面试题

# 一、Java基础

## 1、（数据类型？）Java都有哪些数据类型？基本数据类型有哪些？分别占多少字节？多少位？引用数据类型又有哪些？

基本数据类型：byte（1）、short（2）、int（4）、long（8）、float（4）、double（8）、char（2）、boolean（1），一个字节8个比特位

引用数据类型：接口、数组、对象

## 2、（Java特性？）Java语言的几大特性是什么？分别怎么理解？

(封装、继承、多态的好处)

**封装**：封装隐藏对象的属性和实现细节，仅对外提供公共访问方式，将变化隔离，便于使用，提高复用性和安全性。封装把一个对象的属性私有化，同时提供一些可以被外界访问的属性的方法，如果属性不想被外界访问，我们大可不必提供方法给外界访问。但是如果一个类没有提供给外界访问的方法，那么这个类也没有什么意义了。

封装隐藏对象的属性和实现细节，仅对外提供公共访问方式，将变化隔离，便于使用，提高复用性和安全性。

**继承**：继承是使用已存在的类的定义作为基础建立新类的技术，新类的定义可以增加新的数据或新的功能，也可以用父类的功能，但不能选择性地继承父类。通过使用继承我们能够非常方便地复用以前的代码。

1.子类拥有父类非private 的属性和方法。

2子类可以拥有自己属性和方法，即子类可以对父类进行扩展。

3.子类可以用自己的方式实现父类的方法。(以后介绍)。

**多态**：所谓多态就是指程序中定义的引用变量所指向的具体类型和通过该引用变量发出的方法调用在编程时并不确定，而是在程序运行期间才确定，即一个引用变量到底会指向哪个类的实例对象，该引用变量发出的方法调用到底是哪个类中实现的方法，必须在由程序运行期间才能决定。（适配器模式）

多态主要通过三种方式来实现:

①通过子类对父类的重写来实现

②通过在一个类中对方法的重载来实现

③通过将子类对象作为父类对象使用来实现

多态的前提:①要有继承；②要有方法的重写；③父类的引用指向子类的对象

## 3、（权限修饰符？）Java的权限修饰符有哪些？都能加在哪些地方？分别代表什么意义？

定义：Java中，可以使用访问修饰符来保护对类、变量、方法和构造方法的访问。Java 支持 4 种不同的访问权限。

分类：

private : 在同一类内可见。使用对象：变量、方法。 注意：不能修饰类（外部类）

default (即缺省，什么也不写，不使用任何关键字）: 在同一包内可见，不使用任何修饰符。

protected : 对同一包内的类和所有子类可见。使用对象：变量、方法。 注意：不能修饰类（外部类）。

public : 对所有类可见。使用对象：类、接口、变量、方法

## 4、（重写重载？）什么是重写？什么是重载？

重载： 发生在同一个类中，方法名必须相同，参数类型不同.个数不同.顺序不同， 方法返回值和访问修饰符可以不同，发生在编译时。

重写： 发生在父子类中，方法名.参数列表必须相同，返回值范围小于等于父类， 抛出的异常范围小于等于父类， 访问修饰符范围大于等于父类；如果父类方法访问修饰符为 private 则子类就不能重写该方法。

## 5、（final关键字？）final关键字能加在哪些地方？分别代表什么？

final：用于声明属性，方法和类

1.被final修饰的类不可以被继承

2.被final修饰的方法不可以被重写，JVM会尝试将其内联，以提高运行效率

3.被final修饰的变量不可以被改变。如果修饰引用，那么表示引用不可变，引用指向的内容可变。

4.被final修饰的常量，在编译阶段会存入常量池中。

回答出编译器对final域要遵守的两个重排序规则更好：

1.在构造函数内对一个final域的写入，与随后把这个被构造对象的引用赋值给一个引用变量,这两个操作之间不能重排序。

2.初次读一个包含final域的对象的引用，与随后初次读这个final域,这两个操作之间不能重排序。

## 6、（static关键字？）static关键字能加在哪些地方？分别代表什么？

1、修饰成员变量 2、修饰成员方法 3、静态代码块 4、修饰类【只能修饰内部类也就是静态内部类】5、静态导包

1、被static修饰的变量或者方法是独立于该类的任何对象，也就是说，这些变量和方法不属于任何一个实例对象，而是被类的实例对象所共享。

2、在该类被第一次加载的时候，就会去加载被static修饰的部分，而且只在类第一次使用时加载并进行初始化，注意这是第一次用就要初始化，后面根据需要是可以再次赋值的。

3、static变量值在类加载的时候分配空间，以后创建类对象的时候不会重新分配。赋值的话，是可以任意赋值的！

4、被static修饰的变量或者方法是优先于对象存在的，也就是说当一个类加载完毕之后，即便没有创建对象，也可以去访问。

## 7、（接口和抽象类？）接口中可以有哪些成员？抽象类呢？接口和抽象类又有什么区别？（注意JDK1.8接口中是可以出现非抽象方法的：default方法、静态方法）

一、接口中

1.成员变量：默认被public static final修饰。

2.成员方法：默认被public abstract修饰。

3.构造方法：接口中不能有构造方法

二、抽象类中

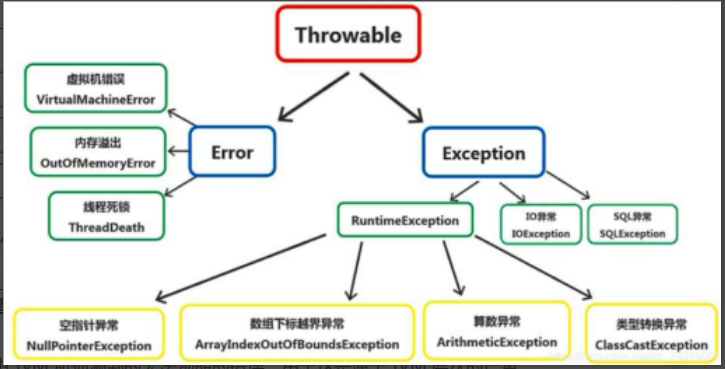
1.成员变量：可以是变量也可以是常量

2.成员方法：可以有抽象方法：限定子类必须完成某些动作，也可以有非抽象方法：提高代码复用性。

3.构造方法：有构造方法，但是不能实例化，其构造方法用于子类访问父类数据的初始化



## 8、（异常？）Java异常体系是什么?(编译)异常有什么区别？常见的运行时异常有哪些？



Throwable 是所有 Java 程序中错误处理的父类，有两种子类：Error 和 Exception。Error：表示由 JVM 所侦测到的无法预期的错误，由于这是属于 JVM 层次的严重错误，导致 JVM 无法继续执行，因此这是不可捕捉到的，无法采取任何恢复的操作，顶多只能显示错误信息。 Exception：表示可恢复的异常，这是可捕捉到的。

1.运行时异常：都是 RuntimeException 类及其子类异常，如 NullPointerException(空指针异常)、IndexOutOfBoundsException(下标越界异常)等， 这些异常是不检查异常，程序中可以选择捕获处理，也可以不处理。这些异常一般是由程序逻辑错误引起的，程序应该从逻辑角度尽可能避免这类异常的发生。运行时异常的特点是 Java 编译器不会检查它，也就是说，当程序中可能出现这类异常，即使没有用 try-catch 语句捕获它，也没有用 throws 子句声明抛出它，也会编译通过。

2.非运行时异常（编译异常）：是 RuntimeException 以外的异常，类型上都属于 Exception 类及其子类。从程序语法角度讲是必须进行处理的异常，如果不处理，程序就不能编译通过。 如 IOException、SQLException 等以及用户自定义的 Exception 异常，一般情况下不自定义检查异常。

常见的 RunTime 异常几种如下：

NullPointerException - 空指针引用异常

ClassCastException - 类型强制转换异常。

IllegalArgumentException - 传递非法参数异常。

ArithmeticException - 算术运算异常

ArrayStoreException - 向数组中存放与声明类型不兼容对象异常

IndexOutOfBoundsException - 下标越界异常

NegativeArraySizeException - 创建一个大小为负数的数组错误异常

NumberFormatException - 数字格式异常

SecurityException - 安全异常

UnsupportedOperationException - 不支持的操作异常

## 9、== 和 equals的异同？

== 的作用： 基本类型：比较的就是值是否相同 引用类型：比较的就是地址值是否相同

equals 的作用: 引用类型：默认情况下，比较的是地址值。 特：String、Integer、Date 这些类库中 equals 被重写，比较的是内容而不是地址！

面试题：请解释字符串比较之中 “ == ” 和 equals() 的区别？ 答： ==：比较的是两个字符串内存地址（堆内存）的数值是否相等，属于数值比较； equals()：比较的是两个字符串的内容，属于内容比较。

## 10、&与&&、|与||的区别？

&&运算符是短路与运算。逻辑与跟短路与的差别是非常巨大的，虽然二者都要求运算符左右两端的布尔值都是true 整个表达式的值才是 true。&&之所以称为短路运算是因为，如果&&左边的表达式的值是 false，右边的表达式会被直接短路掉，不会进行运算。很多时候我们可能都需要用&&而不是&，例如在验证用户登录时判定用户名不是 null 而且不是空字符串，应当写为 username != null &&!username.equals("")，二者的顺序不能交换，更不能用&运算符，因为第一个条件如果不成立，根本不能进行字符串的 equals 比较，否则会产生 NullPointerException 异常。注意：逻辑或运算符（|）和短路或运算符（||）的差别也是如此。

## 11、String可以修改本身吗？为什么？

String使用 final 关键字修饰字符数组来保存字符串，private final char value[] ，String 对象是不可变的，也就可以理解为常量，线程安全。

所以String是不可修改的，因为修改的只是String的引用指向的地址，字符串常量本身是不会变的。String对象只是一个引用，每次新建一个String对象s，编译器都会去堆中的字符串常量池中寻找有没有要匹配的字符串，有s就指向它；没有就新建一个字符串，然后s指向它，所以我们经常用String对象来存储HashMap等容器的键

## 12、StringBuffer和StringBuilder的区别是什么？

String 字符串常量

StringBuffer 字符串变量（线程安全）

StringBuilder 字符串变量（非线程安全）

String 中的 String 类使用 final 关键字修饰字符数组来保存字符串，private final char value[] ，String 对象是不可变的，也就可以理解为常量，线程安全。 AbstractStringBuilder 是 StringBuilder 与 StringBuffer 的公共父类，定义了 一些字符串的基本操作，如 expandCapacity、append、insert、indexOf 等公共方法。 StringBuffer 对方法加了同步锁或者对调用的方法加了同步锁，所以是线程安全的。 StringBuilder 并没有对方法进行加同步锁，所以是非线程安全的。

小结：

（1）如果要操作少量的数据用 String；

（2）多线程操作字符串缓冲区下操作大量数据用 StringBuffer；

（3）单线程操作字符串缓冲区下操作大量数据用 StringBuilder

## 13、valueOf和toString的区别？

共同点:在 JavaScript 中，toString()方法和valueOf()方法，在输出对象时会自动调用。

不同点：

(1)、二者并存的情况下，在数值运算中，优先调用了valueOf，字符串运算中，优先调用了toString。

(2)返回值类型的差别：

toString一定将所有内容转为字符串

valueOf取出对象内部的值，不进行类型转换

(3)用途的差别：

valueOf专用于算数计算和关系运算

toString专用于输出字符串

(4)共同的缺点

无法获取null和undefined的值。

## 14、（字符串拼接）大量字符串用 "+" 号进行拼接效率高吗？为什么？应该用什么替代？为什么？

字符串的+操作其本质是创建了 StringBuilder 对象进行 append 操作，然后将拼接后的 StringBuilder 对象用 toString 方法处理成 String 对象

通常我们在使用String的“+”进行字符串连接时会知道它其实会产生多个StringBuilder对象，从而使性能变差，所以我们通常比较推荐StringBuffer或StringBuilder的append方法进行字符串连接,这样就只会创建一个对象

## 15、创建一个类的实例都有哪些办法？

Java类的创建有四种方式：

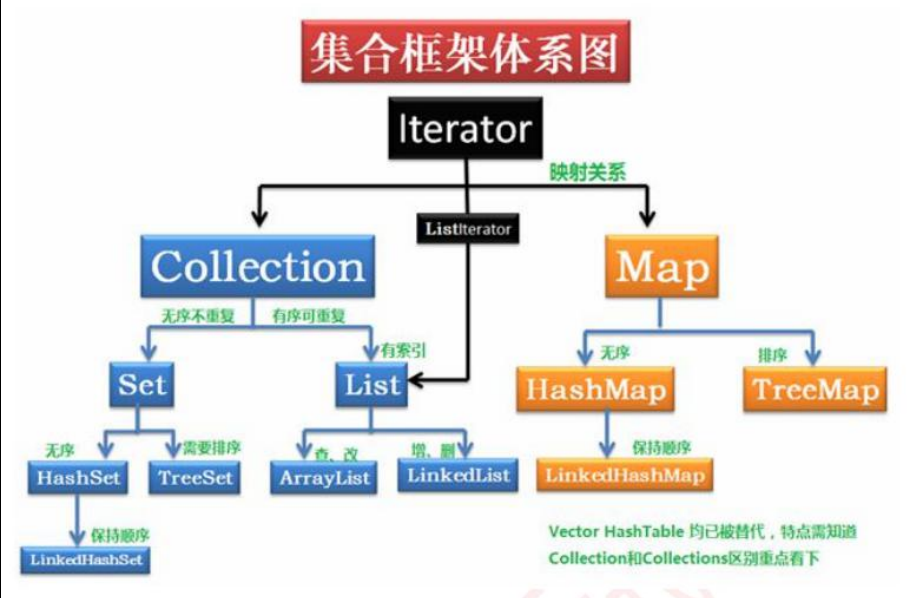
（1）用new 语句创建对象，这是最常用的创建对象方法。

（2）运用反射手段，调用Java.lang.Class或者java.lang.reflect.Constructor类的newInstance()实例方法。

（3）调用对象的clone()方法

（4）运用反序列化手段，调用java.io.ObjectInputStream对象的readObject()方法。

## 16、Java集合的体系是什么样的？



## 17、Set和List分别有哪些特点？Set去重的原理？

List 有序,可重复

ArrayList 优点: 底层数据结构是数组，查询快，增删慢。 效率高

缺点: 线程不安全

Vector 优点: 底层数据结构是数组，查询快，增删慢。

缺点: 线程安全，效率低, 已给舍弃了

LinkedList 优点: 底层数据结构是链表，查询慢，增删快。

缺点: 线程不安全，效率高

Set 无序,唯一

HashSet 底层数据结构是哈希表。(无序,唯一)

如何来保证元素唯一性? 依赖两个方法：hashCode()和 equals()

LinkedHashSet 底层数据结构是链表和哈希表。(FIFO 插入有序,唯一)

1. 由链表保证元素有序

2. 由哈希表保证元素唯一

TreeSet 底层数据结构是红黑树。(唯一，有序)

### 如何保证元素排序的呢?

自然排序 比较器排序

### 如何保证元素唯一性的呢?

根据比较的返回值是否是 0 来决定

1. Map 接口有四个实现类：

HashMap 基于 hash 表的 Map 接口实现，线程不安全，高效，支持 null 值和 null 键。

HashTable 线程安全，低效，不支持 null 值和 null 键；

LinkedHashMap 线程不安全，是 HashMap 的一个子类，保存了记录的插入顺序；

TreeMap 能够把它保存的记录根据键排序，默认是键值的升序排序，线程不安全。

## 18、ArrayList底层原理是什么？扩容原理？

ArrayList的特点：

　　1.ArrayList的底层数据结构是数组，所以查找遍历快，增删慢。

　　2.ArrayList可随着元素的增长而自动扩容，正常扩容的话，每次扩容到原来的1.5倍。

　　3.ArrayList的线程是不安全的。

ArrayList的扩容：

　　扩容可分为两种情况：

　　第一种情况，当ArrayList的容量为0时，此时添加元素的话，需要扩容，三种构造方法创建的ArrayList在扩容时略有不同：

　　　　1.无参构造，创建ArrayList后容量为0，添加第一个元素后，容量变为10，此后若需要扩容，则正常扩容。

　　　　2.传容量构造，当参数为0时，创建ArrayList后容量为0，添加第一个元素后，容量为1，此时ArrayList是满的，下次添加元素时需正常扩容。

　　　　3.传列表构造，当列表为空时，创建ArrayList后容量为0，添加第一个元素后，容量为1，此时ArrayList是满的，下次添加元素时需正常扩容。

　　第二种情况，当ArrayList的容量大于0，并且ArrayList是满的时，此时添加元素的话，进行正常扩容，每次扩容到原来的1.5倍。

## 19、LinkedList底层原理是什么？和ArrayList的区别是什么？

ArrayList 优点: 底层数据结构是数组，查询快，增删慢，效率高。

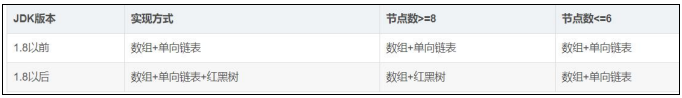
缺点: 线程不安全；

LinkedList 优点: 底层数据结构是链表，查询慢，增删快，效率高。

缺点: 线程不安全。

## 20、HashMap的底层原理是什么？map添加数据put的过程？扩容原理？

HashMap 在 JDK1.8 之前的实现方式 数组+链表, 但是在 JDK1.8 后对 HashMap 进行了底层优化,改为了由数组+链表或者数组+红黑树实现,主要的目的是提高查找效率



1. JDK1.8之前哈希表的结构是数组+链表的结构。先创建一个默认长度为16，默认加载因子为0.75的名为table的数组，当存入元素时，根据元素哈希值及数组长度计算出存入的位置，（经过两次hash运算，第一次算hashcode，第二次是用高16位与原始hashcode进行抑或运算，降低哈希冲突的概率），然后判断当前位置是否为null，如果为null则直接存入，如果不为null则调用equals方法比较属性值，如果一样则不存，不一样就存入形成链表结构，老元素挂在新元素下面。

JDK1.8及之后哈希表结构是数组+链表/红黑树的结构（且扩容由头插改为了尾插，因为多线程环境下头插的话transfer方法有问题，会造成链表成环，造成死锁，所以改为尾插），当一条链表的长度超过8，底层就会判断当前数组长度是否大于64并判断是否为空，如果不大于64，则先将数组扩容，如果大于64，长度超过8的链表就会自动转换为红黑树结构，小于6之后又会退化为链表。

2.当 new HashMap()时：底层没有创建数组，首次调用 put()方法示时，底层创建长度为16的数组，jdk8 底层的数组是：Node[],而非 Entry[]，用数组容量大小乘以加载因子得到一个值，一旦数组中存储的元素个数超过该值就会调用rehash 方法将数组容量增加到原来的两倍，专业术语叫做扩容，默认的负载因子大小为 0.75，数组大小为 16。也就是说，默认情况下，那么当 HashMap 中元素个数超过 16\*0.75=12 的时候，就把数组的大小扩展为 2\*16=32，即扩大一倍。在做扩容的时候会生成一个新的数组，原来的所有数据需要重新计算哈希码值重新分配到新的数组，所以扩容的操作非常消耗性能.

3.在我们 Java 中任何对象都有 hashcode，hash 算法就是通过 hashcode 与自己进行向右位移 16位（高16位） 的异或运算（第二次哈希）。这样做是为了计算出来的 hash 值足够随机，足够分散，还有产生的数组下标足够随机，

map.put(k,v)实现原理

（1）首先将 k,v 封装到 Node 对象当中（节点）。

（2）先调用 k 的 hashCode()方法得出哈希值，并通过哈希算法转换成数组的下标。

（3）下标位置上如果没有任何元素，就把 Node 添加到这个位置上。如果说下标对应的位置上有链表。此时，就会拿着 k 和链表上每个节点的 k 进行 equal。如果所有的 equals 方法返回都是 false，那么这个新的节点将被添加到链表的末尾。如其中有一个 equals 返回了 true，那么这个节点的 value 将会被覆盖。

map.get(k)实现原理

（1）、先调用 k 的 hashCode()方法得出哈希值，并通过哈希算法转换成数组的下标。

（2）、在通过数组下标快速定位到某个位置上。重点理解如果这个位置上什么都没有，则返回 null。如果这个位置上有单向链表，那么它就会拿着参数 K 和单向链表上的每一个节点 的 K 进行 equals，如果所有 equals 方法都返回 false，则 get 方法返回 null。如果其中一个节点的 K 和参数 K 进行 equals 返回 true，那么此时该节点的 value 就是我们要找的 value 了，get 方法最终返回这个要找的 value。

4. Hash 冲突：不同的对象算出来的数组下标是相同的这样就会产生 hash 冲突，当单线链表达到一定长度 后效率会非常低。

## 21、concurrentHashMap原理是什么？

ConcurrentHashMap 是 JDK1.7 使用了锁分段技术来保证线程安全的。JDK1.8ConcurrentHashMap 取消了 Segment 分段锁，采用 CAS 和 synchronized 来保证并发安全。数据结构跟 HashMap1.8 的结构类似，数组+链表/红黑二叉树。 synchronized 只锁定当前链表或红黑二叉树的首节点，这样只要 hash 不冲突，就不会产生并发，效率又提升 N 倍。

## 22、JDK8对于HashMap做了哪些优化？

HashMap 在 JDK1.8 之前的实现方式 数组+链表, 但是在 JDK1.8 后对 HashMap 进行了底层优化,改为了由 数组+链表或者数值+红黑树 实现,主要的目的是提高查找效率；且扩容由头插改为了尾插，因为多线程环境下头插的话transfer方法有问题，会造成链表成环，造成死锁，所以改为尾插。

## 23、什么是socket？什么是IO/NIO/BIO/AIO？区别是什么？

实际上socket是对TCP/IP协议的封装，它的出现只是使得程序员更方便地使用TCP/IP协议栈而已。socket本身并不是协议，它是应用层与TCP/IP协议族通信的中间软件抽象层，是一组调用接口（TCP/IP网络的API函数）

简答

BIO：Block IO 同步阻塞式 IO，就是我们平常使用的传统 IO，它的特点是模式简单使用方便，并发处理能力低。

NIO：Non IO 同步非阻塞 IO，是传统 IO 的升级，客户端和服务器端通过 Channel（通道）通讯，实现了多路复用。

AIO：Asynchronous IO 是 NIO 的升级，也叫 NIO2，实现了异步非堵塞 IO ，异步 IO 的操作基于事件和回调机制。

详细回答

BIO (Blocking I/O): 同步阻塞I/O模式，数据的读取写入必须阻塞在一个线程内等待其完成。在活动 连接数不是特别高（小于单机1000）的情况下，这种模型是比较不错的，可以让每一个连接专注于自 己的 I/O 并且编程模型简单，也不用过多考虑系统的过载、限流等问题。线程池本身就是一个天然的漏斗，可以缓冲一些系统处理不了的连接或请求。但是，当面对十万甚至百万级连接的时候，传统的 BIO 模型是无能为力的。因此，我们需要一种更高效的 I/O 处理模型来应对更高的并发量。

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

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

## 24、什么是反射？可以用来干嘛？列举一下反射应用场景？什么是暴力反射？

在 Java 中的反射机制是指在运行状态中，对于任意一个类都能够知道这个类所有的属性和方法；并且对于任意一个对象，都能够调用它的任意一个方法；这种动态获取信息以及动态调用对象方法的功能成为 Java 语言的反射机制。

获取 Class 对象的 3 种方法 ：

调用某个对象的 getClass()方法

Person p=new Person(); Class clazz=p.getClass();

调用某个类的 class 属性来获取该类对应的 Class 对象

Class clazz=Person.class;

使用 Class 类中的 forName()静态方法(最安全/性能最好)

Class clazz=Class.forName("类的全路径");

在我们平时的项目开发过程中，基本上很少会直接使用到反射机制，但这不能说明反射机制没有用，实际上有很多设计、开发都与反射机制有关，例如模块化的开发，通过反射去调用对应的字节码；动态代理设计模式也采用了反射机制，还有我们日常使用的 Spring等框架也大量使用到了反射机制。

举例：①我们在使用JDBC连接数据库时使用Class.forName()通过反射加载数据库的驱动程序；②Spring框架也用到很多反射机制，最经典的就是xml的配置模式。Spring 通过 XML 配置模式装载 Bean 的过程：1)将程序内所有 XML 或 Properties 配置文件加载入内存中; 2)Java类里面解析xml或properties里面的内容，得到对应实体类的字节码字符串以及相关的属性信息; 3)使用反射机制，根据这个字符串获得某个类的Class实例; 4)动态配置实例的属性.

暴力反射：反射里的Constructor,Field,Method三个类都有一个getDeclaredXxx方法,可以不受权限控制的获取类的构造函数,字段,方法,如果想要私有构造函数创建对象,字段赋值,方法调用的话,会自动的访问类的isAccessable,默认的是false,所以,你想要访问类中的私有成员的时候,就要调用setAccessable()方法,将其改为true,这样,你就可以对类中的私有成员进行操作了

## 25、算法了解过吗？冒泡排序、选择排序、快排原理？

二分查找：

public class MyBinarySearchDemo {

public static void main(String[] args) {

int [] arr = {1,2,3,4,5,6,7,8,9,10};

int number = 11;

//1,我现在要干嘛? --- 二分查找

//2.我干这件事情需要什么? --- 数组 元素

//3,我干完了,要不要把结果返回调用者 --- 把索引返回给调用者

int index = binarySearchForIndex(arr,number);

System.out.println(index);

}

private static int binarySearchForIndex(int[] arr, int number) {

//1,定义查找的范围

int min = 0;

int max = arr.length - 1;

//2.循环查找 min <= max

while(min <= max){

//3.计算出中间位置 mid

int mid = (min + max) >> 1;

//mid指向的元素 > number

if(arr[mid] > number){

//表示要查找的元素在左边.

max = mid -1;

}else if(arr[mid] < number){

//mid指向的元素 < number

//表示要查找的元素在右边.

min = mid + 1;

}else{

//mid指向的元素 == number

return mid;

}

}

//如果min大于了max就表示元素不存在,返回-1.

return -1;

}

}

冒泡排序：

public class MyBubbleSortDemo2 {

public static void main(String[] args) {

int[] arr = {3, 5, 2, 1, 4};

//1 2 3 4 5

bubbleSort(arr);

}

private static void bubbleSort(int[] arr) {

//外层循环控制的是次数 比数组的长度少一次.

for (int i = 0; i < arr.length -1; i++) {

//内存循环就是实际循环比较的

//-1 是为了让数组不要越界

//-i 每一轮结束之后,我们就会少比一个数字.

for (int j = 0; j < arr.length - 1 - i; j++) {

if (arr[j] > arr[j + 1]) {

int temp = arr[j];

arr[j] = arr[j + 1];

arr[j + 1] = temp;

}

}

}

printArr(arr);

}

private static void printArr(int[] arr) {

for (int i = 0; i < arr.length; i++) {

System.out.print(arr[i] + " ");

}

System.out.println();

}

}

## 26、JDK1.8的新特性有哪些?(lamda表达式、stream流、函数式接口、接口中默认方法、方法引用等等)

## 27、IO流体系

按照流的流向分，可以分为输入流和输出流；

按照操作单元划分，可以划分为字节流和字符流；

按照流的角色划分为节点流和处理流。

JavaIO流共涉及40多个类，这些类看上去很杂乱，但实际上很有规则，而且彼此之间存在非常紧密的联系，

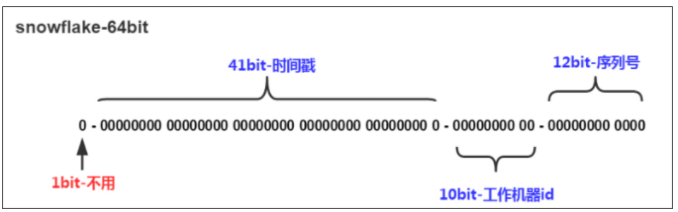
JavaIO流的40多个类都是从如下4个抽象类基类中派生出来的。

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

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

## 28、如何实现分布式主键自增？

Snowflake（雪花算法）是Twitter开源的分布式ID生成算法，结果是一个64位long型的ID。其核心思想是：首先一个是符号位，永远是0，使用41bit作为毫秒数，10bit作为机器的ID（5个bit是数据中心，5个bit的机器ID），12bit作为毫秒内的流水号（意味着每个节点在每毫秒可以产生 4096 个 ID），总体上依然是自增的。



# 二、JVM

## 1、JDK和JRE的区别是什么？

JVM

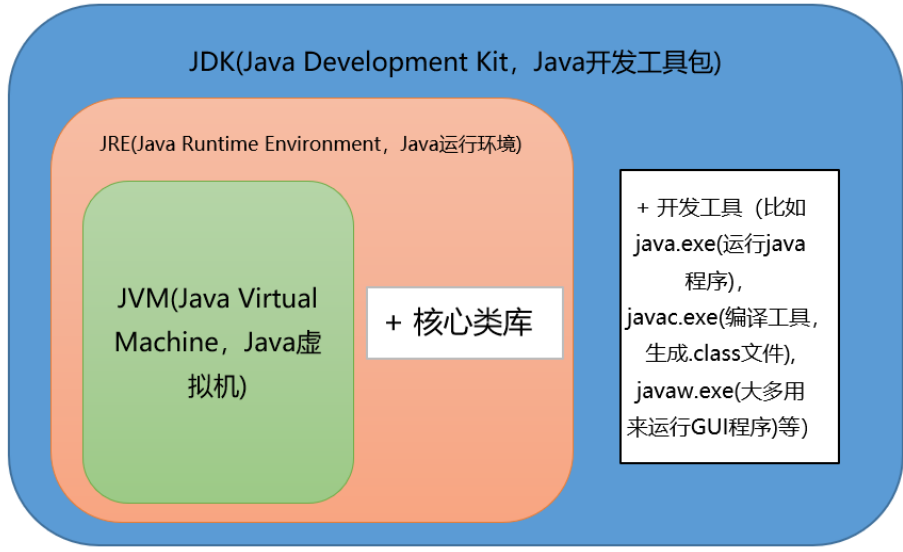
Java Virtual Machine是Java虚拟机，Java程序需要运行在虚拟机上，不同的平台有自己的虚拟机，因此Java语言可以实现跨平台。

JRE

Java Runtime Environment包括Java虚拟机和Java程序所需的核心类库等。核心类库主要是java.lang包：包含了运行Java程序必不可少的系统类，如基本数据类型、基本数学函数、字符串处理、线程、异常处理类等，系统默认加载这个包，如果想要运行一个开发好的Java程序，计算机中只需要安装JRE即可。

JDK = JRE（JVM+核心类库）+开发工具

Java Development Kit是提供给Java开发人员使用的，其中包含了Java的开发工具，也包括了JRE。所以安装了JDK，就无需再单独安装JRE了。其中的开发工具：编译工具(javac.exe)，打包工具(jar.exe)等



## 2、JVM的内存模型描述下？



1）程序计数器: 线程私有的(每个线程都有一个自己的程序计数器), 是一个指针. 代码运行, 执行命令. 而每个命令都是有行号的,会使用程序计数器来记录命令执行到多少行了.记录代码执行的位置

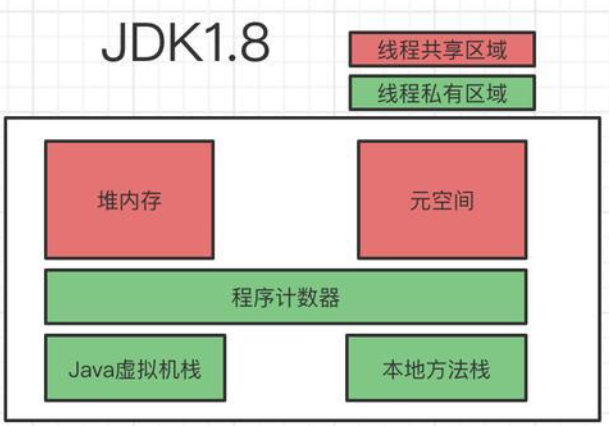
2）Java虚拟机栈: 线程私有的(每个线程都有一个自己的 Java 虚拟机栈). 一个方法运行, 就会给这个方法创建一个栈帧, 栈帧入栈执行代码, 执行完毕之后出栈(弹栈)消失，存引用变量，基本数据类型

3）本地方法栈: 线程私有的(每个线程都有一个自己的本地方法栈),和Java 虚拟机栈类似, Java虚拟机栈加载的是普通方法,本地方法栈加载的是native 修饰的方法. native:在 java中有用 native 修饰的,表示这个方法不是 java 原生的.

