方法区的类被回收的三个条件？

1该类的所有实例对象被回收

2加载该类的classLoader被回收

3没有任何引用指向该类的class对象

为什么要做一个大操作后要初始化连接？

1. mysql每一个连接都又一个管理连接的对象
2. 在执行操作时占用的临时内存放在这个对象在管理
3. 只有连接释放，管理该连接的对象才会释放，内存才会释放

为什么rows\_examined的大小不一定等于存储引擎实际扫描的行数？

这就要从rows\_examined的计算方式说起了，rows\_examined是指执行器扫描到的行数，是根据执行器调用一次存储引擎接口获取到的行数累加得到的。假设执行器调用存储引擎获取到一行，rows\_examined就会+1，但是这个过程中存储引擎可能扫描了多行数据。

什么是脏读、不可重复读、幻读？

脏读，读到了一些脏数据，本质受由于事务可以读取别的事务未提交的数据导致的。举个例子，事务A将某行的数据从1改为0，修改已经完成，但还没提交。此时事务B读取改行数据的这个值，发现值为0，但事务A提交之前出现了错误，事务进行了回滚，在该行该列的值回滚成了1，而事务B读取到的值为0，事务B将0返回给程序徐，程序会根据这个值为0做出逻辑判断。举个生活中的例子，假如我和我的朋友去买衣服，我们看上了同一件衣服，我朋友先拿到了这件衣服，去商店付款，付款完成后，商店系统会修改该衣服的库存，我以看库存减为0，基于这个判断，我于是去了另外一家店购买衣服。但是由于某些原因，朋友不想要这件衣服了，于是选择当场退货，这件衣服的库存又被回复成了1。而我去了别家店买衣服，造成了商店的损失，这个原因是我读了朋友购买衣服这个事务还没有提交的数据。

不可重复读：针对数据脏读，数据库使用了“读提交”的隔离机制，防止其它事务读取某个事务未提交的数据。但都提交隔离机制无法解决不可重复读的问题，因为某个事务读取c=1的值后，另外一个事务对c进行了修改c=2，并提交，此时该事务再读c是，会读到c=2。这就是不可重复读。 不可重复读的解决办法是使用行级锁，将读取的某行数据要锁定该锁，直到该事务将所有读取操作完成，才释放该锁，允许别的事务进行修改。

幻读是针对整张表，读取数据总量前后有区

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 错误 | 描述 | 产生原因 | 针对的级别 | 针对的操作 | 解决办法 |
| 脏读 | 读取到了脏数据 | 读取了其他事务未提交的数据 | 行 |  | 读提交 |
| 不可重复读 | 同一行，前后读取的数据不一致 | 在事务读的过程其它事务修改了该行并提交了 | 行 | Update | 行级锁 |
| 幻读 | 同一张表，前后读取的数据总量不一样 | 在事务读的过程中其它事务插入了会删除了该表中的某些行 | 表 | Insert  Delete | 表级锁 |

隔离性

读未提交、读提交、重复度、串行化

乐观锁是用了解决多个事务同时尝试修改同一条数据而导致数据不一致的问题，其核心思想是，在要更新数据时检查数据是否被更新过（当前版本号<=读出来的版本号），若被其它事务更新过，则当前事务的更新作废，要重新开始

介绍一下Object常见的方法？

Object类Java类层级最最顶部的类，所有类的父类，Object有一些方法：

1、hashcode方法，它的声明语句是：public native int hashcode(); 该方法会生成一个对象的hash值，是int类型。hashcode方法设计出来是为了方便使用哈希表。

2、equals方法，该方法声明语句是public boolean equals(Object obj); 该方法用于判断两个对象是否相等，默认的实现方式是返回两个对象的地址值是否相等

HashMap、HashSet等数据结构通常不会有equals来比较，而是使用compareTo或者compare方法。因为equals方法不是强制重写的，如果某个类没有重写equls方法，就会使用默认的equlas，着显然不太合理，因为默认的是比较地址值，对于需要其它比较方法的类，实现comparable或comparator接口会比较好一些。通过阅读HashSet的源码，因为一个set里面不能有重复的元素，发现对象的相等比较是通过compare或compareTo方法。

2、toString() 方法，该方法声明语句是public String toString();。默认是返回【类名@hashcode的十六进制表示】，在System.out.print()中，对象+字符串常量，对象+字符串变量，一些情况JVM会自动调用toString方法。该方法不是必须重写的，因为有默认实现。

// 下面几个是final修饰，不允许重写

3、getClass，public final native Class<?> getClass(); 返回该类的class对象 final修饰，不可以被重写。native修饰，由本地语言实现。

4、wati，由三个重载的方法，都用final修饰。

5、notify notiyAll final修饰，不允许重写

// 下面几个是protected，无法重写

6、clone

7、finalize

Object是所有Java类的父类，里面包含一些重要的方法

1. toString toString方法返回一个字符串，默认返回类名@hashcode的十六进制，在调用print函数或者一个对象与字符串相加时等情况JVM会自动调用toString方法。自定义类通常会重写这个方法以输出人方便阅读的信息。
2. equals和hashcode，equals的返回值是boolean 默认的实现方式是用==比较，即比较地址值是否相同，hashcode返回对象的hash值，这个值与对象的内存地址有关。在用hashset或hashmap存储自定义类是通常会重写该方法，以保证存储的结果符合预期。如果没有重写，在hashmap比较两个key是否重复是会先比较hashcode值，再比较equals，其中有一个为true就认为重复，所以当两个对象hashcode不同但equals结果相同时，不会认为时重复的，依然会存入hashset中，这就会造成困惑。所以要重写这两个方法以保证它们再比较两个对象是否相同时一致。如果两个对象相等，则hashcode值应该相同，如果不相同，则hashcode值也不同。String类的源码就同时重写这两个方法，Integer类也是，它们都重写了hashcode和equals方法以保证两个对象是否相同时的一致性
3. wait notify notifyall，中三个方法用于线程同步，wait方法由被上锁的对象调用，会导致当前调用该方法的线程进入等待状态，并释放锁，直到该对象调用notify或notifyall方法，这两个方法的区别是notify方法会重等待该对象的线程中随机挑选一个线程进入Runnable状态。notifyall放啊会把所以等待该对象的线程唤醒，让它们去抢占锁，抢到锁的线程会执行同步代码，没有抢到锁的线程会进入blocked状态。这三个方法加上synchronized关键字可以实现一个生产者消费者模型。生产者生产消息后会调用notify唤醒消费者，消费者消费完消息后会调用wait进入等待状态。同样当队列满后生产者会调用wait进入等待状态，当消费者消费消息后会notify唤醒生产者。

Java中Object类不是一个抽象类，为什么可以声明一个没有函数体的函数？

我们以hashcode函数为例，hashcode函数声明是public native int hascode();从形式上看上去是没有函数体，这是由于native关键字存在，它告诉JVM该函数的实现有本地语言C/C++，而不是Java语言来实现，所以看上去是没有函数体，实际上是有的。

为什么hashtable不能把空作为key？

因为hashtable必须要存储的对象实现hashcode和equals方法，该类会调用

为什么哈希表这个数据结构要使用到hashcode函数？

由hashtmap源码可知，在比较key是否重复是，会调用对象的hashcode函数计算hash值，如果哈希值映射到hash表上的索引以及存放了对象，会进一步比较这两个对象用hash值比较、==比较、equals比较

为什么byte类型的数据取值范围是[-128,127]？

简单一句话来说就是-128在8位带符号整数是没有直接的机器数表示的，-128是人们将-0作为8位带符号整数的最小值

什么是机器数、真值、原码、反码、补码？

机器数：数在计算机中二进制表示，用于计算机的存储和表示

真值：由于机器数符号位的存在，直接将机器数转换位十进制的结果不一定正确。真正就是考虑到符号位后机器数的真正的数值

原码=符号位+数值位

反码：正数：反码=原码；负数：反码=原码出符号位外按位取反

补码：正数：补码=原码；负数：补码=原码的反码 + 1

String a = “a” 和String b = new String(“b”)经历了什么过程？为什么没有像Integer那样，String.valueof()没有被调用？用图能否画出来？用文字能否描述一遍？

1、与Integer、Double、Float、Bollean、Long类型不同，String直接就是引用数据类型，没有对应的基本数据类型，所以不存在装箱和拆箱的操作，也就没有调用String.valueOf

2、字符串有常量池的概念，位于JVM内存的方法区中。



程序编译为字节码后被加入到JVM内存进行加载。程序执行到main方法时会分配一个mian方法的栈帧用于存放局部变量。执行到String a 时会在main栈中声明一个a的变量，然后程序进一步执行发现等号右边是一个字符串常量。这是JVM会直接去方法区的常量池中查找有么有字符串“a”，如果有就将其地址返回给变量a，如果没有，就在常量池中创建字符串常量”a”，并将地址范围。对于第二条语句，前面在main栈中创建b变量与第一条语句一致，不同的是等号执行到右边，JVM发现是new String，所以JVM会在堆中创建一个String类对象，这个String类对象里面有一个成员变量value，用private final 修饰，是一个字符串数组，value本身是一个数组引用，然后，JVM回去方法区中的常量池中查找有没有字符串”b”，如果没有就创建。将常量池中”b”串的地址值传给value，即value指向了常量池中字符串”b”。最后，将这个在堆中的String对象的地址值传给b。

什么是元数据，metadata？

元数据是指用于描述其它数据的数据。举个例子，一本书的内容就是数据，而描述这本书的出版日期、作者、ISBN号等这些就是元数据。一个类的成员包括变量和方法，变量和方法的具体内容就是数据，而一个类有哪些字段或方法是类信息(元数据)，类名、父类、实现的接口这些描述类的数据就是元数据

方法区用来存放哪些内容？

类信息、常量、静态变量

为什么说一个String类对象是保存一个字符串常量？

这个可以通过阅读String类的源码得知，String对象保存的数据本质上是一个字符数组，这个字符数组是这样定义的private final char value[]；关键字final修饰，表示这是一个字符串常量。

char a = ‘我’ 对不对？为什么？

这是对的，因为Java中字符使用Unicode编码，一个char类型占两个字节。

你能想到有哪些String类的构造器？



一些常用的String类构造器

值得注意的是，new String(String original) 这个构造函数会执行两个步骤

this.value=original.value

this.hash = original.hash

注意第二个，hash值也会传过来。所以两个不同地址的变量，hashcode值的结果是相同的，因为hash值会复制过来

实现comparable和comparator接口有什么区别？

comparable需要通过类对象调用comparTo方法。comparator可以用第三方类调用comapare方法比较两个类对象。

串行化是什么？为什么要串行化？为什么实现了串行化就可以在网络中传输？串行化的反义词是什么？

串行化是将对象变成字节流。在网络通信中，只有基本数据类型和字节流可以传输。原因：网络传输TCP/IP协议是以字节为基本传输形式，更高级的数据结构这些网络协议是无法识别的，它们只能发送或接收字节流，更底层数据链路层或物理层的协议连字节的概念都没有，只是发送bit流。而基本数据类型可以直接转换为字节，如btte是1个字节、short和char类型是2个字节、int类型是4个字节等 long类型是8个字节

实现charsequence接口有什么用？

实现charsequence可以统一对字符串序列的使用规范。该接口定义了length、charAt等方法

为什么说“字符串的本质是char数组”?

由String类的源码可知，真正存储字符串的是字符数组

final int[] v = {1, 2, 3} 为什么v[0] = 4可以运行？不可修改值的是什么不可修改？

final修饰变量表示该变量的值不能修改，这里v是只想堆中数组的引用，存储的是地址值，所以修改v的地址值，即指向一个新的int数组区域是不允许的，但用v[0]修改的是堆中某个区域的值，这不是产量，是可以改变的。所以这里的不可修改是指v的指向不能修改。

Java中声明数组有哪些方式？

int[] a;

int a[];

String类为什么hashcode相同的字符串，并不一定指向同一个内存地址即s1!=s2?

hashcode的计算通常与内存地址相关，如果hashcode相同，内存地址应该也相同。但是对于这个情况：String a1 = “a”; String a2 = new String(“a”); a1和a2的地址值是不同的，前者指向常量区、后者指向堆区。但在第二条构造语句执行时会执行this.hash = “a”.hash; 于是a1.hashcode是等于a2.hashcode值的。

String s = “hello” + “abc”; 创建了多少个对象？

JVM会帮我门优化，等价于String s = “helloabc”; 编译器会看一下，所以一个创建了一个对象。编译器会判断常量池中的对象是否有引用变量指向。

String a = “hello”; String b = “abc”; String c = a +b; 这个过程创建了多少个对象？

关键是分析String c = a + b是怎么执行的？这个过程可以debug一下(可以的)？

这个过程最终池中是3个对象

画出下面这段代码的内存动态图？做一个视频？





Java8之后方法区被metaspace代替?

首先这个说法是错误的，要先弄清楚几个概念：

方法区：JVM在运行时管理的一块数据区域。方法区只是一个规范，一个定义。具体实现，不同版本的JVM由出入。

永久代：对于HotSpot虚拟机，在jdk1.8之前，方法区的具体实现叫做Permanent Generation

元空间：对于HotSpot虚拟机，从jdk1.8开始，PermG被舍去，方法区的具体实现叫做Metaspace

Metaspace存储class metadata，描述类的信息、方法的元数据、静态变量、常量池，字节码

存储已被虚拟机加载的类信息、常量、静态变量、即使编译器编译后的代码缓存。

你能不能手写一个泛型类？

可以。

Java Object wait方法有哪些重载？是用来干什么的？为什么要用它？为什么出现在Object类中？

wati(timeout) 这个是native的，由本地语言实现

wait(timeout nanoseconds)

wait() 实际调研wait(0) 表示无限期等待

该方法只能被上锁的对象调用，synchronized(对象){

对象.wati(); // 执行该同步代码的线程会主动进入wait状态，并释放锁，直到该对象发出notify通知或者notifiyAll通知

}

进程和线程的区别？

进程是程序的一次执行过程，或者是一个运行的程序就是进程。

代码(程序)---运行🡪 进程🡪占用内存空间。

线程：由进程创建的，它是进程的一个实体。一个进程可以拥有多个线程

进程是操作系统资源分配的最小单位，用于个理资源，比如内存资源，线程是CPU调度的最小单位，线程包含在进程中，是进程实际运作单位。

什么是并发和并行？

并发：单核CPU，同一个时刻只有一个线程在执行，不同时刻切换到不同的线程进行执行，但由于CPU太快了，人看起来是同时在执行，这就叫并发

并行：多核CPU，同一时刻有多个线程同时在执行，这就是并行

对于多核CPU，并发和并行也有可能是同时进行的

创建线程的方式？

Java中创建线程有两种方式

1、继承Thread类，重写run方法

2、实现Runnable接口，重写run方法

3、通过Callable接口和Furture类实现

4、通过线程池创建

为什么有了继承的方式创建多线程，还要发明接口的方式？

因为Java是单继承的，如果某个类已经继承了其他类，再使用继承的方式实现多线程就无法实现。

如何停止一个线程的运行？

1. 自动方式：线程完成自己的任务后就会自动退出
2. 通知方式：使用变量控制run方法退出，通知该线程此时需要结束run方法了

Runnable和Callable创建线程有什么区别？

1. Callable方式创建线程可以得到返回值，重写call方法是由返回值的，而Runnable重写的run方法没有返回值
2. Runnable里面重写run方法不能抛出异常，Callable可以
3. Runnable和Callable都需要用Thread.start启动线程
4. Callable需要用FutureTake.get获取返回值，当在主线程调用该方法时，主线程会阻塞，直到结果返回，不调用该方法，主线程不会阻塞。

Thread类和Runable接口的最大区别是什么？

1. 从本质上来将，继承Thread类的方式实现多线程和实现Runnable方式没有本质的区别，因为从Java源码可以发现，Thread类本身也实现了Runnable接口。
2. 实现Runnable接口更加适合多个线程共享一个资源的情况，并且避免了单继承的限制。

为什么说实现Runnable接口更加适合多个线程共享一个资源的情况？

凭什么说这句话，当多个线程需要共享同一个资源时，例如访问同一个文件、数据库连接、网络连接时，我们把这些资源作为一个对象的属性，并写入调用这些资源的方法，然后实现Runnable接口，在run方法里面调用上述对象的方法。只需要创建一个资源对象，当创建新的线程时，就把该对象传入。从而让多个线程共享同一个资源。

如果采用继承Thread类的方式，创建一个新的线程，就会new一个新的对象，没有线程拥有一份独立的内存空间，每个线程都会拥有自己的资源、而不是共享一份资源。会造成资源的浪费

线程的几种状态，以及状态的转换，生命周期？



什么是线程安全？为什么不安全？

线程安全不是指线程的安全，而是指内存的安全。这与操作系统有关，现在主流的操作系统都是多任务的，即多个进程同时运行，为了保证进程之间互不干扰，操作系统会给每个进程分配一块内存区域，每个进程只能访问自己的内存空间，而不能访问别人的内存空间。每个进程的内存空间会有一块特殊的公共区域，被称为堆，进程中的所有线程都可以访问，这就是造成问题的潜在原因。线程安全是指某块内存，或者某个对象，在被多个线程同时访问，线程间按照某种方式访问时，访问的结果总是正确的，就说明这个对象、这个内存时线程安全的。

1. 线程安全不是指线程的安全，而是指内存的安全或者对象的安全
2. 当多个线程同时访问同一个对象时，线程间按照某种调度方式访问，调用这个对象的行为都可以获得正确的结果，就称这个对象时线程安全的，反之称这个对象是线程不安全的。

为什么不安全：由于堆内存的数据可以被任何线程访问到，在没有限制的情况下存在被意外修改的风险，堆内存空间在没有保护机制的情况下，对多线程是不安全的，因为放进去的数据，可能被别的线程破坏。

线程安全不是指线程的安全，而是指内存的安全，更具体一点，是指对象的安全，因为JAVA中对象是存放在一块被称为堆的公共内存区域，任何线程都可以访问。如果没有保护机制，某个线程存放到堆中的数据有可能被其它线程修改的风险。而如果多线程并发访问同一个对象，按照某种方式进行访问，调用对象的行为都是正确的，我们就称这个对象是线程安全的。

一个类中有类变量、成员变量、局部变量，它们存放的位置分别是？

方法区、堆、栈内存

前两个是共享变量，需要考虑线程安全。最后一个是线程安全的，因为每一个线程，JVM都会分配一个线程栈，而局部变量存放在各个线程的栈空间中，线程之间是无法访问对方的栈内存的，栈内存空间是独立的空间，所以是线程安全的。

阻塞状态和等待状态的区别？

两者都是当前线程暂停执行的状态，阻塞状态时被动的、等待状态是主动的。更近一步，阻塞状态实在同步代码之外，等待某个对象的锁，等待状态是获取到某个对象的锁后，进入同步代码后主动执行wait函数，主动释放锁、等待其它进程使用完该对象后发出notify或notifyall，然后结束等待状态，进入runnable状态。

怎么实现线程安全？

1. 无状态实现。比如一个计算阶乘的静态方法、给的特定的输入就会输出特定的结果，不依赖外部状态，也不维护自身状态。因此，该方法是线程安全的。
2. 不可变实现。想要保证某个对象是线程安全的，在声明该对象的类时将所有的字段声明为privat 和 final 并且不提供setter。那么，该对象一定是线程安全的。
3. 线程私有ThreadLocal字段。

锁怎么实现线程安全？

以同步锁为例，java sychronized可以实现对对象加上同步锁，sychronized实现线程同步由两种方式：同步代码块、同步方法。以同步代码块为例，它的使用格式是synchronized (对象){} 要求多个线程上锁的对象保持一致、不然就失去了意义。花括号里面是执行同步代码，同步锁保证在任意一个时刻最多只有一个线程在执行该同步代码，因为对象只有一把锁，哪个线程获得了就可以进来执行，在当前线程执行完同步代码里面的任务时，其它线程只能被阻塞，其它线程也无法访问共享的内存空间、这是就保证了共享资源的线程安全。

多线程什么场景下会发生死锁 ， 有什么具体的办法可以避免死锁？

多个线程占用了对方锁资源，需要等待对方将锁释放后，自己才能释放锁，结果对方线程也是这样等待，大家都不想让，最终造成死锁。

首先是要在写业务代码的时候避免多个线程给对方的资源上锁，然后在发现死锁的时候让其中某个线程主动释放锁，先让别的线程完成任务。

避免死锁的方式：

1. 按顺序加锁
2. 获取锁的时限，获取某个锁加上时限，超过这个时间就释放锁
3. 死锁检测

wait和sleep的区别，没有说清楚，HR提醒用wait时会释放锁，sleep不会?

wait和sleep都会让线程进入Waiting状态或Timed\_Wait，如果wait加上timeout参数，则也会进入Timed\_Wait状态。而wait在让当前线程进入Waiting状态时会主动释放锁，sleep则不会。另外如果不加timeout参数，就会无限期的等待该对象调用notify

1. wait是Object里面的，所有对象都可以调用，Sleep是Thread类所有的
2. wait是非静态成员方法，需要通过对象调用，sleep是静态成员方法，可以通过类名调用
3. 调用wait可以不需要参数，调用sleep必须要参数
4. sleep会抛出interrput异常
5. wait让当前线程进入等待状态并释放锁，sleep让当前线程进入等待状态不会释放对象锁。

多线程了解吗，为什么要多线程？

为什么要多线程：第一，假设我们的程序只有一个线程，当该程序处理某个比较耗时的任务、如网络传输、文件读写时，其它任务会被阻塞，只能等待该任务执行完毕，在用户看来，程序就卡住了，不会执行其它请求，这对用户的体验非常不好。如果采用多线程， 用一个线程去处理比较耗时的任务而不会影响主线程的执行，程序还是可以接收用户请求完成其它任务，所以多线程可以提高程序的并发现和响应性。第二，由于硬件的发展，现在的CPU多是多核的，可以并行执行多个线程，使用多线程可以提高CPU的利用率。

总的来说，为什么要用多线程分为两个方面: 提升用户体验；提高CPU利用率。

如何启动一个线程？

1. 某个类继承Thread类或者实现Runnable接口
2. 重写run方法，这里面是业务方法
3. 创建该类对象
4. 如果是接口方式：创建Thread类对象，将该类对象传入
5. 使用该类对象调用start方法，如果是接口方式：使用Thread类对象调用start方法

为什么用start不直接用run？

启动一个线程、start、最终会调用run方法。如果直接在mian方法里面调用run方法，相当于直接用主线程栈里面调用方法，并没有真正实现多线程。run方法就是一个普通的方法，相当于是一个串行化的执行。而start才会真正地去启动线程。在start0里面用多线程机制调用run方法。 start调用native的start0后，start0会启动一个新的线程，并调用run方法，然后将该线程变成可运行状态，等待CPU时间片，run方法具体什么时候执行是由CPU决定的，而不是start0决定，由CPU统一调度。run方法只是一个普通方法。

什么叫同步机制？

从以下两个方面理解同步机制：

1. 同步机制是保证在任意一个时刻t，最多只有一个线程在访问某个资源。
2. 同步机制：当有一个线程在对内存进行操作时，其它线程都不可以对这个内存地址进行操作，直到该线程操作完成，其它线程才能对该地址进行操作。
3. 假设有三个线程t1,t2,t3，t1在操作某一个资源，这时如果t2和t3也想操作该资源，就会被阻塞，直到t1操作完成之后，t2或t3中某一个线程在被选择来操作该资源，这就叫同步

多线程环境下为什么要引入同步的机制？

1. 在多线程编程中，一些敏感数据不允许被多个线程【同时访问】，所以引入同步机制，保证数据在任意一个确定的时刻，最多只有一个线程访问，以保证数据的完整性。
2. 引入同步机制，是为了控制多线程对共享资源的访问，以保证数据的正确性和一致性。

知道了什么是同步，为什么要同步，现在问在JAVA里面如何实现同步呢？

1 最简单的方式，加Synchronized关键字。

Synchronized有两种情况：同步代码块、同步方法

同步代码块 synchronized (对象){ //得到对象的锁，才能够操作同步代码。

// 同步代码，任意一个确定的时刻，最多只有一个线程在执行该部分代码

}// 操作完成后，归还对象的锁。对象要求是同一个对象，如果对象一直在变，上锁就没用了。

同步方法

public synchronized void m(String name){

// 需要被同步的代码

}

在任意一个时刻，最有一个线程在执行该方法

同步方法：同步非静态方法、同步静态方法。同步非静态方法等价于给this对象加锁==synchronized(this){}。同步静态方法等价于给该类的class对象加锁

synchronized有两种实现方式：同步代码块、同步方法；但本质上只有一种：即同步代码块。

同步代码块：synchronized(对象) 要求不同线程的对象必须是同一个内存空间且

同步静态方法==synchronized(当前类.class){}

同步方法==synchronized(this){}

必须保证多个线程锁的对象是同一个对象

为什么要线程安全，举例子，说了数据库脏读的例子， Java方面不清楚，面试官给了非元子性的操作 i++的例子怎么实现线程安全？

线程安全本质是内存的安全，java中由于堆是一公共的内存区域，任何线程都可以访问，当某个对象被多个线程共享时，当前线程操作块内存区域时有可能其它线程也修改了内存区域，这导致对象的行为结果不确定或不正确。举个例子，Java实现一个售票系统，一共有3个售票窗口，开启了三个线程并发地卖票，这3个线程共享剩余票数这个变量，每个售票窗口在执行售票业务代码时会判断剩余票数是否大于0，如果小于等于0，就提示票已经售空，否走就执行售票业务代码，并将票数减一，若售票窗口1此时访问剩余票数这个变量，发现为1，于是判断成功，准备执行售票业务代码，但售票窗口2比售票窗口1先一步拿到，先将票数减1了，但售票窗口1已经进入了售票逻辑，最后剩余票数就会变成-1

i++操作为什么当i是成员变量时是线程不安全的？

1. 线程安全本质上是内存安全，多线程访问同一个内存空间时会导致线程安全问题
2. 如果i是局部变量，i会存放在栈中，而JVM会给每个线程分配一个栈，这个栈空间是相互独立的，线程之间无法访问，所以i如果是局部变量，就一定是线程安全的。
3. 当i是成员变量是，i所属的对象是存放在堆中的，这块内存是任意线程都可以访问的，所以可能存在线程安全问题。
4. 在3的基础上，i++不是一个原子性的操作，i本身是在堆中，i++会被看作i=i+1，当前线程有一个线程栈，首先会把在堆中的i的值读取到栈中tp=i，压入栈顶，然后将常量1压入栈顶，接着计算tp2 = tp +1，将tp2压入栈顶。最后把栈顶的tp2重新写回堆中的i。综上所述：当i是成员变量时，i++的过程可以看成3步：读取i的值、计算i+1、向i写入结果。
5. 所以i++不是一个原子操作🡪在线程1计算i++还没写入时，如果线程2就读取了一个脏数据，所以这些线程不安全的。

Java里面的锁有哪些？

1、互斥锁，synchronized(非公平锁，才释放锁的线程可以立即去抢锁)

java里面的线程和操作系统的线程一样吗？

不一样

Java中线程实在JVM上执行的线程，是一种轻量级线程，由java语言支持，并受到JVM控制。

操作系统线程是在操作系统中执行的线程，是一种重量级线程，由操作系统支持。并受到操作系统控制。

Java线程实在操作系统线程之上进行封装的，它们可以映射到操作系统线程上。但是Java线程不能直接访问操作系统线程。必须通过JVM间接访问。

wait-notify、线程的状态？

Waiting状态

notify 和 notifyAll 的区别？

某个对象调用notify和notifyAll都需要当前获取到该对象的锁。

对象.notify会随机唤醒一个在等待当前对像的线程，使其进入runnable状态

对象.notifyAll会唤醒所有在等待当前对像的线程，让他们竞争该对象的锁，没有竞争到的线程进入Blocked状态。

线程怎么才能称为一个对象监视器的所有者？

线程获取到对象的锁即可，synchronized(读写)、同步方法、同步静态方方法

调用某个对象的wait和notify/notifyAll方法都需要当前线程称为该对象的监视器所有者，不然就会报IllegalMonitorStateException异常

什么叫程序？

为了完成某种任务，用某种语言编写一组指令的集合。通俗地将，就是我们写的代码就叫程序。代码就是程序

什么是进程？

运行的程序叫做进程

什么是静态代理，什么是动态代理，他们有什么区别？

静态代理和动态代理都是设计模式中的代理模式。代理模式本质是：利用一个中间代理对象来控制原目标对象，以达到增强原目标对象的功能。

代理提供了对目标对象的额外访问方式，即通过代理对象访问目标对象，这样可以在不改变目标对象的前提下、提供额外的功能、拓展目标对象的功能。

静态代理：之所以叫静态，是因为目标对象和代理对象实现同样的接口，在编译阶段编译器就知道目标对象要代理的方法，不需要等到程序运行起来。优点是可以在不修改目标对象的前提下拓展目标对象的功能。缺点是：1 由于要代理类和目标类要实现同一个接口，会导致代理类的数量增多，冗余。2 不易维护，如果接口里面的方法变化或着要增加方法，那么，所有的实现类都要修改。

动态代理：又称为JDK代理或接口代理。要求目标对象必须实现接口。代理对象根据目标对象实现的接口动态的在内存中构造代理对象。

它们的区别在于：1、静态代理在编译阶段已经实现，编译完成后代理类是一个class文件。2 动态代理实在运行阶段动态生成的，编译后代理类没有class文件，而是动态生成字节码，加载到JVM中。

Java中new一个对象时，初始化过程是什么？

类加载器加载类信息、首先执行类层级别代码(静态成员变量、静态代码块)

分配对象空间，并将对象中的成员变量设置为默认值

执行初始化代码块

执行成员变量初始化语句，（成员变量声明语句，没有初始化的用默认值，不会执行）

执行构造函数中的代码

一个类中由哪些东西？

类级：静态成员、静态代码块

对象级: 成员、代码块

构造方法

Java各个类各个部分的执行顺序是什么【代码块部分】？

类级别的部分在JVM加载类的时候就会执行，JVM加载类的时机有：调用该类的静态成员、声明该类的引用、构造该类的对象

对象级的部分只有在构造该类对象的时候才会调用

执行顺序：

类层级🡪成员变量默认值(分配对象空间是就会初始化一个默认的值)🡪成员变量初始化语句🡪构造方法

类层级🡪对象级🡪构造方法



Integer对象是不可变的？

是

所有的包装类都是不可变的吗？

是的，包括String类也是。

synchronized (number) {

if (number <= 0) {

System.out.println("售票结束...");

break;

}

try {

Thread.sleep(50);

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println("窗口" + Thread.currentThread().getName() + "售出一张票，剩余票数=" + (--number));

}

为什么这段上锁没有成功？

1. 上锁不是给引用上锁，而是给引用指向的内存区域上锁
2. Integer类是包装类，所有的包装类、String类都是不可变类，在Integer类真正存储数据的是pubic final int value，由final修饰，所以Integer类是不可变类
3. 当执行number++的时候：number会指向一个新的内存地址，其它线程进来一看，发现该内存地址没有上锁，于是对其进行操作。导致结果依然不正确。

Java使用synchronized上互斥锁时，什么时候会释放锁？

1. 当前线程的同步代码块、同步方法执行结束。
2. 在同步代码块、同步方法中遇到break、return语句。
3. 在同步代码块、同步方法中遇到未处理异常或者Error，导致异常退出。
4. 在在同步代码块、同步方法中执行了线程对象的wait方法，当前线程暂停，并释放锁。

什么时候不会释放锁？

Thread.sleep()

Thread.yield()

线程执行同步代码时，其它线程调用了该线程的suspend方法，也不会释放锁

挂起还是在Runnable状态

如何实现一个生产者-消费者模型？

1. synchronized关键字：解决线程安全问题，不会出现不正确的结果
2. synchronized无法解决忙等状态，就是生产者一直抢占cpu时间片判断队列满或消费者一直抢占cpu时间片判断队列是否空

什么是Spring？

Spring是一个轻量级、非侵入式的IOC和AOP框架。

轻量级：spring提供了很多服务，但这些提供给用户的服务完全由用户决定，想用什么服务，就用什么服务，而不是一股脑的全部给用户。

非侵入式：非侵入性是引入了框架(jar包)，对现有的类的结构没有影响。

IOC：控制反转

AOP：面向切面编程

从狭义的角度：spring就是指spring框架。



从广义的角度：spring生态体系。

如何能够使用spring？

spring本质也是Java语言写的，由很多Java类组成。相应使用spring的服务，就要引入相关jar包，jar包里面包含很多类。这些类提供了相应的服务。

引入jar包这个事情很麻烦，可以通过maven管理jar包，只需创建一个maven工程。

需要用到什么服务就引入对应的依赖

spring发展历程？

1.X 支持IoC AOP

2．5X 支持注解

3.X 基于java 5

4.X 支持Spring boot 1.x

5.X 基于java8



Spring 有哪些特性？

1. IoC和DI的支持：Spring核心就是一个大的容器工厂，可以维护所有对象的创建和依赖关系。Spring工厂用于生成Bean，并且管理Bean的声明周期，实现高内聚，低耦合的设计理念。高内聚是指模块内、低耦合是指模块间
2. AOP编程的支持，Spring提供了面向切片编程，可以方便实现对程序进行权限拦截、运行监控等切面功能
3. 声明式事务的支持，支持通过配置来完成对声明式事务的支持，只要声明某个方法是事务，不需要通过硬编码方式。以前重复的一些事务提交、回滚的JDBC代码，都可以不用自己写了。
4. 快捷测试支持：Spring提供对Junit支持，可以通过注解快捷地测试Spring程序
5. 快速集成功能：方便集成各种优秀框架，Spring不排斥各种优秀的开源框架，其内部提供了对各种优秀框架的直接支持
6. 复杂API封装：Spring对Java EE开发中非常难用的API 如JDBC JavaMail、远程调用等都提供了模块化封装，这些封装API使得使用难度大大降低。

Spring 有哪些模块呢？

Spring Core：Spring框架核心模块，提供了IoC和DI特性

Spring Context：Spring上下文容器，它是BeanFactory加强的一个子接口

Spring Web 提供web开发支持

Spring MVC 针对Web开发中MVC思想实现

Spring AOP 面向切面编程，提供了与AOP联盟兼容的编程实现。

Spring 有哪些常用注解呢？

1. Web
2. 容器
3. AOP：
4. 事务：

@Transactional：该注解作用于要开启事务的方法，即声明式事务开启



Spring 中应用了哪些设计模式呢？



工厂模式：Spring本质是一个大工厂，使用工厂模式通过BeanFactory、ApplicationContext创建bean对象

代理模式：Spring AOP就是通过代理模式实现的，分为静态代理和动态代理

单例模式：Spring中的Bean默认都是单例的，这样有利于容器对Bean的管理

模板模式：Spring JdbcTemplate, RestTemplate等以template结尾对数据库、网络连接进行操作的模板类就用到了模板模式

观察者模式：Spring事件驱动模型就是观察者模式很经典的一个应用

适配器模式：Spring AOP的增强或通知使用了适配器模式

策略模式：Spring中有一个Resource接口，它的不同实现类会根据不同策略取访问资源。

说一说什么是 IOC？什么是 DI?

什么是IOC：

1. IOC全称是Inverse of Control，IOC是一种设计理念。
2. 它是指将对象的控制权交由第三方管理。由代理人负责创建和管理对象，消费者通过代理人获取对象。
3. Ioc的目的是降低对象之间直接耦合
4. 加入IoC容器将对象统一管理，让对象关联变为弱耦合

什么是DI依赖注入：

1、IoC是设计理念，是宏观目标

2、Dependence Injection是实现IoC理念的具体的技术实现，是微观实现

3、将某个类所依赖的对象实例注入给这个类。

4、DI在java中是利用反射技术来实现运行时对象的创建与关联。

所谓IoC就是将我们对象的控制权，由程序本身转嫁给第三方，这个第三方通过一个容器，统一地对我们系统中的对象进行管理，这就是IoC的理念。

spring IoC容器的职责？

对象的控制权交由第三方统一管理(IoC控制反转)

利用Java反射技术实现运行时对象的创建于关联(DI 依赖注入)

基于配置提高应用程序的可维护性和可扩展性。

什么是反射？

反射技术是指在程序运行过程中可以拿到一个对象的信息。

如何对spring IoC进行配置，Spring IoC核心配置文件是什么？

在resource目录下创建application.xml文件，这是spring IoC核心配置文件。所有对象的创建以及对象之间如何关联就是在这个文件中设置的。

XML需要有一个xml头，XML声明，schema定义，定义xml里面的标签以及格式。有了这些头部内容，spring框架就可以去解析该xml文件了。

程序中绝大部分静态的信息不用硬编码到程序中，而是采用配置的形式放到xml文件中，一旦这些信息发生变化，不需要重新编译，而是直接修改xml文件就行了

idea一个project里面有module，一个module里面有facets，这个facets是什么？

facets是一个额外的框架特定的配置。

聊一聊spring实例化对象？

spring实例化对象是通过IoC容器，IoC容器的实现机制在程序运行阶段，利用反射技术获取对象所有信息，然后动态地进行实例化。实现IoC容器有三种方式：基于XML配置文件，基于注解 基于Java Config类。总体上分为两种：基于XML配置方式，基于注解方式。每个方式都各有优劣，使用XML配置方式将对象的创建于关联写入配置文件，便于后期的改进于维护，也便于项目切分。而使用注解方式则摆脱了繁琐的XML配置，对于程序员编码更加方便。

spring管理Bean的方式有哪些？

基于XML配置文件

基于注解

基于Java代码config配置

spring如何通过xml方式管理对象(Java Bean)？

引入依赖

创建applicationContext.xml

引入xml头

使用bean标签 id class属性

property子标签 name value ref属性

Application 是一个接口，有很多实现类，ClassPathxmlApplicationContext是实现类，会读取applicationContext.xml文件并初始化IoC容器并创建对象。这个contex就是ioc容器

三种XML实例化Bean的方式？

1. 基于构造方法实例化对象（大多数常见）
   1. 基于默认构造方法实例化对象
   2. 基于有参构造方法实例化对象

constructor-arg 子标签

property子标签是调用Bean的set方法，没有走构造函数。

在IoC容器创建对象时去调用对象的有参构造函数构造对象

constructor-arg 有name属性 index属性

index是通过位置设置对应构造方法参数的值

根据constructor-arg标签数量去调用对应的构造方法

有多个构造方法时会根据name、数量、type来查找调用哪个构造方法

xml只能写字符串，调用对应的构造方法时会自动转换

尽量按名称设置构造方法的参数，尽量不要按照索引

使用constructor-arg标签表示使用带参构造，有两种该方式设置参数：名称和索引。具体调哪个构造函数与参数数量、name、type有关

1. 基于工厂（很少用）
   1. 基于静态工厂实例化对象

静态工厂通过静态方法创建对象，隐藏创建对象的细节‘

bean标签里面class写上静态工厂类，加上factory-method对象

* 1. 基于工厂实例的实例化方法实例对象

1. IoC容器对工厂实例化
2. 工厂实例化对象创建对象

需要两个bean，第一个bean指向工厂本身， 第二个bean是apple，属性有factory bean 和factory method

这两种方法最大的区别就是创建对象的方法是不是static的，如果不是，还需要多加一步，spring ioc创建实例化工厂实例，所以要多一个bean标签

基于工厂的方式还是会把new对象写入源代码中，这和原来的方式有什么区别吗？

作为工厂来说，其职责就是通过工厂创建对象，隐藏创建细节。其主要目的是在IoC容器之外，通过程序的方法帮助我们组织对象，这样做有一个好处，就是扩展spring IoC容器的功能

如何从IoC容器中获取Bean？

从ApplicationContex的实现类对象调用getBean方法

getBean可以传入类，也可以不传入后面用强转。

Bean标签 id 和 name属性的相同点和不同点？

相同的：

1. 作为id和name都是为对象设置在IoC容器中设置唯一标识，本质来讲都是一样的。
2. 两者在配置文件中都不允许重复，即使id与name之间也不能重复。
3. 如果有多个配置文件，是会允许重复的，后加载的对象会覆盖先加载对象，但指向它们的id name不会变，堆中的对象位置不会改变，内容会被覆盖

不同点：

一个对象可以有多个不同的name，但只能有一个id

当一个Bean标签既没有id也没有name，我们该如何获取呢？

要想从IoC容器中获取该bean，需要在getBean中添加该bean所属类的完整名称，getBean(“com.imooc.spring.ioc.entity.Apple”，Apple.class)

什么是工厂实例？

工厂实例是和静态工厂相对的，静态工厂的方法是用static修饰属于类层级。而工厂实例需要创建工厂实例对象在调用实例方法创建对象。

工厂模式是什么？

隐藏创建对象的细节，通过一个额外的工厂类组织要创建的对象

聊一聊IoC容器中加载配置文件时所使用的路径表达式？

什么是路径表达式：指在我们加载spring ioc配置xml文件是传入的特殊格式的字符串—“classpath:applicatonContext.xml”

作为ApplicationContex接口的实现类，ClassPathXmlApplicationContext类需要传入一个路径表达式告知这个类去哪里找到xml配置文件。

classpath 类路径是指编译后，有一个classes文件夹，里面有Java类顶层包以及其它文件，这就是类路径，Idea会把resource路径下的自动文件复制一份到类路径下。这就是路径表达式的意义所在

|  |  |
| --- | --- |
| 路径表达式 | 说明 |
| classpath:config.xml | 扫描当前项目的classpath路径下 |
| classpath\*:config.xml | 扫描当前项目的classpath路径下和所有jar包下的classpath路径下 |
| classpath:com/imooc/config.xml | 扫描当前项目的classpath路径下com/imooc路径下 |
| classpath:com/imooc/\*.xml | 扫描当前项目的classpath路径下com/imooc路径下所有以xml结尾的的文件 |
| classpath:com/\*\*/config.xml | 扫描当前项目的classpath路径下com路径下：任意子路径，一层或多层子路径 |
| file:c:/config.xml | file表示采用文件系统路径  c盘根路径下的config.xml |

能简单说一下 Spring IOC 的实现机制吗？

IoC是在运行时利用反射技术，在运行时，动态地将对象实例化，以及利用反射技术，对属性进行注入

IoC容器本质是一个map键值对对象，将beanId与对应的对象进行绑定，如果需要创建对象，就利用Class.forName获取要创建对象的类的类对象，然后调用newInstacne，使用默认的构造函数构造对象。如果遇到proerty标签，则利用method对象进行方法的动态调用。

说说 BeanFactory 和 ApplicantContext?

