Java 热门面试题-基础

题目 1: JVM 内存区域

运行时内存划分如下区域：

方法区:存放类,静态变量,静态方法,常量(常量不等同于常量值,基本类型都属于常量值)

(局部常量,成员常量,静态常量),成员方法,线程不安全

堆区:存放分配的对象,线程不安全

栈区:存放局部变量,以及运行的方法.

程序计数器:记录当前线程走到哪一步了.

本地方法栈:运行本地方法,一般是其他语言(本地方法相当于库函数,封装了对操作系统的

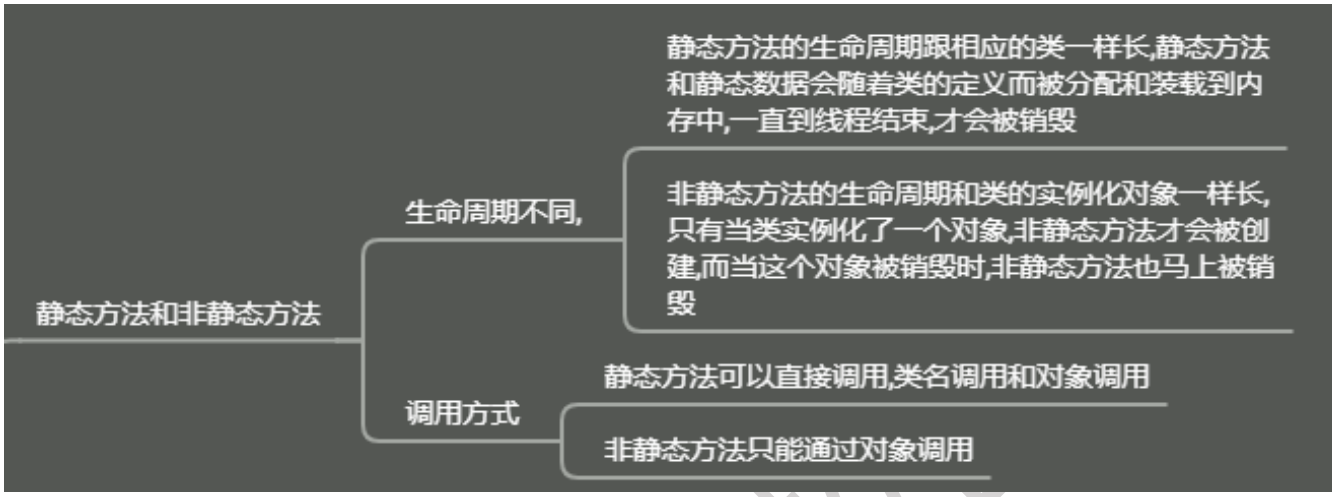
题目 2：基本数据类型，int 初始值，integer 初始值

8 中基本数据类型，int 初始 0，integer 初始 null

题目 3：静态变量与实例变量区别？静态方法与非静态方法区别？

- 静态变量： 静态变量由于不属于任何实例对象，属于类的，所以在内存中只会有一份，在类的 加载过程中，JVM 只为静态变量分配一次内存空间。
- 实例变量： 每次创建对象，都会为每个对象分配成员变量内存空间，实例变量是属于实例对象 的，在内存中，创建几次对象，就有几份成员变量。

  静态方法是所有对象共享的（公共的方法），非静态方法是实例方法，必须通过实例化对象来调用的方法。



题目 4：String、StringBuffer、StringBuilder 的区别？

- 第一点: 可变和适用范围。String 对象是不可变的，而 StringBuffer 和 StringBuilder 是可变字符序列。每次对 String 的操作相当于生成一个新的 String 对象，而对 StringBuffer 和 StringBuilder 的操作是对对象本身的操作，而不会生成新的对象，所以对于频繁改变内容的字符串避免使用 String，因为频繁的生成对象将会对系统性能产生影响。
- 第二点: 线程安全。String 由于有 final 修饰，是 immutable 的，安全性是简单而纯粹的。StringBuilder 和 StringBuffer 的区别在于 StringBuilder 不保证同步，也就是说如果需要线程安全需要使用 StringBuffer，不需要同步的 StringBuilder 效率更高。
- 总结：

  –操作少量的数据 = String

  –单线程操作字符串缓冲区下操作大量数据 = StringBuilder

  –多线程操作字符串缓冲区下操作大量数据 = StringBuffer

题目 5：String 类的常用方法有哪些？

- indexof(); 返回指定字符的的索引。
- charAt(); 返回指定索引处的字符。
- replace(); 字符串替换。

- trim(); 去除字符串两端空格。
- split()；字符串分割，返回分割后的字符串数组。
- getBytes()；返回字符串 byte 类型数组。
- length()；返回字符串长度。
- toLowerCase(); 将字符串转换为小写字母。 •toUpperCase();将字符串转换为大写字母。
- substring(); 字符串截取。
- equals(); 比较字符串是否相等。

题目 6：字符串操作：如何实现字符串的反转及替换？

可用字符串构造 StringBuffer 对象,然后调用 StringBuffer 中的 reverse 方法即可实现字符串的反转,调用 replace 方法即可实现字符串的替换。

题目 7：==与 equals 的区别？

区别 1. ==是一个运算符 equals 是 Object 类的方法

区别 2. 比较时的区别

- 用于基本类型的变量比较时: ==用于比较值是否相等，equals 不能直接用于基本数据类型的比较，需要转换为其对应的包装类型。
- 用于引用类型的比较时。==和 equals 都是比较栈内存中的地址是否相等 。相等为 true 否则为 false。但是通常会重写 equals 方法去实现对象内容的比较。

题目 8：接口和抽象类的区别是什么？

- 内容上的区别
- 不同:
- 抽象类:
- 1.抽象类中可以定义构造器

- 2.可以有抽象方法和具体方法(非静态方法/静态方法)
- 3.接口中的成员全都是 public 的
- 4.抽象类中可以定义成员变量(非静态变量/静态变量,常量)
- 5.有抽象方法的类必须被声明为抽象类，而抽象类未必要有抽象方法
- 6.抽象类中可以包含静态方法
- 7.一个类只能继承一个抽象类
-
- 接口:
- 1.接口中不能定义构造器
- 2.可以定义抽象方法,静态方法(jdk8 开始),默认方法(jdk8 开始),私有方法(jdk9 开始)
- 3.抽象类中的成员可以是 private、默认、protected、public
- 4.接口中定义的成员变量实际上都是常量
- 5.一个类可以实现多个接口
-
- 相同:
- 1.不能够实例化
- 2.都是引用类型,都一作为方法参数和返回值类型等使用
- 3.一个类如果继承了某个抽象类或者实现了某个接口都需要对其中的抽象方法全部进行实现，否则该类仍然需要 被声明为抽象类
-
- 设计层面上的区别
- 1）抽象类是对一种事物的抽象，即对类抽象，而接口是对行为的抽象。
- 抽象类是对整个类整体进行抽象，包括属性、行为，但是接口却是对类局部（行为）进行抽象。
- 2）设计层面不同，抽象类作为很多子类的父类，它是一种模板式设计。而接口是一种行为规范，它是一种辐射式设计。

题目 9：sleep 和 wait、yield 的区别？

- sleep 和 wait 都是陷入了阻塞态.醒来进入就绪态.yield 陷入就绪态.
- sleep 不会释放锁,wait 会释放锁.
- sleep 是线程方法,wait 是 object 方法.
- sleep 适用于所有场景,wait 适用于同步代码块.

- sleep 是线程类（Thread）的方法，导致此线程暂停执行指定时间，给执行机会给其他线程，但是监控状态依然保持，到时后会自动恢复。调用 sleep 不会释放对象锁。

- wait 是 Object 类的方法，对此对象调用 wait 方法导致本线程放弃对象锁，进入等待此对象的等待锁定池，只有针对此对象发出 notify 方法（或 notifyAll）后本线程才进入对象锁定池准备获得对象锁进入运行状态。

- wait()方法为什么不写在 Thread 里面？

  wait 和 notify 是两个线程之间的通信机制，每个对象都可上锁

  在 Java 中，所有对象都有一个监视器线程在监视器上等待，为执行等待，我们需要 2 个参数：一个线程和一个监视器(任何对象)

  在 Java 设计中，线程不能被指定，它总是运行当前代码的线程。但是，我们可以指定监视器(这是我们称之为等待的对象)。这是一个很好的设计，因为如果我们可以让任何其他线程在所需的监视器上等待，这将导致“入侵”，影响线程执行顺序，导致在设计并发程序时会遇到困难。请记住，在 Java 中，所有在另一个线程的执行中造成入侵的操作都被弃用了(例如 Thread.stop 方法)

  https://blog.51cto.com/u_15009384/2562837

题目 10：Java 中的 final 关键字有哪些用法？

- 修饰类：表示该类不能被继承；
- 修饰方法：表示方法不能被重写；

- 修饰变量：表示变量只能一次赋值以后值不能被修改（常量）。

题目 11：try catch 中有 return，发生了异常，是走 return 还是 finally

1. 不管有没有异常，finally 块中代码都会执行；
2. 当 try.catch 中有 return 时，finally 仍然会执行；
3. finally 中最好不要包含 return，否则程序会提前退出，返回值不是 try 或 catch 中保存的返回值。
4. 在执行时，是 return 语句先把返回值写入内存中，然后停下来等待 finally 语句块执行完，return 再执行后面的一段。
5. 至于返回值到底变不变，当 finally 调用任何可变的 API，会修改返回值；当 finally 调用任何的不可变的 API，对返回值没有影响。

题目 12：接口 1.8 后新特性

java1.8 以后,接口中可定义默认(default)和静态方法(static),这两种方法都可以有具体实现，

实现该接口的类也可继承这两种方法去直接使用，也可对其进行重写

默认方法存在的两大优势：

1. 可以让接口更优雅的升级，减少使用人员操作的负担

不必随着接口方法增加，从而修改实现代码，因为默认方法在子类中可以不用实现

2. 可以让实现类中省略很多不必要方法的空实现

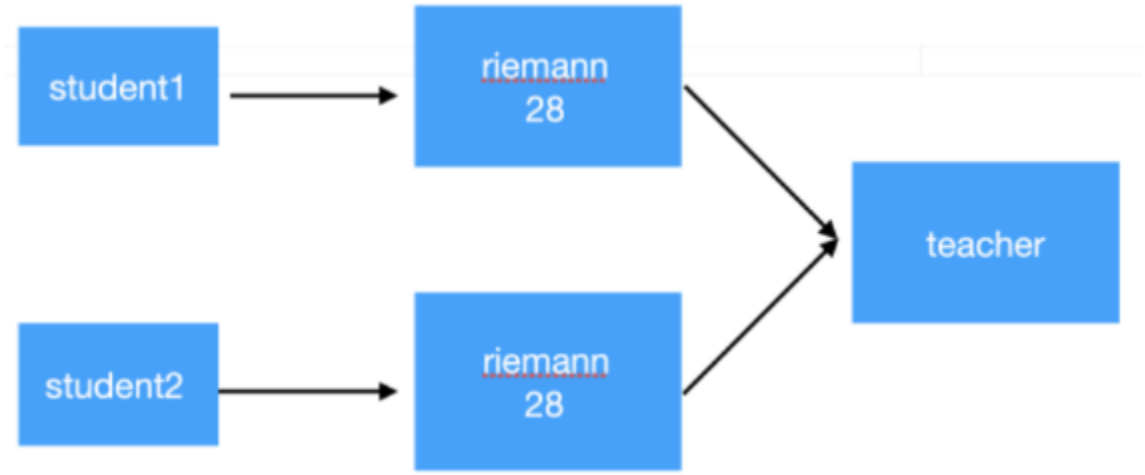
题目 13：浅拷贝 深拷贝区别

浅: 被复制对象的所有变量都含有与原来的对象相同的值，而所有的对其他对象的引用仍然指向原来的对象。即对象的浅拷贝会对“主”对象进行拷贝，但不会复制主对象里面的对象。”里面的对象“会在原来的对象和它的副本之间共享。

```
Teacher teacher = new Teacher();
teacher.setName("riemann");
teacher.setAge(28);

Student student1 = new Student();
student1.setName("edgar");
student1.setAge(18);
student1.setTeacher(teacher);

Student student2 = (Student) student1.clone();
```



