# JAVA

## 线程

### 1.多线程有几种实现方案，分别是哪几种？

答：两种。

(1)继承Thread类

(2)实现Runnable接口

扩展一种：实现Callable接口，得和线程池结合

### 2.同步有几种方式，分别是什么？

答：两种

(1)同步代码块

(2)同步方法

### 3.启动一个线程是run()还是start()?他们的区别是什么？

答：start();

区别：

run()：封装了被线程执行的代码，直接调用仅仅是普通方法的调用；

start()：启动线程，并由JVM自动调用run()方法

### 4.sleep()和wait()的区别是什么？

答：sleep()：必须指定时间；不释放锁

wait()：可以指定时间，也可以不指定时间；会释放锁

### 5.为什么wait(),notify(),notifyAll()等方法都定义在Object类中？

答：因为这些方法的调用是依赖于锁对象的，而同步代码块的锁对象是任意锁。而Object是所有对象的父类，所以定义在这里面。

### 6. lock和synchronized的区别

(1) lock是一个接口，而synchronized是java的一个关键字。

(2) synchronized在发生异常时会自动释放占有的锁，因此不会出现死锁；而lock发生异常时，不会主动释放占有的锁，必须手动来释放锁(建议在finally中释放锁)，不然可能引起死锁的发生。

(3)使用synchronized时无法判断锁的状态，使用lock可以通过tryLock()来判断有没有获取锁。

(4)使用synchronized时，如果有一个线程获得锁，则另一个线程会一直等待；而使用lock时，如果有一个线程获得锁，则另一个线程会不断尝试获得锁，不用一直等待。

(5)synchronized和lock在竞争资源不激烈的情况下，性能相差不大。synchronized引入了偏向锁、轻量级锁、重量级锁以及锁升级机制来实现锁的优化；lock则是使用了自旋锁实现锁的优化。

### 7.线程的生命周期图

答：新建—就绪—运行—死亡

新建—就绪—运行—阻塞—就绪—运行—死亡



### 8.线程的状态转换图



### 9.线程池的理解

线程池是一种池化技术，用于提高资源的复用性，常见的还有内存池、连接池和对象池。线程池复用的是线程资源，它减少了线程的频繁创建和销毁带来的性能开销，其本身也会有参数来控制线程创建的数量，避免了无休止创建线程带来的资源利用率过高的问题。

### 10.Thread和Runnable的区别

(1)Thread是一个类，Runnable是一个接口，类只能被单继承，而接口可以被多实现。

(2)Runnable表示一个线程的顶层接口，Thread则是实现了Runnable。无论是使用Thead类还是实现Runnable接口，都需要去实现run()。

(3)站在面向对象的思想来说，Runnable是一个任务，而Thread才是真正处理线程，一般是实现Runnable用于定义具体的任务，然后交由Thread来处理，达到松耦合的目的。

### 11.ThreadLocal的底层原理

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

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

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

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

### 12.线程池的底层和工作原理

(1)如果此时线程池中的线程数量小于corePoolSize，即使线程池中的线程都处于空闲状态，也要创建新的线程来处理被添加的任务。

(2)如果此时线程池中的线程数量等于corePoolSize，但是缓冲队列workQueue未满，那么任务被放入缓冲队列。

(3)如果此时线程池中的线程数量大于等于corePoolSize，缓冲队列workQueue满，并且线程池中的数量小于maximumPoolSize，会创建新的线程来处理被添加的任务。

(4)如果此时线程池中的线程数量大于corePoolSize，缓冲队列workQueue满，并且线程池中的数量等于maximumPoolSize，那么通过 handler所指定的策略来处理此任务。

(5)当线程池中的线程数量大于corePoolSize时，如果某线程空闲时间超过keepAliveTime，线程将被终止。这样，线程池可以动态的调整池中的线程数。

### 13.当任务数超过线程池的核心线程数时，如何让它不进入队列

正常情况下，当任务数超过线程池的核心线程数，也就是线程池的核心线程数量等于核心线程数(corePoolSize)且核心线程都在执行任务时，新的任务会被放入阻塞队列等待线程执行。一旦堵塞队列满了的情况下，才会在线程池中的线程数小于最大线程数时再创建一个工作线程来执行任务。如果想让任务数超过线程池的核心线程数时，不让它进入堵塞队列，那么可以在通过构造方法创建线程池的时候，指定堵塞队列的类型为SynchronousQueue(同步队列)。该类型是一种特殊的阻塞队列，其本身是没有容量大小的，进去一个元素，必须等待取出之后，才能再往里面放一个元素。它可以避免任务进入堵塞队列，而是直接启动最大线程数量去处理任务。

### 14.如果一个线程两次调用start()方法会出现什么问题

java里的线程只能调用一次start()方法，第二次调用会抛出一个IllegalThreadStateException。一个线程本身是具有一个生命周期的，java中线程的生命周期包括6中状态。

(1)NEW(新建)，表示线程创建好了但还没有调用start()进行启动。

(2)RUNNABLE(就绪)，表示线程可能在运行，也可能在就绪队列中等待操作系统进行调度分配CPU资源。

(3)BLOCKED(阻塞)，表示线程处于锁等待状态。

(4)WAITING(等待)，表示线程处于条件等待状态。

(5)TIMED\_WAITING(超时等待)，和WAITING状态相同，只是它多了一个超时条件触发。

(6)TERMINATED(终止)，表示线程执行结束。

重复调用start()方法不管是从线程的安全性考虑，还是线程本身的逻辑来看都是不合理的，因此java中在调用start()方法时会判断线程的运行状态。

### 15.线程池如何知道一个线程的任务已经执行完成

### 16.守护线程

守护线程是专门为用户线程提供服务的一种线程，它的生命周期是依赖用户线程的，只有JVM仍然存在用户线程正在运行的情况下，守护线程才有存在的意义。守护线程和用户线程的创建方式是相同的，只需要额外设置线程的setDaemon方法即可将线程设置为守护线程。基于守护线程的特性，其更适合一些后台通用型服务的一些场景里面，比如JVM里面的垃圾回收，它不适合应用在一些线程池和IO的一些任务场景里，因为一旦JVM退出，则守护线程也会立即结束，会造成任务没有执行完或者资源没有被释放的一些问题。

## 设计模式

### 1.单例模式的思想是什么？请写一个代码体现

答：思想：保证类在内存中只有一个对象。

代码在项目learn的pattern.pattern3包下。

补充：

开发时：饿汉式(不会出问题)

面试时：懒汉式(多线程环境下可能会出问题，需要使用同步锁)

jdk中的Runtime类是单例模式的饿汉式

Runtime：每个Java应用程序都有一个Runtime类实例，使应用程序能够与其运行的环境

相连接。

exec(String command)：可以用来执行cmd命令。

### 2.介绍下策略模式和观察者模式

策略模式和观察者模式都属于行为模式。

(1)策略模式主要是用在根据上下文动态去控制类的行为的一个场景，一方面可以解决多个if...else判断带来的代码复杂性和维护性的一个问题，另外各方面把类的不同行为进行封装，使得程序可以进行动态的扩展和替换，增加了程序的灵活性，比如像支付路由这样一个场景就可以使用策略模式。

(2)观察者模式主要是用在一对多的对象依赖关系的一个处理过程中实现某一个对象状态变更之后的感知的一个场景。一方面可以降低对象依赖关系的耦合度，弱化依赖关系。另外一方面通过这种状态的通知机制可以保证这种依赖对象之间的状态协同。在spring源码中有大量运用到这种观察者模式来实现事件的传播和感知。

## 基础知识

### 1.byte型变量和常量相加的问题

byte b1 = 3,b2 = 4,b;

b=b1+b2;

b=3+4;

哪句会编译失败？为什么？

答：第二句会失败。因为byte类型的变量做加法，首先b1和b2都转化成int，再做加法，而结果自然也是int型，但b是byte型，不是int，所以报错，int转byte可能会损失精度；

但是常量相加，会先将结果计算出来，然后看是否在byte范围内，如果在就不报错，所以第三句不报错。

### 2. byte型数据溢出的计算

byte b = 130;有没有问题？如果想正常赋值，怎么做？结果是多少呢？

答：有问题，因为byte的范围是-128~127，130超出了byte的范围。要想正确赋值，可以强制转换，即byte b = (byte)130;结果是-126。

### 3.short型变量相加的问题

short s=1; s = s+1;

short s=1; s+=1;

以上两个代码有没有问题，如果有，那里有问题？

答：第一个会报可能损失精度的错误(参考byte型变量的相加)，而第二个不会。因为s+=1;等价于s = (s的数据类型)(s + 1);相当于强制转换了。

### 4.请自己实现两个整数变量值的交换

int a = 10;

int b = 20;

答：(1) 借助第三个变量

int c = a;

a = b;

b = c;

(2) 使用异或运算符^

a = a ^ b;

b = a ^ b;

a = a ^ b;

(3) 用变量相加的做法

a = a + b;

b = a – b;

a = a – b;

(4) 一句话搞定

b = (a+b) – (a=b);

### 5.用最有效率的方式写出2乘以8的结果

答：2 << 3，即2 \* 2的3次幂，等于16。

<<即为左移，即按二进制形式把所有的数字向左移动对应的位数，高位移出(舍弃)，低位的空位补零。

### 6.创建类的对象时，静态代码块、构造代码块、构造方法的执行顺序

答：(1)若有父类，顺序为：

静态代码块Fu—>

静态代码块Zi—>

构造代码块Fu—>

构造方法Fu—>

构造代码块Zi—>

构造方法Zi。

(2)若没有父类，顺序为：

静态代码块—>

构造代码块—>

构造方法。

补充：static{语句} 为静态代码块，{语句}为构造代码块。

### 7.Override和Overload的区别，Overload能改变返回值类型吗？

答：Override：方法重写，指在子类中，出现和父类中一模一样的方法声明的现象。

Overload：方法重载，指在用一个类中，出现的方法名相同，但参数列表不同的现象。

方法重载能改变返回值类型，因为它和返回值类型无关。

### 8.this和super分别代表什么，以及他们各自的使用场景

答：this：代表当前类的对象引用。

super：代表父类存储空间的标识。(可以理解为父类的引用，通过这个东西可以访问父类的成员)

场景：

成员变量：

this.成员变量

super.成员变量

构造方法：

this(…)

super(…)