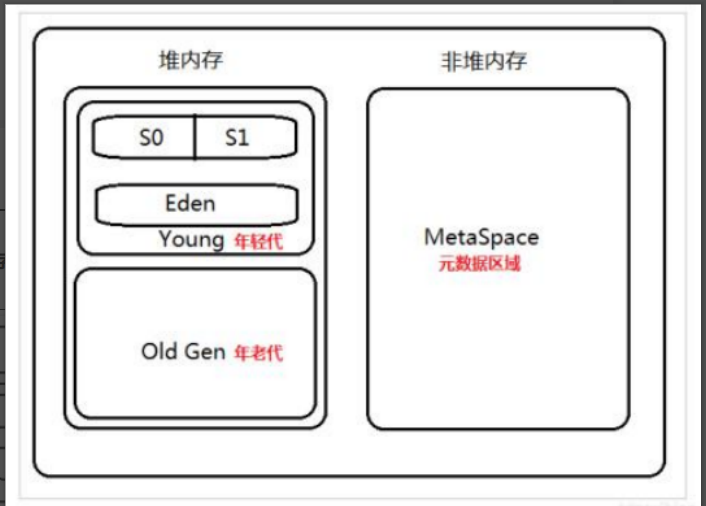
4）堆: 线程共享的(所有的线程共享一份). 存放对象的,new的对象都存储在这个区域.还有就是字符串常量池.

5）元空间: 存储.class信息, 类的信息,方法的定义,静态变量等.而字符串常量池放到堆里存储，JDK1.8 和 JDK1.7 的 jvm 内存最大的区别是, 在 1.8 中方法区是由元空间(元数据区)来实现，运行时常量池含（静态变量、字节码等信息）.

1.8 不存在方法区,将方法区的实现给去掉了.而是在本地内存中,新加入元数据区(元空间).

JDK1.8 堆内存结构（高薪常问）



Young 年轻区（代）: Eden+S0+S1, S0 和 S1 大小相等, 新创建的对象都在年轻代，新生代内存按照8:1:1 的比例分为一个eden区和两个Survivor

(survivor0,survivor1) 区，大部分对象在 Eden 区中生成。

Old Gen 年老区: 经过年轻代多次垃圾回收（15次MinorGC，可设置）存活下来的对象存在年老代中，内存比新生代也大很多(大概是 2 倍)。

永久代：用于存放静态文件，如 Java 类、方法等，Jdk1.7 和 Jdk1.8 的区别在于, 1.8 将永久代中的对象放到了元数据区, 不存永久代这一区域了。

## 3、JVM双亲委派加载机制，为什么JVM这么做？有违反双亲委派的例子吗？

通俗：

如果一个类加载器收到了类加载请求，它并不会自己先去加载，而是把这个请求委托给父类的加载器去执行，如果父类加载器还存在其父类加载器，则进一步向上委托，依次递归，请求最终将到达顶层的启动类加载器，如果父类加载器可以完成类加载任务，就成功返回，倘若父类加载器无法完成此加载任务，子加载器才会尝试自己去加载，这就是双亲委派模式。

双亲委派模型的工作过程如下：

（1）当前类加载器从自己已经加载的类中查询此类是否已经加载，如果已经加载则直接返回原来已经加载的类。

（2）如果没有找到，就去委托父类加载器去加载（如代码c = parent.loadClass(name, false)所示）。父类加载器也会采用同样的策略，查看自己已经加载过的类中是否包含这个类，有就返回，没有就委托父类的父类去加载，一直到启动类加载器。因为如果父加载器为空了，就代表使用启动类加载器作为父加载器去加载。

（3）如果启动类加载器加载失败（例如在$JAVA\_HOME/jre/lib里未查找到该class），则会抛出一个异常ClassNotFoundException，然后再调用当前加载器的findClass()方法进行加载。

双亲委派模型的好处：

（1）主要是为了安全性，避免用户自己编写的类动态替换 Java的一些核心类，比如 String。

（2）同时也避免了类的重复加载，因为 JVM中区分不同类，不仅仅是根据类名，相同的 class文件被不同的 ClassLoader加载就是不同的两个类。

图1

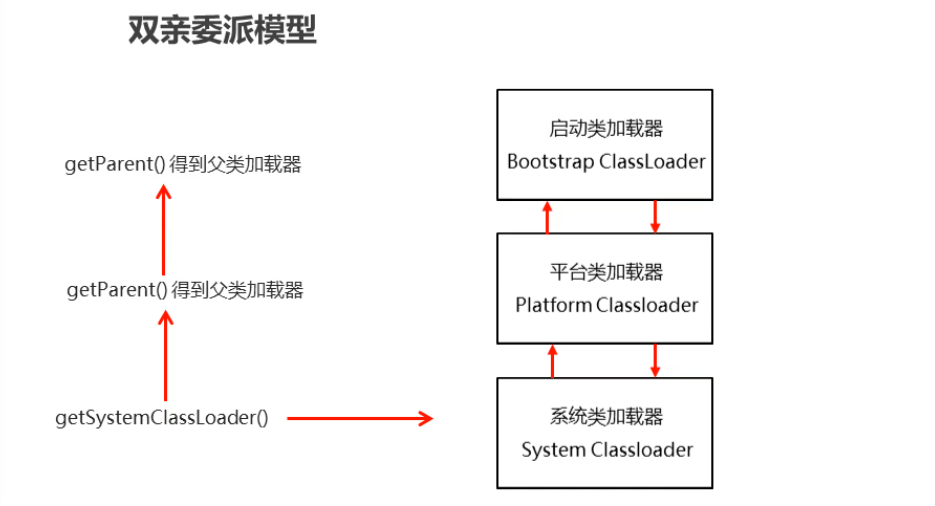
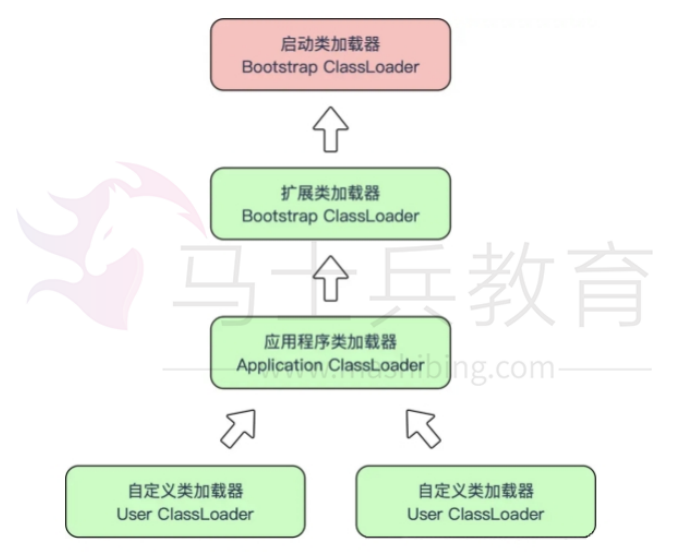


图2



双亲委托有个弊端：

不能向下委派，不能不委派

怎么打破双亲委派机制：（也就是能向下委派和不委派）

自定义类加载器（不委派）

spi机制（向下委派）

打破双亲委派的两种方式：

1.通过spi机制，使用ServiceLoader.load去加载

2.通过自定义类加载器，继承classloader，重写loadclass方法

SPI机制

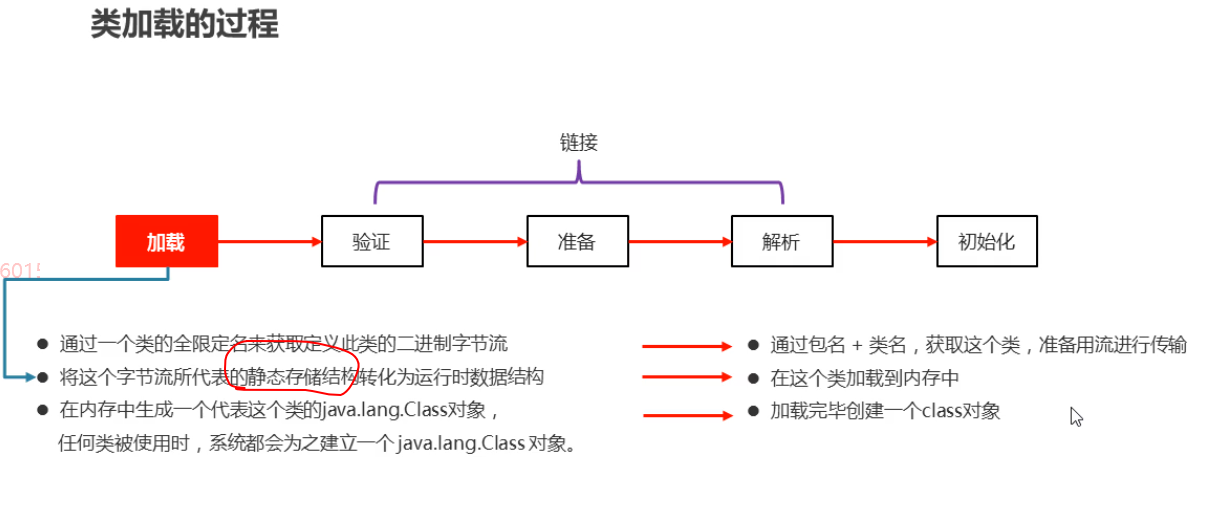
spi机制是一种服务发现机制。它通过在ClassPath路径下的META-INF/services文件夹查找文件，自动加载文件里所定义的类。这一机制为很多框架扩展提供了可能，比如在JDBC中就使用到了SPI机制。

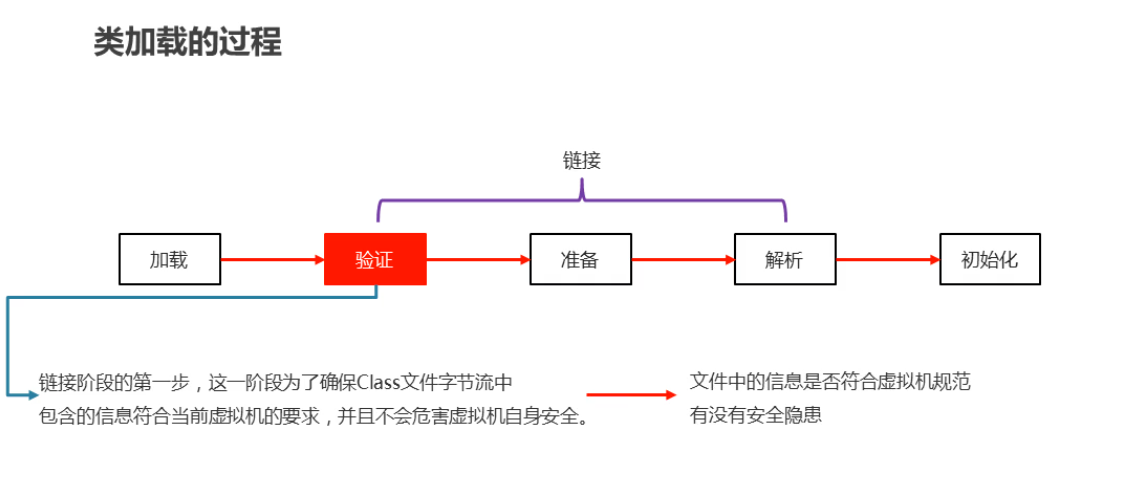
为什么通过spi机制就能打破双亲委托？

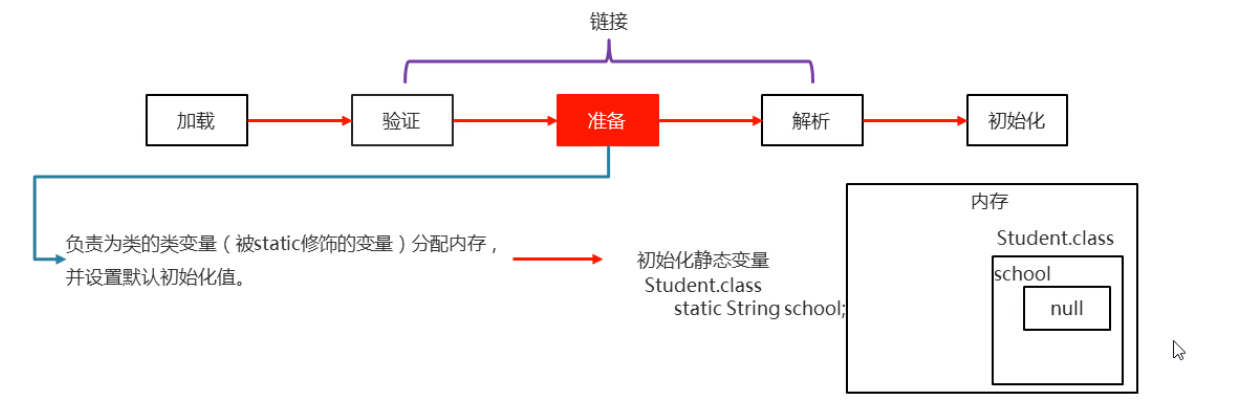
因为在某些情况下父类加载器需要委托子类加载器去加载class文件。受到加载范围的限制，父类加载器无法加载到需要的文件。

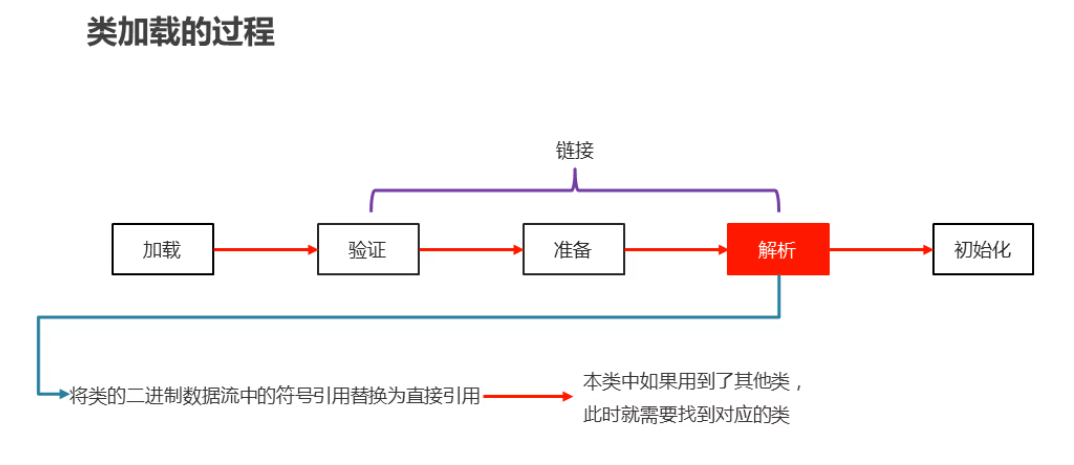
以Driver接口为例，DriverManager通过Bootstrap ClassLoader加载进来的，而com.mysql.jdbc.Driver是通过Application ClassLoader加载进来的。由于双亲委派模型，父加载器是拿不到通过子加载器加载的类的。这个时候就需要启动类加载器Bootstrap ClassLoader来委托子类来加载Driver实现，从而破坏了双亲委派。

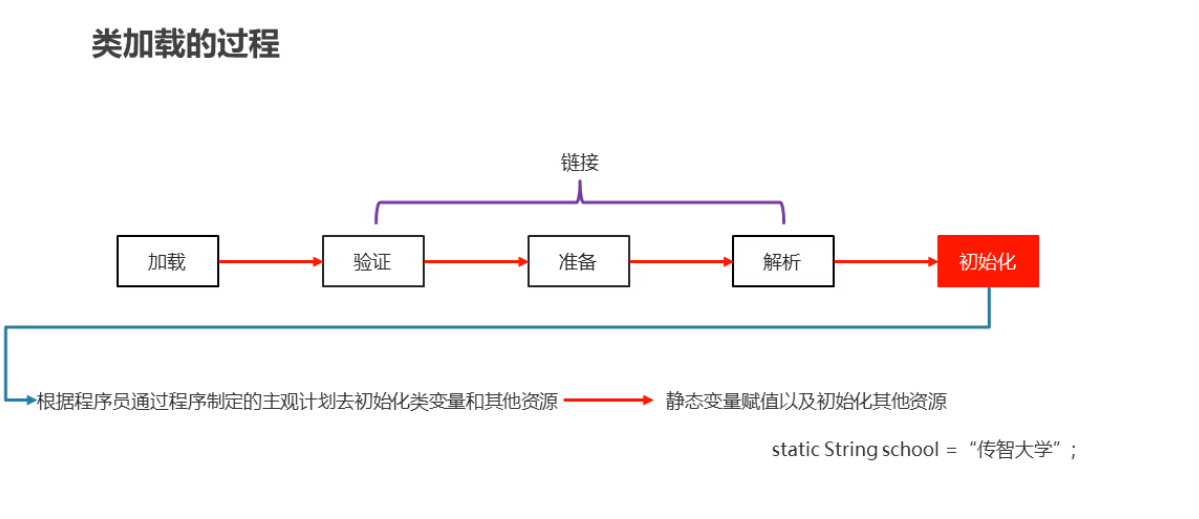
## 4、类的加载流程是什么样的，每个阶段解释一下











## 5、JVM的GC的主要区域以及各自的GC机制是什么样的？

(1) 年轻代（Young Generation）

1. 所有新生成的对象首先都是放在年轻代的。

2. 新生代内存按照8:1:1 的比例分为一个eden区和两个Survivor(survivor0,survivor1) 区。大部分对象在 Eden 区中生成。回收时先将 eden 区存活对象复制到一个 survivor0 区，然后清空 eden 区，当这个 survivor0 区也存放满了时，则将 eden 区和 survivor0 区存活对象复制到另一个 survivor1 区，然后清空 eden 和这个 survivor0 区，此时 survivor0 区是空的，然后将 survivor0 区和 survivor1 区交换，即保持 survivor1 区为空， 如此往复。

3.当 survivor1 区不足以存放 eden 和 survivor0 的存活对象时，就将存活对象直接存放到老年代。若是老年代也满了就会触发一次 Full GC，也就是新生代、老年代都进行回收。

4.新生代发生的 GC 也叫做 Minor GC，MinorGC 发生频率比较高(不一定等 Eden 区满了才触发)

(2) 年老代（Old Generation）

1.在年轻代中经历了 15次垃圾回收后仍然存活的对象，就会被放到年老代中。因此，可以认为年老代中存放的都是一些生命周期较长的对象。

2.内存比新生代也大很多(大概是 2 倍)，当老年代内存满时触发 Major GC 即 Full GC， Full GC 发生频率比较低，老年代对象存活时间比较长，存活率比较高。

(3) 持久代（Permanent Generation）

用于存放静态文件，如 Java 类、方法等。持久代对垃圾回收没有显著影响，从 JDK8以后已经废弃, 将存放静态文件，如 Java 类、方法等这些存储到了元数据区。

## 6、JVM的GC算法都有哪些？

6.3.1 如何发现垃圾

Java 语言规范并没有明确的说明 JVM 使用哪种垃圾回收算法，但是常见的用于「发现垃圾」的算法有两种，引用计数算法和根搜索算法。

1．引用计数算法

该算法很古老（了解即可）。核心思想是，堆中的对象每被引用一次，则计数器加 1，

每减少一个引用就减 1，当对象的引用计数器为 0 时可以被当作垃圾收集。

优点：快。

缺点：无法检测出循环引用。如两个对象互相引用时，他们的引用计数永远不可能为 0。

2．根搜索算法(也叫可达性分析)

根搜索算法是把所有的引用关系看作一张图，从一个节点 GC ROOT 开始，寻找对应的引用节点，找到这个节点以后，继续寻找这个节点的引用节点，当所有的引用节点寻找完毕之后，剩余的节点则被认为是没有被引用到的节点，即可以当作垃圾。

Java 中可作为 GC Root 的对象有

1.虚拟机栈中引用的对象

2.本地方法栈引用的对象

3.元数据区中静态属性引用的对象

4.元数据区中常量引用的对象

6.3.2 如何回收垃圾

Java 中用于「回收垃圾」的常见算法有 4 种：

1. 标记-清除算法（mark and sweep）

分为“标记”和“清除”两个阶段：首先标记出所有存活的对象，在标记完成之后统一回收掉所有未被标记的对象。

缺点：首先，效率问题，标记和清除效率都不高。其次，标记清除之后会产生大量的不连续的内存碎片。

2. 标记-整理算法

是在标记-清除算法基础上做了改进，标记阶段是相同的，但标记完成之后不是直接对可回收对象进行清理，而是让所有存活的对象都向一端移动，在移动过程中清理掉可回收的对象，这个过程叫做整理。

优点：内存被整理后不会产生大量不连续内存碎片。

3. 复制算法（copying）

将可用内存按容量分成大小相等的两块，每次只使用其中一块，当这块内存使用完了， 就将还存活的对象复制到另一块内存上去，然后把使用过的内存空间一次清理掉。

缺点：可使用的内存只有原来一半。

4. 分代收集算法（generation）

当前主流 JVM 都采用分代收集(Generational Collection)算法, 这种算法会根据对象存活周期的不同将内存划分为年轻代、年老代、永久代，不同生命周期的对象可以采取不同的回收算法，以便提高回收效率。新生代区用复制算法、老年代区用标记-整理/清除。

## 7、JVM调优参数

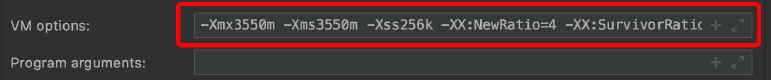


- Xmx3550m：设置 JVM堆内存 最大可用内存为 3550M。

- Xms3550m：设置 JVM堆内存 初始内存为 3550m。注意：此值一般设置成和-Xmx 相同，以避免每次垃圾回收完成后 JVM 重新分配内存。

- Xmn2g：设置年轻代大小为 2G。整个 JVM 堆内存大小=年轻代大小 + 年老代大小 + 持久代大小。此值对系统性能影响较大，Sun 官方推荐配置为整个堆的 3/8。

- Xss256k：设置每个线程的栈大小。JDK5.0 以后每个线程栈大小为 1M，以前每个线程栈大小为 256K。根据应用的线程所需内存大小进行调整。在相同物理内存下，减小这个值能生成更多的线程。



- XX:NewRatio=4:设置年轻代（包括 Eden 和两个 Survivor 区）与年老代的比值（除去持久代）。设置为 4，则年轻代与年老代所占比值为 1：4。（该值默认为 2）

- XX:SurvivorRatio=4：设置年轻代中 Eden 区与 Survivor 区的大小比值。设置为 4，则两个 Survivor 区与一个 Eden 区的比值为 2:4。

# 三、Spring & SpringBoot

## 1、说下对SpringIOC的理解，怎么理解控制反转？IOC容器中Bean的生命周期？

1. Spring 的两大核心是：IOC（控制反转）和 AOP（面向切面编程） DI（依赖注入）

2. IOC 的意思是控制反转，是指创建对象的控制权的转移，以前创建对象的主动权和时机是由自己把控的，而现在这种权力转移到 Spring 容器中，并由容器根据配置文件去创建实例和管理各个实例之间的依赖关系，对象与对象之间松散耦合，也利于功能的复用。最直观的表达就是，IOC 让对象的创建不用去 new 了，可以由 spring 根据我们提供的配置文件自动生产，我们需要对象的时候，直接从 Spring 容器中获取即可. Spring 的配置文件中配置了类的字节码位置及信息, 容器生成的时候加载配置文件识别字节码信息, 通过反射创建类的对象。

Spring 的 IOC 有三种注入方式 ：构造器注入, setter 方法注入, 根据注解注入。

3. DI 的意思是依赖注入，和控制反转是同一个概念的不同角度的描述，即应用程序在运行时依赖IOC容器来动态注入对象需要的外部资源。

**2.2 Spring 的生命周期?(高薪常问)**

0. 通过AbstractBeanFactory.doGetBean()方法中的getSingleton()方法先从三级缓存中逐级去拿，拿不到就去创建：doCreateBean();

在doCreateBean方法中，主要做了三件事:1.实例化、2.属性填充、3.初始化

1. 实例化一个 Bean，也就是我们通常说的 new

1)、通过createBeanInstance方法去实例化对象：底层堆内存创建了一片空间，执行Bean的构造器方法

2)、通过addSingletonFactory方法将已完成实例化，但是未完成属性赋值和相关初始化的一个不完整bean添加到三级缓存中

2. 按照 Spring 上下文对实例化的 Bean 进行配置，也就是 IOC 注入

通过populateBean()方法进行属性复制、依赖注入

3. 如果这个 Bean 实现了 BeanNameAware 接口，会调用它实现的 setBeanName(String beanId)方法，此处传递的是 Spring 配置文件中 Bean 的 ID

4. 如果这个 Bean 实现了 BeanFactoryAware 接口，会调用它实现的 setBeanFactory()，传递的是 Spring 工厂本身（可以用这个方法获取到其他 Bean）

5. 如果这个 Bean 实现了 ApplicationContextAware 接口，会调用 setApplicationContext(ApplicationContext)方法，传入 Spring 上下文，该方式同样可以实现步骤 4，但比 4 更好，因为ApplicationContext 是 BeanFactory 的子接口，有更多的实现方法