ApplicationContext继承BeanFactory接口，所以ApplicationContext用于BeanFactory接口的所有的功能

BeanFactory是使用懒加载方式，意味着只有在getBean的时候才会区实例化对象，而ApplicationContext接口在容器启动阶段就会去加载scope为singleton的对象

类对象的getResource(name)方法，name加 /和不加 /的区别

如果加/：会从classpath路径下开始查询

如果不加/：会把name看作时一个相对路径，相对于当前类所在的文件夹

聊一聊 spring IoC注解？

除了用XML文件配置Spring IoC容器外，Spring还支持使用注解和JavaConfig方式配置IoC容器。这三种方式只是表现形式不同，其本质原理都是一样的。

如何利用注解配置IoC容器？

注解是对所描述的类或者方法、变量进行额外的功能拓展或者增强。作为Spring，提供了大量的注解

注解的优势：摆脱繁琐的XML形式的bean与依赖注入的配置

注解是写在源代码中的配置信息，基于声明式的原则。因为注解本质上是一种声明，在代码中使用注解时，本质上是指声明一些信息，注解本身不会完成它所声明的功能，而是实现类利用反射机制，扫描某个类、方法、变量上有什么样的注解，再进一步执行功能代码。所以所注解的本质是声明。

利用注解可以让代码的可读性变得更好

Spring中，按照注解功能划分可以划分为哪几类注解？

1. 组件类注解：声明当前类的功能与自责
2. 自动装配注解：更加属性特性自动注入对象、或者注入值
3. 元数据注解：更细化辅助IoC容器管理对象

|  |  |  |
| --- | --- | --- |
| 类型 | 注解 | 说明 |
| 组件类注解 | @Component | 组件注解，通用注解，告知spring，当前类应该被IoC容器管理并实例化 |
| @Controller | 说明当前类是MVC引用中的控制类，Controller层 |
| @Service | 说明当前类用于服务层，Service层，是日常开发中用于实现功能的最核心代码，核心业务代码放在service层 |
| @Repository | 说明当前类用于数据持久层，Dao层 |
|  |  |  |
|  |  |  |
|  |  |  |

需要再spring IoC配置文件中开启注解扫描才可以使用注解

context：component-scan标签 base-package属性

component-scan会再IoC容器初始化时会自动扫描四种类型的组件类型注解

组件类型的注解默认的beanId是默认类名首字母小写，当然也可以进行手动设置

你知道 Spring 容器启动阶段会干什么吗？

容器启动阶段首先加载配置文件，然后利用对象的默认构造方法创建对象，然后注入对象的属性，然后对象执行init-method方法，容器启动完毕

能说一下 Spring Bean 生命周期吗？



将IoC容器中Bean的生命周期就是值在IoC容器，哪个阶段，Bean做了什么事情。

|  |  |  |
| --- | --- | --- |
| Bean生命周期 | | |
| 时间 | 容器行为 | 对象行为 |
| t1 | IoC容器准备初始化，解析XML配置文件 |  |
| t2 |  | 对象实例化(包括字面量和Bean对象)，执行构造方法 |
| t3 | 为对象注入属性 |  |
| t4 |  | 调用init初始化方法,有bean标签inti-method属性指定，在对象注入属性值后，基于这个属性值完成对象的初始化工作 |
| t5 | IoC容器初始化完毕 |  |
| t6 |  | 执行业务代码 |
| t7 | IoC容器准备销毁 |  |
| t8 |  | 调用destroy-method方法，释放资源 |
| t9 | IoC容器销毁完毕 |  |

Bean表进行属性设置的时候如何让程序自动计算属性之间的结果？

在init-method方法中计算，如果是单例模式，该方法只会执行1次

Bean 定义和依赖定义有哪些方式？

xml

注解

JavaConfig

有哪些依赖注入的方法？

依赖注入是指运行时将容器内的对象利用反射赋给其它对象的操作

依赖注入说白了就是将两个对象关联起来

1. 利用setter方法实现对象依赖注入

property标签

* 1. 是静态数值的输入 name value

“123”、“红色”、字符串、包装类

<property name="tittle" value="红富士"/>

首先创建”红富士”String对象里面的value引用指向常量区中的”红富士”，然后将这个对象用setTittle方法赋给tittle属性实现了字符串对象的依赖注入

1.2 对象注入

name ref

首先创建两个不想干的Bean到IoC容器中

然后通过调用set方法将其中一个Bean赋给另一个Bean

1. 利用构造方法实现对象依赖注入

contructor-arg标签

1、name value 静态

2、name ref 动态

Spring IoC如何注入集合对象？

bean标签的子标签property的子标签list可以设置对象

list标签分为value子标签或ref子标签分别对应静态值(字面值)和动态bean对象

和前面对象依赖注入意义 name：字面值，ref：动态bean对象

如何查看Spring IoC容器内有多少个对象？

查看Spring IoC有多少个对象、有哪些类型：

contex.getBeanDefinitionNames 获取IoC容器中所有的对象的名称

1、内部bean标签是不会出现在结果中的，虽然Spring IoC会创建该对象并赋给map的value

2、如果bean标签没有写id或者name，就会用类名名称#0 作为id com.imooc.spring.ioc.entity.Apple#0

3、com.imooc.spring.ioc.entity.Apple#1，如果有多个相同类型的匿名Bean，会在后面编号进行区别

4、如果有多个相同类型的匿名bean，用类名进行获取，会默认返回第一个匿名bean，要想获取后面的，必须在类名后面加#再加编号

通过cotext.getBean(beanName).getClass.getName得到类的名称，得到运行时类型，这是javase的内容

获取内容，执行打印对象即可，如果重写了toString就可以查看内容

Spring 有哪些自动装配的方式？

自动装配：指的是再IoC容器运行的过程中自动地为某个属性注入数据。自动装配注解就是为了依赖注入所存在的

|  |  |  |
| --- | --- | --- |
| 分类 | 注解 | 说明 |
| 按类型装配：不关心beanId是什么，把与属性相同类型的对象注入 | @Autowired | 按容器内对象动态注入属性 |
| @Inject |  |
| 按名称装配 | @Named | 按属性名自动装配属性 |
| @Resource |  |

注：使用xml配置文件中，property标签的name ref是按照bean的名称进行注入

@Autowired注解属性是，变量名必须与容器内的beanId匹配吗？

不需要，因为@Autowired是按照类型装配，即根据要注入的属性的类型，从容器中找到向匹配的类型进行装配，与变量名没有关系。所以不是必须的

@Autowire在IoC容器中找到多个对象与当前属性匹配怎么办，会装配容器内哪个对象注入到该属性中？

当@Autowire在IoC容器中找到多个对象与当前属性匹配时，会抛出一个异常，因为spring也不知道选哪个，与变量名称、beanId名称没有关系。无论怎么修改要注入属性的名称都没有用。此时的解决办法有两个：以实@Primary注解，在找到的多个类中选一个加上该注解，spring会优先注入该类的对象

1. 如果@Autowired 在IoC容器中只找到一个bean与当前要注入的属性类型相同，则与属性名称没有关系，属性名称就可以任意
2. 如果@Autowired 在IoC容器中找到多个bean与当前要注入的属性类型相同，则属性名称必须与对应类的首字母小写，如果对应类注解时没有指定beanId
3. 解决注入属性找到多个符合的对象，使用@Primary注解和@Qualifier注解

解决方式：

1 修改变量名称

2 注解：@Primary @Qualifier

Spring 中的 Bean 的作用域有哪些?

bean scope属性用于决定对象何时被创建，以及作用范围是什么

bean scope决定容器内的bean数量

默认值为 singleton 即单例模式，全局共享一个对象实例

bean scope 属性

|  |  |
| --- | --- |
| bean标签的scope属性 | 说明 |
| singleton(默认设置) | 单例模式，只有一个对象，全局共享 |
| prototype | 多例模式，每次要使用对象时，IoC容器会自动创建一个全新的对象 |

为什么Spring IoC容器会默认使用单例模式？

如果每一次在需要某个对象的时候都去创建一个新的bean，会额外创建内存空间，同时创建对象也会占用cpu资源，由于频繁创建对象所带来的资源损耗必须要考虑在内，而IoC使用单例模式可以有效解决创建对象时占用资源的问题，因为全局只用创建一次就行了。

Spring 中的单例 Bean 会存在线程安全问题吗？

会

要看这个Bean是不是无状态的，如果是无状态的，即没对自身的成员变量以及外界变量、对象、数组进行修改，则是线程安全的，如果依赖于外界的状态，则不是线程安全的。

singleton的线程安全问题：a.setnum(1)🡪a对象num属性为1🡪程序调用🡪sout(num)🡪1

但在这个过程中其它线程修改num的值就会导致线程1看起来结果不正确

这个过程可以通过线程同步方式，用synchronized给同步代码加锁

还有一个解决方法就是给每一个线程创建一个对象，在使用getBean获取对象时，每个线程都会创建一个新的Bean从而解决线程安全问题，prototype在容器中会创建多个实例，每个线程使用到某个Bean就会创建一个新对对象。每次使用IoC容器里面的对象，就会创建一个新对实例，不会产生线程安全问题

singleton和prototype对比

|  |  |  |
| --- | --- | --- |
|  | singleton | prototype |
| 对象数量 | 全局唯一 | 存在多个 |
| 对象实例化 | IoC容器启动时 | 要使用对象时：getBean或者依赖注入时都会产生一个新对象 |
| 线程安全 | 存在 | 不存在 |
| 执行效率 | 高 | 低 |

装配注解@AutoWired在成员变量上和在方法上有什么不同？

如果装配注解@AutoWired放在方法上，则自动按类型或者名称对方法参数进行注入，并自动调用该方法。

如果装配注解@AutoWired放在字段上，SpringIoC容器会通过反射技术将字段的访问修饰符修改成public，然后直接进行赋值，最后在复原，不执行setter方法。

为什么controller service dao层的类会设成单例？

因为这写层的类没有修改自身字段的函数，setter方法，所以这些类对象本身时线程安全的，至于这些对象调用的方法里面用到的对象、内存空间是不是线程安全的与单例还是多例已经无关。如果方法会修改某一块公共的内存地址，则该内存不是线程安全的，与调用该方法的类是不是单例已经没有关系了。所以这几个层的类基本上都是单例模式，根源在于这些类本身的字段不会修改。

service层，dao层，controller为什么采用单例模式，单例对象是线程安全的吗，开发所这些层的类的字段不会提供setter方法进行修改，所以是线程安全的。而且，使用@Autowired注解，Spring IoC容器不会通过setter方法进行属性的注入，而是将要注入的变量的访问修饰符修改为public然后赋值，单例模式在IoC启动的时候只执行一次，后面多线程访问时访问修饰符又变回public，所以是线程安全

@AutoWired可以注解在方法上，会对形参进行注入并执行方法

如何理解无状态？

无状态：不依赖于自身或者外界状态

1. 函数无状态：指函数执行时不会对外界变量、对象、数组进行修改，输入与输出是一对一映射的，每一个线程调用该函数都只会操作线程栈内部的局部变量，而这些变量是线程安全的。所以无状态函数是线程安全的
2. Bean无状态：没有字段或不提供修改字段的方法，bean里面的成员方法都是无状态的，则bea时无状态的，多个线程调用都不会修改Bean，这个对象就是线程安全的
3. Spring 单例模式的Bean如果提供了setter方法，或者bean的有无状态的成员方法，就不是线程安全的，因为在多线程调用时会修改Bean的状态

聊一聊基于Java Config配置IoC容器？

Spring IoC容器的配置方式有三种：基于XML、基于注解、基于Java Config

基于Java Config是在spring 3.0之后提出的一种全新的配置方式，它的主要原理是使用Java代码来替代传统的XML文件：

1. 基于Java Config可以完全摆脱XML的束缚，使用独立Java类管理对象与依赖
2. 注解配置相对分散，利用Java Config可对配置集中管理
3. 可以在编译阶段进行依赖检查，不容易出错

Java Config核心注解

|  |  |
| --- | --- |
| 注解 | 说明 |
| @Configuration | 放在配置类，说明当前类是Java Config配置类，作用：完全替代XML文件 |
| @Bean | 用在方法上，方法返回的对象有Spring IoC容器管理，返回的对象的beanId默认为方法名 |
| @ImportResource | 放在类上，加载静态文件，可使用@Value进行获取 |
| @ComponentScan | 放在类上，与context:componentscan是一样的，用于扫描项目中四种组件类型：@Repository，@Service，@Component，@Controller  mapper有mapperScan |

@Bean注入方法参数跟@Autorwired一样，@Bean也可以和@Primary配合使用，也可以加上Scope等等以前用XML标签完成的事情

@Autowired 的实现原理？

1、@Autowired时按类型进行装配

2、@Autowired放在字段上

3、@Autowired放在方法上

Spring IoC有哪几类注解？

1. 组件类注解
2. 自动装配类注解：按类型、按名称
3. 元数据注解

|  |  |
| --- | --- |
| 元数据注解 | 说明 |
| @Primary | 按类型装配是，如果有多个bean匹配，则有@Primary类的bean会优先注入 |
| @PostConstruct | 相当于xml配置中bean标签的init-method属性 |
| @PreDestroy | 相当于xml配置中bean标签的destroy-method属性 |
| @Scope | 设置对象的scope属性  singleton，prototype |
| @Value | 为属性注入静态数据 |

@Value注解实现原理？

与@Autowired一样，@Value注解会将字段的访问修饰符通过反射机制修改为public，然后直接赋值，并不会调用setter方法，也没有setter方法提供调用。需要在Spring IoC的xml配置文件中引入properties文件，通过property-placeholder引入，然后@Value中写入el表达式形式的字符串，${}里面填上设置的key，就可以得到value

使用Java Config与使用XML配置对比？

|  |  |  |
| --- | --- | --- |
|  | XML | Java Config |
| 配置文件 | xml文件 | Java类 |
| 加载配置文件 | ClassPathXmlApplicationContext | AnnotationConfig  ApplicationContext |
| Junit4 | @ContextConfiguration  value默认属性 | @ContextConfiguration  classes属性 |
|  | bean标签 | bean注解 |
|  | property | set方法 |
|  | ref | 形参 |
|  | bean scope | @Scope |
|  | 扫描组件类注解 | @ComponentScan |
| 加载properties或properies文件 | context:property-placeholder |  |

Java Config可以完全替代XML！

spring IoC阶段注解总结：



说说什么是 AOP？

面对切面编程：在软件运行过程中，在执行前，或者执行后，都可以去额外地增加相应的扩展功能，而这个扩展功能称之为切面

切面有点类似浏览器中的插件

Spring中的可插拔组件技术

根据面对对象的思想：万物皆对象，一个功能也是对象，抽象为功能类，称之为切面类。 切面放在执行前、执行后、或者同时都有（surround）



什么是面向过程编程和面向对象编程？

面向过程编程是一种以过程为中心的编程思想，该思想首先会分析出解决问题所需步骤，然后用函数一步一步把这些步骤实现。面向过程的编程思想是一种基础的顺序思维的方式，它将程序分为数据和操纵数据两部分，其核心问题是数据结构和算法的开发和优化。

面向对象编程思想是按照人认识客观世界的方式，采用基于对象的概念建立模型，模拟客观世界分析设计实现。面对对象编程思想把所有的事务都当成对象，处理问题过程中所属思考的不再是用怎样的数据结构描述问题，而是考虑重现问题中各个对象之间的关系。

说说你平时有用到 AOP 吗？

有

aop 切点的execution?



访问修饰符如果时public，是可以省略的

形参列表那里是写类型名

(..,String)

(..,Str\*)

(Fl\*,In\*,..,String)

(int,float,..)

AOP 五种通知类型？

@Before

@AfterReturning ：执行成功，返回对象后通知

@AfterThrowing：运行出错后通知，与@AfterReturning 互斥

@After： 类似try catch finally中的finally，不管执行成功还是失败，一定会通知的

@Around：自定义通知时间，甚至可以决定目标方法执不执行

说说 JDK 动态代理和 CGLIB 代理 ？

JDK动态代理:

代理类和委托类需要实现同一个接口， 通过反射机制，代理类可以在运行是动态生成，不用像动态代理那样手写代理类。

JDK动态代理需要实现InvocationHandler接口，这是JDK提供的反射类，用于在JDK动态代理中实现功能拓展增强，类似于环绕通知

实现 invoke方法：对目标方法进行增强的部分

方法返回值：Object，目标方法运行后的返回值

方法参数：Object proxy，代理类对象，JDK动态代理自动生成

Method method：目标方法的方法类对象

Object args：目标方法传入的实际参数

throwable： 目标方法抛出的异常

调用method.invoke(target,实参列表)，等价于proceedingJointPoint.proceed()

Proxy是JDK反射包下，newProxyInstance用于动他创建代理类对象

需要传入委托类的类加载器、委托类的接口对象，invocationHandler

JDK反射机制实现动态代理，强制要求委托类实现接口

CGLib实现对类的增强：不要求目标类实现接口

CGLib Code Generated Library 运行时代字节码增强技术

CGLib 会在运行时，生成目标基层继承类字节码的方式进行拓展

本质原理是运行时多态

如果目标类 实现了接口，则Spring底层优先使用JDK动态代理技术来实现动态代理

如果目标类没有实现接口，则Spring会利用CGLib技术，继承的方式实现目标类的拓展

AOP 实现原理？

核心是基于设计模式中的代理模式实现功能的拓展

如果目标类实现了接口：通过JDK动态代理实现功能拓展

如果目标类没有实现接口：通过CGLib第三方组件实现功能拓展

什么是代理模式：指通过代理对象实现对原对象的功能拓展

静态代理：实现Runnable接口的类，用Thread作为代理类，调用start方法

动态代理：

客户类，代理类，委托类

代理类和委托类实现同一个接口

客户通过代理类完成需要的功能





静态和动态的区别在于：代理类是在哪个阶段生成的，如果是编译阶段，则是静态代理，如果是运行阶段则是动态代理。静态代理需要手动地创建代理类，开发起来虽然很简单，但是很麻烦

说说 Spring AOP 和 AspectJ AOP 区别?

AspectJ AOP是Eclise实现的，新语言

Spring AOP 使用AspectJ中的aspectjweaver实现切点的织入

Spring AOP 利用Java语言，代理模式实现

AspectJ 是Eclipse提供的，是一种面向切面编程的语言，AspectJ有一套完整的体系，可以在运行时实现AOP的理念。Spring AOP依赖于AspectJ，但不完全使用AspectJ实现AOP

Spring AOP利用AspectJ AOP实现AspectJWeaver实现类于方法的匹配

在引入Spring-contex之后会自动引入spring aop，但是还无法满足全部要求，还必须引入aspectjweaver依赖，其核心作用就是，圈定切点的作用范围。

作为Spring AOP，并不是完全依赖AspectJ AOP，在运行时对目标方法进行功能拓展时是由Spring AOP来实现的，其核心原理时利用代理模式。

AOP中的概念

|  |  |
| --- | --- |
| Aspect | 具体的可插拔组件功能类，通常一个切面只实现与一个通用方法，一个切面类就是一个标准的Java类 |
| Targe Class/Method | 真正要执行的业务代码，目标方法、类、对象可以通过JointPoint方法，利用反射机制，动态获取。 |
| PointCut | 切点，使用execution表达式说明切面要作用域系统的哪些方法上，说明了切面要作用的范围 |
| JointPoint | 连接点，切面运行过程中包含了目标方法、目标类元数据的对象 |

切点决定了 地点

通知决定了 时间

方法决定了 事情

AOP配置过程

1. 引入依赖 spring-context, aspectjweaver
2. 实现切面类，方法
3. 配置Aspect Bean
4. 定义PointCut
5. 配置Advice

JointPoint连接点有哪些方法？

getTarget: 获取IoC容器中目标类对象

getSignature: 获取目标方法

getArgs: 获取目标方法的参数

使用spring IoC、spring AOP、spring事务的最小依赖？

spring IoC： spring-context

spring AOP： spring-context aspectjweave

spring事务：

为什么Spring Ioc可以降低对象之间的耦合？

首先要理解耦合的概念：在软件工程中，耦合是指一个模块与另一个模块的关联程度，这个模块可以是类或方法、对象。如果一个模块的改变会影响到另一个模块，我们就说这两个模块关联程度较低，也就是高耦合的。反之，如果一个模块的改变很少影响另一个模块，则说明这两个模块关联程度较低，也就是低耦合的。

两个对象之间什么时候会产生耦合呢，就是当两个对象之间是依赖关系是。依赖关系是指一个对象中需要使用另一个对象，具体由三种不同的情况：另一个对象作为局部变量，另一个对象作为方法参数，向另一个对象发送消息。当两个对象由依赖关系时，这两个对象是耦合的，spring ioc通过将容器来管理这种依赖关系，降低耦合读。举个例子，加上对象A需要使用对象B的功能，在没有IoC的时候，需要在A里面创建B的实例，如果B的实现方法发生变化，A也会跟着改变，此时A和B是高耦合的。而如果用IoC实现，只需在A中声明要使用B实例，至于B的实例是如何创建的，A不需要关系，只需要关注自己的业务代码，后面如果B的实现方式改变，也不会影响A的代码，这极大降低了两个对象之间的耦合度。

依赖关系和关联关系的区别？

A依赖B

依赖关系：B的实例对象作为方法参数出现在A中、B的实例对象作为局部变量出现在A中，A中向B发送了消息 、B的实例对象作为返回参数出现在A中

关联关系比依赖关系更强，A关联B，B的实例对象作为A类的成员变量出现在A中

依赖关系是一种临时关系，只有在方法调用的时候会产生，而关联关系是一种长期的关系，主要体现在成员变量，只要对象一声明，这种关系就存在。

关联关系本质上也是依赖，只是强度更强。

什么是Java Bean？

Bean就是Java中的类，是符合一定规范的类。

规范1：字段是私有的

规范2：通过setter getter读写字段

规范3：必须由默认的构造函数

如何使用spring整合的测试框架？

传统程序员管理对象方式有哪些不足？

1. 类属性发生改变时需要修改源代码
2. 如果要增加或减少对象的数量也需要修改源码
3. 对象之间时硬关联，在编译阶段就将对象之间的关系确定了，这是一件非常死板的事情。

修改源码需要走繁琐的审批流程、重新编译上线，这是非常麻烦的。

需要在修改源代码的情况下才能对程序进行调整

所以对象的管理交由第三方管理可以避免上述问题。通过引入IoC容器的方式、对系统中的所有对象进行管理，并且根据一定规则来约定对象是如何创建的以及如何关联的。

Spring服务最小的依赖范围？

spring-context，只有引入了它，才能使用spring最核心的IoC容器，才能使用由这个特性引发出来的其他功能。

引入spring-contex会引入

spirng-aop：面向切面编程模块

spring-beans：对对象继续管理的模块，对象实例化或者设置管理

spring-context：通过ApplicatContex对象可以通过代码对spring IoC容器创建。

spring-core: 所有spring最核心的代码,

spring-expression：spring内置的表达式模块，spring有自己的表达式语言，通过它进行解析和处理。

spring-jcl：这是spring与日志进行交互的模块，在运行过程中产生的底层日志由它处理。

这是引入spring最小的服务范围，这些模块共同完成了Spring IoC的工作。

junit4和junit5的区别？

引入时包不同

注解不同

spring如何使用junit4进行单元测试？

引入spring-test依赖

引入junit4依赖

创建测试类

@Runwith注解 这是junit4的注解

@ContextConfiguration 这是spring的注解，org.springframework.test.context

如何用spring来管理对象的创建？

1. 创建一个maven工程
2. 引入spring-context
3. 创建Java Bean
4. 在resource目录下创建applicationContext文件
5. 写入xml头部内容，去官网复制
6. 通过bean标签声明要创建一个对象

id:对象的名称

class：创建哪个类的对象

bean标签里面由property标签，通过property设置对象的属性

原来创建对象，需要将这些信息硬编码程序源代码中，现在只需要使用配置文件，以后对象属性改变、对象数量改变、对象间关系改变时，只需要修改配置文件，而无需修改源代码。配置文件的修改不用重新编译，审批。极大地缩减了流程。

spring如何启动IoC容器？

在项目入口，即main方法里面需要新建ApplicationContext接口的实现类对象，这个对象就是一个IoC容器。

classpath是什么路径？

是项目编译后，classes路径下面，包顶层所在路径，resource目录下的文件，idea会自动复制一份过去。

springIoC容器的applicationContext文件中，如何关联两个对象？

在bean标签里面的property标签使用ref关联相应的bean id

spring IoC容器在启动过程中根据applicationContex，进行动态实列化以及动态绑定，这是利用java反射技术实现的

为什么要用配置文件来进行对象创建以及关系绑定呢？

这是为了程序上线后可维护性考虑的，采用硬编码的方式，如果对象需要改变源代码，但利用配置文件，不需要调整任何源代码，不需要重新编译即可完成对象修改。

利用IoC容器可以使得对象与对象解耦。

hashset是如何比较对象是否重复的？

首先比较hash值、hash值通过对象的hashcode函数计算，如果hashcode函数不同则不会再去比较equals方法，如果equals相同，不会会认为是重复的key。所以，为了保证hashset或hashmap在使用自定义类作为key时的正确性，要确保hashcode不同时，equlas的结果也是不同的。如果hashcode结果不同，而equals相同，JVM会认为这两个对象不是重复的。所以，在重写是，通常需要对这两个方法都重写：hashcode和equals

utf8和utf8mb4的区别？

utf8mb4是utf8的超集，能够表示更多特殊的符号，如果只表示unicode编码格式下的字符，utf8就够了

什么是事务？

事务是一组对数据库的操作，事务是原子性的，即这一组对数据库的操作要么都成功，要么都失败。事务是以一种可靠的，一致性的方式访问和操作数据库的程序单元。

事务依赖于底层数据库的实现，mysql通过事务区作为数据缓冲地带，当所有操作完成后提交，就可以将事务区中的数据一次性写入实际的表中，如果出现了问题就进行回滚。

Spring 事务的种类？

spring通过Jdbc中的TransactionManager事务管理器实现事务控制

TransactionManager提供commit和rollback实现事务的提交与回滚。

1. 编程式事务（手动式事务）

编程式事务指通过代码手动编程实现提交回滚事务的事务控制方法

1. 声明式事务（注解型事务,自动式事务）

注解的本质是声明。

声明式事务的底层原理就是AOP，它是AOP理念的一个典型应用

1. 当目标方法执行成功时机：自动提交事务
2. 当目标方法遇到运行时异常时：自动回滚事务

声明式事务是利用AOP中的环绕通知解决，完成应用程序拓展，在不修改源代码的前提下实现事务配置

配置TransactionManager事务管理器

配置事务通知与事务属性

为事务绑定PointCut切点，限定事务通知的范围

Spring 的事务传播机制？

事务传播行为：是指多个拥有事务的方法，在嵌套调用时，事务控制方式。

propagation来配置





声明式事务实现原理了解吗？

声明式事务其根本是一个Spring AOP的环绕通知

什么是包装类？

包装类是对Java中八大基本数据类型的一种拓展

装箱：基本数据类型🡪包装类

拆箱：包装类🡪基本数据类型

自动装箱底层调用的是valueOf静态方法

自动拆箱底层调用的是intValue()实例方法

Jdk1.5之前需要手动拆箱与装箱

Jdk1.5及以后就可以自动拆箱与装箱了

Integer integer = 1; 🡺 Integer integer = Integer.valueOf(1);

int i = integer; 🡺 int i = integer.intValue();

包装类和String类都是用常量来存储真正的值

所以包装类和String类的对象都是不可变的

private final int value // 非引用变量，存储值

private final char[] value // 引用变量，存储地址

Object obj1 = true ? Integer.valueOf(1):Double.valueOf(2.0)，打印obj会输出什么？

1.0

首先三元表达式，为true选第一个

其次Integer和Double在一个表达式中，Integer的值会自动提升为Double类型

所以结果不是1，而是1.0

关键：一真大师，三元运算夫看成一个整体，与if else语句不同，后者是分别计算，不会提升优先级。

自动调用对象toString方法的场景？

与字符串运算

打印

String.valueOf()

包装类和String类相互转化有哪些方式？

包装类🡪String类

与字符串做运算 （自动转）

toString方法

打印包装类对象 （自动转）

String.valueOf

这几种方式本质上都在调用对象的toString方法

String类🡪包装类

构造方法

parse方法

new Integer(112)和Integer.valueOf(112)的区别？



new Integer(112)会直接在堆中创建一个Integer对象，该对象有一个私有final修饰的int变量，其值会被赋为112，Ineger.valueOf(112)会先判断112是不是在缓存池范围中，缓存池的范围为-128到127，112在里面，会直接返回缓存池中的一个创建好的Integer对象，而不会再创建一个新的Integer对象。

介绍一下自动装箱和自动拆箱？

装箱：基本数据类型🡪包装类

拆箱：包装类🡪基本数据类型

自动装箱底层调用的是valueOf静态方法

自动拆箱底层调用的是intValue()实例方法

Jdk1.5之前需要手动拆箱与装箱

Jdk1.5及以后就可以自动拆箱与装箱了

Integer integer = 1; 🡺 Integer integer = Integer.valueOf(1);

int i = integer; 🡺 int i = integer.intValue();

当包装类和非包装类运算、比较时，会自动拆箱

Integer创建对象的几种方式和区别？

new

Integer.valueOf()

自动装箱

int与Integer的区别？

int是基本数据类型，Integer是包装类类型，是int类型的拓展

Integer类型量必须实例化后才能使用，int类型变量不需要

Intger类型变量默认值为null，int类型默认值为0

Integer类型的变量是引用变量，存储的是对象的地址值，执行堆中的一个Integer对象，int类型变量是非引用变量，存储的是实际数值

1. int是java中的基本数据类型，Integer是int的包装类，是对int类型的拓展
2. Integer类型变量必须实例化之后才能使用，int类型变量不需要
3. Integer类型变量默认值为null，int类型变量默认值为0
4. Integer类型变量是一个引用变量，存储的是地址值，指向堆中一个Integer对象，而int变量是非引用变量，存储的是实际的数值

Integer.valueOf() 与new Integer()的结果一定不是同一个对象吗？

是的，首先new Intege()的结果一定不在缓存池中。然后两new Integer()的对象一定不是同一个对象。

而Integer.valueOf(xxx)，当xxx属于[-128,127]时，Integer.valueOf(xxx)的对象再缓存池中，满足第一种情况，不是同一个对象。当xxx不在这个范围时，会new Integer(xxx)，满足第二种情况，不是同一个对象。

综上所述，Integer.valueOf() 与new Integer()的结果一定不是同一个对象

Integer i = 100; i++, 画出内存执行图?



i是Integer类型

i++ 🡺 i = i + 1 🡺 i = 拆箱 + 1 🡺 i = 101 🡺 装箱

如果i是int类型

i = 100;

i++ 🡺 i = i + 1 🡺

1. i是int但i是局部变量，i++是线程安全的。
2. i是成员变量，i在堆中，i++不是线程安全的。

i++不是原子操作 + i不是局部变量🡺线程不安全

为什么说Java是平台无关性语言？

Java中有一句话，Compile once，Run anywhere，这是如何实现的呢：

java分为编译时和运行时两个阶段

编译时：javac 指令，全称时java compile，即将源码.java文件编译成字节码，并存入对应的.class文件中。class文件就是java文件翻译成的二进制字节码文件，java文件中类的属性、方法、常量信息等会被存储在.class文件中，除了这些，还会添加一个共有的静态常量属性，Class类的示例，这个属性记录了类的相关信息以及类型信息。真正加载class文件用到虚指令，通过javap -c xxx xxx是class文件名，会出现一些指令。

JVM将字节码加载进内存，并将字节码转换为本操作系统可识别的机器码去执行。将对应的class文件拷贝到其他装有相同版本的jdk或这jre的操作系统，也可以执行。

java源码首先被编译成.class文件，再由不同平台的JVM解析。Java在不同平台运行时不需要重新编译。JVM在执行字节码文件时，会将字节码转换成对应平台的机器码

Java源文件首先被编译成class文件，再由不同平台的JVM解析，JVM解析时将字节码文件加载进内存，并将其转换为对应操作系统的机器码，所以java只需要编译一次，就可以在不同平台上运行，这也就是为什么java是平台无关性语言。

运行时：JVM将.class文件加载进内存，将其转换为对应操作系统的机器码。

JVM这一块主要有哪些点要掌握？

1、类加载

2、运行时内存模型

3、垃圾回收

我们写的Java代码到底是如何运行起来的？

jar包和war包有什么区别？

jar全称是java archive，它是基于zip文件格式的归档文件，用于将多个class文件整合成一个文件

war的全称是web application archive，是用于分发和部署web应用程序的jar文件格式，把war包放在tomcat的webapp目录下，tomcat服务器在启动时会解压war包，将class文件加载进内存。

jar包中通常只包含class文件和一些部署文件，但war包里面就很全了，包括class文件、配置文件、静态文件…

对于编译好的这些“.class”字节码，是怎么让他们运行起来的呢？



每运行一个java程序(可以是一个文件、一个项目…)，操作系统就会启动一个java.exe的进程，这个进程就是JVM，负责加载对应的class文件，JVM会根据本机操作系统的类型，将class文件翻译为对应的机器码。所以启动一个java程序或者java项目，本质上是启动了一个JVM，再本质上是启动了一个名为java.exe的进程，这个进程负责运行这台机器上运行的这个系统。



所以一个JVM==一个系统进程

我们用java编写的应用程序启动并不是该程序进程启动了，而是启动了一个JVM进程，然后改JVM进程将应用程序加载进内存，由该JVM负责这个应用程序的运行，内存分布、垃圾回收等等各种

JVM运行机制是怎么样的？

1、.java文件编译为.class文件

2、启动一个java.exe进程，或者JVM进程

3、将class文件加载到JVM内存中，这个加载是懒加载

4、执行类中的代码，需要用到哪个类再加载哪个类

为什么会有类加载器这个东西？

class文件🡪内存

一个java文件中，有多少个外部类，就有多少个class文件。所以我们可以这样说，一个class文件就代表一个类。

1. 一个class文件代表一个类
2. 把class文件加载进内存==把类加载进内存

而把类加载进内存有一个专门的模块，叫做类加载器



所以什么叫做类加载：把class文件加载进内存就叫做类加载

什么叫类加载器：负责类加载的模块叫做类加载器

这个加载的过程时懒加载，不会一股脑的把我们编写好的系统中的所有class文件全部加载，而是需要哪个就加载哪个。

JVM是个什么东西？

JVM本质上是操作系统中一个名为java.exe的进程，会在操作系统中使用java命令运行一个class文件或者jar包时启动。

JVM跟我们平时运行在机器上的系统之间是什么关系？

我们编写的，平时运行在机器上的系统本质上不是这个系统本身启动了一个进程，而是启动了一个JVM，由JVM管理该系统的运行。

类加载器的概念?

类加载器：JVM负责类加载的模块

类加载：将class文件加载进内存

class文件本质就是一个类，这就是类加载名字的来由。

针对加载进内存的类进行代码的执行？

类加载器采用的是懒加载模式，即需要用到哪个类才会去加载对应的类，执行到哪一步，遇到要使用某个类，才回去加载。

如何保证class文件的安全，不被反编译回源文件？

1、加密：在编译阶段可以通过一些工具给字节码文件加密

2、解密：在类加载的时候采用自定义的类加载器来解密字节码文件

这样就可以保证源码不被窃取。

JVM在什么情况下会加载一个类？

需要用到一个类的时候:

1. 包含main方法的主类，一定会在JVM进程启动后加载到内存，然后从main方法里面开始执行代码。

其他情况：

1. 调用某个类的静态属性、静态方法
2. 构造某个类的对象
3. 声明某个类的变量

为什么class文件经过编译阶段已经检查过了符合java语法规范，在类加载时还需要验证阶段？

因为编译后的class文件有被篡改的风险，从而不符合JVM规范，如果没经过验证直接给JVM运行可能会直接出错。

初始化语句在类加载准备阶段会分配内存空间吗？

根据调试的结果，初始化语句会在类加载准备阶段分配内存，在类加载初始化阶段，执行初始化语句直接初始化，无需再分配内存。

类加载准备阶段：所有的类变量，不管有没有初始化语句，只要声明了，就根据类型分配内存空间，然后进入类加载解析阶段，将字符引用改为直接引用，然后进入类加载初始化阶段，执行初始化语句。



实例化过程中实例变量也是遵循一样的规则，分为准备阶段和初始化阶段

父类加载准备->父类加载初始化->子类加载准备->子类加载初始化->父类实例化准备->父类实例化初始化->父构->子类实例化准备->子类实例化初始化->子构

java中有类加载器？

类加载：将class加载到内存中

类加载器：JVM中负责类加载的模块，JVM本身就是一个java程序。类加载器就是负责类加载的类

java中负责类加载的类：

|  |  |  |
| --- | --- | --- |
| 类型 | 英文名称 | 说明 |
| 启动类加载器 | BootStrap ClassLoader | 负责将Java按照目录下的核心类加载到内存中。一旦JVM启动，这些类就会自动加载进内存，保证运行我们编写的系统所需的依赖 |
| 扩展类加载器 | Extension ClassLoader | lib\ext,这里面一些类也是用来支持系统运行，JVM启动时也会自动加载 |
| 应用程序类加载器 | Application Classloader | 这个类加载器负责去加载ClassPath环境变量所指定的路径中的类。可以通过-classpath或-cp指定路径 |

什么是双亲委派机制？

1. JVM的类加载器有亲子结构
2. 双亲委派机制基于这个亲子结构

一句话说明：先找父亲加载，不行再有儿子加载。

应用程序类加载器要想加载一个类会先去让扩展类加载

扩展加载之前会先去找启动类加载器加载，如果启动类加载器没有找该类，则会让扩展类加载器加载，如果扩展类加载器没找到该类，则会下放给应用程序类加载器加载。



Tomcat的类加载机制应该怎么设计，才能把我们动态部署进去的war包中的类，加载到Tomcat自身运行的JVM中，然后去执行哪些我们写好的代码？

Tomcat本身就是Java写的，一个运行的tomcat本质上就是一个JVM进程

Tomcat可以自定义一个类加载器专门负责webapp目录下类加载，这样通过双亲委派机制，父类加载器在各自作用范围均找不到类时就会下放给自定义类加载器进行加载。

JVM中有哪些内存区域，分别都是用来干嘛的？

什么是JVM内存区域划分？

使用java命令运行一个java程序或者项目时，系统会启动一个名为java.exe的进程，这个进程也叫JVM进程。JVM会分配一块独立内存区域，别的进程无法访问。对于这个块内存区域的划分、使用由JVM负责

1. 需要使用到某个类时，会把该class字节码文件加载进内存成为一个类，这个类需要有地方存储
2. 调用类的方法时，方法有一个方法栈存储局部变量
3. 实例化对象时，成员变量需要有地方存储

为什么要程序计数器？

.java文件编译称.class文件，这个class文件是字节码文件，这个字节码文件里面是一条条字节码指令，对应一条条机器指令。计算机只有读到这种机器指令，才知道要干什么。字节码指令通过一个叫字节码引擎的东西来执行，在执行一条条指令过程中，字节码引擎需要知道指令执行到哪一条了，程序计数器就是起到这个作用，帮助字节码引擎记录目前程序执行到那一条字节码指令了。

什么是内存?

1. 内存是一个硬件
2. cpu有cache可以存储数据
3. 速度：cache>内存>硬盘

容量: cache<内存<硬盘

1. 内存用于暂时存储CPU的数据，以及与硬盘等外部存储器或者设备交换的数据。
2. 内存是CPU能直接寻址的存储空间
3. 计算机中所有程序都运行在内存中
4. 操作系统本质是一个程序（软件），也会在内存中运行，随着计算机的启动，操作系统首先加载进内存，占用一定的内存空间，操作系统占用的内存空间是保护起来的，不允许别的进程访问

方法区中存储的类信息是什么？

1. 类变量
2. 描述类的元数据



描述这段程序JVM运行过程？

首先Kafa.java通过javac命令编译成class文件，会得到两个class文件，Kafa.class和ReplicaManager.class。

使用java命令运行系统，操作系统会在后台启动一个java.exe的进程，并为其分配一定的内存，这个进程就是JVM，JVM负责我们编写的系统的运行。

JVM首先查找main方法程序入口，发现它在Kafa类中，于是类加载器会首先加载Kafka类，然后启动一个main线程，为这个main线程分配一个mian线程栈，在该线程中调用main方法，该线程会在线程栈中为main方法分配一个栈帧，保存main方法信息和局部变量。同时JVM字节码引擎会将字节码指令一条条指向，JVM会为main线程分配一个程序计数器，用于记录本线程的字节码指令执行到了那一条。开始执行main方法里面的代码，发现需要使用到ReplicaManager类，于是类加载器将ReplicaManager通过同样的验证、准备、解析、初始化阶段加载进内存，在main线程栈的main方法栈帧中会压入一个局部变量replicaManager，该变量是一个引用变量，存储的是地址，指向堆中的new出来的ReplicaManager对象。

该引用变量调用loadReplicaFromDisk方法，会在main线程栈中为该方法分配栈帧，栈帧中压入 replicCount局部变量，当该方法运行结束后，会出栈。

我们在Java堆内存中分配的那些对象，到底会占用多少内存？一般怎么来计算和估算我们系统创建对象内存占用的一个压力？

一个对象占用的空间=对象本身的信息占用空间+对象的实例变量占用空间

什么是垃圾回收机制？

JVM进程自身由垃圾回收机制，它是一个后台自动运行的线程，只要启动一个JVM，就会自带一个垃圾回收的线程。这个线程会在后台不断检查Java堆内存中的各种实例对象，是否有人引用，如果一个对象没有任何一个局部变量指向它，也没有任何静态变量，包括常量等指向它，那么这个垃圾回收线程，就会把这个没人指向的对象给回收掉，从内存中清楚掉，让他不占用任何资源。

垃圾就是指Java堆中没有任何变量指向的对象。

垃圾回收机制：垃圾回收线程会在后台定期回收垃圾，不断释放内存资源

为什么需要垃圾回收机制？

当某个方法执行完退出栈帧后，里面局部变量也会出栈，而局部变量执行的堆内存中的对象不会自己消失，这些对象没有任何局部变量、静态变量、常量引用，是不可操作的对象，被称为垃圾。如果没有垃圾回收对象，这些不能被操作的对象会 越来越多，而内存资源是有限的，最终导致堆区堆满垃圾对象，而无法在new对象。垃圾回收机制就应运而生，它是JVM自带的一个线程，随着JVM启动自动在后台启动，他会的查看堆中的对象是否由变量引用，如果没有就视为垃圾，然后定期的进行清理，以保证内存资源合理的利用。

Java堆区中的对象会被垃圾回收，那么加载到方法区中的类会被垃圾回收机制回收吗，什么时候回收呢？

会回收。满足以下三个条件时：

1. 该类的所有实例化对象都已经从Java堆中回收
2. 该类的类加载器已经被回收
3. 该类的class对象没有任何变量引用

初始化子类，父类会初始化吗？

父类会初始化，但此时父类和子类都是初始化同一快内存区域，所以初始化两次（或多次），只会产生一个对象。

为什么要双亲委派机制？

双亲委派机制可以防止类加载器重复加载类。

Tomcat支持动态部署和静态部署，阐述这两者的区别？

动态部署，又称热部署，指在Tomcat运行时部署或更新应用。

静态部署，指在Tomcat启动时加载和部署应用。

如果配置的热部署，则直接将war包放入webapp目录下，tomcat会自动检测到新的系统，自动部署或更新应用，无需重新启动tomcat。

如果配置的时静态部署，更新或部署新的项目时，需要停止tomcat容器，将新系统放在webapp目录下之后再启动容器。

热部署：运行时，无需重启tomcat

静态部署：启动时，需要重启tomcat

什么叫内存泄漏？

内存泄漏：程序没有释放已经不再使用的内存。

内存泄漏常发生在创建一个对象，不再使用这个对象后却没有删除它。

之所以叫泄漏，可以理解为对于程序员来说，随着没有释放的垃圾内存越来越多，好像可使用的内存在减少，就像泄漏一样。

内存泄漏会导致内存溢出问题

类加载器有三层，如果在第二层的类加载器可以加载这些类的话，就没有必要往上去找他的父类加载吗？

不对，第二层如果不是顶层类加载器的话，不管自己能不能加载，都会让父级类加载器先去加载，知道传递到顶层类加载器，如果顶层类加载器能够加载，就直接加载，如果不能够加载就下放，重复这个过程。

什么是年轻代、老年代、永久代？

JVM内存的一个分代模型

在我们代码中创建的对象，大部分都是存货周期很短的，随着方法调用的结束，该对象没有任何变量指向，成为垃圾，被回收。

有存活很短的对象：局部变量指向

也要存活很久的对象：静态变量指向

由于编写代码方式不同，创建和使用对象的方式不同，不同对象有不同的生命周期。所以JVM将堆内存区域划分成了两块区域：老年代，年轻代

老年代：长期存在的对象

年轻代：很快要被收回的对象

永久代：方法区，存放类信息

首先，大部分新创建的对象都在年轻代，经过一段时间的运行，一些对象没有任何变量引用，变成垃圾，而少部分对象任然被引用，这通常是被静态变量引用，于是JVM会将这部分对象从年轻代移动至老年代，并对年轻代的对象做一次清理。

为什么要区分老年代和年轻代呢？

这跟垃圾回收机制有关，因为对于不同生命周期的对象有不同的垃圾回收算法，所以通过区域来管理对象，方便算法的执行。年轻代和老年代分别进行垃圾回收。

对象什么时候会进入年轻代，什么时候会进入老年代？

1、大部分的正常对象，都是优先在年轻代分配内存

垃圾回收机制触发条件？

当年轻代里面的对象没有任何引用时，并不会立马进行清理，当年轻代堆积了大量的实例对象，然后程序员想在new一个对象时，JVM发现年轻代里面的内存不够了，此时就会让垃圾回收线程做一次垃圾回收，这个对年轻代里面的垃圾回收叫做Young GC或者Minor GC。它会尝试把新生代中没有人引用的对象给回收掉。我们在代码中创建的对象大多时生命周期很短的对象，大量对象创建引发新生代内存空间不足，然后垃圾回收机制进行Young GC，在这个过程中，长期存活的对象会躲过多次垃圾回收。

随着系统的启动运行，不停在创建对象，年轻代很快变满，然后再垃圾回收一次，大量对象被回收。但一些长期存活的对象在多次垃圾回收中都没有被回收掉，因为有变量一直在引用，需要一直使用。所以JVM规定，如果一个实例对象在新生代，成功地在15次垃圾回收之后，还没有被回收掉，就说明这个对象已经15岁了。

对象的年龄怎么定义？

如果每垃圾回收一次，这个对象没有被回收掉，年龄就会加1

年轻代和老年代分别进行垃圾回收是如何实现的？

Java虚拟机栈要进行垃圾回收吗？

不会、JVM垃圾回收机制时针对年轻代、老年代、永久代。方法一旦执行完毕，就会栈帧出栈，局部变量就直接从内存中清理出去了

JVM 内存分配核心参数?

|  |  |
| --- | --- |
| 参数 | 说明 |
| -Xms | Java堆内存大小；这是JVM启动时堆内存的大小 |
| -Xmx | Java堆内存最大大小；随着程序的运行堆内存可能不够，这里设置了允许扩大的最大内存大小。 |
| -Xmn | 年轻代内存大小;  堆内存=年轻代内存+老年代内存  老年代内存 = 堆内存 – 年轻代内存 |
| -XX:PermSize | 永久代内存大小 |
| -XX:MaxPermSize | 永久代最大大小 |
| -Xss | 每个线程栈内存大小 |