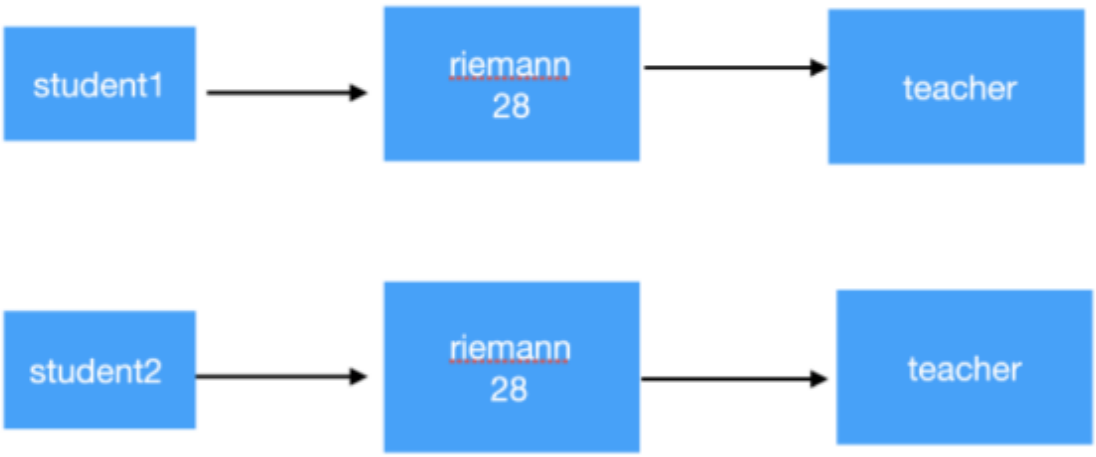
深：深拷贝是一个整个独立的对象拷贝，深拷贝会拷贝所有的属性,并拷贝属性指向的动态分配的内存。当对象和它所引用的对象一起拷贝时即发生深拷贝。深拷贝相比于浅拷贝速度较慢并且花销较大。

简而言之，深拷贝把要复制的对象所引用的对象都复制了一遍

```
Teacher teacher = new Teacher();
teacher.setName("riemann");
teacher.setAge(28);

Student student1 = new Student();
student1.setName("edgar");
student1.setAge(18);
student1.setTeacher(teacher);

Student student2 = (Student) student1.clone();
System.out.println("-------------拷贝后-------------");
```



题目 14：Java 内存泄漏

内存泄露是: **尽管对象不再被程序所使用，但垃圾回收器却无法将其回收的情况——因为对象仍然处于被引用的状态**。 久而久之，不能被回收的内存越来越多，最终导致内存溢出 OOM(OutOfMemoryError)。

1. 内存泄漏发生的重灾区——堆。因为堆区是用来存储新生的 Java 对象的地方，这里也常会有不被使用的对象未被回收。**为堆设置更小的内存**是解决堆区内存泄漏的常用方法。在我们启动程序的时候，便可以调整我们需要的内存空间：

-Xms(初始化堆内存) –Xmx（最大堆内存）

2. 静态类型的对象的引用也会导致 Java 内存泄漏。

它在全生命周期内都不会被 JVM 回收。**格外注意对关键词 static 的使用**，对任何集合或者是庞大的类进行 static 声明都会使其声明周期与 JVM 的生命周期同步，从而使其无法回收。

3.未关闭的流

忘记关闭流也是一种导致内存泄漏发生的常见情况。一个未能关闭的流会导致两种类型的泄漏，一种是低层资源泄漏，一种是内存泄漏。低层资源泄漏是 OS 层面的资源，例如文件描述符，打开连接等的泄漏。JVM 会跟踪记录这些重要的资源，进一步也就导致了内存泄漏。

4.未关闭的连接

对未关闭的连接的处理（例如数据库，FTP 服务器等）。同样，错误的实现方法也可能导致内存泄漏。使用完连接后要及时关闭。

题目 15：什么是双亲委派模型？为什么要使用双亲委派模型？

类加载器，顾名思义就是一个可以将 Java 字节码加载为 java.lang.Class 实例的工具。这个过程包括，读取字节数组、验证、解析、初始化等。另外，它也可以加载资源，包括图像文件和配置文件

类加载器的特点：

动态加载，无需在程序一开始运行的时候加载，而是在程序运行的过程中，动态按需加载，字节码的来源也很多，压缩包 jar、war 中，网络中，本地文件等。类加载器动态加载的特点为热部署，热加载做了有力支持。

全盘负责，当一个类加载器加载一个类时，这个类所依赖的、引用的其他所有类都由这个类加载器加载，除非在程序中显式地指定另外一个类加载器加载。所以破坏双亲委派不能破坏扩展类加载器以上的顺序。

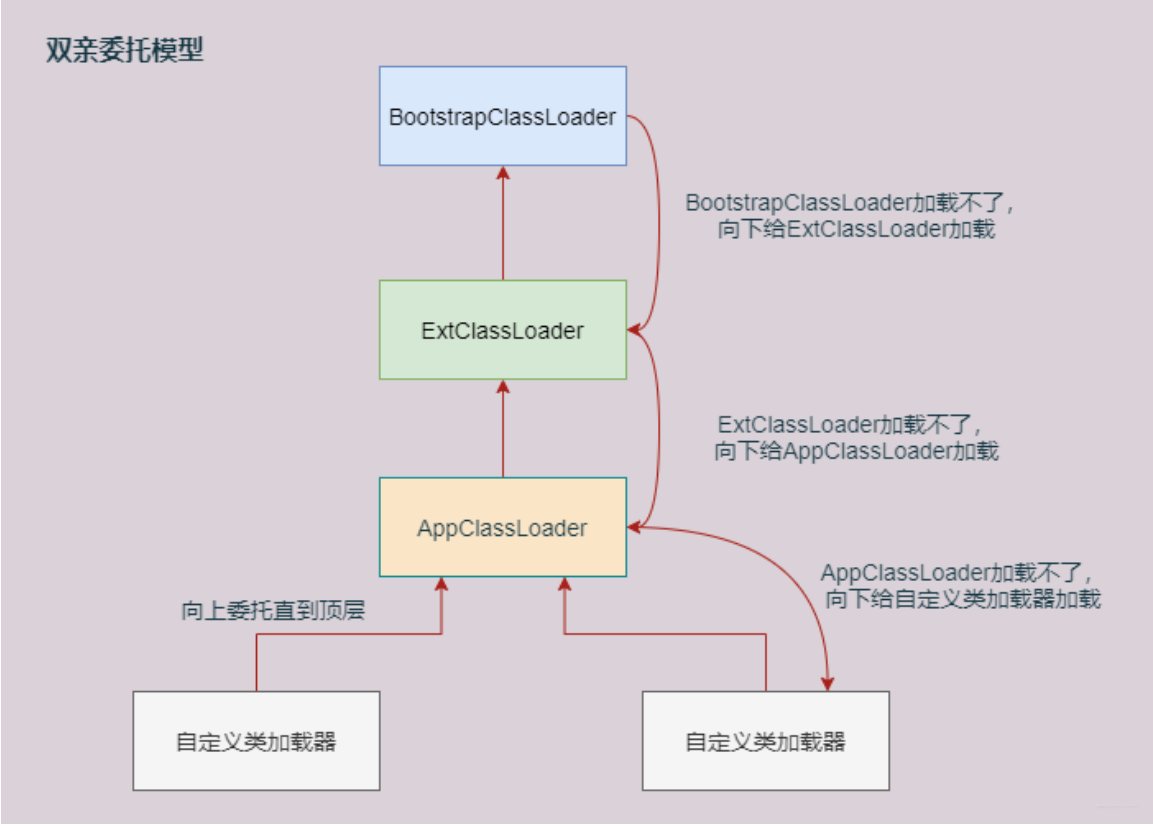
类加载器可以分为两种：一种是**启动类加载器**，由 C++语言实现，是虚拟机自身的一部分；另一种是继承于 java.lang.ClassLoader 的类加载器，包括**扩展类加载器**、**应用程序类加载器**以及自定义类加载器。

启动类加载器（`Bootstrap ClassLoader`）：负责加载`<JAVA_HOME>\lib`目录中的，或者被`-Xbootclasspath`参数所指定的路径，并且是虚拟机识别的（仅按照文件名识别，如rt.jar，名字不符合的类库即使放在lib目录中也不会被加载）类库加载到虚拟机内存中。**启动类加载器无法被Java程序直接引用**，用户在编写自定义类加载器时，如果想设置`Bootstrap ClassLoader`为其`parent`，**可直接设置null。**

扩展类加载器（`Extension ClassLoader`）：负责加载`<JAVA_HOME>\lib\ext`目录中的，或者被`java.ext.dirs`系统变量所指定路径中的所有类库。该类加载器由`sun.misc.Launcher$ExtClassLoader`实现。扩展类加载器由启动类加载器加载，其父类加载器为启动类加载器，即`parent=null`。

应用程序类加载器（`Application ClassLoader`）：负责加载用户类路径（`ClassPath`）上所指定的类库，由`sun.misc.Launcher$App-ClassLoader`实现。开发者可直接通过`java.lang.ClassLoader`中的`getSystemClassLoader()`方法获取应用程序类加载器，所以也可称它为系统类加载器。应用程序类加载器也是启动类加载器加载的，但是它的父类加载器是扩展类加载器。在一个应用程序中，系统类加载器一般是默认类加载器。

双亲委派：



为什么双亲委派：

双亲委派保证类加载器，自下而上的委派，又自上而下的加载，保证每一个类在各个类加载器中都是同一个类。

一个非常明显的目的就是保证 java 官方的类库<JAVA_HOME>\lib 和扩展类库 <JAVA_HOME>\lib\ext 的加载安全性，不会被开发者覆盖。