6. 如果这个 Bean 关联了 BeanPostProcessor 接口，将会调用 postProcessBeforeInitialization(Object obj, String s)方法，BeanPostProcessor 经常被用作是 Bean 内容的更改，并且由于这个是在 Bean 初始化结束时调用 After方法，也可用于内存或缓存技术

7. 如果这个 Bean在 Spring 配置文件中配置了 init-method 属性会自动调用其配置的初始化方法

8. 如果这个Bean关联了BeanPostProcessor 接口，将会调用 postProcessAfterInitialization(Object obj, String s)方法

注意：以上工作完成以后就可以用这个 Bean 了，那这个 Bean 是一个 single 的，所以 一般情况下我们调用同一个 ID 的 Bean 会是在内容地址相同的实例

9. 当 Bean 不再需要时，会经过清理阶段，如果 Bean 实现了 DisposableBean 接口，会调用其实现的 destroy 方法

10. 最后，如果这个 Bean 的 Spring 配置中配置了 destroy-method 属性，会自动调用其配置的销毁方法。

## 2、Spring如何解决IOC中的循环依赖问题？



基于单例场景下才可以解决，多例与构造器循环spring无法解决。

SpringIOC 通过三级缓存来解决循环依赖问题，三级缓存指的是三个Map（一级、二级是concurrentHashMap，线程安全，三级是hashmap，线程不安全）：

singletonObjects：一级缓存，key为BeanName，value为Bean，日常获取Bean的地方

earlySingletonObjects：二级缓存，key为BeanName，value为Bean，已经实例化但还没有进行属性注入的Bean，由三级缓存放入

singletonFactories：三级缓存，key为BeanName，value为对象工厂（ObjectFactory）

在实际使用中，要获取一个bean，先从一级缓存一直查找到三级缓存，缓存bean的时候是从三级到一级的顺序保存，并且缓存bean的过程中，三个缓存都是互斥的，只会保持bean在一个缓存中，而且，最终都会在一级缓存中。

解决循环依赖

SpringIOC解决循环依赖的思路就是依靠缓存，同时还得引出个概念即早期暴露引用。我们知道在IOC容器里Bean的初始化的过程分为三个步骤：创建实例、属性注入实例、回调实例实现的接口方法。

解决思路就在这：当我们创建实例与属性注入实例这俩个步骤之间的时候，我们引入缓存，将这些已经创建好但是并没有注入属性的实例放到缓存里，而这些放在缓存里但是没有被注入属性的实例对象，就是解决循环依赖的方法。

打个比方：A对象的创建需要引用到B对象，而B对象的创建也需要A对象，而此时当B对象创建的时候直接从缓存里引用A对象（虽然不是完全体A对象，毕竟没有赋值处理），当B对象完成创建以后再被A对象引用进去，则A对象也完成了创建。

解决循环依赖具体过程

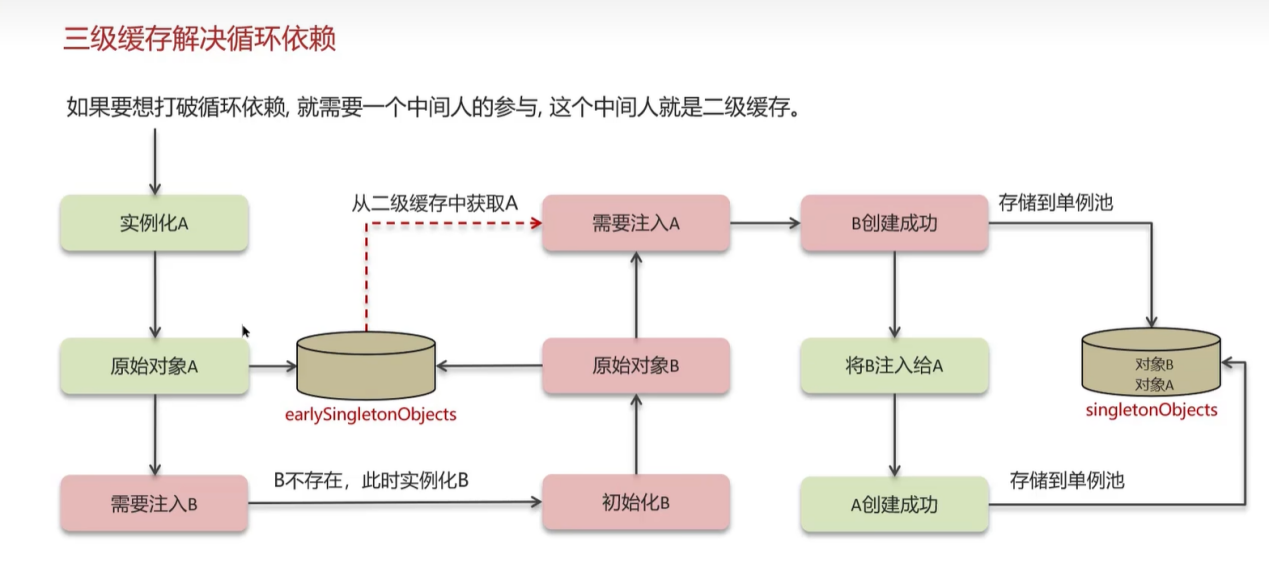
对Bean的创建最为核心三个方法解释如下：

createBeanInstance：实例化，通过反射调用对象构造方法实例化对象；

populateBean：填充属性，主要是对bean的依赖属性进行赋值；

initializeBean：初始化，可以回调InitializingBean、initMethod等方法。

实例化A的时候，先将A创建的早期对象放入三级缓存（singletonFactories）中。这个时候虽然属性没有赋值，但是容器已经能认识这个是A对象，只是属性全是null而已。在populateBean方法中对属性赋值的时候，发现A依赖了B，那么就先去创建B，又走一遍bean的创建过程（创建B）。同样也会把B的早期对象放入三级缓存（singletonFactories）中。当B又走到 populateBean方法（负责填充Bean实例属性的方法）的时候，发现依赖了A，我们又去创建A，但是这个时候去创建A，发现我们在三级缓存（singletonFactories）能找到A（早期对象），此时会通过A的ObjectFactory获取A，并把A从三级缓存移到二级缓存。然后就可以把B的A属性赋值了，这个时候B就初始化完成了，初始化完成后就会把B从三级缓存移到一级缓存。完成B实例化后，回到A调用的populateBean方法中。返回的就是B对象了，对A的B属性进行赋值就可以了。



## 3、说下对SpringAOP的理解、有哪些通知？使用场景有哪些？（底层原理：两种动态代理）

AOP，一般称为面向切面编程，作为面向对象的一种补充，用于将那些与业务无关，但却对多个对象产生影响的公共行为和逻辑，抽取并封装为一个可重用的模块，这个模块被命名为“切面”（Aspect）. SpringAOP 使用的是动态代理，所谓的动态代理就是说 AOP 框架不会去修改字节码，而是每次运行时在内存中临时为方法生成一个代理对象，这个代理对象包含了目标对象的全部方法，并且在特定的切点做了增强处理，并回调原对象的方法。

Spring AOP 中的动态代理主要有两种方式，JDK 动态代理和 CGLIB 动态代理：(1)JDK 动态代理只提供接口代理，不支持类代理，核心 InvocationHandler 接口和Proxy 类，InvocationHandler 通过 invoke()方法反射来调用目标类中的代码，动态地将横切逻辑和业务编织在一起，Proxy 利用 InvocationHandler 动态创建一个符合某一接口的实例, 生成目标类的代理对象。

(2) 如果代理类没有实现 InvocationHandler 接口，那么 Spring AOP 会选择使用CGLIB 来动态代理目标类。CGLIB（Code Generation Library），是一个代码生成的类库，可以在运行时动态的生成指定类的一个子类对象，并覆盖其中特定方法并添加增强代码，从而实现 AOP。CGLIB 是通过继承的方式做的动态代理，因此如果某个类被标记为 final，那么它是无法使用 CGLIB 做动态代理的。

Spring AOP的通知类型有五种，分别为：

前置通知[]before]：在切点运行之前执行

后置通知[after-returning]：在切点正常结束之后执行

异常通知[after-throwing]：在切点发生异常的时候执行

最终通知[after]：在切点的最终执行

Spring 还有一种特殊的通知,叫做环绕通知[around]，环绕通知运行程序员以编码的方式自己定义通知的位置, 用于解决其他通知时序问题。

AOP的使用场景： 日志、权限、缓存、同步、事务

Authentication 权限

Caching 缓存

Context passing 内容传递

Error handling 错误处理

Lazy loading　懒加载

Debugging　　调试

logging, tracing, profiling and monitoring　记录跟踪　优化　校准

Performance optimization　性能优化

Persistence　　持久化

Resource pooling　资源池

Synchronization　同步

Transactions 事务

## 4、说下SpringMvc的流程（从访问一个URL到得到页面结果的具体流程：DispatcherServlet的职责流程）

1. 用户发送请求到前端控制器（DispatcherServlet）

2. 前端控制器（ DispatcherServlet ）收到请求调用处理器映射器（HandlerMapping），去查找处理器（Handler）

3. 处理器映射器（HandlerMapping）找到具体的处理器(可以根据 xml 配置、注解进行查找)，生成处理器对象及处理器拦截器(如果有则生成)一并返回给 DispatcherServlet。

4. 前端控制器（DispatcherServlet）调用处理器适配器（HandlerAdapter）

5. 处理器适配器（HandlerAdapter）去调用自定义的处理器类(Controller，也叫后端控制器)。

6.自定义的处理器类(Controller，也叫后端控制器)将得到的参数进行处理并返回结果给处理器适配器（HandlerAdapter）

7. 处理器适配器 （ HandlerAdapter ）将得到的结果返回给前端控制器（DispatcherServlet）

8. DispatcherServlet(前端控制器) 将 ModelAndView 传给视图解析器(ViewReslover)

9. 视图解析器(ViewReslover)将得到的参数从逻辑视图转换为物理视图并返回给前端控制器（DispatcherServlet）

10. 前端控制器（DispatcherServlet）调用物理视图进行渲染并返回

11. 前端控制器（DispatcherServlet）将渲染后的结果响应

## 5、对Spring声明式事务的理解？Spring的事务隔离级别？Spring事务传播行为？

所谓声明式事务，就是通过配置的方式，比如通过配置文件（xml）或者注解的方式，告诉spring，哪些方法需要spring帮忙管理事务，然后开发者只用关注业务代码，而事务的事情spring自动帮我们控制。

比如注解的方式，只需在方法上面加一个@Transactional注解，那么方法执行之前spring会自动开启一个事务，方法执行完毕之后，会自动提交或者回滚事务，而方法内部没有任何事务相关代码，用起来特别的方便。

Spring事务的隔离级别：

① ISOLATION\_DEFAULT：这是个 PlatfromTransactionManager 默认的隔离级别，使用数据库默认的事务隔离级别。

② ISOLATION\_READ\_UNCOMMITTED：读未提交，允许另外一个事务看到这个事务未提交的数据。

③ ISOLATION\_READ\_COMMITTED：读已提交，保证一个事务修改的数据提交后才能被另一事务读取，而且能看到该事务对已有记录的更新。解决脏读问题。

④ ISOLATION\_REPEATABLE\_READ：可重复读，保证一个事务修改的数据提交后才能被另一事务读取，但是不能看到该事务对已有记录的更新。行锁。

⑤ ISOLATION\_SERIALIZABLE：可序列化，一个事务在执行的过程中完全看不到其他事务对数据库所做的更新。表锁。

Spring事务的传播行为：

spring 事务的传播行为说的是，当多个事务同时存在的时候，spring 如何处理这些事务的行为。

备注(方便记忆): propagation 传播

require 必须的/support 支持/mandatory 强制托管/requires-new 需要新建/not -supported 不支持/never 从不/nested 嵌套的

① PROPAGATION\_REQUIRED：如果当前没有事务，就创建一个新事务，如果当前存在事务，就加入该事务，该设置是最常用的设置，是默认的。

② PROPAGATION\_SUPPORTS：支持当前事务，如果当前存在事务，就加入该事务，如果当前不存在事务，就以非事务执行。

③ PROPAGATION\_MANDATORY：支持当前事务，如果当前存在事务，就加入该事务，如果当前不存在事务，就抛出异常。

④ PROPAGATION\_REQUIRES\_NEW：创建新事务，无论当前存不存在事务，都创建新事务。

⑤ PROPAGATION\_NOT\_SUPPORTED：以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。

⑥ PROPAGATION\_NEVER：以非事务方式执行，如果当前存在事务，则抛出异常。

⑦ PROPAGATION\_NESTED：如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则按 REQUIRED 属性执行（新建事务加入）。

## 6、什么情况下会让spring事务失效

（1）抛出检查异常而不是运行时异常

（2）业务自己捕获了异常

（3）本类无事务的方法调用了本类有事务的方法，事务失效（代理执行了原始无事务的方法，注入自己的实例）

（4）非public修饰的方法

（5）方法用final修饰

（6）方法用static修饰

（7）错误的传播行为

## 7、Spring有哪些核心注解？分别的作用？

1. @Component(任何层) @Controller @Service @Repository（dao）: 用于实例化对象

2. @Scope : 设置 Spring 对象的作用域

3. @PostConstruct @PreDestroy : 用于设置 Spring 创建对象在对象创建之后和销毁之前要执行的方法

4. @Value: 简单属性的依赖注入

5. @Autowired:引用类型的依赖注入

6. @Qualifier: 要和@Autowired 联合使用，代表在按照类型匹配的基础上，再按照名称匹配。

7. @Resource 按照属性名称依赖注入

8. @ComponentScan: 组件扫描

9. @Bean: 标在方法上,用于将方法的返回值对象放入容器

10. @PropertySource: 用于引入其它的 properties 配置文件

11. @Import: 在一个配置类中导入其它配置类的内容

12. @Configuration: 被此注解标注的类,会被 Spring 认为是配置类。Spring 在启动的时候会自动扫描并加载所有配置类，然后将配置类中 bean 放入容器

13. @Transactional 此注解可以标在类上，也可以表在方法上，表示当前类中的方法具有事务管理功能。

## 8、Spring和SpringBoot的关系？

（1）Spring

Spring是一个开源容器框架，可以接管web层，业务层，dao层，持久层的组件，并且可以配置各种bean,和维护bean与bean之间的关系。其核心就是控制反转(IOC),和面向切面(AOP)，简单的说就是一个分层的轻量级开源框架。

（2）SpringBoot

Springboot是一个微服务框架，延续了spring框架的核心思想IOC和AOP，简化了应用的开发和部署。Spring Boot是为了简化Spring应用的创建、运行、调试、部署等而出现的，使用它可以做到专注于Spring应用的开发，而无需过多关注XML的配置。提供了一堆依赖打包，并已经按照使用习惯解决了依赖问题—>习惯大于约定。

## 9、SpringBoot的自动装配原理是什么？

A. SpringBoot 自动配置的注解是 @EnableAutoConfiguration.

B. 我们用的时候是在启动类上加@SpringBootApplication,这个注解是复合注解,内部包含 @EnableAutoConfiguration

C. @EnableAutoConfiguration 内部有一个@Import, 这个注解才是完成自动配置的关键.

D. @Import 导入一个类(AutoConfigurationImportSelector),这个类内部提供了一个方法(selectImports). 这个方法会扫描导入的所有 jar 包下spring.factories 文件， 解析文件中自动配置类 key=value, 将列表中的类创建,并放到 Spring 容器中。

## 10、SpringBoot的核心注解是哪个？详细说下

启动类上面的注解是@SpringBootApplication，它也是 Spring Boot 的核心注解，主要组合包含了以下4类注解：

（1）第一类: JDK 原生注解 4 个

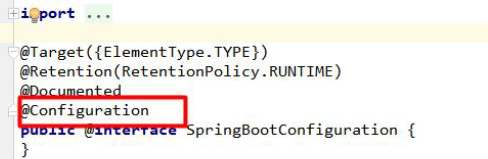
@Target(ElementType.TYPE) //当前注解的使用范围@Retention(RetentionPolicy.RUNTIME) //生命周期

@Documented //声明在生成 doc 文档时是否带着注解

@Inherited //声明是否子类会显示父类的注解

（2）第二类: @SpringBootConfiguration

点开该注解源码, 会发现本质是@Configuration,定义该类是个配置类功能等同于 xml配置文件.



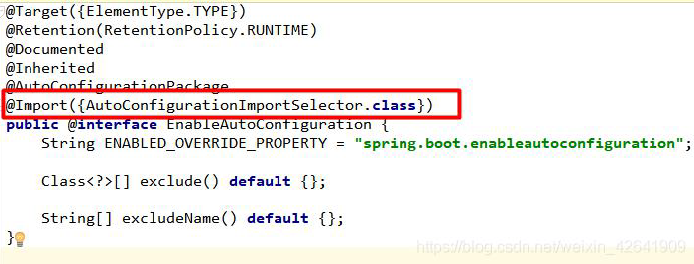
提到@Configuration 就要提到他的搭档@Bean, 使用这两个注解就可以创建一个简单的 Spring 配置类, 可以用来替代相应的 xml 配置文件.可以理解为创建了 IOC 容器了。

（3）第三类: @ComponentScan, 包扫描功能

这个注解对应 Spring 的 XML 配置中的@ComponentScan，其实就是自动扫描并加载符合条件的组件(比如@Component 和@Repository 等)或者 bean 定义, 最终将这些bean 定义加载到 IoC 容器中. 也可以通过 basePackages 等属性来细粒度的定制@ComponentScan 自动扫描的范围, 如果不指定, 则默认扫描@ComponentScan 所在类的 package 及子包进行扫描。注：所以 SpringBoot 的启动类最好是放在 root package 下，因为默认不指定basePackages, 这样能扫描 root package 及子包下的所有类。

(4)第四类: @EnableAutoConfiguration

点开源码会发现,本质是@import, 自动导入功能



1. @EnableAutoConfiguration 也是借助@Import 的帮助，将所有符合自动配置条件的 bean 定义加载到 IoC 容器.

@EnableAutoConfiguration 会根据类路径中的 jar 依赖为项目进行自动配置, 如：添加了spring-boot-starter-web 依赖, 会自动添加 Tomcat 和 SpringMVC 的依赖, SpringBoot会对 Tomcat 和 SpringMVC 进行自动配置。

## 11、SpringBoot项目的启动加载流程大概说下

一、SpringBoot启动的时候，会构造一个SpringApplication的实例，构造SpringApplication的时候会进行初始化的工作，初始化的时候会做以下几件事：

1、把参数sources设置到SpringApplication属性中，这个sources可以是任何类型的参数.

2、判断是否是web程序，并设置到webEnvironment的boolean属性中.

3、创建并初始化ApplicationInitializer，设置到initializers属性中 。

4、创建并初始化ApplicationListener，设置到listeners属性中 。

5、初始化主类mainApplicationClass。

二、SpringApplication构造完成之后调用run方法，启动SpringApplication，run方法执行的时候会做以下几件事：

1、构造一个StopWatch计时器，用来记录SpringBoot的启动时间 。

2、初始化监听器，获取SpringApplicationRunListeners并启动监听，用于监听run方法的执行。

3、创建并初始化ApplicationArguments,获取run方法传递的args参数。

4、创建并初始化ConfigurableEnvironment（环境配置）。封装main方法的参数，初始化参数，写入到 Environment中，发布 ApplicationEnvironmentPreparedEvent（环境事件），做一些绑定后返回Environment。

5、打印banner和版本。

6、构造Spring容器(ApplicationContext)上下文。先填充Environment环境和设置的参数，如果application有设置beanNameGenerator（bean）、resourceLoader（加载器）就将其注入到上下文中。调用初始化的切面，发布ApplicationContextInitializedEvent（上下文初始化）事件。

7、SpringApplicationRunListeners发布finish事件。

8、StopWatch计时器停止计时，日志打印总共启动的时间。

9、发布SpringBoot程序已启动事件(started())

10、调用ApplicationRunner和CommandLineRunner

11、最后发布就绪事件ApplicationReadyEvent，标志着SpringBoot可以处理就收的请求了(running())

## 12、SpringBoot项目读取配置文件的方式有几种？

有三种方式读取配置文件：

1.使用Environment

2.使用@Value（”${ }”）

3.使用@ConfigurationProperties注解映射到bean中，自定义对象封装指定数据，注解中添加prefix属性

## 13、如何自定义SpringBoot starter？

springboot加载starter原理，其实就是加载依赖jar包下的spring.factories文件。所以我们要自定义starter，就需要在项目中建立一个META-INF的文件夹，然后在文件夹下面建一个spring.factories文件，文件里将你需要提供出去的bean实例信息配置好就行

## 14、BeanFactory和FactoryBean的区别？

1）BeanFactory 是接口，提供了IOC容器最基本的形式，给具体的IOC容器的实现提供了规范，顶层接口。

2）FactoryBean 也是接口，为IOC容器中Bean的实现提供了更加灵活的方式，FactoryBean在IOC容器的基础上给Bean的实现加上了一个简单的工厂模式和装饰模式 ，我们可以在getObject()方法中灵活配置.

区别：FactoryBean是个Bean，在Spring中，所有的Bean都是由BeanFactory(也就是IOC容器)来进行管理的。但对FactoryBean而言，这个Bean不是简单的Bean，而是一个能生产或者修饰对象生成的工厂Bean,它的实现与设计模式中的工厂模式和修饰器模式类似。

## 15、Spring Bean的生命周期是什么？

0. 通过AbstractBeanFactory.doGetBean()方法中的getSingleton()方法先从三级缓存中逐级去拿，拿不到就去创建：doCreateBean();

在doCreateBean方法中，主要做了三件事:1.实例化、2.属性填充、3.初始化

1. 实例化一个 Bean，也就是我们通常说的 new

1)、通过createBeanInstance方法去实例化对象：底层堆内存创建了一片空间，执行Bean的构造器方法

2)、通过addSingletonFactory方法将已完成实例化，但是未完成属性赋值和相关初始化的一个不完整bean添加到三级缓存中

2. 按照 Spring 上下文对实例化的 Bean 进行配置，也就是 IOC 注入

通过populateBean()方法进行属性复制、依赖注入

3. 如果这个 Bean 实现了 BeanNameAware 接口，会调用它实现的 setBeanName(String beanId)方法，此处传递的是 Spring 配置文件中 Bean 的 ID

4. 如果这个 Bean 实现了 BeanFactoryAware 接口，会调用它实现的 setBeanFactory()，传递的是 Spring 工厂本身（可以用这个方法获取到其他 Bean）

5. 如果这个 Bean 实现了 ApplicationContextAware 接口，会调用 setApplicationContext(ApplicationContext)方法，传入 Spring 上下文，该方式同样可以实现步骤 4，但比 4 更好，因为ApplicationContext 是 BeanFactory 的子接口，有更多的实现方法 （事件发布/监听支持，多语言实现适应不同的语言环境。AOP支持 定时任务，数据访问支持，可以使用JDBC、ORM框架等进行数据访问）

6. 如果这个 Bean 关联了 BeanPostProcessor 接口，将会调用 postProcessBeforeInitialization(Object obj, String s)方法，BeanPostProcessor 经常被用作是 Bean 内容的更改，并且由于这个是在 Bean 初始化结束时调用 After方法，也可用于内存或缓存技术

7. 如果这个 Bean在 Spring 配置文件中配置了 init-method 属性会自动调用其配置的初始化方法

8. 如果这个Bean关联了BeanPostProcessor 接口，将会调用 postProcessAfterInitialization(Object obj, String s)方法

注意：以上工作完成以后就可以用这个 Bean 了，那这个 Bean 是一个 single 的，所以 一般情况下我们调用同一个 ID 的 Bean 会是在内容地址相同的实例

9. 当 Bean 不再需要时，会经过清理阶段，如果 Bean 实现了 DisposableBean 接口，会调用其实现的 destroy 方法

10. 最后，如果这个 Bean 的 Spring 配置中配置了 destroy-method 属性，会自动调用其配置的销毁方法。

# 四、Mysql & Mybatis？

## 1、什么是索引？

概念：

索引存储在内存中，为服务器存储引擎为了快速找到记录的一种数据结构。索引的主要作用是加快数据查找速度，提高数据库的性能。

优点：

（1）创建唯一性索引，保证数据库表中每一行数据的唯一性

（2）大大加快数据的检索速度，这也是创建索引的最主要的原因

（3）加速表和表之间的连接，特别是在实现数据的参考完整性方面特别有意义。

（4）在使用分组和排序字句进行数据检索时，同样可以显著减少查询中分组和排序的时间。

## 2、Mysql的数据结构是什么（mysql索引的数据结构）？为什么用这种结构？（如何提高磁盘IO效率）

Mysql的数据结构是B+树，MyISAM、InnoDB两种存储引擎都是基于B+树数据结构存储表数据的。不同之处是：

**MyISAM**的B+树**只存储了索引key值地址**，真正的数据存储在别的地方

​**InnoDB** 存储引擎中的B+树既**存储了索引也存储了数据。**

由于数据存储于物理磁盘，所以要尽量减少从磁盘IO数据的次数。因为B+树检索一次最多需要访问h个节点，h通常也不会大于3，数据库系统的设计者巧妙利用了磁盘局部预读原理，预读的大小通常为页的倍数，将一个节点的大小设为等于一个页，这样每个节点只需要一次I/O就可以完全载入，所以通常一次Mysql检索磁盘的IO不大于3次，3层的B+树就可以容纳2100万条数据，因此用B+Tree作为索引结构效率是非常高的。

## 3、Mysql的数据IO查找流程是什么样的？

以一颗高度为3的B+树为例，

1.首先根据根节点找到对应磁盘块，读入内存中，进行一次磁盘IO，

2.比较要查找的数据与根节点的索引键值，找到第二层对应子节点的指针，

3.根据指针存储的子节点的地址信息找到对应的磁盘块，读入内存中，进行一次磁盘IO，

4. 比较要查找的数据与该节点的索引键值，找到第三层对应子节点的指针，

5. 根据指针存储的子节点的地址信息找到对应的磁盘块，读入内存中，进行一次磁盘IO，

6.在第三层叶子节点对应的磁盘块中找到对应的数据。

共进行3次磁盘的IO及3次内存中的查找，内存中的关键字是一个有序表结构，可以利用二分法查找提高效率。而3次磁盘I/O操作是影响整个B+Tree查找效率的决定因素。百万千万级数据一次索引查询通常不大于3次磁盘IO。

1 查看缓存中是否存在索引，

2 如果有 则从内存中访问，否则要访问磁盘，

3 并将索引数据存入内存，利用索引来访问数据,

4 对于数据也会检查数据是否存在于内存，

5 如果没有则访问磁盘获取数据，读入内存。

6 返回结果给用户。

## 4、B+tree和Btree由什么组成？他们的异同？

B树由根节点、非叶子节点、叶子节点、指向子节点的指针组成，B+树还多了一个叶子节点间的顺序指针。

（1）、B+tree是B-tree的变体；

​（2）、在B-tree的基础上增加了叶子节点间的顺序访问指针，B+Tree提高了顺序访问的性能；

​（3）、B-tree每个结点的指针上限为2d+1，B+tree每个结点的指针上限为2d；

​（4）、B+tree非叶子结点只存储索引值，叶子结点存储真实数据，B-tree所有结点上都存储数据。

## 5、Mysql两种存储引擎（InnoDB和MyISAM）的区别？这两种引擎B+tree的叶子结点和非叶子结点分别存储的什么？

（1）MyISAM 存储引擎

MySQL5.5 版本之前的默认存储引擎

查询MyISAM更优

主要特点：

1、不支持事务、外键；

2、支持表级锁；（表级锁是 MySQL 中锁定粒度最大的一种锁，表示对当前操作的整张表加锁）

3、支持全文索引；

4、MyISAM的B+树只存储了索引key值地址，真正的数据存储在别的地方

每个 myisam 在磁盘上存储为 3 个文件，文件名和表名相同，扩展名分别是

.frm -------存储表定义

.MYD --------MYData，存储数据

.MYI --------MYIndex，存储索引

（2）InnoDB 存储引擎

MySQL5.5 版本之后的默认存储引擎；

增删改InnoDB更优

主要特点：

1、支持事务与外键；

2、支持行级锁（行级锁是 Mysql 中锁定粒度最细的一种锁，表示只针对当前操作的行进行加锁）；

3、支持聚集索引方式存储数据；

4、不支持全文索引；

5、InnoDB 存储引擎中的B+树既存储了索引也存储了数据，数据和索引是集中存储的，.ibd

两种引擎索引结构的区别：

InnoDB索引是聚簇索引，MyISAM索引是非聚簇索引。

InnoDB的主键索引的叶子节点存储着行数据，因此主键索引非常高效。

MyISAM索引的叶子节点存储的是行数据地址，需要再寻址一次才能得到数据。

InnoDB非主键索引的叶子节点存储的是主键和其他带索引的列数据，因此查询时做到覆盖索引会非常高效。

两种引擎适用场景：

如果没有特别的需求，使用默认的Innodb即可。

MyISAM：以读写插入为主的应用程序，比如博客系统、新闻门户网站。

Innodb：更新（删除）操作频率也高，或者要保证数据的完整性；并发量高，支持事务和外键。比如OA自动化办公系统。

## 6、Mysql索引有哪些类型？什么场景使用哪种索引？

（1）主键索引（聚集索引）:数据列不允许重复，不允许为NULL，

一个表只能有一个主键。

（2）唯一索引:数据列不允许重复，允许为NULL值，

一个表允许多个列创建唯一索引。

（3）普通索引:基本的索引类型，没有唯一性的限制，允许为NULL值。

（4）全文索引：是目前搜索引擎使用的一种关键技术，只能在 CHAR、VARCHAR 或 TEXT 类型的列上创建。

索引的数据结构有B树索引、哈希索引等。

## 7、如何进行Mysql优化？（sql优化层面和服务器优化层面）

（1）SQL语句优化层面（理解透彻）

​ 1）、尽量避免使用select \*；

​ 2）、规范sql语句大小写，sql是有缓存的，避免每次都需要解析；

​ 3）、使用exsits代替in，要更高效；

​ 4）、mysql sql解析执行过程从右至左，基于这个规则，from后面能过滤掉更多数据的基础表放后面，where后面能过滤掉更多数据的查询条件放后面；

​ 5）、查询条件中用相同类型去查询，比如避免数值列用字符串查询条件；

​ 6）、合理使用索引

1.为合适的列添加索引(主键、唯一索引、组合索引)；

①索引字段不宜过大

②在查询频繁的字段建索引

③区分度要高

④索引不是越多越好，因为索引越多B+树越多空间成本大，然后增删改会变慢

2.尽量建立联合索引，也省空间成本；

3.尽量使用覆盖索引；

4.避免以下会使索引失效的操作

①索引列有null值不走索引---错误需要删掉 有null会走索引，但是最好不要有null值

②使用is null或is not null不走索引---错误 is null会走索引（char特殊相反）

③各种负向查询not ，not in， not like ，<> ,!= ,!> ,!< 不会使用索引

④like将%放左边不走索引

⑤查询条件的数据类型做了隐式转换会导致索引失效---string类型的索引字段使用了int类型的查询条件不走索引，反之走索引

如果索引列是字符串类型的整数, 条件查询的时候不加引号会造成索引失效。

⑥使用in或union代替or，or两侧有非索引列就不会走索引---or除了两侧都是使用主键（id>10 or id<5）才走索引，其他不会走索引

⑦尽量保持索引列干净，不在索引列上使用函数转换、运算

⑧联合索引要遵循最左匹配原则.