你的对象在JVM中是如何分配，如何流转的？

平时在代码中创建的对象分为两种：

1. 短期存活的对象
2. 长期存活的对象

老年代会垃圾回收吗？

会

从JVM角度，聊一聊每日百万交易支付系统的压力？

每天会在JVM中创建和销毁上百万个支付订单对象

这里牵扯出来一些问题：

1. 我们支付系统需要部署多少台机器
2. 每台机器多大的内存空间
3. 操作系统需要给JVM进程分配多大的内存空间
4. JVM需要给堆内存分配多大的空间（最核心的参数）
5. 支付系统每秒需要处理多少笔订单

假设每天100万个订单，交易行为一般集中发生在高峰期，假设高峰期大概是几个小时，假设3个小时，1000000 / (3 \* 60 \* 60) = 92.59; 每秒大约100个订单；假设我们部署了三台机器，每台机器每秒大概处理30个订单

1. 每个订单要耗时多久：

假设一个支付请求的处理，包含一个支付订单的创建，大概需要1秒钟的时间



1. 每个支付订单大概需要多大的内存空间：

假设一个订单对象占据500字节内存空间 500b

1. 每秒发起的支付请求对内存的占用：

30 \* 500 = 15000b = 15kb

1. 让支付系统运行起来分析：

每1秒15kb内存

5个小时之后就有300多M的空间，这只是算上了支付订单对象的内存。如果算上其他对象，很快堆内存中的年轻代就占满了。然后触发Minor GC，腾出内存空间，继续在内存中分配对象，这就是业务的系统的运行模型。

10、对完整的支付系统占用内存预估：

按照之前支付订单对象的预估的结果扩大10~20倍，也就是说，每秒钟处理在系统中创建支付订单对象，还会创建数十种其他对象。每秒钟创建除了来被栈内存局部变量引用的对象大致占内存的几百KB到1M

假设按1M计算，每秒进来的请求大概会创建1M对象在新生代内存中，然后变成垃圾，下一秒又创建1M对象。

11、如何设置JVM堆内存大小：

线上业务系统常见的机器配置是2核4G

JVM进程最多2G内存

2G = 方法区 + 栈 + 堆

堆最多可能就是1G内存空间

1G = 新生代 + 老年代

新生代500M，老年代500M

如果新生代只有500M，系统每一秒就要创建1M对象

500秒也就是8分钟左右，新生代内存就满了

没8分钟左右就会导致一次Minor GC，如此频繁的Minor GC会影响系统的性能

反面的情况：

假设每秒处理1000笔交易，只有1台机器，那么每秒占用的堆内存就在10M以上，新生代内存为500M，且进来的1000条请求1秒可能处理不完

Tomcat、Spring boot启动项目时输入设置JVM参数？

Spring boot项目在启动时就可以设置JVM参数，Tomcat在bin目录下catalina.sh加入JVM参数

什么情况下JVM内存中的对象会被垃圾回收？

1. 新创建的对象大多在新生代
2. 当新生代里的对象越来越多，占满新生代内存时就会触发垃圾回收机制，对没有变量引用的对象进行回收

JVM如何判断一个对象是不是可以回收的？

JVM使用一种可达性分析算法来判断一个对象是不是可以回收。对每个对象，分析一下有谁在引用它，然后向上层层判断，看看是否有一个GC roots，如果有，就不回收。

可以视作GC roots：局部变量引用、静态变量引用

Java中对象不同的引用类型？

|  |  |  |
| --- | --- | --- |
| 类型 | 说明 | 发生垃圾回收时 |
| 强引用 | A a1 = new A(); // 局部变量  static A a2 = new A(); // 类变量  A a3 = new A(); // 成员变量 | 绝对不回收 |
| 软引用 | SoftReference<String> softReference = new SoftReference<String>("abc");  SoftReference是Java.lang.ref包的类 | 可回收，可不回收 |
| 弱引用 | WeakReference<String> weakReference = new WeakReference<>("abc"); | 绝对要回收 |

JVM中有哪些垃圾回收算法，每个算法的优劣？

对新生代进行垃圾回收时，具体地算法是怎么执行的？

针对新生代的垃圾回收算法叫做复制算法。

所谓复制算法，就是将年轻代内存区域划分成为两块，然后只使用其中一块创建对象，当那块内存要满时，通过可达性分析算法计算出哪些对象是不会回收的对象，然后将这些对象复制到另一块区域，并以一种较为紧凑的方式排列，避免产生过多的内存碎片，然后另一快就复制新对象的创建。本快内存只剩下需要进行垃圾回收的无人引用的垃圾对象，垃圾回收线程对这些对象进行回收，然后空出一块内存。重复上述步骤。这两块内存循环使用。

复制算法的优缺点？

优点：降低内存碎片，减少内存浪费

缺点：如果新生代是1G内存，1G = 512 M + 512M；那么复制创建对象的内存区域只有512M，另外512M要一直空在那里，然后创建对象的区域满了，移动到另一块区域。从始至终，只有一块内存区域在使用，降低了内存的使用率。

为什么要用复制算法，而不是检测一个然后处理一个？

如果才用后者，会让存活的对象东一个、西一个，造成很多内存碎片，这些内存碎片加起来的内存可能很充裕，但是无法找到单独的一块碎片来创建对象，从而造成内存的浪费。

Eden区和Survivor区算法的思想？

该算法任务我们代码中创建的对象，绝大多数都是存活周期非常短的对象，可能创建出来1ms就没人引用了。可能一次垃圾回收，99%的对象都被回收了，之存活下来1%的对象，这1% = 长期存活的对象 + 短期存活但没有使用完的对象。所以长期存活的对象肯0.5%都不到。

所以该算法提出了将新生代内存划分成3块区域 1块Eden区占80%，两块Survivor区分别占10%。对象在Eden区里面分配，当Eden区占满是，进行一次Minor GC，将存活的对象先移动到一块Survivor区，然后再进行垃圾回收，此时新生代内存中两块区域都用上了，一个是Eden区，负责再创建新对象，一个是一块Survivor区，负责存放上一次Minor GC存活下来的对象。新生代内存的利用率为90%。再一次，当Eden区的对象满了的时候会触发Minor GC，此时Survivor区中也可能由垃圾对象，于是，JVM会将这两个区里面存活的对象移动到另一块空白的Survivor区，然后将垃圾对象回收。

Eden区和Survivor区有哪些万一的情况？

万一垃圾回收之后，存活的对象超过10%怎么办

万一遇到一个超大的对象，新生代找不到连续的空间来存放

新生代中的对象需要在这个三块内存区域倒腾多少次才回去老年代？

为什么要有两块Survivor区？

因为存活对象长期存活对象，如果只有一块Survivor区，里面的垃圾对象进行垃圾回收时，就只能一个一个检测在清理，这样就回到了原始时代，会产生很多内存碎片。

在分配对象时，Survivor区会参与吗？

不会

年轻代和老年代分别适合什么样的垃圾回收算法？

新生代中的对象多久进入老年代？

JVM启动时有一个参数可以设置 -XX:MaxTenuringThreshold，默认是15

意思是躲过15次GC就可以进入老年代了。

除了这个规则，还有一个动态年龄判断，也许没有15岁就进入了老年代：

假设Survivor区存在一批对象S，且S满足：S.size > Survivor.size \* 50%

那么此时，某些对象A，满足：A.age S.age； 则A将移入老年代。这就是动态年龄判断规则

S = {年龄1，年龄2，…, 年龄n}，S中有多个年龄的对象，S.age会取最大的年龄n作为比较的年龄





除了15次、动态年龄判断、还有一个规则：大对象直接进入老年代

JVM可以设置参数 -XX:PretenureSizeThreshold

为什么要设置动态年龄判断呢？

我的思考是这样符合人类社会的尝试，什么叫老人，比大多数老的就叫老人。

为什么要设置大对象直接进入老年代？

放在大对象在新生代的三块区域倒腾过去，倒腾过去倒腾过来，复制大对象很麻烦，耗时间，所以直接然大对象去老年代。

这也基于了一个假设，既然声明了一个大对象，就不会轻易放弃它，会用这个大对象做很多事。

是先GC再移动存活对象吗？

是的。

Minor GC之后存活的对象超过10%怎么办？

比如某次GC之后存活了150M，超过了Survivor区100M的内存大小，就会把这150M对象全部加入老年代。

Minor GC之前为什么先询问老年代剩余内存？

因为Minor GC之后可能有新生代的对象进入老年代，而这些对象的大小可能大于老年对象。之所以在Minor GC之前就询问，是因为如果在Minor GC之后询问，这部分对象可能无处可去了。

为什么比较老年代剩余内存和新生代对象总额?

因为新生代Minor GC之后最极端的情况是一个垃圾对象都没回收，所有对象都存活了，如果老年代剩余内存还大于这个值，就说明新生代可以放心大胆的进行Minor GC，因为总不会超过Minor GC钱所有对象大小总和。

新生代和老年代垃圾回收触发时机：

新生代垃圾回收触发：

1.2 Eden区Survivor区满了，

1.2老年代剩余内存>新生代所以对象总和

2、

2.1 Eden区Survivor区满了，但老年代剩余内存<新生代所以对象总和

2.2老年代剩余内存<新生代所以对象总和

2.3 XX:handlePromotionFailure参数设置了

2.4 老年代剩余内存>之前每次Minor GC进入老年代的对象平均大小

3、

3.1 Eden区Survivor区满了，但老年代剩余内存<新生代所以对象总和

3.2老年代剩余内存<新生代所以对象总和

3.3 XX:handlePromotionFailure参数没有设置

3.4 执行Full GC后会冒险执行一次Minor GC

老年代垃圾回收触发机制：

1、Minor GC之前

1.1 Eden区Survivor区满了，但老年代剩余内存<新生代所以对象总和

1.2 XX:handlePromotionFailure参数没有置

2、Minor GC之后

2.1 冒险进行Minor GC之后，存活对象>老年代剩余内存

OOM触发条件：

Full GC之后，Minor GC之后，剩余对象还是无法进入老年代

Minor GC触发时机：

1. 放心Minor GC，1条路径
2. 冒险Minor GC，2条路径

Full GC触发时机：

3条路径

Minor GC前，2条路径

Minor GC后，1条路径

OOM：

1条路径

JVM优化的本质是什么？

尽可能的让对象在新生代里进行分配和回收，尽量少的对象进入老年代，避免老年代垃圾回收。

同时给系统充足的内存大小，避免新生代频繁的垃圾回收

到底什么时候会尝试触发Minor GC？

1. 老年代剩余内存大于新生代所有对象总大小
2. 设置了-XX:HandlePromotionFailure参数，老年代剩余内存大于之前每次进入老年代对象平均内存大小
3. Full GC之后

触发Minor GC之前会如何检查老年代大小，涉及哪几个步骤和条件？

什么时候在Minor GC之前就会提前触发一次Full GC？

老年代内存<新生代对象总大小

1. 没有设置-XX:HandlePromotionFailure参数
2. 设置了-XX:HandlePromotionFailure参数，老年代剩余内存<之前每次进入老年代对象的平均大小

Full GC的算法是什么？

标记整理发

标记

整理

清楚

Minor GC过后可能对应哪几种情况？

1. Survivor区能够放下存活对象
2. Survivor区不能够放下存活对象，但老年区能够放下存活对象
3. Survivor区不能够放下存活对象，且老年区也不能够放下存活对象

哪些情况下Minor GC后的对象会进入老年代？

Minor GC后Survivor区不能够放下存活对象，但老年区能够放下存活对象

JVM参数总结？

-Xms: 堆内存区域大小

-Xmx: 堆内存最大大小

-Xmn: 新生代大小

-XPermSize: 永久代大小

-XMaxPermSize: 永久代最大大小

-Xss: 线程栈大小

-XX:HandlePromotionFailure

老年代垃圾回收算法？

标记整理法

1. 标记存活对象（方便整理）
2. 整理存活对象（避免内存碎片）
3. 清理垃圾对象

什么情况下，新生代对象会进入老年代？

1. 年龄>15
2. 动态年龄判断
3. 超大对象
4. 极端情况

一个日处理上亿的计算系统多久会填满新生代？

每分钟执行100次数据计算任务

每次1万条数据，需要计算10秒中

100 \* 10 = 1000秒

假设每条数据在1KB左右

一万条数据一万KB，10M

机器是4核8G的配置，给JVM进程分配了4G内存，堆内存分配了3G，新生代和老年代分别是1.5G；新生代中Eden区是1.2G, Survivor 1和Survivor 2 分别是100M左右

每执行一个计算任务，就会在Eden区创建10M对象

一分钟之后，执行了100次任务，创建了1G对象，Eden区几乎满了

所以，新生代里Eden区，基本1分钟左右就满了

那么触发Minor GC后有多少对象会进入老年代：

此时假设新生代Eden区在1分钟之后满了，然后接着执行计算任务，比如会导致Minor GC回收一部分垃圾对象，在真正执行Minor GC前会检测老年代剩余内存大小是否大于新生代全部对象大小。此时老年代是空的，剩余内存是1.5G，新生代Eden区大概有1.2G对象，1.5 > 1.2，新生代是可以放心进行Minor GC的。

从此时，倒推10s之内的创建的对象还是处于使用状态，因为一个计算任务要10s才能完成，假设这10s有20个计算任务还没完成，一个计算任务有10M对象；10M \* 20个 = 200M对象存活。

此时，有1G对象要被回收，200M对象存活。

这200M对象会进入老年代。

也就是说，没过1分钟，老年代就会产生200M对象

每7、8分钟就会执行一次full GC

出现这么频繁的Full GC根本原因时每次Minor GC后的200M内存，Survivor区内存放不下， 所以跳过Survivor区的内存就可以缓解。如这样进行分配：

JVM进程有4G内存 = 堆3G内存 + 其他； 堆3G = 2G新生代+1G老年代；

1.6G新生代内存= 1.6G Eden区 + 200M S0区 + 200M S1区；

这样每次Minor GC之后的200M内存，S区正好可以放下。此时

通过一个小小的JVM参数的改动，成功将Full GC的次数从7、8分钟一次，降低到几个小时一次。

根据动态年龄判断规则，Survivor区中的同龄对象大小超过Survivor区内存的一半会全部进入老年代，这句话对不对？

S = {s1,s2,…,sn},这些对象同年龄的

S.size > Survivor.size \* 50 %

任意对象x，x.ageS.age，x会被放入老年代，而S中的对象全部满足这一条，所以这一批对象全部要放入老年代中。

Survivor区中同龄对象大小超过Survivor区的一半会全部进入老年代。

50 %这个参数可以通过 -XX:SurvivorRatio条件例如-XX:SurvivorRatio=8，就表示要超过Survivor区大小的80%才动态判断。或者通过降低Eden区的大小从而提高Survivor区大小来实现

JVM中都有哪些常见的垃圾回收器，各自特点是什么？

垃圾回收线程🡪垃圾回收器🡪垃圾回收算法

Serial，针对新生代，单线程回收

ParNew，针对新生代，多线程回收

CMS，针对老年代，多线程回收

什么是JVM的”Stop The World”？

在垃圾回收的时候，为了尽可能让垃圾回收器专心致志地工作，JVM不会让Java系统继续创建对象。此时JVM会在后台直接进入 Stop The World状态。JVM会停止我们所有写的Java系统所有工作线程栈，让我们的代码不再运行。

S区参与分配对象吗？

不参与

复制算法的过程？

1. 标记存活（Eden区和Survivor0区）
2. 转移存活，转移到Survivor1区
3. 一次性清理垃圾（Eden区和Survivor0区）

GC是需要花费时间的，在开始GC，到结束GC这段时间可以创建对象吗？

不可以。

多线程垃圾回收器一定比单线程垃圾回收器好吗？

如果程序部署在客户端机，而客户端机是一个单核CPU，如果还使用多线程垃圾回收器，反而会加重CPU的负担。

聊一聊，新生代垃圾回收器ParNew是如何工作的？

1. ParNew是多线程
2. ParNew使用复制算法，标记、转移、清理

为什么要用ParNew呢：因为现在我们部署的服务器大多是多核CPU的，为了在垃圾回收的时候充分利用CPU资源，一旦我们使用了ParNew垃圾回收器，它会检测部署系统CPU的核数，然后将要启动的垃圾回收线程与CPU核数设置相同。-XX:+USerParNewGC参数可以设置使用ParNew垃圾回收器。ParNew的垃圾回收线程数一般不用手动设置，就默认和CPU核数一样可以充分并行运行。当然，也可以通过-XX:ParaleeGCThreads参数来设置。

ew+cms的gc，如何保证只做ygc，jvm参数如何配置？

要想避免Full GC，可以从两个方面解决，第一是避免新生代对象进入老年代。第二点是避免新生代对象发生Minor GC前避免触发Full GC。针对第一点，新生代进入老年代有几个时机。躲过多次垃圾回收的对象会进入老年代，默认为15次，我们可以通过-XX:MaxTenuringThreshold参数将这个值调大一点。动态年龄判断规则，如果一批同龄的对象超过S区的一般，那么这批对象会全部进入老年代，我们可以通过-XX:SurvivorRatio来调整Eden区和Survivor区的比例，通过增加Survivor区的内存，降低动态年龄判断的概率。当Eden区满了时候，再创建对象，会触发Minor GC，如果Minor GC之后S区放不下，这些对象会直接进入老年区，这个也可用同调打S区的内存来解决。

最理想的情况是所有新对象都创建在Eden区，然后每次垃圾回收之后，存活对象都能进入S区，下一次垃圾回收之后存活对象进入另一块S区。这样几乎有很少对象进入老年代，也几乎不会触发Full GC。

Survivo区太小，容纳不了每次Minor GC后存活下来的对象，导致对象频繁进入老年代，频繁触发Full GC。

聊一聊老年代垃圾回收器CMS？

CMS是针对老年代的垃圾回收器，使用多线程回收垃圾，采用标记清理算法，通过可达性分析算法标记处哪些对象是垃圾对象，然后把这些垃圾对象清理掉。标记清理法不会整理存活对象，会导致大量内存碎片，造成内存浪费。

可达性分析算法是查看该对象是否被GC roots引用，局部变量、静态变量都可以是GC roots；由于Stop The World对用户体验不太友好，CMS采用垃圾回收线程和系统工作线程栈尽量同时执行模式来处理的。 CMS垃圾回收一共分为四个阶段：

1. 初始标记

这个阶段会让系统的工作线程栈全部停止，进入Stop The World状态

初始标记就是标记出所有GC Roots直接引用的对象。这一阶段虽然会停止过系统的工作线程栈，但影响不大，因为仅标记GC Roots的工作很快。

1. 并发标记

尽可能对已有的对象进行GC Roots追踪。对于间接引用的对象，会一层一层向上追踪，知道找到GC Roots。这个阶段最耗时，因为一层一层递归的追踪对象的GC Roots，但这各阶段适合系统并发执行的，所以并不会阻塞系统。

1. 重新标记
2. 这个阶段会让系统的工作线程栈全部停止，进入Stop The World状态。

重新标记第二阶段中被系统重新变动过的少数对象，耗时较短。

1. 并发清理

清理对象是比较耗时的操作，但这个阶段会让系统继续运行，所以也不会阻塞系统的运行。

为什么老年代的垃圾回收要比新生代的慢？

1. 新生代存活对象少，GC Roots追踪时很快就标记出垃圾对象和存活对象。而老年代存活对象多，GC Roots 会追踪大量对象，导致耗时增加。
2. 采用复制算法，存活对象放入S区，垃圾对象一下子清理，而 老年代要挨个从内存分散的区域清理对象，导致耗时增加。
3. 老年代清理完垃圾之后还会整理存活对象，尽可能紧凑地排列他们，减少内存碎片，导致耗时增加。
4. 极端情况，老年代会产生Concurrent Mode Failure导致程序变得非常慢

总之，老年代垃圾回收会比新生代慢很多。

聊一聊CMS垃圾回收会导致CPU紧张？

在CMS并发标记和并发清理两个阶段，是比较耗时的。并发标记需要深度追钟大量对象的GC Roots，一个一个检查对象是否存活，所以耗时较高。并发清理需要把对象从内存中的各个随机位置清理掉，所以耗时也比较高。

在这两个阶段，是非常耗费CPU资源的，CMS默认会开启（CPU核数 + 3）/4个垃圾回收线程。假设是2核4G的机器，CMS会有1个线程去进行垃圾回收，本来系统工作线程已经在等待CPU资源，比较紧张了，线程垃圾回收线程还要参与CPU时间片的竞争。

在CMS垃圾回收期间，会产生一些浮动垃圾，所以为了老年代在并发清理阶段能够放心年轻代传过来的对象，一般在执行FULL GC的时候会预留一下空间。CMS垃圾回收触发机制，其中有一个就是老年代内存使用达一定比例，就自动执行GC。

万一老年代执行CMS垃圾回收时，新进入的对象放不下怎么办？

此时不会报OOM错误，OOM错误是Full GC之后某个时机报的。

这个时候，会发生Concurrent Mode Failure，也就是并发垃圾回收失败了，一边回收，一边往里面放对象进入老年代，内存不够。此时就会自动调用Serial Old垃圾回收器代替CMS，Serial Old是单线程垃圾回收器，会强制停止系统工作线程，进入Stop The World 状态，重新进行长时间的GC Roots追踪，标记出来全部对象，不允许新的对象生成，然后一次性把垃圾对象回收掉，恢复系统工作线程。所以Concurrent Mode Failure是很严重的错误，需要设置好触发CMS老年代的内存比例，防止这种情况发生。

CMS垃圾回收会产生内存碎片问题吗？

会的，因为CMS采用的垃圾回收算法是标记清理算法，每次存活的对象在内存里面随机分布，会造成大量内存碎片。如果内存碎片太多，找不到一块连续区域存放对象，又会触发Full GC。CMS有一个参数可以设置不仅仅使用标记清理算法，-XX:CMSFullGCsBeforeCompaction，执行多少次Full GC后执行一次内存碎片整理，默认是0，即每次Full GC之后都会进行内存碎片整理。

老年代Full GC触发时机？

1. 老年代剩余内存小于新生代所有对象大小总和，如果没有设置空间担保参数，会提前进入Full GC，所以这个参数一般都会打开。
2. 老年代剩余内存小于历代进入老年代对象的平均大小，会触发Full GC。
3. 老年代剩余内存大于历代进入老年代对象平均大小，但老年代剩余内存超过了-XX:CMSInitiatingOccupancyFaction，会触发Full GC
4. 老年代剩余内存大于历代进入老年代对象平均大小，老年代剩余内存不超过-XX:CMSInitiatingOccupancyFaction，新生代冒险执行一次Minor GC，但存活对象S区放不下，老年代也放不下，此时会执行一次Full GC

每日上亿请求量的电商系统，年轻代垃圾回收参数如何优化？

每日上亿请求量的电商系统，每日有多少活跃用户：

按每个用户每天平均访问20次请求，那么1亿/20 = 500万日活用户

每天有500万个用户进来浏览，假设其中有10%的人会下单，每天大概有500万 \* 10% = 50万用户下单，那么每天大致有50万个订单。

一般下单的时机集中在高峰期，假设集中在高峰期的4个小时，产生了50万个订单。500000 / (4\*3600) = 34个/秒，每秒会产生几十个订单。这样计算下来似乎不是很多。

但是考虑电商促销场景，比如双11，很多人等着00:00的时候开始购物，可能10分钟就产生了50万个订单。500000 / (10 \* 60) = 8333个/秒。假设我们部署了30台机器，那么每台机器每秒就要处理8333/30=277，近300个订单，就按300算。每台机器每秒要产生300个订单。

假设机器时4核8G的，从机器本身的CPU资源和内存资源角度，抗住每秒下单300个的请求是没问题的。

问题在于如何对JVM有限的内存资源进行合理的分配和优化，以及对垃圾回收优化，尽可能减少JVM GC次数，且尽量避免Full GC。从而降低垃圾回收对高峰期并发压力大的系统的影响。

大促高峰期订单系统内存模型估算：

每秒并发处理300个下单请求时可以的，假设订单对象按1KB估算，则每秒会产生300KB对象，设计下单流程其他业务对象，一般要扩大10~20倍，除了下单操作外，还有其他操作，可以再扩大10倍数。

所以每秒会产生300KB \* 20 \* 10 = 60M对象，但是1秒过后，这60M对象就变成垃圾了，然后新请求又进来。

内存该如何分配：

4核8G的机器，出去操作系统以及一些其他必须的进程占用的内存，JVM进程一般会分到4G内存。其中堆内存可以给到3G，-Xmx3G -Xms3G，新生代是1.5G, -Xmn1.5G, 除去新生代内存就是老年代内存：1.5G；

一个虚拟机栈一般是1M，每秒会并发处理300个请求，所以会生成300个线程，就有300个线程栈，300M，按700M算。剩下的就给永久代，差不多256M。

8G = 4G + 4G

4G = 3G + 700M + 256M

3G = 1.5G + 1.5G

1.5G = 1.3

要注意把-XX:HandlePromotionFailure，空间担保参数打开，避免频繁的Full GC

综上所述,参数设置如下：

-Xms3072M -Xmx3072M -Xmn1536M -Xss1M -XX:PermSize256M -XX:MaxPermSize256M -XX:HandlePromotionFailure

注意：-XX:HandlePromotionFailure参数在JDK1.6就舍弃了，JDK1.6之后发现老年代内存小于新生代对象大小，不回去查找空间担保参数，提前触发Full GC，而是直接比较老年代内存与历代进入老年代的新生代对象平均内存。

所以JDK1.6之后，参数设置如下:

-Xms3072M -Xmx3072M -Xmn1536M -Xss1M -XX:PermSize256M -XX:MaxPermSize256M

每秒会产生60M对象，1.5G / 60M = 25.6 差不多25秒之后，新生代就满了，需要进行一次Minor GC，此时老年代剩余1.5G

在Minor GC之前会比较1.5G > 1.5G，不成功。所以会看是否设置空间担保参数，设置了。于是比较1.5G>0M，可以冒险进行Minor GC。

假设按99%的垃圾对象的比例，一次Minor GC之后存活的对象为1.5G \* 1% =15M，但可能最后一秒的对象还在处理，可能存活对象在100M左右。

如果-XX:SurvivorRatio参数默认是8，则Eden区是1.2G，S0和S1分别是150M。是足够存放Minor GC之后的存活对象的。

新时代垃圾回收的优化之一就是：Survivor内存空间够不够。

按照上述推导，虽然存活对象S区放得下，但是这批同龄对象超过了S区的一半，岂不是也要加入老年代。而且150M比100M多不了多少，鲁棒性不够，如果系统波动，存活对象一下子变成155M，那么S区就放不下了。

根据上面两点原因的分析，目前S区的大小是完全不够的。

这里可以通过调整新生代和老年代的比例，因为这种普通的业务系统，明显绝大部分对象都是短生命周期的，根本不会进入老年代，没有必要给老年代过大的内存空间。首先得让对象留在新生代。通过-Xmn参数调大新生代的内存。

所以，此时可以考虑将新生代内存调整会2G

此时S区会变成200M，100M不会超过50%，波动情况也放得下。

除了优化S区大小，还需优化新生代对象躲过多少次垃圾回收进入老年代。

按照20秒触发一次Minor GC，那么-XX:MaxTenuringThreshold的默认值为15，也即是说，20 \* 15 = 300秒 = 5分钟后，新生代对象会进入老年代，一个对象存活5分钟进入老年代是很合理的。不能盲目的提高这个参数，因为有些对象类似@Service @Controller注解标注类的对象本来就会在内存中存活很长时间，如果把这个进入老年代的阈值提高，这些对象会占着新生代的内存。有时候甚至可以降低这个参数的值，保证这些本来应该是老年代的对象别站着新生代的空间。

-Xmx3072M -Xms3072M -Xmn2048M -Xss1M -XX:PermSize=256M -XX:MaxPermSize=256M -XX:SurvivorRatio=8 -XX:PreTenuringMaxThreshold=1M -XX:MaxTenuringThreshold=5 -XX:UseParNewGC -XX+UseConcMarkSweepGC

S区大小要考虑放得下，动态年龄判断两个点

老年代的参数如何设计：

假设订单系统在大促期间，每隔5分钟会在Minor GC后有200M对象进入老年代

最新的G1垃圾回收器原理，你能聊聊吗？

以前的垃圾回收采用ParNew + CMS组合，但有一个最大的痛点，就是无法避免Stop The World问题，这是G1垃圾回收器发明出来的初衷。

G1垃圾回收器可以同时搞定新生代和老年代的垃圾回收。

G1垃圾回收器特点：

1. 将堆内存划分从多个大小相同的Region



1. 有新生代和老年代概念，但是逻辑上的概念。新生代和老年代对应一些Region



1. 可以让我们设置一个垃圾回收的预期停顿时间

程序员可以指定G1垃圾回收器在一个时间段内，垃圾回收导致的系统停顿不能超过多久，然后全权由G1垃圾回收器负责

1. 一个Region随时可能属于新生代或者老年代，随时在切换。一个新生代Region经过一次垃圾回收之后可能变成老年代或者什么也不是
2. 由于第4点的存在，给老年代和新生代分配内存参数就没有意义了，有G1垃圾回收器自己控制。

G1 垃圾回收器为什么可以做到垃圾回收导致的系统停顿可控呢？

G1垃圾回收器要做到这一点，需要追踪每个Region的回收价值

G1垃圾回收求的核心原理：G1垃圾回收器可以让程序员指定垃圾回收对系统产生影响，通过将堆内存划分成一个个大小相同的Region，同时追踪每一个Region垃圾回收产生的垃圾对象大小和消耗时间，最好在垃圾回收的时候，需按照Region进行垃圾回收以保证垃圾回收对系统的影响在程序员设定的时间范围内，并在有限的时间，尽可能多地回收垃圾对象。

什么是G1垃圾回收器中Region的回收价值？

Region里的对象有多少是垃圾对象，剩余对象就是存活对象。

如果对这个Region进行垃圾回收，需要消耗多长时间，可以回收掉多少垃圾

G1是如何工作的？

1. 划分堆内存为一个个Region
2. 老年代、新生代内存在动态变化，Region的归属在动态变化
3. E区和S区的内存也不是固定的，用多少占多少，有一个最大值，如果E区满了，就会将对象移入S区，如果S区不够放，就会把新的Region加入S区
4. 垃圾清理的时候并不会一次性清理完，而是有选择的清理Region，保证时间。

对象什么时候进入新生代的Region？

创建的时候就进入新生代的Region

什么时候触发Region GC？

当新生代占用的内存超过堆内存的60%时

什么时候对象进入老年代的Region？

新生代和老年代都会占据一定的Region，按照新生代最多占60%Region算，老年代最多占40%Region，2048 \* 40 % = 812 大约800个Region

而新生代对象进入老年代的时机基本与前面相同

1、-XX:MaxTenuringThreshold，年龄判断

2、动态年龄判断：存活对象超过了S区的50%，大于等于这批存活对象年龄的对象就会进入老年去。

G1大对象会进入老年代吗？

不会，G1专门提供了大对象的Region，如果一个对象就直接超过了Region的50%，按一个Region 2M计算，那么一个对象如果超过1M，就会直接放入专门的Region中。

而且，一个对象如果太大了，就会横跨多个Region来存放

由于Region的所属不是静态的，是动态的，所以一个Region可以是老年代，也可以是新生代，也可以用来存放大Region。在新生代和老年代回收的时候会顺便查看大对象的Region是否要回收

什么时候触发老年代的Region GC？

G1垃圾回收器的设计思想？

G1垃圾回收器把堆内存划分成一个个大小相同的Region，新生代和老年代各对应一部分Region，垃圾回收的时候尽可能选择停顿时间最少且回收垃圾最多的Region。尽量保证达到在程序员指定的系统停顿时间之内

G1垃圾回收器为什么好，原理是什么？

G1中Region要设置多大，要设置多少个Region？

-Xms和-Xms设置堆内存大小

-XX:UseG1GC 设置使用G1垃圾回收器

此时G1垃圾回收器会自动用堆大小除以2048

因为JVM最多有2048个Region，且Region的大小必须是2的x次方，如1M，2M，4M

比如堆大小是4G = 4096M，4096 / 2048 = 2M，即每个Region大小为2M

当然也可以通过-XX:G1HeapRegionSize来指定。

刚开始的时候，新生代默认占堆内存的5%，也就是4G \* 5% = 200M = 100个Region \* 2M； 所以新生代大概有100个Region，也可以通过-XX:NewSizePercent调节，在系统运行过程中，会给新生代增加更多的Region，但是新生代最多占比不会超过60%，可以通过-XX:MaxNewSizePercent调节。Region如果进入垃圾回收，新生代Region数量还会减少，这些都是动态的。

新生代还是有Eden区和Survivor区划分，可以通过-XX:SurvivorRato参数调节E区和S区的比例

聊一聊G1的Young GC?

假设按默认参数，随着系统不停的想Eden区对应的Region存放对象，系统会不断地把Region加入新生代，知道新生代内存占据堆内存的60%。

一个2048个Region，新生代Region数量 = 2048 \* 60 % = 1228个Region

假设按1200个Region算

新生代1200个Region = Eden区1000个Region + S0区100个Region + S1区100个Region

这时候再创建对象，会触发新生代GC，G1垃圾回收器就会通过复制算法进行垃圾回收。进入Stop The World状态，把Eden区中对应的Region中存活对象复制到S0对应的Region中，然后进行垃圾清理。

但是这个垃圾清理与之前不一样，不一定是一次性全部清理完，而是根据-XX:MaxGCPasueMills参数指定的时间范围，以及追中的每个Region垃圾对象大小和垃圾回收消耗时间的信息，选择一部分Region进行垃圾回收，保证停顿时间控制在XX:MaxGCPasueMills参数之内，然后尽可能多地回收垃圾

G1垃圾回收器也有存活对象S区放不下，进入老年代的情况

G1什么时候触发新生代+老年代混合垃圾回收？

G1有一个参数是 -XX:InitiatingHeapOccupancyPercent。默认是45%；

如果堆一个有2048个Region 当老年代的Region占据2048\*45%，接近1000个Region时，就会触发混合回收：

1. 触发初始标记操作，这个过程需要进入Stop the World，只标记GC Roots直接引用的对象，所以这个过程是非常快的。
2. 并发标记
3. 最终标记
4. 混合回收：最后一个阶段可以执行多次，系统运行，停一会，执行混合回收最后阶段，系统运行，过一会系统停止，再执行混合回收最后一阶段

可以通过-XX:G1MixedGCCountTarge参数调节

混合回收是基于复制算法的，要先把回收的Region里的存活对象放入其他Region，然后一次性清理Region剩余垃圾，得到一个空的Region，这样就不断有空的Region生成，当空Region占堆内存的-XX:HeapWastePercent参数值时，默认是5%，就会停止回收

G1整体都是基于复制算法的，不会想CMS那样产生内存碎片，还要整理

为什么初始标记很快？

因为只需要标记GC Roots直接引用的对象，而局部变量和静态变量作为GC Roots，垃圾回收线程可以直接去扫描线程栈和方法区的局部变量和静态变量所引用的对象，然后直接标记，而线程栈和方法区内存较小，很集中，扫描起来很快。

线上系统部署如果采用G1垃圾回收器，应该如何设置参数？

G1垃圾回收器和ParNew+CMS比较？

百万级用户教育在线平台，如何基于G1垃圾回收器优化性能？

每天十几万日活，高峰期是上课那几个小时。

假设晚上3个小时共有60万个用户，平均每个用户会使用1个小时左右来上课，每个小时有20万个在线用户。这个20万个用户平均每1分钟都会与系统交互一次，那么一小时就会有20万\*60=1200万次交互，平均到每秒就是1200万/3600 = 3333.333 约3000次左右的请求。

系统每秒要承受3000次左右的并发请求，部署5台4核8G的机器来抗是差不多的，每台机器每秒抗3000 / 5 = 600条请求

假设一个请求会产生5KB对象，那么一秒就会产生5KB\*600=3M对象

机器是4核8G，给JVM进程分配5G内存

JVM给堆分配4G内存，方法区256M内存，一个线程栈1M

堆4G内存，新生代初始占比默认为5%，最大占比60%。

-Xms4096M -Xmx4096M -Xss1M -XX:PermSize=256M -XX:MaxPermSize=256M

-XX:+UseG1GC -XX:G1NewSizePercent=5 -XX:G1MaxNewSizePercent=60

此时堆内存大小是4096，默认一共有2048个Region，每个Region大小是2M

新生代初始有 2048 \* 5 % = 102，100个左右的Region。200M空间。

设置G1停顿时间：

-XX:MaxGCPasueMills ，默认值是200ms，先保持默认

多长时间会触发新生代GC：

系统运行起来后，会不断在新生代Eden区分配对象，每秒产生3M对象

Eden区不够时会触发GC：

假设系统里，G1回收300个Region(600M内存)，大致需要200ms

随着系统的运行，每秒创建3M对象，一分钟左右会产生3 \* 60 = 180M对象，大约会占用100个Region。G1此时可能认为如果现在就进行GC，只不过回收200M左右的内存，只需要几十ms，里200ms相差甚远。且如果现在GC，一分钟过后是不是又要GC，这样太频繁了，不如给新生代一些新的Region，让系统继续在新生代创建对象，不必过于频繁的GC

-XX:MaxGCPasueMills为何重要？

因为这个参数影响了新生代GC，与Mixed GC

1. 影响新生代GC：
   1. 如果这个参数设置过小，会导致才分配几十M对象，G1认为回收这些对象会达到-XX:MaxGCPasueMills而进行GC，每次GC虽然时间很短，但是很频繁
   2. 如果这个参数设置过大，G1会让新生代不断地分配对象，并让Region加入新生代，因为G1认为回收目前的对象离XX:MaxGCPasueMills相差甚远
2. 对Mixed GC的影响：

2.2 如果这个值设置过大，会让新生代堆积很多对象，从而存活下来的对象更多，有可能S区放不下，或者一批同龄对象超过S区的50%，就会导致大批对象进入老年代，引发频繁Mixed GC

基于JVM运行的系统最怕什么？

系统卡顿问题

G1垃圾回收器与ParNew相比好在哪里：

对于新生代GC，当堆内存较小时，G1和ParNew是差不多的，因为几十分钟或者一个小时新生代满了，采用复制算法清理也就几十ms就清理完了，而对于几十G的大内存机器，每次新生代满可能要清理十几个G的垃圾，会消耗十几秒，系统每个一个小时就卡死十几秒。这是不可容忍的。

所以，对于大内存机器，就体现出了G1垃圾回收器的好处了，程序员可以设置系统停顿的时间，G1根据这个阈值，自动调节GC的时机，防止新生代中有十几G的垃圾对象。

解释一下什么是Young GC和Full GC？

Minor GC/ Young GC: 新生代可以称之为“年轻代”，对应的英文分别是Minor和Young。所以Minor GC和Young GC是等价的。

Full GC/Old GC: 老年代GC=Old GCFull GC

Full GC: 指新生代、老年代、永久代的全体内存的垃圾回收

Major GC：少提，如果提到了，要问清楚指的是什么

Mixed GC：G1中特有的概念，当老年代所占的Region的内存占堆内存的45%时，会触发Mixed GC，有四个阶段，初始标记、并发标记、最终标记、混合清理

Young GC和Full GC分别在什么情况下会发生？

1. Young GC触发时机：新生代的Eden区要满了的时候。
2. Old GC触发时机：(1)Young GC之前：老年代剩余内存小于历代进入老年代的对象的平均大小，(2)Young GC 之后：Young GC之后存活对象S区放不下，老年代也放不下。(3)老年代中的内存使用率超过92%了

永久代内存满了怎么办？

触发Full GC通常会把新生代、老年代、永久代里面的垃圾对象都清理

如果真发生永久代内存满了，就会OOM

每秒10万并发的BI系统是如何频繁发生Young GC的？

什么是BI系统：给商家提供服务，平台的平台

实时数据报表：每隔几秒发送后台请求刷新数据

假设每秒有500个请求

每个请求产生100KB数据

每秒就会产生50M对象

4核8G，JVM进程4G，堆3G

新生代1.5G，老年代1.5G

E区1G，S区250M左右

1s 50M对象，20s左右就触发一次Young GC

1个G对象Young GC可能几十ms

所以用户感知不到

升级配置：采用大内存机器

16核32G机器，新生代20G，E区占到16G

每秒几千次请求，每次会产几百M对象

此时频繁触发Young GC就会出现问题，因为一次处理16G的对象会让系统卡顿1s或几s，用户会明显感觉卡顿

采用G1垃圾处理器优化大内存机器Young GC 性能

JVM有哪些参数？

|  |  |
| --- | --- |
| 参数 | 说明 |
| -XX:+UseParNewGC | 使用ParNew作为垃圾回收器 |
| -XX:HandlePromotionFailure | 设置如果老年代内存<新生代所有对象大小时要不要Full GC，如果没设置就要Fuul GC，设置了就不要Full GC，而是进一步检查。 |
| 内存相关参数 | -Xmx -Xms -Xmn -XX:PermSize  -XX:MaxPermSize |
| XX:SurvivorRatio | 设置Eden区和S区的比例，如果XX:SurvivorRatio=8，则Eden区与S区的比例是8:1:1 |
| -XX:ParallelGCThreads | 设置ParNew垃圾回收线程数 |
| -XX:MaxTenuringThreshold | 判断对象为老年代的年龄的参数 |
| XX:PretenureSizeThreshold | 判断对象为大对象进入老年代的参数 |
| -XX:CMSInitiatingOccupancyFaction | 老年代内存达到多少比例触发CMS垃圾回收 |
| -XX:CMSFullGCsBeforeCompaction | 执行多少次Full GC后执行一次内存碎片整理，默认是0 |
| -XX:+UseConcMarkSweepGC | CMS:C:Conc, Concurrent; Mark; Sweep；并发标记清理垃圾回收器 |
| -XX:+UseG1GC | 设置使用G1垃圾回收器 |
| -XX:G1HeapRegionSize | 指定Region大小 |
| -XX:G1NewSizePercent | 新生代占用堆内存比例(初始) |
| -XX:G1MaxNewSizePercent | 新生代最大堆内存占比(最大) |
| -XX:MaxGCPasueMills | 设定G1执行GC的时候可以让系统停顿多长时间 |
| -XX:InitiatingHeapOccupancyPercent | 默认值45%, G1的参数，如果老年代占用的Region超过这个值，此时就会尝试触发新生代+老年代一起混合回收阶段 |
| -XX:G1MixedGCCountTarge | 在一次混合回收过程中，最后一个阶段执行几次，默认是8次 |
| -XX:HeapWastePercent | 默认5%，混合回收时，如果空Region占堆内存的这个参数时停止回收 |
| -XX:MixedGCLiveThresholdPercent | 默认85%，当一个Region存活对象占比85%以下时才回收。  如果存活对象过多，复制起来反而更麻烦，不如不回收。 |
| 日志打印参数 | XX:+PrintGCDetils：打印详细的gc日志  -XX:+PrintGCTimeStamps：这个参数可以打印出来每次GC发生的时间  -Xloggc:gc.log：这个参数可以设置将gc日志写入一个磁盘文件 |
|  |  |
|  |  |
|  |  |

为什么G1 Mixed GC最后阶段回收的时候要设置多次呢？

因为将一个大任务切分成多个小任务，原本一次要停顿很久，用户会有明显的感觉，但通过每次回收一点，每次停顿一点点，用户不会有明显的感知。

万一Mixed GC，回收Region时存活对象没有空闲Region存放？

此时情况很严重，会触发Full GC。停止系统工作线程栈，采用单线程对堆内存进行GC，知道清理出空余Region，这个过程极其缓慢。

介绍一下Object常见方法？

好的，Object类作为Java中所有类默认的父类有很多方法，我下面列举一些常用的方法：

toString()方法，该方法默认人实现是返回类名@哈希值的十六进制表示，通常我们会重写该方法以打印对象相关信息。当打印一个对象、一个对象与字符串做运算时JVM会自动调用对象的toString方法。

hashcode和equals方法，hashcode会返回一个对象的哈希码，这个哈希码通常与对象的内存地址相关，hashcode是用natvie修饰的，所以默认是本地语言实现的。equals方法用于比较对象与另一个对象是否相等，默认实现是比较两个对象的地址值。通常如果要使用哈希表来存储对象，需要同时重写这两个方法，以保证这两个方法的行为一致。

wait，notify()/notifyAll()，这一组方法用于多线程同步代码，被上锁的对象调用wait方法，会让当前线程进入等待状态，并释放锁，指定该对象的锁被别的线程拿到并调用notify方法或者notifyAll()。notify会从等待该对象锁的线程随机唤醒一个线程，抢占锁的资源。而notifyAll方法会唤醒所有等待该对象锁的线程，并让它们一起去抢占锁，没有抢到锁的线程会进入Blocked状态，直到拿到锁。

为了保证元素在HashMap和HashSet等集合中正确存储，通常需要将它俩一起重写。

Java为什么被成为平台无关性语言？

因为有JVM的存在，Java中有一句话叫做一次编译，到处运行。指的是一个后缀为.java的Java源文件经过javac命令编译为后缀为.class字节码文件后，就可以在不同平台的JVM上运行。在不同平台用java命令运行一个程序是，操作系统会启动一个JVM进程，这个JVM进程会将字节码加载进内存，这些字节码文件本质上一条条字节码指令，操作系统本身是看不懂的，JVM会把这些字节码指令翻译成对应操作系统的机器码指令，程序能够被不同平台执行。这提高了Java程序的可移植性，只需要针对不同的操作系统提供对应的JVM即可，这就是为什么Java是平台性无关语言。

