比，函数式编程里的函数计算可以随时调用。
- 函数式编程中，函数是一等功明。

2.scala中的闭包

定义：你可以在任何作用域内定义函数:包，类甚至是另一个函数或方法。在函数体内，可以访问到相应作用域内地任何变量。(重点)函数可以在变量不再处于作用域内时被调用。

3.var，val和def三个关键字之间的区别?

var是变量声明关键字，类似于Java中的变量，变量值可以更改，但是变量类型不能更改。

val常量声明关键字。

def 关键字用于创建方法（注意方法和函数的区别）

还有一个lazy val（惰性val）声明，意思是当需要计算时才使用，避免重复计算

4. trait（特质）和abstract class（抽象类）的区别?

（1）一个类只能集成一个抽象类，但是可以通过with关键字继承多个特质；

（2）抽象类有带参数的构造函数，特质不行（如 trait t（i：Int）{}，这种声明是错误的）

5.object和class的区别

object是类的单例对象，开发人员无需用new关键字实例化。如果对象的名称和类名相同，这个对象就是伴生对象

6.case class （样本类）是什么

样本类是一种不可变且可分解类的语法糖，这个语法糖的意思大概是在构建时，自动实现一些功能。样本类具有以下特性：

（1）自动添加与类名一致的构造函数（这个就是前面提到的伴生对象，通过apply方法实现），即构造对象时，不需要new；

（2）样本类中的参数默认添加val关键字，即参数不能修改；

（3）默认实现了toString，equals，hashcode，copy等方法；

（4）样本类可以通过==比较两个对象，并且不在构造方法中定义的属性不会用在比较上。

7.unapply 和apply方法的区别， 以及各自使用场景

先讲一个概念——提取器，它实现了构造器相反的效果，构造器从给定的参数创建一个对象，然而提取器却从对象中提取出构造该对象的参数，scala标准库预定义了一些提取器，如上面提到的样本类中，会自动创建一个伴生对象（包含apply和unapply方法）。

为了成为一个提取器，unapply方法需要被伴生对象。

apply方法是为了自动实现样本类的对象，无需new关键字。

8.伴生对象是什么

前面已经提到过，伴生对象就是与类名相同的对象，伴生对象可以访问类中的私有量，类也可以访问伴生对象中的私有方法，类似于Java类中的静态方法。伴生对象必须和其对应的类定义在相同的源文件。

9.Scala类型系统中Nil, Null, None, Nothing四个类型的区别

Null是一个trait（特质），是所以引用类型AnyRef的一个子类型，null是Null唯一的实例。

Nothing也是一个trait（特质），是所有类型Any（包括值类型和引用类型）的子类型，它不在有子类型，它也没有实例，实际上为了一个方法抛出异常，通常会设置一个默认返回类型。

Nil代表一个List空类型，等同List[Nothing]

None是Option monad的空标识

10. Unit类型是什么

Unit代表没有任何意义的值类型，类似于java中的void类型，他是anyval的子类型，仅有一个实例对象"( )"

11.call-by-value和call-by-name求值策略的区别

（1）call-by-value是在调用函数之前计算；

（2） call-by-name是在需要时计算

12.Option类型的定义和使用场景

在Java中，null是一个关键字，不是一个对象，当开发者希望返回一个空对象时，却返回了一个关键字，为了解决这个问题，Scala建议开发者返回值是空值时，使用Option类型，在Scala中null是Null的唯一对象，会引起异常，Option则可以避免。

Option有两个子类型，Some和None（空值）

13.yield如何工作

yield用于循环迭代中生成新值，yield是comprehensions的一部分，是多个操作（foreach, map, flatMap, filter or withFilter）的composition语法糖。

14.解释隐示参数的优先权

在Scala中implicit的功能很强大。当编译器寻找implicits时，如果不注意隐式参数的优先权，可能会引起意外的错误。因此编译器会按顺序查找隐式关键字。顺序如下：

（1）当前类声明的implicits ；

（2）导入包中的 implicits；

（3）外部域（声明在外部域的implicts）；

（4）inheritance

（5）package object

（6）implicit scope like companion objects

15.comprehension（推导式）的语法糖是什么操作?

comprehension（推导式）是若干个操作组成的替代语法。如果不用yield关键字，comprehension（推导式）可以被forech操作替代，或者被map/flatMap，filter代替

16.什么是vaule class?