如建立联合索引（A,B,C）,查询顺序如下：

ABC会走索引，AB会走索引，A也会走索引，但是不能断开，如AC|CA|BC|CB|B|C都不会走索引---AC|CA只有A走索引，最左的联合索引字段只要在查询条件中，就会走索引，与其在查询条件中的位置无关，即ABC与CBA是一样的，都走索引，只要A在查询条件中就可以

⑨使用比较运算或between会使联合索引从使用比较运算的下一个索引处断开

7）、explain命令进行sql慢查询排查

​ 对于执行查询很慢的sql，通常对sql语句执行explain命令进行分析，具体信息如下：

​1. id //select查询的序列号，包含一组数字，表示查询中执行select子句或操作表的顺序，id相同，执行顺序从上至下 //id值越大，优先级越高，越先执行

​2. select\_type //查询类型 SIMPLE、PRIMARY、SUBQUERY、DERIVED、UNION、UNION RESULT

​3. table //正在访问哪个表

​4. partitions //匹配的分区

​5. type //访问的类型 效率从快到慢：NULL>system>const>eq\_ref>ref>ref\_or\_null>index\_merge>range>index>ALL

​6. possible\_keys //显示可能应用在这张表中的索引，一个或多个，但不一定实际使用到

​7. key //实际使用到的索引，如果为NULL，则没有使用索引

​8. key\_len //表示索引中使用的字节数，可通过该列计算查询中使用的索引的长度

​9. ref //显示索引的哪一列被使用了，如果可能的话，是一个常数，哪些列或常量被用于查找索引列上的值

​10. rows //根据表统计信息及索引选用情况，大致估算出找到所需的记录所需读取的行数，这个数量越小越好

​11. filtered //查询的表行占表的百分比

​12. Extra //包含不适合在其它列中显示但十分重要的额外信息。

2）、优化架构方面（了解）

​ 在数据达到一定量级以后，需要对数据库从主从、分库分表、数据分片方面进行优化：

​ **主从读写分离**：主节点写，从节点读

主库db的更新事件(update、insert、delete)被写到binlog二进制日志

主库创建一个binlog dump thread，把binlog的内容发送到从库

从库启动并发起连接，连接到主库

从库启动之后，创建一个I/O线程，读取主库传过来的binlog内容并写入到relay log中继日志中

从库启动之后，创建一个SQL线程，从relay log里面读取内容，从Exec\_Master\_Log\_Pos位置开始执行读取到的更新事件，将更新内容写入到slave的db

​ **分库**：根据业务或者其他维度把数据存放到不同数据库

​ **分表**：

​ 1、水平分表：字段都一样，分多张表存放不同时间范围或不同维度的数据，如实时数据表、历史数据表。

​ 2、垂直分表：将不同字段放在多张表，使用外键关联。查询频次高的单独作为一张表，不高的一张表。

​ 常用**分库分表中间件**：阿里的Cobar及开源社区基于Cobar维护的Mycat等。

## 8、Sql调优你会从何入手（措施）？

（1）命令:show status like 'Com\_%',通过这条命令, 我们可以知道当前数据库是以查询为主还是更新为主，如果是查询为主, 就重点优化查询；如果增删改多就重点优化写入操作。

（2）开启查询慢sql日志。

修改my.cnf文件，增加或修改参数slow\_query\_log 和slow\_query\_log\_file后，然后重启MySQL服务器。

慢查询时间设置。默认情况下long\_query\_time的值为10秒，可以使用命令修改，也可以在my.cnf参数里面修改

slow-query-log=1

slow\_query\_log\_file="D:\mysql-5.7.35-winx64\log\mysql\_slow.log"

long\_query\_time=2

（3）开启执行计划，用explain命令进行sql慢查询排查

（4）降低sql语句的复杂度

（5） 采用上述sql调优方法进行sql调优

​

## 9、Mysql中如何合理使用索引？有哪些会使索引失效的情况？

（1）为合适的列添加索引(主键、唯一索引、组合索引)；

①索引字段不宜过大

②在查询频繁的字段建索引

③区分度要高

④索引不是越多越好，因为索引越多B+树越多空间成本大，然后增删改会变慢

（2）尽量建立联合索引，也省空间成本；

（3）尽量使用覆盖索引；

（4）避免以下会使索引失效的操作

①索引列有null值不走索引---错误需要删掉 有null会走索引，但是最好不要有null值

②使用is null或is not null不走索引---错误 is null会走索引

③各种负向查询not ，not in， not like ，<> ,!= ,!> ,!< 不会使用索引，索引会从使用比较符的下一个索引开始失效

④like将%放左边不走索引

⑤查询条件的数据类型做了隐式转换---string类型的使用了int类型的不走索引，反之走索引

如果索引列是字符串类型的整数, 条件查询的时候不加引号会造成索引失效。Mysql内置的优化会有隐式转换

⑥使用in或union代替or，or两侧有非索引列就不会走索引---or除了两侧都是使用主键（id>10 or id<5）才走索引，其他不会走索引

⑦尽量保持索引列干净，不在索引列上使用函数转换、运算

⑧联合索引要遵循最左匹配原则.

如建立联合索引（A,B,C）,查询顺序如下：

ABC会走索引，AB会走索引，A也会走索引，但是不能断开，如AC|CA|BC|CB|B|C都不会走索引---AC|CA只有A走索引，最左的联合索引字段只要在查询条件中，就会走索引，与其在查询条件中的位置无关，即ABC与CBA是一样的，都走索引，只要A在查询条件中就可以

⑨使用比较运算或between会使联合索引从使用比较运算的下一个索引处断开

## 10、Mysql如何排查慢查询（哪个关键字）？分别会列出来哪些信息项？

采用explain命令进行sql慢查询排查

​ 对于执行查询很慢的sql，通常对sql语句执行explain命令进行分析，具体信息如下：

​1. id //select查询的序列号，包含一组数字，表示查询中执行select子句或操作表的顺序，id相同，执行顺序从上至下 //id值越大，优先级越高，越先执行

​2. select\_type //查询类型 SIMPLE、PRIMARY、SUBQUERY、DERIVED、UNION、UNION RESULT

​3. table //正在访问哪个表

​4. partitions //匹配的分区

​5. type //访问的类型 效率从快到慢：NULL>system>const>eq\_ref>ref>ref\_or\_null>index\_merge>range>index>ALL

​6. possible\_keys //显示可能应用在这张表中的索引，一个或多个，但不一定实际使用到

​7. key //实际使用到的索引，如果为NULL，则没有使用索引

​8. key\_len //表示索引中使用的字节数，可通过该列计算查询中使用的索引的长度

​9. ref //显示索引的哪一列被使用了，如果可能的话，是一个常数，哪些列或常量被用于查找索引列上的值

​10. rows //根据表统计信息及索引选用情况，大致估算出找到所需的记录所需读取的行数，这个数量越小越好

​11. filtered //查询的表行占表的百分比

​12. Extra //包含不适合在其它列中显示但十分重要的额外信息。

## 11、事务的特性是什么？Mysql事务隔离级别有哪几种？分别会产生什么问题？Mysql默认隔离级别是什么？Oracle呢？

（1）事务的特性：ACID

​**原子性（A）**：事务中所有操作要么全部提交成功，要么全部失败回滚，不能出现一部分失败，一部分成功的现象；

​**一致性（C）**：指在事务的执行前后保持数据库的一致性；

​**隔离性（I）**：一个事务所做的修改在最终提交之前，对其他事务是不可见的；

​**持久性（D）**：一旦事务提交，它所做的修改将会永久保存到数据库中，即使系统发生崩溃，事务执行结果也不会丢失；

（2）Mysql事务的隔离级别有4种，默认的隔离级别是可重复读，Oracle默认的隔离级别是读已提交，具体如下：



产生的问题：

1）脏读：一个事物读到了另一个事务尚未提交的数据，不符合事务的隔离性。

​2）不可重复读：同一个事务中针对同一行记录两次读出来的结果不一样，原因就是第二次读到了其他事务修改提交的数据。（逻辑正常）

​3）幻读：同一个事务中针对同一范围内的数据两次读出来的结果不一样，原因就是第二次读到了其他事务新增提交的数据

## 12、Mysql的行锁、表锁，悲观锁、乐观锁？

（1）行锁和表锁

1）主要是针对锁粒度划分的，一般分为：行锁、表锁、库锁；

行锁：访问数据库的时候，锁定整个行数据，防止并发错误。Innodb

表锁：访问数据库的时候，锁定整个表数据，防止并发错误。Myisam

2. 行锁和表锁的区别：

行锁：开销大，加锁慢，会出现死锁；锁定粒度小，发生锁冲突的概率低，并发度高

表锁：开销小，加锁快，不会出现死锁；锁定粒度大，发生锁冲突概率高，并发度最低

（2）乐观锁：

​ 思想为：乐观的认为本次事务操作数据不会有别的事务干扰，操作数据前不进行加锁，只是预先记录版本号，真正修改数据时再进行比对，如果版本号没变则修改数据，版本号变了则表明别的事务在本次事务过程中修改了数据，本次事务不修改数据。乐观锁适用于多读的应用类型，这样可以提高吞吐量。

​ 具体步骤：

​ 1、先查出版本号：select status,version from t\_goods where id=#{id}

​ 2、真正修改数据时进行版本号比对修改：update t\_goods set status=2,version=version+1 where id=#{id} and version=#{version};

​（3）悲观锁：

​ 思想为：悲观的认为本次事务一定会有别的事务干扰，操作数据前必须先加锁

​ 常见实现方式为 for update加行锁： select ... from table where id = #{id} for update，这里注意，由于mysql锁是建立在索引上面的，所以查询条件必须用主键索引，否则会造成数据全表扫描。传统的关系型数据库里边就用到了很多这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁。

## 13、Mysql的vachar和char的区别？

char长度固定，为创建表时声明的长度，长度值范围是1到255，当char值未填满指定长度时，其他空间会用空格进行填充，检索CHAR值时需删除尾随空格。

​ vachar长度为可变的，实际使用多少空间就占多少空间

使用varchar（255）的字段建索引可以吗？

​ 理论上是可以的，拿Innodb和utf编码场景来说，索引长度极限为 767字节，utf8 编码时1个字符占3个字节，varchar 类型能建立索引的最大长度为

​ 767/3 = 255，这也是为什么varchar最大长度为255的原因之一，但是实际业务场景中一般不会建如此大的索引，影响检索效率。

## 14、什么是内连接(inner join)、外连接（left join）？

（1）内连接查询相当于查询A、B的交集

隐式内连接：

SELECT 字段列表 FROM 表1,表2… WHERE 条件;

显示内连接：

SELECT 字段列表 FROM 表1 [INNER] JOIN 表2 ON 条件;

（2）外连接

左外连接：相当于查询A表所有数据和交集部分数据

右外连接：相当于查询B表所有数据和交集部分数据

左外连接

SELECT 字段列表 FROM 表1 LEFT [OUTER] JOIN 表2 ON 条件;

右外连接

SELECT 字段列表 FROM 表1 RIGHT [OUTER] JOIN 表2 ON 条件;

## 15、平时Mysql的sql练习要练到位！！

收到

## 16、Mybatis底层的原理？一级缓存和二级缓存是什么？

Mybatis底层是采用的动态代理，使用SqlSession 原始的增删改查方法。

（1）Mybatis的一级缓存也叫本地缓存，是指SQLSession，作用域是SQlSession, Mabits默认开启一级缓存。 在同一个SqlSession中，执行相同的SQL查询时，第一次会去查询数据库，并写在缓存中，第二次会直接从缓存中取。 当执行SQL时候两次查询中间发生了增删改的操作，则SQLSession的缓存会被清空。每次查询会先去缓存中找，如果找不到，再去数据库查询，然后把结果写到缓存中。 Mybatis的内部缓存使用一个HashMap，key为hashcode+statementId+sql语句。Value为查询出来的结果集映射成的java对象。SqlSession执行insert、update、delete等操作commit后会清空该SQLSession缓存。

2）二级缓存是全局缓存，基于namespace级别的缓存，Mybatis默认是没有开启二级缓存的。 第一次调用mapper下的SQL去查询信息，查询到的信息会存放在该mapper对应的二级缓存区域。第二次调用namespace下的mapper映射文件中，相同的sql去查询，会去对应的二级缓存内取结果。

## 17、mybatis #{}和${}的区别？

#{} 是占位符，预编译处理，${}是字符串替换。

Mybatis 在处理#{}时，会将sql中的#{}替换为?号，调用 PreparedStatement 的 set方法来赋值；

Mybatis 在处理${}时，就是把${}替换成变量的值。

使用#{}可以有效的防止 SQL 注入，提高系统安全性。

## 18、Mysql存储过程、存储函数、触发器、视图（View）分别用来干嘛的？创建语法是什么？

（1）存储过程

存储过程是一个预编译的SQL语句，优点是允许模块化的设计，就是说只需要创建一次，以后在该程序中就可以调用多次。如果某次操作需要执行多次SQL，使用存储过程比单纯SQL语句执行要快；

（2）存储函数

1、与存储过程本质相同，没有区别

2、函数的存在是让过程调用，存储过程内部又是调用存储函数

3、存储函数可以在sql语句中直接调用 如 select 函数 from 表名

4、存储过程能实现的，存储函数也都可以实现，存储函数可以实现的，存储过程也都可以实现

5、存储过程中可以没有返回值，也可以有多个，但存储函数中强制返回一个return

（3）触发器

触发器是用户定义在关系表上的一类由事件驱动的特殊的存储过程。触发器是指一段代码，当触发某个事件时，自动执行这些代码。

（4）视图

为了提高复杂SQL语句的复用性和表操作的安全性，MySQL数据库管理系统提供了视图特性。所谓视图，本质上是一种虚拟表，在物理上是不存在的，其内容与真实的表相似，包含一系列带有名称的列和行数据。但是，视图并不在数据库中以储存的数据值形式存在。行和列数据来自定义视图的查询所引用基本表，并且在具体引用视图时动态生成。

## 19、union和unionAll有什么区别？

一、区别1：取结果的交集

1、union: 对两个结果集进行并集操作, 不包括重复行,相当于distinct, 同时进行默认规则的排序;

2、union all: 对两个结果集进行并集操作, 包括重复行, 即所有的结果全部显示, 不管是不是重复;

二、区别2：获取结果后的操作

1、union: 会对获取的结果进行排序操作

2、union all: 不会对获取的结果进行排序操作

union all只是合并查询结果，并不会进行去重和排序操作，在没有去重的前提下，使用union all的执行效率要比union高

## 20、创建表、删除表、更新表字段语句？

（1）创建表：

CREATE TABLE 表名 (

字段名1 数据类型1,

字段名2 数据类型2,

…

字段名n 数据类型n

);

（2）删除表：

DROP TABLE 表名;

DROP TABLE IF EXISTS 表名;

（3）修改表：

1）修改表名

ALTER TABLE 表名 RENAME TO 新的表名;

2）添加字段

ALTER TABLE 表名 ADD 列名 数据类型;

3）修改列数据类型

ALTER TABLE 表名 MODIFY 列名 新数据类型;

4）修改列名与数据类型

ALTER TABLE 表名 CHANGE 列名 新列名 新数据类型;

5）删除列

ALTER TABLE 表名 DROP 列名;

## 21、mysql左外连接语句的写法？

SELECT 字段列表 FROM 表1 LEFT [OUTER] JOIN 表2 ON 条件

## 22、听过InnoDB的Mvcc技术吗？说下是什么？

全称 Multi-Version Concurrency Control，多版本并发控制。指维护一个数据的多个版本，使得读写操作没有冲突，快照读为MySQL实现MVCC提供了一个非阻塞读功能可以解决可重复读的隔离级别下产生幻读的问题。MVCC的具体实现，还需要依赖于数据库记录中的三个隐式字段（行标识、事务ID、回滚指针）、undo log日志、readView。

对一些一致性要求不高的场景和对单一数据的操作的场景还是可以发挥作用的，比如多个事务同时更改用户在线数，如果某个事务更新失败则重新计算后重试，直至成功。这样使用MVCC会极大地提高并发数，并消除线程锁。

## 23、Java实现动态代理有哪些方式？区别是什么？

JDK动态代理（基于接口）、CGLIB动态代理（基于实现类）

（1）JDK 动态代理只提供接口代理，不支持类代理，核心 InvocationHandler 接口和Proxy 类，InvocationHandler 通过 invoke()方法反射来调用目标类中的代码， Proxy 利用 InvocationHandler 动态创建一个符合某一接口的实例, 生成目标类的代理对象。

（2）如果代理类没有实现 InvocationHandler 接口，那么可选择使用CGLIB 来动态代理目标类。CGLIB（Code Generation Library），是一个代码生成的类库，可以在运行时动态的生成指定类的一个子类对象，并覆盖其中特定方法并添加增强代码，从而实现 AOP。CGLIB 是通过继承的方式做的动态代理，因此如果某个类被标记为 final，那么它是无法使用 CGLIB 做动态代理的。

# 五、多线程

## 1、创建线程的方式有哪些？相比继承Thread类，实现Runable接口的好处是什么？

方式1：

①定义一个XX类继承Thread类，并在该XX类中重写run方法；

②定义一个测试类Test，创建XX类的对象；

③调用start方法即可启动一条线程，创建多个XX类的对象，均调用start方法即可启动多线程。

方式2：

①定义一个XX类实现Runnable接口，并在该XX类中重写run方法；

②定义一个测试类Test，创建XX类对象；

③创建Thread对象，将XX对象作为参数传递；

④调用start方法即可启动一条线程，创建多个XX类的对象，均调用start方法即可启动多线程。

方式3：

①定义一个XX类实现Callable接口，并在该XX类中重写call方法；

②定义一个测试类Test，创建XX类对象；

③创建FutureTask对象，将XX对象作为构造方法参数传递；

④创建Thread对象，将FutureTask对象作为构造方法参数传递；

⑤调用start方法即可启动一条线程，创建多个XX类的对象，均调用start方法即可启动多线程。

方式4：

使用线程池创建（使用 java.util.concurrent.Executor 接口）

优劣势：

（1）使用继承 Thread 类的方式创建多线程

优势是：

编写简单，如果需要访问当前线程，则无需使用 Thread.currentThread()方法，直接使用 this 即可获得当前线程。

劣势是：

线程类已经继承了 Thread 类，所以不能再继承其他父类。

（2）采用实现 Runnable、Callable 接口的方式创建多线程。

优势是：

线程类只是实现了 Runnable 接口或 Callable 接口，还可以继承其他类。在这种方式下，多个线程可以共享同一个 target 对象，所以非常适合多个相同线程来处理同一份资源的情况，从而可以将 CPU、代码和数据分开，形成清晰的模型，较好地体现了面向对象的思想。

劣势是：

编程稍微复杂，如果要访问当前线程，则必须使用 Thread.currentThread()方法。

（3）Runnable 和 Callable 的区别

1.Callable 规定（重写）的方法是 call()，Runnable 规定（重写）的方法是 run()。

2.Callable支持泛型且其的任务执行后可返回值，而 Runnable 的任务是不能返回值的。

3.Call方法可以抛出异常，run 方法不可以。

4.运行 Callable 任务可以拿到一个 Future 对象，表示异步计算的结果。它提供了检查计算是否完成的方法，以等待计算的完成，并检索计算的结果。通过 Future对象可以了解任务执行情况，可取消任务的执行，还可获取执行结果。

## 2、线程的状态有哪些？

（1）新建（new）：新建了一个线程，但是还没有调用开启线程的start（）方法；

​（2）就绪（runnable）：调用了开启线程的start（）方法，但是还没有获得到CPU的执行权，处于等待获取执行权状态；

​（3）运行（running）：已经获得到了CPU执行权，并处于正常执行状态；

​（4）阻塞（blocked）：当一个线程试图获取一个对象锁，而该对象锁被其他的线程持有，则该线程进入Blocked状态；当该线程持有锁时，该线程将变成Runnable状态；

​（5）等待（waiting）：造成线程等待的原因有两种，分别是调用Object.wait()、join()方法。处于等待状态的线程，正在等待其他线程去执行一个特定的操作。例如：因为wait()而等待的线程正在等待另一个线程去调用notify()或notifyAll()；一个因为join()而等待的线程正在等待另一个线程结束；

（6）计时等待（timed\_waiting）：一个在限定时间内等待的线程的状态。也称之为限时等待状态。造成线程限时等待状态的原因有三种，分别是：Thread.sleep(long)，Object.wait(long)、join(long)；

​（7）死亡（dead）：线程run()、main() 方法执行结束，或者因异常退出了run()方法，则该线程结束生命周期。死亡的线程不可再次复生。

## 3、run()和start()方法有哪些区别？

run（）方法只是一个封装任务的普通方法，调用run（）并不会开启线程，start（）方法会开启线程，底层会调用run（）方法；

## 4、实现线程间通讯的方法有哪些？

方式一：使用 volatile 关键字

基于 volatile 关键字来实现线程间相互通信是使用共享内存的思想，大致意思就是多个线程同时[监听](https://so.csdn.net/so/search?q=%E7%9B%91%E5%90%AC&spm=1001.2101.3001.7020" \t "_blank)一个变量，当这个变量发生变化的时候，线程能够感知并执行相应的业务。这也是最简单的一种实现方式

方式二：使用Object类的wait() 和 notify() 方法

众所周知，Object类提供了线程间通信的方法：wait()、notify()、notifyaAll()，它们是[多线程](https://so.csdn.net/so/search?q=%E5%A4%9A%E7%BA%BF%E7%A8%8B&spm=1001.2101.3001.7020" \o "多线程)通信的基础，而这种实现方式的思想自然是线程间通信。

注意： wait和 notify必须配合synchronized使用，wait方法释放锁，notify方法不释放锁

方式三：使用JUC工具类 CountDownLatch

jdk1.5之后在java.util.concurrent包下提供了很多并发编程相关的工具类，简化了我们的并发编程代码的书写，\*\*\*CountDownLatch\*\*\*基于AQS框架，相当于也是维护了一个线程间共享变量state

方式四：基于LockSupport实现线程间的阻塞和唤醒，LockSupport 是一种非常灵活的实现线程间阻塞和唤醒的工具，使用它不用关注是等待线程先进行还是唤醒线程先运行，但是得知道线程的名字。

## 5、wait、notify、notifyAll分别的作用是什么？可以用在同步代码块之外吗？为什么？

Wait: 导致当前的线程等待，直到其他线程调用此对象的notify( ) 方法或 notifyAll( ) 方法；

Notify: 唤醒在此对象监视器上等待的单个线程；

NotifyAll: 唤醒在此对象监视器上等待的所有线程；

wait( )，notify( )，notifyAll( )都不属于Thread类，而是属于Object基础类，也就是每个对象都有wait( )，notify( )，notifyAll( ) 的功能，因为每个对象都有锁，锁是每个对象的基础，当然操作锁的方法也是最基础的。

当需要调用以上的方法的时候，一定要对竞争资源进行加锁，如果不加锁的话，则会报IllegalMonitorStateException 异常。

wait/notify/notifyAll方法要放在synchronized里面，除此之外，还要非常非常注意一个重点：调用wait/notify/notifyAll方法的对象，必须要和synchronized的对象锁是一致的。

## 6、Sleep和Wait的区别？

1、调用对象不同：wait()、notify()、notifyAll()等都是 Object 上的方法，任何对象都可以作为锁对象进行调用；

​ sleep()、join()、yield()、interrupted()等都是属于Thread类的静态方法；

​ 2、是否释放锁：sleep()方法不会释放锁，只让出了CPU执行权，但是wait()会释放锁，而且会加入到等待队列中。

​ 3、使用位置：sleep()方法可以在任何地方使用；wait()方法则只能在synchronized同步方法或同步块中使用；

​ 4、重运行机制：sleep()过了指定睡眠时间从阻塞状态自动回到可运行状态，wait()方法需要手动调用notify()或者notifyAll()方法手动进行唤醒，然后进入锁池等待。

## 7、什么是线程安全问题？什么情况下会产生？如何解决？

就是多线程环境中，且存在数据共享 , 一个线程访问的共享数据被其他线程修改了, 那么就发生了[线程安全](https://so.csdn.net/so/search?q=%E7%BA%BF%E7%A8%8B%E5%AE%89%E5%85%A8&spm=1001.2101.3001.7020" \t "_blank)问题

（1）什么情况下会出现多线程安全问题？

1.多线程环境；

2.有共享数据；

3.有多条语句操作共享数据。

（2）如何解决多线程安全问题？

1.采用synchronized锁锁住同步代码块，即把多条语句操作共享数据的代码给锁起来，让任意时刻只能有一个线程执行即可；

2.采用lock锁锁住同步代码块。

3.多个线程必须使用同一个锁对象。

## 8、什么是死锁？如何防止产生死锁？

当资源有限且出现了同步嵌套的情况时会出现死锁现象，线程死锁是指由于两个或者多个线程互相持有对方所需要的资源，导致这些线程处于等待状态，无法前往执行。

一次性申请所有的资源，或者避免同步嵌套可以防止产生死锁。

## 9、Synchronized关键字的底层原理是什么?

当线程执行到monitorenter时，代表即将进入到同步代码块中，线程首先需要去获得Object的对象锁，而对象锁处于每个java对象的对象头中，对象头中会有一个锁的计数器，当线程查询对象头中计数器，发现内容为0时，则代表该对象没有被任何线程所占有，此时该线程可以占有此对象，计数器于是加1。

线程占有该对象后，也就是拿到该对象的锁，可以执行同步语句块里面的方法。此时，如果有其他线程进来，查询对象头发现计数器不为0，于是进入该对象的锁等待队列中，一直阻塞到计数器为0时，方可继续执行。

第一个线程执行到monitorexit后，释放了Object的对象锁，此时第二个线程可以继续执行。

同步方法（字节码中看不出来需要看JVM底层实现）依靠的是方法修饰符上的ACC\_SYNCHRONIZED实现。

这样，synchronized就可以保证方法或者代码块在运行时，同一时刻只有一个方法可以进入到临界区，同时它还可以保证共享变量的内存可见性。Java 中每一个对象都可以作为锁，这是 synchronized 实现同步的基础：