== 和 equals有什么区别？

==会直接比较变量的值，对于引用变量，存储的值是实际指向的对象在内存中的地址，所以两个引用变量用==比较，会比较两个变量所指向的对象是不是同一个。equals是Object类中的一个方法，默认实现就是用==比较，所以一个对象如果没有重写equals方法，==和equals是等价的。但我们写的类通常会重写equals方法来达到比较两个对象符合现实世界的比较方法，不同的类实现方法不同，怎么比较取决于别人怎么写。

首先，==是一个操作符，equals是Object类中的一个方法，equals方法默认使用==来比较的。==是比较变量的值，对于基本数据类型变量，值就是存储在变量中，对于引用变量，存储的是对象的地址，所以用==比较两个引用变量实际上是在比较两个对象的内存地址是否一样。

hashcode与equals的区别与联系？

hascode方法通过将对象的内存地址，通过一个hash函数返回一个整数，代表该对象的内存地址，是一个简单的内存地址映射。hashcode的速度非常快。

equals比较两个对象是否“相同”。equals可以是很复杂的实现，也可以是默认实现，等价于==。

在一些极端的请款下，不同的对象hashcode值可能相同，hashcode并不是一个十分可靠的方法。equals考虑的就比较多了。

|  |  |
| --- | --- |
| 序号 | 描述 |
| 1 | equals成立，hashcode一定成立 |
| 2 | equals不成立，hashcode可能成立 |
| 3 | hashcode成立，equals不一定成立 |
| 4 | hashcode不成立，equals一定不成立 |

理解技巧：把equals就作为默认实现，等价于==

通常如果要使用哈希表来存储对象，需要同时重写这hashcode和equlas，以保证这两个方法的行为一致，为什么？

p.hash == hash

&&

(k = p.key) == key

||

(key != null && key.equals(k)

)

有源码可知，它们不是或运算连接的，而是与运算连接。equals排在最后，首先比较hashcode，然后用==运算符比较，最后才来看equals

所以我们要同时重写hascode和equals，因为hashcode有一票否决权，如果hashcode不相同，而equals相同，那么也不会是重复的。

String类和Integer类重写了hashcode和equals方法吗？

重写了

|  |  |  |
| --- | --- | --- |
| 类 | hashcode | equals |
| Integer | 返回 value | 比较value值 |
| String | 与value字符串数组的内容有关 | 比较value字符数组 |

HashMap源码追踪？

HashMap<String, Integer> hashMap = new HashMap<>();

hashMap.put("a", 1);

1这里会执行装箱,Integer.valueOf(1)

put方法 🡪 public V put(K key, V value)

K: String, V: Integer

putVal(hash(key), key, value, false, true);

hash函数返回依据key的hashcode计算的值

putVal🡪 V putVal(int hash, K key, V value, boolean onlyIfAbsent, boolean evict)

hash: key的hash值

key:“a”

value: 1

if ((tab = table) == null || (n = tab.length) == 0)

(tab = table) == null

||

(n = tab.length) == 0

首先看table是不是null，或者tab的长度为0，就是还没放过元素，第一次调用put，这个是true

n = (tab = resize()).length;

获取一个tab，大小为16

if ((p = tab[i = (n - 1) & hash]) == null)

(p = tab[i = (n - 1) & hash]) == null

tab中”a”hash值转为数组下标后有没有值存放

tab[i]会创建一个新node

如果这地方不为空，意思前面有key占据了

p就不为空，而是一个node

p.hash == hash

&&

(

(k = p.key) == key

||

(key != null && key.equals(k))

)

这里使用了三种比较对象的方法

hashcode

==

equals

它们的关系是：

hashcode && (== || equals)

1. hashcode如果不相同，则这两个对象一定不重复
2. hashcode如果相同：==或者equals有一个相同就是重复

根据原源码追踪，我们自定义类在用Hashtable存储时，要重写hashcode和equals方法，在equals成立时，hashcode一定成立。hahcode不相等时，equals一定不相等。

hashcode值相同，但不是同一片内存地址，现在就要判断equals是不是相同，如果是，则是重复key。如果不是，则不是重复key。

如果不重写hashcode，那么hashcode方法和==就是等价的: ==

如果不重写equals方法，那么equals方法和==就是等价的: hashcode && ==

聊一聊 hashcode和equals，什么时候要重写，怎么重写， 为什么这么重写？

首先，equals是Object类的一个方法，默认实现是通过==运算符比较。hashcode也是Object类的方法，native修饰，由本地语言实现，hashcode会把对象的信息如内存地址，经过一个hash函数变成一个int值返回，这个数值成为哈希值。

当我们想自定义两个对象是否相等的比较规则的时候，需要重写equals，但是为了保证类的对象在HashMap、HashSet等集合里面正确存储，同时要重写hashcode。

举个例子，定义一个石头类，有一个属性是重量，重写了equals方法，范围两块石头的重量是否相同，如果相同就为true，反之为false。分别创建出了两个重量相同的对象添加到HashMap中。

我们期望HashMap中最终出现的只有一块石头，因为我们认为重量相同的两块石头是重复的，HashMap在判断两个对象是否重复时首先会用hashcode值判断，然后才回去比较equals。如果没有重写hashcode，由于创建了两次对象，hashcode值时不同的，所以Hash Map会认为这两个石头对象不是重复的。

解决办法是重写hascode，直接返回石头的重量。

1. 回答了重写时机：(1)当需要自定义类相等的比较规则时；(2)当需要用HashMap、HashSet存储时。
2. 回答了为什么要重写：举了一个例子
3. 回答了怎么重写：equals返回石头重量比较结果；hashcode直接返回石头重量。

聊一聊重载和重写的区别？

重载是有多个相同方法名的方法，但方法的形参列表不同，JVM通过调用者传入的实参列表来匹配相应的方法执行。子类与父类同名，但形参列表不同的方法不会覆盖父类。

重写是子类对父类同名且同实参列表的方法覆盖。

重载不需要继承或实现接口，在同一个类中可以实现同名方法的重载。

重写发生在子类继承父类，重写父类的方法，实现类实现接口，重写接口中的方法时。当父类的方法没有方法体时，子类如果不是虚函数，就必须实现该方法。

重载：同一个类中存在多个方法，这些方法拥有相同的方法名，但是参数的类型、个数、顺序不同。重载提供了一种灵活的方式实现相似的功能。

重写：子类重写定义并覆盖父类的方法。使得同名方法可以根据子类的不同具体类型调用。

对于JVM来讲，方法重载是“静态分配”阶段实现的，即在编译阶段JVM通过实参的静态类型，决定程序应该调用方法的哪个版本。方法重写是动态分配的，程序运行起来，字节码加载进内存，通过字节码中的invokevirtual指令动态实现，该指令会根据调用者的具体类型来决定使用哪个方法。

方法重写对应动态分配过程，在编译阶段，JVM并不知道变量调用的是子类的重写方法，还是父类的方法。在运行阶段，字节码被加载进内存，JVM通过字节码中invokeviartual指令实现方法重写，该指令会根据调用者的具体类型决定方法的执行版本。

抽象类和接口的区别？

1. 可以包含抽象方法的类
2. 抽象方法是指没有方法体的方法。

当父类的某些方法需要申明，但又不明确怎么实现的时候，就可以将这些方法声明为抽象类，

抽象类是为了解决父类方法不确定性问题。当父类中某些方法，不知道该如何实现又不得不实现时，就可以把这个方法定义为抽象方法，这个类就是抽象了。抽象方法必须在抽象类中。

抽象类不可以被实例化

抽象类不一定要包含抽象方法，反过来不行，包含抽象方法的一定是抽象类。

abstract只能修饰类或方法

抽象类还是类，可以有任意的成员。

继承抽象类，必须实现所有的抽象方法，或者把本类也申明为抽象类。

抽象方法不能被parivate、final、static修饰，因为这几个修饰符都是和重写相违背。注意静态方法是无法重写的，因为调用静态方法时本质上是通过类名调用，静态方法是属于类层级的，重写没有意义。

Java 类和接口的区别与联系？

Java类是一系列具有相同属性和行为的对象的模板或蓝图

接口是一种特殊的Java类，里面定义了一系列抽象方法，用于定义某些规则的规范。

抽象类可以有构造器吗？

可以的，虽然抽象类不能被实例化，但是拥有一般的类的所有东西

关于抽象类的八个细节？

1. 抽象类不可以被实例化
2. 抽象类可以不包含抽象方法
3. 抽象方法一定属于抽象类
4. abstract只能修饰类、方法
5. 抽象类可以包含普通类拥有的一切：构造方法、成员变量、静态变量、成员方法、静态方法…
6. 抽象方法没有主体
7. 如果一个类继承了抽象类，这个类必须重写所有抽象方法或者定义为抽象类
8. 抽象方法不能被private、static、final修饰，因为这些修饰符违背了重写原则



接口、类、抽象类概念？

什么叫做对象：是人们要研究的任何事物

什么叫做类：类是一系列具有相同属性和行为的对象的原型或者蓝图

什么叫做抽象类: 当父类中的方法需要定义但又不直到具体的行为什么，就可以将这个方法定义为没有方法体的方法，即抽象方法，而这个类就是抽象列。

什么叫做接口：对象通过公有方法与外部世界进行交互，方法形参了对象与外部世界的接口。接口的本质是公有方法。

接口类型：一系列抽象方法的集合。

|  |  |  |
| --- | --- | --- |
| 接口 | 普通类 | 抽象类 |
| 抽象方法，default修饰的成员方法，静态方法，常量 | 成员方法，成员变量，静态方法，静态变量，代码块，静态代码块。  常量 | 普通类+抽象方法 |

jdk1.8及以后

jdk1.7及之前：接口里面的方法都没有方法体

为什么要接口类型：一系列类，他们都需要有与外界进行交互的同一个接口。那么我直接定义一个接口类型，让只要实现了这个接口类型的类都拥有了这个接口。

接口的使用细节？

1. 接口不能被实例化
2. 接口里面的方法可以不用写修饰符，默认用public abstract修饰，如何证明
3. 一个类实现接口，必须将该接口的所有抽象方法都实现
4. 抽象类实现接口，可以不实现接口里面的方法
5. 单一继承，多实现接口
6. 接口里面的变量=public static final，接口里面的方法=public abstract
7. 接口可以继承接口，且是多继承public interface InterfaceSon extends InterfaceFather1,InterfaceFather2
8. 接口里面的方法要么不写，默认是public，要么写public
9. 接口中的属性和方法都会传递给实现类

接口里面可以定义静态变量，为什么必须初始化？

因为接口里面的变量会把JVM解析成public static final 修饰，由于有final修饰，所以必须要初始化。

public修饰：满足接口的型装，接口是与外部世界交互的公有方法，如果都私有了，违背接口的定义

static: 因为接口不能实例化，所以定义变量必须static

final：一系列类与外部世界进行交互时用到的参数因该统一且不允许其他类修改。举个例子：电脑usb接口的大小、深度都是固定死的参数，如果产家随意修改就乱套了。

实现implement和继承extends的区别？

继承：如果多个类具有部分相同的属性或行为，那么可以将这部分公共的属性和行为抽象成一个类，让这些类都继承父类。

实现：如果多个类处理目标是一样的，但是处理方式不同，那么就定义一个接口，让这些实现这个接口。

联系：实现和继承都可以将接口，父类的属性和方法传递到实现类，子类。达到代码复用的效果

区别：1、修饰符不同；2、数量不同；3、属性不同；4、调用不同；

当子类继承父类，就拥有了父类所有的功能

如果需要拓展功能，就用接口实现。

继承的价值：主要是解决代码复用和可维护问题

接口的价值：在于设计。设计好规范的方法，定义规则，让其他类去实现。

接口更加灵活

继承是 is a关系，接口是 like a。

接口在一定程度上实现代码解耦，即接口的规范性再配合动态绑定机制。

聊一聊接口多态性？

什么是多态：顾名思义，多种形态，同一事物，多种形态。在Java中，多态是指针对同一个方法的调用，其真正执行的方法的版本取决于运行时，调用者时机的类型。

多态参数

多态数组

接口的多态传递现象：接口继承

当父类和实现接口类有同名变量时，需要清晰地告诉编译器使用哪个变量。

什么是解耦？

解耦就是降低耦合性，耦合性就是指两个模块间的关联性，关联性当一个模块发生改变时，是否会影响到另一个模块。解耦就是要降低这个影响。

聊一聊设计模式中的模板模式？

通过将多个类里面共有的行为给抽象出来，放在抽象类中，这个抽象出来的方法父类可以不用实现，只需要声明为抽象方法即可，而抽象方法必须出现在抽象类中。

抽象类和接口的区别？

抽象类可以用于一个普通类所拥有的一切东西，构造方法、成员变量、成员方法，静态变量，静态方法，常量等等

接口中可以只能由于静态常量

jkd1.7及以前，接口只能拥有抽象方法，jdk1.8及以后，接口中可以拥有默认方法，静态方法。

1. 从代码定义来讲：在Java中，抽象类指被abstract修饰的类，接口是特殊的类，用关键字interface修饰。
2. 从功能上来讲：抽象类主要是用于代码复用和可维护性。比如一些类拥有通用的功能，为了不让代码在每个类中重复一遍，可以定义一个抽象类，将共有的方法写在抽象类中。这样依赖，只需要将其他类继承抽象类即可，如果需要修改该方法，只需修改抽象类中的方法。如果子类有不同的实现，只需要重写即可。这些功能，普通类也可以实现，为什么采用抽象类，是因为父类中可能有一些方法需要声明，但是又不知道怎么写实现时，就可以把方法定义为没有方法体的抽象方法。而这些抽象方法所在的类就是抽象类。
3. 接口的出现就是为了解决继承的缺点，因为继承只能是单继承，一个类只能继承一个父类，拥有父类全部的功能，是is a的关系，但是当需要功能拓展时，继承就办不到了，这时候，接口发明出来。
4. 接口的主要作用是用于设计。定义好规范的方法，让其它类来实现，达到功能拓展的目的。接口是like a的关系，接口可以多实现。在表示has a的关系时，可以用接口。

这道题本质上是在问继承和实现的区别：

抽象类 🡪 继承 🡪 代码复用

接口 🡪 实现 🡪 功能拓展

继承用于代码的复用和可维护性

接口用于设计。定义规范的方法，通过实现达到功能拓展的作用

继承的缺点：单继承，只能继承一个类，当需要新功能时不好拓展

接口发明出来解决继承的缺点。

接口一系列抽象方法的集合。比如定义一套方法的规范，可以将这套方法的规范抽象成一个接口。每一个实现这个接口的实现类都必须实现接口所有的抽象方法。

1. 抽象类和接口的Java代码定义：在Java中抽象类指用abstract修饰的类，接口指用interface修饰的类。
2. 抽象类的作用：抽象类主要用于代码的复用和可维护性。当一些类具有相同的属性和行为时，可以将这些属性和行为抽象出来放到一个父类中，这些去继承父类，不用把代码在每一个子类中写一份，如果一些子类需要有不同的实现，可以重写父类的方法。当然，这个功能普通的类也可以做到。抽象类是当父类中的某些方法必须要申明，但又不知道怎么实现时，就可以将这个类定义为没有方法体的抽象方法，而包含这些抽象方法的类就是抽象类。
3. 继承的缺点：继承时is a的关系。在Java中，由于继承都是单继承，所以当一个子类在继承父类时会用于父类的全部行为，但是如果要进行功能拓展时继承就很难办到。
4. 接口的作用：接口是一些列抽象方法的集合。接口主要用于设计。定义出一套规范的方法。接口是like a关系，当一个类需要用到某些功能时，就可以定义一个接口实现功能扩展



谈一谈你对final关键字的理解？

总述: final是Java里面的关键字，可以用来修饰变量、方法、类

分述：

1、当final修饰类时，表示这个类不允许被继承。例如，Java希望String类是不允许被继承的，就在String类的前面加上final。

2、当final修饰方法时，表示该方法不允许被重写。例如Object类中有一个getClass方法，Java希望该方法不允许被重写，于是用final修饰。

3、当final修饰变量时，表示该变量不允许被修改，这里有两层含义，第一，对于基本数据类型定义的变量，变量里面的存储的是实际的值，如果被final修饰，对变量赋值、修改变量的值都不允许。第二，对于引用数据类型，变量里面存储的是对象的地址值，如果被final修饰，表示不能修改变量的指向，但对于引用变量指向的对象，却没有限制，可以任意修改。

聊一聊枚举？

枚举是一个动作，就是将所有情况一一列举出来。

在Java中，枚举是一个特殊的类型，一组有限的特定对象的集合。

在具体一定：枚举是一组常量的集合。

枚举使用enum关键字修饰。

枚举中的对象是public static final修饰的常量，所以用大写。

枚举实现的特点：

1. 私有化构造函数：枚举拥有一组有限的常量。在创建枚举的时候就已经将所以常量放在枚举类中，不允许外部构造新对枚举对象。
2. 枚举对象的变量名大写，因为符合常量的书写规范
3. 枚举中的属性：枚举拥有的对象。对象中的属性

枚举是Java中用enum关键字修饰的类，枚举的作用是存储一组有限的特定对象。枚举在是现实有会私有化构造函数，因为枚举在创建时就已经将所有的枚举对象放在枚举类中，不允许在外部新建对象。不提供set方法，因为枚举对象时产量对象不允许外部修改。枚举类中的属性分为：枚举常量对象和枚举类的成员变量。枚举常量对象的修饰符是public static final

为什么私有化构造函数：防止外部创建

为什么不提供set方法：防止外部修改

为什么要public：提供外部使用

为什么要static：因为私有化构造函数，无法通过实例对象获取，只能通过类名获取

为什么要final：因为static可以通过类名获取对象，防止外部修改

枚举对象作为属性，用public static final修饰。变量名字用大写，符合常量的规范。

枚举对象作为对象可以用于多个属性。

枚举类的特定：

1. 构造器私有化
2. 不提供set方法
3. 在枚举类中创建一组对象，对外暴露对象
4. 暴露对象，用public static final修饰

enum使用时的一些细节?

当使用enum定义枚举类时，默认会继承Enum类，可以通过javap 反编译命令证明。public final class 韩顺平Java.枚举.关键字实现枚举.Season extends java.lang.Enum<韩顺平Java.枚举.关键字实现枚举.Season>

public static final 韩顺平Java.枚举.关键字实现枚举.Season SPRING;

public static final 韩顺平Java.枚举.关键字实现枚举.Season SUMMER;

public static final 韩顺平Java.枚举.关键字实现枚举.Season AUTUMN;

public static final 韩顺平Java.枚举.关键字实现枚举.Season WINTER;

枚举对象用public static final修饰

而且enum类是一个final类型。

枚举对象必须放在枚举类的行首

如果枚举里面有无参构造，则枚举对象括号都不用写

总结：使用enum关键字后就不能继承其它类了，因为enum隐式继承了Enum类，而java是单继承的。但实现接口是可以的。

枚举类常用方法 ？

枚举类会自动继承Enum类，其实这个问题是在问Enum中常用的方法：

|  |  |
| --- | --- |
| 枚举类中继承至Enum的方法 | 说明 |
| name() | 得到当前枚举常量的名称 |
| ordinal() | 从0开始编号，该枚举常量的次序 |
| values() | 返回有限枚举常量数组 |
| toString | 返回name，当前枚举常量的名称 |
| valueOf | 根据字符串得到枚举对象 |
| compareTo | 比较两个枚举对象的编号 |
|  |  |

你说你的项目用到了枚举，是怎么用的，聊聊你对枚举的理解？

我在项目中定义发生错误时返回给前端的异常代码，因为这些异常代码和描述通常是有限的几种，所以我才用了枚举类来存储。

在java中，枚举是指用enum关键字修饰的类。枚举存储了一组有限的常量对象。枚举在定义时就会将所有对象创建枚举类内部，所以枚举类的构造函数时私有的并不提供set方法。枚举常量对象定义可以直接大写的变量加上括号，括号里面传入对象的属性，如果枚举类定义了无参构造，这个括号也不需要。枚举类产量对象在JVM会加上public static final修饰符，加public是为了暴露对象，static是为了能够用类名获取，因为构造函数私有化了。final是为了产量对象不会被修改。

enum修饰的类会隐式继承Enum类，所以不能再继承其他类，但enum修饰的类可以实现接口。

enum修饰的类JVM会在类名前面加上final关键字，所以其他类不能够继承枚举类。

枚举产量对象的定义要写在枚举类行首。

说一说你对异常的理解？（指导：就是回答异常有啥用之类的）

在Java中，将程序在执行过程中发生的不正常情况称之为异常。异常分为两类：

1. Error(错误)，Java虚拟机无法解决的严重问题。如JVM系统内部错误，资源耗尽等情况。比如StackOverflowError(栈溢出错误)和OOM内存溢出等错误。无法处理
2. Exception，因编程错误或者外在因素导致的一般性问题。可以使用针对性代码处理。如空指针异常，除以0产生的算数异常，试图访问不存在的文件的异常。Exception分为编译时异常和运行时异常，指异常发生的时机不同。

异常对应的单词是Exception

异常体系图：

在Java，程序从执行到运行过程中会遇到一些问题，这些问题分为两大类：错误和异常。

错误是指JVM无法解决就的严重问题。比如StackOverflowError和OOM。

异常是指由编程错误或外部因素引起的一般性问题。可以使用针对性代码进行解决。比如空指针异常，数组越界异常，类型转换异常，算数异常等。

异常分为运行时异常和编译时异常两类。

编译时异常和运行时异常的区别？

在Java中，继承RuntimeException的类都是编译时异常，在程序运行阶段满足特点条件才会显示出来。在编译阶段不是必须要处理的。

编译时异常是直至继承Exceptption的但不是RuntimeException的子类的类。编译时异常在编译阶段就会被检测出来，要求程序员必须处理，要么抛出，要么捕获。

五大常见的运行时异常？

1. NullPointerException
2. ArrayIndexOutOfBoundsException
3. ArithmeticException
4. ClassCastException: 通常发生在父类对象转子类对象时，向下转型会抛出这个异常。如果两个对象都没有继承关系，则编译阶段就已经会报语法错误了。inconvertible types
5. NumberFormatException

常见的编译异常？

编译异常是指在程序编译阶段就必须处理的异常，否则代码不会允许通过。常见的编译期异常有：

|  |  |
| --- | --- |
| 异常 | 说明 |
| SQLException | 操作数据库时，查询表可能发生的异常。 |
| IOException | 操作文件时，发生的异常 |
| FileNoteFoundException | 访问一个不存在的文件时，发生的异常 |
| ClassNotFoundException | 加载类，但类不存在时发生的异常 |
| EOFException | 操作文件，到文件末尾发生的异常 |
| IllegArgumentException | 参数异常 |

为什么要处理异常？

如果程序抛出了异常，交由JVM处理，程序就会崩溃，停止。对于一个大的项目来说，如果一出现一下小异常就导致程序无法运行的话，程序的健壮性就不够好。所以我们要自己处理异常，以增强程序的健壮性。

总之：不应该出现一个不算致命的问题，就导致整个系统崩溃。

Java设计者提供了一个叫异常处理机制来解决这个问题。

异常有哪些种类，可以举几个例子吗？（指导：先回答分类，之后举几个自己经常看到的例子）



throw 和 throws 有啥区别？直接 try catch 不好吗，为啥还要抛呢？（指导：有时候自己无法处理，必须得让调用该方法的人来处理，于是得用抛出）

throws放在方法形参列表括号后面，方法体之前，用于声明该方法要抛出的异常类型，不需要new对象。作用是声明该方法会抛出哪些异常。

throw用于创建异常对象，而throws和try-catch都是解决出现异常的机制。

为什么要抛出：

throws解决办法是将异常抛出，交给别人解决，这在有时自己无法处理时会抛出，比如创建一个文件输入流，FileInputStream就会抛出一个FileNotfoundException编译时异常，要求调用者必须处理。FileInputStream类之所以不自己处理是因为不同的人调用该方法出现问题后解决方式可能不同，如果用try-catch就写死了，不利用程序的拓展，这样写的程序不灵活。所以FileInputSream会将该编译时异常抛出，有调用者解决，可以满足不同的人定制化的需要。

throw在方法体内部，用于具体抛出的异常对象，需要new一个异常类对象。

throw抛出的的异常如果是编译时异常，必须使用throws将throw抛出的异常类型或者其父类抛出。

|  |  |  |
| --- | --- | --- |
| Throw创建的东西 | Thrwos行为 | 为什么 |
| Error及其子类 | 可以抛，也可以不抛 | 因为Error是很严重的错误，程序员也无法通过针对性的代码捕获解决 |
| RuntimeException及其子类 | 可以抛，也可以不抛 | 在运行阶段才会出现，编译阶段可以不处理 |
| 编译时异常：Throwable，Exceptiton，Exception的子类中不属于RuntimeException子类的类 | 必须抛，且必须是Throw抛出的异常类或者其父类(多态) | 在编译阶段就出现，所以必须处理。 |

Throws是一种声明，告诉调用者， 我这个方法会抛出哪些异常

Thrwo是一种行为，真的抛出一个具体的异常。

对于运行时异常和错误，你声不声明都无所谓。因为一个调用者可以选择不解决，一个调用者无法解决。

对于编译时异常：必须声明了，因为要求调用者解决，既然你要求人家解决，总得告诉人家你对异常是什么把。

Throws是把异常抛出去，交给外部处理，如果外部没有try-catch，直接Throws，那么最初Throw的对象就会交由JVM处理。

Throw用于创建异常，Throws是处理异常的机制，处理方式是将异常抛出。

一个是创建异常，一个是抛出异常。

throw一个异常，如果没有catch的话，就会像return语句那样，throw那里就会停止运行。所以finally会在return之前。

try中有错误，但是没有catch，会在错误抛出之前执行finally

证明对于运行时异常，如果程序员没有try-catch，默认会抛出？

证明：如果出现1/0的代码，不try-catch，程序会一直向外抛，知道交给JVM处理，JVM会中止程序，打印异常。

finally中的代码执行时机？

方法马上就要结束之前，很短的时刻：

1. return紧挨着之前
2. 抛出异常紧挨着之前

注意这个时机，有时候JVM会对i++，++i等语句进行优化。

为什么运行时异常不会要求程序员显示地处理？

因为JVM在遇到运行时异常的时候，如果程序员没有显示的处理，会使用throws关键字一路抛出异常，直到抛给JVM。

try catch会影响性能吗？为什么抛出异常的时候会影响性能？

try-catch-finally 中，如果 catch 中 return 了，finally 还会执行吗？（指导：finally是异常的地方，可以说一说

会执行。finally的代码会才catch return之前执行。finally是异常不管发布发生，都会执行，而且会在方法放回之前执行finally里面的代码在执行return。所以finally里面一般存放释放资源的代码

try-catch-finally执行的细节？

1. try中的发生异常的语句，后面的代码不会执行
2. catch中的代码只有发生异常才会执行
3. finally不管有没有异常，都会执行，且都会在return结束之前先去看有没有finally，先执行finally里面的代码在return
4. 如果finally里面有retrun，一定是return final的值，try和catch里面的return都作废。

1

catch (Exception e) {

return 2;

} finally {

return 4;

}

JVM底层优化:

catch (Exception e) {

return 4;

} finally {

;

}

2

catch (Exception e) {

return ++i;

} finally {

return ++i;

}

JVM底层优化

catch (Exception e) {

++i;

} finally {

++i;

return i;

}

3

catch (Exception e) {

return ++i;

} finally {

++i;

}

JVM底层优化

catch (Exception e) {

++I;

temp = i;

return temp;

} finally {

++i;

}

throw VS throws？

|  |  |  |  |
| --- | --- | --- | --- |
| 关键字 | 意义 | 位置 | 后面跟的东西 |
| throws | 异常处理机制的一种 | 方法声明处 | 异常类型 |
| throw | 创建异常 | 方法体内部 | 异常对象 |

说一说你对异常的理解？

Java程序编译执行过程中会遇到一些问题，这些问题主要有两类:

1. 错误Error，这一般是JVM无法解决的严重的问题。如栈溢出错误StackOverflowError和内存溢出错误OutOfMemoryError
2. 异常Exception，这一般是值由编写程序或外部因素引起的一般性问题。可以用采用针对性的代码解决。如空指针异常NullPointerException，类型转换异常ClassCastException，数组下标越界异常ArrayIndexOutofBoudsException

数字格式异常NumberFormatException，算术异常ArithmeticException。异常主要分为两类：编译时异常和运行时异常

在程序运行过程中难免会遇到一些问题，如果JVM默认处理会停止程序。对于一些不是很严重的问题，就停止程序运行，会影响项目的健壮性。尤其是一个大项目，停止重新运行的代价太大。所以，Java提出异常的概念，并让程序员可以捕获这些异常，让程序继续运行，不至于崩溃。

异常有哪些种类，可以举几个例子吗？（指导：先回答分类，之后举几个自己经常看到的例子）



|  |  |
| --- | --- |
|  | JVM默认处理方式 |
| Error, RuntimeException， RuntimeException子类 | throws |
| Throwable，Exception，Exception子类中不是RuntimeException的子类的类 | JVM无默认处理方式，所以需要程序员显示的进行处理 |

throw 和 throws 有啥区别？直接 try catch 不好吗，为啥还要抛呢？（指导：有时候自己无法处理，必须得让调用该方法的人来处理，于是得用抛出）

|  |  |  |  |
| --- | --- | --- | --- |
| 关键字 | 意义 | 位置 | 后面跟的类型 |
| throw | 创建一个异常对象 | 方法体内 | 异常对象 |
| throws | 异常处理机制的一种 | 方法声明处 | 异常类型 |

当出现一个异常，可以通过try-catch直接在方法里面处理，但有时候创建方法的人不知道怎么处理，或者每个调用者遇到这个问题的处理方式是不一样的，没有统一的处理方式，这时候就需要throws关键字了，将问题抛出，交给调用者解决。比如new一个文件输入流FileInputSteam时，该构造方法会抛出一个FileNotFoundException的编译时异常，由于不同的项目遇到文件找不到异常由不同的处理方式，FileInputStream不知到具体的处理方式，所以没有使用try-catch，而是使用thrwos关键字，交给调用者自行处理。

try catch会影响性能吗？

几乎不会。

1. 当程序没有异常时：try-catch加不加对程序的性能都没有影响
2. 当程序有异常时：try-catch对程序性能有轻微的影响。处理异常的catch语句在class文件中是用异常实现的



|  |  |  |
| --- | --- | --- |
| 字段 | 行号 | 含义 |
| From | 6 | try块开始 |
| To | 10 | try块结束 |
| Target | 11 | catch块开始 |
| Type |  | 异常的类型 |

如果try中有异常，会跳转到Target对应的行号

如果try中没有异常，会通过一条goto指令，跳转到try-catch块后面的代码运行。

通过上面的分析，如果有异常，会查询异常表进行跳转，所以对性能有影响。

try-catch-finally 中，如果 catch 中 return 了，finally 还会执行吗？（指导：finally是异常的地方，可以说一说

finally会在马上要结束之前运行，比如在return马上要退出方法之前执行，以及出现异常，没有catch，方法马上要退出之前也会执行finally

1. 如果finally中由return，程序会从finally中的return出口出去。
2. 如果finally中由异常，会覆盖try，catch中抛出的异常。

finally异常覆盖和return覆盖

会的。无论在try中有未捕获的异常还是return语句，都会在方法真正要结束之前执行finally里面的代码。这里需要注意的是finally里面如果有异常会覆盖try里面的异常。如果finally里面有return语句，也会覆盖try里面的return语句，直接从finally里面的return出去。

1、如果try中抛出了异常，但是没有用捕获，此时finally里面的代码也出现了异常，那么finally里里面的异常会覆盖try中的异常。成为这个方法最终抛出的异常。

2、如果try中有return语句且finally里面也有return语句，如果程序没有异常，方法会从finally里面的return语句出去。

String 为什么要设计为不可变类？（指导：可以从安全性，性能等方面来考虑）

主要是出于线程安全和性能两方面考虑。

1. 线程安全：由于String是不可以变得，多线程共享同一个String时不用担心它的同步问题。性能体现在缓冲哈希值和设计常量池上。
2. 性能：

2.1 缓冲哈希值，String类有一个int类型的属性hash，默认值是0。用于缓冲该字符串的哈希值。调用hashcode，程序会判断hash是否为0以及字符串长度是否大于0。如果两个条件都满足，说明这是第一次计算hashcode的值，且字符串不是空串，所以会根据字符串内容计算字符串哈希值，并缓冲到hash这个属性。当第二次调用时，就会直接返回hash值，因为此时hash值不等于0，不满足条件。试想，如果把String类改成可变的，那么每次计算hashcode值，程序都要计算一次，因为有可能字符串已经变了。

缓冲哈希值的作用：缓冲哈希值使得String类适合用来作为Map的Key，可以快速获得哈希值，提高查询效率。

* 1. 字符串常量池设计。字符串常量池的设计可以让内容相同的字符串共享同一片内存空间，以达到节约内存的作用。两个不同的String类对象，如果他们的字符串内容相同，最终指向的还是同一个内存地址。因为String类中并不会真正存储字符串，而是存储一个字符数组引用变量value，这两个String类对象中的value最终指向的是同一片内存区域，这样相同的字符串指向在内存中创建一次。如果String类可变的，那么这个产量池的设计就不合理了，因为两个类的对象指向同一片内存空间，又可能对方会修改这片内存空间的内容。

String a = new String(“aa”) + “bb” “这句话创建了多少个对象？为什么？（指导：没啥好说，这种笔试考的多）

1、public StringBuilder() {

super(16);

}

2、调用StringBuilder类的构造函数，创建一个StringBuilder类对象实例sb

3、”aa”字面量会在常量池中创建一个String类对象。

4、new String会在堆中创建一个String类对象，value属性指向常量池中内容为”aa”的String类对象的value。

6、sb.append(“aa”)

6、”bb”字符串字面量会在常量池中创建一个String类对象

7、sb.append(“bb”)

8、sb.toString() 🡺 该方法会new 一个String类对象

所以整个过程创建了一个StringBuilder对象。四个String对象

String类对象在堆中和在常量池中有什么不同吗？

没有什么不同，都是String类对象。

value是一个字符数组，这个数组的创建由JVM负责。

数组一定是在堆中。

所以常量池中的String类对象，最后也指向的是堆中某个字符数组。

String s1 = “aaa”; 看到这个”aaa”字面量，在编译阶段，JVM就会去常量池中查询有没有内容为”aaa”的String类对象，如果没有，就在常量池中创建一个String类对象，将其value指向堆中的一个字符数组[‘a’,’a’,’a’]，这个字符数组也是JVM在编译阶段就创建的。

为什么是编译阶段呢：证明：在编译阶段就可以用字符串字面量调用函数。

然后将用引用变量s1指向常量池中的这个String类对象。



String s2 = new String(“aaa”);

String(String original){

this.value = original.value;

this.hash = original.hash; // 哈希缓存机制

}

JVM会在什么时机创建String类中value的字符数组？

出现字符串字面量的时候。

如果是硬编码在程序里面，JVM在编译阶段就会在常量池中创建String类对象，并在堆中创建字符数组，让常量池中的String类对象的value属性指向这个堆中的字符数组。

String 对象最多可以存放多少个字符（长度）？（指导：可以从源码角度分析勒）

* 1. String中存放字符串的是char类型的字符数组。其长度为int类型，int类型是4个字节，占32位，由于int是有符号的，所以去掉一位符号位，最大可以表示的整数是2^31-1。所以String的理论长度位2^31 -1
  2. 一个char在Java中占2b，2b \* (2^31 - 1) = (2^32 – 2) b; 2^32b = 2^22Kb=2^12M=2^2G = 4G，理论上，一个长为2^31 -1的Stirng类对象大约占用4G的空间
  3. 不过，由于JVM堆内存限制，编译时，String类的长度最多是2^16-2，运行时长度最多是2\*31-1。
  4. 所以在编译时可以定义较短的字符串字面量，运行时再进行拼接。

字符串常量池是放在堆中吗？

不是，放在方法区中，当然，方法区只是一个概念上的区域，不同版本的jdk有不同实现。实际上，jdk1.8之前，字符串常量池位于永久代中，jdk1.8及以后

字符串常量池位于一个叫Metaspace的空间中

jdk1.8和java8这种命名感觉很奇怪，其历史是什么？

最初Sun公司发明Java，版本定位1.1, 1.2一直到1.5，Sun公司突然宣布采用java 5来取代前面1.x的命名方式

后来，Orcale公司收购了Sun公司，继续开发Java，并在Java的第8个版本Java8，决定同时使用两套命令规则，也就是java8等价于jdk1.8。

String中 “+” 和 StringBuilder中的 append 会有性能上的差别吗？

1. 当两个字符串字面量相加时，利用+法运算符性能更高。因为，JVM只会创建一个String类对象，内容是两个字符串字面量相加的结果。
2. 如果+运算符两边有一个是变量，+法运算的性能要更差，因为这个过程首先会创建StringBuilder对象，接着常见两个String类对象，然后调用append方法。将两个字符串连接，最后调用toString(), 而toString又会创建一次String类对象。所以当+两边有变量时，本质上还是会调用StringBuilder的append方法，所以性能不如直接使用StringBuilder的append方法。

聊一聊你对多态的理解？

多态可以理解为事物运行时不同的状态，在Java中具体值，通过动态绑定，在运行时，根据对象的实际类型来调用对应的方法。多态可以通过继承或接口实现。拿继承来说：父类引用指向子类对象，如果子类重写了父类中的方法，父类引用在调用方法时，实际运行过程中还是执行的子类方法的版本。

运行时多态

1. 多态的定义：多态就是指事物有多种形态。
2. 我们说多态一般是指运行时多态，子类继承并重写父类的方法，调用方法时，在运行阶段根据对象的实际类型才能确定调用方法的版本。运行时多态的实现原理时，程序启动后，JVM将字节码指令加载进内存，通过invokeVirtual指令和方法表实现。方法表里面存放了方法实际的入口地址。invokeVirtual 指令会根据方法接受者的具体类型选择执行方法的版本。
3. 除了运行时多态，我还听说过一个争论比较多的编译时多态，由方法重载实现。多个同名的方法，根据参数的类型、顺序、个数进行重载，在编译阶段根据实参的静态类型决定调用方法的版本。

多态的字面意思是同一个事物有多种形态。

在Java中，多态是时指当父类引用指向子类对象，或者接口类引用指向实现类对象时。无法在编译阶段确定方法这个引用变量调用的方法是哪个版本，只有当程序运行起来，才能根据具体的类型确定调用的方法的版本。

（总述）多态，顾名思义，就是一个事物有多种形态。

（分述）在Java中，多态有编译时多态和运行时多态。：

编译时多态本质上就是方法重载，同一个类中有多个同名的方法，这些方法的参数类型、个数、顺序不同。为什么叫编译时多态呢，因为在编译阶段，JVM根据参数的静态类型决定使用哪一个版本的方法。编译时多态提供了一种灵活的方式实现相似功能。

运行时多态通过继承和方法重写override方式实现的，方法重写是子类重新定义并覆盖父类中的方法。重新要求子类方法和父类被重写的方法同名同参。为什么叫运行时多态呢，因为在运行阶段，JVM将字节码加载进内存，执行字节码中的inovkevirtual指令，该指令会根据方法调用者的实际类型，决定执行的方法的版本。运行时多态降低程序模块之间的耦合性。

1. 多态的定义：同一事物，不同形态
2. 运行时多态的定义：在Java中，运行时多态，通过动态绑定，在运行是根据对象实际类型来调用对应的方法。
3. 运行时多态怎么实现：运行时多看由继承或接口实现。拿继承来说，子类重写父类所有的方法，当用一个父类引用变量指向子类对象时，用这个父类引用调用方法，在运行时实际执行的是子类重写版本的方法。
4. 运行时多态实现机制：运行时多态(重写)的实现机制是invokVirtual指令和方法表，在运行阶段，JVM将字节码加载进内存中，对于加载的类有一张放发表，存放了方法的实际入口地址，字节指令中有一个invokeVirtual指令，该指令会根据方法接受者的实际类型查询到方法的实际入口地址，然后执行方法。
5. 我的项目中使用到了多态：社区项目用到多态的地方。
6. 编译时多态有争议：我还听说过有争论的编译时多态，指在编译阶段，同一个类中有多个同名的方法，但方法参数的类型、顺序、个数不同。JVM根据实参的静态类型选择方法执行的版本。

运行时多态实现机制？

在方法区中，存放了方法的元数据信息，调用方法就是根据方法区中的信息选择方法的版本。

JVM说明文档对于invokevirtual指令的解释：

Invoke instance method; dispatch based on class

方法表：存放方法实际入口地址



方法表中的方法如果被本类实现了，就会执行本类的实例对象，如果没有实现，就会向上查询，如果父类实现了，就会执行父类的实例对象。

通过方法表来寻找方法的入口地址，即方法表里面存放了方法的入口地址。通过invokeVirtual index指令来调用方法。通过这些机制一块来实现运行时多态。

方法表里面存到是该方法实际的入口地址，这个入口地址是最终实现了这个方法的那个类的实例。

Java transient关键字是什么？

该关键字放在成员变量前，声明该成员变量不应该序列化。尤其当对象需要网络传输时，一些私密信息或者不必要的大对象都可以用该关键字修饰，放在它们序列化，并被传输到网络中。例如，StringBuffer类有一个toStringCache成员变量，用于存储每次toString方法调用时的缓存，所以该变量是不必要的大对象，前面加上了transient的关键字，声明不应该序列化。

聊一聊StringBuffer类？

java.lang.StringBuffer代表的是可变字符序列，可以对字符串内容进行增删改查。

StringBuffer类声明代码是: public final class StringBuffer extends AbstractStringBuilder implements Serializable, CharSequence{…}

从声明代码中可以得到以下结论:

1. StringBuffer是final类型，不允许被继承
2. StringBuffer继承抽象类AbstracStringBuilder，该类有一个char[] value属性用于存储字符串，所以StringBuffer类是可变的。
3. StringBuffer类实现Serializable接口，可以序列化，用于保存文件或网络传输，除此之外，由于可以网络传输和保存文件， StringBuffer toStringCache成员变量是用transient修饰。用该修饰符修饰的变量不会被序列化。因为otStringCahe是不必要的大对象，里面存储的是每次toString的缓存。
4. StringBuffer类对象没有常量池的概念，因为StringBuffer类是可变的，如果多个对象共享同一个常量池中的对象，会造成同步的问题，所StringBuffer类对象没有常量池的概念。
5. 更新StringBuffer，向char[] value里面添加数据，不用每次都更新地址（即不是每次都创建新对象）。所以效率高于String

String、StringBuilder、StringBuffer的区别？

StringBuffer和StringBuilder非常类似，均代表可变字符序列。而且方法也一样。

String是不可变序列，效率低，但是复用率高（内存利用率高），节约内存，相同内容的字符串共享常量常量池种的同一块内存，比如说在常量池有一个字符串叫“aaa“，不管有多少个变量都会指向同一个”aaa”，如果变化了，再再常量池中创建一个新对字符串常量对象即可。

StringBuilder和StringBuffer是可变序列，效率高，StringBuilder线程不安全。StringBuffer线程安全。

如果我们对String做大量修改，不要使用String。因为每次修改本质上是在内存种创建了一个新对象，然后重新指向这个对象的地址，原来的对象就变成了垃圾，所以如果在一个循环种对String类进行修改，会产生大量垃圾对象。极大影响程序性能。

效率上来看：StringBuilder > StringBuffer > String

StringBuffer类和StringBuilder类其实就是对String类的一种增强。

StringBuffer类和StringBuilder类存储的是可变字符串序列，证明：这两个类都继承自AbstractStringBuilder，该类有一个char[] value 的成员变量，用于存储字符串，而这个变量没有用final修饰，所以是可变的。由于没有访问修饰符修饰，所以默认是包私有级别，lang包之外无法直接访问value属性。

先

|  |  |  |  |
| --- | --- | --- | --- |
|  | String | StringBuffer | StringBuilder |
| 父类 | Object | AbstractStringBuilder | AbstractStringBuilder |
| 是否实现串行化接口 | 是 | 是 | 是 |
| 是否有常量池 | 是 | 否 | 否 |
| 是否可变 | 否 | 是 | 是 |
| 是不是线程安全 | 是 | 是 | 否 |
| 每次更新 | 更新地址，效率较差 | 更新内容，不用每次更新地址，效率较高 | 更新内容，不用每次更新地址，效率较高 |
|  |  |  |  |

没有用访问修饰符修饰，默认的访问范围？

是同包下可访问，包私有。如果一个类、方法、变量没有访问修饰符修饰，那么这个类、方法、变量的访问权限是同包下的类可访问。

聊一聊Serializable接口？

1、该接口是一个标记接口，作用类似于注解。本质是声明该类可以序列化和反序列化，具体怎么序列化和反序列化，由JVM来实现。

2、Java中io包下的ObjectOutputStream和ObjectInputStream可以将一个对象序列化和反序列化。

3、如果一个类声明了序列化，其属性中包含对象，那么这个对象也应该声明序列化。

路径中正斜线/ 和反斜线\有什么区别吗？

|  |  |  |
| --- | --- | --- |
| 平台 | 路径分隔符 | 举例 |
| Linux | / | /usr/local/nginx |
| Windows | \ | D:\Java\projects\school |
| Java | / |  |

String类 vs StringBuffer更新效率？

String类里面存储的是字符串常量，里面的值不能修改。每次String类更新实际上是更新地址。所以效率较差。

StringBuffer类里面存储的是可变字符串，里面的值可以修改。每次更新StringBuilder实际上更新的是内容，所以效率较高。

StringBuffer的构造器有哪些？

|  |  |
| --- | --- |
| 构造器 | 说明 |
| StringBuffer() | 构造一个不带字符的字符串缓冲区，初始容量为16。 |
| StringBuffer(CharSequence seq) | 这里用到了接口多态。 |
| StringBuffer(String str) |  |
| StringBuffer(int capacity) |  |

StringBuffer()构造一个不带字符的字符串缓冲区，初始容量为16？

证明：StringBuffer stringBuffer = new StringBuffer(); 新建一个空的StringBuffer对象，进入无参构造函数，

public StringBuffer() {

super(16);

}

无参构造函数会调用父类的构造函数，传入16，代表capacity

AbstractStringBuilder(int capacity) {

value = new char[capacity];

}

父类的构造函数最终会新建一个大小为capacity（16）的字符数组。

StringBuffer(String str)这个构造器源码追踪?