成员方法：

this.成员方法

super.成员方法

### 9.String s = new String(“hello”)和String s = “hello”的区别？

答：前者会创建1个或2个对象，后者创建1个或0个对象。

String s = new String("hello")首先会在堆中创建一块内存，内存地址返回给栈中s，然后因为"hello"是常量，JVM会去方法区中的字符串常量池查看是否有"hello"字符串的对象，没有的话就分配一个空间来存放"hello"，并将其空间地址存入堆中new出来的对象中；

对于String s="hello"这样创建的对象，JVM会直接检查字符串常量池是否已有"hello"字符串对象，如没有，就分配一个内存存放"hello"，如有了，则直接将字符串常量池中的地址返回给栈中的s而不经过堆中new的对象。

### 10.String、StringBuffer、StringBuilder的区别？

答：String内容是不可变的，其内部的value值被final修饰，变更String值时会产生新的String对象，线程安全。而StringBuffer、StringBuilder内容可变。StringBuffer是同步的，线程安全，底层使用了synchronized，但是效率低；StringBuilder是不同步的，线程不安全，但效率高。

String的值最终都会存储在运行时常量池中，运行时常量池属于方法区；StringBuffer、StringBuilder的值都是存储在堆内存中。

### 11.StringBuffer和数组的区别？

答：二者都可以看作是一个容器，用来装其他的数据。但是StringBuffer的数据最终是一个字符串数据；而数组可以放置多种数据，但必须是同一种数据类型的。

### 12.Integer赋值问题

Integer i1 = 128;

Integer i2 = 128;

System.out.println(i1 == i2);

Integer i3 = 127;

Integer i4 = 127;

System.out.println(i3 == i4);

二者的输出结果是什么？为什么？

答：==比较的是两边的地址值，结果是false和true。

因为java有一个内置的数据缓冲池，范围-128~127，i1和i2的值是128，不在范围内，所以会创建两个不同的对象；而i3和i4的值是127，直接去缓冲池里拿数据，因为他们的地址一致，都是127的地址，所以是true。

### 13.final、finally和finalize的区别

答：final修饰类，类不能被继承；修饰变量，变量是常量；修饰方法，方法不能被重写。

finally：是异常处理的一部分，用来释放资源。

finalize：是Object的一个方法，用于垃圾回收。

### 14.如果catch里面有return语句，请问finally里面的代码还会执行吗？如果会，是在return前还是后呢？

答：会执行，会在return前执行，准确的说是在中间执行。

### 15.java常见的异常

(1) java.lang.NullPointerException  
这个异常大家肯定都经常遇到，异常的解释是"程序遇上了空指针"，简单地说就是调用了未经初始化的对象或者是不存在的对象，这个错误经常出现在创建图片，调用数组这些操作中，比如图片未经初始化，或者图片创建时的路径错误等等。  
对数组操作中出现空指针，很多情况下是一些刚开始学习编程的朋友常犯的错误，即把数组的初始化和数组元素的初始化混淆起来了。数组的初始化是对数组分配需要的空间，而初始化后的数组，其中的元素并没有实例化，依然是空的，所以还需要对每个元素都进行初始化。  
(2) java.lang.ClassNotFoundException  
因找不到指定的类而导致的异常，一般是使用Class.forName()时类名写错了，或者没有导包。  
(3) java.lang.ArithmeticException  
这个异常的解释是"数学运算异常"，比如程序中出现了除以零这样的运算就会出这样的异常，对这种异常，大家就要好好检查一下自己程序中涉及到数学运算的地方，公式是不是有不妥了。  
(4) java.lang.ArrayIndexOutOfBoundsException  
这个异常相信很多朋友也经常遇到过，异常的解释是"数组下标越界"，现在程序中大多都有对数组的操作，因此在调用数组的时候一定要认真检查，看自己调用的下标是不是超出了数组的范围。  
一般来说，显示调用不太容易出这样的错，但隐式调用就经常出错了，还有一种情况，是程序中定义的数组的长度是通过某些特定方法决定的，不是事先声明的，这个时候，最好先查看一下数组的length，以免出现这个异常。  
(5) java.lang.IllegalaccessException  
这个异常的解释是"没有访问权限"，当应用程序要调用一个类，但当前的方法即没有对该类的访问权限便会出现这个异常。对程序中用了package的情况下要注意这个异常。

(6) ClassCastException

类转换异常，将一个不是该类的实例转换成这个类就会抛出这个异常。

(7) NoSuchMethodException

没有这个方法异常，一般发生在反射调用方法的时候

(8) FileNotFoundException

找不到文件异常，如果文件不存在就会抛出这种异常。

(9) IOException

IO，即：Input、Output，我们在读写磁盘文件、网络内容的时候经常会生的一种异常，这种异常是受检查异常，需要进行手工捕获。

### 16.所有整型包装类对象之间值的比较

比如：Integer与Integer之间值的比较

在-128~127之间时，Integer会从常量池中取数据，而==比较对象类型时比较的是地址值。

所以Integer a = 1;Integer b = 1;System.out.println(a==b);结果为true。

但Integer a = 200;Integer b = 200;System.out.println(a==b);结果为false。因为200不在-128~127之间，a和b都是new出来的对象，二者的地址值根本不相等。

所以如果要进行整型包装类对象之间值的比较，最好用equals()。

### 17. java面向对象有哪些特征

(1) 封装

封装隐藏了类的内部实现机制。对属性进行了封装：外界只能通过特定的方法进行访问。对方法进行了封装：外界只能通过定制好的方式调用，不用了解方法内部逻辑，方便使用。保护了数据。便于修改，增强了代码的可维护性和复用性。

(2) 继承

继承是从已有的类中派生出新的类，即子类继承自父类。当子类通过extends关键字继承了父类后，便继承了父类的属性和方法（注意：子类继承了父类的所有属性和方法。但父类的私有属性和方法，子类是不能直接访问的，只是拥有但无法使用）。同时子类还可以具备父类所不具备的属性或方法。

通过继承避免了对各个类中重复的属性和方法进行反复描述。增强了代码的复用性。代码更加简洁。

(3) 多态

多态是类和类的关系。多态必须具备三个条件：1.要有继承关系 2.要有方法的重写 3.父类的引用指向子类的对象。

正是因为两个类有继承关系，并存在方法的重写，才能在调用时有父类的引用指向子类的对象。

通过多态可以是一个对象在不同载体中呈现不同的状态。即同一个父类引用在调用同一个方法时会得到不同的结果。多态减少了类的代码量，提高了代码的可维护性、可扩展性和复用性。

### 18.jdk1.8中的新特性

(1) 接口的默认方法

(2) Lambda表达式

(3) 函数式接口

仅有一个抽象方法的接口被称为函数式接口。

(4) StreamAPI

(5) Optional类

(6) 新的日期时间类

### 19.抽象类和接口的区别

(1) 抽象类

可以定义构造方法；

可以有抽象方法和具体方法；

可以定义成员变量；

可以有静态方法；

(2) 接口

没有构造方法；

jdk1.8之前只有抽象方法，没有具体方法；

没有静态方法；

没有成员变量，只能是成员常量；

其成员只能被public修饰；

### 20.序列化和反序列化的理解

序列化的核心目的是为了解决网络通信之间对象传输的问题，即如何把JVM进程中的一个对象跨网络传输到另一个JVM进程中进行恢复。序列化就是把内存中的对象转化为字节流，以便实现存储和传输。反序列化就是根据文件或者网络上获取到对象的字节流，根据流中的对象描述信息和状态重新构建一个新的对象。序列化的前提是为了保证通信双方对于对象的一个可识别性，所以很多时候会把对象先转化为通用的解析格式，比如json、xml，然后再把它们转化为数据流进行网络传输。市面上开源的序列化技术有Json、Xml、Protobuf、Kyro等，实际选择时需要考虑序列化之后的数据大小、序列化的性能、是否支持跨平台和跨语言、技术的成熟度。

### 21.IO和NIO的区别

(1)IO流可以实现数据从磁盘、内存、网络中读取以及写入，java中提供了字符流和字节流两种方式实现数据流的操作。

(2)当程序面向网络进行数据IO操作的时候，java提供了Socket的方式实现网络IO的通信，这种方式可以实现数据的网络传输。基于Socket的IO通信属于阻塞IO，在连接以及IO事件未就绪的情况下，当前的连接会处于阻塞等待的状态，一般有连接处于阻塞状态，后续的连接都得去等待。NIO是jdk1.4新增的一种new IO，它提供了一种非阻塞的特性，通过NIO进行网络数据传输的时候，如果连接或IO未就绪，服务端不会阻塞当前连接，而是继续轮询后续的连接来处理，能处理更多的并发连接。

### 22.为什么比较两个Integer不能使用==来判断

先说应该使用equals()来进行比较！

在Integer中引入了享元模式，对-128到127之间的数据做了一层缓存，如果Integer类型的值在-128到127之间，会直接从缓存中获取Integer对象的一个实例，否则会创建一个新的Integer对象，优点是减少频繁创建Integer对象所带来的内存消耗，从而提升性能。因此如果定义了两个Integer对象且它们的值都在-128到127之间，用==来判断则必然是true，因为它俩指向的内存地址是同一个。



### 23.Integer和int的区别，为什么要设计封装类

(1)作为成员变量来说，Integer的初始值是null，而int的初始值是0。

(2)Integer是一个对象，因此它存储在堆内存里面，而int类型则是存储在栈空间中。

(3)Integer是一个对象类型，它封装了很多的方法和属性，在使用时更加灵活。

设计封装类的原因是：Java本身是一个面向对象的语言，一切操作都是以对象作为基础的，比如集合里面存储的元素也只支持Object类型。

## 集合

### 1.Map集合的key可以为null吗？

答：HashMap可以，键[和值](https://wenwen.sogou.com/s/?w=%E5%92%8C%E5%80%BC&ch=ww.xqy.chain)都可以为空或null，但是不能同时存在两个相同的键，意思就是不能把两个不同的键都设为空或null，因为键是唯一的。



### 2. ArrayList和LinkedList的区别

(1) 数据结构不同

ArrayList是Array(动态数组)的数据结构，LinkedList是Link(链表)的数据结构。

(2) 执行不同操作时效率不同

当取值和赋值时，ArrayList可以直接通过下标来操作，因此速度比LinkedList快。而对数据进行增加和删除时，LinkedList更快，因为ArrayList在增加和删除后，会对操作点之后所有数据的下标索引产生影响，需要进行操作点之后所有数据的移动。