## 10、Synchronized可以用在哪些地方？分别的锁对象是什么？

Java 中每一个对象都可以作为锁，这是 synchronized 实现同步的基础：

用于同步代码块，锁是括号里面的对象

用于普通同步方法，锁是当前实例对象（this）

用于静态同步方法，锁是当前类的class 对象

## 11、Synchronized和JUC下Lock锁的异同？

1）、Lock是JUC包下提供的封装好的锁，是类的概念，而synchronized是一个虚拟机层面的关键字。

​ 2）、Lock显示的加锁和解锁，且解锁要在finally代码块中，否则可能会死锁，而synchronized为隐式的上锁和解锁。

​ 3）、Lock锁提供了尝试获取锁和设置获取锁时间的机制，可返回取锁状态，当获取不到锁的时候也可以选择放弃取锁，而synchronized无法判断返回取锁状态，取锁不成功只能阻塞，没有Lock灵活。

​ 4）、Lock锁阻塞可被打断，而synchronized阻塞不可被打断。

​ 5）、Lock实现可重入、可公平锁，而synchronized是可重入、非公平锁。

​ 6）、Lock可以很灵活的根据线程角色类型去创建Condition监视器对象，调用await()、signal()、signalAll()进行线程通讯调度，而synchronized使用Object

​ 对象本身作为监视器对象去调用wait() 、notify()、notifyAll()进行线程通讯调度。

​ 7）、Lock提供了更丰富的锁分类，如读锁、写锁，可以更细粒度的关注线程安全问题。

## 12、Synchronized在JDK1.6做了什么优化？

自适应的CAS自旋、锁消除、锁粗化、偏向锁、轻量级锁等技术。

锁可以升级不可降级，即 无锁 -> 偏向锁 -> 轻量级锁 -> 重量级锁是单向的。 这种策略是为了提高获得锁和释放锁的效率

自适应自旋锁：根据上一次的自旋操作动态调节自旋次数，如果成功自旋，则允许自旋次数更多；如果自旋操作失败，则减少自旋的次数或忽略自旋过程，以节省资源。

锁消除：JVM检测是否不存在数据竞争，不存在则去除锁操作

锁粗化：对同一对象重复使用锁时，将锁的范围扩大

轻量级锁：使用CAS操作代替操作系统互斥量实现锁

偏向锁：减少轻量级锁的CAS操作消耗

## 13、Synchronized是公平锁还是非公平锁？获取不到锁时会阻塞吗？

Synchronized是非公平锁，获取不到锁时会阻塞。

## 14、同步代码块中执行完wait/notify/notifyAll后会立马释放锁吗？

当线程执行wait()方法时候，会释放当前的锁（sleep不会释放锁，只是让出CPU执行权），然后让出CPU执行权，进入等待状态。

只有当 notify/notifyAll() 被执行时候，才会唤醒一个或多个正处于等待状态的线程，然后继续往下执行，直到执行完synchronized 同步代码块的代码或是中途遇到wait() ，再次释放锁。

也就是说，notify/notifyAll() 的执行只是唤醒沉睡的线程，而不会立即释放锁。

## 15、Lock锁有哪些实现？分别的特点是什么？

ReentrantLock（可重入锁）：可重入，可中断，公平锁

ReentrantReadWriteLock（可重入的读写锁）：可重入，互斥性与锁升降级：读读不互斥，读写&写写互斥，公平锁

StampedLock：StampedLock是比ReentrantReadWriteLock更快的一种锁，支持乐观读、悲观读锁和写锁。和ReentrantReadWriteLock不同的是，StampedLock支持多个线程申请乐观读的同时，还允许一个线程申请写锁

## 16、JUC下Lock的监视器对象是哪个类？与Synchronized的监视器有什么异同？

Lock可以很灵活的根据线程角色类型去创建Condition监视器对象，调用await()、signal()、signalAll()进行线程通讯调度，而synchronized使用Object

​ 对象本身作为监视器对象去调用wait() 、notify()、notifyAll()进行线程通讯调度。

## 17、什么是线程可重入？Synchronized具备吗？Lock呢？

简单来说就是：同一线程在调用自己类中其他 synchronized 方法/块或调用父类的 synchronized 方法/块都不会阻碍该线程的执行，就是说同一线程对同一个对象锁是可重入的。Synchronized和Lock都具备可重入的特性。

重入锁实现可重入性原理或机制是：每一个锁关联一个线程持有者和计数器，当计数器为 0 时表示该锁没有被任何线程持有，那么任何线程都可能获得该锁而调用相应的方法；当某一线程请求成功后，JVM会记下锁的持有线程，并且将计数器置为 1；此时其它线程请求该锁，则必须等待；而该持有锁的线程如果再次请求这个锁，就可以再次拿到这个锁，同时计数器会递增；当线程退出同步代码块时，计数器会递减，如果计数器为 0，则释放该锁。（AQS）

## 18、什么是AQS？

java.util.concurrent.locks.AbstractQueuedSynchronizer 抽象类，简称 AQS ，是一个用于构建锁和同步容器的队列同步器，它是整个JUC包下Lock体系的核心，如ReentrantLock、ReentrantReadWriteLock、CountDownLatch、Semaphore都是基于它来实现的，它的核心构成部分为：使用一个先进先出的FIFO的队列存储排队等待锁的线程，使用一个用volatile修饰的int类型的state同步状态来记录​当前是否有线程持有锁，0表示没有线程获得锁，1表示有，上锁state就加1，释放锁就对应减1，有重入锁现象，这个值就大于1，然后需要逐级去释放。

## 19、什么是CAS？什么是CAS的ABA问题？如何解决？

CAS其实就是乐观锁的一种实现方式，CAS全称compare and swap——比较并替换，保证对数据更改的原子性，它是并发条件下修改数据的一种机制，包含三个操作数：

​ > 需要修改的数据的内存地址（V）；

​ > 对这个数据的旧预期值（A）；

​ > 需要将它修改为的值（B）；

​ CAS的操作步骤如下：

​ 1）、修改前记录数据的内存地址V；

​ 2）、读取数据的当前的值，记录为A；

​ 3）、需要修改值时查看地址V下的值是否仍然为A，若为A，则用B替换它；若地址V下的值不为A，表示在自己修改的过程中，其他的线程对数据进行

​ 了修改，则不更新变量的值，而是重新从步骤2开始执行，这被称为自旋；

​ CAS 贯穿于整个AQS体系，是AQS实现的基础

（1）优点

   CAS是一种乐观锁，其优点就是不需要加锁就能进行原子操作；

（2）缺点

    CAS的缺点主有三点：

​ 1）CAS机制只能用在对某一个变量进行原子操作，无法用来保证多个变量或语句的原子性（synchronized可以）；

​2）假设在修改数据的过程中经常与其他线程修改冲突，将导致需要多次的重新尝试；

3）有可能产生ABA问题；

ABA问题是指在CAS操作时，其他线程将变量值A改为了B，但是又被改回了A，等到本线程使用期望值A与当前变量进行比较时，发现变量A没有变，于是CAS就将A值进行了交换操作，但是实际上该值已经被其他线程改变过。

解决办法： 在变量前面加上版本号，每次变量更新的时候变量的版本号都+1，即A->B->A就变成了1A->2B->3A。只要变量被某一线程修改过，变量对应的版本号就会发生递增变化，从而解决了ABA问题。

  （3）适用场景

   由上面分析的优缺点可以看出，CAS适用于并发冲突发生频率较低的场合，而对于并发冲突较频繁的场合，CAS由于不断重试，反倒会降低效率。

​ 总结：CAS是一种在并发下实现原子操作的机制，但是只能用来保证一个变量的原子性，适用于并发冲突频率较低的场合。

## 20、你了解JUC下的哪些工具类，分别有什么作用？（CountdownLatch、Cyclicbarrier、Semaphore）

1）、CountdownLatch

​ 具有计数器的功能，构造方法初始化数量，通常调用await方法控制某个线程等待，其他线程执行完会countDown一次进行减1，直到countDown为0才会执行阻塞的线程

​ 值得注意的是CountDownLatch计数的次数一定要与构造器传入的数字一致，比如构造器传入的是3，则countDown()一定要执行3次，否则await的线程将一直阻塞。

​ CountDownLatch通常用来控制某个线程等待，让某组线程执行完才能执行调用await的线程。

2）、Cyclicbarrier

​ 类似CountdownLatch，区别是线程在countDown()之后，会继续执行自己的任务，而CyclicBarrier会在所有线程任务结束之后，才会进行后续任务。

区别：

CountDownLatch: 一个线程(或者多个)， 等待另外N个线程完成某个事情之后才能执行（学生考试交卷老师最后走）。 CyclicBarrier : N个线程相互等待，任何一个线程完成之前，所有的线程都必须等待（一局游戏大家都100%才开始）。

3）、Semaphore也是一种计数器，用来保护一个或者多个共享资源的访问。如果线程要访问一个资源就必须先获得信号量。如果信号量内部计数器大于0，信号量减1，然后允许共享这个资源；否则，如果信号量的计数器等于0，信号量将会把线程置入休眠直至计数器大于0，当信号量使用完时，必须释放。

​ Semaphore常用于控制某一共享资源访问的访问线程的最大数量。

## 21、说下volatile关键字，有什么作用？原理是什么？

1）、讲讲volatile

​ volatile 是java关键字，是一个变量类型修饰符，被voltile修饰的变量具有以下特性：

​ **①可见性**：保证了不同线程对这个变量进行操作时的可见性，即**一个线程修改**了某个**变量的值**，这**新值对其他线程来说**是**立即可见**的。

​ 实现(缓存共享协议)：

​ 对于用volatile修饰的变量，线程**写入本地内存中的同时**会将数据**立即刷新到主内存中**。其他线程读取该变量时，发现被volatile修饰，会**将本地变量值置为无效**，然后**从主内存中读取**。

​② **有序性**：禁止进行指令重排序。为提高执行效率，在不影响最终执行结果的前提下，代码在编译成字节码的时候有可能进行**指令重新排序**，这在**单线程情况下是没有问题的**，但是在**多线程的情况下会出现问题**。**volatile修饰的变量**则**可以避免这个问题**。

​ **③不保证原子性**：volatile 只能保证对单次读/写的原子性。i++ 这种操作不能保证原子性。关于volatile 原子性可以理解为把对volatile修饰变量的单个读/写，看成是使用同一个锁对这些单个读/写操作做了同步。

## 22、说下ThreadLocal，有什么作用？有哪些主要方法，实现原理是什么？为什么会有内存泄漏问题？如何解决？

（1）ThreadLocal是除了加锁这种同步方式之外的另一种可以规避出现多线程安全问题的思路。每个线程有一个与之绑定的容器：Map，这个Map中保存着当前线程的所有的本地变量，这个Map叫ThreadLocalMap，这个map的键为当前ThreadLocal对象，值为放进去的值

线程内部的存储类，赋予了线程存储数据的能力。

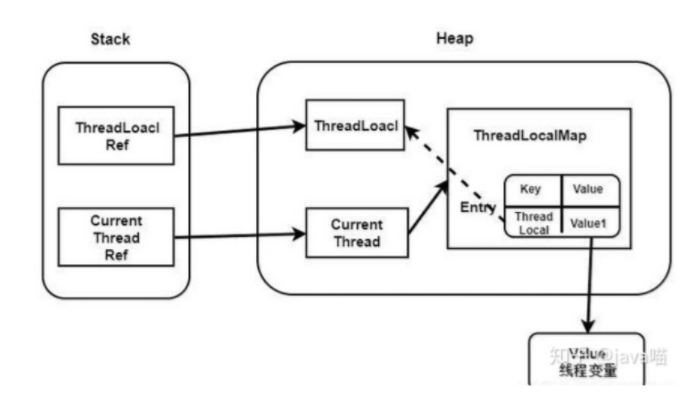
同一线程内调用get方法都可以从ThreadLocal中获取同一个对象。

多个线程中ThreadLocal数据相互隔离，从而规避了多线程操作同一数据产生的安全问题。

​（2） 主要方法有：get（）、set（）、remove（）

​

​（3） 可能造成的问题：内存泄漏：



​ ThreadLocalMap使用ThreadLocal的弱引用作为key，如果一个ThreadLocal不存在外部强引用时，Key(ThreadLocal)势必会被GC回收，这样就会导致ThreadLocalMap中key为null， 而value还存在着强引用，只有thead线程退出以后,value的强引用链条才会断掉。

但如果当前线程再迟迟不结束的话，这些key为null的Entry的value就会一直存在一条强引用链：

Thread Ref -> Thread -> ThreaLocalMap -> Entry -> value

这就会永远无法回收，造成内存泄漏。

（4）为什么ThreadLocalMap中的key使用弱引用而不是强引用：

如果key 使用强引用：当ThreadLocalMap的key为强引用，回收ThreadLocal时，因为ThreadLocalMap还持有ThreadLocal的强引用，如果没有手动删除，ThreadLocal不会被回收，导致Entry内存泄漏。

如果key使用弱引用：当ThreadLocalMap的key为弱引用，回收ThreadLocal时，由于ThreadLocalMap持有ThreadLocal的弱引用，即使没有手动删除，ThreadLocal也会被回收。当key为null，在下一次ThreadLocalMap调用set(),get()，remove()方法的时候会被清除value值。使用弱引用可以让GC回收方面多一层保障：弱引用ThreadLocal不会内存泄漏，对应的value会在下一次ThreadLocalMap调用set(),get(),remove()的时候会被清除。因此，ThreadLocal内存泄漏的根源是：由于ThreadLocalMap的生命周期跟Thread一样长，如果没有手动删除对应key就会导致内存泄漏，而不是因为弱引用。

但是，当key为null后，之后再也没有调用过set()、get()、remove()方法，这样就会使value一直不会被回收

（5）​解决：

每次使用完ThreadLocal都调用它的remove()方法清除数据；

## 23、说下线程池的几大核心参数？分别有什么作用？有几种默认的线程池？他们的7个核心参数为什么要那么设置？

（1）为什么用线程池：

1、使用线程池可以重复利用已有的线程继续执行任务，避免线程在创建销毁时造成的消耗

2、由于没有线程创建和销毁时的消耗，可以提高系统响应速度

3、通过线程池可以对线程进行合理的管理，根据系统的承受能力调整可运行线程数量的大小等。

（2）线程池包括以下四个基本组成部分：

1、**线程池管理器**（ThreadPool）：用于创建并管理线程池，包括创建线程池，销毁线程池，添加新任务；

​ 2、**工作线程**（PoolWorker）：线程池中线程，在没有任务时处于等待状态，可以循环的执行任务；

​ 3、**任务接口**（Task）：每个任务必须实现的接口，以供工作线程调度任务的执行，它主要规定了任务的入口，任务执行完后的收尾工作，任务的执行状态等；

​ 4、**任务队列**（taskQueue）：用于存放没有处理的任务。提供一种缓冲机制。

（3）默认的线程池：

1. newCachedThreadPool：

创建一个可进行缓存重复利用的线程池

2. newFixedThreadPool：

创建**固定大小**的线程池。每次提交一个任务就创建一个线程，直到线程达到线程池的最大数量,这时线程规模将不再变化。线程池的大小一旦达到最大值就会保持不变，如果某个线程因为执行异常而结束，那么线程池会补充一个新线程。

3. newSingleThreadExecutor：

​ 创建一个单线程的线程池，它创建单个工作线程来执行任务，如果这个线程异常结束，会创建一个新的来替代它。

​ 它的特点是：能确保依照任务在队列中的顺序来串行执行

4. newScheduledThreadPool：

创建一个线程池，它可安排在给定延迟后运行命令或者定期的执行

（4）创建线程的7个参数

1、corePoolSize 参数，核心线程数，当线程数 < corePoolSize ，会创建线程执行任务。

​ 2、maximumPoolSize参数，最大线程数， 当线程数 >= corePoolSize 的时候，会把任务放入workQueue队列中。

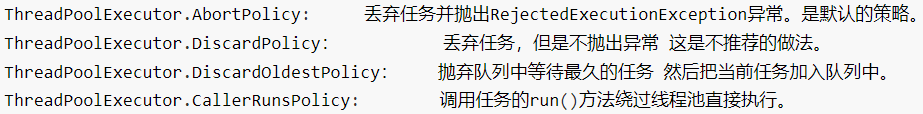
​ 3、keepAliveTime 参数，保持存活时间，当线程数大于 corePoolSize 的空闲线程能保持的最大时间。

​ 4、unit 参数，参数3的时间单位。

​ 5、workQueue参数，保存任务的阻塞队列。

​ 6、threadFactory 参数，创建线程的工厂。

​ 7、handler参数：当到达最大线程数量后执行的拒绝策略



## 24、单例模式写法有哪几种？（懒汉和饿汉式）那么懒汉式中保证线程安全的写法是什么？为什么要用双重检查模式？

单例模式有饿汉式单例模式、懒汉式单例模式和双检锁单例模式三种。

饿汉式：一开始就初始化，线程安全。

懒汉式：延迟初始化，非线程安全。

双检锁：延迟初始化，线程安全，既解决线程安全问题，又解决懒加载问题，同时保证了效率。

# 六、Redis & 多级缓存

## 1、Redis是一个什么样的数据库？读写速度怎么样？

Redis是一个用C语言开发的，开源的，基于内存的高性能键值对key-value结构非关系型数据库，支持事务与持久化（redis事务是一次执行多条命令，其中任意失败了其余的也会执行，没有回滚，没有原子性）。基于内存存储，读写性能高，官方提供的数据是可以达到100000+的QPS（每秒内查询次数）。它存储的value类型比较丰富，也被称为结构化的NoSql数据库，适合存储缓存、验证码、热点数据（热点商品、资讯、新闻）。

## 2、Redis有哪些数据类型，分别的特点？在你们项目中常见的应用场景有哪些？请列举。

Redis存储的是key-value结构的数据，其中key是字符串类型，value有5种常用的数据类型：

字符串 string---普通字符串，适用于存验证码，例如token、UUID、setnx分布式锁

哈希 hash---存储对象，例如用户互动数据（点赞数、喜欢数、关注数等）、首页菜品数据、热点数据

列表 list---简单的string字符串列表，按插入顺序排序，可重复 如最新的回复

集合 set---string类型的无序集合，且唯一有交集并集实现共同好友、共同关注 等等

有序集合 sorted set / zset---string 类型元素的有序集合，且不允许重复的成员，按分数排序，例如热点文章排行榜

## 3、Redis的持久化机制是什么样的？

### 1）、什么是RDB？RDB的持久化机制（save、bgsave）?

RDB即Redis的内存快照，默认持久化机制，它是在某一个时间点将Redis的内存数据全量写入一个临时文件，当写入完成后，用该临时文件替换上一次持久化生成的文件，这样就完成了一次持久化过程，默认的文件名为dump.rdb。

（1）save触发：该命令会阻塞当前Redis服务器，执行save命令期间，Redis不能处理其他命令，直到RDB过程完成为止。

（2）bgsave触发：执行该命令时，Redis会在后台异步fork出一个子线程进行快照操作，快照同时还可以响应客户端请求。

RDB原理：开始bgsave后，Redis会后台异步fork出一个子线程，Redis主进程通过一个页表与物理内存进行关系映射，进行数据的读写，fork的子线程会复制该页表，与物理内存进行同样的关系映射，对当前的物理内存内的数据进行快照操作，子线程快照后写为新的RDB二进制数据文件存在磁盘持久化，在快照期间，共享数据只能读，主线程要对数据进写操作则会在物理内存中将要写的数据先复制一份，对复制的数据进行写操作，映射关系也会转移到复制的数据。

### 2）、触发RDB的时间点是什么？

save：Redis停机时会执行一次save命令，实现RDB持久化；

bgsave：自动触发是由我们的配置文件来完成的。在redis.conf配置文件可以配置触发Redis的RDB持久化条件，也就是什么时候将内存中的数据保存到硬盘。比如“save m n”。表示m秒内数据修改n次时，自动触发bgsave。

### 3）、RDB的优缺点分别有哪些？

（1）优点：

​ ①由于RDB文件是一个非常紧凑的二进制文件，所以加载的速度会快于 AOF 方式；

​ ②fork子进程方式，除了fork线程阶段，其他时候不会阻塞；

​ ③RDB文件代表着 Redis 服务器的某一个时刻的全量数据，所以它非常适合做冷备份和全量复制的场景；

​（2）缺点：

​ 没办法做到实时持久化，会存在丢数据的风险。定时执行持久化过程，如果在这个过程中服务器崩溃了，则会导致这段时间的数据全部丢失。

### 4）、什么是AOF？分为几个阶段？

AOF，即append only file，它是将每一行对Redis数据进行修改的命令以独立日志的方式存储起来。由于Redis是将“操作 + 数据” 以格式化的方式保存在日志文件中，他代表了这段时间所有对 Redis 数据的操作过程，所以在数据恢复时，我们可以直接replay该日志文件，即可还原所有操作过程，达到恢复数据的目的。它的主要目的是解决数据持久化的实时性问题，保证数据的安全性。

AOF总共分为三个阶段：

（1）第一阶段是命令写入：将命令写入缓冲区；

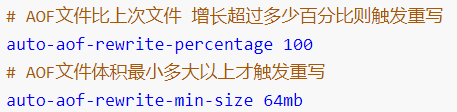
（2）第二阶段是文件同步：命令写入到缓冲区后，根据不同的策略刷到硬盘中。Redis提供了三种不同的同步策略：



​ （3）第三阶段是文件重写：随着命令的不断写入，AOF文件会越来越庞大，直接的影响就是导致“数据恢复”时间延长，而且有些历史的操作是可以废弃的（比如超时、del等等），为了解决这些问题，Redis 提供了 “文件重写”功能，该功能有手动和自动两种方式触发。

手动方式：通过执行bgrewriteaof命令进行AOF文件重写

自动方式：Redis也会在触发阈值时自动去重写AOF文件。阈值也可以在redis.conf中配置



重写AOF主要做了以下事情：

​ 1、已过期的数据不再写入文件。

​ 2、保留最终命令。例如 set key1 value1 、set key1 value2、....set key1 valuen，类似于这样的命令，只需要保留最后一个即可。

​ 3、删除无用的命令。例如 set key1 valuel;del key1,这样的命令也是可以不用写入文件中的。

​ 4、多条命令合并成一条命令。例如 lpush list a、lpush list b、lpush list c，可以转化为 lpush list a b c

### 5）、AOF命令同步至日志文件分为哪几种？

分为三种策略

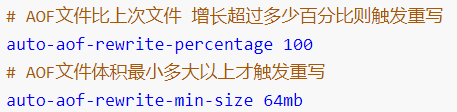


### 6）、AOF文件重写时间点？重写做了什么？

AOF重写有手动和自动两种方式触发。

手动方式：通过执行bgrewriteaof命令进行AOF文件重写

自动方式：Redis也会在触发阈值时自动去重写AOF文件。阈值也可以在redis.conf中配置



重写AOF主要做了以下事情：

​ 1、已过期的数据不再写入文件。

​ 2、保留最终命令。例如 set key1 value1 、set key1 value2、....set key1 valuen，类似于这样的命令，只需要保留最后一个即可。

​ 3、删除无用的命令。例如 set key1 valuel;del key1,这样的命令也是可以不用写入文件中的。

​ 4、多条命令合并成一条命令。例如 lpush list a、lpush list b、lpush list c，可以转化为 lpush list a b c

### 7）、AOF的优缺点分别有哪些？

（1）优点

​ ① 相比于RDB，AOF更加安全，默认同步策略为everysec即每秒同步一次，所以顶多我们就失去一秒的数据；

​ ②根据关注点不同，AOF提供了不同的同步策略，我们可以根据自己的需求来选择；

​ ③ AOF文件是以append-only方式写入，相比于RDB全量写入的方式，它没有任何磁盘寻址的开销，写入性能非常高；

​ （2）缺点

​ ① 由于AOF日志文件是命令级别的，所以相比于 RDB 紧致的二进制文件而言加载速度会慢些。

​ ② AOF开启后，支持的写 QPS 会比 RDB 支持的写 QPS 低。

### 8）、AOF和RDB同时开启会优先使用哪种进行数据恢复？

AOF与RDB同时开启会优先使用AOF进行数据恢复。

### 9）、RDB-AOF混合模式是什么？优点有哪些？

​ Redis 4.0 推出了RDB-AOF混合持久化方案，该方案是在AOF重写阶段创建一个同时包含 RDB 数据和 AOF 数据的 AOF 文件，其中 RDB 数据位于AOF 文件的开头，它存储了服务器开始执行重写操作时 Redis 服务器的数据状态（RDB 快照方案），重写操作执行之后的 Redis 命令，则会继续 append 在 AOF 文件末尾，一般这部分数据都会比较小。这样在 Redis 重启的时候，则可以先加载 RDB 的内容，然后再加载 AOF 的日志内容，这样重启的效率则会得到很大的提升，而且由于在运行阶段 Redis 命令都会以 append 的方式写入 AOF 文件，保证了数据的实时性和安全性。

## 4、Redis主从结构能解决什么问题？

单节点Redis的并发能力是有上限的，要进一步提高Redis的并发能力，就需要搭建主从集群，实现读写分离，master节点进行数据的写操作，slave节点进行数据的读操作，提高Redis的可用性及稳定性。

## 5、Redis主从同步具体流程是什么？

1）、建立连接：

①slave连接master，发送slaveof ip port命令；

②master响应返回它的ip及端口；

③slave保存master的ip及端口，异步建立socket连接；

④slave周期性的发送ping请求，验证socket连接是否可用，判断master是否可以处理请求；

⑤master响应pong；

⑥slave进行权限验证（如果master有密码）；

⑦slave发送其端口号，master保存slave的端口号。

2）、数据同步

主从第一次建立连接时执行全量同步（①第一次建立连接；②主节点非安全关闭导致replid改变再次连接时；③从节点所需要同步数据的偏移量 offset 不在复制积压缓冲区中，也会进行全量同步）：

① slave携带replid与offset偏移量请求数据同步；

②master判断是否为第一次同步，是第一次，返回master的数据版本信息，即replid与offset偏移量，slave保存主节点版本信息；

③master执行bgsave，生成RDB文件，将RDB文件发送给slave，同时记录RDB期间的操作命令至复制积压缓冲区；

④slave清空本地数据，加载RDB文件，执行复制积压缓冲区的命令，同步数据；

如何避免全量同步？（全量同步非常耗时）

①适当提高repl\_baklog（复制积压缓冲区）的大小；②安全重启使replid不发生变化；③发现slave宕机时尽快实现故障恢复，尽可能避免全量同步。

增量同步（用于从节点短暂中断的情况（网络中断、短暂的服务宕机））：

①slave携带replid与offset偏移量请求数据同步；

②master判断是否为第一次同步，不是第一次，回复continue；

③master从复制积压缓冲区获取offset之后的数据，发送给slave；

④slave执行复制积压缓冲区的命令，同步数据；

3）、命令传播

slave执行接收到的命令，保持与master之间的同步。

## 6、如何优化主从同步效率？

（1）从尽量避免全量同步的方面入手；（①适当提高repl\_baklog（复制积压缓冲区）的大小；②安全重启使replid不发生变化；③发现slave宕机时尽快实现故障恢复，尽可能避免全量同步。）

（2）在master中配置repl-diskless-sync yes启用无磁盘复制，避免全量同步时的磁盘IO；

（3）Redis单节点上的内存占用不要太大，减少RDB导致的过多磁盘IO；

（4）限制一个master上的slave节点数量，如果实在是太多slave，则可以采用主-从-链式结构，减少master压力。

## 7、Redis的故障恢复依靠什么机制？哨兵机制的主要工作范围、工作流程和作用？

哨兵机制

作用范围：对Redis集群的所有实例发挥作用

（1）监控：3个心跳

①10秒一次：更新redis集群最新的一个拓补情况

②2秒一次：用于Sentinel哨兵节点之间交流对redis数据节点的看法

③1秒一次：用于发出ping，检测redis节点是否还存活（主观下线：如果某sentinel节点发现某实例未在规定时间响应，则认为该实例主观下线。客观下线：若超过指定数量（quorum）的sentinel都认为该实例主观下线，则该实例客观下线。quorum值最好超过Sentinel实例数量的一半）

（2）通知：用于master发生了变动，将变化推送给客户端

（3）故障转移：当对主节点进行客观下线后，选举出一个从节点（

①首先会判断slave节点与master节点断开时间长短，如果超过指定值（down-after-milliseconds \* 10）则会排除该slave节点

②然后判断slave节点的slave-priority值，越小优先级越高，如果是0则永不参与选举

③如果slave-prority一样，则判断slave节点的offset值，越大说明数据越新，优先级越高

④最后是判断slave节点的运行id大小，越小优先级越高）

选举后发送slaveof no one命令使其成为新的主节点，并且给其他从节点发送slaveof ip port命令连接上新的主节点，最后将原来的master标记为从节点。

## 8、什么是缓存雪崩？缓存击穿？缓存穿透？分别如何解决？什么是缓存预热？

（1）缓存雪崩：

缓存雪崩是指缓存中数据大批量到过期时间，而查询数据量巨大，引起数据库压力过大甚至宕机。和缓存击穿不同的是，缓存击穿指并发查同一条数据，缓存雪崩是大批量数据都过期了，大量数据都从redis中查不到，从而查数据库，造成数据库压力过大。