StringBuffer stringBuffer2 = new StringBuffer("aaa");

public StringBuffer(String str) {

super(str.length() + 16);

append(str);

}

super(3 + 16)，创建一个长度为19的字符串缓冲区

然后将字符串append到value中

String和StringBuffer相互转换？

String 🡺 StringBuffer

方式一：构造器

方式二：空构造器+append 方法

StringBuffer🡺 String

方式一：toString

方式二：构造器

StrinBuffer常见方法？

append

delete(start,end) [start,end)

replace(start,end) [start,end)

indexOf

在System.out.println(对象)的时候，会自动调用对象toString方法吗？

不会。具体的做法：

1. 如果对象是String类型，会单独有一个方法。

public void println(String x) {

synchronized (this) {

print(x);

newLine();

}

}

1. 如果对象不是String类型：

public void println(Object x) {

String s = String.valueOf(x);

synchronized (this) {

print(s);

newLine();

}

}

会先用String.valueOf方法将x转为String类型

String.valueOf会看x是否为空，如为空：返回”null”。

如果不为空，返回x.toString方法

最终都会调用print(s)，

public void print(String s) {

if (s == null) {

s = "null";

}

write(s);

}

这里还会判断一次字符串是否为空。

结论:System.out.println里面传入的对象并不是直接调用toString，因为可能存在空的情况，已经传入的本身就是String类对象。

用String.valueOf，是因为有可能对象是空的，而该方法会考虑这个情况返回”null”，如果不是空的，调用toString方法。

System.out.println源码追踪，到底怎么实现任何对象输出字符串？



StringBuilder append(对象)源码追踪？

StringBuffer sb = new StringBuffer();

sb.append(对象);

这里如果里面传入的是类数据类型，append重载了两个版本

append(Object obj)

1. 如果对象是String类型，会调用append(String str)

public synchronized StringBuffer append(String str) {

toStringCache = null;

super.append(str);

return this;

}

会直接调用父类的append(String str)。

if (str == null)

return appendNull();

父类的append放啊有一个判断语句，判断str是否为空，若为空专门有一个appendNull函数，然后结束。

value[c++] = 'n';

value[c++] = 'u';

value[c++] = 'l';

value[c++] = 'l';

appdenNull方法会插入”null”

1. 如果传入的对象不是String类型，会调用append(Object obj)

然后调用String.valueOf(obj)

下面调用父类的append反方法，于第一种情况调用的方法一样。

而String.valueOf(obj)：如果obj为null，则返回”null”。如果obj不为空，则返回obj.toString();

所以，StringBuffer append(对象)方法调用的结果：

1. 对象为空，”null”
2. 对象非空：

2.1 “” 什么都不添加

2.2 其它，添加相应的字符

所以append可以传入null，但是new StringBuffer(String str)不能传入null，因为要调用str.length来确定value的初始长度。如果str是null，会报空指针异常。NullPointerException.

聊一聊StringBuilder类?

StringBuilder也是一个可变字符序列，它提供了与StringBuffer兼容的API，但不保证同步。该类设计出来是用作StringBuffer类的简易替换，用在字符缓冲区被单个线程使用的时候。如果是单线程访问的情况，优先使用StringBuilder，因为它在大多数实现上都比StringBuffer快，证明：

如果是多线程，要使用StringBuffer或String类，因为它们是线程安全的。虽然实现线程安全的方式不一样。

String，StringBuilder，StringBuffer该如何选择？

1. 如果存在对字符串大量修改，使用StringBuilder或StringBuffer：

如果是在多线程环境下，使用StringBuffer，因为StringBuffer可以保证线程安全。如果实在单线程环境下，使用StringBuilder，因为StringBuilder方法的实现大部分情况下更快。

二、如果对字符没有大量修改操作，且被多个对象引用，使用String类，内存的复用率较高。

String中 “+” 和 StringBuffer 中的 append 会有性能上的差别吗？

有。String中的加法运算符操作效率远低于StringBuffer的append，尤其是对字符串做大量修改操作时，修改的量越大，这种差距越明显。这时因为String类是不变类，每次修改String类本质上是新创建了一个String对象，然后将将引用指向新对内存地址，所以效率很低。而StringBuffer类对字符修改时，是更新内容，不用每次都更新地址。所以效率比加法操作高。

StringBuilder、StringBuffer有什么区别？

StringBuilder是用于对StringBuffer的简易替换，它用于与StringBuffer兼容的API。StringBuilder和StringBuffer一样，存储的可变字符序列，【在对字符进行增删改查时，不用每次都创建新对象。】

但是StringBuilder不是线程安全的。这时因为对字符串修改时比如append方法没有像StringBuffer类那样，使用synchronized关键字进行同步。所以，在多线程情况下，使用StringBuffer。但在单线程的情况下，优先使用StringBuilder，因为其效率比StringBuffer更高。

什么是序列化？什么情况下需要序列化？序列化在Java中是怎么实现的？

在Java中，序列化是指将一个对象变成字节流。

在网络传输或者写文件的时候，需要将对象序列化。因为操作系统或者网络传输协议无法解析对象这种高级数据结构，需要将其转化为字节流，使其可以写入文件和进行网络传输。在内存中，对象是动态的，出了内存（进入文件，网络传输）需要将对象转换为静态的字节流。

在Java中，如果一个类想要其实例对象能够被序列化，需要实现一个标记接口Serializable，无需重写任何方法，只需声明即可。

1. 什么是序列化：在Java中，序列化是指将对象转换为二进制字节流
2. 什么情况下需要序列化：在对对象持久化时，需要将对象序列化成静态的二进制字节流，然后保存到文件或数据库中。在进行网络传输时，也需要将对象转换为字节流才能在网络中传送。
3. 怎么实现序列化：首先，对于某个类，如果希望自己的实例对象能被序列化，该类应该首先一个标记接口Serializable接口，无需重写任何方法。然后Java中的序列化与反序列化实际上由ObjectOutputStream类中的writeObject方法和ObjecInputStream类中的readObject方法实现的。

什么是动态代理？有什么用？Java中可以怎么样实现动态代理？

反射的问题也可以往这上面带。

如何使用spring?

spring本质上也是用java语言写的很多的java文件，变成class文件后打成jar包，需要用到spring哪些功能，就引入对应的jar包就行了。而引入jar包的方式有两种:

1. 手动引入，手动下载然后引入
2. maven引入

现在基本上都使用第二种。

|  |  |
| --- | --- |
| 使用spring的功能 | 需要引入的最小jar包 |
| ioc | spring-context |
| aop | spring-context, aspectjweaver |
|  |  |
|  |  |

引入spring-contextjar包会自动引入：其它5个包。



对于AOP功能：其实spring-context已经包含了aop的jar包，但是pointcut excution表达式解析成对应的方法会使用到eclipse发布的aspectjweaver jar包)

新建applicationContext.xml文件，去官网查看xml头。

Spring依赖注入本质？

OrderService orderService = OrderService.getInstance(); //使用单例模式获得一个对象

这种方式在测试时真的回去创建一个OrderService对象，去做一些事情，比如连接数据库等。但又不想在测试环境下做这些：

setOrderService(OrderService orderService )

可以传进来一个mokcOrderService来模拟真实的OrderService

但此时需要把OrderService 变成一个接口或者抽象类，才能打对象传进来。为什么呢：因为接口对应实现，抽象类对应继承。这时实现多态的两种方式。

面向对象编程的要求：对接口编程，而不是对实现编程。

Spring ioc xml文件获取bean的过程？

首先在xml配置文件中指定需要创建bean的类的完全限定名和bean id，然后使用ClasspathXmlAplicationContext类加载该并解析该xml配置文件。spring会读取要创建的bean的类名，利用反射机制获取到该类的class对象，然后调用该类的构造函数创建实例对象。

1. 解析xml，获取各个元素
2. 通过Java反射机制把各个bean的实例创建，（创建好所有实例之后才会进行下一项）
3. 通过Java反射机制调用setter和getter方法，实例注入进去

反射的第一个用途：Sprint IOC创建bean实例对象

什么是代理模式？

1、代理模式是设计模式的一种，指自己无法实现的功能代理给被人去做。

2、代理模式的原理是代理类通过实现和委托类相同接口，达到对委托类功能扩展的目的。



3、代理模式可以分为静态代理模式和动态代理模式。这两者的区别是代理类的出现时机，对于静态代理模式，代理类在编译阶段就已经实现好，然后编译成class字节码文件。对于动态代理模式，代理类在运行阶段，根据加载进来的委托类，动态的生成字节码文件然后生成代理类，实现与委托类同一个接口。

4、静态代理模式的应用：静态代理模式有一个很经典的应用就是用Runnable方式创建多线程，委托Thread类启动线程。委托类和代理类Thread会实现同一个接口Runnable。Thread有一个 Runable类型的成员变量，变量为target。由于实现了Runnable接口，Thread类需要重写run方法，里面的实现是条约target的run方法。在创建线程时首先创建委托类对象，然后new Thread类对象，把委托类对象作为参数传递进去，target会指向委托类对象。然后调用Thread类对象的start方法，该方法会调用start0方法，start0方法是natvie修饰，表示由本地语言实现，该方法会启动一个线程并调用run方法。而由于target指向的对象是委托类对象，根据运行时多态，所以在运行阶段，实际上运行的是委托类里面的run方法

5、动态代理有两种实现方式：JDK实现，CGLib实现。

JDK实现：

java.lang.reflect包下，有一个Proxy类。可以通过getProxyInstance静态方法获取代理对象，该方法接收三个参数

newProxyInstance(ClassLoader loader,Class<?>[] interfaces,InvocationHandler h)

参数1 ClassLoader loader, 用于指定一个类加载器，通常是当前类的类加载器，即AppClassLoader 应用类加载器。

参数2 Class<?>[] interfaces, 代理类要实现哪些接口

参数3 ,InvocationHandler h，生成的代理对象要干什么事情，这是reflec包下的一个接口，我们new 一个匿名内部类来实现：

new InvocationHandler() {

@Override

public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {

return null;

}

}

new一个匿名对象，该匿名内部类需要重写invoke方法。inovke方法是jdk代理的重点，因为代理干了什么是实际上是由inovke方法指定的。invoke方法有三个参数：

Object proxy

Method method

Object[] args

Invoke是一个回调方法，当我们把代理对象创建好，即有了starProxy之后，会调用starProxy.sing(“小幸运”)，starProxy.dance() 🡺 starProxy.invoke(传入三个参数)，starProxy.invoke(传入三个参数) 🡺 starProxy.invoke(starProxy,sing方法的Methd对象，sing方法参数)，starProxy.invoke(starProxy,dance方法的Methd对象，dance方法参数)

第一个参数：starProxy，当前自己这个对象传进去。

第二个参数：根据反射机制，获取外部调用这个类的方法的方法对象。

第三个参数：这个方法有哪些参数

程序为什么需要代理，代理长什么样？

1、对象如果嫌自己身上的事干得太多，可以通过代理来转移部分职责。

2、代理长得样子和对象查不对，对象有什么方法想被代理，代理就一定要有对应的方法。代理可以比委托类更多方法（功能拓展）

让中介和对象有同样的方法🡺它们实现一个接口；委托类是代理类的子类。

Method类的对象invoke方法如何实现？

public abstract java.lang.String JDK动态代理.Star.sing(java.lang.String)，这时该方法的完全限定名，inoke(obj,参数)，obj的类型必须是Star的子类或实现类，一个无关类型的类，即使实现了sing方法，也会报错。因为是根据完全限定名来查询方法的实际入口的。

JDK代理如何实现？

java.lang.reflect.Proxy类

newProxyInstance方法，该方法有三个参数：

1. 类加载器，指定动态生成的代理类的类加载器
2. 接口，指定代理类长什么样
3. InvocationHandle，指定代理类要干什么事。

JDK动态代理如何实现invoke？

public class Main{

Proxy.newProxyInstance()

}

会动态生成两个类：

public final class $Proxy0 extends Proxy implements Star{

public final String sing(String var1){

super.h.invoke(this, m3, new Object[]{var1});

}

}

final class Main$1 implements InvocationHandler{

public Object invoke(Object proxy, Method method, Object[] args)

}

通过InvocationHandler的invoke方法指定代理干事时，这个invoke本质上是被Main$1的对象调用，Main$1也是动态生成的类，他会实现InvocationHandler接口，重写invok方法，这个方法体里面的内容就是我们当时创建代理对象是里面方法的内容。

Proxy类有一个h属性，protected InvocationHandler h; 实际指向的就是Main$1类的对象。在调用starProxy的sing方法的时候，会找到父类的h属性，然后调用iovke方法。

聊一聊CGLib动态代理？

CGLib是一个强大的代码生成库。采用字节码技术，对指定目标了生成一个代理子类，并对子类进行增强，以达到对目标方法的增强。

jdk动态代理也会操作字节码，为什么它就不能通过继承实现非接口类的代理呢？

这要从JDK代理的实现机制说起了，JDK在生成代理类Proxy$1时，会让其继承java.lang.reflect.Proxy类

public final class $Proxy0 extends Proxy implements Star

这里为什么要继承Proxy类呢，是因为Proxy类里面维护了一个InvokecationHandler h属性，该变量指向了一个实现类，里面会重写invoke方法，而这个方法的内容就是我们创建代理类时写入的。在调用代理类的方法时实际上会去父类找到h这个变量，然后调用inovke方法。

这也就是为什么JDK代理无法使用继承来实现动态代理了，因为已经继承了Proxy，而java中没有多继承的概念。还有一个问题，为什么不把InvocationHandleer属性下放给代理类，而是让Proxy维护，其他类去继承呢：有一种解释：动态代理最原始的需求只是保护目标对象和增强目标对象的功能，而实际的开发模式都是面向接口开发的。所以，基于接口的实现的需求和场景都符合。因为在Java里面，类的集成关系，更多考虑共性抽象能力，从而提高代码的复用性和可维护性，而动态代理也在做相同的事。就是把所以代理类公有的InvocationHandler属性抽象到父类里面，把公共的逻辑放到Proxy里面。这也是正常的设计思路。

聊一聊2 3 树？

2表示2节点，3 表示3节点

一个n节点，可以存储n-1个数据

2🡺2节点🡺1个数据



3🡺3节点🡺2个数据





为什么用树数据结构做文件系统索引时，树的高度会影响I/O？

因为每个节点代表一个索引，而索引是存储在磁盘上的，如果拓展新一层，会去读取一次磁盘。

B树的定义？

B树是一棵满足下面条件的m(m>2)叉树

1 树种的每个节点最多有m个子节点，即最多m-1个关键字

2 关键字是有序的

m = 3, 3阶B树，2-3树

m = 4, 4阶B树，2-3-4 树

内存与磁盘交互？

内存的最小存储单位是页

为什么在windows查看一个文件时，实际空间和占用空间不同的情况？

由于文件读写数据最小单位是块



由于磁盘块是以2的整数被为大小，所以会有文件实际大小和占用大小不同的情况出现。

B plus树相较于B树改进了哪点？

B plus树的非叶子节点只存主键和指针

叶子节点存储数据

这也是为什么会有B+树

搜索树？



聊一聊HashMap?

Map接口实现类的特点：

1. Map接口与Collection接口并列存在，用于保存具有映射关系的数据。
2. HashMap存放一个key，value的是一个HashMap$Node，HashMap里有一个HashMap$Node类型的数组存放节点。Node<K,V>[] table
3. Map中的key不允许重复
4. Map中的value可以重复
5. Map的key可以为null，value也可以为null；key为空只能有一个，value为空可以有多个。
6. 常用String作为Map的key：因为String有hash缓存设计。
7. key和value之间存在单向一对一的关系，即通过指定的key，一定可以找到对应的value
8. map存放key-value数据，一对k-v是存放在Node静态内部类中，该类实现了Entry接口。

Set set = map.keySet();

Collection values = map.values();

set和values中并不是存放了真正的key和value，而是存放了引用，指向，HashMap$Node中的key和value

源码解读：

Map map = new HashMap();

map.put("no1", "韩顺平");

map.put("no2", "张无忌");

1. k-v最后是通过newNode(hash, key, value, null); 返回一个HashMap$Node

2. k-v为了方便程序员遍历，还会创建EntrySet集合，该集合里面存放的是HashMap$Node，这里使用了接口类多态。一个EntrySet集合里面存放了Entry的实现类对象。定义时类型是HashMap$Entry，运行时类型是HashMap$Node

因为Java设计者不想暴露Node，所以用Entry作为set的类型。

Entry接口提供了两个重要的方法：getKey() getValue()；

Set set = map.entrySet(); 一个set本质上是map，只是value是常量。

而此时 set.table和map.table是指向同一个地址，即两个是同一个数组



this$0是什么字段？

this$0是Java编译器生成的字段，用于存储外部类的对象引用，以便可以获取外部类的字段和方法。当内部类类使用到外部类的Father.this时，编译器会内部类中创建this$0

map的六大遍历方式：keyset, values, entrySet; 每一个有增强for和ierator两种方式，所以时3x2 = 6种遍历方式

map小结：

1. Map接口的常用实现类：HashMap Hashtable Properties
2. HashMap是Hash接口最常使用的实现类
3. key不允许重复，运行使用null。value允许重复，允许使用null
4. 与HashSet一样，不保证映射顺序，因为底层是一个hash表的形式来存存储的。数组+链表+红黑树
5. HashMap底层就是一个HashMap，所以HashMap有的性质，HashSet也有。
6. 添加相同的key，val会覆盖
7. hashmap没有实现线程同步，所以不是线程安全的。

HashMap底层机制和源码分析：

HashMap$Node[] table; 数组

class Node{

K k;

V v;

Node next;

}

next用于链表

class TreeNode{

TreeNode left;

TreeNode right;

…

}

TreeNode是红黑树的节点

综上所述：jdk1.7: 数组+链表

jdk1.8: 数组+链表+红黑树



扩容机制源码追踪：

1. Map map = new HashMap();初始化一个HashMap对象，table为null，加载因子loadFactor为默认值0.75。为什么要加载因子呢：因为HashMap扩容时并不是table真正装不下了才去扩容，而是预留了一部分空间，当table里面的对象个数达到table.size \* loadFactor时，就会进行扩容。
2. map.put(new A(),"a"); 添加key-value键值对

首先put方法会去计算传进来的key的哈希值，通过hash(key)函数

putVal(hash(key), key, value, false, true); put方法会去调用putVal

if ((tab = table) == null || (n = tab.length) == 0)；putVal方法会查看是否是第一次put对象，如果是第一次put，该条件会成立，然后扩容，调用resize()方法。

newCap = DEFAULT\_INITIAL\_CAPACITY;

newThr = (int)(DEFAULT\_LOAD\_FACTOR \* DEFAULT\_INITIAL\_CAPACITY);

DEFAULT\_INITIAL\_CAPACITY：16

DEFAULT\_LOAD\_FACTOR：0.75

newCap = 16

newThr = 16 \* 0.75 = 12

if ((p = tab[i = (n - 1) & hash]) == null) 判断完是否为第一次put时，接着会通过哈希值计算得到表的索引，然后查看该索引位置是否已经有元素：

如果没有：直接进行添加：tab[i] = newNode(hash, key, value, null);

如果有：有两种情况

第一种情况：新加入的key与该位置元素的key相等。

if (p.hash == hash &&((k = p.key) == key || (key != null && key.equals(k))))

这是判断相等的条件，如果满足，说明新添加进来的元素重复了，这是会让

e = p;

e：临时创建出来的Node<K,V> e；引用变量

p：指向前面tab[i]位置的引用变量。

然后执行e.value = value; value是传进来的参数，此次添加的value会覆盖原来元素的value。

第二种情况：新加入的元素的key与该位置元素的key不相等，即上面那个if条件不满足。此时会让新加入的元素挂在该位置元素后面，需要查看是以链表形式挂，还是以树形式挂。所以会判断else if (p instanceof TreeNode) 当前元素对象是不是TreeNode类型。做出相应的添加操作。对于链表添加，在添加后会判断if (binCount >= TREEIFY\_THRESHOLD - 1)

TREEIFY\_THRESHOLD：8

如果满足，会调用treeifyBin(tab, hash);函数，但该函数并不会直接将链表树化，而是判断(n = tab.length) < MIN\_TREEIFY\_CAPACITY)

MIN\_TREEIFY\_CAPACITY：64

如果table大小小于64，不会进行树化，而是扩容table大小。这里table大小就是指table length arrary里面非空和空的总和，扩容后的大小。

resize()扩容机制：

1. 第一次添加需要扩容，table容量为16，临界值12
2. 以后达到临界值需要再次扩容，table容量变为原来2倍，临界值也变为2倍。 16 32 64…

else if ((newCap = oldCap << 1) < MAXIMUM\_CAPACITY &&

oldCap >= DEFAULT\_INITIAL\_CAPACITY)

newThr = oldThr << 1; // double threshold

这时底层实现扩容的代码，在原来的基础上左移1位

总结:

1. 执行构造器，new HashMap()

初始化loadFactor=DEFAULT\_LOADFACTOR=0.75；

HashMap$Node[] table = null

1. 执行put方法。

public V put(K key, V value) {

return putVal(hash(key), key, value, false, true);

}

hash方法会计算key 的哈希值，具体算法是：(h = key.hashCode()) ^ (h >>> 16)

1. 执行putVal

判断重复：1、哈希值相同。2、在1的基础上，两个key是同一个对象，或者，两个key不是同一个对象，但equals相同。

最主要是putVal里面有很多流程

树化

剪枝—删除节点，树越来越小，当树小的时候，会把树转回列表

table.length table的长度 table.length > MIN\_TREEIFY\_THRESHOLD

table.size table的大小 table.size > threshold

HashMap删除元素机制：

往HashMap里面添加元素，达到一定条件会扩容，树化。那么删除元素会不会做逆操作呢：

1. table.length 删除元素后不会变
2. 红黑树在链表减少后剪枝，退化成链表
3. 在删除元素，红黑树退化成链表时，变回Node类型

如果一条链表上有8个节点，再插入节点时，会调用treeifyBin

代码模拟HashMap扩容机制：

hashcode值会节点落在table的那个索引位置上

table并不会无限制的扩容，而是当tab.length>=MAXIMUM\_CAPACITY 时，就不会再扩容了，而是直接返回oldTab，后面继续put不会在扩大table length，如果索引有冲突，就往链表或红黑树上挂。

HashMap计算哈希值的hash函数设计优化？

h = key.hashCode()

h ^ (h >>> 16)

首先会拿到key的hashcode值，这是int类型，32位有符号整数。

h>>>16 无符号右移动，左边总是补0

h

0001 1110 1011 1110 0101 0001 0110 0001

0000 0000 0000 0000 0001 1110 1011 1110

-----------------------------------------------------

0001 1110 1011 1110 0100 1111 1001 1111

让h的高16位与低16位进行异或

这种计算方式有两点好处：

1. 位运算操作快，可以使算法更高效
2. 可以让ash更加随机，减少哈希冲突

在得到索引位置时使用(n - 1) & hash，也是采用位运算，比取模操作更快。

(n - 1) & hash 的结果是小于等于n-1且小于等于hash的。

jdk1.7 HashMap 头插法？

jdk1.7的HashMap在多线程put的情况下，会产生循环路径依赖问题

jdk1.7 HashMap底层的数据结构是 数组+链表 HashMap会给每一个key计算一个hash值，然后利用hash值计算得到数组下标索引。索引位置不同的元素放在数组不同位置，索引位置相同的元素放在同一个索引位置的链表中，最新的元素放在链表头部，因为采用头插法。

问题出在transfer方法，该方法是在数组扩容时调用，扩容时会new一个新的数组，数组大小为原来大小的两倍。需要将旧表里面的数据转移新表里面。下面是transfer方法的源码：

for (Entry<K,V> e : table) 遍历旧数组

每一个位置就是一条链表

while(null != e) 遍历链表

next = e.next;

e = next;

得到每个位置，每条链上的每个结点。

计算结点在新数组的下标索引位置

e.next = newTable[i];

newTable[i] = e;

这两句代码是精髓

问题就出现在这里

假设线程1和线程2同时进入了transfer方法

Entry<K,V> next = e.next;

int i = indexFor(e.hash, newCapacity);

e.next = newTable[i];

newTable[i] = e;

e = next;

线程1执行到next = e.next; 被cpu挂起

线程2开始执行，执行完整个代码

HashMap 了解吗？平时在什么地方使用过它呢？（说明：发现没有，我喜欢问使用场景，希望大家也是能够思考使用场景的，因为掌握了这个，你说话更加有说服力）

HashMap是什么：HashMap就是哈希表，存储key-value形式的键值对。HashMap在底层利用数组支持下标随机访问数据的特性，实现快速的对键值对进行增删查操作。

要去项目里面找。

对象🡪对象.hashcode()🡪hash🡪数组下标索引((n-1)&hash)

HashMap 底层数据结构说一下？（指导：直接说最新的即可，不需要去对比以前的版本，因为面试官也听烦了，另外在说的时候，为了你语言的严谨，一定要强调下是哪个JDK版本的哈）

在JDK1.8中，HashMap的底层数据结构时数组+链表+红黑树。HashMap利用数组下标支持随机访问数据的特性，将Key的哈希值映射到数组下标上，实现对键值对的快速访问。当发生哈希冲突时，即两个相同哈希值的Key映射到同一数组下标，HashMap采用链地址法解决，把数组的每一个位置，当成是一条链表，所有哈希值相同的元素都会放在同一个位置的链表中。由于对链表的查询是线性复杂度的，随着链表长度的增加，读取效率会下降。HashMap的解决办法是当某条链表长度大于8且当前哈希表的长度大于64时，该链表就会转换为红黑树，进一步提示性能。

在jdk1.8中，HashMap底层是用数组加链表加红黑树实现的。在构造一个HashMap后，会创建一个Node类型的数组table，指向null。Node是HashMap的内部类，用于存放key，value键值对。像HasMap里面不断put元素，当table的size达到阈值时会扩容。

当发生哈希冲突是，HashMap的解决方法是链路径法。当新添加的元素，在table里面的索引位置已经有元素时，会以链表节点形式挂在当前索引位置链表的末尾。当链表的长度大于8并且table的长度大于64时，该链表会转为红黑树，并且table里面存放的元素类型会变成TreeNode，TreeNode也是HashMap的内部类。

在JDK1.8中，HashMap的底层实现采用 数组+链表+红黑树 实现。当哈希表发生哈希冲突时，采用链地址法解决冲突。也就是哈希表中的每一个槽位，都会对应一个链表。所有哈希值相同的元素会被放到同一个槽位对应的链表中。但随着链表长度的增加，元素读取效率会下降，知道链表的长度大于8且哈希表的长度大于64，HashMap会将链表转换为红黑树。

1. jdk1.8 数组+链表+红黑树
2. 数组：HashMap利用数组下标支持随机访问数据的特性，将哈希值转换为数组索引，实现键值对的快速访问
3. 链表：当发生哈希冲突时，HashMap的解决方法是链路径法，即哈希表中的每一个槽位，都会对应一条链表。所有哈希值相同的元素会被放到同一个槽位对应的链表中。
4. 红黑树：随着链表变长，读取元素效率会下降，因为又变成线性时间复杂度了。当链表长度大于8且哈希表长度大于64时，该链表会转换为红黑树，进一步提升性能。

为什么用红黑树呢？用平衡二叉树不可以吗？或者你讲一讲他们各自的优缺点吗？

此题等价于问红黑树与AVL树的区别：

AVL树追求决定严格的平衡，即左右子树高度差不超过1。红黑树放弃追求完全平衡，但红黑树的选择次数更少，插入最多两次选择，删除最多三次旋转，所以对于搜索，插入，删除操作较多的情况，红黑树的效率要高于平衡二叉树。

平衡二叉树也可以，而且二者在进行数据查询时，时间复杂度时一样的。只是在构造平衡二叉树时会有大量的旋转操作，而红黑树的选择操作会少很多，所以构造红黑树的效率要高一些。

|  |  |  |
| --- | --- | --- |
| 操作 | AVL树 | 红黑树 |
| 查询 | 效率更高，因为更平衡 |  |
| 插入 | 最多2次旋转 | 最多两次旋转 |
| 删除 | 最多n次旋转 | 最多三次旋转 |

可以使用平衡二叉树，甚至平衡二叉树的查询效率更高。但是平衡二叉树由于追求严格的绝对平衡，即每个节点的左右子树高度差不超过1，导致插入和删除结点时要恢复平衡所作的旋转操作很多。而红黑树放弃了完全平衡，插入结点最多两次旋转，删除结点最多三次旋转，所以对于插入和删除操作较多的情况，红黑树的效率是由于平衡二叉树的。当然，红黑树也有缺点，红黑树需要额外的字段记录结点的颜色，因此会占用更多的存储空间。

为什么选择 8 之后转为红黑树呢？另外链表转为红黑树之后，还会继续转为链表吗？

如果哈希算法设计的够好，哈希值足够随机，很少会出现冲突，理想情况下链表长度达到8的概率只有千万分之一。所以，一般情况下，并不会发生链表到红黑树的转化。更多的是防止自己选取的哈希算法不好的一种保底策略。在极端情况下，仍有较好的效率

当红黑树的结点小于6时，红黑树又会转回链表，原因是数据量很小的情况下，空间和时间上链表都要比红黑树更优秀。之所以把这个阈值设为6，是因为防止元素在8附近，导致在两种数据结构频繁转换

简单描述下 put 的流程？可以说一下JDK位了效率更快，在 put 的时候，做了哪些优化不？

首先会计算key的哈希值，然后得到在数组中的索引位置，判断该索引位置是否有元素，如果没有直接插入，如果有元素，则会遍历该位置的所有元素，并比较要插入的key与这些元素的key是否相等，如果相等，要添加的value会覆盖之前的value然后返回。如果都不相等，会真正执行插入方法，插入到链表或红黑树中。最后插入后会判断是否需要扩容。

调用put方法首先会计算key的hash值，利用hash函数，然后作为参数传入putVal函数，putVal函数是真正指向添加的方法。首先该方法会判断是不是第一次添加，如果是，要进行扩容，然后通过hash值映射得到数组索引下标，查看该索引位置是否已有元素，如果没有，就直接在该索引位置添加新结点。如果该索引位置有元素，则会遍历该位置上的所有元素，如果能找到与当前要插入元素key相等的元素，则会将其value值更新并返回旧值，如果找不到与当前key相等的键值对，则需执行真正的插入操作，将其插入到链表或者红黑树中。最后判断插入后是否需要扩容。则会比较新添加的元素是否和该位置的索引元素相等，如果相等，新添加的value会覆盖当前位置元素的value。如果不相等，则会根据判断节点的类型是链表还是树，然后进行相应的插入操作。

1. 如果发现当前索引位置有元素，当前所有位置所有的键值对，如果找到与要添加的元素的key相等的键值对，就覆盖其value值并返回旧值
2. 执行完插入操作后会判断是否需要扩容

优化：在计算哈希函数时会用获取key的hashcode值，这是一个32位的有符号整数，HashMap会让这个值的低16位与高16位进行异或得到hash值。这个优化有两点好处，首先，位运算的速度很快，可以提高算法的效率。其次，使用高16位和低6位的异或可以增加哈希值的随机性，减少哈希冲突。

多线程情况下，put 是线程安全的吗？可以简单举个例子，说一下哪里不安全吗？

采取总分的形式

总：不会。7有说明问题，8解决了什么问题，但仍有什么问题

分1：7的死循环问题具体描述

分2：8的解决办法描述，以及8仍然无法解决的问题描述

不是。jdk1.7在多线程情况下，put方法会导致数据覆盖问题，以及在数组扩容的时候导致循环依赖的问题。jdk1.8解决了死循环的问题，但是没有解决覆盖问题。

在jdk1,7中，当多个线程同时进行了扩容，并调用transfer方法将旧表数据复制到新表数据时，采用头插法插入链表，这回改变链表中结点原来的顺序并导致循环依赖出现死循环问题。jdk1.8采用尾插入，保留原来结点的顺序，解决循环依赖的问题，但任然还是有数据覆盖的问题。比如两个线程同时判断某个索引位置为空，并都执行插入逻辑。结构后执行完的线程的数据会覆盖先执行完的线程的数据。

put不是线程安全的。

在jdk1.7中，链表的插入采用头插法，在对数组扩容后会执行transfer方法，重新计算结点的下标，并将旧表里面的元素移动到新表上。当两个线程同时进入transfer方法，可能会导致新表的链路中出现循环依赖从而出现死循环的问题。

在jdk1.8，在扩容复制元素到新数组由头插法改成了尾插法，在插入过程中会保留原始链表的顺序，不会出现链表成环的问题。但是任然不是线程安全的，jdk1.8在多线程调用put方法是容易出现覆盖问题，就是在两个线程同时判断某个位置为空，然后执行插入逻辑，后执行完的线程的数据会覆盖先执行完的线程。

如果我想要让 hashmap 变成线程安全的，你觉得可以怎么做？（有时候会扯到 concurrentHashMap，不过咱们这里先不追击这个）

1. synchronizedMap
2. hastable
3. concurrentHashMap(暂时不追这么深)

头插法会导致死循环，那你觉得在以前的版本中，为啥会使用头插法呢？

采用头插法的话，最新插入的数据会在链表的头部，Java设计者考虑最新的数据可能最近不就就要访问，那么就可以以O(1)时间复杂度找到这个元素并返回

那我们再说一说 HashMap 的扩容吧，什么时候会扩容呢？你觉得为啥负载因子为啥选择 0.75 呢？

创建一个HashMap，会新建一个Node数组，这个数组指向null。当第一次put数据时，HashMap会判断table是不是为空，如果为空就会进行第一次扩容。默认的大小是16，负载因子是0.75，所以临界值会变成16\*0.75，12。当数组里面的元素达到临界值的时候会扩容，扩容后数组大小变为原来的2倍，临界值也变成原来2倍。有时候，没有达到临界值也可能进行扩容，就是当某该索引位置的链表结点个数大于8时，就会进行调用treefiybin函数进行树化，该函数并不会马上执行树化逻辑，而是判断数组的长度是否大于64，如果不大于，就会再次执行resize()方法扩容，如果大于64，则会将这个链表转换为红黑树。

负载因子选0.75我想是因为 空间利用率 和 哈希冲突 的一个折中，原因是负载因子是用来计算阈值的，阈值为底层table的长度乘以负载因子，当HashMap的容量大于阈值时会触发扩容。所以，如果负载因子过小，HashMap没填几个元素就要扩容，虽然降低了哈希冲突的概率，但是浪费空间，空间利用率不高。如果负载因子过大，HashMap，提高了空间利用率，但是产生哈希冲突的概率大大增加。所以，综合各种考量选择了0.75。

1、负载因子选择 0.75是考虑空间利用率和哈希冲突之后的折中选择。

2、什么是负载因子：负载因子是hashmap底层用来计算什么时候扩容的阈值，阈值等于hashmap的长度乘以0.75，当hashmap中的元素个数大于阈值时会触发扩容。

3、负载因子过小，HashMap存入几个元素就要扩容，虽然降低了哈希冲突的概率，但是空间利用率太低，浪费内存空间。

4、负载因子过大，HashMap可以提高空间利用率，但是哈希冲突的概率大大增加。

5、综上所属，Java设计者考虑这两种因素并进行了充分的实验，选择了0.75这个值。

频繁扩容会导致效率比较低下，那你觉得在平时，在实际的开发场景中，可以怎么优化来避免频繁扩容呢？

HashMap中有一种情况会频繁触发resize函数，就是当多个元素发生哈希冲突被挂在同一条链表上，当链表的结点个数大于8且HashMap底层的数组长度小

64时，每往这个链表添加结点，就会扩容一次。所以为了避免这种情况，在自定义类中重写hashcode函数时要优化。

在实际开发中要提前预估业务的存储量，设置一个比较大的初始容量，这是不用考虑这个容量是不是2的次幂，HashMap会自己通过这个初始容量计算出一个2的次幂的容量。还可以调整负载因子的大小来减少扩容的发生。

一个场景题：只存60个键值对，需要设置初始化容量吗？设置的话设置多少初始化容量比较好呢？

需要

如果不设置初始值的话，HashMap初始化，第一次扩容默认的初始值是16，然后32，64，阈值是12，24， 48。所以，扩容到64还会扩容一次变成128.

那么HashMap最终的容量就会变成128。由于，HashMap会更加传入的初始值参数计算一个大于等于初始值的并且为2的次幂的容量。所以只需传入65到128的初始值即可。

0100 0000 0000 0000 0000 0000 0110 0001

0110 0000 0000 0000 0000 0000 0110 0001

0111 1000 0000 0000 0000 0000 0110 0001

0111 1111 1111 0000 0000 0000 0110 0001

0111 1111 1111 1111 1111 0000 0110 0001

0111 1111 1111 1111 1111 1111 1111 1111

jdk和jre的区别？

jdk： java development kit Java语言软件开发工具包。在java11之前，jdk安装目录下有一个jre目录，里面有两个文件夹bin和lib，bin是jvm的实现，lib是jvm运行java程序所需的类库

jre：java runtime environment，Java运行环境。包含JVM标准实现(bin)以及Java核心类库，JRE是java运行环境，并不是一个开发环境，所以没有包含任何开发工具（如编译器和调试器）

JVM本质上一个进程，操作系统将磁盘上的jre的bin目录下的文件加载进内存

成为一个java进程，JVM可以将java编译后的字节码文件翻译成对于操作系统的机器码指令。

jdk11文件夹下的目录：

|  |  |
| --- | --- |
| 目录 | 说明 |
| bin | 包含了所有jdk的工具目录，如java, javac, Javadoc, jarsigner |
| lib | jdk所需的库文件 |

jdk11没有包含jre目录，因为jre直接集成在了bin和lib目录里面，所以不用使用命令还去特意生成jre目录，java设计者之所以将jre集成jdk里面。就是希望开发程序和运行保持一体。所以对于tomcat，运行时要指定JRE\_HOME，如果是jdk8，指定jre目录即可，如果是jdk11，直接指定jdk安装路径即可。

什么是classpath路径？

classpath是类路径。就是java命令运行一个java编写的程序时，JVM需要将类文件加载进内存，而去那些地方找这些class文件，就是类路径。

在jdk1.5之前，classpath没有默认值，需要手动配置。通常配置的路径为：

.;%JAVA\_HOME%\lib\dt.jar;%JAVA\_HOME%\lib\tools.jar

.表示当前运行的class文件路径

还回去jdk安装路径下的lib文件夹中找dt jar包和tools jar包。

dt.jar： java运行环境的类库

tools.jar：是工具类库

java5之后，就不需要配置CLASSPATH环境变量了，因为JVM会自动去当前路径加载类，以及自动去lib文件夹下加载dt.jar和tools.jar

jdk1.8 hashtable 插入部分源码？

if (oldTab != null) { //判断是不是第一次扩容，如果不是，需要复制元素到新表

// 遍历旧表每个位置，每条链表

for (int j = 0; j < oldCap; ++j) {

// 查看该位置是否由链表，如果没有就跳过

if ((e = oldTab[j]) != null){

oldTab[j] = null; //先让该位置指向空

if (e.next == null) // 判断e的next是否为空，为空的话就一个结点，无需考虑是树结点，还是链表结点

newTab[e.hash & (newCap - 1)] = e; // 计算新表的索引位置，指向e

else if (e instanceof TreeNode) // e.next不为空，多个节点的—可能是树

else{

// e.next不为空，有多个结点，这里是链表结点。

// 将链表复制到新表上，尾插法，保持原来链表的顺序

// 这段代码是执行尾插法的主要逻辑

}

}

}

}

执行尾插法主要逻辑：

if (loTail != null){

loTail.next = null;

newTab[j] = loHead;

}

if (hiTail != null){

hiTail.next = null;

newTab[j] = hiHead;

}

lo是low的缩写，hi是high的缩写

会通过(e.hash & oldCap) == 0判断是高位还是低位。

所以尾插法插入逻辑只需看一部分段代码即可

do{

next = e.next

// 这里执行尾插法，假设是低位插入

if (loTail == null)

loHead = e;

else

loTail.next = e;

loTail = e;

e = next;

}while(e!=null)

Spring如何实现AOP？

实现方式一：配置文件：

配置bean

配置aop-config，配置切面：

切点

通知

实现方式二：基于注解：

开启注解扫描

开启自动代理

切点表达式execution注意细节？

访问修饰符那里不写就代表所有，不能写\* 注意。

Spring AOP，JDK动态代理，CGLib动态代理？

Spring AOP采用动态代理实现。

默认的实现方式是JDK代理，这时java自带的实现方式，需要目标类实现接口，然后在运行时，生成代理类，代理类会实现与目标了相同的接口，以达到增强目标类方法的目的。

如果目标类没有实现接口，Spring AOP也有办法，它集成了CGLib动态代理，可以在运行时生成代理子类，继承目标类，并对目标方法进行增强。

public class UserService$$EnhancerBySpringCGLIB$$d4f3f4d extends UserService

什么是动态代理？有什么用？Java中可以怎么样实现动态代理？

1. 动态代理就是在不改变原有代码的前提下对原有方法增强
2. 对于一些方法， 我们可能要处理统一的逻辑，比如计算方法的执行实际，这时候我们就可以通过动态代理创建代理对象， 实现对目标方法的功能增强。
3. 实现动态代理的方时有两种：jdk动态代理和cglib动态代理

动态代理是代理模式的一种，指在运行阶段，根据目标类实现的接口动态生成代理类，实现与目标类相同的接口，以达到对目标对象功能增强的目的。在Java中，实现动态代理可以通过lang报下reflect包里面的Proxy类，调用newProxyInstance静态方法，该方法要传入三个参数：类加载器，指定动态生成的代理类的加载器，接口数组，指定代理类要实现的接口，也就是指定代理类该长什么样，该实现哪些方法。第三个是InvocationHandler，指定了代理类要干什么事，同参会传入一个匿名内部类，重写invoke方法，该方时里面写了代理类要做的增强逻辑。

cglib是一个强大的代码生成库，采用字节码技术，通过继承的方时实现对目标类的增强。代理类会继承目标类从而获得目标类要增强的方法，实际创建的对象是代理类对象，所以调用的方法是代理类对象的方法。

cglib是一个代码生成库。它面向的是父类的动态代理，也就是说它代理 一个对象就是通过继承目标类对原有方法进行增强。这意味着，它可以代理目标类对象的所有方法，并且由于反射机制的存在，它可以获取夫类方法上的所有注解。

动态代理经常出现在框架中，比如Spring 的 AOP技术，就是通过动态代理实现对目标方法的增强。Spring整合了两种动态代理技术，默认会使用JDK代理方时，如果目标类没有实现接口，则会使用cglib动态代理实现AOP。

Java 中的反射是什么意思？有哪些应用场景？有哪些优缺点？

1. 反射是Java程序运行过程中，动态地拿到一个对象所有信息的技术。
2. 反射应用在Spring IoC控制反转：在XML配置文件定义bean的信息，bean id和类的完全限定名，运行过程中，spring会解析xml配置文件，读取bean的类名，利用反射机制，获取该类的class对象，然后调用构造方法创建对象，实现了控制反转，将对象的控制权交给了Spring
3. 反射还应用在动态代理：拿JDK动态代理来说，在运行阶段，会读取目标类对象，利用反射机制获取目标类实现了哪些接口，然后在内存中动态的生成字节码文件，即目标类，并实现这些接口，重写接口里面方法，就使得目标类和代理类用于了相同的方法，可以进一步对目标类进行功能拓展或增强。
4. 反射的优缺点：反射的优点是可以提高程序灵活性和可拓展性，比如Spring的IoC将对象创建权交给spring，同时也简化了某些功能的实现。反射也有一些缺点，主要是性能和安全问题。反射涉及到动态类型解析，会有一定的性能开销，所以反射操作的效率要比非反射操作效率低。其次，反射可以绕过访问修饰符的限制，对于私有程序变量也可以访问，这可能会导致安全问题。