(3) 自由性不同

ArrayList使用时需要指定容量，但它有下标，使用方便；LinkedList使用时不需要指定容量，能够动态的随数据量的变化而变化，但是它没有下标，使用不方便。

(4) 主要控件开销不同

ArrayList主要控件开销在于需要在List列表预留一定空间；而LinkedList主要控件开销在于需要存储节点信息以及节点指针。

### 3. 高并发中的集合有哪些问题

(1) 第一代线程安全集合类：Vector、Hashtable

保证线程安全的方法：使用synchronized修饰方法 效率低下。

(2) 第二代非线程安全集合类：ArrayList、HashMap

线程不安全，但是性能好，用于代替Vector、Hashtable。

如果需要线程安全，可以调用一层方法(装饰器模式)来实现：

Collections.synchronizedList(list);

Collections.synchronizedMap(map);

底层使用的也是synchronized。

(3) 第三代线程安全集合类

java.util.concurrent.ConcurrentHashMap(jdk1.8的该类底层更新，放弃使用Lock锁)

java.util.concurrent.CopyOnWriteArrayList：

a.首先CopyOnWriteArrayList内部也是通过数组来实现的，在添加元素时，会复制一个

新的数组，写操作在新数组上进行，读操作在原数组上进行。

b.写操作会加锁，防止出现并发写入数据会丢失的情况。

c.写操作结束后，会把原数组指向新数组。

d. CopyOnWriteArrayList允许在写操作时读取数据，大大提高了读的性能，因此适合读

多写少的场景。但是CopyOnWriteArrayList也因此比较占用内存，同时读取的数据

可能不是最新的数据，所以不适合实时性很高的场景。

java.util.concurrent.CopyOnWriteArraySet

### 4.Arraylist的扩容机制

Arraylist是一个数组结构的存储容器，在创建对象时默认数组长度是10，也可以在创建对象时手动指定初始长度。当Arraylist没有足够的容量存储数据时，会触发自动扩容。首先创建一个新的数组，其长度为原来数组长度的1.5倍，然后用Arrays.copyOf()将老数组中的数据拷贝到新数组里，扩容完成以后，再将新数据添加到新的数组里。

### 5.HashMap的扩容机制

(1)jdk1.7版本：

a.首先生成一个长度两倍于原数组的新数组。

b.遍历老数组中每个位置上的链表上的每个元素。

c.取每个元素的key，并基于新数组长度，计算出每个元素在新数组中的下标。

d.将元素添加到新数组中。

e.所有元素转移完后，将新数组赋值给HashMap对象的table属性。

(2)jdk1.8版本：

a.首先生成一个长度两倍于原数组的新数组。

b.遍历老数组中每个位置上的链表上或红黑树上的每个元素。

c.如果是链表，则直接将链表中的每个元素重新计算下标，并添加到新数组中。

d.如果是红黑树，则先遍历红黑树，先计算出红黑树中每个元素对应在新数组中的下标

位置。

e.统计每个下标位置的元素个数，如果该位置下的元素个数超过了8，则生成一个新的

红黑树，并将根节点的添加到新数组的对应位置。如果该位置下的元素个数没有超

过8，那么则生成一个链表，并将链表的头节点添加到新数组的对应位置。

f.所有元素转移完后，将新数组赋值给HashMap对象的table属性。

## 并发

### 1.如何理解volatile关键字

在并发领域中，存在三大特性：原子性、有序性、可见性。volatile关键字用来修饰对象的属性，在并发环境下可以保证这个属性的可见性。对于加了volatile关键字的属性，在对这个属性进行修改时，会直接将CPU高级缓存中的数据写回到主内存，对这个变量的读取也会直接从主内存中读取，从而保证了可见性，底层是通过操作系统的内存屏障来实现的，由于使用了内存屏障，所以会禁止指令重排，同时也就保证了有序性，在很多并发场景下，如果用好volatile关键字可以很好的提高执行效率。

### 2. ReentrantLock中的公平锁和非公平锁的底层实现

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

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

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

### 3. ReentrantLock中的tryLock()和lock()方法的区别

(1)lock()没有返回值，且这个方法在没有获取到锁的时候会阻塞，这样后续代码就不会执行。

(2)tryLock()会返回一个boolean类型的返回值，当返回值为true时表示获取到了锁，当返回值为false时表示没有获取到锁。且该方法不会阻塞。

### 4.死锁发生的原因以及如何避免死锁

死锁就是两个或者两个以上的线程在执行的过程中去争夺同样的一个共享资源造成的相互等待的一个现象，如果没有外部的干预，线程会一直阻塞，无法往下去执行。这样一直处于相互等待的线程称为死锁线程。导致死锁需要同时满足四个条件：

(1)互斥条件，共享资源X和Y只能被一个线程占用。

(2)请求和保持条件，线程T1已经取得共享资源X，在等待共享资源Y的时候，不释放共享资源X。

(3)不可抢占条件，其他线程不能强行抢占线程T1占有的资源。

(4)循环等待条件，线程T1等待线程T2占有的资源，线程T2等待线程T1占有的资源就是循环等待。

导致死锁时只能通过人工干预来解决，比如重启服务或者kill掉这个线程，所以最好是根据死锁的四个条件出发避免死锁的发生。其中互斥条件是没有办法破环的，因为它是互斥锁的基本约束，其他三个条件都可以被破环。比如请求和保持条件，可以一次性申请所有的资源，这样就不存在锁要等待了。对于不可抢占条件，占用部分资源的线程在进一步申请其他资源的时候如果申请不到，可以主动释放它占有的资源，这样不可抢占的条件就被破环掉了。对于循环等待，可以按序申请资源来预防，按序申请就是指资源是有线性顺序的，申请的时候可以先申请资源序号小的，然后再去申请资源序号大的，这样线性化之后就不存在循环了。

### 5.MVCC过程中会加锁吗

MVCC全称Multi-Version Concurrency Control(多版本并发控制)，它确保在高并发情况下多个事务读取数据的时候不加锁也能多次读取相同的值。MVCC通常不需要加锁来控制并发的访问，相反，每个事务都可以读取到已提交的快照而不需要去获取共享锁或者排他锁。在写操作的时候，MVCC会使用一种叫“写时复制”的技术，也就是在修改数据之前，先将数据复制一份从而创建一个新的快照。当一个事务需要修改数据的时候，MVCC会首先检查修改数据的快照版本号是否与该事务的快照版本一致，如果一致就表示可以修改这条数据，否则这个事务需要等待其他事务完成对该数据的修改。另外，这个事务在新快照的修改的结果不会影响原始数据，其他事务可以继续读取原始数据的快照从而解决脏读和不可重复读的问题。所以正是有了MVCC机制，让多个事务对同一个数据进行读写的时候不需要加锁也不会出现读写冲突问题。

### 6.介绍幂等和如何解决幂等性问题

在计算机编程领域，幂等一般指的是方法被多次重复执行的时候所产生的影响和第一次执行的影响是相同的。之所以考虑幂等性问题，是因为在网络通信里面存在两种行为可能会导致接口被重复调用。

(1)用户的重复提交或者用户的恶意攻击导致这个请求会被多次重复执行。

(2)在分布式架构中，为了避免网络通信导致的数据丢失，在服务之间进行通信的时候都会设计超时重试的机制，而这种机制有可能导致服务端接口被重复调用。

解决的方式有这几种。

(1)使用数据库的唯一约束实现幂等。比如对于数据插入类的场景。创建订单，因为订单号肯定是唯一的，所以如果是多次调用就会触发数据库的唯一约束异常，从而避免一个请求创建多个订单的问题。

(2)使用redis里面提供的setNX指令。比如对于MQ消费的场景，为了避免MQ重复消费导致数据多次被修改的问题，可以在接收到MQ的消息时，把这个消息通过setNX写入到redis里面，一旦这个消息被消费过，就不会再次消费。

(3)使用状态机来实现幂等。所谓状态机是指一条数据的完整运行状态的转换流程，比如订单状态。因为它的状态只会向前变更，所以多次修改同一条数据的时候，一旦状态发生变更，那么对这条数据修改造成的影响只会发生一次。

(4)基于token、去重表机制来实现。

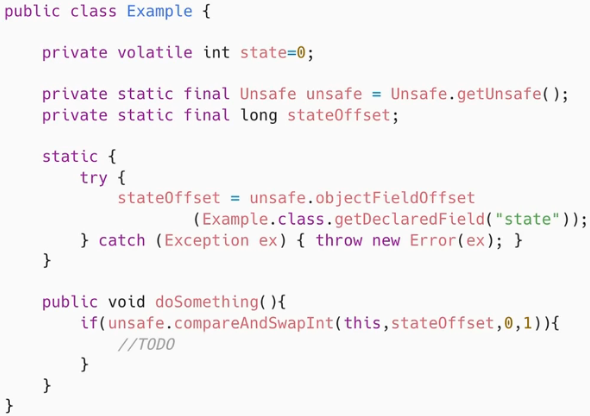
补充：无论是哪种实现方法，总归属于两种方式中的一种。一是接口只允许调用一次，比如唯一约束、基于redis的锁机制。二是对数据的影响只会触发一次，比如状态机、乐观锁。

### 7.CAS机制

CAS是Java中Unsafe类里面的一个方法，全称是compare and swap(比较并交换)。其主要功能是保证在多线程的环境下对共享变量修改的一个原子性。多线程环境下有问题的代码：



经过CAS修饰后的多线程环境安全的代码：



上述代码中起主要作用的是Unsafe类的compareAndSwapInt方法，这个方法有4个参数，分别是当前对象实例、成员变量在内存地址中的一个偏移量、预期值、替换值。CAS机制会去比较变量state在内存地址的偏移量对应的值和传入的预期值0是否相同，相同即将0改为替换值1，否则返回false。这个过程是一个原子性的，不会存在任何线程安全问题。compareAndSwap是一个native方法，实际上它还是会面临多线程问题，就是先从内存地址中读取state值，然后再去比较，最后再去修改。所以在compareAndSwap的底层实现中，如果是在多核CPU指令下会增加一个Lock指令来对缓存或者总线去加锁，从而保证比较并交换这两个操作的原子性。CAS主要是使用在并发场景下，第一个是jdk的JUC包里面的一些原子类，比如AtomicInteger、AtomicLong。第二个是实现多线程对共享资源竞争的互斥性质，比如AQS、ConcurrentHashMap、ConcurrentLinkedQueue等。