解决方案：

1. 设置缓存数据的过期时间设置随机，避免同一时间出现大量数据过期现象发生；

​②如果缓存数据库是分布式部署，可以将热点数据均匀分布在不同的缓存数据库中；

​③允许的话，设置热点数据永远不过期；

​④要保证redis的高可用，可以使用主从+哨兵或redis cluster集群，避免服务器不可用；

​⑤使用redis的持久化RDB+AOF组合策略，防止缓存丢失并且可以快速恢复数据。

（2）缓存击穿：

缓存击穿是指缓存中没有但数据库中有的数据（一般是缓存时间到期），这时由于并发用户特别多，同时读缓存没读到的某一个数据，又同时去数据库去取数据，引起数据库压力瞬间增大，造成过大压力。

解决方案：

①设置热点数据不过期；

​②分布式锁，第一时间去数据库获取数据填充到redis中，但是这个过程需要加分布式锁，防止所有线程都去读取数据库，一旦有一个线程去数据库获取数据了，其他线程取锁失败后可设置一个合理睡眠时间之后再尝试去redis中获取数据；

（3）缓存穿透：

缓存穿透是指缓存和数据库中都没有的数据，而用户不断发起请求，如发起为id为“-1”的数据或id为特别大不存在的数据。这时的用户很可能是攻击者，攻击会导致数据库压力过大。

解决方案：

①接口层增加校验，如用户鉴权或者白名单校验，id做基础校验，id<=0的直接拦截；

​②从缓存取不到的数据，在数据库中也没有取到，这时也可以将key-value对写为key-null，缓存有效时间可以设置短点，如30秒（设置太长会导致正常情况也没法使用）。这样可以防止攻击用户反复用同一个id暴力攻击；

​③引入布隆过滤器，过滤一些异常的请求。

## 9、Redis是单线程的，为什么读写效率还那么高？（6.0版本开始伪多线程）

Redis完全基于内存​，数据结构简单，采用单线程，避免了加锁、释放锁、死锁、线程间上下文切换等消耗，使用多路I/O复用模型，非阻塞IO（NIO）。

## 10、Redis的线程模型是什么样的？（典型的NIO，非阻塞式IO）

Redis的线程模型是基于非阻塞的IO多路复用机制的线程模型，单线程Redis 是基于reactor模式开发了网络事件处理器，这个处理器叫做文件事件处理器（file event handler）。由于这个文件事件处理器是单线程的，所以Redis才叫做单线程的模型。采用IO多路复用机制同时监听多个Socket，根据Socket上的事件来选择对应的事件处理器来处理这个事件。

## 11、Redis过期数据的删除策略是什么？有哪些？

Redis中可以设置键的过期时间，到期后自动进行删除，有如下三种删除策略：

（1）定时过期：每个设置过期时间的key都需要创建一个定时器，到过期时间就会立即清除。该策略可以立即清除过期的数据，对内存很友好；但是会占用大量的CPU资源去处理过期的数据，从而影响缓存的响应时间和吞吐量。

（2）惰性过期：只有当访问一个key时，才会判断该key是否已过期，过期则清除。该策略可以最大化地节省CPU资源，却对内存非常不友好。极端情况可能出现大量的过期key没有再次被访问，不会被清除，则占用大量内存。

（3）定期过期：每隔一定的时间（750ms），花250ms去扫描一定数量的数据库的expires字典中一定数量的key，即每1秒扫描一次，并清除其中已过期的key。如果本次扫描删除超过75%，会再一次进行扫描这个库。该策略是前两者的一个折中方案。通过调整定时扫描的时间间隔和每次扫描的限定耗时，可以在不同情况下使得CPU和内存资源达到最优的平衡效果。

## 12、Redis的数据淘汰策略是什么？有哪些？

当新数据进入redis时，如果内存不足，是下面这样一个流程：

1、首先执行存储命令前会调用freeMemoryIfNeeded()检测内存是否充足。

2、如果内存不满足新加入数据的最低存储要求，redis要临时删除一些数据为当前指令清理存储空间，清理数据的策略称为逐出算法。

3、逐出算法（maxmemory-policy）分为下面八种，其中lru是使用时间最短，离当前时间最久，lfu是使用的次数最少。



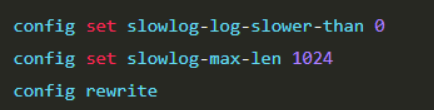
## 13、Redis的慢查询如何排查？

（1）Redis慢查询可通过配置两个参数进行：

slowlog-log-slower-than：设置慢查询预设的超时阈值，单位是微秒，默认是10毫秒，一般都会调低，若设置 `slowlog-log-slower-than = 0`，则会记录所有命令，若设置 `slowlog-log-slower-than < 0`，则不会记录任何命令

slowlog-max-len：表示慢查询日志存储的条数，一般设置1000以上，可以减缓慢查询被剔除的可能

（2）Redis 中有两种修改配置的方法，一种是修改配置文件，另一种是使用 config set 命令动态修改



（3）Redis 会使用一个列表来存储慢查询日志，slowlog-max-len 就是该列表的最大长度。一个命令如果满足慢查询阈值条件则会加入到该列表来，但是如果该列表已经处于最大长度时，那么会删除最开始的一条记录，然后将最新的命令插入到末尾，所以慢查询日志列表是一个有限的先进先出列表。

​ 通过slowlog get [n]命令获取慢查询日志：



返回的慢查询日志由 4 个属性组成：1、日志的标识 id 2、发生的时间戳 3、命令耗时 4、执行的命令和参数。

## 14、如何正确使用Redis的分布式事务锁？（Zookeeper也可以实现分布式锁）

通过setnx上锁方式实现，但是不注意写法很可能会出现很多问题；

​ （1）错误用法：先通过setnx上锁，再通过expire设置过期时间，最后执行完任务后手动del释放锁；

​ （2）场景一问题（死锁）：通过setnx上锁后出现异常，导致无法去expire设置锁的过期时间，更无法最后去手动释放锁，造成死锁！

​ 解决：使用上锁最新写法，保证上锁、设置过期时间一步完成的原子性：set(lockKey,value,nx,ex,expireTime);

​（3） 场景二问题（误删锁）：A机器中上锁并设置过期时间完成以后，系统出现了阻塞，导致锁到了过期时间并自动删除了，这时还没有执行手动释放锁的操作，这个时候B机器上锁成功，并去执行任务，任务还未执行完，A机器反应过来了，继续执行了手动释放锁的操作，把B机器上的锁给误删了。

​ 解决：上锁同时加上一个锁id，如当前线程ID，将锁id存入value值并记录在变量中，手动释放锁的时候比较一下value中的锁id跟变量中id是否一致，也就是判断一下是否自己还在持有锁，如果不是，就不执行删除操作了。

​ （4）场景三问题（误删锁）：这种误删锁是基于场景2判断锁id和释放锁操作这两步没有保证原子性所导致的。

具体为：A机器带锁id方式取锁、设置过期时间并执行完任务后，希望通过判断比较锁id之后去释放锁，判断通过后系统出现阻塞，阻塞到锁也到了过期时间自动释放了锁，这时还未进行手动释放锁操作，这个时候B机器上锁成功，并去执行任务，任务还未执行完，A机器反应过来了，继续执行了手动释放锁的操作，把B机器上的锁给误删了。

​ 解决：保证判断锁和释放锁的原子性：使用redis执行LUA脚本，保证一步执行判断锁和释放锁。

​ （5）场景四问题（锁续命）：这种场景是指线程中任务还没执行完，锁就已经到过期时间，这种情况可以给任务执行线程添加守护线程，守护线程负责对锁的expire时间进行监控，每当到过期前一秒就对过期进行判断，如果任务还在进行且锁马上过期，则对过期时间重新进行设置。

记住：

​ 综上：正确使用redis分布式事务锁需要保证两个原子性：

​ 1、上锁和设置过期时间需要保证原子性；

​ 2、 判断锁ID是否为自己所有和解锁需要保证原子性；

使用redis客户端Redisson都能够得到很好的解决，redisson内部已经实现了上述问题的解决机制，原理同上，底层基于JUC的lock锁。

## 15、Redis的双写一致性如何保证？

为了保证Redis缓存与数据库的双写一致性，可以考虑

（1）延时双删---先删缓存，再写数据库

①先删除缓存②再写数据库③此时休眠1秒（根据业务自行配置，通常几百毫秒），再次删除缓存。

这么做，可以将设定时间内所造成的缓存脏数据，再次删除。确保读请求结束，写请求可以删除读请求造成的缓存脏数据。

（2）使用先写数据库，再删缓存的方案， 当出现如下情况会出现缓存与数据库数据不一致：

1）缓存过期时间到了，自动失效。

2）请求b查询缓存，发缓存中没有数据，查询数据库的旧值，但由于网络原因卡顿了，没有来得及更新缓存。

3）请求a先写数据库，接着删除了缓存。

4）请求b更新旧值到缓存中。

该方案虽说不能100%避免缓存和数据库的数据不一致问题，但出现该问题的概率，相对于其他方案来说是最小的，需要同时满足以下条件才可以：

1）缓存刚好自动失效。

2）请求b从数据库查出旧值，更新缓存的耗时，比请求a写数据库，并且删除缓存的还长

通常写数据库的时间是要比查询数据库的时间长的，在该方案中，如果删除缓存失败了，可以加MQ异步重试机制或者采用mysql 结合canal中间件监听binlog，当数据库发生更新，发起MQ消息进行缓存删除进而解决。

## 16、项目搭建多级缓存的好处是什么？实现多级缓存的流程是什么？（加分项）

多级缓存就是充分利用请求处理的每个环节，分别添加缓存，减轻服务器压力，提升服务性能。

1）实现多级缓存的流程

第一层Nginx缓存、第二层Redis缓存、第三层JVM内存缓存，都没有就会去查数据库

## 17、Redis的hash槽一共有多少个？数据是如何进行入槽的？如何实现动态扩容？

Redis 集群中内置了 16384 个哈希槽，当需要在 Redis 集群中放置一个 key-value时，redis 先对 key 使用 crc16 算法算出一个结果，然后把结果对 16384 求余数，这样每个 key 都会对应一个编号在 0-16383 之间的哈希槽，redis 会根据节点数量大致均等的将哈希槽映射到不同的节点。

动态扩容：redis 的动态扩容操作都是通过redis源码文件夹中的redis-trib.rb脚本文件来完成的，为redis集群动态增加节点(也就是添加的时候不影响正在运行的节点), 需要先主后从, slot槽迁移在添加主节点之后；①首先添加主节点到集群，②然后重新分片, 为新节点迁移slot槽，此时需要指定新的节点中迁移的slot数量以及从哪些节点迁移，完成扩容。

# 七、Rabbitmq

## 1、Rabbitmq消息模式有哪些？你们用的哪种？（5种）

有五种消息模式，分别是简单队列模式、工作队列模式、发布订阅根据交换机类型不同分为三种：Fanout广播模式、Direct路由模式、Topic会话通配模式，我们主要采用的是Topic会话通配模式，这种模式的功能最为强大，可以对消息指定路由并使用\*（：代表1个词）或者#（代表0个或多个词）通配符进行Routing key的通配匹配

## 2、Rabbitmq如何保证mq消息可靠性？（3大方面）

（1）生产者方面：保证消息投递到MQ不丢：开启生产者消息确认机制

1）、开启ConfirmCallBack回调：消息投递至交换机的回调（成功、失败、异常）

2）、开启ReturnCallBack回调：消息从交换机路由到队列的回调

（2）MQ方面：保证消息到达MQ之后不丢（MQ宕机了也不影响）：消息持久化(springboot下三者默认就是持久化的)

1）、交换机的持久化（DirectExchange）

2）、队列的持久化（durable项）

3）、消息的持久化（指定delivery-mode）

（3）消费者方面：保证消费者不丢（必须至少消费一次）：开启消费者消息确认机制

1）、none：没有确认机制，如果出现异常也会直接丢弃

2）、manual：手动确认

①可以自己灵活的控制消息的确认和拒收

②注意：开启这个机制后一定要记得手动回执，否则将造成同一消息大量堆积问题

3）、auto：自动确认，由spring监测listener代码是否出现异常（rabbitmq默认的方式）

优点：出现异常返回nack，正常执行返回ack，无需编写业务代码、无侵入（原理：AOP）

缺点：当消费消息出现异常返回nack时会一直反复投递，解决：开启消费者重试机制（有一个最大重试次数）

消费者重试机制：消息消费失败，消费者会自己发起重试，重试间隔和最大重试次数可灵活配置

当超过最大重试次数时，消费者对于消息会有三种执行策略：

1、直接丢弃：该消息从MQ删除

2、放回队列：将消息放回至队列，然后投递给消费者，消费者进行重试，如此循环往复

3、投递至指定的交换机（常用）：将消息投递至专用的交换机，绑定队列后进行精准消费

## 3、Rabbitmq如何实现延时消息？（2种）

（1）、利用死信交换机+TTL机制实现延时投递

原理：消息超时未被消费会被投递至死信队列

设置超时：

1）、针对消息设置延时时间

2）、针对队列设置延时时间

（2）、利用延时交换机插件实现延时投递

原理：交换机基于内存存储消息，延时路由，到达指定时间会被路由至队列

## 4、什么是死信队列？什么样的消息会进入死信队列？

死信队列：与死信交换机绑定，专门用于存放死信的队列称为死信队列，以下三类垃圾消息会进入死信交换机，路由到与死信交换机绑定的死信队列：

1）、被拒绝或者nack（false）的消息

2）、队列放不下的消息

3）、超时过期的消息

## 5、Rabbitmq如何解决消息堆积问题？（3种思路）

（1）针对同一队列增加多个消费者进行消费

（2）针对同一消费者开启多个线程进行消费

（3）加大队列的存储容量：惰性队列

惰性队列特性：

①接收到消息后直接存入磁盘而非内存

②消费者要消费消息时才会从磁盘中读取并加载到内存

③支持数百万条的消息存储

优点：存储容量大，减少从内存刷消息数据至磁盘的次数，性能更稳定

缺点：性能取决于磁盘IO

## 6、如何保证消息的幂等性？（从业务层面进行判断）

幂等性：使用同样的条件，一次请求和重复多次请求对系统资源的影响是一致的

保证幂等性：给消息携带一个唯一性的业务id

## 7、如何保证消息顺序消费?

（1）让同一组消息有序的存入同一队列；

（2）同一队列只能有一个消费者进行消费；

（3）只有一个消费者如何保证消费速度？

1）可以起多个队列去存放这一类消息

2）消费者开启多个线程去消费消息。

# 八、微服务

## 1、概念问题

### 1）、什么是微服务？解决微服务各种问题都用了哪些组件？

微服务是一种经过良好架构设计的分布式架构方案，微服务的架构特征：

①单一职责：微服务拆分粒度更小，每一个服务都对应唯一的业务能力，做到单一职责

②自治：团队独立、技术独立、数据独立，独立部署和交付

③面向服务：服务提供统一标准的接口，与语言和技术无关

④隔离性强：服务调用做好隔离、容错、降级，避免出现级联问题

微服务常见的组件有：

①服务注册发现：Eureka、Nacos、Consul

②统一配置管理：SpringCloudConfig、Nacos

③服务远程调用：Openfeign、Dubbo

④统一网关路由：SpringCloudGateWay、Zuul

⑤服务链路监控：Zipkin、Sleuth

⑥流控、降级、保护：Hystrix、Sentinel

### 2）、什么是单体架构、什么是分布式架构、什么是微服务架构？

①单体架构：将业务的所有功能集中在一个项目中开发，打成一个包部署，适合小型项目，例如学生管理系统

优点：架构简单、部署成本低

缺点：耦合度高（维护困难、升级困难）、扩展性差

②分布式架构：根据业务功能对系统做拆分，每个业务功能模块作为独立项目开发，称为一个服务，适合大型互联网项目。

优点：降低服务耦合、有利于服务升级和拓展

缺点：服务调用关系错综复杂

③微服务架构：微服务是一种经过良好架构设计的分布式架构方案

优点：拆分粒度更小、服务更独立、耦合度更低

缺点：架构非常复杂，运维、监控、部署难度提高

### 3）、微服务有哪些特点？

微服务是一种经过良好架构设计的分布式架构方案，微服务的架构特征：

①单一职责：微服务拆分粒度更小，每一个服务都对应唯一的业务能力，做到单一职责

②自治：团队独立、技术独立、数据独立，独立部署和交付

③面向服务：服务提供统一标准的接口，与语言和技术无关

④隔离性强：服务调用做好隔离、容错、降级，避免出现级联问题

## 2、远程调用

### 1）、什么是Feign，用来做什么的？Feign底层调用是怎么实现的？底层协议是什么？优势是什么？

Feign是一个http请求调用的轻量级框架，其作用就是帮助我们优雅的实现http请求的发送，Feign会完全代理HTTP请求，我们只需要像调用方法一样调用它就可以完成服务请求及相关处理。Feign整合了Ribbon和Hystrix，可以让我们不再需要显式地使用这两个组件，解决RestTemplate发起远程调用的代码可读性差，编程体验不统一、参数复杂，URL难以维护的问题。

Feign底层调用：Spring Cloud应用在启动时，Feign会扫描标有@FeignClient注解的接口，生成代理，并注册到Spring容器中。生成代理时Feign会为每个接口方法创建一个RequestTemplate对象，该对象封装了HTTP请求需要的全部信息，请求参数名、请求方法等信息都是在这个过程中确定的，Feign的模板化就体现在这里。

### 2）、服务间调用，其中一个服务宕机了，这个时候怎么做呢？

如果采用的服务注册发现组件是eureka，首先服务启动后会注册到eureka注册中心，eureka会保存服务的地址及映射关系，当服务配置了集群时，调用其服务时会通过ribbon实现负载均衡来选择一个服务器，所有的服务每隔30s向eureka发送一次心跳，即一次请求，报告自己的状态，当服务宕机，超过一定时间没有发送心跳，eureka注册中心就会认为微服务故障了，从服务列表中剔除，从而排除故障的服务器。

最重要的是先抢通业务，而不是抢修故障的服务器，可以通过负载均衡，服务切换，或者启用应急流程，业务抢通后，定位服务器的故障原因，确定硬件故障还是软件故障后按相应流程进行处理修复。

### 3）、Ribbon是什么？负载均衡策略有哪些？底层原理是什么？默认是哪种？

Ribbon 是 Netflix 发布的SpringCloud中服务开源项目，依赖注册中心给客户端提供负载均衡, 也就是说Ribbon 是作用在消费者方的，简单来说, 它是一个客户端负载均衡器, 它会自动通过负载均衡算法去分配你要连接的机器，SpringCloud 认为 Ribbon 这种功能很好, 就对它进行了封装, 从而完成负载均衡。

SpringCloudRibbon的底层采用了一个拦截器，拦截了RestTemplate发出的请求，对地址做了修改，基本流程如下：

拦截我们的RestTemplate请求<http://userservice/user/1>

RibbonLoadBalancerClient会从请求url中获取服务名称，也就是user-service

DynamicServerListLoadBalancer根据user-service到eureka拉取服务列表

eureka返回列表，localhost:8081、localhost:8082

IRule利用内置负载均衡规则，从列表中选择一个，例如localhost:8081

RibbonLoadBalancerClient修改请求地址，用localhost:8081替代userservice，得到<http://localhost:8081/user/1>，发起真实请求。

负载均衡的策略有7种：

1）RoundRobinRule---轮询策略

2）RandomRule---随机策略，随机选择一个

3）RetryRule---重试策略，带有重试功能

4）WeightedResponseTimeRule---权重策略，根据权重来选择，如果一个服务的平均响应时间越短则权重越大，那么该服务实例被选中执行任务的概率也就越大

5）BestAvailableRule---最小连接数策略，忽略那些短路的服务器，并选择并发数较低的服务器

6）AvailabilityFilteringRule---可用敏感性策略，先过滤掉非健康的服务实例，然后再选择连接数较小的服务实例

7）ZoneAvoidanceRule---区域敏感策略，根据服务所在区域（zone）的性能和服务的可用性来选择服务实例，在没有区域的环境下，该策略和轮询策略类似

Ribbon 默认负载均衡策略是ZoneAvoidanceRule区域敏感策略，没有区域环境下与轮询策略类似。

### 4）、Ribbon是如何实现轮询的？如果让你自己实现轮询，如何实现？

### 5）、Feign和Ribbon的关系是什么？

[ribbon](https://so.csdn.net/so/search?q=ribbon&spm=1001.2101.3001.7020" \t "_blank)和feign都是客户端的提供负载均衡的工具，feign集成了ribbon。

     它们两个的区别是ribbon是通过[restTemplate](https://so.csdn.net/so/search?q=restTemplate&spm=1001.2101.3001.7020" \t "_blank)进行调用，feign是通过面向接口的调用

实现的区别：

1.启动类注解不同，Ribbon是@RibbonClient而feign的是@EnableFeignClients

2.服务指定的位置不同，Ribbon是在@RibbonClient注解上声明，Feign则是在定义抽象方法的接口中使用@FeignClient声明。

3.调用方式不同，Ribbon需要自己构建http请求，模拟http请求然后使用RestTemplate发送给其他服务，步骤相当繁琐。Feign只需要将调用的方法定义成抽象方法即可。

### 6）、你们项目中如何使用Feign的（Feign的最佳实践）

将Feign的Client抽取为独立模块，引入feign的starter依赖，并且把接口有关的POJO、默认的Feign配置都放到这个模块中，其余的消费者模块通过引入这个Feign的Client模块依赖，来调用其方法，实现远程调用，要注意其余的消费者模块的启动类@EnableFeignClients注解上声明要调用的feign客户端所在的包。使用HttpClient连接池代替默认的URLConnection：

① 引入feign-httpClient依赖

② 在微服务的application.xml配置文件开启httpClient功能，设置连接池参数。

### 7）、Feign远程调用时的日志级别有哪些？

通过在配置文件中feign.Logger.Level来修改日志级别，日志的级别分为四种：

NONE：不记录任何日志信息，这是默认值

BASIC：仅记录请求的方法，URL以及响应状态码和执行时间

HEADERS：在BASIC的基础上，额外记录了请求和响应的头信息

FULL：记录所有请求和响应的明细，包括头信息、请求体、元数据

### 8）、如何优化Feign的调用性能？

1.日志级别尽量用basic

2. 提高Feign的性能主要手段就是使用连接池代替默认的URLConnection，使用HttpClient或OKHttp代替URLConnection：

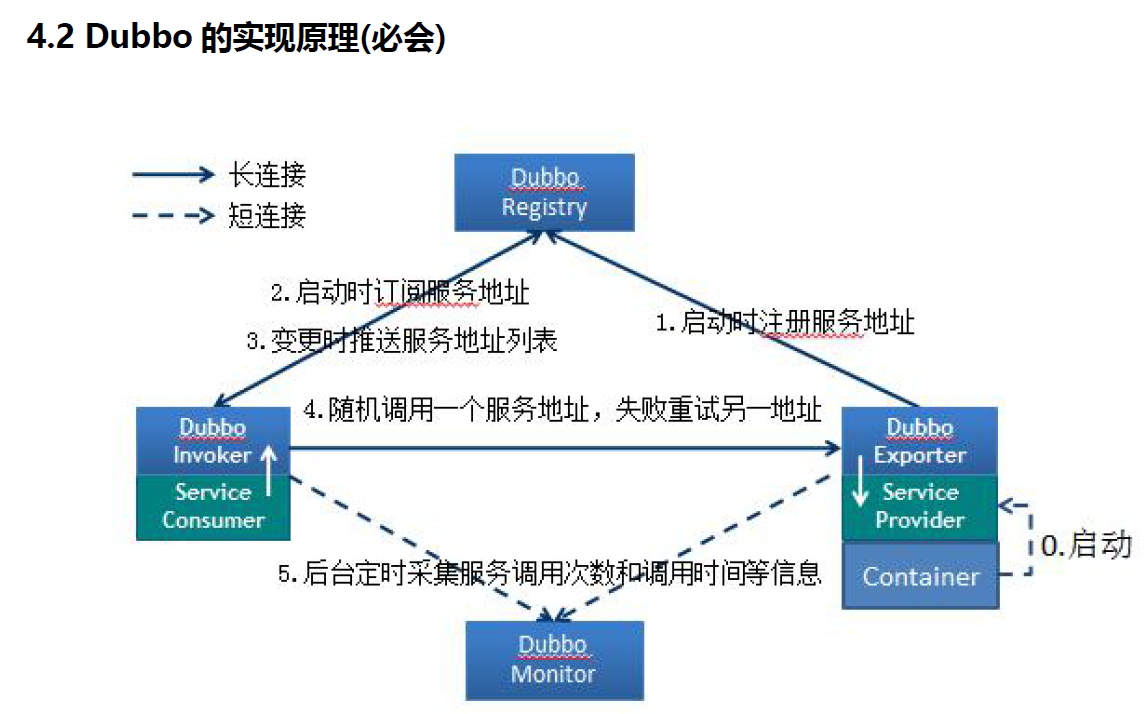
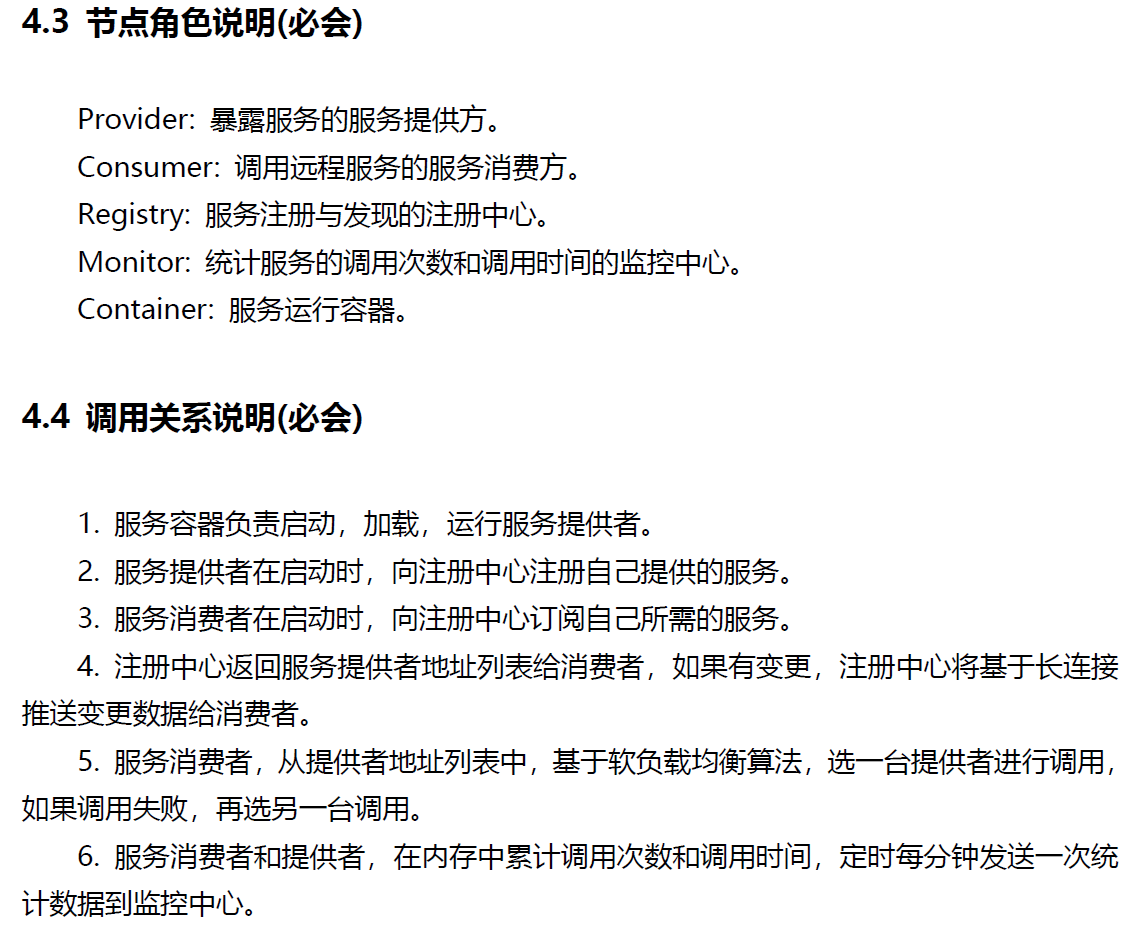
① 引入feign-httpClient依赖

② 在微服务的application.xml配置文件开启httpClient功能，设置连接池参数

### 9）、Feign的默认超时时间是多久？重试次数是几次？

Feign的重试机制在源码中默认关闭的，因为Ribbon的重试机制和Fiegn原来的重试机制冲突，所以在一般情况下，Feign的重试机制指的就是Ribbon的重试机制，默认超时时间为1s，重试次数是1次，加上原本的一次，总共是2次。