如果我想要让 hashmap 变成线程安全的，你觉得可以怎么做？（有时候会扯到 concurrentHashMap，不过咱们这里先不追击这个）

1. synchronizedMap

Collections的synchronizedMap方法会传入一个map，返回一个，线程同步的map。这map的运行时类型时SynchronizedMap，SynchronizedMap，它是Collections中的一个静态内部类，该类实现了Map接口，所以返回对象可以用Map变量来接，传入的map会被作为这个类的一个常量m所指向。调用put、get、remove等方法时都会用sychronized关键字进行同步，该关键字保证了任意时刻，最多只有一个进程在执行同步代码块。值得注意的是：真正执行put、get方法的是传进来的map。

1. hastable

hashtable存放的是键值对，但是建和值都不能为null。HashTable的用法基本和HashMap一样，但是HashTable是线程安全的，因为hashtable 对map操作的方法都加了synchronized关键字进行了线程同步。HashMap和HashTable是平级的关系，HashTable有一个子类是Properties

底层有数组，时HashTable$Entry类型的，Entry时HashTable的静态内部类。并实现了Map$Entry。

构造HashTable时就已经初始化了table，大小默认是11，不用想HashMap那样等到第一次put时才扩容。而此时临界值是8 = 11 \* 0.75得到的

hashtable扩容机制：int newCapacity = (oldCapacity << 1) + 1;

左移1位再加1，所以从11变成了23=11\*2+1；

HashTable可以实现线程安全，但是效率么有HashMap高。

HashTable不可以插入null键和null值

HashMap键和值都可以为null

3、concurrentHashMap(暂时不追这么深)

为什么HashMap传入的key和value都可以为null，但是HashTable传入的key和value 都不允许不允许为null？

1. 这是因为它们计算哈希值的算法不同导致的key不同：

1.1、HashMap中，计算一个key的哈希值的算法：

int hash(K key){

return (key == null) ? 0 : ((h=key.hashcode())^h>>16);

}

可以看到，这里就以已经考虑到key为null的情况，如果key为null，哈希值就为0。

1.2、HashTable计算哈希值的方式是：

int hash = key.hashcode()

所以传入的key为null，会有NullPointerException异常。

2、设计出来的理念不同，导致value不同

如果value为null，Hashtable会主动报一个空指针异常。

因为HashTable设计出来是用于多线程的，如果一个线程在获取一个value时，另一个线程将其改为了null，可能会出现异常。

3、HashSet HashSet的底层本质上就是一个HashMap。所以它和HashMap性质一样，也可以传入null。

|  |  |  |  |
| --- | --- | --- | --- |
|  | HashMap | HashTable | HashSet |
| key | 可以为null | 不可以为null | 可以为null |
| value | 可以为null | 不可以为null | 常量 |

聊一聊ArrayList？

ArrayList以数组形式实现List接口。它具有动态扩展的能力，因此ArrayList也被称为动态数组

public class ArrayList<E> extends AbstractList<E>

implements List<E>, RandomAccess, Cloneable, java.io.Serializable

继承AbstractList类

实现List接口

实现了RandomAccess接口，可以随机访问

实现Cloneable接口，可以被克隆

实现Serializable接口，可以序列化

ArrayList源码追踪：

List list = new ArrayList();

构造一个ArrayList对象，有一个elementData引用变量，指向一个Object类型，名为DEFAULT\_EMPTY\_ELEMENTDATA数组，大小为0，该数组被所有用默认构造函数构造的arraylist共享。当第一次添加时，会扩容，然后不同的arraylist的elementData会指向不同的内存地址。

list.add(1);

调用add方法

ensureCapacityInternal(size + 1)

首先会指向ensureCapacityInternal函数来保证数组大小，这里传进来的size+1表示所需要的最小容量minCapacity。该函数源码如下：

private void ensureCapacityInternal(int minCapacity) {

ensureExplicitCapacity(calculateCapacity(elementData, minCapacity));

}

该函数首先会取调用calculateCapacity()函计算需要大多的数组。该函数计算方式如下：

if (elementData == DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA){

首先会判断elementData是不是指向默认空数组，由于是第一次调用add，所以是。

return Math.max(DEFAULT\_CAPACITY, minCapacity);

这里返回DEFAULT\_CAPACITY（默认是10）和minCapacity（传进来的是1）之间的大小，返回较大的一个，这里返回10

}

继续执行ensureExplicitCapacity(10) 这个10是之前计算的结果。该函数首先会判断if (minCapacity - elementData.length > 0) 要扩容的minCapacity是不是比elementData的长度大然后执行grow(minCapacity)方法。

private void grow(int minCapacity){

// 这段代码才是真正的数组扩容

// 记录原来的长度，这里为0

int oldCapacity = elementData.length;

// 计算新的容量，这里结果为0，

int newCapacity = oldCapacity + (oldCapacity >> 1);

if (newCapacity - minCapacity < 0)

// 如果新容量比要扩容的容量小，就按minCapacity来扩容，反之就按计算结果来扩容

newCapacity = minCapacity;

把

elementData = Arrays.copyOf(elementData, newCapacity)

}

list.add(2);

第二次执行add

private static int calculateCapacity(Object[] elementData, int minCapacity) {

if (elementData == DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA) {

return Math.max(DEFAULT\_CAPACITY, minCapacity);

}

return minCapacity;

}

这里会返回2，因为不是不是第一次add，所以if判断不成功

ensureExplicitCapacity(2)

if (minCapacity - elementData.length > 0) // 这里判断不会成功

因为minCapacity是2，数组长度是10，所以不会去扩容。知道minCapacity为11是，也就是add第11个元素时：

if (minCapacity - elementData.length > 0) 此时minCapacity为11，elementData.length为0，所以if判断成功，会进入grow方法

new = old + (old >> 1) = 10 + 10 \*/2 = 15

15 > 10

所以

elementData = Arrays.copyOf(elementData, newCapacity);

elementData的容量变成15

再下一次

15 + 15 / 2 = 15 + 7 = 22

List list = new ArrayList(0);

这里如果给了一个初始化大小为0

list.add(1);

第一次add的时候也会进行判断

private static int calculateCapacity(Object[] elementData, int minCapacity) {

if (elementData == DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA) {

return Math.max(DEFAULT\_CAPACITY, minCapacity);

}

return minCapacity;

}

但是这里的if条件就不成立了，因为elementData指向的是EMPTY\_ELEMENTDATA，所以计算的所需的最小容量是1并不是10。所以第一次扩容的时候要看构造的ArrrayList是怎么样，如果是无参构造，那么容量就会是10。如果是有参构造，那么计算的容量就是size+1；

扩容路径:

0 🡪 1 🡪 2

if (newCapacity - minCapacity < 0)

newCapacity = minCapacity;

因为 1+1/2 = 1，但是newCapacity小于minCapacity，所以按minCapacity来。

0 🡪1 🡪2 🡪3 🡪 4 🡪 6🡪9

x + x / 2 = x \* (3/2) = x\*1.5 原来的1.5倍

ArrayList注意事项：

1. ArrayList里面可以加入null，并且可以加入多个
2. ArrayList的实现基本等同于Vector，但是ArrayList不是线程安全的。因为Vector的方法都加上了synchronized关键字。

聊一聊LinkedList？

LinkedList是一个全能战士，因为它同时实现了list接口和deque接口（双端队列，两头都可以添加和删除的队列）。所以LinkedList即可以是list，也可以是栈或队列，这却决于你怎么看待。这是由于LinkedList底层是双向队列。LinkedList有以下一些注意事项：

1. LinkedList实现了双向链表和双端队列的特点
2. 可以添加任意元素，包括null
3. 线程不安全

LinkedList 底层是双向链表，维护了两个Node类型引用，first和last，分别指向头节点和尾节点。Node是LinkedList的内部类

private static class Node<E> {

E item;

Node<E> next;

Node<E> prev;

}

Node类型就维护了三个属性，item表示当前节点的值，next指向下一个节点，pre指向之前的结点，所以是双链表

List list = new LinkedList();

构造LinkedList，first和last都指向null

list.add(1);、

linkLast(e);

add方法会把元素添加到结点末尾。

void linkLast(E e) {

// l指向last

final Node<E> l = last;

// 构造一个新节点

final Node<E> newNode = new Node<>(l, e, null);

last = newNode; // 让last指向新结点

如果l为空，就会让first指向newNode。此时first和last指向同一个节点，因为只有一个结点

如果l不为空，就让l.next指向last

}

linkedList删除元素有两种方式

1. 按索引位置删除
2. 按元素删除

如果传入的是一个int类型，就会按索引位置删除

如果传入的不是基本数据类型，就会按元素删除

会调用equals方法来比较元素是否相等

对于如果传入的是null，会单独用==运算符比较

list.remove(1); 按索引位置删除

list.remove(Integer.valueOf(1)); //删除与1equals相等的结点

这两句代码含义是不同的

List集合的选择：

|  |  |  |  |
| --- | --- | --- | --- |
|  | 底层结构 | 增删效率 | 查询效率 |
| ArrayList | 数组 | 低 | 高 |
| LinkedList | 双向链表 | 高 | 低 |

请你说一说 ArrayList 和 LinkedList 区别？

ArrayList底层是数组实现的，所以ArrayList支持数据随机访问，查询的时间复杂度时O(1)。但是删除ArrayList元素有数组复制和移动，时间复杂度为O(n)，效率较低。

LinkedList底层是双向链表实现的，每个结点都有可以找到其前驱节点和后继结点。所以删除一个结点时只需要修改对应的引用即可，时间复杂度时O(1)，但是ArrayList不支持随机访问，查询某个结点，需要遍历整个链表，所以查询时间复杂度时O(n)。

1. 聊一聊底层实现 ok
2. 聊一聊不同的效率 ok

底层实现不同，ArrayList底层是数组。LinkedList底层是双向链表实现。

由于底层实现不同，所以在进行增删操作的时候LinkedList操作效率要比ArrayList高，执行改查操作时，ArrayList效率更高。

ArrayList支持对数据的随机访问，LinkedList只能逐个遍历访问元素。

如果我要删除第 k 个元素，也就是会执行 remove(k)，那么这个 remove 的操作，它们的时间复杂度各自是多少？

要删除第k个元素，首先要找到第k个元素。ArrayList底层是数组实现，支持数据随机访问，所以找到第k个元素的时间复杂度为O(1)。LinkedList底层是链表实现，找到第k个元素需要遍历整个链表，所以时间复杂度是O(n)。

找到第k元素后要执行删除逻辑，对于ArrayList，底层采用将数组第k+1以后的元素复制到当前数组从k开始的元素，所以第k个元素回被覆盖，等同于删除，然后数组大小减一，这个复制和移动数组的操作时间复杂度是O(n)。对于LinkedList，通过第k个节点可以找到其前驱节点和后继结点，修改这些节点的引用即可，时间复杂度的为O(1)。

综上所述，ArrayList和LinkedList删除一个元素，综合的时间复杂度都是O(n)

对于ArrayList，remove(k) 首先会去找到k索引位置的元素，这个时间复杂度时O(1)，然后回移动数组，将k+1到数组长度的元素向前移动一位，在ArrayList底层实现是通过数组拷贝，将k+1到数组size的元素拷贝到当前k元素开始，最后将数组大小减一，并将原来最后一个值指向空，这个操作是O(n)

LinkedList的寻找的时机复杂度是O(n)，删除的时间复杂度是O(1)

可以说一说它们的使用场景吗？或者说一说你平时在处理什么事情的时候，用过它们？

根据项目来说。

对于list需要大量增删的地方使用LinkedList

对于只做查询和修改的地方使用ArrayList

AarrayList 底层实现是数组，数组就会有容量限制，可以简单说一下 ArrayList 的扩容机制吗？（）

如果是无参构造一个ArrayList，初始化后回得到一个ArrayList，它的elementData属性指向DEFAUTL\_EMPTY\_ELEMENTDATA数组，大小为0。当第一次add时，ArrayList回判断elementData是不是指向DEFAUTLCAPACITY\_EMPTY\_ELEMENTDATA。如果是，就会将要扩容的大小设置为10，然后进行扩容，所以第一次add之后emeentData数组的大小变为10，以后每次数组满了之后就会扩大原来的1.5倍

如果是有参构造，在构造时制定了容量的大小，那么初始化的时候就会得到一个对应大小的Object数组，以后每次数组满了，需要扩容就按这个容量大小的1.5倍进行扩容。

简单说一下 list 和 set 的区别？以及使用场景？

两者继承Collection接口，都是用来存储数据的集合。List接口回会维护元素的插入顺序，并且根据索引进行数据的查询和操作。Set接口只强调元素的不可重复性，不保证元素的插入顺序，不支持索引查询。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 接口 | 父接口 | 实现类 | 插入顺序 | 索引查询 |
| List | Collection | ArrayList  LinkedList | 保证 | 支持 |
| Set | Collection | HashSet  TreeSet | 不保证 | 不支持 |

List接口的常见实现类有ArrayList和LinkedList，前者底层用数组实现，适合查询和修改多的场景；后者是LinkedList，采用双向链表实现，时候删除和插入多的场景

Set接口场景实现类是HashSet和TreeSet，前者底层是HashMap，提供快速的插入、删除、查询操作。后者的底层实现是红黑树，不保证元素的插入顺序，但保证了元素的大小顺序，可使用比较器进行排序。

一条sql语句是如何执行的？

select \* from T where ID=10；

对于这条语句的查询

mysql的大体结构：

mysql大体可以分为Server层和存储引擎层。

Server层：核心服务+内置函数+跨存储引擎功能，Server层包括连接器、查询缓存、分析器、优化器 、执行器

存储引擎：0、负责数据真正的存储和提取。1、插件式，即插即用。2、常见的存储引擎：InnoDB MyISAM memory。3、mysql5.5之后InnoDB成为默认的存储引擎，即在创建表的时候可以指定存储引擎，如果没指定，默认是InnoDB

连接器：

mysql -h43.142.181.137 -P3306 -uroot -p

我在客户端机器上想要访问数据库，首先要进行连接，这时候接待我的就是连接器。

连接器主要负责两块内容：管理连接，权限验证。

管理连接包括：建立连接、维持连接

权限连接：在用户名密码都正确的情况下，连接器回让这个连接请求通过，并拿着用户名去权限表里面查询出该用户拥有的权限，此后，只要这个连接每断开，对数据库的访问、操作都基于此权限，即使root修改了该用户的权限。



什么时候回断开连接呢：当客户端太长时间没有操作，这个时间是由参数wait\_timeout设置的，默认是8小时。

所以连接分为长连接和短连接。长连接是指客户端持续有请求过来，一直使用一个请求。短链接是指每次执行完很少的查询就断开连接，下一次再重新建立连接。

由于建立连接是一个比较复杂的过程，所以尽量减少建立连接的动作，使用长连接。但是，mysql的执行过程中临时内存都是存放在连接对象中的，只有当连接断开，这些对象才会释放。所有长连接里面有很多垃圾内存，可能已经不使用了，但是连接对象没释放，所以这些垃圾内存也不会释放，最终导致内存飙升。

针对这个问题有两种解决办法：

1、定期断开长连接，主动释放内存资源；或者在程序里面判断如果 执行了一个占用大内存的查询之后，就主动断开连接，之后要查询再重连

2、如果是Mysql5.7之后的版本，可以在执行一个占用大内存的查询后，执行一次mysql\_reset\_connection命令，来重新初始化连接资源。该命令会将当前状态恢复到刚刚建立连接时的状态，将连接对象里的内存释放。但是改命令不需要重连和重新权限校验。

查询缓存

分析器：分析器会分析你要干什么 通过词法分析和语法分析

优化器：优化执行的方案。例如

select \* from t1 join t2 using(ID) where t1.c=10 and t2.d=20;

这里默认是内连接，也就是两边都满足的行才会保留

这里有两种连接顺序：

1. 找到t1中所有c=10的ID，再根据ID值关联到t2，再判断t2里面d值是否等于20
2. 找到t2中所有d=20的ID，再根据ID值关联到t1，再判断t1里面c值是否等于10

Mysql通过分析器知道你要做什么，通过优化器知道怎么做好，然后就会进入执行器：执行器通过操作存储引擎来执行sql语句

开始执行之前，需要判断以下该用的权限，这个权限是根据前面连接器查出来的权限为准。

只有用于对某张表的权限才能进行查询、修改

mysql> select \* from T where ID=10;

以这条语句为例，他在执行阶段具体是怎么执行的呢，怎么操作存储引擎呢：

1. 调用InnoDB存储引擎，读取T表第一行数据，判断ID值是否位10，如果不是则跳过，如果是则将该行保存到结果集中。
2. 调用引擎接口读取下一行，重复相同的判断逻辑，直到读取到这个表的最后一行
3. 执行器将上述遍历过程中满足条件的行组成的记录集作为结果集返回客户端

至此，执行器执行完了SQL语句。

对于有索引的表，执行器调用一次，取的是满足条件的第一行。然后满足条件的下一行。

rows\_examined的计算是根据调用一次引擎获取到的行数累加的。但是执行次调用一次引擎接口，引擎可能会扫描多行，因此，引擎扫描行数根rows\_examined并不是完全相同。

为什么查询缓存可以提高查效率，但是还是不建议使用呢？

因为我们在设计缓存时，如果更新了表，那么这个表上的所有缓存都要清空。很可能出现这样一种清空，刚把缓存建立起来，结果还没用，就更新了表，然后缓存清空。对于更新压力大的系统，查询缓存的命中率会非常低，除非业务就是有一张静态表，很长时间才更新一次。比如一个系统配置表，对于这张表上的查询才适合使用查询缓存

一条mysql语句是如何更新的？

一条查询语句是经过连接器、分析器、优化器、执行器等功能模块到达存储引擎。那么一条更新语句的执行流程是什么呢:

mysql> update T set c=c+1 where ID=2;

与查询语句一样，首先需要通过连接器建立连接并校验权限。然后分析器通过词法分析和语法分析知道这是一个更新语句，优化器决定使用ID这个索引。然后，执行器负责具体执行，首先找到这一行，让后更新值。

重做日志（redo log）

粉板，专门用来记录客人的赊账记录。如果赊账的人不多，就可以把顾客和账目写在粉板上。如果赊账的人多了，粉板上记不下，这时候掌柜有一个专门用来记录赊账的账本。

某个时刻，如果有人需要赊账或者换账，老板有两种选择：

1. 直接把账本翻出来：把这次赊账加上或扣除掉。
2. 先找粉板上记录下这次的账，等打烊后再把账本翻出来核算。

磁盘就相当于账本，日志就相当于粉板

这就是MySql的WAL技术，Writing-Ahead Logging技术，它的关键点就是先写日志，在写磁盘。

具体来说，当一条记录需要更新时，InnoDB引擎会先把记录写redo log并更新内存中的数据。InnoDB引擎会在适当的时候将操作记录更新到磁盘里面。而这个更新往往是系统比较空闲的时候。

redo log 满了怎么办？

redo log日志本身也是一个文件，假设一组4个日志文件，每个文件的大小是4GB，那么这块“粉板”就最多可以记录4GB的操作。从头开始写，写到末尾右从头开始写



write pos 是告诉InnoDB从这里从这里开始写，check point是上一次擦除的位置，它们之间的绿色的部分就是InnoDB还可以写的内容。当又有新的操作进来时，进行往绿色部分写，知道write pos追上check point，此时InnoDB就不能再往redo log 里面写任何日志了。InonoDB会停止所有更新操作，先擦除一部分记录，让check point往前走。

有了redo log， InnoDB就可以保证即使数据库发生异常重启，之前提交的记录都不会丢失，这个能力成为crash-safe

聊一聊binlog？

Mysql分为两层，一块时Server层，负责Mysql功能层面的事情，另一块是存储引擎层，负责存储相关的具体事宜。redo log日志是InnoDB特有的日志文件，这也是它为啥会成为默认的存储引擎。而Server层也有自己的日志文件，binlog 归档日志

为什么mysql会有两份日志？

这是由于历史原因。最开始Mysql的存储引擎是MyISAM，而这个存储引擎没有redo log日志文件，所以没有creash-safe能力，binlog只能用于归档。而InnoDB是另一家公司，以插件形式被Mysql引入，既然只依靠binlog没有crash-safe能力，所以InnoDB使用另外一套日志系统，也就是redo log来实现crash-safe能力。而binlog也有其独特作用，就保留了binlog + redo log的两套日志。

redo log和binlog的区别？

|  |  |  |
| --- | --- | --- |
|  | redo log | bin log |
| 所属者 | InnodDB特有的，其它存储引擎没有 | MySql server层实现的，所有存储引擎都可以使用 |
| 日志类型 | 物理日志  “哪个数据页上做了什么修改” | 逻辑日志  记录这句sql语句的原始逻辑  “给ID=2的一行的字段c加1” |
| 写的方式 | 循环写  会覆盖以前的文件1  写满了从头开始写 | 追加写  不会覆盖以前的文件  写满了开一个新文件 |

mysql> update T set c=c+1 where ID=2;这条语句在执行器阶段怎么执行的？

这条语句通过连接器、分析器、优化器，来到了执行器。

执行器负责和InnoDB存储引擎交互，其执行流程如下：

1. 执行器先找引擎取ID=2这一行。ID是主键，引擎直接用树搜索找到这一行。在查找过程中，如果ID=2这一行所在的数据页本身就在内存中，就直接返回，如果不在内存中，则需要去磁盘读取，然后返回。
2. 执行器拿到了存储引擎返回的一行数据，把这行的c值加上1，得到新的一行的数据，在调用存储引擎写入这行新数据。
3. 存储引擎将这行新数据更新到内存中（ID=2这一行所在的数据页此时一定在内存中，因为有第一步，如果ID=2这一行所在的数据页不在内存中，回去磁盘读取，所以ID=2这一行数据所在的数据页此时一定在内存中），同时将这个更新操作记录到redo log里面。此时redo log处于prepare状态。然后告知执行器，完成了，随时可以提交事务（这里没有显示的写事务，mysql会自动把一条语句作为一个默认事务，这个事务的提交是自动的）
4. 执行器生成这个操作的binlog，并把binlog写入磁盘。执行器调用引擎提交事务接口（事务真正提交是存储引擎做的）但是这个接口的调用是执行器调用的。存储引擎将刚才写的redo log改成提交（commit）状态。更新完成



为什么要把提交搞得这么复杂，必须两阶段提交呢？

这是为了让redo log 和binlog两份日志之间的逻辑保持一致

binlog会记录所有的逻辑操作，并且是以追加写的方式。如果DBA说可以恢复到半个月内任意一秒的状态，必须满足以下条件：

1. 备份系统一定会保存最近至少半个月的binlog。
2. 系统会定期做整库备份。

这里的定期取决于系统的重要性，可以是一天一备份，可以是一周一备。

整库备份是指整个数据库所有的表做备份。

当需要恢复到指定一秒时，比如在2月3号14:23:12，又一次误删表操作，需要找回数据，该怎么做：

1. 找到最近的一次全量备份，比如找到2月1号9:20:17做了一次全量备份。那么我们就使用整个备份做数据恢复。
2. 从备份时间开始，将备份的binlog依次取出来，重放到误删表的那一刻



对于为什么需要两步法提交，这里用反证法证明：

如果不用两步法提交，那么binlog和redo log总要有一个写入顺序，那么进行分类讨论。update T set c=c+1 where ID=2; 还是执行这条语句，假设c的值为0，再假设执行update语句过程中，在写完第一个日志文件后，第二个日志文件还没写完就发生了crash。

1. 先写入redo log 再写如binlog

假设在redo log写完，binlog还没有写完的时候，MySql进程异常重启。由于之前我们说过的，redo log写完之后，系统即使崩溃，任然能把数据恢复回来。恢复过来，数据表中的这一行c值为1。由于binlog在写的时候发生了崩溃，所以里面没有修改c值这一行记录。



如果用这个binlog来恢复临时库的话， 会发现临时库和原库的值不同，对不上号。

1. 先写binlog，再写redo log

binglog写完后，redo log还没写完，MySql发生了异常重启，奔溃后这个事务无效，由于redo log没有写完，所以c值修改这一行就没记录，磁盘中改行的c值仍为0。但是binlog里面记录c值修改的操作，所以在某次恢复临时库时会把c值修改为1，但是原库里面c值为0，所以临时库和原库不同



总结，binlog 和 redo log都可以用于表示事务的提交状态。这句话怎么理解呢：首先对于server层，binlog写完了就任务事务已经提交了。对于InnodDB，redo log写完了事务已经提交了。单独对它们每个人来说，这两个文件分别表示事务的提交状态。而两阶段提交就是让这两个状态在逻辑上保持一致。在写入任何一个日志文件时发生奔溃，另一个日志文件也不会保存。

为什么binlog没有crash-safe能力？

要回答这个问题，首先需要直到什么是crash-safe，crash-safe是指MySql数据库在运行过程中发生崩溃，重启后能恢复到崩溃前的状态。

binlog是无法做到的，因为binlog是逻辑日志，如果崩溃前没有做全量备份，就无法恢复到重启前的状态。

而redolog是物理日志，它会记录具体哪个数据页，修改成什么样。所以只要写入了redolog，就可以立马从redolog知道哪个表，哪一行，哪个字段，该是什么指。

binlog是逻辑日志，记录了操作是什么。老值+操作=新值

redo log是物理日志，记录了值是什么。 读取redo log =>得到新值=>覆盖老值。

innodb\_flush\_log\_at\_trx\_commit，设置成1，表示每次事务的redo log都直接持久化到磁盘，这样可以把保证mysql异常重启后数据不丢失。如果每次事务的redo log放到内存中的redo log buffer，那么重启后，这个buffer里面的数据没有持久化，就恢复不了了

一次事务=一个和多个操作。

sybc\_binllog参数设置为1，每次事务的binlog都直接持久化到磁盘。保证mysql异常重启后binlog不丢失。

说说binlog和redolog，redolog两阶段提交的过程？

简单来说就是存储引擎将更新日志写入redo log不会马上提交，而是处于prepare状态，此时执行器可以随时提交事务。执行器将更新操作写入binlog中，写完之后调用存储引擎的提交事务的接口，存储引擎将redo log从prepare状态修改为commit，事务提交结束。这就是两阶段的过程。

从mysql update语句出发， 要求说出binlog 与 redolog一致性问题

假设执行update T set c = c + 1 where ID = 2，再假设没有二阶段提交。这是这两个日志总有一个提交顺序，我们分类讨论：

1. 如果redo log先提交，binlog后提交：

在redo log提交后，写入binlog是，mysql发生了崩溃重启，由于redo log里面已经写入，所以原表可以恢复此次修改，c的值为1，后来，通过备份数据库恢复临时数据库时，由于binlog没有记录c=c+1这次操作。会导致临时表与原表数据不一致。

1. 如果binlog先提交，redo log 后提交：

假设binglog已经提交，在写入redo log时，发生了mysql奔溃。由于redo log还没有写入磁盘，所以此次修改不会发生，原库中的c仍然是0。还来进行临时表恢复时，由于binlog里面记录了此次修改，c的值会变为1。此时临时库与原库的数据不一致。

介绍一下 binlog 和 redo log，他们两有啥区别？

|  |  |  |
| --- | --- | --- |
|  | binlog | redo log |
| 所属 | 属于mysql server层，所有存储引擎共享。 | 属于InnoDB存储引擎特有。 |
| 日志 | 逻辑日志  记录操作 | 物理日志  记录值是什么 |
| crash-safe | 不支持 | 支持 |

两阶段提交了解吗？介绍一下，为啥需要两阶段提交呢？

binlog 和redo log 分别代表了server层和存储引擎认为事务的状态。如果binlog成功写入，server层就认为事务提交。如果redo log成功写入，存储引擎层就认为事务提交了，两阶段提交就是为了保证这两种事务状态的一致性。

什么是隐式事务？

对于单条更新语句如：update T set c = c + 1 where ID = 2; 如果执行失败了，数据库也不会更新成功，相当于自动回退事务了。这是因为mysql会隐式地加上事务，在成功写入redo log 和binlog之后会自动提交事务。

事务隔离，为什么你改了我还看不见？

事务最经典的例子就是转账，转账包括查询余额，做加减法，更新余额等操作。

这些操作必须保证是一体的，因为如果不是一体的，在做加减法之前，可以再查询一次余额给别人。

事务定义：事务是一组对数据库的操作，这些操作要么全部成功，要么全部失败。在Mysql中，事务支持是在引擎层实现的。Mysql可以插件式的使用很多存储引擎，但不是所有存储引擎都支持事务，其中MyISAM就不支持事务，所以被InnoDB淘汰了。

事务的四大性质：ACID，Atomicity，Consistency，Isolation，Durability ；原子性、一致性、隔离性、持久性。

原子性好理解，不可再分割的操作。

一致性：从一个一致性状态转移到另一个一致性状态。

持久性：已提交的事务应持久化，即使系统崩溃，也能恢复。

隔离性：多个事务并发执行时，一个事务的执行不应该影响到其他事务。

隔离级别：多个事务并发执行时，会产生很多问题，不同的隔离级别可以解决不同的问题。

隔离性与隔离级别的区别？

隔离性是一个事务应该满足的性质。

隔离级别是为了解决事务并发执行过程中产生的各种问题。

事务并发执行过程中会产生哪些问题？

脏读、不可重复读，幻读。

脏读：读到了其他事务修改后但是还未提交修改的数据。

|  |  |  |
| --- | --- | --- |
| 时刻 | 事务A | 事务B |
| t0 |  | c = 1 |
| t1 |  | c = c + 1 |
| t2 | c=2 |  |
| t3 | c = c+1 |  |
| t4 |  | B发生错误，事务回滚 |
| t5 | commit |  |

事务A读取了事务B修改后，提交前的数据

针对这种情况，我们让事务只能读提交即可解决

不可重复读：在同一个事务内，第一次读的数据，和第二次读的数据不一致，并且在这个事务内，并没有对该事务进行修改

|  |  |  |
| --- | --- | --- |
| 时刻 | 事务A | 事务B |
| t0 | c = 1 |  |
| t1 |  | c = 1 |
| t2 |  | c=c+1 |
| t3 |  | commit |
| t4 | c=2 |  |
| t5 | c = c + 1 |  |
| t6 | commit |  |

事务A前后两次读取的指不一样，虽然事务只能读提交，但是坏就坏在t4，虽然是读提交，在事务A的视角来看，我又没有修改c，但是c前后读取的数据却不一致，所以很矛盾。需要把隔离级别设置成不可重复读才能解决。

幻读：虽然设置了不可重复读隔离级别，我读的时候限制那一行数据保持前后一致，但是无法限制其他事务对那张表其他行数据进行增加删除修改

|  |  |  |
| --- | --- | --- |
| 时刻 | 事务A | 事务B |
| t0 | 数据量是100条 |  |
| t1 |  | 新增100条数据 |
| t2 | 其他操作 |  |
| t3 |  | commit |
| t4 | 查询数据量是200条 |  |
| t5 | commit |  |

就好像发生了幻觉一样，站在事务A的视角，我已经解决了在该行数据上可以重复读的，但是整张表的数据量却在改变。

脏读：读取了其他事务修改后，但是提交前的数据。解决方案：事务只能读取已提交的数据。

不可重复读：第一次读取数据与第二次读取数据之间，其他事务对改行数据修改并进行了提交，导致两次读取数据的结果不一样。

幻读：第一次读取数据与第二次读取数据之间，其他事务对改行之外的其他行做了删除或对表做了添加行操作。导致两次读取的数据总量不一样。

脏读：事务读取了其他事物已修改，但是没有提交的数据

|  |  |  |
| --- | --- | --- |
| 时间 | 事务A | 事务B |
| t0 | 开始事务 |  |
| t1 |  | 开始事务 |
| t2 |  | c=1 |
| t3 |  | c = c+1 |
| t4 |  |  |
| t5 | c=2 |  |
| t6 |  | 事务B发生错误，事务回滚 |
| t7 | c = c + 1 |  |
| t8 | commit |  |

对于脏读，规定事务只能读取已提交的数据就可以解决。

不可重复读：第一次读取数据和第二次数据之之间，其他事物修改了数据并提交，导致一个事务内两次读取的数据不一致，且这两次读取之间，该事务并没有修改数据。读提交隔离级别无法解决不可重复读问题。

|  |  |  |
| --- | --- | --- |
| 时间 | 事务A | 事务B |
| t0 | 开始事务 |  |
| t1 |  | 开始事务 |
| t2 | c = 1 |  |
| t3 |  | c=1 |
| t4 |  | c = c+1 |
| t5 |  | commit |
| t6 | c = 2 |  |
| t7 | c = c + 1 |  |
| t8 | commit |  |

对于事务A来说，它遵守了读提交的隔离级别，读取别人已提交的数据，但是造成了自己前后数据读取不一致，不可重复读的问题。解决不可重复读需要对要读的行加行锁

幻读：虽然对要操作的行加了行锁，但是其他事务仍然可以操作改行以外的地方，导致数据前后读取到底表的数据总数不一致

|  |  |  |
| --- | --- | --- |
| 时间 | 事务A | 事务B |
| t0 | 开始事务 |  |
| t1 |  | 开始事务 |
| t2 | count(T)=100 |  |
| t3 |  | 向T插入100条数据 |
| t4 |  | commit |
| t5 | count(T)=200 |  |
| t6 | commit |  |

不可重复读和幻读有什么区别？

|  |  |  |
| --- | --- | --- |
|  | 不可重复读 | 幻读 |
| 针对数据 | 某个字段被修改 | 某行被插入或删除 |
| 针对操作 | update | insert delete |
| 解决方法 | 行级锁，锁定该行，事务A多次读取操作完才释放该锁。 | 表级锁，锁定整张表，事务A读取多次表的总量数据操作完才释放该锁。 |

|  |  |  |  |
| --- | --- | --- | --- |
| 隔离级别 | 脏读 | 不可重复读 | 幻读 |
| read uncommitted | ✓ | ✓ | ✓ |
| read committed | × | ✓ | ✓ |
| repeatable read | × | × | ✓ |
| serializable | × | × | × |

✓：表示会出现该问题

×：表示不会出现该问题

悲观锁和乐观锁

并发控制：保证在并发操作时，通过一定手段保证在并发请款下数据的准确性，通过并发控制，可以使用户和别人同时操作的结果和他单独操作的结果保持一致。

乐观锁：乐观并发控制。

乐观锁和悲观锁是一种思想，不知在关系型数据库中出现。

悲观锁思想：认为数据被并发修改的概率较大，在修改之前加上。

数据库的锁是锁。

我们在写数据前，先对数据加上锁，防止其他事务并发修改。这个行为是受悲并发控制思想指导的。

思想：悲观并发控制思想，乐观并发控制思想

思想知道行为：

行为1：我们在写数据前，先对数据加上锁，防止其他事务并发修改。

行为2：我们在写数据前，不对数据做任何限制，其他并发事务也可以进来访问、修改数据，只是在提交更新的时候比较以下版本号，如果版本错误就给用户报错。

|  |  |  |
| --- | --- | --- |
|  | 读锁 | 写锁 |
| 读锁 | 不互斥 | 互斥 |
| 写锁 | 互斥 | 互斥 |

聊一聊乐观锁和悲观锁？

并发控制：在并发的情况下，采取一定的措施，保证并发访问数据的正确性，使得用户在与其他用户同时操作的结果与用户自己单独操作的结果是一样的。

悲观并发控制又叫悲观锁，乐观并发控制又叫乐观锁。悲观锁和乐观锁是一种思想，又这个思想出发，产生了不同的行为

悲观锁的思想：数据被并发修改的概率大

乐观锁的思想：数据被并发修改的概率小

基于悲观锁的思想指导下，我们产生了这样的行为：在对数据库的数据进行读或写是，事务首先会尝试去给这个数据加上读锁或写锁，如果加锁失败，说明又其他并发事务正在访问该数据，当前事务等待，知道加锁成功，然后操作数据，操作完毕后释放锁，留给别的事务继续争抢锁。

基于乐观锁的思想指导下，我们产生了这样的行为，在事务操作数据整个过程中，不排斥其他并发事务来访问或修改数据，因为我们认为并发修改产生数据错误的概率很低，知道要提交更新的时候，我们才去检查数据是否产生冲突，如果发生冲突了，则返回错误信息，让用户去决定怎么做。

悲观锁的实现，往往依靠数据提供的锁机制。在数据库中，悲观锁的流程：

1. 在对记录进行修改前，先尝试为该记录加上排他锁
2. 如果加锁失败，说明该记录正在被修改，那么当前查询要等待或者抛出异常，具体响应方式又程序开发者决定。
3. 如果加锁成功，就可以对记录修改，事务完成后就会解锁
4. 期间如果又其他事物尝试对该记录做修改或者加排他锁，都会等待我们解锁或直接抛出异常

select quantity from items where id=1 for update; 可以对id=1的记录加上行级锁，行级锁是基于索引的，如果一条sql语句用不到索引是不会使用行级锁，会使用表级锁把整张表锁住。

要使用悲观锁，必须把mysql 的 autocommit设置为0，避免执行一个更新操作后mysql将结果立即提交，因为我想手动提交事务

乐观锁的实现方式：乐观锁不需要依赖数据库锁机制，主要有两个步骤，冲突检测和数据更新。

CAS技术 Compare And Swap 当多个线程尝试使用CAS同时修改一个变量时，其中只有一个线程能够更新变量的值，而其它线程都失败，失败的线程并不会被挂起，而是被告知在这次竞争中失败，并可以再次尝试。

//查询出商品库存信息，quantity = 3

select quantity from items where id=1

//修改商品库存为2

update items set quantity=2 where id=1 and quantity = 3;

在更新以前，先查询一下当前的库存数，如果还是为3我才执行更新。保证了可重复读

|  |  |  |
| --- | --- | --- |
| 时间 | 事务A | 事务B |
| t0 | quantity = 3 |  |
| t1 |  | quantity = 3 |
| t2 |  | quantity += 1 |
| t3 |  | commit |
| t4 | quantity = 2 更新失败 |  |
| t5 |  |  |

|  |  |  |
| --- | --- | --- |
| 时间 | 事务A | 事务B |
| t0 |  | quantity = 3 |
| t1 |  | quantity += 1 |
| t2 | quantity =4 |  |
| t3 |  | 事务B发生异常，事务回滚(quantity=3) |
| t4 | update items set quantity=2 where id=1 and quantity=4  这条语句不会更新，因为quantity前后不一样 |  |
| t5 |  |  |

以上更新语句会存在ABA问题

|  |  |  |
| --- | --- | --- |
| 时间 | 事务A | 事务B |
| t0 | quantity = 3 |  |
| t1 |  | quantity = 3 |
| t2 |  | quantity \*= 2 |
| t3 |  | quantity -= 3 |
| t4 |  | commit |
| t5 | quantity = 3，还是为3，更新成功  事务A会认为之中间过程quantity没被修改过  但实际上已经被修改很多次了。 |  |

为了解决ABA问题，可以通过一个单独的顺序递增的version字段，修改方式如下：

|  |  |  |  |
| --- | --- | --- | --- |
| 时间 | 事务A | 事务B | version |
| t0 | quantity = 3 |  | 0 |
| t1 |  | quantity = 3 |  |
| t2 |  | quantity \*= 2 | 1 |
| t3 |  | quantity -= 3 | 2 |
| t4 |  | commit |  |
| t5 | 虽然quantity还是为3，但是verison和第一次读取的时候不一样，所以更新失败。 |  |  |

乐观锁 version是怎么实现的？

案例

|  |  |  |
| --- | --- | --- |
| 时间 | A操作员 | B操作员 |
| t0 | 读取账户余额1000元 |  |
| t1 |  | 读取账户余额1000元 |
| t2 | 增加100元 |  |
| t3 | 提交 |  |
| t4 |  | 扣除50元 |
| t5 |  | 提交 |

账户余额为950元，错误。

给数据表增加一个version字段，具体解决方式：

1. 读取数据时，将次版本号一同读出。
2. 更新数据时，将版本号加1
3. 提交数据版本号 > 数据库表的版本号，予以更新
4. 提交数据版本号 数据库表的版本号，认为是过期数据

根据上面的例子，假设添加了一个version字段，当前version的值为1，账户余额为1000元。A提交完后B提交：

1. A操作员此时将其读出（version=1），并将账户余额增加100元，1100
2. B操作员将数据读出（version=1），并将账户余额减少50元，950
3. 操作员A完成了修改操作，将数据版本加1（version=2），连同账户余额（balance=1100），提交至数据库进行更新。此时，由于提交的version版本 > 数据库表中version版本，数据被更新，数据库表中的version变为2
4. 操作员B完成了修改操作，将数据库版本加1（version=2），连同账户余额（balance=1100），提交至数据库进行更新，此时，由于提交的version版本 数据库表中的version版本，所以认为是过期数据，操作员B的操作被驳回，避免了操作员B基于version=1的旧数据修改的结果覆盖操作员A操作的结果。

乐观锁定义：乐观锁是一种思想，乐观锁的观点是认为并发访问数据过程中发生冲突的概率很小，所以在数据访问、更新过程中都不会取限制其它线程，只有在数据更新提交时才会进行冲突检测，如果发现了数据冲突，就会返回错误给用户，让用户决定该怎么做。

CAS 🡪 ABA问题

version、时间戳

乐观锁 version实现核心语句:

select quantity, version from T where id = 1; // 这里查出来version 是2, quantity =10

业务逻辑中对quantity 修改 quantity = 20

update quantity = 20, version = version + 1 where id = 1 and version = 2; // 这里的version要和第一次查出来的version保持一致。而且update的时候要把version加1。

对乐观锁version值更新时，即version + 1 写入数据库时，需要对利用数据库锁机制，对version加上锁，防止其它线程并发修改，所以最好利用索引来更新version的值，因为sql语句如果找到索引加的是行锁，如果找不到索引，加的是表级锁。

聊一聊MVCC？

MVCC全称是Multi-Version Concurrency Control 即多版本并发控制，主要是为了提高数据库的并发性能，基于InnoDB引擎的来讲，因为MyISAM不支持事务。

同一行数据同时发生读写请求时，会上锁阻塞，但MVCC用更好的方式去处理读写请求，可以不用数据库锁机制。

读分为：当前读，快照读，当前读会对数据加锁。

MVCC就是多版本并发控制，MVCC提出是为了提高数据库的并发性能，提高的方式是让数据库在读写的时候不用加锁，自然就提高了数据库的读写性能。

MVCC主要是处理 读请求，这个读指的是快照读

什么是当前读，什么是快照读？

快照读：普通的select读

当前读: 例如 update c = c + 1 from T where id =1; 第一步需要把id=1这一行数据读出来，再去修改字段c的值，所以类似update、delete 、insert、select … for update 都是当前读

|  |  |  |
| --- | --- | --- |
|  | 快照读 | 当前读 |
| 语句 | select | select for update（排他锁）  select lock in share mode（共享锁）  update（排他锁）  insert（排他锁）  delete（排他锁） |

快照读的实现是基于MVCC的，快照就是数据表的一张“照片”，不是当下最新的数据。

在读和写的时候，不用去竞争锁，因为读锁和写锁本身是互斥的，如果没有MVCC，读和写就会竞争。现在有了MVCC，MVCC的读是快照读，不会和写竞争锁。

在数据库层面，MVCC保证了读和写不会有竞争，提高数据库的并发性能。

什么是undo log、版本链、read view？

undo log 撤销日志，指某条记录的历史版本，可退回的版本

版本链是将某行数据的最新记录与历史记录通过回滚指针连接起来的链条。

readview：就是一个快照。版本链里面有很多版本，readview可以让你知道去版本链里面选择那条记录。



readview数据结构如下，readview就是一个对象

class ReadView{

m\_ids;

min\_trx\_id;

max\_trx\_id;

creator\_trx\_id;

}

|  |  |  |
| --- | --- | --- |
| 字段 | 时刻 | 含义 |
| m\_ids | 生成ReadView对象时，生成快照时 | 活跃的事务id列表。  活跃的事务即未commit的事务 |
| min\_trx\_id; | 生成ReadView对象时，生成快照时 | 在m\_ids列表中，最小值 |
| max\_trx\_id | 生成ReadView对象时，生成快照时 | 在m\_ids列表中，最大值，再加1 |
| creator\_trx\_id | 生成ReadView对象时，生成快照时 | 生成该快照，或者生成该ReadView对象的事务id |

例如：当前有5个并发的事务同时在访问数据库，它们的事务id分别是1-5，在id为2的事务中创建了一个ReadView对象（或快照），其每个字段的值应该是：

m\_ids = [1,2,3,4,5]

min\_trx\_id = 1

max\_trx\_id = 5 +1 = 6

creator\_trx\_id = 2

注意，事务的id随时都是在变化的，字段里面记录的只是创建快照时的情况，下一秒情况就可能变化了。因为事务id时自增的。

当前事务如何通过ReadView判断版本链中那个版本可用？

trx\_id: 版本链中某个记录的事务id。

1. trx\_id creator\_trx\_id 当前事务可以访问该版本记录
2. trx\_id < min\_trx\_id 当前事务可以访问该版本记录
3. trx\_id > max\_trx\_id 当前事务不可以访问该版本记录
4. trx\_id [min\_trx\_id, max\_trx\_id]

4.1 如果trx\_id 在 m\_ids中：当前事务不可以访问该版本记录

4.2 如果trx\_id 不在 m\_ids 中：当前事务可以访问该版本记录



为什么叫undo log 和 redo log?