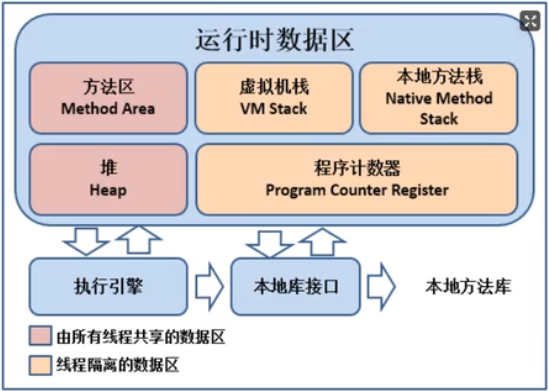
### 8.雪花算法

雪花算法是生成分布式全局唯一id的一个算法，它会得到一个64位长度的long类型的数据。这个64位的数据有四个部分组成。第一个bit位是一个符号位，因为id不会是负数所以它一般是0。接下来的41个bit位来表示毫秒单位的时间戳。再用10个bit位来表示工作机器的id。最后用12个bit位来表示递增的序列号。把这64个bit位拼接成一个long类型的数字，这就是雪花算法的一个实现。

## JVM

### 1.JVM中哪些是线程共享区

堆区和方法区是所有线程共享的。栈、本地方法栈、程序计数器是每个线程独有的。



### 2.JVM中哪些可以作为GC Root

什么是GC Root。JVM在进行垃圾回收时，需要找到“垃圾”对象，也就是没有被引用的对象，但是直接找“垃圾”对象是比较耗时的，所以反过来，先找“非垃圾”对象，也就是正常对象。那么就需要从某些“根”开始去找，根据这些“根”的引用路径找到正常对象，而这些“根”有一个特征，就是它只会引用其他对象，而不会被其他对象引用，例如：栈中的本地变量、方法区中的静态变量、本地方法栈中的变量、正在运行的线程等可以作为GC Root。

### 3.对象的创建过程

在实例化对象时，JVM首先会检查目标对象是否已经被加载并初始化，如果没有，则JVM需要立即加载目标类，然后调用目标类的构造器完成初始化。目标类的加载是通过类加载器实现的，主要是将类的字节码文件加载到内存中，然后是初始化，主要是将静态变量、成员变量、静态代码块进行初始化。当目标类被初始化以后，在常量池中就可以找到对应的类元信息了，并且目标对象的大小在类加载完成之后就已经确定了。之后是根据目标对象的大小在堆内存中为实例对象分配内存空间。内存分配的方式有两种，第一种是指针碰撞，第二种是空闲列表，JVM会根据堆内存是否规整来确定分配方式。接下来JVM会把目标对象中的普通成员变量赋默认值，比如int类型是0，String类型是null，这主要是保证实例对象中的变量不用初始化就可以直接使用。然后JVM还需要对目标对象的对象头做一些设置，比如对象所属的类元信息、对象的GC分代年龄、hashcode、锁标记等。最后是执行目标对象内部生成的init方法，初始化成员变量的值，执行构造块，调用目标对象的构造方法完成对象的创建。init方法是java文件编译之后在字节码文件里面生成的，它是一个实例构造器，会把构造块、变量初始化、调用父类构造器等这样一些操作组织在一起，所以调用init方法可以完成一系列的初始化动作。

## 算法

### 1.递归与while循环的通俗比较

(1)递归的优点：代码更简洁清晰，可读性更好。缺点：由于递归需要系统堆栈，所以空间消耗要比非递归代码要大很多。而且，如果递归深度太大，可能系统内存撑不住。

(2)while循环优点：速度快，结构简单。缺点：并不能解决所有的问题。

(3)循环能干的事递归都能干，反之则不一定。如果使用循环并不困难的话，最好使用循环。

# 其他

## Web

### 1.Tomcat如何优化

对于Tomcat调优，可以从两个方面来进行调整：内存和线程。

首先启动Tomcat，实际上就是启动了一个JVM，所以可以按JVM调优的方式来进行调整，从而达到Tomcat优化的目的。另外Tomcat中设计了一些缓存区，比如appReadBufSize、bufferPoolSize等缓存区来提高吞吐量。

还可以调整Tomcat的线程，比如调整minSpareThreads参数来改变Tomcat空闲时的线程数，调整maxThreads参数来设置Tomcat处理连接的最大线程数。并且还可以调整IO模型，比如使用NIO、APR这种相比于BIO更加高效的IO模型。

### 2.Cookie和Session的区别

(1)Cookie是客户端浏览器用来保存服务端数据的一种机制，当我们通过浏览器去进行网页访问时，服务器可以把某一些状态数据以key-value的形式写入到Cookie里面并存储到客户端浏览器，然后客户端下一次再访问服务器的时候就可以携带这些状态数据发送到服务器端，服务器端可以根据Cookie里面携带的内容去识别使用者。

(2)Session表示一个会话，它是属于服务器端的一个容器对象，默认情况下，它会针对每一个浏览器的请求，Servlet都会分配一个Session对象，Session本质上可以认为是一个ConcurrentHashMap，它可以用来存储当前会话产生的一些状态数据。

(3)Http协议本身是一个无状态协议，也就是服务器端并不知道客户端发送的多次请求是否属于同一个用户，Session可以弥补Http无状态的一个不足。服务器端可以利用Session来存储客户端在同一个会话里面产生的多次请求的记录，基于Session和Cookie就可以实现一个有状态的Http协议。原理：首先客户端第一次访问服务器端时，服务器端会针对这次请求创建一个会话并生成一个唯一的SessionId来标注这个会话，然后服务器端把这个SessionId写入到客户端浏览器的Cookie里面用来实现客户端状态的一个保存。在后续的请求里面都会携带SessionId，服务器端就可以根据SessionId来识别当前这个会话的状态。

### 3.敏感数据怎么加解密和传输

常见的加密算法有两种，一种是对称加密，即通信双方共享同一个密钥。另一种是非对称加密，即通过公钥和私钥分别进行加密和解密。对于客户端和服务端的数据传输，可以采用非对称加密的方式来实现。首先客户端用提前分配好的公钥对数据加密，然后再把密文传输到服务器端，服务器端通过私钥解密。常见的非对称加密算法有RSA、DSA两种。除了加密算法以外，还需要通过安全的通信协议进行传输，可以采用https协议。最后是确保公钥和私钥存储的安全性，防止被第三方拿到密钥后破解内容。

## 网络

### 1.什么是IO的多路复用机制

IO的多路复用机制其核心思想是让单个线程去监视多个连接，一旦某个连接就绪，也就是触发了读/写事件的时候，那么就会去通知对应的应用程序去主动获取这个就绪的连接去进行读写的操作，也就是在应用程序里面可以使用单个线程同时去处理多个客户端连接。在对系统资源消耗比较小的情况下去提升服务端的连接处理数量。在IO的多路复用机制的实现原理中，客户端请求到服务端以后，此时客户端在传输数据的过程中为了避免Server端在read客户端数据的时候堵塞，那么服务端会先把请求注册到一个Selector的一个复路器，服务端此时不需要等待，只需要启动一个线程通过selector.select()方法去堵塞轮询复路器上就绪的channel就可以了。常见的IO的多路复用机制的实现有很多，比如select、poll、epoll等，这些都是Linux系统中提供的实现，其中select、poll都是基于轮询的方式去获取就绪的连接的，而epoll是基于事件驱动的方式去获取就绪连接的，从性能的角度来看，基于事件驱动的方式更好。

### 2.如何提高接口的性能

(1)从接口本身的实现维度出发：

a.如果接口中有操作数据库层面的代码，可以优化数据库IO的效率。比如SQL优化、数据库层面的优化等。

b.如果存在部分频繁访问数据库的热数据，可以采用缓存机制。

c.如果涉及到远程调用或者耗时的方法调用，可以采用异步方式避免同步阻塞。

d.代码本身的优化，可以利用合适的算法减少时间复杂度，避免一些很明显的重复计算等问题。

(2)宏观链路纬度出发：

a.网络带宽，带宽的大小会影响数据的传输效率。

b.服务器的硬件资源如CPU、内存等，会影响接口中代码的执行效率。

c.单个部署节点的计算能力瓶颈，也会影响接口性能，可以采用分布式部署的方式优化。

### 3.浏览器发出一个请求到收到响应经历了哪些步骤

(1)浏览器解析用户输入的URL,生成一个HTTP格式的请求。

(2)先根据URL域名从本地hosts文件查找是否有映射ip,如果没有将域名发送给电脑所配置的DNS进行域名解析,得到ip的地址。

(3)浏览器通过操作系统将请求通过四层网络协议发送出去。

(4)途中可能会经过各种路由器,交换机,最终到达服务器。

(5)服务器收到请求后,根据请求所指定的端口,将请求传递给绑定了该端口的应用程序,比如8080被tomcat占用了。

(6)tomcat接受到请求数据后,按照http协议的格式进行解析,解析得到所要访问的servlet。

(7)然后servlet来处理这个请求,如果是SpingMVC中的DispatcherServlet,那么则会找到对应的Controller的方法,并执行该方法得到结果。

(8)Tomcat得到响应结果后封装HTTP响应的格式,并将再次通过网络发送给浏览器所在的服务器。

(9)浏览器所在的服务器拿到结果后再传递给浏览器,浏览器则负责解析并渲染。

### 4.http请求头和响应头中的信息

(1)HTTP Request Header 请求头

User-Agent：标识浏览器的详细信息，包括名称、版本、操作系统等。

Accept：指定客户端接受的数据类型，可以是文本、HTML、XML、JSON 等等。

Accept-Encoding：指定客户端可以接受的压缩编码类型，如 gzip、deflate 等。

Accept-Language：指定客户端接受的语言类型和优先级。

Cache-Control：指定客户端的缓存策略，如 no-cache、max-age 等。

Connection：指定客户端和服务器之间的连接类型，如 keep-alive、close 等。

Cookie：用于客户端和服务器之间传递会话信息，如用户登录状态等。

Host：指定要访问的服务器地址。

Referer：指定请求来源网页的 URL。

User-Agent：标识浏览器的详细信息，包括名称、版本、操作系统等。

(2)HTTP Responses Header 响应头

Cache-Control(对应请求中的Cache-Control)：

Content-Type：

# 框架

## Mybatis

### 1. #{}和${}的区别

(1) #是一个占位符，$是拼接符

使用#{}方式引用参数的时候，Mybatis会把传入的参数当成是一个字符串，自动添加单引号；使用${}引用参数时，不做任何处理，会直接将值拼接在sql语句中，一般用作表名或者字段名的拼接。