### 10）、Dubbo服务注册与发现的原理？（官方原理图）

### 11）、Dubbo负载均衡策略有哪些？默认是哪种？

Dubbo的负载均衡策略有4种，默认是random 按权重随机：

•Random ：按权重随机，默认值。按权重设置随机概率。

•RoundRobin：按权重轮询。

•LeastActive：最少活跃调用数，相同活跃数的随机。

•ConsistentHash：一致性Hash，相同参数的请求总是发到同一提供者。

### 12）、Dubbo支持哪些通信协议？默认是哪种？一般用哪种协议？有什么好处？

支持dubbo协议、rmi协议、webservice协议、http协议、hessian协议、thrift协议、memcache协议、redis协议。

默认是dubbo协议，单一TCP长连接和NIO异步通讯，

优点是：适合大并发小数据量的服务调用，以及服务消费者远大于提供者的情况，并可以实现Hessian二进制序列化；缺点是不适合传送大数据包的服务。

### 13）、注册中心挂了影响服务调用吗？为什么？

不影响。首先对于正在运行的 Consumer 调用 Provider 是不需要经过注册中心，所以不受影响。

​ 其次，Consumer 进程中，内存已经缓存了 Provider 列表。

​ 再其次 如果 Provider 下线，如果 Provider 是**正常关闭**，它会主动且直接对和其处于连接中的 Consumer 们，发送一条“我要关闭”了的消息，那么，Consumer 们就不会调用该 Provider ，而调用其它的 Provider 。

​ 另外，因为 Consumer 也会持久化 Provider 列表到本地文件。所以，此处如果 Consumer 重启，依然能够通过本地缓存的文件，获得到 Provider 列表。

​ 最后，一般情况下，注册中心是一个集群，如果一个节点挂了，Dubbo Consumer 和 Provider 将自动切换到集群的另外一个节点上。

### 14）、Dubbo启动检查如何设置？多版本支持如何设置？

Dubbo默认会在启动时检查依赖的服务是否可用，不可用时会抛出异常，阻止 Spring 初始化完成，以便上线时，能及早发现问题，默认 check=“true”。

可以通过 check=“false” 关闭检查 。

在 Dubbo 中为同一个服务配置多个版本，可以用版本号过渡，版本号不同的服务相互间不引用。可以按照以下的步骤进行版本迁移：

　　①在低压力时间段，先升级一半提供者为新版本

　　②再将所有消费者升级为新版本

　　③然后将剩下的一半提供者升级为新版本

### 15）、Dubbo的默认超时时间是多久？重试次数是几次？

Dubbo默认超时时间是1s，重试2次，加上第一次总共是3次。

### 16）、Dubbo进行服务注册和发现的核心注解是哪个？

@EnableDubbo, 该注解组合了注解@EnableDubboConfig和注解@DubboComponentScan。

### 17）、Dubbo服务如何进行监控和管理？

一旦使用 Dubbo 做了服务化后，必须要做服务治理，也就是说，要做服务的管理与监控。当然，还有服务的降级和限流。

Dubbo 管理平台 + 监控平台

①dubbo-monitor 作为监控平台，基于 Dubbo 的【monitor 监控层】，实现相应的监控数据的收集到监控平台。

②dubbo-admin 作为管理平台，基于注册中心，可以获取到服务相关的信息

## 2、注册中心

### 1）、Eureka

1、eureka是属于什么体系的技术（Spring Cloud）

Eureka是属于spring cloud体系的技术，作为微服务的服务注册发现中心。

2、eureka技术体系有哪些角色？（服务端用作注册中心，客户端用作微服务）

Eureka体系包含有eureka服务端作为服务注册发现中心，各个微服务作为其客户端，客户端又分为服务的提供者及服务的消费者。

3、eureka的自我保护机制是什么？什么时候开启？为什么开启？开启了会发生什么？

Eureka体系中服务的注册和发现都是可控制的，可以关闭也可以开启。默认都是开启

 客户端注册后需要向注册中心发送心跳，心跳周期默认 30 秒一次，超过 90 秒没发心跳认为宕机

 服务拉取默认 30 秒拉取一次

 Eureka 每隔 60 秒会剔除标记为宕机的服务

 Eureka 会有自我保护，当心跳失败比例超过阈值(默认 85%)，那么开启自我保护，不再剔除服务。

 Eureka 高可用就是多台 Eureka 互相注册在对方上

4、eureka作为注册中心的原理是什么？心跳检测某个服务是否健康的原理详细说下？

Eureka-Server：就是服务注册中心（可以是一个集群），对外暴露自己的地址。

 提供者：启动后向 Eureka 注册自己信息（地址，服务名称等），并且通过DiscoveryClient.java这个类发送定时任务，定期发送心跳进行服务续约，心跳周期默认 30 秒一次，超过 90 秒没发心跳认为宕机。

 消费者：服务调用方，会定期去 Eureka 拉取服务列表，默认 30 秒拉取一次，然后使用负载均衡算法选出一个服务进行调用。

 心跳(续约)：提供者定期通过 http 方式向 Eureka 刷新自己的状态

 客户端注册后需要向注册中心发送心跳，心跳周期默认 30 秒一次，超过 90 秒没发心跳认为宕机

 服务拉取默认 30 秒拉取一次

 Eureka 每隔 60 秒会剔除标记为宕机的服务

 Eureka 会有自我保护，当心跳失败比例超过阈值(默认 85%)，那么开启自我保护，不再剔除服务。

5、eureka集群是属于AP还是CP？(AP)

Eureka集群属于AP，可用、分区容错。

Consistency（一致性）：用户访问分布式系统中的任意节点，得到的数据必须一致

Availability（可用性）：用户访问集群中的任意健康节点，必须能得到响应，而不是超时或拒绝

Partition tolerance （分区容错性）：因为网络故障或其它原因导致分布式系统中的部分节点与其它节点失去连接，形成独立分区

CAP定理：在分布式系统中，系统间的网络不能100%保证健康，一定会有故障的时候，而服务又必须对外保证服务。因此Partition Tolerance不可避免，在P一定会出现的情况下，A和C之间只能实现一个

### 2）、Nacos

1、nacos是属于什么体系的技术（Spring Cloud Alibaba）

Nacos是属于Spring Cloud Alibaba体系的技术，作为微服务体系的服务注册中心及配置管理中心。

2、nacos作为注册中心的原理是什么？

nacos服务注册中心（可以是一个集群），对外暴露自己的地址，主体原理与eureka差不多，Nacos集群默认采用AP方式，当集群中存在非临时实例时，采用CP模式。

 提供者：对于临时的实例，启动后向 nacos注册自己信息（地址，服务名称等），并且发送定时任务定期发送心跳进行服务续约；对于非临时的实例，nacos会主动询问该服务的提供方是否健康。如果实例上次⼼跳时间⼤于15s就将其状态设置为不健康，如果超出30s，则直接将该实例删除，且临时实例心跳不正常会被剔除，非临时实例则不会被剔除，而是等待其恢复正常。

 消费者：服务调用方，会定期去nacos拉取服务列表，保存到本地缓存，然后使用负载均衡算法选出一个服务进行调用，如果服务的提供方有变更，nacos每隔10s定时推送数据到服务的消费者。

3、nacos如何确定唯一的一个服务?（通过namespace、group、service、集群唯一确定一个服务）

通过namespace（命名空间）、group（分组）、service（服务名）、集群唯一确定一个服务。

4、nacos中namespace、group分别的作用是什么？

Namespace：命名空间，

用于进行配置隔离。不同的命名空间下，可以存在相同的 Group 或 Data ID 的配置。Namespace 的常用场景之一是不同环境的配置的区分隔离，例如开发测试环境和生产环境的资源（如配置、服务）隔离等。

Group：

Nacos 中的一组配置集，是组织配置的维度之一。通过一个有意义的字符串（如 Buy 或 Trade ）对配置集进行分组，从而区分 Data ID 相同的配置集。当在 Nacos 上创建一个配置时，如果未填写配置分组的名称，则配置分组的名称默认采用 DEFAULT\_GROUP 。配置分组的常见场景：不同的应用或组件使用了相同的配置类型，如 database\_url 配置和 MQ\_topic 配置。

DataId：

Nacos 中的某个配置文件的 ID。Data ID是划分配置的维度之一。一个系统或者应用可以包含多个配置集，每个配置集都可以被一个有意义的名称标识。Data ID 通常命名为[服务名称]-[profile].[后缀名]

5、nacos和eureka的异同有哪些？

Nacos可以作为注册中心，也可以作为配置中心，eureka只能作为注册中心

当nacos作为注册中心时：

Nacos与eureka的共同点

①都支持服务注册和服务拉取

②都支持服务提供者心跳方式做健康检测

Nacos与Eureka的区别

①Nacos支持服务端主动检测提供者状态：临时实例采用心跳模式，非临时实例采用主动检测模式。

②临时实例心跳不正常会被剔除，非临时实例则不会被剔除

③Nacos支持服务列表变更的消息推送模式，服务列表更新更及时

④Nacos集群默认采用AP方式，当集群中存在非临时实例时，采用CP模式；Eureka采用AP方式。

6、nacos的临时节点和非临时节点有什么区别？

可以通过配置文件设置ephemeral属性，为true则为临时实例（默认值）；

为false则为非临时实例，①Nacos检测提供者状态：临时实例采用心跳模式，非临时实例采用主动检测模式。

②临时实例心跳不正常会被剔除，非临时实例则不会被剔除，而是等待其重连。

③ Nacos集群默认采用AP方式，当集群中存在非临时实例时，采用CP模式。

7、nacos集群是属于AP还是CP？(AP或CP)

Nacos集群默认采用AP方式，当集群中存在非临时实例时，采用CP模式；Eureka采用AP方式。

### 3）、Zookeeper

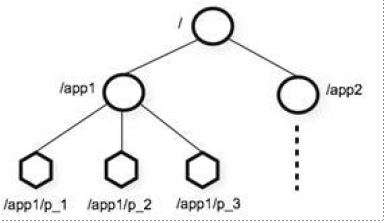
1、Zookeeper的内部结构是什么？

 ZK 本质上是一个分布式的小文件存储系统.

 ZK 表现为一个分层的文件系统目录树结构，既能存储数据, 而且还能像目录一样有子节点. 每个节点可以存最多 1M 左右的数据.

 每个节点称做一个 Znode, 每个 Znode 都可以通过其路径唯一标识.

 而且客户端还能给节点添加 watch, 也就是监听器, 可以监听节点的变化, 这个功能常在实际开发中作为监听服务器集群机器上下线操作。



图中的每个节点称为一个 Znode。 每个 Znode 由 3 部分组成:

① state：此为状态信息, 描述该 Znode 的版本, 权限等信息

② data：与该 Znode 关联的数据

③ children：该 Znode 下的子节点

Znode 有 2 大类 4 小类, 两大类分别为永久节点和临时节点.

 永久节点(Persistent): 客户端和服务器端断开连接后，创建的节点不会消失, 只有在客户端执行删除操作的时候, 他们才能被删除.

 临时节点(Ephemeral): 客户端和服务器端断开连接后，创建的节点会被删除. Znode 还有一个序列化的特性, 这个序列号对于此节点的父节点来说是唯一的, 这样便会记录每个子节点创建的先后顺序. 它的格式为“%10d”(10 位数字, 没有数值的数位用 0 补充, 例如“0000000001”)，因此节点可以分为 4 小类:

 永久节点(Persistent)

 永久\_序列化节点(Persistent\_Sequential)

 临时节点(Ephemeral)

 临时\_序列化节点(Ephemeral\_Sequential)

2、使用Zookeeper作为分布式事务锁的原理是什么？

分布式锁使用的是 Zookeeper 的临时有序节点，

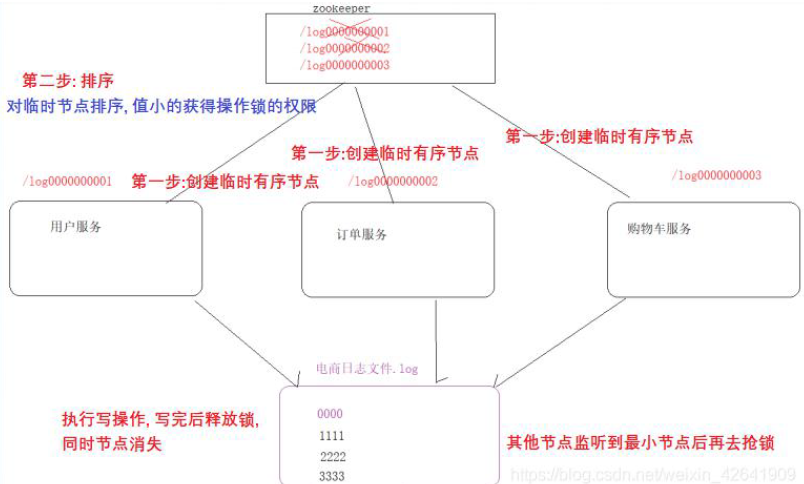
1. 所有需要执行操作的主机都去 Zookeeper 上创建一个临时有序节点.

2. 然后获取到 Zookeeper 上创建出来的这些节点进行一个从小到大的排序.

3. 判断自己创建的节点是不是最小的, 如果是, 自己就获取到了锁; 如果不是, 则对最小的节点注册一个监听.

4. 如果自己获取到了锁, 就去执行相应的操作. 当执行完毕之后, 连接断开, 节点消失, 锁就被释放了.

5. 如果自己没有获取到锁, 就等待, 一直监听节点是否消失，锁被释放后, 再重新执行抢夺锁的操作.



3、Zookeeper集群属于AP还是CP？(CP)

Zookeeper 在一个系统中一般会充当一个很重要的角色, 所以一定要保证它的高可用, 这就需要部署 Zookeeper 的集群. Zookeeper 有三种运行模式: 单机模式, 集群模式和伪集群模式.

 单机模式: 使用一台主机部署一个 Zookeeper 来对外提供服务, 有单点故障问题,仅适合于开发、测试环境.

 集群模式: 使用多台服务器, 每台上部署一个 Zookeeper 一起对外提供服务, 适合于生产环境.

 伪集群模式: 在服务器不够多的情况下, 也可以考虑在一台服务器上部署多个Zookeeper 来对外提供服务.

Zookeeper集群属于CP（一致性、分区容错性）

## 3、配置中心

### 1）、实现配置中心都可以使用哪些技术？

可以使用SpingCloudConfig、Nacos作为配置中心

### 2）、使用nacos作为配置中心，如何实现热更新？

Nacos作为配置中心，可以有两种方式实现热更新：

（1）在@Value注入的变量所在类上添加注解@RefreshScope

（2）在类或方法上使用@ConfigurationProperties注解代替@Value注解。

例如在user-service服务中，添加一个类，类上打注解@ConfigurationProperties(prefix = "pattern")就可以通过类的成员变量：private String dateformat; 来读取patterrn.dateformat属性

在UserController中使用这个类代替@Value。（@Autowired注入这个类，就可以获取类的dateformat成员变量了）

### 3）、nacos作为配置中心，为什么需要用到bootstrap文件？

微服务要拉取nacos中管理的配置，并且与本地的application.yml配置合并，才能完成项目启动。但如果尚未读取application.yml，又如何得知nacos地址呢？因此spring引入了一种新的配置文件：bootstrap.yaml文件，会在application.yml之前被读取，项目的核心配置，需要热更新的配置才有放到nacos管理的必要。基本不会变更的一些配置还是保存在微服务本地比较好。

### 4）、远程配置文件和本地配置文件属性加载优先级是什么样的？

远程的服务名-profile.yaml >远程的服务名称.yaml > 本地配置

### 4）、远程配置文件和本地配置文件属性加载优先级是什么样的？

远程的服务名-profile.yaml >远程的服务名称.yaml > 本地配置

### 5）、使用配置中心的好处是什么？能解决什么问题？

微服务意味着要将单体应用中的业务拆分成一个个子服务，每个服务的粒度相对较小，因此系统中会出现大量的服务，由于每个服务都需要必要的配置信息才能运行，所以一套集中式的，动态的配置管理设施是必不可少的。我们每一个微服务自己带着一个application.yml，那大量的配置文件修改起来，令人头疼！SpringCloudConfig与Nacos作为配置中心可以解决这个问题，

集中式管理配置文件

（1）不同环境，不同配置，动态化的配置更新，分环境部署，比如 /dev /test /prod /beta /release

（2）运行期间动态调整配置，不再需要在每个服务部署的机器上编写配置文件，（3）服务会向配置中心统一拉取自己配置的信息

（4）通过热更新配置，当配置发生变动时，服务不需要重启，即可感知到配置的变化，并应用新的配置

## 4、服务保护

### 1）、Hystrix

1、hystrix是属于什么体系的技术？

Hystrix是SpringCloud体系的一种服务熔断降级保护技术

2、hystrix可以用来干嘛？

Hystrix可以用于服务熔断降级：

 有时候可能是网络问题, 一些其它问题, 导致代码无法正常运行, 这时服务就挂了, 崩溃了. 熔断器就是为了解决无法正常访问服务时, 提供的一种解决方案.

 解决因为一个服务崩溃而引起的一系列问题, 使问题只局限于这个服务中,不会影响其他服务.

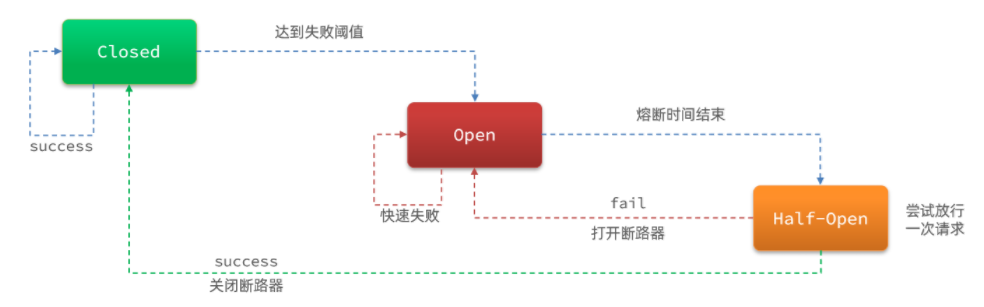
 Hystrix 提供了两种功能, 一种是服务降级, 一种是服务熔断.

（1）服务降级原理

 Hystrix 为每个服务分配了小的线程池, 当用户发请求过来, 会通过线程池创建线程来执行任务, 当创建的线程池已满或者请求超时(这里和多线程线程池不一样,不存在任务队列), 则启动服务降级功能.

 降级指的是请求故障时, 不会阻塞, 会返回一个友好提示(可以自定义, 例如网站维护中请稍后重试), 也就是说不会影响其他服务的运行.

（2）服务熔断原理



断路器的状态机有 3 个状态：

 Closed：关闭状态（断路器关闭），所有请求都正常访问。

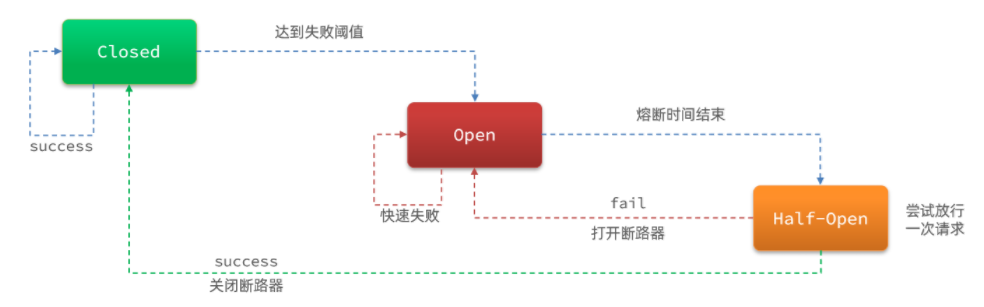
 Open：打开状态（断路器打开），所有请求都会被降级。Hystix 会对请求情况计数，当5秒之内服务调用异常比例达到一半或者失败次数超过20次，则触发熔断，断路器会完全打开。

 Half Open：半开状态，open 状态不是永久的，打开后会进入休眠时间（默认是 5S）。随后断路器会自动进入半开状态。此时会释放 1 次请求，若这个请求是健康的，则会关闭断路器，否则继续保持打开，再次进行 5 秒休眠计时。

3、hystrix默认的触发熔断策略是什么？

（5秒之内服务调用异常比例达到一半或者失败次数超过20次）

服务熔断原理



状态机有 3 个状态：

 Closed：关闭状态（断路器关闭），所有请求都正常访问。

 Open：打开状态（断路器打开），所有请求都会被降级。Hystix 会对请求情况计数，当5s之内服务调用异常比例达到一半或者失败次数超过20次，则触发熔断，断路器会完全关闭。

 Half Open：半开状态，open 状态不是永久的，打开后会进入休眠时间（默认是 5S）。随后断路器会自动进入半开状态。此时会释放 1 次请求通过，若这个请求是健康的，则会关闭断路器，否则继续保持打开，再次进行 5 秒休眠计时。

4、hystrix的隔离是基于什么？ （线程池隔离，低扇出）

**线程池隔离（**hystrix采用**）**：给每个服务调用业务分配一个线程池，利用线程池本身实现隔离效果，优点是支持主动超时，支持异步调用，缺点是线程池的额外开销大，适用场景是低扇出。

**信号量隔离**（Sentinel默认采用）：不创建线程池，而是计数器模式，记录业务使用的线程数量，达到信号量上限时，禁止新的请求。优点是轻量级，无额外开销，缺点是不支持主动超时，不支持异步调用，适用场景是高频调用，高扇出。

### 2）、Sentinel

1、sentinel可以用来干嘛？

Sentinel是SpringCloudAlibaba的一款服务降级保护的组件，可以进行服务的限流、隔离、熔断、降级

2、什么是微服务雪崩现象？如何解决微服务雪崩问题？

微服务中，服务间调用关系错综复杂，一个微服务往往依赖于多个其它微服务，服务器支持的线程和并发数有限，请求一直阻塞，会导致服务器资源耗尽，从而导致所有其它服务都不可用，当前服务也就不可用了。那么，依赖于当前服务的其它服务随着时间的推移，最终也都会变的不可用，形成级联失败，雪崩就发生了

解决方式一般有四种：

（1）超时处理：设定超时时间，请求超过一定时间没有响应就返回错误信息，不会无休止等待；

（2）仓壁模式：我们可以限定每个业务能使用的线程数，避免耗尽整个tomcat的资源，因此也叫线程隔离；

（3）断路器模式：由断路器统计业务执行的异常比例，如果超出阈值则会熔断该业务，拦截访问该业务的一切请求。

（4）流量控制：限制业务访问的QPS，避免服务因流量的突增而故障

3、sentinel的限流模式有哪些？分别的运用场景是什么？

三种流控模式：直接、关联、链路

**直接：**统计当前资源的请求，触发阈值时对当前资源直接限流，也是默认的模式。

**使用场景**：比如用户支付时需要修改订单状态，同时用户要查询订单。查询和修改操作会争抢数据库锁，产生竞争。业务需求是优先支付和更新订单的业务，因此当修改订单业务触发阈值时，需要对查询订单业务限流

**关联：**统计与当前资源相关的另一个资源，触发阈值时，对当前资源限流。

**使用场景：**两个有竞争关系的资源，一个优先级比较高，一个优先级比较低。

**链路：**统计从指定链路访问到本资源的请求，触发阈值时，对指定链路限流。

**使用场景：**api级别的针对来源

4、sentinel的限流效果有哪些？分别的运用场景是什么？

流控效果是指请求达到流控阈值时应该采取的措施，包括三种：

快速失败：达到阈值后，新的请求会被立即拒绝并抛出FlowException异常。是默认的处理方式。

warm up：预热模式，对超出阈值的请求同样是拒绝并抛出异常。但这种模式阈值会动态变化，从一个较小值逐渐增加到最大阈值。

排队等待：让所有的请求按照先后次序排队执行，两个请求的间隔不能小于指定时长

5、sentinel支持对热点参数进行限流吗？

sentinel支持对热点参数进行限流

在实际开发中，可能部分商品是热点商品，例如秒杀商品，我们希望这部分商品的QPS限制与其它商品不一样，高一些。那就需要配置热点参数限流的高级选项了，有两个例外：

•如果参数值是100，则每1秒允许的QPS为10

•如果参数值是101，则每1秒允许的QPS为15

6、实现微服务调用隔离有两种方式（信号量隔离和线程池隔离），区别是什么？sentinel是使用的哪种？

**线程池隔离（**hystrix采用**）**：给每个服务调用业务分配一个线程池，利用线程池本身实现隔离效果，优点是支持主动超时，支持异步调用，缺点是线程池的额外开销大，适用场景是低扇出。

**信号量隔离**（Sentinel默认采用）：不创建线程池，而是计数器模式，记录业务使用的线程数量，达到信号量上限时，禁止新的请求。优点是轻量级，无额外开销，缺点是不支持主动超时，不支持异步调用，适用场景是高频调用，高扇出。

7、什么是熔断？熔断的原理是什么？什么时候会触发sentinel的熔断？断路器的三种状态是哪些？是怎样进行切换的？

服务熔断就是相当于我们电闸的保险丝,一旦发生服务雪崩的,就会熔断整个服务,通过维护一个自己的线程池,当线程达到阈值的时候就启动服务降级,如果其他请求继续访问就直接返回fallback的默认值。

Sentinel熔断达到触发条件: 当每秒接口数超过20%的处理产生异常,便会开启熔断，

断路器状态机包括三个状态：

closed：关闭状态，断路器放行所有请求，并开始统计异常比例、慢请求比例。超过阈值则切换到open状态

open：打开状态，服务调用被**熔断**，访问被熔断服务的请求会被拒绝，快速失败，直接走降级逻辑。Open状态5秒后会进入half-open状态

half-open：半开状态，放行一次请求，根据执行结果来判断接下来的操作。

请求成功：则切换到closed状态

请求失败：则切换到open状态

断路器熔断策略有三种：慢调用、异常比例、异常数

8、什么是降级？如何实现降级？

降级指的是请求故障时, 不会阻塞, 会返回一个友好提示(可以自定义, 例如网站维护中请稍后重试), 也就是说不会影响其他服务的运行。

利用feign远程调用，在配置文件中开启feign的Sentinel功能。

给FeignClient编写失败后的降级逻辑

①方式一：FallbackClass，无法对远程调用的异常做处理

②方式二：FallbackFactory，可以对远程调用的异常做处理

9、sentinel授权规则是用来干什么的？

授权规则可以对调用方的来源做控制，有白名单和黑名单两种方式。

白名单：来源（origin）在白名单内的调用者允许访问

黑名单：来源（origin）在黑名单内的调用者不允许访问

10、sentinel的规则持久化方式有哪些？一般使用哪种？

规则是否能持久化，取决于规则管理模式，sentinel支持三种规则管理模式：

原始模式：Sentinel的默认模式，将规则保存在内存，重启服务会丢失。

pull模式：控制台将配置的规则推送到Sentinel客户端，而客户端会将配置规则保存在本地文件或数据库中。以后会定时去本地文件或数据库中查询，更新本地规则

push模式：控制台将配置规则推送到远程配置中心，例如Nacos。Sentinel客户端监听Nacos，获取配置变更的推送消息，完成本地配置更新

## 5、网关

### 1）、网关有什么作用？在你们项目中用网关来干嘛了？

Gateway网关是我们服务的守门神，所有微服务的统一入口。

网关的**核心功能特性**：请求路由、过滤、权限控制、负载平衡、限流

我们项目用的是SpringCloudGateway网关用于接收从Nginx反向代理过来的请求，经过全局过滤后负载平衡地路由到各个微服务，同时在网关中对用户登录生成的JWT的TOKEN进行校验（用户登录校验）、权限校验。

### 2）、网关的核心技术点有哪些？

请求路由、过滤、权限控制、负载平衡、限流

### 3）、网关的路由是用来干嘛的？分为哪几种？

一切请求都必须先经过gateway，但网关不处理业务，而是根据某种规则，把请求转发到某个微服务，这个过程叫做路由。当然路由的目标服务有多个时，还需要做负载均衡。

### 4）、网关的过滤器是用来干嘛的？分为哪几种？

GatewayFilter是网关中提供的一种过滤器，对路由的请求或响应做加工处理，比如添加请求头。

分为指定微服务生效的过滤器（filters）、默认过滤器（default-filters对所有的路由都生效）、全局过滤器（实现GlobalFilter接口）

### 5）、网关局部过滤器和全局过滤器的区别有哪些？

局部过滤器的作用域是指定的微服务，默认过滤器是所有指定的路由都生效，提供的过滤器作用都是固定的，全局过滤器的作用也是处理一切进入网关的请求和微服务响应，与GatewayFilter的作用一样。