例如类 java.lang.Object，它存放在 rt.jar 之中，无论哪个类加载器要加载这个类，最终都是委派给启动类加载器加载，因此 Object 类在程序的各种类加载器环境中都是同一个类。

如果开发者自己开发开源框架，也可以自定义类加载器，利用双亲委派模型，保护自己框架需要加载的类不被应用程序覆盖。

Java 热门面试题-集合与 IO

- **题目 1：集合类中主要有几种接口？**

  –Collection： Collection 是集合 List、 Set、 Queue 的最基本的接口。

  –Iterator：迭代器，可以通过迭代器遍历集合中的数据

  –Map：是映射表的基础接口

- **题目 2：集合类的底层数据结构？**

  –List

  - Arraylist： Object[] 数组
  - Vector：Object[] 数组
  - LinkedList： 双向链表(JDK1.6 之前为循环链表，JDK1.7 取消了循环)

  –Set

  - HashSet(无序，唯一): 基于 HashMap 实现的，底层采用 HashMap 来保存元素
  - LinkedHashSet: LinkedHashSet 是 HashSet 的子类，并且其内部是通过 LinkedHashMap 来实现的。有点类似于我们之前说的

LinkedHashMap 其内部是基于 HashMap 实现一样，不过还是有一点点区别的

- TreeSet(有序，唯一): 红黑树(自平衡的排序二叉树)

–Queue

- PriorityQueue: Object[] 数组来实现二叉堆
- ArrayQueue: Object[] 数组 + 双指针

再来看看 Map 接口下面的集合。

–Map

- HashMap： JDK1.8 之前 HashMap 由数组+链表组成的，数组是 HashMap 的主体，链表则是主要为了解决哈希冲突而存在的（“拉链法”解决冲突）。JDK1.8 以后在解决哈希冲突时有了较大的变化，当链表长度大于阈值（默认为 8）（将链表转换成红黑树前会判断，如果当前数组的长度小于 64，那么会选择先进行数组扩容，而不是转换为红黑树）时，将链表转化为红黑树，以减少搜索时间

- LinkedHashMap： LinkedHashMap 继承自 HashMap，所以它的底层仍然是基于拉链式散列结构即由数组和链表或红黑树组成。另外，LinkedHashMap 在上面结构的基础上，增加了一条双向链表，使得上面的结构可以保持键值对的插入顺序。同时通过对链表进行相应的操作，实现了访问顺序相关逻辑。

- Hashtable： 数组+链表组成的，数组是 Hashtable 的主体，链表则是主要为了解决哈希冲突而存在的

- TreeMap： 红黑树（自平衡的排序二叉树）

- **题目 3：HashSet 如何检查重复？**

当你把对象加入 HashSet 时，HashSet 会先计算对象的 hashcode 值来判断对象加入的位置，同时也会与其他加入的对象的 hashcode 值作比较，如果没有相符的 hashcode，HashSet 会假设对象没有重复出现。但是如果发现有相同 hashcode 值的对象，这时会调用 equals()方法来检查 hashcode 相等的对象是否真的相同。如果两者相同，HashSet 就不会让加入操作成功。

- ## 题目 4：HashMap 和 HashTable 区别？

1. **线程是否安全：** HashMap 是非线程安全的，Hashtable 是线程安全的，因为 Hashtable 内部的方法基本都经过 synchronized 修饰。（如果你要保证线程安全的话就使用 ConcurrentHashMap 吧！）；
2. **效率：** 因为线程安全的问题，HashMap 要比 Hashtable 效率高一点。另外，Hashtable 基本被淘汰，不要在代码中使用它；
3. **对 Null key 和 Null value 的支持：** HashMap 可以存储 null 的 key 和 value，但 null 作为键只能有一个，null 作为值可以有多个；Hashtable 不允许有 null 键和 null 值，否则会抛出 NullPointerException。
4. **初始容量大小和每次扩充容量大小的不同 ：** ① 创建时如果不指定容量初始值，Hashtable 默认的初始大小为 11，之后每次扩充，容量变为原来的 2n+1。HashMap 默认的初始化大小为 16。之后每次扩充，容量变为原来的 2 倍。② 创建时如果给定了容量初始值，那么 Hashtable 会直接使用你给定的大小，而 HashMap 会将其扩充为 2 的幂次方大小（HashMap 中的 tableSizeFor()方法保证，下面给出了源代码）。也就是说 HashMap 总是使用 2 的幂作为哈希表的大小,后面会介绍到为什么是 2 的幂次方。
5. **底层数据结构：** JDK1.8 以后的 HashMap 在解决哈希冲突时有了较大的变化，当链表长度大于阈值（默认为 8）（将链表转换成红黑树前会判断，如果当前数组的长度小于 64，那么会选择先进行数组扩容，而不是转换为红黑树）时，将链表转化为红黑树，以减少搜索时间。Hashtable 没有这样的机制。

- ## 题目 5：ConcurrentHashMap 和 HashTable 区别？

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

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

–**实现线程安全的方式（重要）：** ① **在 JDK1.7 的时候，ConcurrentHashMap（分段锁）** 对整个桶数组进行了分割分段(Segment)，每一把锁只锁容器其中一部分数据，多线程访问容器里不同数据段的数据，就不会存在锁竞争，提高并发访问率。 **到了 JDK1.8 的时候已经摒弃了 Segment 的概念，而是直接用 Node 数组+链表+红黑树的数据结构来实现，并发控制使用 synchronized 和 CAS 来操作。（JDK1.6 以后 对 synchronized 锁做了很多优化）** 整个看起来就像是优化过且线程安全的 HashMap，虽然在 JDK1.8 中还能看到 Segment 的数据结构，但是已经简化了属性，只是为了兼容旧版本；② **Hashtable(同一把锁)** :使用 synchronized 来保证线程安全，效率非常低下。当一个线程访问同步方法时，其他线程也访问同步方法，可能会进入阻塞或轮询状态，如使用 put 添加元素，另一个线程不能使用 put 添加元素，也不能使用 get，竞争会越来越激烈效率越低。

- **题目 6：ArrayList 和 linkedList 区别？**

1. **是否保证线程安全：** ArrayList 和 LinkedList 都是不同步的，也就是不保证线程安全；
2. **底层数据结构：** Arraylist 底层使用的是 **Object 数组**；LinkedList 底层使用的是 **双向链表** 数据结构（JDK1.6 之前为循环链表，JDK1.7 取消了循环。注意双向链表和双向循环链表的区别，下面有介绍到！）
3. 插入和删除是否受元素位置的影响：

   –ArrayList 采用数组存储，所以插入和删除元素的时间复杂度受元素位置的影响。 比如：执行 add(E e)方法的时候， ArrayList 会默认在将指定的元素追加到此列表的末尾，这种情况时间复杂度就是 O(1)。但是如果要在指定位置 i 插入和删除元素的话（add(int index, E element)）时间复杂度就为 O(n-i)。因为在进行上述操作的时候集合中第 i 和第 i 个元素之后的(n-i)个元素都要执行向后位/向前移一位的操作。

   –LinkedList 采用链表存储，所以，如果是在头尾插入或者删除元素不受元素位置的影响（add(E e)、addFirst(E e)、addLast(E e)、removeFirst() 、removeLast()），近似 O(1)，如果是要在指定位置 i 插入和删除元素的话（add(int index, E element)，remove(Object o)） 时间复杂度近似为 O(n) ，因为需要先移动到指定位置再插入。
4. **是否支持快速随机访问：** LinkedList 不支持高效的随机元素访问，而 ArrayList 支持。快速随机访问就是通过元素的序号快速获取元素对象(对应于 get(int index)方法)。

5. **内存空间占用：** ArrayList 的空 间浪费主要体现在在 list 列表的结尾会预留一定的容量空间，而 LinkedList 的空间花费则体现在它的每一个元素都需要消耗比 ArrayList 更多的空间（因为要存放直接后继和直接前驱以及数据）。

• 题目 7：HashMap 底层实现原理？

–JDK1.8 之前

`HashMap` 底层是 **数组和链表** 结合在一起使用也就是 **链表散列**。**HashMap 通过 key 的 hashCode 经过扰动函数处理过后得到 hash 值，然后通过 (n - 1) & hash 判断当前元素存放的位置（这里的 n 指的是数组的长度），如果当前位置存在元素的话，就判断该元素与要存入的元素的 hash 值以及 key 是否相同，如果相同的话，直接覆盖，不相同就通过拉链法解决冲突。**

所谓扰动函数指的就是 HashMap 的 hash 方法。使用 hash 方法也就是扰动函数是为了防止一些实现比较差的 hashCode() 方法 换句话说使用扰动函数之后可以减少碰撞。

–JDK1.8 及以后

相比于之前的版本， JDK1.8 之后在解决哈希冲突时有了较大的变化，当链表长度大于阈值（默认为 8）（将链表转换成红黑树前会判断，如果当前数组的长度小于 64，那么会选择先进行数组扩容，而不是转换为红黑树）时，将链表转化为红黑树，以减少搜索时间。

TreeMap、TreeSet 以及 JDK1.8 之后的 HashMap 底层都用到了红黑树。红黑树就是为了解决二叉查找树的缺陷，因为二叉查找树在某些情况下会退化成一个线性结构。

• 题目 8：HashMap 什么时候扩容？

当 hashmap 中的元素个数超过数组大小 loadFactor 时，就会进行数组扩容，loadFactor 的默认值为 0.75，也就是说，默认情况下，数组大小为 16，那么当 hashmap 中元素个数超过 160.75=12 的时候，就把数组的大小扩展为 216=32，即扩大一倍，然后重新计算每个元素在数组中的位置，而这是一个非常消耗性能的操作，所以如果我们已经预知 hashmap 中元素的个数，那么预设元素的个数能够有效的提高 hashmap 的性能。比如说，我们有 1000 个元素 new HashMap(1000),但是理论上来讲 new HashMap(1024) 更合适，不过上面 annegu 已经说过，即使是 1000，hashmap 也自动会将其设置为 1024。 但

是 new HashMap(1024) 还不是更合适的，因为 0.75*1000 < 1000, 也就是说为了让 0.75* size > 1000, 我们必须这样 new HashMap(2048) 才最合适，既考虑了 & 的问题，也避免了 resize 的问题。

- **题目 9：HashMap 中的 key 我们可以使用任何类作为 key 吗？**

平时可能大家使用的最多的就是使用 String 作为 HashMap 的 key，但是现在我们想使用某个自定

义类作为 HashMap 的 key，那就需要注意以下几点：

–如果类重写了 equals 方法，它也应该重写 hashCode 方法。

–类的所有实例需要遵循与 equals 和 hashCode 相关的规则。

–如果一个类没有使用 equals，你不应该在 hashCode 中使用它。

–咱们自定义 key 类的最佳实践是使之为不可变的，这样，hashCode 值可以被缓存起来，拥有更好的性能。不可变的类也可以确保 hashCode 和 equals 在未来不会改变，这样就会解决与可变相关的问题了。