(2) 使用#{}能够防止sql注入，${}不能避免注入攻击

#{}的方式引用参数，mybatis会先对sql语句进行预编译，然后再引用值，能够有效防止sql注入，提高安全性。${}的方式引用参数，sql语句不进行预编译。因此建议使用#{}的方式。

### 2.Mybatis是如何分页的

Mybatis的分页方式有三种：

(1)直接在select语句中增加数据库提供的分页关键字，然后在程序中传递当前页以及每页展示的条数就好了。

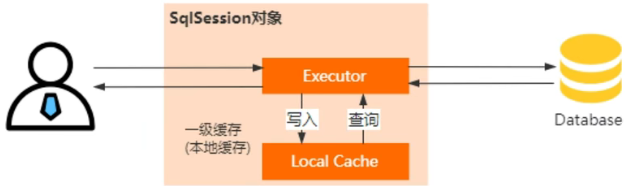
(2)使用Mybatis提供的RowBounds这样一个对象，实现内存的分页。这种方式不适合数据量大的场景，而且有可能会因为JDBC的分批次查询会频繁访问数据库。

(3)基于Mybatis里面的Interceptor拦截器，可以拦截select语句，在select语句执行前，动态去拼接分页的关键字。好处是统一处理分页，不用再单独做分页功能，PageHelper、Mybatis-Plus、tkmybatis都是基于拦截器做的分页处理。

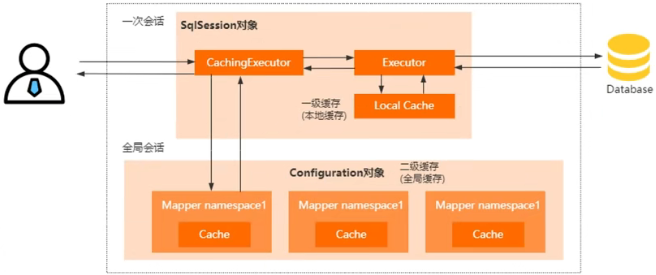
### 3.Mybatis的二级缓存

Mybatis设计了二级缓存的机制来提升数据的一个检索效率，也就是避免每次查询都去查询数据库。一级缓存是SqlSession级别的一个缓存，也叫本地缓存。因为每一个用户在执行查询的时候都需要使用SqlSession来执行，为了避免每一次都去查询数据库，Mybatis把查询出来的数据缓存到SqlSession的本地缓存里面，后续的Sql如果在命中缓存的情况下就可以去本地缓存直接读取这样一个数据。如果想要跨SqlSession级别的一个缓存，一级缓存无法实现，所以引入了二级缓存，也就是当多个用户在查询数据的时候，只要有任何一个SqlSession拿到了数据就会放入到二级缓存里面，其他的SqlSession就可以直接从二级缓存里面去加载数据。原理：

(1)在SqlSession里面会持有一个Executor，每一个Exactly里面会有个叫LocalCache的对象，当用户发起查询时，Mybatis会根据执行语句在Local Cache里面去查找，如果命中了就直接返回数据，如果没有命中，再去数据库查询出来，再写入到Local Cache里面。所以一级缓存的生命周期是SqlSession。需要注意的是在多个SqlSession或者分布式环境下可能会因为一级缓存导致脏读出现。



(2)二级缓存是在原来的Executor上做了一个装饰，引入了叫CachingExecutor这样一个装饰器。所以在进入一级缓存的查询之前，会先通过CachingExecutor进行二级缓存的查询。开启二级缓存之后，其会被多个SqlSession共享，所以它是一个全局的缓存。它的查询流程也变成了先查二级缓存，再查一级缓存，再查数据库。二级缓存实现了SqlSession之间的缓存数据的共享，同时缓存力度也能够控制到namespace的级别，并且还可以通过Cache这样一个接口来实现不同缓存实现类的一个组合，对Cache的可控性也更高了。



## Spring

### 1.spring中有两个相同id的bean，会报错吗

(1)在同一个xml配置文件中，不能存在id相同的两个bean，因为id表示bean的唯一标志符号。当id重复时，spring容器在启动时会报错，错误发生在spring对xml文件进行解析转化为BeanDefinition的阶段。

(2)在两个不同的spring配置文件中，可以存在id相同的两个bean，此时ioc容器在加载bean时，默认会把多个相同id的bean进行覆盖。

(3)spring3.x版本后，出现了@Configuration来声明一个配置类，@Bean用来注册bean。如果有多个相同名称的Bean，则ioc只会注册第一个声明Bean的实例。



### 2.@Resource和@Autowired的区别

(1)@Resource是jdk提供的一个注解，它默认先根据名称实现bean的依赖注入，如果找不到该名称的bean，会继续根据类型来注入。它有name和type两种属性，分别手动指定根据名称、类型注入。



(2)@Autowired是spring提供的一个注解，它默认根据类型实现bean的依赖注入。它有一个required属性，默认值为true，指强制要求bean实例的注入。当容器中有多个相同类型的bean时，启动时会报错，解决方法有两个。一是用@Primary修饰@Bean修饰的方法，@Primary表示主要的bean，这样@Autowired会优先使用被@Primary修饰的bean；二是和@Qualifier一起修饰变量，@Qualifier起到筛选的作用，可以根据bean的名称找到唯一的bean。



### 3.spring ioc的工作流程

(1)ioc全称Inversion Of Control(控制反转)，其核心思想是将对象的管理权限交给了容器。应用程序如果需要使用某个对象的实例可以直接从ioc容器中去获取。这种设计的好处是降低了程序里面对象与对象之间的耦合，使程序的体系结构更灵活。

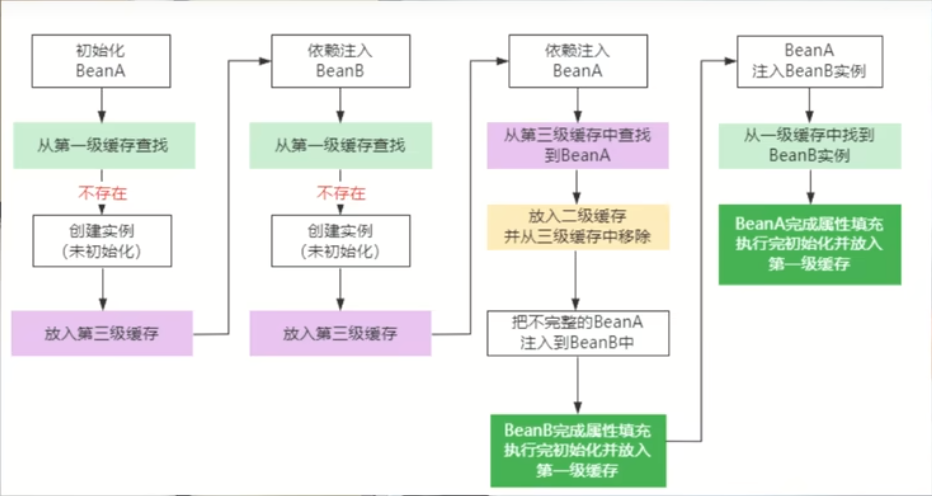
(2)spring提供了很多方式去声明一个Bean，比如xml配置文件中的<Bean>标签，或者通过@Service、@Bean等注解。spring在启动时会解析这些Bean，然后保存到ioc容器中。

(3)ioc容器的初始化阶段：将声明的Bean通过解析和加载后生成BeanDefinition，BeanDefinition包含bean的一些定义和一些基本属性，然后将BeanDefinition注册到ioc容器中。ioc容器会对这些注册的Bean的定义信息进行处理和维护，它是ioc控制反转的核心。

第二个阶段是完成Bean的初始化和依赖注入。首先是通过反射去针对没有设置lazy-init属性的单例bean进行初始化。然后完成Bean的依赖注入。最后是Bean的使用，通常是使用@Autowired、@Resource、BeanFactory.getBean()等方式从ioc容器中获取指定Bean的实例。另外针对设置了lazy-init属性以及非单例Bean的实例化是在获取每一次bean对象的时候调用bean的初始化方法来完成实例化的，且ioc容器不会去管理这些Bean。

### 4.spring如何解决循环依赖问题

Spring设计了三级缓存来解决循环依赖问题。第一级缓存里面存储完整的Bean实例，这些实例是可以直接被使用的；第二级缓存里面存储实例化后还没有设置属性值的Bean实例，即还没有进行依赖注入；第三级缓存是用来存放Bean工厂，它主要用来生成原始Bean对象并放到第二个缓存里面。三级缓存的核心思想是将Bean的实例化和Bean里面的依赖注入进行分离，采用一级缓存存储完整的Bean实例，采用二级缓存存储不完整的Bean实例，通过不完整的Bean实例解决循环依赖问题。第三级缓存主要是解决代理对象的循环依赖问题。



补充：三级缓存只能解决单实例的循环依赖，有四种无法解决的循环依赖需要手动修改。

(1)多实例的setter注入导致的循环依赖，需要把Bean改成单例。

(2)构造器注入导致的循环依赖，可以通过@Lazy注解解决。

@Lazy放在@Component及其衍生注解的类上表示在启动的时候不会创建bean对象，

当使用的时候才会创建。

@Lazy放在@Bean注解的方法上，表示在启动的时候不会创建bean对象，当使用的时

候才会创建。

@Lazy放在@Autowired注解的属性上，并不会直接给属性赋上真正的值，只是会赋值

一个代理对象，当真正使用到这个属性的时候，才会去容器中找到一个符合的对象。在使用的时候，也会先执行代理对象的逻辑，然后再是真正bean对象的逻辑。使用场景：循环依赖的时候可以在循环依赖的对象加上@Lazy注解。