undo log 和redo log 这两个名字都有意义

后面的log告诉别人，这是一个日志

前面的单词告知了这个日志的作用：

undo 撤销

redo 重做

bin 归档

MVCC如何实现RC和RR？

RC Read Committed

RR Repeatable Read

|  |  |  |
| --- | --- | --- |
| id | name | age |
| 1 | 张三 | 18 |

select \* from user

手动开启两个事务

mysql会给每个事务分配一个id，这个id是唯一的

|  |  |
| --- | --- |
| 事务A，id=20 | 事务B，id=60 |
| 手动开启事务 Begin | 手动开启事务 Begin |
| update hero set name=’李四’ where id=1  update hero set name = ‘王五’ where id =1 | # 更新了一些其他表的记录 |

id=1这条数据对应的版本链：



假设此时

有一个事务id=0的事务

创建

|  |  |
| --- | --- |
| m\_ids | [20,60] |
| min\_trx\_id | 20 |
| max\_trx\_id | 61 |
| creator\_trx\_id | 20 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 某版本 |  |  |  |  |
| 王五 |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

MVCC注意事项？

Multi-Version Concurrency Control是一种方法， 是一种乐观锁，用于实现读提交和可重复读两种隔离级别。针对MVCC不同存储引擎有不同的实现。我们常说MVCC是由mysql的InnoDB存储引擎实现的，而不是由mysql实现的，注意者两者的区别。

MVCC的实现本质：多个并发啊事务同时访问一行数据时，每个数据修改的这行数据的不同版本。InnoDB只需要维护这行数据的版本链，就可以实现一个select操作的并发执行。

具体来说，对于所有的写操作（inser，update，delete）都会创建一个最新的readview，而读操作是去读旧版本的快照。读和写是分离的不会互斥。

什么是undo log？

当我们对记录做了变更操作时，就会尝试一条Undo记录。事务在发生异常回滚或手动回滚时可以回滚到的历史版本记录

对于InnoDB存储引擎，一个主键索引的记录中，一定会有隐藏的两个字段trx\_id和roll\_pointer，这两个字段存储与B+树的叶子节点中，（非叶节点只存储主键）。

trx\_id：只要有任意一个事务对该条记录进行修改，该trx\_id字段就会记录这个事务的id

roll\_pointer，某条记录被修改，该指针会指向上一个历史版本



InnoDB有几种版本链呢？

两种：

1. 插入操作产生
2. 修改操作

注：读操作不会产生版本链，只会根据规则去读取版本链上的记录

MVCC如何实现可重复读？

注意read view中max\_trx\_id值的含义：需要分配给下一事务的事务id

假设事务A （id=20）和事务B（id=30）同时对主键id=1进行操作。

那么这两个事务会创建各自的ReadView

事务A：

|  |  |
| --- | --- |
| m\_ids | [20,30] |
| min\_trx\_id | 20 |
| max\_trx\_id | 31 |
| creator\_trx\_id | 20 |

事务B：

|  |  |
| --- | --- |
| m\_ids | [20,30] |
| min\_trx\_id | 30 |
| max\_trx\_id | 31 |
| creator\_trx\_id | 30 |

读一致性问题？

1. 写的过程一定要加写锁
2. 写不存在什么问题。
3. 读过程不一定加锁，所以会产生一致性读问题

|  |  |  |
| --- | --- | --- |
| 一致性读的问题 | mysql解决办法（悲观锁） | InnoDB解决办法（乐观锁） |
| 脏读 | 读提交隔离级别，加行锁 | MVCC 多个ReadView |
| 可重复读 | 可重复读隔离级别，加行锁 | MVCC 复用一个ReadView |
| 幻读 | 加表锁 | MVCC 不能完全解决，  有特例 |

MVCC解决的是读的问题

|  |  |  |
| --- | --- | --- |
|  | 当前读 | 快照读 |
| 含义 | 写之前的读 | 纯读，普通的读 |
| MVCC能否解决 | 不能 | 能 |
| 针对语句 | select for update  select for delete  update  delete | select |
| 机制 | 1. 加写锁，不允许其他事务修改了   2、读取最新的数据， | 读一个快照，在这个普通读过程中始终用这个快照。 |

对于更新删除语句，跟不会去比较MVCC的事务id，而是上来就加写锁，如果加不上，说明其他事务再操作，当前事务会等待或抛出异常。如果能够加上锁，就开始进行当前读，然后操作，操作结束后释放锁。

|  |  |  |  |
| --- | --- | --- | --- |
| 时间 | 事务A id =20 | 事务B id = 30 | 事务C id = 40 |
| t0 | 开始事务 |  | select \* from from T where id =1 |
| t1 | update T set age += 1 where id =1 | 开始事务 |  |
| t2 | commit |  |  |
| t3 |  | update T set age += 1 where id =1 | select \* from from T where id =1 |
| t4 |  | update T set age += 1 where id =1 |  |
| t5 |  | update T set age += 1 where id =1 |  |
| t6 |  |  |  |
| t7 |  | commit |  |
| t8 |  |  | select \* from from T where id =1 |
|  |  |  |  |
|  |  |  |  |

多个select语句，ReadView会复用，一般不会产生幻读问题。

多个select语句，ReadView会复用，除非它们之间夹杂着修改操作，会产生一次当前读，导致ReadView重新生成，而导致幻读问题。

mysql啥时候用表锁，啥时候用行锁？

1. 当存储引擎不支持行锁时，使用表锁
2. 当mysql语句没有匹配到索引时，使用表锁

|  |  |  |
| --- | --- | --- |
| 类别 | 是否唯一 | 加锁 |
| 无索引 | 非唯一 | 表锁 |
| 索引 | 唯一 | 一行行锁 |
| 非唯一 | 多行行锁 |

mysql中的锁

|  |  |  |
| --- | --- | --- |
|  | 名称 | 使用场景 |
| 全局锁 | 全局锁 | 执行全库逻辑备份时 |
| 表级锁 | 表锁 | 1. 存储引擎不支持行锁时 2. sql语句没有匹配到索引时 |
| 元数据锁 | 1. 对表做增删改查时，会加锁MetaData Lock 读锁 2. 对表结构做变更时，会，会加上MDL写锁 |
| 意向锁 | 对表中的行记录加锁时 |
| 行级锁 | 行级锁 | 增删改查匹配到索引时，会使用行级锁 |

DDL、DML、MDL？

DDL：Data Definition Language

DML：Data Manipulation Language

MDL：MetaData Lock

|  |  |  |
| --- | --- | --- |
|  | DDL | DML |
| 针对语句 | create, alter,drop, | insert update delete select |
| 针对对象 | 表结构 | 表中数据 |

聊一聊全局锁？

全局锁就是对整个数据库进行加锁。

mysql提供了一个全局加读锁的方法：

flush table with read lock

FTWRL

一旦执行这个命令，以下语句会被阻塞：

数据更新语句

数据定义语句

更新类事务的提交

在做全库逻辑备份时，可以用这个命令锁住全库的变更操作，只有读操作可以进行。

在做全库逻辑备份时，不是看重数据是不是最新的，而是看重在备份过程中不会出现一致性读的问题，也就是读的时候前后一致。所以，对于MyISAM存储引擎，只能通过加全局锁来保证全局的一致性读。而对于InnoDB，在做全库逻辑备份时，不必要使用FTWRL命令，因为MVCC既可以解决一致性读的问题，不会出现脏读、不可重复读，在保证连个select语句中没有更新语句时不会出现幻读。

聊一聊表级锁？

表锁，顾名思义，就是对某个表加锁。

锁的范围越大，颗粒度越粗。

所以我们说，MyISAM不支持行级锁，所以只能用更粗颗粒度的锁，表锁来实现加锁。

对于MySQL没有匹配到索引的情况：mysql无法确定哪些行需要加锁，需要一行一行的遍历，比较值才行，所以干脆加个表锁。

对于MySQL匹配到索引的情况：可以快速定位到哪一行或哪几行需要加锁。使用行级锁可以更细的颗粒度管理。

聊一聊元数据锁？

元数据指描述数据的数据。在mysql元数据表指描述表结构这些数据的表

比如有一个字段类型是什么，长度是什么，是否是主键等

当对表数据进行增删改查时，需要加上MDL读锁，不允许其他线程加写锁，即不允许其他线程修改表结构。

当对表结构进行修改时，加上MDL写锁，不允许其他线程加读锁和写锁，即不允许其他线程对表数据进行增删改查且不允许其他线程对表结构进行修改。

聊一聊意向锁？

在InnoDB，由于有行锁的存在，我们在想要给整张表加排他锁的时候，需要遍历每行，然后查询是否加上了锁，如果一张表里面所有的行都没有锁，才可以对这个表加排他锁。

这个遍历的过程很麻烦，为了解决这个问题，提出了意向锁：

1. 如果某张表有意向锁，说明该表的某行记录被锁住了
2. 如果某张表没有意向锁，说明该表没有行被锁住

不用遍历所有行，只需查询该表的意向锁。

在对表中某行记录加锁时，就会用到意向锁。

聊一聊行级锁？

行级锁是存储引擎级别的锁，需要存储引擎支持才有效，InnoDB支持行级锁。

表级锁是mysql server层的，所有存储引擎共享这个功能。

假设使用的是InnoDB数据，

如果sql语句可以匹配到索引🡪行级锁

如果sql语句匹配不到索引🡪表级锁。

你是怎么理解事务的？说一下你的理解？

事务是一组对数据库的操作，要么全部成功，要么全部失败。一些涉及对数据库数据修改的业务，可能有多个步骤，如果对数据库的写入已经生效，而发生了异常，如果没有事务，就不能回滚大修改前的状态。事务保证了在做一系列连续动作时状态的一致性，如果一个事务中有任何地方发生异常，都会导致修改失败。

事务的四大特性了解吗？介绍一下解释一下？

ACID

Atomicity 原子性，指一组操作要么同时成功，要么同时失败。

Consistency 一致性，指事务开始到提交应该时一个一致性的状态转到另一个一致性的状态，比如甲乙各有500元，甲向乙转了50元，事务开始前甲乙一共1000元，转账成功后双方也应该有1000元。

Isolation 隔离性，并发的事务操作应该不会相互影响。

Durability 持久性，任何提交的事务产生的修改都应该永久地保存到磁盘中，即使数据库崩溃了，也能保留这些事务产生的修改。

事务有好几种隔离级别，介绍一下常见的有哪些？主要解决了哪些问题呢？又各自存在哪些问题呢？

|  |  |  |  |
| --- | --- | --- | --- |
|  | 脏读 | 不可重复读 | 幻读 |
| 读未提交 | ✓ | ✓ | ✓ |
| 读提交 | × | ✓ | ✓ |
| 可重复读 | × | × | ✓ |
| 串行化 | × | × | × |

快照了解吗？介绍一下？

快照：某一时刻数据状态记录

备份：某一时刻数据的副本

快照记录了某一时刻数据库的状态。

快照读在提交读和可重复读级别下有什么区别？

在读提交的隔离级别下，一个事务中如果有多次读取，每次都要创建一个ReadView。

在可重复读的隔离级别下，多次快照读共享一个ReadView

mysql不同隔离级别，都使用了什么锁？

事务隔离级别：是为了解决并发访问数据一致性问题。

不同的事务隔离级别，解决了不同程度的数据一致性。

全局锁、表锁、行锁； MVCC：事务隔离级别的具体实现

其中mysql锁是基于悲观锁的思想

MVCC是基于乐观锁的思想。

总结：

1. 在读未提交的隔离级别下，读写操作可以同时进行，但是写写操作不可以。该隔离级别下只会用到行级的记录锁，并不会使用间隙锁。
2. 在读提交隔离级别下，读写操作可以同时进行，写写操作不可以同时进行。解决了脏读问题。只会使用记录锁
3. 在可重复读隔离级别下，读写操作可以同时进行，写写操作不可以同时进行。解决了不可重复读问题，会使用记录锁、间隙锁和Next-Key锁。

写的读为什么不会出现各种问题？

普通读会出现各种问题是因为普通读一般不想加锁

写的读是当前读，先加写锁，加上之后别的事务一定修改也读取不了数据了，现在在慢慢去读当前最新的数据。所以写的读=当前读=先加写锁，如果加上了在慢慢读取最新的数据。所以当前读不会有那些问题。

MVCC也可以解决脏读问题，那他具体是如何解决的？

|  |  |  |
| --- | --- | --- |
| 时间 | 事务A, id = 30 | 事务B, id = 20 |
| t0 |  | 开始事务 |
| t1 | 开始事务 |  |
| t2 |  | update T set c = 20 where id = 1 |
| t3 | select \* from T where id = 1 |  |
| t4 |  | 事务B发生异常，回滚 |
| t5 | commit |  |



InnoDB的行锁？

记录锁：某个索引记录的锁。

间隙锁：两个索引记录之间的空隙锁

Next-Key锁：记录锁+间隙锁

MySQL 读已提交可以解决脏读问题，那它具体是如何解决的？

auto commit作用？

|  |  |  |
| --- | --- | --- |
|  | sql语句 | 事务个数 |
| @@autocommit=1 | select \* from T where id = 1  update T set c = 2 where id = 1  delete from T where id = 1 | 3个 |
| begin;  select \* from T where id = 1  update T set c = 2 where id = 1  delete from T where id = 1  commit; | 1个 |
| @@autocommit=0 | select \* from T where id = 1  commit;  update T set c = 2 where id = 1  commit;  delete from T where id = 1  commit; | 3个 |
| select \* from T where id = 1  update T set c = 2 where id = 1  delete from T where id = 1  commit; | 1个 |

不同锁的关键字？

|  |  |  |
| --- | --- | --- |
| 类型 | 锁名 | 关键字 |
| 表级锁 | 意向排他锁 | lock mode IX |
| 插入意向锁 | lock\_mode X locks gap before rec insert intention |
| 记录锁 | lock\_mode X locks rec but not gap |
| 行级锁 | 间隙锁 | lock\_mode X locks gap before rec |
| Next-Key锁 | lock\_mode X |

mysql server 服务层和数据库innodb 存储引擎层各自有什么

|  |  |  |
| --- | --- | --- |
| server层 | binlog | 所有存储引擎共享 |
| innodb 存储引擎 | redo log，undo log、MVCC、事务 | innodb存储引擎特有 |

串行话隔离级别是怎么实现的？

串行化是mysql最高的隔离级别，可以解决绝大多数并发访问的数据冲突问题，但是效率也最低。串行化读会加读锁，写会加写锁。当读写发生冲突时，后一个到的必须等正在占用锁资源的事务释放锁。

以下操作在不同隔离级别下V1，V2，V3分别是什么值？



|  |  |  |  |
| --- | --- | --- | --- |
|  | V1 | V2 | V3 |
| 读未提交 | 2 | 2 | 2 |
| 读已提交 | 1 | 2 | 2 |
| 可重复读 | 1 | 1 | 2 |
| 串行化 | 1 | 1 | 2 |

|  |  |
| --- | --- |
|  | MVCC |
| 读未提交 | 没有视图概念 |
| 读已提交 | 在每个普通读SQL语句开始执行时创建视图 |
| 可重复读 | 一个事务里的所有连续的普通读的sql语句复用一个视图，即第一个普通读的sql语句执行时创建的视图 |
| 串行化 | 没有视图概念，直接上锁 |

|  |  |
| --- | --- |
|  |  |
| 读已提交 | 事务{  普通读sql\_1; // 创建一次视图  普通读sql\_2; // 创建一次视图  普通读sql\_3; // 创建一次视图  当前读sql\_4; // 先上写锁，上锁成功读取当前最新数据  普通读sql\_5; // 创建一次视图  } |
| 可重复读 | 事务{  普通读sql\_1; // 创建一次视图  普通读sql\_2; // 复用第一个视图  普通读sql\_3; // 复用第一个视图  当前读sql\_4; // 先上写锁，上锁成功读取当前最新数据  普通读sql\_5; // 由于当前读了一次，创建一次视图（这也是MVCC幻读可能发生的地方）  } |

什么时候需要“可重复读”的场景呢？

假设正在管理一个人的银行账户表。

一个表存了账户余额

一个表存了账单明细。

到月底做数据校对，判断上个月余额和当前余额的差值，就得到了这个月用了多少钱，然后把这个差值于账单明细做校对，校对的过程中，即使发生了一笔新的交易，由于差值已经确定了，所以我不关心最新的数据，我不希望看到当前的账单明细有改动，这就产生了可重复读的要求，从而产生了可重复读的隔离级别。这就是可重复读的场景。

undo log 注意事项？

undo log叫回滚日志

从MVCC机制角度说明为什么不要使用长事务？

回滚日志在内存中不会一直存在，过一段时间就会删除，具体的删除时机时：

当没有比该历史记录更早的readview时就删除。

而一个长事务，在系统里可能会存在很老的readview，由于这个事务可能随时访问数据库里的任何日志，所以在这个事务提交前，数据库里它所有可能用到的回滚日志都需要保留，这会导致大量的内存空间被占用。

事务启动方式，到底使用哪一种？

|  |  |  |
| --- | --- | --- |
|  | 显示启动 | 隐式启动 |
| autocommit | 1 | 0 |
| 优点 | 避免了忘记写commit导致长事务 | 1. 每句sql不会自动提交，而是等到手动commit时才提交 2. 少一次begin命令，少一次交互 |
| 缺点 | 1. 如果没显示begin，commit，每句sql都是一个事务，会带来性能问题 2. 事务多一次begin命令，会多一次交互 | 如果忘了写commit，会导致长事务 |

删除使用自动提交，即autocommit=1，因为长事务带来的问题太严重了。而且对于多一次交互问题，可以解决：

在autocommit=1的情况下，用begin显示的启动事务，如果执行commit则提交事务。如果执行commit work and chain，则是提交事务并自动启动下一个事务。省去了执行begin语句的开销，同时带来的好处是明确的知道每个语句是否处于事务中。

可以通过以下命令查询持续时间超过60s的事务：

select \* from information\_schema\_innodb\_trx where TIME\_TO\_SEC(time\_diff(now(),trx\_started))>60;

聊一聊B树？

B树是多路平衡查找树。一棵m阶的B树指每颗结点最多有m个子节点。

为什么要有B树，红黑树不行吗，红黑树作为优秀的平衡搜索树，在很多地方都是用到，但是用作索引时一般不采用红黑树，因为红黑树是二叉树，但结点大量增加时数会变得非常高，如果结点数据都在内存中那道没什么，但是索引存在于磁盘块中，从开始出发，探索一层，就要从磁盘中读以下索引，进行一次IO，而当树非常高时，大量的IO操作会降低性能。所以引入了多路平衡查找树，解决结点很多时，二叉树变得很高导致的IO性能问题。

从磁盘查找数据效率低，一般是什么原因？

首先的弄懂cpu怎么从磁盘找数据：由于cpu的速度远远大于磁盘的速度，所以cpu和磁盘不能直接交互。所以，引入了内存，将磁盘中的数据先加载到内存中，再由内存，然后内存去执行cpu的指令。在这个过程中，数据查找效率低最大的原因是磁盘到内存的相互交互IO较低。

而影响IO的因素：

1. 读写的文件大小
2. 读写的次数

针对第一个点，sql可以优化，select \* 变成 select 特定字段

针对第二个点, 线性遍历查找->二叉搜索树->平衡二叉搜索树->B树->B+树

|  |  |
| --- | --- |
|  | 读写次数 |
| 线性遍历查找 | n |
| 红黑树 |  |
| B树 | ，这个m做不到很大很大 |
| B+树 | ，这个m可以很大很大 |

索引：为了更快地查询数据。key-value

文件系统的索引，数据库的索引就是这种键值对的设计

设计一个文件系统的索引（或者数据库索引），该如何设计呢？

1、用线性的方式不适合。

2、用哈希表的方式，文件系统的索引基本上是key-value键值对形式实现的。



这种哈希表方式查找某个索引的的效率是o(1)

但是不支持范围查找，比如查询2<id<4的范围内的索引，需要退化成线性查找。

哈希表作为文件系统索引方式的缺点

1. 哈希冲突后散列不均匀，产生大量线性查询
2. 等值查询可以，但是范围查询就得挨个遍历

优点是：等值查询块

树，二叉树，二叉查找树，平衡二叉树，红黑树，B树，B+树

二叉树本身是没序的，是遍历查找

二叉查找树可以用二分法查找，log2(n)，二叉树的问题是

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | 优点 | 问题 |
|  | 线性表 |  | O(n)时间复杂度 |
|  | 哈希表 | 等值查询块 | 1. 范围查找退化成线性遍历 2. 当哈希冲突后散列不均匀退化成线性遍历 |
|  | 二叉树 |  | O(n)时间复杂度 |
|  | 二叉查找树 | log2(n)时间复杂度 | 极端情况下，比如插入多个有序数据，会退化成线性遍历 |
|  | AVL | 解决了二叉查找树极端问题 | 插入，删除旋转过多，影响性能 |
|  | 红黑树 | 不要求完美平衡，减少旋转次数 | 当key很多时🡪树的高度会增加很快🡪查找路径上的结点会变多🡪IO次数会增多 |
|  | B树 | m叉，树的深度降低 | 由于内存限制，m的值不能很大 |
|  | B+树 | m可以很大 |  |

文件系统的索引是保存在磁盘当中的，每一个结点是一个索引，放在磁盘中，读到那个结点，就要去将该索引所占的磁盘块读入内存，当树变高，查找到目标节点的路径边长，路径上的节点变多，IO次数变多。

B树：

1. 每个节点最多有m个孩子节点（m-1个关键字）

磁盘预读：

1. 内存的最小逻辑单元是页
2. 页的大小由操作系统决定，一般是4k或8k，在进行数据交互时，可以取页的整数倍来读取

假设内存页的大小时4k，我们要读取2.78k的数据的话，需要读取4k的数据。

一般说要把页大小和磁盘块大小设置为一样

B树为什么不能把m设得很大：

B树，非叶子节点存索引+指针+数据，叶子节点存数据

B+树，非叶子节点只存索引和指针，叶子节点存数据

什么是索引？

索引是一种帮助mysql提高查询效率的数据结构。索引就是key，数据就是value。key的组织方式有多种：线性表、哈希表、树（二叉排序树，AVL，红黑树，B树，B+树）。本质上是对磁盘块的组织方式。

索引的优点: 大大增加查询的效率

索引的缺点：维护索引需要消耗数据库资源；索引需要占用磁盘空间；对数据进行增删改，因为要维护索引，速度受到影响。

索引的数量不是越多越好，通常只对需要经常搜素的字段设置索引。

MySql中有哪些索引？

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 存储引擎 | InnoDB | | | | MyISAM |
| 索引名称 | 主键索引，聚簇索引 | 单值索引，单列索引，普通索引 | 唯一索引 | 复合索引 | Full Text全文索引 |
| 定义 | 创建主键时自动建立主键索引 | 即一个索引只包含一个列 | 索引值必须唯一 | 一个索引包含多个列 | 类似于elastic search |
| 注意事项 | 主键索引不允许有空值 | 1. id name (index) age(index) 2. 组合查询where name and age时用不到 | 允许有空值 | id (name age) index  在复合条件时可以用到索引  where name and age |  |

复合索引注意事项？

创建复合索引语句：

create index multil\_index on t(name,age,sex)

按name，age，sex这个顺序创建的复合索引

在查询的时候如果基于name，age 查能否使用索引：

1. 最左前缀原则
2. mysql优化器会优化sql语句，会动态调整where语句的查询顺序以便都用上索引

经典面试题

|  |  |  |
| --- | --- | --- |
|  |  | name，age，sex索引 |
| 不需要调整的基础情况 | name | 可以 |
| name，age | 可以 |
| name，age，sex | 可以 |
| 需要mysql优化器调整，但不一定都管用 | 其它 | 1. 如果经过优化器优化，sql语句顺序能符合最左匹配原则就可以 2. 如果经过优化器优化，sql语句顺序不能符合最左匹配原则则不可以 |

主键索引为什么要排序？

为了提高查询效率，就可以按照顺序搜索。

mysql页的大小默认是16KB

mysql是按页管理索引的，一次读取一页的索引数据

不管你磁盘是多少，但通常，我们将磁盘的大小设置成和页大小一致

B+树的优点:页数据，页目录区分开来

B树：页数据和目录没区分

mysql innodb引擎一页的大小默认是16kb。

页目录只保存记录和指针，不保存数据，假设主键记录是int，大小是4个字节，指针是6个字节，则一个记录大概要存10个字节，由于页大小是16KB，所以每个页目录可以存1600个记录，如果非叶子节点有三层，就可以存1600的三次方吗，四十几亿条数据。

B+树页目录由于只存关键字和指针，不存数据，可以存很多条记录，结果就是每个非叶子节点可以有大量的分叉，两三层高度树就可以存储大量的数据

树根，最顶层的页是常驻内存的，不用走磁盘

一个三层树高的B+树，顶层是常驻内存的

查询一个数据，按主键索引，是两次IO，但是如果是按非主键索引，首先第一次会按非主键索引找到其对应的主键，在去主键索引查两次，索引总共是三次IO。

你能给我说说mysql索引的底层实现吗？

mysql索引底层是由B+树实现的。

B+树是一种多路平衡查找树，节点之间是有序的，一颗m阶的B+树最多有m个孩子节点。B+树中非叶子节点只存放记录和指针，叶子节点存放记录和数据。一个叶子节点内部是有序的，叶子节点之间也是有序的，用指针连接。

为什么主键用int?

因为int方便排序和比较。

聊一聊聚簇索引和非聚簇索引？

聚簇索引 clustered index

clustered：成群的，聚集在一起的

将数据的存储和索引放到了一块，索引结构的叶子节点保存了行数据。

主键索引一定是聚簇索引，但是聚簇索引不一定是主键索引。

非聚簇索引：将索引与数据分开存储，索引结构的叶子节点指向了数据对应位置。

注意：一个表中只能有一个聚簇索引，剩下的索引都是非聚簇索引（单值、复合、唯一）

为什么辅助索引不报错数据的地址？

因为增删改的时候会修改数据的地址，主键索引里面要修改一次，辅助索引里面也要修改一次。



辅助索引总是需要二次查找。

聚簇索引和非聚簇索引的区别？

聚簇索引：clustered index，它们区别如下:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 存储引擎 | 使用的索引类型 | 实现索引的数据结构 | 叶子节点 | 辅助键索引 |
| InnoDB | 聚簇索引 | B+树 | 存放数据 | 依赖主键索引 |
| MyISAM | 非聚簇索引 | B+树 | 存放地址 | 不依赖主键索引 |

聚簇索引：将数据存储与索引存储放到一起，索引结构的叶子节点保存了数据。

非聚簇索引：将数据存储和索引存储分开，索引结构的叶子节点保存了数据的真实地址。



不能说主键索引就是聚簇索引，而是说基于聚簇索引的主键索引和基于非聚簇索引的主键索引。

InnoDB根据索引查找一个数据的过程？

首先，InnoDB是聚簇索引，根据索引查找数据要分两种情况：

1. 如果使用主键索引查询数据，会从B+树的顶层节点开始，在实现上，InnoDB会把B+树的顶层节点常驻内存，所以这一次不需要IO，从顶层节点出发找到对应的叶子节点，然后将将该页数据加载进内存，查询目标数据。
2. 如果是辅助键索引，前面的过程和主键索引的B+树一样，但是辅助键索引对应的B+树叶子节点并不保存数据，而是保存该辅助键对应的主键，然后拿着主键做一次主键索引。也就是辅助键索引不能直接查找到数据，而是多一次主键索引查找。

InnoDB 聚簇索引注意事项？

InnoDB中聚簇索引默认是主键，如果表中没有定义主键，InnoDB会选择一个唯一且非空的索引代替，如果没有这样的索引，InnoDB会隐式地定义一个主键(类似Oracle的rowid)来作为聚簇索引。

如果使用了主键索引，还想单独设置聚簇索引，需要将主键索引删除，然后设置聚簇索引，最后再把主键索引恢复。

MyISAM存储引擎根据索引查询数据时是怎么样的？

由于MyISAM使用的是非聚簇索引，所以在在MyISAM中，主键索引的B+树和辅助键索引的B+树是一样的，叶子节点存储了数据地址，而没有存储数据。在查询时，直接走一次B+树的查询逻辑即可，辅助键索引不会像基于聚簇索引的辅助键索引那样多一次查询。

既然基于聚簇索引的辅助键索引还多查一次，为什么还要使用聚簇索引呢？

1. 由于聚簇索引是把索引和数据一起存储的，将叶节点读取到内存后，数据就直接在内存中了，下一次读取时，直接去内存中读就可以了。而非聚簇索引，将数据和索引分开存储，将叶子节点读取到内存后，还要根据叶子节点存储的磁盘真实地址去磁盘中读取数据，下一次读取到叶子节点，还是要重新做一次IO。
2. 对数据进行插入、删除操作时，可能涉及到节点的分类，B+树结构调整，此时基于聚簇索引的辅助键索引，由于不直接指向数据，而是保存对应的主键，所以不需要维护，只需维护主键索引的B+树即可。而基于非聚簇索引的辅助键索引在数据地址生变化时也需要维护，所以就要维护两次。

聚簇索引的优势？

1、由于行数据和聚簇索引的叶子节点存放在一起，同一页中会有多条行数据，访问同一页不同行记录时，已经把页加载到了内存。再次访问时，会在内存中完成访问，不必访问磁盘。

1.1 两次访问同一行数据，聚簇索引和非聚簇索引都可以在第二次时在内存中直接返回

1.2 两次访问不同行数据，聚簇索引：如果这两行数据在同一页（大概率），第二次就可以不用访问内存。非聚簇索引，由于此时内存中存储的是叶子节点和第一次访问的行数据，而叶子节点中存储的是行数据的地址值，所以需要再次读取磁盘。

2、辅助索引的叶子节点，存储主键值，而不是数据存放地址。

2.1 当数据发生变化时，索引树节点也需要分裂变化，辅助键索引的不需要改变。

2.2 当上一次IO读写缓存中没有此次要查询的数据，需要发送一次新的IO时，可以避免对辅助键索引的维护工作。

3、辅助键索引存放的时主键值，减少了占用空间大小。

为什么通常使用自增id？

1. 因为使用自增id，每次向B+树插入节点时，会插入到最又端的叶子节点中，而不会插入到中间，这样对B+树的结构破话最小，B+树分裂重建的的次数减少。
2. 由于聚簇索引的物理顺序和逻辑顺序时一致的，就是在逻辑上id=2在id=1的后面，那么在物理磁盘上id=2的数据一定在id=1的前面，这样可以提高查询效率，一旦找到了数据的开始位置，就可以连续地往后读，无需在磁盘上跳跃寻找。
3. 使用自增的数据，在磁盘上一页一页连续地写，更好地利用磁盘空间，避免磁盘碎片。

什么情况下无法使用索引？

1. like关键字，%在最前面时，如果%不在最前面，可以用索引
2. 复合索引，不满足最左前缀，以及经优化器优化后还是不满足最左前缀
3. 查询语句使用OR关键字，且OR的左右有一个没有索引，就不会使用索引。

背包问题？



例题，当每个物品数量是一个时，最大价值是多少；每个物品数量是无数个时，最大价值是多少？

|  |  |  |
| --- | --- | --- |
| 背包最大容量为4 | 重量 | 价值 |
| 物品0 | 1 | 15 |
| 物品1 | 3 | 20 |
| 物品2 | 4 | 30 |

01背包的情况

dp[i][j] 编号为0-i的物品可选且背包最大容量为j时，最大价值为dp[i][j]

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 |
| 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 |  |  |  |  |
| 2 | 0 |  |  |  |  |

什么是索引？

索引是一种数据结构，可以大大加快mysql的查询顺序。

索引类似于目录，可以通过索引快速定位数据的位置。

索引分类？

主键索引，唯一，不允许为空

单值索引=普通索引=单列索引

唯一索引：唯一，允许为空

复合索引

Full Text 全文索引

create multiple\_index on T.(name, age, bir)

1. 最左前缀原则
2. mysql优化器优化

索引的基本操作？

在建表时添加索引

在建表后添加索引

索引的底层实现原理？

索引底层实现是用B+树。

非叶子节点只保存记录和指针，叶子节点保存数据，叶子节点之间是有序的并用指针连接，叶子节点内部也是有序的。

为什么数据明明没有按顺序插入,为什么查询时却是有顺序呢?

由于mysql底层索引实现是一棵B+树，而B+树是一棵排序树。插入到页的数据，mysql会根据数据大小插入到相应的位置。

1、mysql底层为主键自动创建索引

为了进一步提高效率mysql索引又进行了优化？

使用页目录。

索引结构称之为B+树数据结构,那么什么是B+树呢?

B+树是基于B树优化来的，B树是一个多路平衡查找树。一颗m阶的B树每个节点最多又m个孩子节点（m-1个关键字），根据比较记录的大小，放在不同的节点。B+树与B树最大的不同就是非叶子节点只保存记录和指针，不保存数据，而叶子节点用来专门保存数据。这可以大幅提高B+树每个节点可以分叉的数量，从而在较低的树高就可以存储大量的数据。

B+树是在B树上的一种优化，使其更适合存储索引结构。InnoD存储引擎就是用B+树实现索引结构。

B树是多路平衡查找树，一颗m阶的B树每个非叶子节点最多有m个孩子。B树的结点中不仅包含key值，指针，还包含数据。每一页是一个节点，而页的大小是有限制的，当数据较大时，一页存储不了多少条记录。如果存储的数据量很大， 会导致树高变大，树变高会增加查询路径上的节点，导致磁盘IO变多，从而降低了查询性能。

B+向较于B树有几点不同：

1. 非叶子节点只保存key值和指针
2. 所有数据保存在叶子节点中
3. 叶子节点之间用指针连接

由于B+树非叶子节点只保存key值和指针，一页可以保存更多的记录，一个节点可以分叉的数量大大增加，用较低的树高就可以存储大量的数据。而树的高度越低磁盘IO次数越少。从而提高了查询性能。

什么是聚簇索引和非聚簇索引？

聚簇索引是指将索引和数据存储在一起，数据保存到叶子节点。

非聚簇索引是指将索引和数据分开存储，叶子节点保存数据在磁盘中的地址。

InnoDB是基于聚簇索引。

MyISAM是基于非聚簇索引。

使用聚簇索引的优势？

1. 聚簇索引将数据和索引存储到一起，非叶子节点存储了多条行记录，第一次查询某条行记录是，innoDB会按页读取，也就是将整个叶子节点读入内存，而叶子节点页存储了其他行数据，下一次如果查询到同页的不同行数据时，就可以直接从内存中返回，而不需要再读一次磁盘。非聚簇索引由于叶子节点保存的数据在磁盘中的数据，即使把叶子节点那一页读入内存，查询不同行数据时，还是需要去读一次磁盘。
2. 基于聚簇索引的辅助键索引使用B+树结构，叶子节点保存的不是数据，而是该辅助键对应的主键，在利用辅助键查询时，会先去辅助键对应的B+树，走一次查询算法找到对应的主键，然后拿着主键去主键索引的B+走一次查询算法。而基于非聚簇索引的辅助键索引，叶子节点保存的也是数据的地址，也就是辅助键索引树和主键索引树没有什么区别。在利用辅助键查询时，只查询一次，不用在去主键索引树查询。虽然非聚簇索引比聚簇索引要少一次，但是我们还是用聚簇索引，是因为在对数据进行插入删除操作导致树结构变化，数据地址发送改变时，聚簇索引只需要维护主键索引即可，而非聚簇索引则需要维护主键索引和辅助键索引。

聚簇索引需要注意什么?

1. 主键类型不要使用uuid这种不好排序，离散特别大的数据类型
2. 主键类型通常选择自增，int，占用内存不要太大。

为什么主键通常建议使用自增id？

1. 自增id，每次插入数据都是在B+树的最有端，对树的结构破坏最小。减少分裂调整的次数。
2. 由于聚簇索引的逻辑顺序和物理顺序是一致的，也就是说在逻辑上id=2的数据一定在id=1的数据之后，在物理磁盘块上也满足这样的顺序，id=2的数据一定在id=1的数据之后，所以在查询的时候只需要找数据开始的位置，往后面连续地读磁盘，不要跳跃寻找，提高性能。连续的存储数据也可以降低磁盘碎片，提高磁盘利用率。

什么情况下无法利用索引呢?

1. like语句%在最前面时
2. 复合索引不满足最左前缀原则，且优化后不满足最左前缀原则
3. or左右有一个没有索引。

到底什么是索引，索引是如何工作的？

索引是一种数据结构。索引的出现是为了提高数据的查询效率，就像目录一样。对于数据库表而已，索引其实就是它的目录。

实现索引的方式有很多：

1. 哈希表，哈希表是一种键值对形式存储的数据结构。通过关键字key，可以找到对应的值value。哈希表利用数组支持随机访问数据，将利用哈希函数将关键字key映射到数组下标，然后将value放在对应的索引下标的位置。哈希函数不可以避免的会出现哈希冲突，即多个key映射到同一个下标位置，为了解决这个问题，常用的解决办法是：开放定址法、链地址法。常用解决冲突的方法是链地址法，即数组的每一个索引位置都是一条链表，对应哈希值相同的原始会被存储在同一个索引位置的链表上。

2、

|  |  |  |
| --- | --- | --- |
|  | 优点 | 缺点 |
| 哈希表 | 1. 做等值查询很快 2. 插入删除数据很快 | 1. 哈希冲突增大，链表变长时退化成线性查询 2. 做范围查询需要一条一条遍历，退化成线性查询 |
| 有序数组 | 1. 支持等值查询 2. 支持范围查询 3. 折半查找效率log(n) | 插入、删除记录需要移动数组元素，效率低 |
| 搜索树 | 查询效率 log(n)  插入和删除效率 log(n) |  |

为什么二叉搜索效率很高了，还是要用多叉树？

因为索引不仅在内存中，还要写入磁盘。

对于一个100万节点的平衡二叉树，树高20，一次查询可能要访问20个磁盘块。如果一个磁盘块访问需要10ms，则一次查询可能需要20\*10ms的时间

为了让一次查询减少访问磁盘块的次数，必须减少路径长度，即降低树高。采用m叉树可以降低书高。

索引模型就是实现索引不同的方式。

索引是存储引擎层实现的，不同的存储引擎实现索引的方式不一样。不同索引有不同的索引标准，对于同一索引标准，不同索引实现方式也可能不一样。

MySQL server层存储引擎层各自有什么？

|  |  |
| --- | --- |
| server层 | 核心函数、bin log，表级锁 |
| 存储引擎层 | redo log、undo log、事务、索引，MVCC，行级锁 |

聊一聊InnoDB的索引模型？

InnoDB存储表的方式是索引组织表，即表都是根据主键的顺序，以索引形式存放的，索引和数据存放在一起（聚簇索引）。而InnoDB使用B+树这种索引模型，所以数据都是存储在B+树中。

聚簇索引，索引组织表表达的含义是一样的。

索引组织表：表中的数据按照主键的顺序，以索引的方式存储在磁盘中。

聚簇索引：数据和索引存放在一起，存储结构的非叶子节点存放记录和指针，叶子节点存储数据。

存储方式是索引组织表 + 索引模型是B+树 🡺 InnoDB的数据存放在B+树中

根据以下建表语句，描述底层数据结构以及查询数据过程？

mysql> create table T(

id int primary key,

k int not null,

name varchar(16),

index (k))engine=InnoDB;

1. 两个索引：主键索引和单值索引。
2. 每一个索引都有一颗B+树，所以有两颗B+树
3. 存储引擎是InnoDB，主键索引和辅助键索引是基于聚簇索引的。即主键索引的B+树叶子节点存储数据，辅助键索引B+树叶子节点不保存数据，而是保存该辅助键对应的主键。
4. 查询时如果是利用辅助键索引会多查一次，将主键查出来后，还会做一次主键索引的B+树查询算法。这个过程叫回表

主键索引，聚簇索引

非主键索索引，辅助键索引，二级索引

什么叫回表？

回表是指如果索引是聚簇索引类型，利用辅助键索引查询数据并不会之间查到数据，而是查到对应的主键，在拿主键去主键索引查询一次。

B+树的维护？

为了维护B+树的有序性，在插入新值是需要进行维护，如果插入的数据的key值在一页的中间，需要往后挪动数据，如果插入后一页满了，需要进行页分裂，页分类会影响性能以及空间的利用率，由原来的一页，分散成两个页，空间利用率为50%。当然，有页分裂，就有页合并，由于两页删除了数据，空间利用率很低，就会在合适的时候进行合并。可以认为是分裂的一个逆过程。

主键的大小不宜过大，因为二级索引的叶子节点存储的是主键的值，如果主键过大，叶子节点上就存储不了多少。

主键长度越小，普通索引的叶子节点就越小，普通索引占用的空间就越小。

为什么要尽量使用主键索引进行查询？

因为用主键索引查询只需查找一棵树。

分析以下sql语句，在下面这个表 T 中，如果我执行 select \* from T where k between 3 and 5，需要执行几次树的搜索操作，会扫描多少行？

mysql> create table T (

ID int primary key,

k int NOT NULL DEFAULT 0,

s varchar(16) NOT NULL DEFAULT '',

index k(k))

engine=InnoDB;

insert into T values(100,1, 'aa'),(200,2,'bb'),(300,3,'cc'),(500,5,'ee'),(600,6,'ff'),(700,7,'gg');

分析：主键是ID，int类型。字段k是单列索引

select语句使用辅助键索引，查询3到5的行数据。

首先在k索引对应的B+树上会去查询3到5的数据对应的主键：300，500

然后拿着主键去主键索引对应的B+树上找行记录所在的磁盘块，读入内存。

你描述的过程需要回表，又没可能优化，避免回表呢？

有。可以使用覆盖索引，select \* from T where k between 3 and 5；我们可以考虑select后面是不是必须写\*，如果我们只需要ID数据，可以写成select ID from T where k between 3 and 5；这样就避免了回表，因为二级索引对应的B+树叶子节点存储的就是对应ID值。

以上过程，通过修改我们要select的东西，使得索引覆盖了我们查询需求，这就是覆盖索引。



由于覆盖索引可以避免回表，减少了搜索树的次数，显著提升了查询性能，所以使用覆盖索引是一种常用的优化性能的手段。

InnoDB 范围查询的查找逻辑？

对于 where k between 3 and 5，

1. 首先找到k=3这条记录
2. 然后从k=3往后读：这里有两个方面的原因：a)聚簇索引的逻辑顺序和物理顺序是一致的，逻辑上k=4的记录在k=3记录之后，物理上也是一样的。b)B+树的叶子节点内部是连续挨着的，叶子节点之间是用指针相连。
3. 所以往后读，直到第一条不满足3 and 5条件的数据出现，结束。

对于 where k < 5

从最小的数据往后读，直到出现不满足where条件的k

对于 where k > 3

从k=4开始读，直到读取完所有数据。

在一个市民信息表上，是否有必要将身份证号和名字建立联合索引？

(id\_card, name)这种联合索引

优点：建立联合索引，后如果通过身份证查询名字，根据最左匹配原则，会用到索引。

复合索引为什么要用最左匹配原则？

这要结合符合索引的原理来说明

| **name** | **age** |
| --- | --- |
| Alan | 20 |
| Alan | 25 |
| Bob | 22 |
| Bob | 28 |
| Carl | 21 |
| Carl | 23 |

加了一个联合索引 (name,age)，建立索引B+树的时候，key值此时是一个元组，元组内的顺序和定义是一样，例如 (Alan, 20) ，在比较两个元组的大小时，先比较元组最左边的元素，如果不相等就根据大小找对应的位置。如果相等，就比较第二个元素，以此类推。所以我们在用where语句查询的时候，要让条件的顺序满足最左匹配原则，或者，经优化器优化之后满足最左匹配原则。



复合索引对应的B+树，非叶子节点的key值是一个元组，元组在比较大小时是从左到右比。所以我们在写where条件时，如果想用到复合，就必须使用最左匹配原则，或经优化器优化后满足最左匹配原则。

覆盖索引实现方式？

1. 修改select后面要查的东西，保证其在索引对应的B+树叶子节点中。
2. 添加联合索引，比如通过id\_card查询姓名，可以添加(id\_card,name)联合索引，在使用id\_card单独作为where条件时，满足最左原则，可以利用该复合索引查询，而叶子节点正好有我们需要的name。

对于第二个实现方式需要考虑，该查询语句是不是出现很频繁，如果很少出现，建立索引树就会带来冗余，这个是需要权衡的。

单独为一个不频繁的查询请求创建一个索引，感觉优点浪费，该怎么做？

B+树这种索引模型，可以利用最左前缀原则，来定位记录。

最左前缀：1、联合索引最左N个字段

2、字符串最左M个字符 张%

在建立联合索引的时候，如何安排索引内的字段顺序？

1. 第一原则：如果可以通过调整顺序，而少维护一个索引，那么该顺序往往是需要有限 考虑的。举例：现在有两个需求：a)通过身份证查询姓名；b)通过身份证查询地址。其中第一个需求执行次数很多，第二个较少，但是也会执行。