- **题目 10：HashMap 的长度为什么是 2 的 N 次方呢？**

为了能让 HashMap 存数据和取数据的效率高，尽可能地减少 hash 值的碰撞，也就是说尽量把数

据能均匀的分配，每个链表或者红黑树长度尽量相等。

我们首先可能会想到 % 取模的操作来实现。

下面是回答的重点哟：

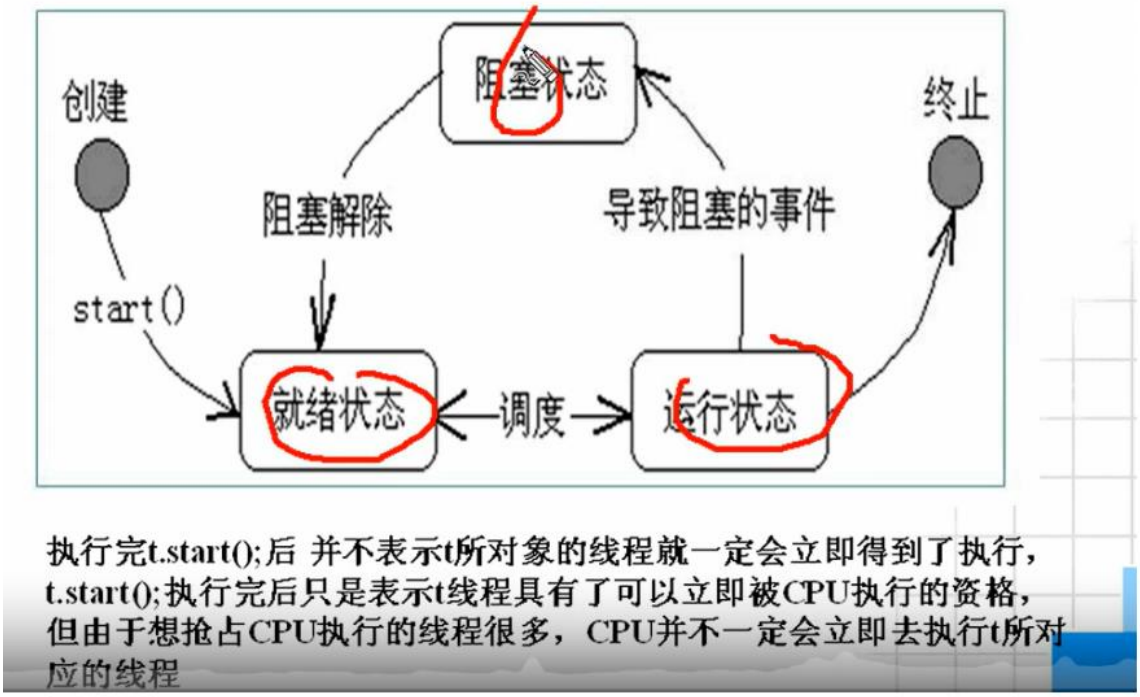
取余（%）操作中如果除数是 2 的幂次，则等价于与其除数减一的与（&）操作（也就是说 hash % length == hash &(length - 1) 的前提是 length 是 2 的 n 次方）。并且，采用二进制位操作 & ，相对于 % 能够提高运算效率。

这就是为什么 HashMap 的长度需要 2 的 N 次方了

。

Java 热门面试题-线程

题目 1：线程的状态？



- 新建(New)

  创建后尚未启动。

- 可运行(Runnable)（就绪状态+运行状态）

  可能正在运行，也可能正在等待 CPU 时间片。

- 阻塞(Blocking)

  1、等待获取一个排它锁，如果其线程释放了锁就会结束此状态。

  2、等待其它线程显式地唤醒，否则不会被分配 CPU 时间片。

进入方法	退出方法
没有设置 Timeout 参数的 Object.wait()	Object.notify() /

方法	Object.notifyAll()
没有设置 Timeout 参数的 Thread.join() 方法	被调用的线程执行完毕
LockSupport.park() 方法	-

3、限期等待(Timed Waiting)

无需等待其它线程显式地唤醒，在一定时间之后会被系统自动唤醒。

调用 Thread.sleep() 方法使线程进入限期等待状态时，常常用“使一个线程睡眠”进行描述。

调用 Object.wait() 方法使线程进入限期等待或者无限期等待时，常常用“挂起一个线程”进行描述。

睡眠和挂起是用来描述行为，而阻塞和等待用来描述状态。

阻塞和等待的区别在于，阻塞是被动的，它是在等待获取一个排它锁。而等待是主动的，通过调用 Thread.sleep() 和 Object.wait() 等方法进入。

进入方法	退出方法
Thread.sleep() 方法	时间结束
设置了 Timeout 参数的 Object.wait() 方法	时间结束 / Object.notify() / Object.notifyAll()
设置了 Timeout 参数的 Thread.join() 方法	时间结束 / 被调用的线程执行完毕
LockSupport.parkNanos() 方法	—-
LockSupport.parkUntil() 方法	—-

- 死亡(Terminated)

可以是线程结束任务之后自己结束，或者产生了异常而结束

题目 2：start 和 run 的区别？

1. start（）方法来启动线程，真正实现了多线程运行。这时无需等待 run 方法体代码执行完毕，可以直接继续执行下面的代码。
2. 通过调用 Thread 类的 start()方法来启动一个线程， 这时此线程是处于就绪状态， 并没有运行。

3. 方法 run()称为线程体，它包含了要执行的这个线程的内容，线程就进入了运行状态，开始运行 run 函数当中的代码。 Run 方法运行结束， 此线程终止。然后 CPU 再调度其它线程。

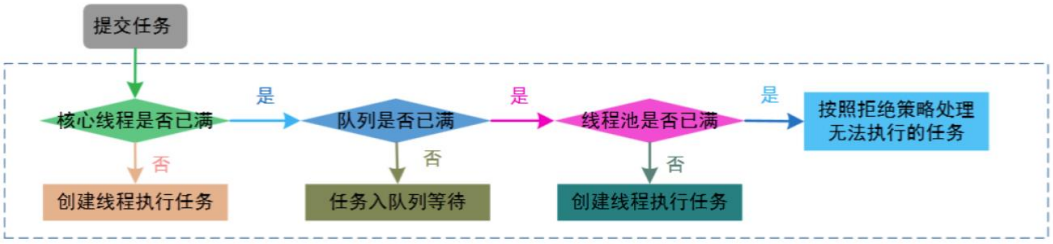
题目 3：为什么使用线程池，优势是什么？

线程的创建和销毁是比较重且好资源的操作，Java 线程依赖于内核线程，创建线程需要进行操作系统状态切换，为避免过度的消耗资源。需要想办法重用线程去执行多个任务，就可以利用线程池技术解决。

线程池做的工作主要是控制运行的线程的数量，处理过程中将任务放入队列，然后在线程创建后启动这些任务，如果线程数量超过了最大数量超出数量的线程排队等候，等其它线程执行完毕，再从队列中取出任务来执行。 他的主要特点为： 线程复用； 控制最大并发数； 管理线程。

线程池优势： 1、重用存在的线程，减少线程创建和销毁的开销，提高性能（陈低资源消耗） 2、提高响应速度，当任务达到时，任务可以不需要等待线程创建就能立即执行（提高响应速度） 3、提高线程的可管理性，统一对线程进行分配、调优和监控（增强可管理性）

题目 4：线程池工作原理？



1. 线程池刚创建时，里面没有一个线程。任务队列是作为参数传进来的。不过，就算队列里面

   有任务，线程池也不会马上执行它们。

2. 当调用 execute() 方法添加一个任务时，线程池会做如下判断：

   a) 如果正在运行的线程数量小于 corePoolSize，那么马上创建线程运行这个任务；

   b) 如果正在运行的线程数量大于或等于 corePoolSize，那么将这个任务放入队列；

c) 如果这时候队列满了，而且正在运行的线程数量小于 maximumPoolSize，那么还是要

创建非核心线程立刻运行这个任务；

d) 如果队列满了，而且正在运行的线程数量大于或等于 maximumPoolSize，那么线程池

会抛出异常 RejectExecutionException。

3. 当一个线程完成任务时，它会从队列中取下一个任务来执行。
4. 当一个线程无事可做，超过一定的时间（keepAliveTime）时，线程池会判断，如果当前运

行的线程数大于 corePoolSize，那么这个线程就被停掉。所以线程池的所有任务完成后，它

最终会收缩到 corePoolSize 的大小。

题目 5：线程池重要参数有哪些？

```
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          ThreadFactory threadFactory,
                          RejectedExecutionHandler handler)
```

1. corePoolSize => 线程池核心线程数量
2. maximumPoolSize => 线程池最大数量（包含核心线程数量）
3. keepAliveTime => 当前线程池数量超过 corePoolSize 时，多余的空闲线程的存活时间，即多

次时间内会被销毁。

4. unit => keepAliveTime 的单位
5. workQueue => 线程池所使用的缓冲队列，被提交但尚未被执行的任务
6. threadFactory => 线程工厂，用于创建线程，一般用默认的即可
7. handler => 拒绝策略，当任务太多来不及处理，如何拒绝任务

题目 6：线程池如何使用？线程池核心线程数配置多大？有没有经验配置值。

Excutors 可以创建线程池的常见 4 种方式：

1. newSingleThreadExecutor：只会创建一个线程执行任务。（适用于需要保证顺序执行各个任 务；并且在任意时间点，没有多线程活动的场景。）SingleThreadExecutorl 也使用无界队列 LinkedBlockingQueue 作为工作队列 若多余一个任务被提交到该线程池，任务会被保存在一个任务队列中，待线程空闲，按先入先 出的顺序执行队列中的任务。

2. newFixedThreadPool：可重用固定线程数的线程池。（适用于负载比较重的服务器） FixedThreadPool 使用无界队列 LinkedBlockingQueue 作为线程池的工作队列 该线程池中的线程数量始终不变。当有一个新的任务提交时，线程池中若有空闲线程，则立即 执行。若没有，则新的任务会被暂存在一个任务队列中，待有线程空闲时，便处理在任务队列 中的任务。

3. newCachedThreadPool：是一个会根据需要调整线程数量的线程池。（大小无界，适用于执行很 多的短期异步任务的小程序，或负载较轻的服务器）CachedThreadPool 使用没有容量的 SynchronousQueue 作为线程池的工作队列，但 CachedThreadPool 的 maximumPool 是无界的。 线程池的线程数量不确定，但若有空闲线程可以复用，则会优先使用可复用的线程。若所有线 程均在工作，又有新的任务提交，则会创建新的线程处理任务。所有线程在当前任务执行完毕 后，将返回线程池进行复用。

4. newScheduledThreadPool：继承自 ThreadPoolExecutor。它主要用来在给定的延迟之后运行 任务，或者定期执行任务。使用 DelayQueue 作为任务队列。