@Lazy修饰构造方法和普通方法的形参上时和修饰属性是一样的，开始注入的只是代理

对象，当真正调用的时候才会调用对应对象的逻辑。

(3)DependsOn导致的循环依赖，这个只能找到依赖的地方迫使其不再循环依赖。

(4)单例的代理对象setter注入导致的循环依赖。可以使用@Lazy注解或者使用@DependsOn

指定加载先后关系。

### 5.@Component和@Bean的区别

(1)用途不同。@Component用于标识一个普通类；@Bean用于在配置类中声明和配置Bean对象。

(2)使用方式不同。@Component是一个类级别的注解，spring通过@ComponetScan扫描@Component修饰的类并将这些类注册到IOC容器中；@Bean修饰一个方法，用于在配置类中手动声明一个Bean的定义。

(3)控制权不同。@Component修饰的类是由spring创建和初始化的；@Bean允许开发人员手动控制Bean的创建和配置，因此更灵活。

### 6.@Conditional注解的作用

@Conditional是为Bean的装载操作提供一个条件判断，只有在满足条件的情况下，Spring才会把当前的Bean装载到ioc容器里面。这个条件的实现我们可以自定义完成，需要定义一个类实现Condition接口并重写matches方法实现自定义的逻辑。在SpringBoot中，对@Conditional注解做了进一步的扩展，比如@ConditionalOnClass、@ConditionalOnBean等一系列注解，可以直接根据语义来判断条件的装配。

### 7.spring中有哪些方式可以把Bean注入到IOC容器中

(1)在xml配置文件中使用<Bean>标签来声明Bean，Spring容器在启动时会加载并解析这个xml文件并把bean装载到IOC容器中。

(2)使用@CompontScan注解来扫描声明了@Controller、@Service、@Repository、@Component这些注解的类。

(3)使用@Configuration注解声明配置类并使用@Bean注解实现Bean的定义，这是xml配置方式的一种演变，是spring迈入到无配置化时代的里程碑。

(4)使用@Import注解导入配置类或者普通的Bean。

(5)使用FactoryBean工厂bean，动态构建Bean的实例，springcloud openFeigh里面的动态代理实例就是使用FactoryBean来实现的。

(6)实现ImportBeanDefinitionRegistrar接口，可以动态注入Bean实例。这个在springboot里面的启动注解有用到。

(7)实现ImportSelector接口，动态批量注入配置类或者Bean对象，这个在springboot里面的自动装配机制里面有用到。

### 8.为什么要使用spring框架

spring是一个轻量级的应用框架，它提供了IOC和AOP这两个核心的功能，其目的是简化企业级的应用开发，使开发者只需要关心业务的需求，不需要关心Bean的管理，而且还可以通过切面的方式对功能进行增强，从而减少代码的侵入性。其关键优势为：

(1)轻量，基本版本为2M大小。

(2)通过IOC实现Bean的生命周期的管理以及通过DI实现依赖注入，降低了代码的耦合度。

(3)提供了面向切面编程(AOP)。

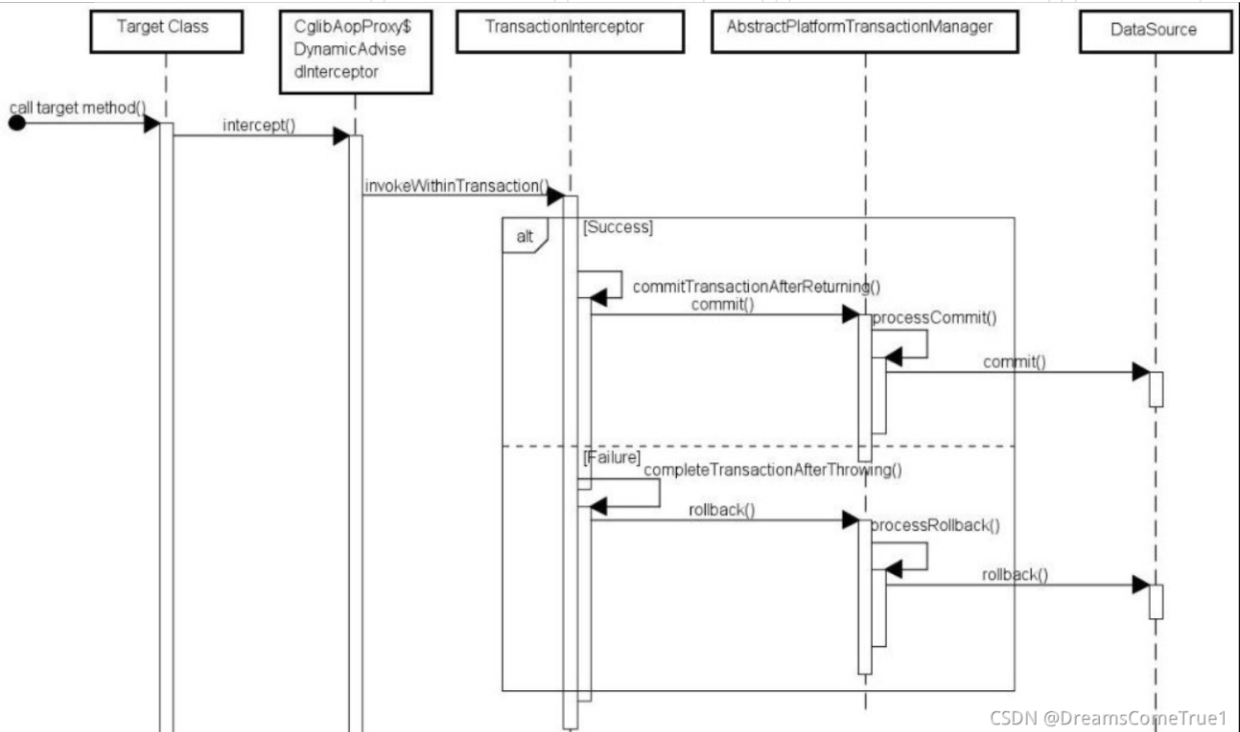
(4)衍生了springmvc框架。

(5)通过AOP实现了事务的统一管理。

(6)spring的生态十分强大，在遇到问题时可以快速的响应。

### 9.@Transactional失效的场景

(1)如果Transactional注解应用在非public 修饰的方法上，Transactional将会失效。



之所以会失效是因为在Spring AOP 代理时，如上图所示 TransactionInterceptor （事务拦截器）在目标方法执行前后进行拦截，DynamicAdvisedInterceptor（CglibAopProxy 的内部类）的intercept方法或JdkDynamicAopProxy 的 invoke 方法会间接调用 AbstractFallbackTransactionAttributeSource的 computeTransactionAttribute 方法，获取Transactional 注解的事务配置信息。此方法会检查目标方法的修饰符是否为 public，不是 public则不会获取@Transactional 的属性配置信息。

(2)Transactional注解属性propagation设置错误失效。比如设置成了以下三种：

TransactionDefinition.PROPAGATION\_SUPPORTS：如果当前存在事务，则加入该事务；如果当前没有事务，则以非事务的方式继续运行。

TransactionDefinition.PROPAGATION\_NOT\_SUPPORTED：以非事务方式运行，如果当前存在事务，则把当前事务挂起。

TransactionDefinition.PROPAGATION\_NEVER：以非事务方式运行，如果当前存在事务，则抛出异常。

(3)Transactional注解属性rollbackFor设置错误失效。rollbackFor可以指定能够触发事务回滚的异常类型。Spring默认抛出了未检查unchecked异常（继承自 RuntimeException的异常）或者 Error才回滚事务；其他异常不会触发回滚事务。如果在事务中抛出其他类型的异常，但却期望 Spring 能够回滚事务，就需要指定 rollbackFor属性。若在目标方法中抛出的异常是 rollbackFor 指定的异常的子类，事务同样会回滚。

(4)同一个类中方法调用，导致@Transactional失效。比如有一个类Test，它的一个方法A，A再调用本类的方法B（不论方法B是用public还是private修饰），但方法A没有声明注解事务，而B方法有。则外部调用方法A之后，方法B的事务是不会起作用的。

这是由于使用Spring AOP代理造成的，因为只有当事务方法被当前类以外的代码调用时，才会由Spring生成的代理对象来管理。

(5)异常的catch导致@Transactional失效。也就是被@Transactional作用的方法，其内部业务逻辑使用了try-catch，一旦逻辑出错，且在catch里没有throw的话，事务将不会回滚。

(6)数据库引擎不支持事务。这种情况几乎没有，事务能否生效数据库引擎是否支持事务是关键。常用的MySQL数据库默认使用支持事务的innodb引擎。一旦数据库引擎切换成不支持事务的myisam，那事务就从根本上失效了。

详细介绍@Transactional：

@Transactional是Spring中的一个注解，属于声明式事务的一种，它基于AOP面向切面，将具体业务与事务处理部分解耦，代码侵入性很低。@Transactional可以修饰接口、类、方法，当@Transactional修饰类，表示该类的所有public方法都配置相同的事务属性信息；当类配置了@Transactional，方法也配置了@Transactional，则方法的事务会覆盖类的事务配置信息；不推荐修饰接口，因为一旦标注在Interface上并且配置了Spring AOP使用CGLib动态代理，将会导致@Transactional注解失效。@Transactional属性：



isolation ：事务的隔离级别，默认值为 Isolation.DEFAULT。

Isolation.DEFAULT：使用底层数据库默认的隔离级别。

Isolation.READ\_UNCOMMITTED

Isolation.READ\_COMMITTED

Isolation.REPEATABLE\_READ

Isolation.SERIALIZABLE

timeout ：事务的超时时间，默认值为 -1。如果超过该时间限制但事务还没有完成，则自动回滚事务。

readOnly ：指定事务是否为只读事务，默认值为 false；为了忽略那些不需要事务的方法，比如读取数据，可以设置 read-only 为 true。

rollbackFor ：用于指定能够触发事务回滚的异常类型，可以指定多个异常类型。

noRollbackFor：抛出指定的异常类型，不回滚事务，也可以指定多个异常类型。

## SpringMVC

### 1.springmvc的理解

(1)springmvc是属于spring生态中的一个模块，它是在Servlet基础上构建并使用MVC模式设计的一个Web框架，目的是简化传统Servlet+JSP模式下的Web开发方式。

(2)浏览器的请求首先会经过核心控制器DispatcherServlet，DispatcherServlet会把请求转发给HandlerMapping，HandlerMapping负责解析请求，根据请求信息和配置信息找到对应的Controller类。如果有配置拦截器，则会按照顺序执行拦截器里面的preHandle方法。Controller处理完成业务逻辑之后会返回ModeAndView。然后DispatcherServlet会寻找一个或多个视图解析器ViewResolver，将ModeAndView指定的视图和数据一起展示到客户端。