创建一个（id\_card,name）,再创建一个(id\_card)单独的索引，就避免了b走全表扫描。

更进一步：(id\_card)单列索引可以不创建，根据最左前缀原则，（id\_card,name）这个索引顺序满足了a b两种需求，都不用走全表扫描。

2、考虑空间大小。例如：a)建立(age,name) 然后再创建一个（name）单值索引。b) 建立(name，age) 然后再创建一个（age）单值索引。b方式更好，因为age占用空间更小

聊一聊索引下推？

mysql> select \* from tuser where name like '张%' and age=10 and ismale=1;

表中有一个复合索引（name,age）

在mysql 5.6之前：找到满足name是张开头的记录就会去拿其主键回表，然后再比较age满不满足

在mysql5.6之后，引入了索引下推，在找到张开头的记录时，会看看age是否满足，如果不满足，就不会去回表了，减少了回表的次数。

索引下推：在查询复合索引对应的B+树时，以前是只要最左边的那个满足条件，就会往下一层看，不会比较元组的其他位置是否满足。然后读出所有最左边的元素满足的行，再回表查询改行记录，查看其他字段是否满足，

什么是mysql索引下推？

关键字：mysql5.6，索引优化…

立马get到，这个问题的是索引下推。

索引下推ICP Index Condition Pushdown，是mysql5.6版本的新特性。它能减少利用辅助键索引查询回表的次数，提高查询性能

索引下推本质是：将部分上次服务器负责的事，交给了存储引擎干。

在没有ICP时，使用辅助键，mysql的查询：

1. 存储引擎读取辅助键索引记录。
2. 拿到主键值，定位读取完整的行记录 （回表）
3. 存储引擎把行记录交给server层，server层检测是否满足where里面的条件

在有ICP时，使用辅助键，mysql的查询：

1. 存储引擎读取辅助键索引记录
2. 判断where条件部分能否利用索引中的其他列信息来做检查，如果不满足，则处理下一行数据。
3. 如果满足条件再去回表，不满足条件则不回表
4. 存储引擎把记录交给server，server检测 该记录是否满足

就是有了索引下推，复合索引才能有用。

联合索引？

一开始，联合索引提出是为了实现覆盖索引，为了减少回表次数，引入了ICP



聊一聊mysql的全局锁和表锁？

全局锁：对整个数据库实例加锁，使用场景是做数据库全库逻辑备份时。

在做全库逻辑备份时不关心最新的数据是什么，只关心能否读到当前时间节点的一致性数据。也就是能不能做到可重复读，不一定是最新的数据。而InnoDB的mvcc可以在不加锁的情况下，实现可重复读隔离级别，让DBA在做全库逻辑备份开始事务的时间节点到结束的事务整个过程的select读到的都是同一个视图。

1. 全库逻辑备份不要求最新数据
2. 全库逻辑备份要求数据一致性

既然有MVCC，为什么还要FTWRL命令呢？

因为MVCC只有InnoDB存储引擎支持，对于其他存储引擎，要做全库逻辑备份时，还是需要FRWRL命令

既然要全库只读，为什么不使用 set global readonly=true 的方式呢？

1. readonly逻辑不仅用在让数据库变成只读，也用在判断当前数据库是主库还是从库
2. 使用FTWRL，如果客户端发生异常断开连接，会释放当前事务的锁资源。使用readonly，数据库发生异常，全局变量值不改变，导致数据库长时间得不到更新。

聊一聊表级锁？

表级锁：表锁，元数据锁（Meta Data Lock MDL）

表锁注意事项：

1. 与FTWRL一样，可以主动unlock释放，也可以在客户端断开连接时释放
2. 表锁在lock一张表后，作用的线程不仅是其他线程，连当前线程也会被阻塞，不允许写或读。

例：事务A执行了 lock tables t1 read, t2 write；在unlock之前，事务A和其他并发事务都不能写t1，读写t2。

MDL锁：由系统自动上锁，上锁时机

1. 执行DML，会上MDL写锁
2. 执行DDL，会上MDL读锁

给表加一个字段，导致库挂了：

给表加字段，或者修改字段，或者加索引需要扫描全表的数据。

在修改表字段的时候，其它读事务还没结束，MDL读锁没释放，修改表字段事务阻塞，导致整个数据库的DML请求全部阻塞

如何安全地给小表加字段？

1、解决长事务，因为MDL锁是自动加，且在事务结束后才事务，如果是长事务，就会一直占用MDL读锁资源。所以，在准备执行DDL语句时，先去mysql的information\_schema库的innodb\_trx表中，可以查询到当前正在执行的事务。如果有长事务，就考虑等待这个长事务执行完，或者手动kill掉这个长事务。

2、如果对这个表的请求很频繁，但又不得不加字段时，可以对DDL语句设置一个等待时间，如果在指定的等待时间拿到MDL写锁最好，如果拿不到就主动退出竞争锁的状态，不要阻塞其他的DDL请求。

什么是-single-transaction参数？

这个参数是mysqldump是备份工具的，如果使用-single-transaction参数，导数据之前会启动一个事务，拿到一致性视图。由于MVCC的支持，这个过程不会影响更新。

全局锁和表锁注意事项？

1. 全局锁用在全库逻辑备份时；表锁用在存储引擎不支持行锁时
2. 如果应用程序有lock table语句：a)系统还在使用MyISAM存储引擎，需要升级；b)存储引擎升级了，但代码没有重构，写的还是lock unlock table 语句，需要重构代码，将其改为begin commit
3. MDL要在事务提交后才会释放锁资源。在做表结构变更时，一定要注意DDL等待写锁资源导致其它DML请求全部被锁死的情况发生。

为什么要有ICP？

说到ICP，必须说到复合索引，说起复合索引，必须说到覆盖索引。

覆盖索引：利用二级索引查询数据时，二级索引的叶子覆盖了我们要查询的字段，减少回表次数。实现方式有两种：1、减少我们要查询的字段，比如我从select \* 变成 select id。2、增加叶子节点的内容，使用复合索引。比如使用身份证号查询name，select name from T where id\_card = 10x，需要回表。如果改请求比较频繁，可以加一个覆盖索引 (id\_card, name)，那么改复合索引对应的B+树的叶子节点就是(id,\_card,name; id)。select name from T where id\_card = 10x就可以不用回表，减少了回表次数。

ICP提出也是为了减少回表次数：

(name,age)

select \* from T where name = ‘张三’ and age = 19;

|  |  |
| --- | --- |
| name | age |
| 张三 |  |
| 张三 |  |
| 张三 |  |

查出了三条name为张三的记录，此时select是\*，不满足覆盖，所以会回表，拿着主键去走一遍主键索引，得到行记录数据，然后把该行数据交给server层的执行器，执行器会判断age等不等于19，如果等于，则加入返回集。回表三次

mysql 5.6之后，引入了ICP

|  |  |
| --- | --- |
| name | age |
| 张三 | 18 |
| 张三 | 19 |
| 张三 | 20 |

存储引擎在(name,age)对应索引树，找到三条name为张三的数据，select \*，不满足覆盖，但是不会着急回表，而是索引下推，做一部分server层执行器的工作，看一看age符不符合条件，如果满足age=19，才会回表。这样回表次数是1，提升了查询性能。

什么是两阶段锁？

在InnoDB事务中，行锁是需要了就加上，不需要时并不会马上释放，而是等到事务提交时才释放，这就是两阶段锁协议。

知道两阶段协议了给我们开发的启示：如果事务中需要锁多个行，要把最可能造成锁冲突，最可能影响并发度的锁往后放。

以下业务，如何安排顺序，对并发影响最小？



这个过程需要执行三个步骤：

1. A账户余额扣除电影票钱
2. B账户余额增加这张电影票钱

这三个步骤涉及到两个update语句，一个insert语句，由于这三条语句是在同一个事务中执行的，索引执行顺序没有关系。那么如何安排执行顺序，对并发访问影响最小。就要看这三条语句哪一条有可能被并发执行。

答案是第二条，B账户余额增加票钱，如果C同时也在买票，需要等待A事务提交之后才能执行B账户余额增加票钱。

聊一聊死锁和死锁检测？

什么是死锁：两个线程相互等待对方的资源释放，自己拥有的锁资源才会释放，造成死锁。

解决死锁的两种方式：

1. 超时等待，innodb\_lock\_wait\_timeout参数设置超时时间
2. 死锁检测，发现死锁后，主动回滚死锁链条中的一个事务，让其它事务得以执行。innodb\_deadlock\_detect参数设置为on即可

所有事物都要更新同一行，进行死锁检测会怎么样？

由于所有事物都要更新同一行，大量的请求涌入时，假设1000个请求涌入，第一个占据到改行锁资源的事物还在处理，后面1000个事务全部阻塞，如果innodb\_deadlock\_detect参数开启了，这1000个事务都会进行死锁检测，而死锁检测的时间复杂度是O(n)，那么这个操作就是1000 \* 1000 = 1000000 一百万级别的操作。

如何解决热点行更新导致的性能问题？

1. 关闭死锁检测（头痛医头，脚痛医脚）
2. 控制并发度
3. 将一行改成逻辑上的多行

事务的启动方式的区别？

|  |  |
| --- | --- |
|  | 创建视图的时机 |
| begin / start transaction | 执行第一个普通读sql语句时创建视图 |
| start transaction with consistent snapshot | 执行改语句时创建视图 |

根据视图读取undo log判断标准？

1. creator\_trx\_id == trx\_id 可以访问
2.  可以访问
3.  不可以访问
4.  如果 trx\_id在m\_ids中，不可以访问；如果trx\_id不在m\_ids中，则可以访问。

说说MySQL中的视图？

mysql的视图出现在两个地方：

1、用查询语句定义的虚拟表，在调用的时候执行查询并生成结果。

2、InnoDB存储引擎使用MVCC使用一致性读视图时，支持RC和RR隔离级别的实现。

为什么表结构不支持“可重复读”？

因为表结构没有undo log

普通索引和唯一索引，应该怎么选择？

在业务代码层保证了该字段不会重复，所以在逻辑上，使用单列索引和唯一索引都可以。但是，在性能上还是有区别：

select \* from T where k= 5;

普通索引查询流程：

1. 在辅助键索引对应的B+树执行查询算法，查询k=5对应的主键，然后拿着主键再去主键索引查询出数据。
2. 在查到(5,id)之后，还要往后查一条记录，看看k是不是等于5

唯一索引查询流程：

在辅助键索引对应的B+树执行查询算法，查询k=5对应的主键，然后拿着主键再去主键索引查询出数据。

但是这两种带来的性能差距微乎其微：InnoDB的数据是按数据页为单位来读写的，也就是说，读取一条行记录时，并不是将这个记录本身从磁盘中读入内存，而是以页为单位，读取对应的磁盘块。在InnoDB中，每个数据页的大小默认为16kb。

因为引擎是按页读取的，所以对应k=5的那条记录，它所在的数据页一定在内存中，而它下一条数据很有可能位于同一个数据页，而处于内存中，不用再进行磁盘IO，速度很快。

而如果k=5正好是该页的最后一行数据，那就必须要读取下一页到磁盘中，但这样发生的概率非常小心，因为辅助键索引叶节点可以存储上千条key。

什么是change buffer？

在更新一个数据页时，如果该数据页正好在内存中，就直接更新；如果该数据页不在内存中，先把更新的操作缓存在change buffer中。这样就不需要重新从磁盘读数据页到内存中了。下次查询需要访问这个数据页时，将数据页从磁盘中读入内存，让后执行change buffer中关于该页的操作，保证数据逻辑正确性。将change buffer中的更新操作应用到对应数据页的这个过程叫做merge。

触发merge的时机：

1. 访问该数据页时
2. 系统后台线程定期merge
3. 数据库正常shutdown时

为什么要有change buffer？

更新是需要占用写锁资源的，而写锁影响比较大的锁，因为写锁之后，读和写都不能进行，所以，在一个事务中，更新操作通常放在最后，事务提交前写。因为已提交才会释放锁资源。还有更新操作时间越短越好，所以引入了change buffer，将原本需要在更新期间做的事，移到查询期间，查询期间有了MVCC，普通读是不用上锁的，所以这时后执行时间不会影响其他读写操作。所以这就是change buffer的好处。

什么条件下可以使用 change buffer 呢？

普通索引可用，唯一索引不可以用。原因：唯一索引所有的更新操作都需要判断是否违反了唯一性原则，比如插入(4,400)这条记录，就要判断这个加了唯一索引字段的表数据中是否已经包含k=4的这条记录。这时，必须要将数据页读入内存当中。

如果要在这张表中插入一个新记录 (4,400) 的话，InnoDB 的处理流程是怎样的？

第一种情况，k=4所在数据页在内存中：

1. 唯一索引：找到3和5之间的位置，判断有没有冲突，插入该行数据
2. 普通索引：找到3和5之间的位置，插入该行数据

唯一索引只是比普通索引多了一次判断，性能差别不大。

第二种情况，k=4所在数据页不在内存中：

1. 唯一索引：将k=4所在数据页读入内存，找到3和5之间的位置，判断有没有冲突，插入该行数据
2. 普通索引：将更新操作写入change buffer 结束

对于这种情况，唯一索引需要进行磁盘IO，这对性能影响是较大的。change buffer由于减少了更新语句的磁盘IO的访问，加快了更新语句执行性能。

普通索引的所有场景，使用 change buffer 都可以起到加速作用吗？

不一定。

1、对于对于写多读少的业务，在一页上做更新的次数越多，收益越大，更新了很多次都不需要做磁盘IO，然后查询一次将数据页读入内存，merge。

在做merge之前，如果一个change buffer内容越多，收益越大。

这种就是账单类，日志类表文件，在写完后马上被访问到底概率很小。

2、对于写少读多的业务，就不适合change buffer了，可能才写完一次，下次马上就读入内存，然后merge。change buffer降低性能就没那么明显。反而增加了维护changer buffer以及做merge的代价。

普通索引和唯一索引怎么选择？

结论：能选择普通索引就尽量选择普通索引，对于需要具有唯一性的字段，可以在业务层实现保证逻辑上不重复，然后使用普通索引。

原因：首先，这两个索引在查询上性能是差不多的，主要的差距是更新语句的性能。

如果所有的更新后面，马上伴随着对这个记录的查询，那么应该关闭change buffer。而在其他请款下，change buffer能显著提升性能。

在实际使用中，普通索引+change buffer的方式，对数据量比较大的表更新优化是很明显的。特别是，当硬盘还是使用机械硬盘时，change buffer这个机制的收效是非常明显的。因为机械硬盘的读写速度更慢，相较于SSD。所以对于一些历史库，使用机械硬盘，考虑对于普通索引字段，设置change buffer，或者加大change buffer的大小。

change buffer 和 redo log？

insert into t(id,k) values(id1,k1),(id2,k2);

k1所在的数据页page1在内存中，k2所在的数据页page2不在内存中

1. Page1在内存中，直接更新Page1，插入一条记录
2. Page2不在内存中，将更新“Page2，插入一条记录”写入change buffer
3. 将上面两个动作写入redo log

这个过程设计两次内存写，和一次磁盘写（且是顺序磁盘写）。所以执行速度还是很快的。

change buffer 节省了磁盘读的时间。

redo log 节省了磁盘随机写的时间。



之后执行select \* from t where k in (k1, k2)



mysql 索引字段的基数怎么计算呢？

采样统计法：

由于全表扫描的代价太高了，InnoDB会采样统计学的方法。默认选择N个数据页作为样本，统计样本页上，某个索引字段的不同值个数，然后除以样本页面数，得到平均值，再用这个平均值乘以总的页面数，就是这个索引字段的基数。而表是在不断更新的，索引字段的值在变化，基数的值也应该变化，所以，当变更数据超过1/M时，会自动触发重新做一次索引统计。

优化器判断语句扫描行数是怎么做的？

首先，在执行器真正执行sql语句之前，优化器是不可能精确知道满足这条sql语句的记录有多少行，只能靠估算。根据什么信息来进行估算呢，根据统计信息来进行估算。这个统计信息就是索引的区分度。

一个索引字段上，不同的值越多，这个索引的区分度就越好。一个索引字段上，不同值的个数，称之为基数。也就是说，基数越大，索引的区分度越好。

可以使用show index命令查看索引的基数。cardinality就是基数



怎么给字符串字段加索引？

例如email是字符串：

create index string\_index2 on t(email(3))

表示给email这个字符串字段加上索引，前缀3，只用到长度为3的字符串，后面的字符串不适用，索引的关键字长度只有3位，这里字符串索引长度可以指定。

长度太长：占用内存空间

长度太短：可能增加扫描次数

字符串字段可以使用前缀索引，而且设置好长度，可以让占用空间降低，也能不增加扫描次数。

如何确定前缀索引长度？

建立索引的时候最终要的数据就是基数，也就是区分度，却分度越高越好，所以我们可以用以下命令查看不同长度前缀，区分度是多少：

select count(distict left(email,len));

前缀索引对覆盖索引的影响?

1、select id,email from SUser where email=’zhangssxyz@xxx.com’;

这条sql语句和

2、select id,name,email from SUser where email='zhangssxyz@xxx.com';

sql2语句没有用到覆盖索引，索引前缀索引没影响。

sql1语句用到了覆盖索引，如果没有前缀索引，直接email索引，查询到叶子叶子节点后可以直接返回id和email。

但是，如果使用了前缀索引email(len)，即使这个len很大，比如email(18)，已经包含一个邮箱最长长度了，innodb还是会拿着主键去主键索引树回表一次，因为innodb引擎不确定你这个截断和真正的业务数据的关系。

这时候，前缀索引就对覆盖索引产生影响了。

对于身份证这种类型的字段，用什么方式比较好呢？

身份证一共18位，前6位是地址码，所以一个县的前6为身份证一般是相同的。如果身份证号，前缀索引长度是6，区分度就很低。后面12位是出生日期（6位）和6位后缀。如果使用前缀索引，要照顾区分度的话，前缀索引长度可能要设置到12位，这又会带来占用空间过大的问题。

那有没有，其他解决方案呢：有

1. 倒序存储，由于身份证最后6位，可以将身份证号倒序存储，前缀索引只用设置6位就有很好的区分度了。记得查询的时候，要把身份证号倒叙查询。

select field\_list from t where id\_card = reverse('input\_id\_card\_string');

1. hash字段，可以在表上增加一个整数字段，来保存身份证的校验码，同时在这个字段上加索引。

alter table t add id\_card\_crc int unsigned, add index(id\_card\_crc);

每次插入新记录时，用crc32这个函数得到身份证号的校验码。

由于校验码可能存在冲突，即两个身份证号的校验码是一样的，所以在查询的时候除了判断校验码，还需要判断身份证号

mysql> select field\_list from t where id\_card\_crc=crc32('input\_id\_card\_string') and id\_card='input\_id\_card\_string'

这样，索引长度变成了整数 4个字节。

倒序存储和哈希表的异同点：

相同点：都不支持范围查询。倒叙无法支持[id\_cardX, id\_cardY]的范围查询，哈希只支持等值查询。

不同点：

1. 占用空间，倒序存储不会添加额外字段，但倒序存储索引长度会长一些
2. cpu消耗方面：reverse函数比hash函数消耗小一些
3. 查询效率，哈希方式更稳定，因为前缀索引更容易找到重复的值，然增加查询次数。而crc32哈希函数遇到冲突概率小，可以认为每次只扫描一行

字符串字段上创建索引的方式比较？

|  |  |  |
| --- | --- | --- |
| 方式 | 优点 | 缺点 |
| 普通索引 | 扫描次数少  支持范围查询  支持覆盖索引 | 占用空间大 |
| 前缀索引 | 占用空间小  支持范围查询 | 扫描次数可能增加 |
| 倒序存储 | 占用空间小 | 不支持范围查询  扫描次数可能增加 |
| 哈希 | 扫描次数少 | 不支持范围查询  增加额外字段，增加空间 |

为什么我的MySQL会“抖”一下？

flush：将内存数据写入磁盘

脏页：当内存中的数据页和磁盘的数据页内容不一致时，内存中的数据页称之为脏页数据，经过flush之后，就变成干净页。

导致mysql突然变慢可能就是在那个时刻，在刷脏页。触发刷脏页的时机：

1、redo log满了，需要擦除一部分，擦除的过程就需要将对应的checkpoint之内的脏页刷到磁盘上。

2、当系统内存不足，此时又需要读新的内存页到内存中时，要把一部分内存页淘汰掉，如果淘汰的时脏页，需要把脏页flush到磁盘，释放一部分内存。

3、mysql认为系统空闲的时候。

4、mysql正常关闭的时候。mysql需要把内存中的脏页数据都flush到磁盘上，这样下次mysql启动的时候，就可以直接从磁盘读数据，然后返回。

以上四种场景，最需要关注的是前面两种对性能的影响。

第一种：redo log写满了，flush脏页。此时系统是不接受更新，所有更新都会阻塞，从监控上看就是更新数降低到0。

第二种：内存不够了，淘汰一部分内存页。这种情况是常态。InnoDB用缓冲池（buffer pool）来管理内存页，缓冲池中的内存页有三种状态：

1. 还没有使用的
2. 使用了并且是干净页
3. 脏页

当缓冲池满了，又要申请一个新的页到内存中时，需要把最近最久未被使用的页淘汰掉，如果这个页是脏页，需要flush到磁盘。

如果等到缓冲池满了再淘汰脏页，内存中可能会有大量脏页，会导致flush磁盘的时候性能变低。

所以，要控制脏页的比例，达到一定比例就开始淘汰页。

为什么不直接把脏页淘汰掉，下一次重磁盘读内存的时候，用redo log应用，变成最新的状态？

这是从性能考虑的。如果刷脏页一定会写盘，就保证了每个数据页两种状态：

1. 数据页在内存中，一定是最新的，正确的结果。有查询请求访问，可以直接返回。
2. 数据页不在内存中，那么磁盘页中的数据页一定是最新的，读入内存后返回，这样效率最高。

InnoDB的脏页控制策略是什么呢？

首先，需要告诉InnoDB所在主机的IO能力，这样InnoDB才能知道全力刷脏页的时候，可以刷多块。

这时会用到innodb\_io\_capacity这个参数，值设置成磁盘的IOPS。磁盘的IOPS可以通过fio命令查询。

如果你来设计策略控制刷脏页的速度，会参考哪些因素呢？

按照全力刷脏页的百分比来刷。

如果刷太慢，内存页脏页太多，redo log也容易写满。

如果刷太快，磁盘被占用，其他请求不能使用磁盘。

所以，innodb刷脏页控制策略在正确知道磁盘的IO能力之后，考虑的是两方面的因素：

1. 内存中脏页比例
2. redo log 写盘速度

innodb会根据这两个因素单独算出两个数字。

参数innodb\_max\_dirty\_pages\_pct是脏页比例上限，默认值是75%。

innodb会根据当前的脏页比例M，算出一个数字：

F1(M)

{

if M>=innodb\_max\_dirty\_pages\_pct then

return 100;

return 100\*M/innodb\_max\_dirty\_pages\_pct;

}

假设innodb\_max\_dirty\_pages\_pct是默认值75%，M是50%

则F1(M) = 66

F1(76%) = 100

innodb每次写入的日志都有一个序号，N = |当前写入序号 – chekpoint对应的页的序号|，F2(N) , N越大，F2(N)越大

R = max(F1(M), F2(N))

刷脏页速度 = innodb\_io\_capacity \* R%

为什么表数据删掉一半，表文件大小不变？

以InnoDB为存储引擎讨论，一个InnoDB表包含两部分，即：表结构定义和表数据

表结构定义：



表数据:



mysql 8.0版本以前，表结构是存储在.frm为后缀的文件中。而mysql 8.0版本，以及允许将表结构定义放在系统数据表中了。而表结构定义占用的空间非常小，所以主要讨论表数据。

为什么简单地删除表数据，达不到回收空间的效果，正确的回收空间的方法是什么？

表数据既可以存在共享表空间里，也可以是单独的文件。这个行为是由参数innodb\_file\_per\_table控制的：

1. 这个参数设置为OFF表示的是：表数据放在系统共享表空间，也就是跟数据字典放在一起。
2. 这个参数设置为ON表示的是：每个InnoDB表数据存储在一个以.ibd为后缀的文件中。

mysql 5.6之后，这个参数的默认值是ON。

建议把这个参数设置成默认值ON，因为，一个表单独存储为一个文件更容易管理。在不需要这个表时，drop table命令：如果表是单独一个文件，则直接删除文件；如果表是在系统共享表空间中，即使表删除了，空间也不会回收。

mysql innodb存储引擎删除表数据的流程是什么？

比如一个表数据，在磁盘上是以B+树索引模型存储的：



如果要删除id=500 即R4这条记录，并不会直接把这个占用的空间去掉，而是将这个位置标记未删除，逻辑删除。之后如果再插入一个300到600之间的记录时，可以直接写到R4这个位置，复用这个位置。也就是删除后，只是标记，磁盘文件的大小不会缩小。

InnoDB如果删除一页的数据会怎么样？

整个数据页都可以被复用了。

但是，数据页的复用，跟行记录的复用不一样：

记录的复用，只限于符合范围条件的数据。

整页的复用，可以复用任何位置。就是一个新页，如果插入一条数据需要一个新页，就直接拿这个页就行了。

如果相邻的两页利用率很小，会将两页合并成一页，让后另一页标记成可复用的。

所以，如果用delete将该表占用的所有页上的数据都删除掉，那么这些页都会标记成可复用的页，而占用磁盘的大小不会改变。

通过delete命令是不能回收表空间的。这些delete的空间是可复用的，而没还被复用的空间看起来就像是空洞。

为什么插入数据也会造成空洞？

如果数据是按索引递增的顺序插入的，那么索引是紧凑的，如果索引是随机插入，如果数据是随机插入，可能造成索引的数据页分裂。

而页分裂后的数据，某一页的末尾就可能留下空洞。

更新索引字段的值也可能造成空洞，因为更新索引字段的值实际上时删除旧值然后插入新的索引值。

也就是说：经过大量增删改的表，都是可能存在空洞的。如果能够把这些空洞去掉，就可以达到收缩表空间的目的。

什么是重建表？

重建表的提出就是为了解决表经过大量增删改之后出现大量空洞，导致表空间占用很大的问题。

alter table A engine=InnoDB命令可以用来重建表。使用该命令，mysql会在后台执行：

1. 创建一个临时表B与A的结构相同
2. 将A中的数据按主键索引ID递增的顺序读出，然后插入B
3. 表B替换表A

由于表B是新建的表，所以没有表A那么多的空洞，而且表B是按主键索引递增顺序插入的，所以biaoB的主键索引更加进程，数据页的利用率更高。

这个过程简化成12个字：转存数据，交换表名，删除旧表。

对于Online DDL，alter table语句在开始时会加DML写锁，但是转存数据过程中退化成DML读锁。

为什么Online DDL alter table MDL写锁会退化成读锁？

这是为了能够在耗时较长的转存数据过程中，不阻塞对数据库的DML请求，即增上改查操作。使得增删改查操作能够正常进行。

非Online DDL 和Online DDL重建表的过程有什么不同？

前者是创建的是临时表，是在server层创建的。

后者是创建临时文件，是InnoDB内部创建的。

前者重建过程中有两张表

后者重建过程中只有一张表

为什么 重建表过程不直接解锁呢？

为了保护自己，禁止其它线程同时对改表做DDL操作。

说一说 Online 和 Inplace的区别？

这两个都是跟DDL有关的概念。

Online：是指在重建表，转存数据的过程中，仍然可以增删改查

inplac：指Online DDL重建表时，根据原表重建出来的数据是放在temp\_file中，这个是有innodb创建的临时文件，mysql server层不会创建临时表，整个过程都砸innodb内部完成，所以，对于mysql来说，就好像原地完成的，并没有临时表参与，这就是inpulace。

DDL过程如果是Online的，就一定是inplace的

反之不正确。

如果你有一个 1TB 的表，现在磁盘间是 1.2TB，能不能做一个 inplace 的 DDL 呢？

不能，因为inplace也需要占用空间。

optimize table、analyze table 和alter table区别？

|  |  |
| --- | --- |
| 操作 | 描述 |
| alter table t engine = InnoDB | recreate；默认是inplace的 |
| analyze table t | 对表索引信息重新统计，没有修改数据，统计索引的基数。这个过程要加MDL读锁 |
| optimize table t | 等于 recreate + analyze |

在收缩一个表的时候，只用delete语句是做不到的，需要用alter table重建表，才能达到让表文件变小的目的。

MySQL count(\*)语句如何实现的，为什么要这么实现？

在不同的存储引擎，count(\*)有不同的实现方式。

1、MyISAM：把一个表的总行数存在了磁盘上，因此执行count(\*)的时候会直接返回这个数，效率很高。

2、InnoDB：执行count(\*)的时候会一行一行的读取，累计计数。

这里假设的是没有where条件的情况下如何返回的。

为什么InnoDB不像MyISAM那样，把表行数存起来？

由于InnoDB MVCC的机制存在的原因。由于不同的事物对行的可见性是不一样的，如果用一个变量将表行数存起来，就无法细化到某行是否对该事务可见，所以count(\*)的执行过程是一行一行的读出，依次判断是否对该事务可见，然后累计结果。

InnoDB 由于MVCC的存在，执行count(\*)的时候会一行一行地扫描，判断是否对该事务可见，那么InnoDB在扫描过程中做了哪些优化?

我们都知道，InnoDB主键索引的叶子节点存储的数据，而二级索引的叶子节点保存的是主键值，所以普通索引对应的B+树要小很多，对于count(\*)遍历操作，遍历哪颗索引树在逻辑上，结果是一样的。因此，mysql优化器会找到最小的那颗B+树来遍历。在保证逻辑结果正确的前提下，尽量减少扫描的数据量，是数据库系统设计的通用法则之一。

获取表总行数的几种方式？

|  |  |
| --- | --- |
| 方式 | 优缺点 |
| MyISAM count(\*) | 很快，但不支持事务 |
| show table status | 很快，但不准确 |
| InnoDB count(\*) | 支持事务，准确，但是很慢 |

如果有一个页面经常要显示交易系统的操作记录总数，到底应该怎么办呢？

自己计数，方式有很多:

1. 用缓存系统保存计数，比如可以用Redis服务来保存这个表的总行数。问题1，可能会丢失更新，比如Redis在加1后异常重启了，读取保存到文件上的值，就会丢失到此次更新。

问题2，Redis即使运行正常，存储的这个值在逻辑上可能是不精确的。

此时会发现最近100条记录新插入了一条，但是redis count没变

|  |  |  |
| --- | --- | --- |
| 时刻 | 事务A | 事务B |
| t0 | 插入一条记录 |  |
| t1 |  | 读取Redis count |
| t2 |  | 读取最近100条记录 |
| t3 | Redis count 加1 |  |

此时会发现，最近100条没有新插入的记录，但是Redis count 加1了

|  |  |  |
| --- | --- | --- |
| 时刻 | 事务A | 事务B |
| t0 | Redis count 加1 |  |
| t1 |  | 读取Redis count |
| t2 |  | 读取最近100条记录 |
| t3 | 插入一条记录 |  |

1. 在数据库保存计数，将这个计数保存到数据库里单独的一张计数表C中。

首先解决了崩溃丢失问题，因为redo log的存在。然后，对于计数不精确的问题如何解决：可以利用事务的特性

|  |  |  |
| --- | --- | --- |
| 时刻 | 事务A | 事务B |
| t0 | begin  表C count 加1 | begin |
| t1 |  | 读取表C count值 |
| t2 |  | 读取最近100条记录  commit |
| t3 | 插入一条记录  commit |  |

由于MVCC的存在，对事物B来说，读取表C count值是普通读，所以是快照读，所以读取的count没有加1，而最近100条记录也是在快照读，也只会读取插入之前。保证了这两个值在逻辑上的正确性。

|  |  |  |
| --- | --- | --- |
| 时刻 | 事务A | 事务B |
| t0 | begin  表C count 加1 | begin  读取表C count值 |
| t1 | 插入一条记录 |  |
| t2 | commit |  |
| t3 |  | 读取最近100条记录  commit |

MVCC的默认隔离级别是可重复读，所以在事务开始的时候会创建一致性视图，后面连续的普通读会复用这个视图。

所以，在这种情况下，也保证了这两个数据逻辑的正确性。

count( \* )这么慢，该怎么办?

count(\*)、count(字段)、count(1)、count(主键id)不同用法的区别？

本质是要了解count(形参)函数的实现方式，它会对返回的结果集一行一行地进行判断，如果这一行的形参不为NULL，累计值就加1，否则就不加。

|  |  |
| --- | --- |
| ount( \* )、count(主键 id)、count(1)和count(?) | 返回结果集的总行数 |
| count(字段) | 返回结果集中，字段不为NULL的行的总行数 |

在两阶段提交的不同瞬间，MySQL 如果发生异常重启，是怎么保证数据完整性的？



1. 在时刻A发生崩溃，也就是redo log写入处于prepare状态，但是bin log还没有写入就发生了崩溃。

由于redo log还处于prepare，还没提交，所以mysql重启后，事务会回滚，回到更新之前的状态，而bin log本来就没写入，所以bin log和redo log是一致的。在后续备份库时，更新也不会传到备考里面

1. 在时刻B发生崩溃，bin log 已经写完，而redo log没提交就发生了崩溃。mysql重启后，会根据一下规则来判断是否回滚：
   1. 如果redo log已经有了commit标识，则直接提交。因为有了commit标识，说明事务是完整的。
   2. 如果redo log没有commit标识，则判断对应事务的bin log是否存在并且完整。i如果是，则直接提交事务。ii如果不是，则回滚事务，bin log这条记录页不会保留。

MySQL 怎么知道 binlog 是完整的?

一个完整的bin log是有完整的格式的：

* statement格式的binlog，最后会有COMMIT
* row格式的binlog，最好会有一个XID event

所以mysql是有方法知道binlog的完整性的。

在redo log中，如何找到对应事务的binlog？

它们有一个共同的字段，XID。崩溃恢复的时候，会顺序扫描redo log：

* 如果碰到既有prepare，又有commit的事务，直接提交。
* 如果碰到的只有prepare，而没有commit的事务，就拿着XID取binlog里面查找对应的事务，判断其是否完整。

处于prepare状态的redo log加上完整的binlog就可以，重启就可以恢复，mysql为什么要这么设计？

这与数据与备份的一致性有关。因为binlog已经写入了，并且是完整的，以后备份库的时候，备库会记录此次更新，而为了让主库与备库保持一致，主库上此次事务需要提交。

如果这样的话，为什么还要两阶段提交呢？干脆先 redo log 写完，再写 binlog。崩溃恢复的时候，必须得两个日志都完整才可以。是不是一样的逻辑？

对于InnoDB存储引擎，redo log一旦commit，就不能回滚了，因为如果对已提交的事务回滚，有可能覆盖其它事务做的更新。所以先写完redo log，提交之后bin log写入出现异常，即使你知道bin log是不完整的，也无法回滚。还是会造成主库与备库不一致的情况。

为什么不能只用bin log？

不能，这是由于binlog本身存储的内容造成的。binglog存入的更新是逻辑上的更新，不是物理上的更新。举个例子，update T set c = c+1 where id =1；逻辑上的更新会记录将id=1字段c的值加1。物理上的更新会记录字段c值变成了扫描。

如果只使用binlog，对于一些以提交的事务，其更新的数据页还在内存中，还没写入磁盘，此时内存中的数据页是最新数据，如果这时候发生崩溃重启，由于这些事务是已经提交的，不会再应用依次binlog，但是，之前数据页没有写入磁盘，就导致这些事务的更新丢失了。

为什么 binlog不支持崩溃恢复，crash-safe？

binlog记录的是逻辑上的更新，如果发生异常重启，内存中还没写入磁盘的数据页已经丢失了，redo log记录这些值最新的值是什么，然后将数据页读到内存，应用redo log，之后再flush回磁盘。而bin log 记录的是这些字段是怎么变化的，必须有最近时刻全库备份才可以恢复到崩溃前的状态。

redo log是怎么崩溃恢复磁盘数据页的？

redo log并没有记录数据页完整的数据，所以，redo log并没有能力自己 取更新磁盘数据页。

1. 如果是正常运行的实例，内存中数据页被修改后，变成脏页，mysql会将脏页flush到磁盘，这个过程和redo log无关。
2. 在崩溃恢复的场景中，InnoDB如果判断到一个数据页可能在崩溃恢复的时候丢失了更新，此时会将这个数据页读入内存，然后让redo log更新内存中的数据页，内存页又变成脏页，回到第一种情况。

|  |  |  |
| --- | --- | --- |
| 时间 | session1（A like B） | session2 （B like A） |
| t0 | begin;  select \* from ‘like’ where user\_id=B and like\_id=A;（返回空） |  |
| t1 |  | begin;  select \* from ‘like’ where user\_id=B and like\_id=A;（返回空） |
| t2 |  | insert into ‘like’ (user\_id, like\_id) values(B,A); |
| t3 | insert into ‘like’ (user\_id, like\_id) values(A,B); |  |
| t4 | commit; |  |
| t5 |  | commit; |

A < =B时

A like B : (A,B,1) A(user\_id)关注B(like\_id)

A > B时

A like B: (B,A,2) A(like\_id) 关注 B(user\_id)

A = 1 , B = 2

|  |  |  |
| --- | --- | --- |
| 时间 | session1（1 like 2） | session2 （2 like 1） |
| t0 | begin;  insert into `like`(user\_id, liker\_id, relation\_ship) values(A, B, 1) on duplicate key update relation\_ship=relation\_ship | 1; |  |
| t1 |  | begin;  insert into `like`(user\_id, liker\_id, relation\_ship) values(A, B, 2) on duplicate key update relation\_ship=relation\_ship | 1; |
| t2 |  | insert into ‘like’ (user\_id, like\_id) values(B,A); |
| t3 | insert into ‘like’ (user\_id, like\_id) values(A,B); |  |
| t4 | commit; |  |
| t5 |  | commit; |

为什么？为什么条件是 where t\_modified=’2018-7-1’的时候可以用上索引，而改成 where month(t\_modified)=7 的时候就不行了？

因为索引树查询的本质是利用排序，比较索引关键字大小，可以知道往节点的哪个分支去找数据，以t\_modified=’2018-7-1’为例，由于t\_modified字段上有索引，而且条件是找到’2018-7-1’，那么就会从根节点开始比较，比如根节点是’2017-6-1’，就会走其孩子节点的某个分支，很快利用索引树就可以找到数据所在的数据页。

而对于month(t\_modified)=7这个条件，使用了函数计算，无法利用索引关键字比较的特性，需要把t\_modified每个字段值都读出来比较一下，这就是全表扫描了。

为什么这些SQL语句逻辑相同，性能却差异巨大？

1. 条件字段函数操作

交易系统有id，tradeid, operator, t\_modified等字段。

记录了从2016年初到2018年底的所有数据

统计发生在所有年份中，7月份的交易记录总数

select count(\*) from tradelog where month(t\_modified) = 7;

解决办法：不适用函数，修改为比较范围

对于selec \* from tradelog where id + 1 = 1000; 也不会走主键索引搜索树功能，而是全索引扫描，因为mysql不知道id+1=1000等价于id=999；需要程序员手动修改sql语句。

1. 隐式类型转换

select \* from tradelog where tradeid=110717;

tradeid上有索引，但这条语句还是走的全表扫描。原因是tradeid类型是varchar，tradeid=110717进行了隐式类型转换

1. 隐式字符编码

为什么进行了数据类型转换，就无法使用索引呢?

因为相当于给字段加了类型转换函数，所以无法使用索引了。

select \* from tradelog where tradeid=110717;等价于select \* from tradelog where CAST(tradid AS signed int) = 110717;将字符转为数字。但是select \* from tradelog where id="83126";等价于select \* from tradelog where id=CAST("83126" AS signed int);没有给字段加类型转换函数，所以可以用到索引

select operator from tradelog where tradeid = CONVERT($R4.tradeid.value USING utf8mb4);

为什么只查一行的语句，也执行这么慢？

1. 有一个线程持有MDL写锁，导致其他线程对表执行增删改查，即使只查一行语句，也要给表加上MDL读锁，导致这些语句被阻塞。所以看起来，只是查了一行数据，但是执行的很慢。这个解决办法是找到占用MDL写锁的线程，将其kill掉

如何避开MVCC，普通读加上读锁？

lock in share mode

select \* from t where id=1 lock in share mode;

这是就会加一个in share mode的锁，也就是读锁。

通过sys.innodb\_lock\_waits表查看谁占用了写锁？

blocking\_pid: 5 // 表示是哪个线程导致了阻塞，这里可以看到是5号线程，导致的阻塞。可以通过 kill query 5 或者kill 5命令关闭这个线程。但是kill query 5并不能解决问题，因为该命令表示停止5号线程正在执行的语句，但不会释放锁资源。只有kill 5才管用，才能去掉写锁。

什么是两阶段锁协议？

就是加锁的时候，在需要的时机马上加

但释放锁不是在不需要这个锁之后立马释放，而是等到事务提交了才会释放锁资源。

幻读是什么？

在一个事务中，前后读出的表的总行数不一致。

一个事务在前后两次查询同一个范围的时候，后一次查询看到了前一次查询没有看到的行。

update t set d = d + 1 where id = 7;

1. id上有索引，扫描过程一下就到id=7了，id=7这一行不存在，加不了行锁。7所在的间隙是(5,10),所以这个间隙也被扫描到了。根据原则1，加锁的基本单位是next-key锁，而next-key是左开右闭区间。所以加的是(5,10]的next-key锁。
2. id有索引，所以是索引上的查询，且是等值查询，根据优化2，向右遍历时，且最后一个值不满足等值条件时，next-key退化为间隙锁。id=10不满足查询条件，所以退化成间隙锁，即(5,10)。



覆盖索引：



根据原则2，只有访问到的对象才加锁。

注意这里，覆盖索引只对lock in shar mode有效，如果是for update，执行for update的时候，mysql会认为你接下来要变更数据，会顺便到主键索引上满足条件的行加上锁。

什么是短连接模式？

短连接模式就是连接到数据库后，执行很少的SQL语句就断开连接。短连接模式避免了长事务的发生，长事务对mysql的影响是比较大的，因为长事务会导致连接对象长时间不是发，而mysql的内存资源是管理在连接对象中的，会导致垃圾内存增多。长事务也会导致undlog 日志增多，内存消耗变大。

什么是短连接风暴？

短连接风暴就是在业务高发期，mysql的并发压力增大，大量连接涌入mysql，导致连接数超过max\_connections参数，此时新进来的连接，mysql就会报Too many connections错误。

如何解决短连接风暴？

1. 处理掉占着连接，但不工作的线程。max\_connections参数的计算不是看哪个连接在running，而是有连接占用，就计算一个，所以我们可以通过kill connection命令主动踢掉。
2. 减少连接过程的损耗

如果查询或更新用到了索引，但是查询或更新的数据特别多，占全表的 80% 甚至更多，这时候是会用表锁，还是行锁呢？

声明式事务在哪些情况下会失效？

Spring 的事务隔离级别？

AOP 原理

AOP 用到了 Java 中的哪些技术 ？

IOC AOP 说说自己的看法

Spring 中 Bean 的生命周期（Bean 的加载步骤）

Spring 的常用注解

Spring 注入的几种方式

Spring 装配 Bean 的几种方式

如何使用 IOC

通常如何获取 Bean

Bean 的扫描路径

AutoWired 是按照什么规则来获取 Bean 的 （Resource 呢？）

Spring 如何解决循环依赖

为什么要用三级缓存，二级缓存不行？

BeanFactory 和 FactoryBean 的区别？

SpringBoot 自动配置怎么实现的？

Spring 事务管理怎么实现的？

Spring 哪里用到了反射？

说说循环依赖?

那 Spring 怎么解决循环依赖的呢？

为什么要三级缓存？⼆级不⾏吗？

MVC Spring MVC 的核心组件？

Spring MVC 的工作流程？

SpringMVC Restful 风格的接口的流程是什么样的呢？

Spring Boot介绍一下 SpringBoot，有哪些优点？

SpringBoot 自动配置原理了解吗？

如何自定义一个 SpringBoot Srarter?

Springboot 启动原理？

Spring Cloud对 SpringCloud 了解多少？

volatile的作用和原理？

什么是原子性？

volatile能不能保证原子性？

你说你的项目用到了ThreadLocal，好好讲讲ThreadLocal是怎么回事吧？

ThreadLocal是什么?为什么要使用ThreadLocal ？

一个ThreadLocal的使用案例 ？

ThreadLocal的原理 ？

为什么不直接用线程id作为ThreadLocalMap的key ？

为什么会导致内存泄漏呢？是因为弱引用吗？

Key为什么要设计成弱引用呢？强引用不行？

InheritableThreadLocal保证父子线程间的共享数据 ？

ThreadLocal的应用场景和使用注意点？

你能跟我说说它隔离有什么用，会用在什么场景么？

除了源码里面使用到ThreadLocal的场景，你自己有使用他的场景么？一般你会怎么用呢？

嗯嗯，你回答得很好，那你能跟我说说他底层实现的原理么？

ThreadLocalMap底层结构是怎么样子的呢？

为什么需要数组呢？没有了链表怎么解决Hash冲突呢？

能跟我说一下对象存放在哪里么？

那么是不是说ThreadLocal的实例以及其值存放在栈上呢？

如果我想共享线程的ThreadLocal数据怎么办？

怎么传递的呀？

spring TransactionSynchronizationManager类？

mysql查询优化？