企业最佳实践：不要使用 Executors 直接创建线程池，会出现 OOM 问题，要使用 ThreadPoolExecutor 构造方法创建，引用自《阿里巴巴开发手册》

> 【强制】线程池不允许使用 Executors 去创建，而是通过 ThreadPoolExecutor 的方式，这 样的处理方式让写的同学更加明确线程池的运行规则，规避资源耗尽的风险。 说明：Executors 返回的线程池对象的弊端如下： 1） FixedThreadPool 和 SingleThreadPool： 允许的请求队列长度为 Integer.MAX_VALUE，可能会堆积大量的请求，从而导致 OOM。2） CachedThreadPool： 允许的创建线程数量为 Integer.MAX_VALUE，可能会创建大量的线程，从而导致 OOM。

创建线程池方式一：new ThreadPoolExecutor 方式

```
ExecutorService executorService = new ThreadPoolExecutor(3,5,10, TimeUnit.SECONDS,new ArrayBlockingQueue<>(3), Executors.defaultThreadFactory(), new ThreadPoolExecutor.AbortPolicy());
for (int i = 0; i < 9; i++) {
executorService.execute(()->{
System.out.println(Thread.currentThread().getName() + "开始办理业务了。。。。。。");
});
}
```

创建线程池方式二：spring 的 ThreadPoolTaskExecutor 方式

```
@Configuration
public class ExecturConfig {
    @Bean("taskExector")
    public Executor taskExector() {
        ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
        executor.setCorePoolSize(3);//核心池大小
        executor.setMaxPoolSize(5);//最大线程数
        executor.setQueueCapacity(3);//队列长度
        executor.setKeepAliveSeconds(10);//线程空闲时间
        executor.setThreadNamePrefix("tsak-asyn");//线程前缀名称
        executor.setRejectedExecutionHandler(new ThreadPoolExecutor.AbortPolicy());//配置拒绝策略
        return executor;
    }
}
```

如何配置线程数？

线程池使用面临的核心的问题在于：**线程池的参数并不好配置**。

一方面线程池的运行机制不是很好理解，配置合理需要强依赖开发人员的个人经验和知识；

另一方面，线程池执行的情况和任务类型相关性较大，IO 密集型和 CPU 密集型的任务运行起来的情况差异非常大。这导致业界并没有一些成熟的经验策略帮助开发人员参考。

配置线程数量之前，首先要看任务的类型是 IO 密集型，还是 CPU 密集型？什么是 IO 密集型？

比如：频繁读取磁盘上的数据，或者需要通过网络远程调用接口。

什么是 CPU 密集型？

比如：非常复杂的调用，循环次数很多，或者递归调用层次很深等。

IO 密集型配置线程数经验值是：2N，其中 N 代表 CPU 核数。

CPU 密集型配置线程数经验值是：N + 1，其中 N 代表 CPU 核数。

线程数的设置需要考虑三方面的因素，服务器的配置、服务器资源的预算和任务自身的特性。具体来说就是服务器有多少个 CPU，多少内存，IO 支持的最大 QPS 是多少，任务主要执行的是计算、IO 还是一些混合操作，任务中是否包含数据库连接等的稀缺资源。线程池的线程数设置主要取决于这些因素。

如果一个请求中,计算机操作需要 5ms,DB 需要 100ms,对于一个 8 核的 cpu 来说需要设置多少个线程

1.可以拆分为两个线程池,cpu 密集型的是 n+1 个线程,,IO 密集型的就是 n*2

2.如果不可以拆分,就是 5/(5+100),cpu 的利用率是在 21%,1/(5/(100+5)) *n 就是 168,也就是需要的线程数

那么如何获取 n 的值?(也就是 cpu 核心数目)

```
int  n = Runtime.getRuntime().availableProcessors();
```

题目 7：线程池的拒绝策略有哪些？

1、ThreadPoolExecutor.AbortPolicy：丢弃任务抛出 RejectedExecutionException 异常打断当前执行流程(默认)

使用场景：ExecutorService 接口的系列 ThreadPoolExecutor 因为都没有显示的设置拒绝策略，所以默认的都是这个。ExecutorService 中的线程池实例队列都是无界的，也就是说把内存撑爆了都不会触发拒绝策略。当自己自定义线程池实例时，使用这个策略一定要处理好触发策略时抛的异常，因为他会打断当前的执行流程。

2、ThreadPoolExecutor.CallerRunsPolicy：只要线程池没有关闭，就由提交任务的当前线程处理。

使用场景：一般在不允许失败的、对性能要求不高、并发量较小的场景下使用，因为线程池一般情况下不会关闭，也就是提交的任务一定会被运行，但是由于是调用者线程自己执行的，当多次提交任务时，就会阻塞后续任务执行，性能和效率自然就慢了。

3、ThreadPoolExecutor.DiscardPolicy：直接静悄悄的丢弃这个任务，不触发任何动作

使用场景：如果提交的任务无关紧要，你就可以使用它 。因为它就是个空实现，会悄无声息的吞噬你的的任务。所以这个策略基本上不用了

4、ThreadPoolExecutor.DiscardOldestPolicy：丢弃队列最前面的任务，重新提交被拒绝的任务

使用场景：这个策略还是会丢弃任务，丢弃时也是毫无声息，但是特点是丢弃的是老的未执行的任务，而且是待执行优先级较高的任务。基于这个特性，我能想到的场景就是，发布消息，和修改消息，当消息发布出去后，还未执行，此时更新的消息又来了，这个时候未执行的消息的版本比现在提交的消息版本要低就可以被丢弃了。因为队列中还有可能存在消息版本更低的消息会排队执行，所以在真正处理消息的时候一定要做好消息的版本比较。

以上内置拒绝策略均实现了 RejectedExecutionHandler 接口，若以上策略仍无法满足实际

需要，完全可以自己扩展 RejectedExecutionHandler 接口。

题目 8：线程池状态有哪些？

1.RUNNING 状态说明：线程池处在 RUNNING 状态时，能够接收新任务，以及对已添加的任务进行处理。 状态切换：线程池的初始化状态是 RUNNING。换句话说，线程池被一旦被创建，就处于 RUNNING 状态，并且线程池中的任务数为 0

2.SHUTDOWN 状态说明：线程池处在 SHUTDOWN 状态时，不接收新任务，但能处理已添加的任务。 状态切换：调用线程池的 shutdown()接口时，线程池由 RUNNING -> SHUTDOWN。

3.STOP 状态说明：线程池处在 STOP 状态时，不接收新任务，不处理已添加的任务，并且会中断正在处理的任务。 状态切换：调用线程池的 shutdownNow()接口时，线程池由(RUNNING or SHUTDOWN) -> STOP。

4.TIDYING 状态说明：当所有的任务已终止，ctl 记录的”任务数量”为 0，线程池会变为 TIDYING 状态。当线程池变为 TIDYING 状态时，会执行钩子函数 terminated()。terminated()在 ThreadPoolExecutor 类中是空的，若用户想在线程池变为 TIDYING 时，进行相应的处理；可以通过重载 terminated()函数来实现。 状态切换：当线程池在 SHUTDOWN 状态下，阻塞队列为空并且线程池中执行的任务也为空时，就会由 SHUTDOWN -> TIDYING。 当线程池在 STOP 状态下，线程池中执行的任务为空时，就会由 STOP -> TIDYING。

5.TERMINATED 状态说明：线程池彻底终止，就变成 TERMINATED 状态。 状态切换：线程池处在 TIDYING 状态时，执行完 terminated()之后，就会由 TIDYING -> TERMINATED。

题目 9：核心线程和非核心线程的销毁机制？

核心线程默认不会销毁，非核心线程要等到 keepAliveTime 后才销毁。如果需要回收核心线程数，需要调用下面的方法：

```
/**
 * Sets the policy governing whether core threads may time out and
 * terminate if no tasks arrive within the keep-alive time, being
 * replaced if needed when new tasks arrive. When false, core
 * threads are never terminated due to lack of incoming
 * tasks. When true, the same keep-alive policy applying to
 * non-core threads applies also to core threads. To avoid
 * continual thread replacement, the keep-alive time must be
 * greater than zero when setting {@code true}. This method
 * should in general be called before the pool is actively used.
 *
 * @param value {@code true} if should time out, else {@code false}
 * @throws IllegalArgumentException if value is {@code true}
 *         and the current keep-alive time is not greater than zero
 *
 * @since 1.6
 */
public void allowCoreThreadTimeOut(boolean value) {
    if (value && keepAliveTime <= 0)
        throw new IllegalArgumentException("Core threads must have nonzero keep alive times");
    if (value != allowCoreThreadTimeOut) {
        allowCoreThreadTimeOut = value;
        if (value)
            interruptIdleWorkers();
    }
}
```

题目 10：线程池内抛出异常，线程池会怎么办？

当线程池中线程执行任务的时候，任务出现未被捕获的异常的情况下，线程池会将允许该任务的线程从池中移除并销毁，且同时会创建一个新的线程加入到线程池中；可以通过 ThreadFactory 自定义线程并捕获线程内抛出的异常，也就是说甭管我们是否去捕获和处理线程池中工作线程抛出的异常，这个线程都会从线程池中被移除

题目 11：submit 和 execute 方法的区别？

1、参数有区别，都可以是 Runnable，submit 也可以是 Callable

2、submit 有返回值，而 execute 没有

3、submit 方便 Exception 处理

题目 12：线程池在实际项目中的使用场景？项目中多个业务需要用到线程池是为每个线程池都定义一个还是定义一个公共的线程池呢？

线程池一般用于执行多个不相关联的耗时任务，没有多线程的情况下，任务顺序执行，使用了线程池的话可让多个不相关联的任务同时执行。

一般建议是不同的业务使用不同的线程池，配置线程池的时候根据当前业务的情况对当前线程池进行配置，因为不同的业务的并发以及对资源的使用情况都不同，重心优化系统性能瓶颈相关的业务。

题目 13：说说 synchronized 的实现原理？

在 Java 中，每个对象都隐式包含一个 monitor（监视器）对象，加锁的过程其实就是竞争 monitor 的过程，当线程进入字节码 monitorenter 指令之后，线程将持有 monitor 对象， 执行 monitorexit 时释放 monitor 对象，当其他线程没有拿到 monitor 对象时，则需要阻塞 等待获取该对象。

题目 14：Synchronized 作用范围?