区别在于GatewayFilter通过配置定义，处理逻辑是固定的；而GlobalFilter的逻辑需要自己写代码实现。

定义方式是实现GlobalFilter接口。

### 6）、网关中局部过滤器、默认过滤器、全局过滤器的执行顺序是什么？

请求路由后，会将局部过滤器、默认过滤器、全局过滤器，合并到一个过滤器链（集合）中，排序后依次执行每个过滤器，排序的规则是什么呢？

每一个过滤器都必须指定一个int类型的order值，**order值越小，优先级越高，执行顺序越靠前**。

GlobalFilter全局过滤器通过实现Ordered接口，或者添加@Order注解来指定order值，由我们自己指定

路由过滤器和defaultFilter的order由Spring指定，默认是按照声明顺序从1递增。

当过滤器的order值一样时，会按照 defaultFilter > 路由过滤器 > GlobalFilter的顺序执行

### 7）、加入网关后，访问一个链接，你们项目的执行流程是什么？

1、我们的前端部署在Nginx中的，前端发起请求，请求到达Nginx

2、通过Nginx的反向代理，将请求代理至网关

3、请求到网关后会经过全局过滤器，全局过滤器判断当前请求URL，如果是获取验证码或登录URL，直接放行

4、全局过滤器放行后，请求经过网关的动态路由，会将请求路由至对应的用户微服务

### 8）、定义全局过滤器需要实现哪几个接口？

实现GlobalFilter接口即可，重写其中的filter方法

## 6、分布式事务

### 1）、什么是本地事务？什么是分布式事务？

本地事务：无需跨越多个服务或者数据源的单体事务，一般由spring控制即可（声明式事务处理：AOP---基于代理，事务四大特性A原子性，C一致性，I隔离性，D持久性）

分布式事务：指一个业务跨越多个服务或者数据源，每个事务叫做分支事务，要保证所有分支事务要么同成功，要么同失败。

### 2）、什么是CAP定理？为什么必须保证P？为什么在保证P的前提下只能保证C或者A其中一个？

1）、C：数据一致性：在任何时候，访问任何节点返回的数据都是一致的

2）、A：服务可用性：在任何时候，访问任何节点都要能够得到响应

3）、P：分区容错性：在网络波动出现分区时要保证服务依然可用（必须保证的）

为什么必须保证P？

因为在分布式系统中，系统间的网络不能100%保证健康，一定会有故障的时候，而服务又必须对外保证服务。因此Partition Tolerance不可避免，一定要保证P；

当保证P分区容错性时，

如果此时要保证**一致性**，就必须等待网络恢复，完成数据同步后，整个集群才对外提供服务，服务处于阻塞状态，不可用。

如果此时要保证**可用性**，就不能等待网络恢复，那节点之间就会出现数据不一致。

也就是说，在P一定会出现的情况下，A和C之间只能实现一个

### 3）、什么是BASE理论？---AP模式

（Basically Available）基本可用：分布式系统在出现故障时，允许损失部分可用性，即保证核心可用。

（Soft State）软状态：在一定时间内，允许出现中间状态，比如临时的不一致状态。

（Eventually Consistent）最终一致性：虽然无法保证强一致性，但是在软状态结束后，最终达到数据一致。

### 4）、seata解决分布式事务的三个角色以及分别的作用是什么？

1）、TC：全局事务协调者

用于记录全局事务和分支事务的状态，协助全局事务管理者进行全局事务的提交或回滚

2）、TM：全局事务管理者

用于定义全局事务的范围，用于进行全局事务的提交或回滚

3）、RM：资源管理器（分支事务的管理器）

用于与TC进行交流，向TC注册、报告自己的事务信息，对本地分支事务进行具体的提交、回滚

### 5）、seata解决分布式事务的四种模式

1.XA模式：强一致性分阶段事务模式，牺牲了一定的可用性，无业务侵入；

2.AT模式：最终一致的分阶段事务模式，无业务侵入，也是Seata的默认模式；

3.TCC模式：最终一致的分阶段事务模式，有业务代码侵入；

4.SAGA模式：长事务模式，有业务侵入。



### 6）、XA模式特点、原理以及应用场景？优缺点有哪些？

XA：强一致性分阶段事务模式，牺牲了一定的可用性，无业务代码侵入（CP），适用于对数据的安全性（一致性、隔离性）要求很高，对性能要求不是很高的情况下使用。

1）、第一阶段：

1、TM：

1、开启全局事务

2、调用分支事务

2、TC:

接收开启全局事务的请求

3、RM:

1、将分支事务注册到TC服务

2、执行本地sql，但是，重点：不提交事务！！！！

3、将本地事务状态报告至TC服务

2）、第二阶段：

1、TM：

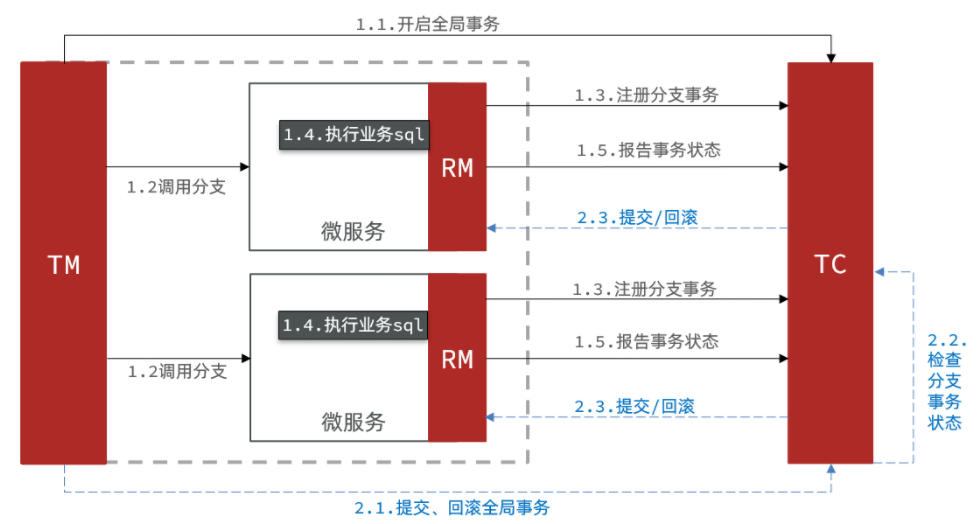
等第一阶段所有分支事务执行完，发起提交/回滚全局事务的命令

2、TC:

接收全局事务提交/回滚请求，核查所有分支事务的状态，对RM发起提交/回滚的命令

3、RM:

提交或者回滚当前的分支事务(依赖于数据库)



3）、优点：

强一致性、无代码侵入、实现简单

4）、缺点：

强依赖于关系型数据库实现回滚、性能比较差。

### 7）、AT模式特点、原理以及应用场景？优缺点有哪些？有可能会出现什么问题？如何解决？

AT：同样是分阶段提交的事务模型，不过缺弥补了XA模型中资源锁定周期过长的缺陷，属于最终一致（AP），基于关系型数据库的大多数分布式事务场景都可以使用。

1）、第一阶段：

1、TM:

1、开启全局事务

2、调用分支事务

2、TC:

接收开启全局事务的请求

3、RM：

1、将分支事务注册到TC服务

2、执行本地sql，重点：提交事务！！！！

3、执行sql前后，生成快照:undo\_log

4、将本地事务状态报告至TC服务

2）、第二阶段：

1、TM：

等第一阶段所有分支事务执行完，发起提交/回滚全局事务的命令

2、TC:

接收全局事务提交/回滚请求，核查所有分支事务的状态，对RM发起提交/回滚的命令

3、RM:

提交或者回滚当前的分支事务（依赖于undo\_log快照数据）

提交：删除快照数据

回滚：根据快照进行数据恢复后删除快照数据

3）、有可能出现的问题

脏写：当全局事务1提交修改的数据后，此时全局事务2又过来修改了这条数据，后续阶段二全局事务1需要利用快照进行回滚，将全局事务2的所有修改进行了覆盖

如何解决？seata内部提供了全局锁的概念（需要在seata server新增一张全局锁的表）

但是全局锁有可能导致死锁（内部通过限制获取全局锁的次数来解决：重试30次，每次10ms）

DB锁和Seata全局锁的区别？

Seata全局锁是一种轻量级的锁，锁定粒度更小，锁定的是行内的操作的字段， DB锁是行锁。

当非seata事务参与修改（脏写），怎么办？通过修改后的快照进行判断，如果与当前的数据一致则没问题，不一致说明有别的事务改动了 需要人工介入。

4）、优点：

1、一阶段完成直接提交事务，释放数据库资源，性能比较好

2、利用全局锁实现读写隔离

3、没有代码侵入，框架自动完成回滚和提交

5）、缺点：

1、两阶段之间属于软状态，属于最终一致

2、框架的快照功能会影响性能，但比XA模式要好很多

### 8）、重点：TCC模式特点、原理以及应用场景？优缺点有哪些？有可能会出现什么问题？什么是空回滚和业务悬挂，如何解决？

TCC模式与AT模式非常相似，每阶段都是独立事务，不同的是TCC通过人工编码来实现数据恢复。属于最终一致（AP），适合对性能有较高要求的，对业务有特殊需求，有非关系型数据库参与的事务情况下使用。

1、TCC过程

1）、T: Try，进行资源的检测和预留

2）、C：Confirm，对资源进行确认操作（业务执行和提交）

3）、C：Cancle，对资源进行回滚操作（预留资源的释放）

2、有可能出现的问题

1）、空回滚

问题描述：当某分支事务的try阶段阻塞时，可能导致全局事务超时而触发二阶段的cancel操作。在未执行try操作时先执行了cancel操作，这时cancel不能做回滚，这就是空回滚。

解决办法：执行cancel操作前，应当判断try是否已经执行，如果尚未执行，就应该空回滚，并在数据库表中添加一条状态为cancel的事务数据。

2）、业务悬挂

问题描述：对于已经空回滚的业务，之前被阻塞的try操作恢复，继续执行try，就永远不可能confirm或cancel ，事务一直处于中间状态，这就是业务悬挂。

解决办法：执行try操作前，应当判断cancel是否已经执行过了，如果已经执行，应当阻止空回滚后的try操作，避免悬挂

3、优点

1）、一阶段完成直接提交事务，释放数据库资源，性能好

2）、相比AT模型，无需生成快照，无需使用全局锁，性能最强

3）、不依赖数据库事务，而是依赖补偿操作，可以用于非事务型数据库

4、缺点

1）、有代码侵入，需要人为编写try、Confirm和Cancel接口，太麻烦

2）、软状态，事务是最终一致

3）、需要考虑Confirm和Cancel的失败情况，做好幂等处理

### 9）、SAGA模式特点、原理以及应用场景？优缺点有哪些？

在 Saga 模式下，分布式事务内有多个参与者，每一个参与者都是一个冲正补偿服务，需要用户根据业务场景实现其正向操作和逆向回滚操作。

分布式事务执行过程中，依次执行各参与者的正向操作，如果所有正向操作均执行成功，那么分布式事务提交。如果任何一个正向操作执行失败，那么分布式事务会去退回去执行前面各参与者的逆向回滚操作，回滚已提交的参与者，使分布式事务回到初始状态

Saga也分为两个阶段：

一阶段：直接提交本地事务

二阶段：成功则什么都不做；失败则通过编写补偿业务来回滚

优点：

事务参与者可以基于事件驱动实现异步调用，吞吐高

一阶段直接提交事务，无锁，性能好

不用编写TCC中的三个阶段，实现简单

缺点：

软状态持续时间不确定，时效性差

没有锁，没有事务隔离，会有脏写

# 九、Docker

## 1、什么是Docker？优点是什么？有哪些核心概念？

Docker是一个快速交付应用、运行应用的技术

优势:

- 可以将程序及其依赖、运行环境一起打包为一个镜像，可以迁移到任意Linux操作系统

- 运行时利用沙箱机制形成隔离容器，各个应用互不干扰

- 启动、移除都可以通过一行命令完成，方便快捷

核心概念:

镜像:对容器运行所需的本地函数库、依赖、环境、配置进行打包形成的【静态数据】，只读，不可修改

容器:基于镜像进行创建而来、是一个位于操作系统上独立运行的进程，容器间相互隔离，对外不可见

## 2、镜像操作命令有哪些？

拉取、推送、查看、查看所有、删除、删除所有、制作镜像、导出镜像、加载镜像

拉取、推送、查看、查看所有、删除、删除所有、制作镜像、导出镜像、加载镜像

1）、拉取：docker pull

2）、推送：docker push

3）、查看：docker images

4）、查看所有镜像ID：docker images -q

5）、删除：docker rmi 镜像名称

6）、删除所有：docker rmi `docker images -q`

7）、制作镜像：docker build .

8）、导出镜像：docker save -o 镜像名称.tar 镜像名称

9）、加载镜像：docker load -i 镜像名称.tar

## 3、容器操作命令有哪些？

查看所有、查看正在运行的容器、删除、强制删除、创建容器、创建并运行、

启动容器、停止容器、重启容器、暂停容器、恢复容器、进入容器

1）、查看所有：docker ps -a

2）、查看正在运行：docker ps

3）、删除：docker rm 容器名称

4）、强制删除：docker rm -f 容器名称

5）、创建容器：docker create -d --name=容器名称 -p 宿主机端口：容器内端口 镜像名称

6）、创建并运行容器：docker run -d --name=容器名称 -p 宿主机端口：容器内端口 镜像名称

7）、启动容器：docker start 容器名称

8）、停止容器：docker stop 容器名称

9）、重启容器：docker restart 容器名称

10）、暂停容器：docker pause 容器名称

12）、恢复容器：docker unpause 容器名称

13）、进入容器：docker exec -it 容器名称 /bin/bash

## 4、数据卷操作命令有哪些？

创建数据卷、查看单个数据卷详情、查看数据卷列表、删除数据卷、删除未使用的数据卷、创建容器时挂载数据卷

1）、创建数据卷：docker volume create 数据卷名称 （位于/var/lib/docker/volume目录）

2）、查看单个数据卷详情：docker volume inspect 数据卷名称

3）、查看数据卷列表：docker volume ls

4）、删除数据卷：docker volume rm 数据卷名称

5）、删除未使用的数据卷：docker volume prune

6）、创建容器时挂载数据卷

①挂载数据卷（会自动创建目录）：docker run -v 数据卷名称：容器内目录路径

②挂载指定目录（要自己创建）：docker run -v 目录绝对路径：容器内目录路径

## 5、docker如何自定义镜像？docker file的语法是什么样的？

运行dockerfile文件

语法:

FROM java:8-alpine

COPY ./app.jar /tmp/app.jar

EXPOSE 8090

ENTRYPOINT java -jar /tmp/app.jar

## 6、docker compose是干嘛的?语法是什么样的？

Docker Compose可以基于Compose文件帮我们快速的部署分布式应用

# 十、ElasticSearch

## 1、什么是ES？由什么语言编写？和Lunce的关系？什么是ELK？

1）、ES是一款分布式、高性能、高扩展，支持海量数据分析、搜索、计算的搜索引擎

2）、基于java语言编写，发起的请求是基于json风格的符合RestFull风格的DSL语句

3）、前身-Lucene诞生于1999年，2004年变为了compass，2010年重构成了现在的ES

4）、ELK：是一个围绕ElasticSearch的技术栈，包含：ElasticSearch、Logstatsh、Kibana，最新版本7.16.3

5）、Solr: 是ES的一款竞品，2016年被ES吊打，主流成为ES

## 2、ES的核心概念有哪些？什么是索引？什么是文档？文档格式是什么？什么是映射？什么是DSL？

1）、倒排索引

1、首先在文档【增删改】的时候对文档进行【合理化的分词】，形成一个不重复的词条列表，其中每一个词条对应一个文档id集合

2、形成一颗根据词条查询id的B+tree

3、形成一颗根据id查询文档的B+tree

将来搜索时先根据用户输入的条件进行【合理化的分词】，再根据词条找id，最后根据id找到相应的文档（涉及到两次Btree查询）

2）、索引-index：同一类型文档的集合，相当于mysql的表

3）、映射-mapping：对索引结构的约束，相当于mysql的schema

4）、文档-document：json格式的数据，相当于mysql的row

5）、字段-field：一个个的字段，相当于mysql的列

6）、DSL语句：json风格的符合restful风格的请求语句，相当于mysql的sql语句

## 3、什么是倒排索引？倒排索引建立过程？

mysql正向索引是通过记录查找值,而倒排索引是通过值查找记录

1、首先在文档【增删改】的时候对文档进行【合理化的分词】，形成一个不重复的词条列表，其中每一个词条对应一个文档id集合

2、形成一颗根据词条查询id的B+tree

3、形成一颗根据id查询文档的B+tree

搜索时先根据用户输入的条件进行【合理化的分词】，再根据词条找文档id，最后根据文档id找到相应的文档（涉及到两次Btree查询）

## 4、ES有哪些数据类型？keyword和text有什么区别？

type:

字符串:text(可分词的文本,需要加分词规则: "analyzer": "ik\_max\_word"或者"analyzer": "ik\_smart",), keyword(精确值,不参与分词)

数值: byte，short，integer(注意这里的首字母i必须小写), long, float，double

布尔:boolean

日期:date

对象:object

## 5、重要：说说用户输入框输入查询条件 进行ES搜索的底层原理过程

1、首先在文档【增删改】的时候对文档进行【合理化的分词】，形成一个不重复的词条列表，其中每一个词条对应一个文档id集合

2、形成一颗根据词条查询id的B+tree

3、形成一颗根据id查询文档的B+tree

将来搜索时先根据用户输入的条件进行【合理化的分词】，再根据词条找id，最后根据id找到相应的文档（涉及到两次Btree查询）

## 6、ES分词器适合在什么字段上使用？分词器在ES中的使用场景有哪些？（建立倒排索引时对文档分词和用户搜索时对搜索条件分词）

ES分词器适合在类型为text的字段上使用，使用场景有建立倒排索引时对文档分词和用户搜索时对搜索条件分词

## 7、你们分词器用的哪种？为什么要自定义拼音分词器？为什么搜索时不能用拼音分词器？

1.ik分词器、自定义的拼音分词器

2.因为要实现自动补全功能，原始的拼音分词器功能不太行，默认的拼音分词器会将每个汉字单独分为拼音，而我们希望的是每个词条形成一组拼音所以我们自定义拼音分词器

3.这样的话当用户搜索中文时会显示这个中文的同音字,这是不合理的

## 8、ES有哪些查询类型，分别用在什么场景？如何实现复合查询？要给指定的数据进行加分如何实现？

**查询所有**：查询出所有数据，一般测试用。例如：match\_all

**全文检索（full text）查询**：利用分词器对用户输入内容分词，然后去倒排索引库中匹配。例如：

match\_query

multi\_match\_query

**精确查询**：根据精确词条值查找数据，一般是查找keyword、数值、日期、boolean等类型字段。例如：

ids

range

term

**地理（geo）查询**：根据经纬度查询。例如：

geo\_distance

geo\_bounding\_box

**复合（compound）查询**：复合查询可以将上述各种查询条件组合起来，合并查询条件。例如：

bool

function\_score

复合查询一般利用布尔查询与算分函数实现：

布尔查询是一个或多个查询子句的组合，每一个子句就是一个**子查询**。子查询的组合方式有：

must：必须匹配每个子查询，类似“与”

should：选择性匹配子查询，类似“或”

must\_not：必须不匹配，**不参与算分**，类似“非”

filter：必须匹配，**不参与算分**

加分利用算分函数实现：

function score query定义的三要素：

过滤条件：哪些文档要加分

算分函数：如何计算function score

加权方式：function score 与 query score如何运算

## 9、ES能对搜索结果进行哪些处理？如何实现高亮？

（1）排序、分页、高亮

（2）高亮是对关键字高亮，因此搜索条件必须带有关键字，而不能是范围这样的查询。

（3）默认情况下，高亮的字段，必须与搜索指定的字段一致，否则无法高亮

（4）如果要对非搜索字段高亮，则需要添加一个属性：required\_field\_match=false

"highlight": {

"fields": { // 指定要高亮的字段名称

"FIELD": { //需要高亮的字段名称

"pre\_tags": "<em>", // 用来标记高亮字段的前置标签

"post\_tags": "</em>", // 用来标记高亮字段的后置标签

"required\_field\_match": "false" //用来标记对非搜索字段高亮

}

}

}

## 10、ES有哪些聚合查询？

**桶（Bucket）**聚合：用来对文档做分组

TermAggregation：按照文档字段值分组，例如按照品牌值分组、按照国家分组

Date Histogram：按照日期阶梯分组，例如一周为一组，或者一月为一组

**度量（Metric）**聚合：用以计算一些值，比如：最大值、最小值、平均值等

Avg：求平均值、Max：求最大值、Min：求最小值、Stats：同时求max、min、avg、sum等

**管道（pipeline）**聚合：以其它聚合的结果为基础做聚合

## 11、ES如何实现自动补全查询

1）、拼音分词器

2）、自定义分词器

原理：

character filter：在tokenizer之前对文本进行处理。例如删除字符、替换字符

tokenizer：将文本按照一定的规则切割成词条（term）。例如keyword，就是不分词；还有ik\_smart

tokennizer filter：将tokenizer输出的词条做进一步处理。例如大小写转换、同义词处理、拼音处理等

位置：建索引的时候需要定义在setting中，与mapping同级

3）、自动补全（DSL、java API）

- 参与补全查询的字段必须是completion类型。

- 字段的内容一般是用来补全的多个词条形成的数组。

## 12、如何自定义分词器？

原理：

character filter：在tokenizer之前对文本进行处理。例如删除字符、替换字符

tokenizer：将文本按照一定的规则切割成词条（term）。例如keyword，就是不分词；还有ik\_smart

tokennizer filter：将tokenizer输出的词条做进一步处理。例如大小写转换、同义词处理、拼音处理等

位置：建索引的时候需要定义在setting中，与mapping同级

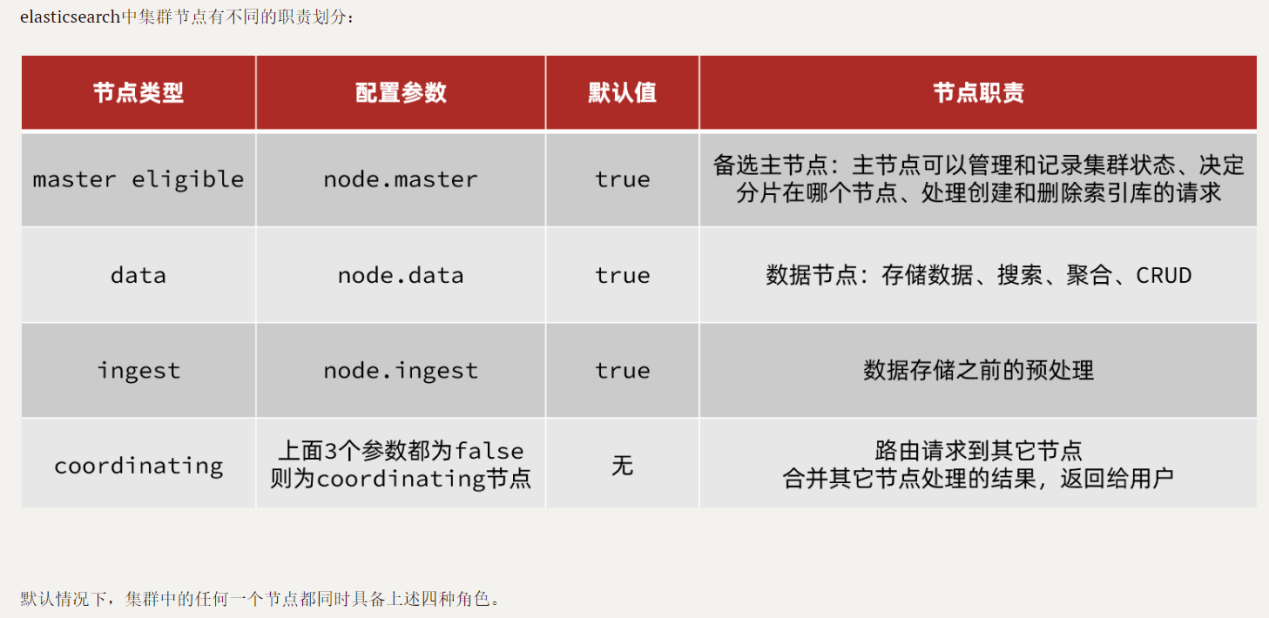
## 13、如何实现es与mysql的数据同步？

1）、直接进行微服务之间的调用，同步

2）、MQ发送消息，异步

3）、监听mysql的binlog实现

## 14、es集群节点有哪些类型？分别的职责是什么？



但是真实的集群一定要将集群职责分离：

- master节点：对CPU要求高，但是内存要求低

- data节点：对CPU和内存要求都高

- coordinating节点：对网络带宽、CPU要求高

职责分离可以让我们根据不同节点的需求分配不同的硬件去部署。而且避免业务之间的互相干扰

## 15、什么是es脑裂问题？

脑裂是因为集群中的节点失联导致的

解决脑裂的方案是，要求选票超过 ( eligible节点数量 + 1 ）/ 2 才能当选为主节点，因此eligible节点数量最好是奇数。对应配置项是discovery.zen.minimum\_master\_nodes，在es7.0以后，已经成为默认配置，因此一般不会发生脑裂问题

例如：3个节点形成的集群，选票必须超过 （3 + 1） / 2 ，也就是2票。node3得到node2和node3的选票，当选为主。node1只有自己1票，没有当选。集群中依然只有1个主节点，没有出现脑裂。

# 十一、压测、高并发性能优化

## 1、jemiter压测工具使用

## 2、多级缓存（nginx共享字典、redis缓存、tomcat进程缓存）

## 3、数据库主从读写分离（mycat）

## 4、发布：

开发环境、测试环境用的shell脚本自动发布（包括从git上拉取代码、打包编译、启动）

预上线、线上环境用的jenkins持续集成

## 5、提交代码之前需要做什么？

提交代码之前要先pull拉取一下仓库的最新代码，防止代码冲突

## 国产数据库

**阿里的数据库：OceanBase**

**腾讯的TDSQL**

**华为的GaussDB**

**达梦数据库：**简称DM8

# 额外凯子：

## 1.bean的作用域

1）singleton

单例模式，使用 singleton 定义的 Bean 在 Spring 容器中只有一个实例，这也是 Bean 默认的作用域。

2）prototype

原型模式，每次通过 Spring 容器获取 prototype 定义的 Bean 时，容器都将创建一个新的 Bean 实例。

3）request

在一次 HTTP 请求中，容器会返回该 Bean 的同一个实例。而对不同的 HTTP 请求，会返回不同的实例，该作用域仅在当前 HTTP Request 内有效。

4）session

在一次 HTTP Session 中，容器会返回该 Bean 的同一个实例。而对不同的 HTTP 请求，会返回不同的实例，该作用域仅在当前 HTTP Session 内有效。

5）global Session

在一个全局的 HTTP Session 中，容器会返回该 Bean 的同一个实例。该作用域仅在使用 portlet context 时有效。

## 2mysql的锁10几种

Mysql中的锁有哪些是什么作用是什么？

#全局锁 ：用作全量备份时，保证表和表之间的数据一致性

flush tables with read lock

使用全局读锁锁定所有数据库的所有表时，会阻塞其他的DML和DDL操作，避免备份过程中数据不一致，最后使用unlock tables解锁

#表级锁InnoDB

表锁：语法 - 加锁 lock tables 表名 read/write 解锁unlock tables

read 可以查询 不能写 共享表锁和IS不冲突

write 查询修改都不可以 排他锁

元数据锁：隐式执行的，作用避免DML和DDL冲突(crud和表定义冲突)

执行表锁会添加元数据锁

执行DDL语句会添加元数据锁

crud也会加元数据锁 不过元数据锁类型不同，DML之间的元数据锁不互斥兼容的；和DDL的元数据锁是互斥的

意向锁：IS（意向共享）和IX（意向排他）主要是避免DML与表锁冲突

DML主要是加行锁，为了表锁不用去检测没行数据是否加锁，可以使用意向锁来减少表锁的判断，意向锁之间不互斥

由DML语句添加 select……lock in share mode加 IS ；insert update delete select……for update 会加IX锁

#行级锁 注意：他们锁定的其实都是索引上的行与间隙，根据索引的有序性来确定间隙

行锁：-工作在RC RR下，锁住的是行，防止其他事务对此行update 或者 delete

间隙锁：-工作在RR下，锁住的是间隙，防止其他事务在这个间隙insert产生幻读

临键锁： -工作在RR下，锁住的一条记录和他前面的间隙，特定情况下可优化为行锁

间隙：行与行之间的空位，索引相邻的没有空位：比如主键索引中 假设表为id 1 2 3 6 7

1-2之间没有 1和无穷小有； 3-6之间就有间隙了因为4 5是索引中不存在的 ;7和无穷大之间也有间隙

1. Seata的工作原理，事务基于mybatis锁

一级锁：seata的全局锁

二级锁：mybatis锁

然后mybatis的锁全部提交完后才会释放自己的全局锁保证事务的一致性

Myvatis的锁比Seata的锁要粒度要细，轻量一点