开发时经常遇到这个的问题，当你使用integer时，希望它代表一些东西，而不是全部东西，例如，一个integer代表年龄，另一个代表高度。由于上述原因，我们考虑包裹原始类型生成一个新的有意义的类型（如年龄类型和高度类型）。

Value classes 允许开发者安全的增加一个新类型，避免运行时对象分配。有一些 必须进行分配的情况 and 限制,但是基本的思想是：在编译时，通过使用原始类型替换值类实例，删除对象分配。

17.Option ，Try 和 Either 三者的区别?

这三种monads允许我们显示函数没有按预期执行的计算结果。

Option表示可选值，它的返回类型是Some（代表返回有效数据）或None（代表返回空值）。

Try类似于Java中的try/catch，如果计算成功，返回Success的实例，如果抛出异常，返回Failure。

Either可以提供一些计算失败的信息，Either有两种可能返回类型：预期/正确/成功的 和 错误的信息。

18.什么是函数柯里化?

柯里化技术是一个接受多个参数的函数转化为接受其中几个参数的函数。经常被用来处理高阶函数。

19.什么是尾递归?

正常递归，每一次递归步骤，需要保存信息到堆栈里面，当递归步骤很多时，导致堆栈溢出。

尾递归就是为了解决上述问题，在尾递归中所有的计算都是在递归之前调用，

编译器可以利用这个属性避免堆栈错误，尾递归的调用可以使信息不插入堆栈，从而优化尾递归。

使用 @tailrec 标签可使编译器强制使用尾递归。

20、scala中的模式匹配

scala的模式匹配包括了一系列的备选项，每个替代项以关键字大小写为单位，每个替代方案包括一个模式或多个表达式，如果匹配将会进行计算，箭头符号=>将模式与表达式分离

例如:

```
obj match{
case 1 => “one”
case 2 => “two”
case 3 => “three”
case _ => default
}
```

21、case class和class的区别

case class:是一个样本类，样本类是一种不可变切可分解类的语法糖，也就是说在构建的时候会自动生成一些语法糖，具有以下几个特点：

1、自动添加与类名一致的构造函数(也就是半生对象，通过apply方法实现)，也就是说在构造对象的时候不需要使用new关键字

2、样本类中的参数默认是val关键字，不可以修改

3、默认实现了toString，equals,hashcode,copy方法

4、样本类可以通过==来比较两个对象，不在构造方法内地二属性不会用在比较上

class:class是一个类

1、class在构造对象的时候需要使用new关键字才可以。

22、谈谈scala中的隐式转换

所谓的隐式转换函数(implicit conversion function)指的事那种以implicit关键字申明的带有单个参数的函数，这样的函数会被自动的应用，将值从一种类型转换为另一种类型

比如：需要把整数n转换成分数n/1

implicit def int2Fraction(n:Int) = Fraction(n,1)

这样就可以做如下表达式求职：

val result = 3 * Fraction(4,5)

此时，隐士转换函数将整数3转换成一个Fraction对象，这个对象接着又被乘以Fraction(4,5)

在scala语言中，隐式转换一般用于类型的隐式调用，亦或者是某个方法内的局部变量，想要让另一个方法进行直接调用，那么需要导入implicit关键字，进行隐式的转换操作，同时，在Spark Sql中，这种隐式转换大量的应用到了我们的DSL风格语法中，并且在Spark2.0版本以后，DataSet里面如果进行转换RDD或者DF的时候，那么都需要导入必要的隐式转换操作。