### 2.过滤器和拦截器的区别

(1)运行的顺序不同，过滤器是Servlet容器接收到请求之后，但是在Servlet被调用之前运行的。拦截器则是在Servlet被调用后，响应被发送到客户端之前运行的。

(2)配置方式不同。过滤器是在web.xml里面进行配置，或者使用@WebFilter注解配置。拦截器是在springmvc配置文件中配置，或者使用注解的方式进行配置。

(3)Filter依赖Servlet容器，而Interceptor不依赖Servlet容器。

(4) Filter只能对request和response进行操作。Interceptor可以对request、response、handler、modelAndView、exception进行操作。

## SpringBoot

### 1.springboot可以同时处理多少请求

在springboot中可以通过配置最大连接数和等待队列的长度来设置其可以处理的同时请求数，其同时请求数就是最大连接数+等待队列的长度。

# tomcat最大连接数，默认值为8192

server.tomcat.max-connections=8192

# tomcat等待队列的长度，默认值为100

server.tomcat.accept-count=100

补充：

# 最少线程数，默认值为10

server.tomcat.threads.min-spare=10

# 最大线程数，默认值为200

server.tomcat.threads.max=200

qps：指的是对于服务器的每秒请求数。

### 2.浅谈对于springboot的理解，它有哪些优点

SpringBoot是快速开发Spring应用的一个脚手架、其设计目的是用来简化spring应用的初始搭建以及开发过程。

(1)springBoot提供了很多内置的starter结合自动配置，对主流框架无配置集成、开箱即用。

(2)springBoot简化了开发，采用Javaconfig的方式可以使用零xml的方式进行开发;

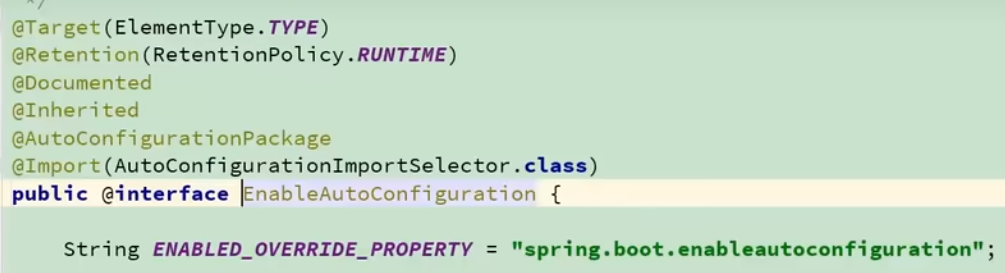
(3)springBoot内置web容器无需依赖外部web服务器，省略了web.xml，直接运行jar文件就可以启动web应用;

(4)springBoot帮我管理了常用的第三方依赖的版本，减少出现版本冲突的问题;

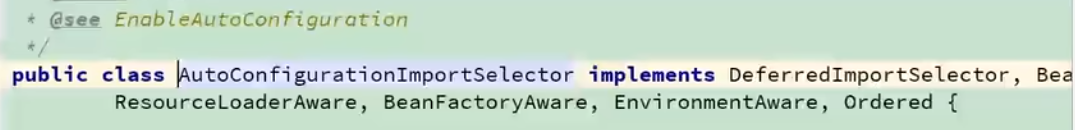
(5)springBoot自带监控功能，可以监控应用程序的运行状况，或者内存、线程池、Http请求统计等，同时还提供了优雅关闭应用程序等功能。

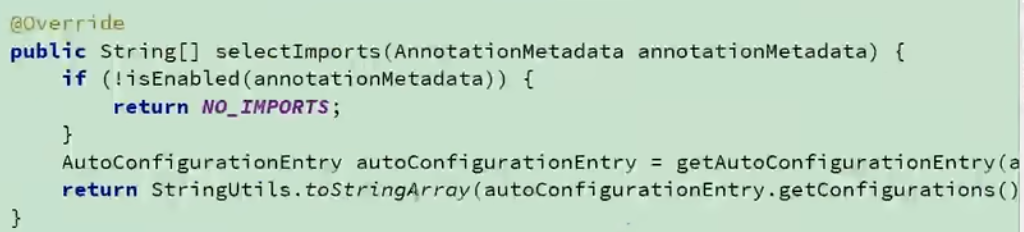
### 3.springboot的自动装配原理

自动装配就是自动把第三方组件的Bean装载到ioc容器中，不需要再去写Bean相关的一些配置。在springboot应用中只需要在启动类上加上@SpringBootApplication注解就可以实现自动装配。@SpringBootApplication是一个复合注解，其包含@SpringBootConfiguration、@ComponentScan、@EnableAutoConfiguration三个注解，真正实现自动装配的是@EnableAutoConfiguration。@EnableAutoConfiguration又包含@AutoConfigurationPackage、@Import，主要是@Import。

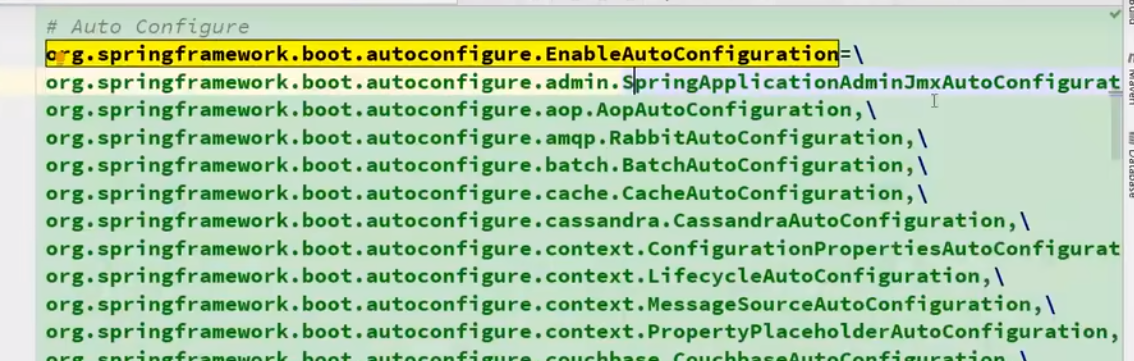


从图中可以看到@Import有个属性value的值为AutoConfigurationImportSelector.class，这个类实现了DeferredImportSelector接口，而DeferredImportSelector又继承了ImportSelector接口，相当于AutoConfigurationImportSelector类实现了ImportSelector接口，则它必须重写selectImports(AnnotationMetadata data)方法，该方法返回一个String数组，数组中的所有类都将加载到ioc容器中。注意：DeferredImportSelector会使自动配置的Bean的实例化放到其他Bean实例化的后面以方便扩展和覆盖。





自动装配主要依靠三个核心的关键技术。第一是引入Starter，启动依赖组件时，组件必须包含一个@Configuration配置类，在这个配置类中需要通过@Bean注解去声明需要装配到ioc容器里面的Bean对象。第二是这个配置类是放在第三方jar包里面，然后通过springboot中约定优于配置的理念去把这个配置类的全路径放在classpath:/META-INF/spring.factories文件里面，这样springboot就可以知道第三方jar包里面这个配置类的位置。这个步骤主要是用到了spring里面的SpringFactoriesLoader来完成的。第三是springboot拿到所有第三方jar包里面声明的配置类以后再通过spring提供的ImportSelector这样一个接口来实现对这些配置类的动态加载，从而完成自动装配。



补充：在spring.factories文件里有很多键值对，而自动配置会过滤出键是

org.springframework.boot.autoconfigure.EnableAutoConfiguration的键值对，如上图所示。

而且在拿到这些配置类后，因为这些配置类是被Condition系列的条件注解(比如

@ConditionOnMissBean)修饰的，因此会有一些配置类不满足被装载到ioc容器的条件，

从而该配置类不会有Bean实例。

### 4.springboot中的starter组件

starter组件是springboot的四大核心功能特性之一，除此之外还有自动装配、Actuator监控等特性。这些特性都是方便开发者在开发springboot应用时，只需要关心业务逻辑，减少对配置和外部环境的依赖。starter组件的重要作用如下：

(1)starter组件以功能为纬度来维护对应jar包的版本依赖，以解决jar包版本冲突导致的问题。

(2)starter组件会把对应功能的所有jar包依赖都导进来。

(3)starter组件内部集成了自动装配的机制，也就是应用启动后，组件会自动集成到spring生态里。

(4)依赖starter组件后，这些组件对应的功能所需要维护的外部化配置会自动集成到springboot中，只需要在springboot的配置文件中进行维护配置就好了。

补充：官方已经维护了很多starter组件，比如redis、mongoDB等。对于一些第三方组件，则是第三方来维护。官方维护的starter组件的命名方式为：spring-boot-starter-xxx，而第三方维护的starter组件的命名方式为：xxx-spring-boot-starter。

# 数据库

## 通用

### 1.数据库连接池的作用

(1)数据库连接池是一种池化技术，旨在实现资源的复用，避免资源的重复创建和销毁带来的开销。实际应用程序每一次向数据库发起CRUD操作时都需要数据库的连接。

(2)其核心思想是应用程序在启动时，去提前初始化一部分的连接，保存在连接池里，当应用程序需要用到连接时，直接从连接池里取，用完后再将连接放到连接池中，从而避免了数据库连接的多次创建与销毁。

(3)数据库连接池的关键参数

初始化连接数(最小连接数)：表示项目启动时，初始化多少个连接保存在连接池中。该数量

是连接池一直保持的数据连接数。

最大连接数：表示连接池能申请的最大连接数，如果连接数不够，后续要获取连接的线程就

会阻塞。

最大空闲连接数：表示没有请求的时候，连接池中要保留的最大空闲连接数。

最小空闲连接数：表示当空闲连接数小于这个值的时候，连接池需要再创建连接来补充至这

个值。

最大等待时间：表示当连接池中的连接用完了以后，新的请求要等待的时间，超过这个时间

就会提示超时异常。

### 2.数据量多大需要分库分表

(1)单表数据量。如果单个表的数据量非常大，例如超过了百万级别，就需要开始考虑分表。在阿里巴巴开发手册上是建议单表记录超过500万行或者单表容量超过2GB才推荐分库分表。