1. 作用于方法时，锁住的是对象的实例(this)；
2. 当作用于静态方法时，锁住的是 Class 实例，又因为 Class 的相关数据存储在永久带 PermGen（jdk1.8 则是 metaspace），永久带是全局共享的，因此静态方法锁相当于类的一个全局锁，会锁所有调用该方法的线程；
3. synchronized 作用于一个对象实例时，锁住的是所有以该对象为锁的代码块。它有多个队列，当多个线程一起访问某个对象监视器的时候，对象监视器会将这些线程存储在不同的容器中

synchronized 修饰代码块：只作用于同一个对象，如果不同对象调用该代码块就不会同步。

synchronized 修饰方法：和同步代码块一样，只作用于同一个对象。

synchronized 修饰静态方法：作用于整个类。

synchronized 修饰类：作用于整个类，两个线程调用同一个类的不同对象上的这种同步

语句，也会进行同步。

题目 15：synchronized 和 volatile 的区别？

volatile 关键字是线程同步的轻量级实现，所以 volatile 性能比 synchronized 关键字要好。但 是 volatile 关键字只能用于变量而 synchronized 关键字可以修饰方法以及代码块。

多线程访问 volatile 关键字不会发生阻塞，而 synchronized 关键字可能会发生阻塞。volatile 关键字主要用于解决变量在多个线程之间的可见性，而 synchronized 关键字解决的 是多个线程之间访问资源的同步性。

volatile 关键字能保证数据的可见性，但不能保证数据的原子性。synchronized 关键字两者都 能保证。

题目 16：Java 中的锁

锁就是用来控制多个线程访问共享资源的方式，简单来说，一个锁能够防止多个线程同时访问共享资源。

1. 悲观锁和乐观锁

悲观锁：认为自己在使用数据的时候一定有别的线程来修改数据，在获取数据的时候会先加锁，确保数据不会被别的线程修改。

锁实现：关键字 synchronized、接口 Lock 的实现类
适用场景：写操作较多，先加锁可以保证写操作时数据正确

乐观锁：认为自己使用数据时不会有别的线程修改数据，所以不会添加锁，只是在更新数据的时候去判断之前有没有别的线程更新了这个数据

锁实现 1：CAS 算法， CAS 即 Compare And Swap，是一种更新的**原子操作**，比较当前值跟传入值是否一样，一样则更新，否则返回 false，不进行任何操作；例如 ActomicInteger 类的原子自增是通过 CAS 自选实现。

锁实现 2：版本号控制：数据表中加上版本号字段 version，表示数据被修改的次数。当数据被修改时，这个字段值会加 1，提交必须满足“ **提交版本必须大于记录当前版本才能执行更新** “ 的乐观锁策略

适用场景：读操作较多，不加锁的特点能够使其读操作的性能大幅提升

2. 自旋锁

自选锁即是指当一个线程在获取锁的时候，如果锁已经被其它线程获取，那么该线程将循环等待（不放弃 CPU 资源），然后不断的判断锁是否能够被成功获取，直到获取到锁才会退出循环，此时获取锁的线程一直处于活跃状态（而非阻塞）。

意义：自旋锁不会使线程状态发生切换。不会使线程进入阻塞状态，减少了不必要的上下文切换，执行速度快

缺点：如果某个线程持有锁的时间过长，就会导致其它等待获取锁的线程进入循环等待，消耗 CPU。使用不当会造成 CPU 使用率极高。

自旋锁和 CAS 的关系是什么？ CAS 是实现自旋锁的基础。基本 CAS 操作+循环就可以实现自旋锁。

3. 读写锁

读写锁既是互斥锁，又是共享锁，read 模式是共享，write 是互斥(排它锁)的。

描述：synchronized 和 RentrantLock 都是排他锁，这些锁在同一时刻只允许一个线程进行访问，而读写锁在同一时刻可以允许多个读线程访问，但是在写线程访问时，所有的读线程和其他写线程均被阻塞。读写锁维护了一对锁，一个读锁和一个写锁，通过分离读锁和写锁，使得并发性相比一般的排他锁有了很大提升。

锁实现：ReentrantReadWriteLock

4. 可重入锁&非可重入锁

可重入锁又名递归锁，是指在同一个线程在外层方法获取锁的时候，再进入该线程的内层方法会自动获取锁（前提锁对象得是同一个对象或者 class），不会因为之前已经获取过还没释放而阻塞。ReentrantLock 和 synchronized 都是可重入锁。

```java
synchronized void setA() throws Exception{
   Thread.sleep(1000);
   setB();
}
synchronized void setB() throws Exception{
   Thread.sleep(1000);
}
```

登录后复制

上面的代码就是一个可重入锁的一个特点，如果不是可重入锁的话，setB可能不会被当前线程执行，可能造成死锁。

不可重入锁：与可重入相反，获取锁后不能重复获取，否则会死锁（自己锁自己）

Java 热门面试题-Web

类型	特点			性能		应用场景	首部字节
	是否面向连接	传输可靠性	传输形式	传输效率	所需资源		
TCP	面向连接	可靠	字节流	慢	多	要求通信数据可靠 （如文件传输、邮件传输）	20-60
UDP	无连接	不可靠	数据报文段	快	少	要求通信速度高 （如域名转换）	8个字节 （由4个字段组成）

题目 1：HTTP 响应状态码有什么特点？

	类别	原因短语
1XX	Informational（信息性状态码）	接收的请求正在处理
2XX	Success（成功状态码）	请求正常处理完毕
3XX	Redirection（重定向状态码）	需要进行附加操作以完成请求
4XX	Client Error（客户端错误状态码）	服务器无法处理请求
5XX	Server Error（服务器错误状态码）	服务器处理请求出错

- 1xx：指示信息–表示请求已接收，继续处理
- 2xx：成功–表示请求已被成功接收、理解、接受
- 3xx：重定向–要完成请求必须进行更进一步的操作

- 4xx：客户端错误–请求有语法错误或请求无法实现
- 5xx：服务器端错误–服务器未能实现合法的请求
- **常见的状态码：**

➢ 200：请求被正常处理

➢ 204：请求被受理但没有资源可以返回

➢ 206：客户端只是请求资源的一部分，服务器只对请求的部分资源执行 GET 方法，相应报文中通过 Content-Range 指定范围的资源。

➢ 301：永久性重定向

➢ 302：临时重定向

➢ 303：与 302 状态码有相似功能，只是它希望客户端在请求一个 URI 的时候，能通过 GET 方法重定向到另一个 URI 上

➢ 304：发送附带条件的请求时，条件不满足时返回，与重定向无关

➢ 307：临时重定向，与 302 类似，只是强制要求使用 POST 方法

➢ 400：请求报文语法有误，服务器无法识别

➢ 401：请求需要认证

➢ 403：请求的对应资源禁止被访问

➢ 404：服务器无法找到对应资源

➢ 500：服务器内部错误

➢ 503：服务器正忙

题目 2：HTTP 协议包括哪些请求？

- GET：对服务器资源的简单请求
- POST：用于发送包含用户提交数据的请求
- HEAD：类似于 GET 请求，不过返回的响应中没有具体内容，用于获取报头
- PUT：传说中请求文档的一个版本
- DELETE：发出一个删除指定文档的请求

- TRACE：发送一个请求副本，以跟踪其处理进程
- OPTIONS：返回所有可用的方法，检查服务器支持哪些方法
- CONNECT：用于 ssl 隧道的基于代理的请求

题目 3：Get 和 Post 的区别？

	GET	POST
后退按钮/刷新	无害	数据会被重新提交（浏览器应该告知用户数据会被重新提交）。
书签	可收藏为书签	不可收藏为书签
缓存	能被缓存	不能缓存
编码类型	application/x-www-form-urlencoded	application/x-www-form-urlencoded 或 multipart/form-data。为二进制数据使用多重编码。
历史	参数保留在浏览器历史中。	参数不会保存在浏览器历史中。
对数据长度的限制	是的。当发送数据时，GET 方法向 URL 添加数据；URL 的长度是受限制的（URL 的最大长度是 2048 个字符）。	无限制。
对数据类型的限制	只允许 ASCII 字符。	没有限制。也允许二进制数据。
安全性	与 POST 相比，GET 的安全性较差，因为所发送的数据是 URL 的一部分。 在发送密码或其他敏感信息时绝不要使用 GET！	POST 比 GET 更安全，因为参数不会被保存在浏览器历史或 web 服务器日志中。
可见性	数据在 URL 中对所有人都是可见的。	数据不会显示在 URL 中。

GET：

- get 重点是从服务器上获取资源
- get 传输数据是通过 URL 请求，以 field（字段） = value 的形式，置于 URL 后，并用“？”连接，多个请求数据间用“&”连接
- get 传输数据量小，因为受 URL 长度限制，但是效率高
- get 是不安全的，因为 URL 是可见的，可能会泄漏私密信息
- get 方式只能支持 ASCII 字符，向服务器传的中文字符可能会乱码

POST：

- post 重点是向服务器发送数据。
- post 传输数据是通过 HTTP 的 post 机制。将字段和对应值封存在请求实体中发送给服务器。这个过程用户是不可见的
- post 可以传输大量数据，所以上传文件时只能用 post

- post 支持标准字符集，可以正确传递中文字符
- post 较 get 安全性高

总结：

- GET 用于获取信息，无副作用，幂等，且可缓存
- POST 用于修改服务器上的数据，有副作用，非幂等，不可缓存

题目 4：HTTP 中重定向和请求转发的区别？

本质区别：

- 转发是服务器行为
- 重定向是客户端行为

重定向特点：两次请求，浏览器地址发生变化，可以访问自己 web 之外的资源，传输的数据会丢失。 **请求转发特点**：一次强求，浏览器地址不变，访问的是自己本身的 web 资源，传输的数据不会丢失。

题目 5：HTTP 和 HTTPS 的区别？

HTTPS = HTTP + SSL

- https 有 ca 证书，http 一般没有
- http 是超文本传输协议，信息是明文传输。https 则是具有安全性的 ssl 加密传输协议
- http 默认 80 端口，https 默认 443 端口

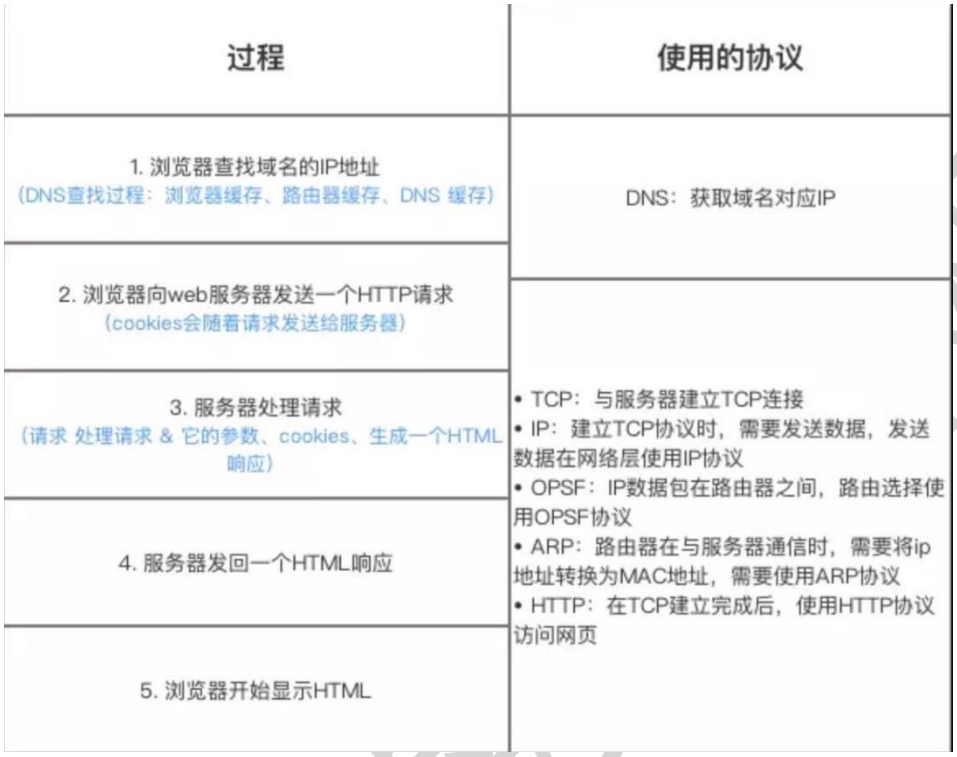
题目 6：HTTP 请求报文与响应报文格式？

请求报文： a、请求行：包含请求方法、URI、HTTP 版本信息 b、请求首部字段 c、请求内容实体

响应报文： a、状态行：包含 HTTP 版本、状态码、状态码的原因短语 b、响应首部字段 c、响应内容实体

题目 7：在浏览器中输入 url 地址到显示主页的过程？

图解（图片来源：《图解 HTTP》）：



总体来说分为以下几个过程:

1、域名解析

2、发起 TCP 的三次握手

3、建立 TCP 连接后发起 http 请求

4、服务器响应 http 请求，浏览器得到 HTML 代码

5、浏览器解析 HTML 代码，并请求 HTML 代码中的资源

6、浏览器对页面进行渲染呈现给用户

7、连接结束

题目 8： Cookie 和 Session 的区别？

Cookie： 是 web 服务器发送给浏览器的一块信息，浏览器会在本地一个文件中给每个 web 服务器存储 cookie。以后浏览器再给特定的 web 服务器发送请求时，同时会发送所有为该服务器存储的 cookie。 **Session：** 是存储在 web 服务器端的一块信息。session 对象存储特定用户会话所需的属性及配置信息。当用户在应用程序的 Web 页之间跳转时，存储在 Session 对象中的变量将不会丢失，而是在整个用户会话中一直存在下去。

区别： ①存在的位置

- cookie 存在于客户端，临时文件夹中；
- session 存在于服务器的内存中，一个 session 域对象为一个用户浏览器服务

②安全性

- cookie 是以明文的方式存放在客户端的，安全性低，可以通过一个加密算法进行加密后存放；
- session 存放于服务器的内存中，所以安全性好

③网络传输量

- cookie 会传递消息给服务器；
- session 本身存放于服务器，不会有传送流量

④生命周期(以 30 分钟为例)

- cookie 的生命周期是累计的，从创建时，就开始计时，30 分钟后，cookie 生命周期结束；
- session 的生命周期是间隔的，从创建时，开始计时如在 30 分钟，没有访问 session，那么 session 生命周期被销毁。但是，如果在 30 分钟内（如在第 29 分钟时）访问过 session，那么，将重新计算 session 的生命周期。关机会造成 session 生命周期的结束，但是对 cookie 没有影响。

⑤访问范围

- cookie 为多个用户浏览器共享；
- session 为一个用户浏览器独享

简单来说 cookie 机制采用的是在客户端保持状态的方案， 而 session 机制采用的是在服务器端保持状态的方案。由于才服务器端保持状态的方案在客户端也需要保存一个标识，所以 session 机制可能需要借助于 cookie 机制来达到保存标识的目的。

题目 9：Cookie 的过期和 Session 的超时有什么区别?

Cookie 的过期和 Session 的超时（过期），都是对某个对象设置一个时间，然后采用轮训机制（或者首次访问时）检查当前对象是否超时（当前对象会保存一个开始时间），如果超时则进行移除。 cookie 保存在浏览器中，不安全。而 session 是保存在服务端的。cookie 的生命周期很长，而 session 很短，一般也就几十分钟。

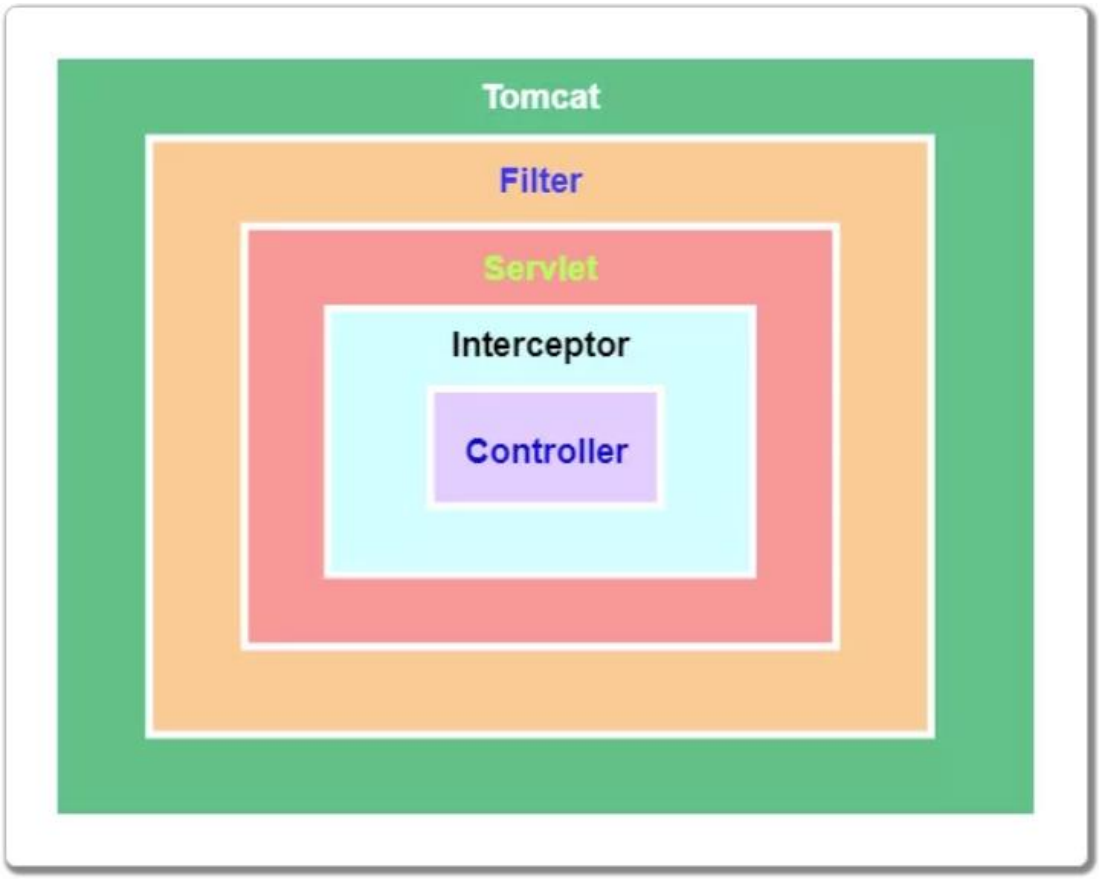
cookie 是保存在客户端，session 保存在服务器端，cookie 保存着 session 相关信息。如果 cookie 没有超时，那么浏览器每次请求都会带上该 cookie 信息，服务器端根据 cookie 信息从 session 缓存中获取相对应的 session。这两个信息有一个超时，用户连接即宣告关闭。

会话的超时由服务器来维护，它不同于 Cookie 的失效日期。 首先，会话一般基于驻留内存的 cookie，不是持续性的 cookie，因而也就没有截至日期。 即使截取到 JSESSIONID cookie，并为它设定一个失效日期发送出去。浏览器会话和服务器会话也会截然不同。

题目 10：如何解决分布式 Session 问题？

- Nginx ip_hash 策略 ，服务端使用 Nginx 代理，每个请求按访问 IP 的 hash 分配，这样来自同一 IP 固定访问一个后台服务器，避免了在服务器 A 创建 Session，第二次分发到服务器 B 的现象。
- Session 复制，任何一个服务器上的 Session 发生改变（增删改），该节点会把这个 Session 的所有内容序列化，然后广播给所有其它节点。
- 共享 Session，服务端无状态话，将用户的 Session 等信息使用缓存中间件来统一管理，保障分发到每一个服务器的响应结果都一致。

题目 11：过滤器和拦截器的区别？



- 原理实现上：过滤器基于回调实现，而拦截器基于动态代理。
- 控制粒度上：过滤器和拦截器都能够实现对请求的拦截功能，但是在拦截的粒度上有较大的差异，拦截器对访问控制的粒度更细。
- 使用场景上：拦截器往往用于权限检查、日志记录等，过滤器主要用于过滤请求中无效参数，安全校验。
- 依赖容器上：过滤器依赖于 Servlet 容器，局限于 web，而拦截器依赖于 Spring 框架，能够使用 Spring 框架的资源，不仅限于 web。
- 触发时机上：过滤器在 Servlet 前后执行，拦截器在 handler 前后执行，现在大多数 web 应用基于 Spring，拦截器更细。

Java 热门面试题-MySQL

题目 1：MySQL 存储引擎对比

特性比较	事务	并发	外键	备份	崩溃恢复	其他
InnoDB	事务型	行级锁	支持	在线热备	概率低	聚簇索引，MVCC特性
MyISAM	非事务型	表级锁	不支持	不支持	慢，易丢失	压缩表，空间数据索引

MySQL 默认是 Innodb 存储引擎，适合比较庞大的应用场景

题目 2：索引的类型？

- **从数据结构角度**

1. 树索引（O(log(n))）
2. Hash 索引

- **从物理存储角度**

1. 聚集索引（clustered index）
2. 非聚集索引（non-clustered index）

- **从逻辑角度**

1. 普通索引
2. 唯一索引
3. 主键索引
4. 联合索引

题目 3：为什么 InnoDB 存储引擎选用 B+ 树而不是 B 树呢？

- B+ 树是基于 B 树和叶子节点顺序访问指针进行实现，它具有 B 树的平衡性，并且通过顺序访问指针来提高区间查询的性能。
- 在 B+ 树中，一个节点中的 key 从左到右非递减排列，如果某个指针的左右相邻 key 分别是 key i 和 key i+1，且不为 null，则该指针指向节点的所有 key 大于等于 key i 且小于等于 key i+1。
- 进行查找操作时，首先在根节点进行二分查找，找到一个 key 所在的指针，然后递归地在指针所指向的节点进行查找。直到查找到叶子节点，然后在叶子节点上进行二分查找，找出 key 所对应的 data。
- 插入、删除操作会破坏平衡树的平衡性，因此在插入删除操作之后，需要对树进行一个分裂、合并、旋转等操作来维护平衡性。