23、scala中的伴生类和伴生对象是怎么一回事

在scala中，单例对象与类同名时，该对象被称为该类的伴生对象，该类被称为该对象的伴生类。

伴生类和伴生对象要处在同一个源文件中

伴生对象和伴生类可以互相访问其私有成员

不与伴生类同名的对象称之为孤立对象

SQL篇

1.SQL语句执行顺序

在实际执行过程中，每个步骤都会为下一个步骤生成一个虚拟表，这个虚拟表将作为下一个执行步骤的输入。 接下来，我们详细的介绍下每个步骤的具体执行过程。

1. FROM 执行笛卡尔积：FROM 才是 SQL 语句执行的第一步，并非 SELECT 。对FROM子句中的前两个表执行笛卡尔积(交叉联接），生成虚拟表VT1，获取不同数据源的数据集。FROM子句执行顺序为从后往前、从右到左，FROM 子句中写在最后的表(基础表 driving table)将被最先处理，即最后的表为驱动表，当FROM 子句中包含多个表的情况下，我们需要选择数据最少的表作为基础表。
2. ON 应用ON过滤器：对虚拟表VT1 应用ON筛选器，ON 中的逻辑表达式将应用到虚拟表 VT1中的各个行，筛选出满足ON 逻辑表达式的行，生成虚拟表 VT2 。
3. JOIN 添加外部行：如果指定了OUTER JOIN保留表中未找到匹配的行将作为外部行添加到虚拟表 VT2，生成虚拟表 VT3。保留表如下：
    - LEFT OUTER JOIN把左表记为保留表
    - RIGHT OUTER JOIN把右表记为保留表
    - FULL OUTER JOIN把左右表都作为保留表
    - 在虚拟表 VT2表的基础上添加保留表中被过滤条件过滤掉的数据，非保留表中的数据被赋予NULL值，最后生成虚拟表 VT3。
    - 如果FROM子句包含两个以上的表，则对上一个联接生成的结果表和下一个表重复执行步骤1~3，直到处理完所有的表为止。

4.WHERE 应用WEHRE过滤器：对虚拟表 VT3应用WHERE筛选器。根据指定的条件对数据进行筛选，并把满足的数据插入虚拟表 VT4。由于数据还没有分组，因此现在还不能在WHERE过滤器中使用聚合函数对分组统计的过滤。同时，由于还没有进行列的选取操作，因此在SELECT中使用列的别名也是不被允许的。

5. GROUP BY 分组：按GROUP BY子句中的列/列表将虚拟表 VT4中的行唯一的值组合成为一组，生成虚拟表VT5。如果应用了GROUP BY，那么后面的所有步骤都只能得到的虚拟表VT5的列或者是聚合函数（count、sum、avg等）。原因在于最终的结果集中只为每个组包含一行。同时，从这一步开始，后面的语句中都可以使用SELECT中的别名。

6.AGG_FUNC 计算聚合函数：计算 max 等聚合函数。SQL Aggregate 函数计算从列中取得的值，返回一个单一的值。常用的 Aggregate 函数包涵以下几种：

- AVG：返回平均值
- COUNT：返回行数

- FIRST：返回第一个记录的值
- LAST：返回最后一个记录的值
- MAX： 返回最大值
- MIN：返回最小值
- SUM： 返回总和

7.WITH 应用ROLLUP或CUBE：

对虚拟表 VT5应用ROLLUP或CUBE选项，生成虚拟表 VT6。

CUBE 和 ROLLUP 区别如下：

CUBE 生成的结果数据集显示了所选列中值的所有组合的聚合。

ROLLUP 生成的结果数据集显示了所选列中值的某一层次结构的聚合。

8.HAVING 应用HAVING过滤器

对虚拟表VT6应用HAVING筛选器。根据指定的条件对数据进行筛选，并把满足的数据插入虚拟表VT7。

HAVING 语句在SQL中的主要作用与WHERE语句作用是相同的，但是HAVING是过滤聚合值，在 SQL 中增加 HAVING 子句原因就是，WHERE 关键字无法与聚合函数一起使用，HAVING子句主要和GROUP BY子句配合使用。

9. SELECT 选出指定列：将虚拟表 VT7中的在SELECT中出现的列筛选出来，并对字段进行处理，计算SELECT子句中的表达式，产生虚拟表 VT8。

10.DISTINCT 行去重：将重复的行从虚拟表 VT8中移除，产生虚拟表 VT9。DISTINCT用来删除重复行，只保留唯一的。

11.ORDER BY 排列：将虚拟表 VT9中的行按ORDER BY 子句中的列/列表排序，生成游标 VC10 ，注意不是虚拟表。因此使用 ORDER BY 子句查询不能应用于表达式。同时，ORDER BY子句的执行顺序为从左到右排序，是非常消耗资源的。

12.LIMIT/OFFSET 指定返回行：从VC10的开始处选择指定数量行，生成虚拟表 VT11，并返回调用者。