(2)数据库性能。当单个数据库的性能无法满足业务需求时，就需要考虑分库。

(3)数据库的访问频率。如果某些表的访问频率非常高，单个数据库节点又无法满足高并发需求，就需要考虑把这些表分到不同的数据库节点上来提高整个数据库的IO性能。

(4)业务拆分。当系统的业务逻辑越来越复杂，不同业务之间的数据耦合度越来越低就需要考虑对整个系统进行拆分，以方便管理和扩展。

### 3.为什么数据库字段建议设置为NOT NULL

(1)数据完整性。通过设置数据库字段为NOT NULL可以确保数据库中的数据是完整的。比如一个用户表中的用户名不是NOT NULL，就可能出现用户名为NULL的情况，这样的数据是不完整。

(2)查询性能。把字段设置为NOT NULL还可以提高查询的效率，数据库在查询的时候不需要额外去处理空值的情况，可以更快定位符合条件的数据行。

(3)开发的友好性。在开发时，如果字段是NOT NULL，就可以在程序中减少对空值的额外处理逻辑，提高代码的整洁性。

(4)数据一致性约束。通过把字段设置为NOT NULL，可以在数据库层面强制实施数据一致性约束，可以避免程序层面忽略或者错误的处理空值情况，减少了数据质量问题的一个可能性。

## Mysql

### 1.索引的优点与缺点

索引是一种能够帮数据库高效的从磁盘去检索数据的一种数据结构。在mysql的InnoDB引擎里采用的是B+树的结构来去实现索引和数据的存储。

优点：通过B+树的结构存储数据，可以大大减少数据检索时的磁盘IO次数，从而提升数据查询的性能；B+树索引在进行范围查找时，只需要找到起始节点，然后基于叶子节点的链表结构往下读取即可，查询效率较高；通过唯一索引约束，可以保证数据表中每一行数据的唯一性。

缺点：数据的增加、修改、删除，需要涉及到索引的维护，当数据量较大的情况下，索引的维护会带来较大的性能开销；一个表中允许存在一个聚簇索引和多个非聚簇索引，但是索引数量过多会造成索引维护的成本过高；创建索引时，需要考虑索引字段值的分散性，如果该字段的重复数据过多，创建索引反而会带来性能的下降。

### 2.什么情况下不需要建索引

(1)数据量太小。这种情况即使没有索引，查询速度也比较快，加了索引反而会增加维护的成本和查询的时间。

(2)数据离散度不高的列。比如说性别、年龄这种，创建索引反而会降低检索效率，从底层来说相当于增加了B+数的扫描范围进而增加了查询时间。

(3)存在函数操作的情况。如果查询条件包含函数操作，这种可能就不会走索引，建索引的意义不大。

(4)频繁变更的表也不适合建索引。比如经常需要更新、删除、插入记录。

### 3.mysql为什么使用B+数作为索引结构

常规的数据库存储引擎一般都是采用B树或者B+树来实现索引的存储。因为B树是一种多路平衡树，用这种存储结构来存储大量数据的情况下，它的整体高度相比二叉树来说矮很多。而对于数据库来说，所有数据必然是存储在磁盘上的，而磁盘的IO效率实际上是很低的，特别是在随机磁盘IO的一个情况下效率更低。所以树的高度能够决定磁盘IO 的一个次数，磁盘IO次数越少，对性能的提升就越大，这也是为什么采用B树作为索引存储结构的原因。但是在mysql的InnoDB存储引擎里面，它采用的是一种增强的B树结构，也就是B+树来作为索引和数据的一个存储结构。相比较于B树的结构，B+树做了几个方面的优化。

(1)B+树的所有数据都存储在叶子节点，非叶子节点只存储索引。

(2)叶子节点中的数据使用双向链表的方式进行关联。

所以使用B+树来实现索引的一个存储结构的优点如下：

(1)B+树非叶子节点不存储数据，所以每一层能够存储的索引数量会增加，意味着B+树在层高相同的情况下存储的数据量要比B树要多，使得磁盘IO次数更少。

(2)在mysql中，范围查询是一个比较常用的操作，而B+树的所有存储在叶子节点的数据使用了双向链表来关联，所以在查询的时候只需要查两个节点进行遍历就行，但是B树需要获取所有节点，所以B+树在范围查询上效率更高。

(3)在数据检索方面，由于所有的数据都存储在叶子节点，所以B+树的IO次数会更加稳定一些。

(4)因为叶子节点存储所有数据，所以B+树的全局扫描能力更强一些，因为它只需要扫描叶子节点，但是B树需要扫描整个树。

补充：基于B+树的结构，如果采用自增的整型数据作为主键，还可以避免增加数据的时候带来的叶子节点分裂导致的大量运算的问题。

### 4.limit分页规则

将查询结果的一部分显示出来，通常用于分页查询。

语法：limit startIndex,length。startIndex是起始下标，length是长度。

起始下标可以省略它，默认是0。如：limit 5 是取前5条。

limit是在order by 之后执行。

取出工资排名在[3-5]名的员工：

select ename,sal from emp order by sal desc limit 2,3;

分页的实现：



注意：limit的另一种写法，limit m offset n，这里m是长度，n是起始下标，正好和上面参数的含义其顺序相反。所以第pageNo页的写法是：

第pageNo页：limit pageSize offset (pageNo-1)\*pageSize

### 5.mysql语句的执行顺序

from->join->on->where->group by->having->select->distinct->order by->limit

## Redis

### 1.Redis哨兵机制和集群的区别

Redis哨兵机制有两种，一种是主从复制，一种是Redis Cluster。

Redis哨兵集群和Redis Cluster的区别如下：

(1) Redis哨兵集群是基于主从复制来实现的，所以它可以实现读写分离，分担Redis读操作的压力。而Redis Cluster集群里面的Slaver节点只是实现冷备的一个机制，它只有在Master宕机之后才会工作。

(2) Redis哨兵集群无法在线扩容，所以它的并发压力受限于单个服务器资源的一个配置。而Redis Cluster提供了基于Slot槽的一个数据分片的机制，它可以实现在线扩容去提升读写的一个性能。

(3)从集群的架构来说，Redis哨兵集群是一主多从，而Redis Cluster则是多主多从。

### 2.redis和mysql如何保证数据一致性

redis可以用来实现应用和数据库之间的一个读操作的缓存层，主要目的是减少数据库的IO次数，它可以提高数据的IO性能。当应用程序需要去读取某个数据的时候，首先会尝试Redis里面去加载，如果命中了就直接返回，如果没有命中就直接从数据库里面查询，查询到数据之后再把数据缓存到Redis里面。这样就会产生一个数据一致性的问题，就是一份数据同时保存在数据库和Redis中，当数据变化时，需要同时去更新Redis和mysql，但是更新操作是有先后顺序的，并且Redis并不像mysql中的多表事务操作满足ACID特性。那么能够选择的方法有以下几种：

(1)先更新数据库再更新缓存。但如果更新缓存失败，那么mysql和Redis中的数据还是不一致的。

(2)先删除缓存再更新数据库。理想情况下是下次应用访问Redis的时候发现Redis中没有此数据，那么就会去数据库加载并保存到Redis里面。也就是说数据理论上是一致的，但是极端情况下，由于删除Redis缓存和更新数据库不是一个原子操作，所以这个过程中如果有线程来访问还是会出现数据不一致的情况。

(3)采用最终一致性。它可以保证在极端情况下的数据一致性，但是会有短期数据不一致的情况。比如：基于RocketMQ的可靠性消息通信来实现数据的最终一致性。还可以直接通过Canal组件监控mysql中binlog的日志，把更新后的数据同步到Redis里面。如果业务场景不能接受数据的短期不一致性，则此种方案就不合适。

### 3.Redis是否存在线程安全问题

(1)从Redis的服务端层面，Redis Server本身是一个线程安全的K-V数据库，也就是说在Redis Server端去执行指令的时候，不需要任何的同步机制，它不会存在任何线程安全的一个问题，虽然Redis 6.0里面增加了多线程的模型，但是增加了多线程模型呢只是去处理网络的IO事件，对于指令的执行过程，仍然是采用主线程来处理的，所以它并不会存在多个线程同时去执行操作指令的一个情况，为什么Redis没有采用多线程来执行指令，我认为呢，有几个方面的原因，首先呢Redis本身可能出现性能瓶颈的点，无非就是IO,CPU和内存嘛，但是CPU又不是Redis本身的一个性能瓶颈点，所以没有必要去使用多线程来执行指令，其次啊如果使用多线程，意味着对于Redis里面的所有指令操作，都必须要去考虑到线程安全性的问题，也就是说需要去通知加锁来解决，那么这种方式呢会带来性能上的影响，反而会导致问题更大。

(2)从Redis 的客户端层面来说，虽然Redis Server中的指令操作是原子的，但是如果有多个Redis 客户端同时执行多个指令的情况下呢，就无法去保证原子性，假设两个redis client在同时去获取Redis Server的key1,同时去进行修改和写入，因为多线程环境下的原子性呢无法被保证以及多线程的情况下共享资源访问的一个竞争问题，使得数据的安全性无法得到保障。当然啊，对于线程安全性的问题呢，解决的方法有很多，比如说，尽可能的使用Redis里面的原子指令，或者对于多个客户端的资源访问去加锁，再或者我们可以通过lua脚本，来实现多个指令的执行操作，从而去满足它的一个原子性。

### 4.Redis主从复制的原理

Redis的主从复制是指在Redis集群里面Master节点和Slave节点的数据同步的一种机制。简单说就是把一台Redis服务器的数据复制到其他的Redis服务器里面，其中负责复制数据来源的节点称为Master，被动接收数据同步的节点称为Slave。Redis的主从复制包括全量复制和增量复制。

(1)全量复制一般是发生在Slave节点初始化阶段，需要把Master上所有的数据全部copy一份。流程是Slave (从)节点会主动向Master (主)节点发起一个同步请求，即一个SYNC命令，主节点收到请求以后会生成一份当前数据的快照发送给从节点。从节点收到数据后会丢弃旧的数据并加载新数据，之后完成全量复制。

(2)增量复制是发生在每一次Master节点发生数据变更的一个场景里面，会把变化的增量数据同步给从节点。原理是Master和Slave都会维护一个复制偏移量offset，用来表示Master向Slave传递的字节数量，每一次进行增量数据的传递，Master和Slave维护的offset都会增加对应的字节数量，Redis只需要根据offset就可以实现增量的数据的同步。

在主从复制过程中，Redis并没有采用数据的强一致性，因此会存在一个数据同步的延迟导致数据的一个不一致的问题。