用 B+ 树不用 B 树考虑的是 IO 对性能的影响，B 树的每个节点都存储数据，而 B+ 树只有叶子节点才存储数据，所以查找相同数据量的情况下，B 树的高度更高，IO 更频繁。数据库索引是存储在磁盘上的，当数据量大时，就不能把整个索引全部加载到内存了，只能逐一加载每一个磁盘页（对应索引树的节点）。

题目 4：什么情况索引会失效？

下面列举几种不走索引的 SQL 语句：

- 索引列参与表达式计算：

```
SELECT 'sname' FROM 'stu' WHERE 'age' + 10 = 30;
```

- 函数运算：

```
SELECT 'sname' FROM 'stu' WHERE LEFT('date',4) < 1990;
```

- %词语%–模糊查询：

```
SELECT * FROM 'manong' WHERE `uname` LIKE '码农%' -- 走索引

SELECT * FROM 'manong' WHERE `uname` LIKE '%码农%' -- 不走索引
```

- 字符串与数字比较不走索引：

```
CREATE TABLE 'a' ('a' char(10));
EXPLAIN SELECT * FROM 'a' WHERE 'a'="1" — 走索引
EXPLAIN SELECT * FROM 'a'WHERE 'a'=1 — 不走索引，同样也是使用了函数运算
```

- 查询条件中有 or ，即使其中有条件带索引也不会使用。换言之，就是要求使用的所有字段，都必须建立索引：

```
select * from dept where dname='xxx' or loc='xx' or deptno = 45;
```

- 正则表达式不使用索引。
- MySQL 内部优化器会对 SQL 语句进行优化，如果优化器估计使用全表扫描要比使用索引快，则不使用索引。

题目 5：说一下 MySQL 的行锁和表锁？

MyISAM 只支持表锁，InnoDB 支持表锁和行锁，默认为行锁。

- 表级锁：开销小，加锁快，不会出现死锁。锁定粒度大，发生锁冲突的概率最高，并发量最低。
- 行级锁：开销大，加锁慢，会出现死锁。锁力度小，发生锁冲突的概率小，并发度最高。

题目 6：MySQL 问题排查都有哪些手段？

- 使用 show processlist 命令查看当前所有连接信息；
- 使用 Explain 命令查询 SQL 语句执行计划；
- 开启慢查询日志，查看慢查询的 SQL。

题目 7：MySQL 性能优化？

- 服务器优化（增加 CPU、内存、网络、更换高性能磁盘）
- 表设计优化（字段长度控制、添加必要的索引）
- SQL 优化（避免 SQL 命中不到索引的情况）
- 架构部署优化（一主多从集群部署）
- 编码优化实现读写分离

题目 8：优化 sql 语句？

SQL 优化是一个分析，优化，再分析，再优化的过程。站在执行计划的角度来说，我们这个过程，就是在不断的减少 rows 的数量。

1 查看表的执行频次,show 【session|global】 status;

如果是查询多,可以优化查询,如何增加和更新多可以优化更新.

2 查看 explain 执行计划来优化

基础的查询优化:

最大化利用索引；

尽可能避免全表扫描；

减少无效数据的查询；

1 使用联合索引要符合最左前缀法则.

2 范围查询 不要出现> <,如果非要使用可以改成>=,<=对于插入操作:要批量插入,主键顺序插入,大批量插入数据还可以使用 load 指令

对于更新操作:更新的条件一定要有索引,要不然行锁会升级成表锁

题目 9： SQL 内连接外连接有什么差别

内连接:内连接 li(inner join)就是 join)利用 where 子句对多表连接形成的笛卡尔积进行筛选。说白了内连接就是获取两个表之间的公共部分内容。

左外连接 left join:如果要获取左边表中的全部内容，就使用左连接

右连接 right join:如果要获取右边表的全部内容，就使用右连接。

题目 10： Msql 去重关键字？

1.作用于单列

select distinct name from A

2.作用于多列

select distinct name, id from A

3.COUNT 统计

select count(distinct name) from A;

题目 11： mysql 搜索引擎?

InnoDB：MySql 5.6 版本默认的存储引擎。InnoDB 是一个事务安全的存储引擎，它具备提交、回滚以及崩溃恢复的功能以保护用户数据。InnoDB 的行级别锁定以及 Oracle 风格的一致性无锁读提升了它的多用户并发数以及性能。InnoDB 将用户数据存储在聚集索引中以减少基于主键的普通查询所带来的 I/O 开销。为了保证数据的完整性，InnoDB 还支持外键约束。

MyISAM：MyISAM 既不支持事务、也不支持外键、其优势是访问速度快，但是表级别的锁定限制了它在读写负载方面的性能，因此它经常应用于只读或者以读为主的数据场景。

Memory：在内存中存储所有数据，应用于对非关键数据由快速查找的场景。Memory 类型的表访问数据非常快，因为它的数据是存放在内存中的，并且默认使用 HASH 索引，但是一旦服务关闭，表中的数据就会丢失

题目 12： 百万数据怎么快速查询出一条数据?

https://zhuanlan.zhihu.com/p/92552787

方法 1: 直接使用数据库提供的 SQL 语句

MySQL 中,可用如下方法: SELECT * FROM 表名称 LIMIT M,N

方法 2: 建立主键或唯一索引, 利用索引(假设每页 10 条)

方法 3: 基于索引再排序

以上方法需要 limit 分页，mysql 大数据量使用 limit 分页，随着页码的增大，查询效率越低下。后续需要对 limit 分页问题的性能优化。

优化方案：如果对于有 where 条件，又想走索引用 limit 的，必须设计一个索引，将 where 放第一位，limit 用到的主键放第 2 位，而且只能 select 主键！

完美解决了分页问题了。可以快速返回 id 就有希望优化 limit ， 按这样的逻辑，百万级的 limit 应该在 0.0x 秒就可以分完。看来 mysql 语句的优化和索引时非常重要的！

其他方案：分表

思路：大表拆成小表，接口引导小表。

按数据库 id 分表多个小表，每个小表设置索引。查询接口根据 id 指向相应的小表。如果数据再多一些，考虑数据库分库。

Java 热门面试题-SSM 框架

题目 1：谈谈你对 Spring 的理解？

Spring 是一个 IOC 和 AOP 容器框架。 Spring 容器的主要核心是：

- 控制反转（IOC），传统的 java 开发模式中，当需要一个对象时，我们会自己使用 new 或者 getInstance 等直接或者间接调用构造方法创建一个对象。而在 spring 开发模式中，spring 容器使用了工厂模式为我们创建了所需要的对象，不需要我们自己创建了，直接调用 spring 提供的对象就可以了，这是控制反转的思想。
- 依赖注入（DI），spring 使用 javaBean 对象的 set 方法或者带参数的构造方法为我们在创建所需对象时将其属性自动设置所需要的值的过程，就是依赖注入的思想。
- 面向切面编程（AOP），在面向对象编程（oop）思想中，我们将事物纵向抽成一个个的对象。而在面向切面编程中，我们将一个个的对象某些类似的方面横向抽成一个切面，对这个切面进行一些如权限控制、事物管理，记录日志等。公用操作处理的过程就是面向切面编程的思想。AOP 底层是动态代理，如果是接口采用 JDK 动态代理，如果是类采用 CGLIB 方式实现动态代理。

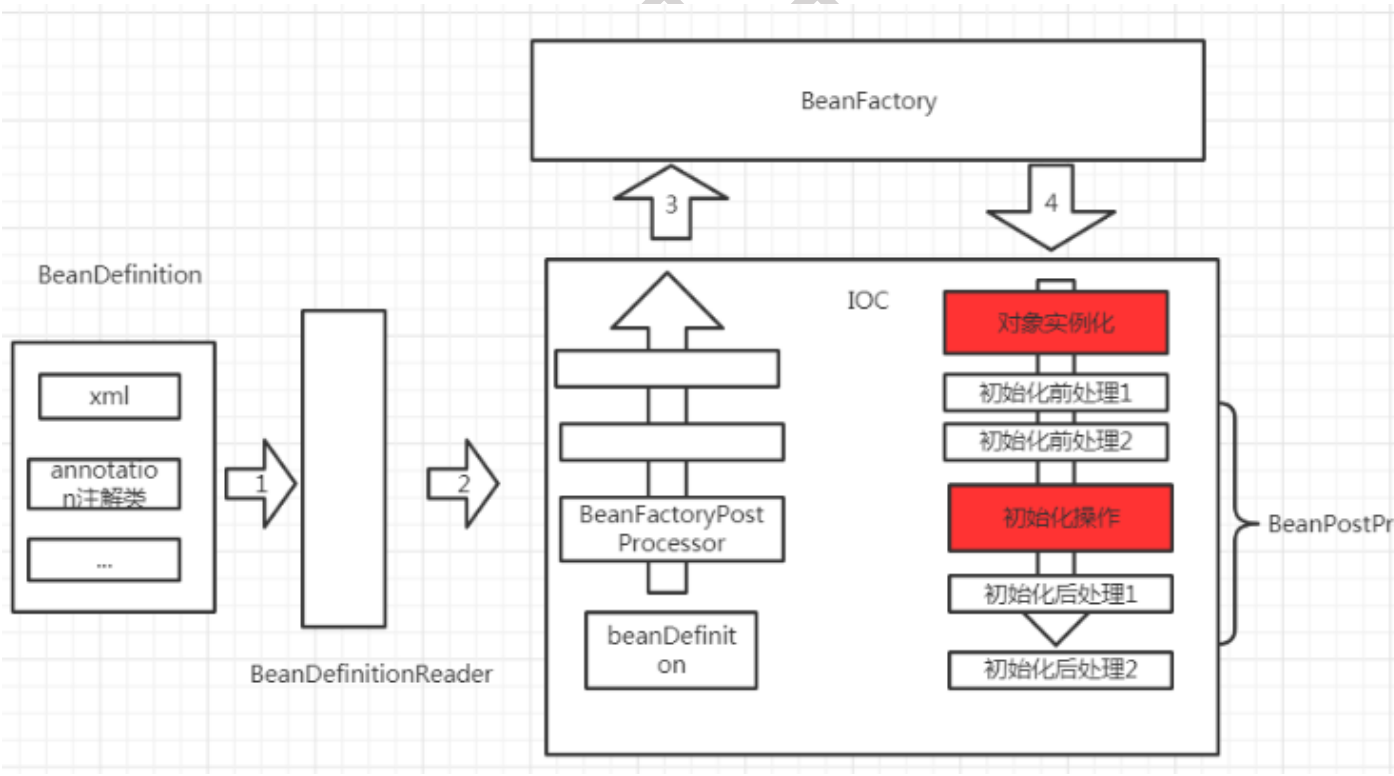
题目 2：IOC 是什么？

- IOC（Inversion Of Controll，控制反转）是一种设计思想，就是将原本在程序中手动创建对象的控制权，交由给 Spring 框架来管理。IOC 在其他语言中也有应

用，并非 Spring 特有。IOC 容器是 Spring 用来实现 IOC 的载体，IOC 容器实际上就是一个 Map(key, value)，Map 中存放 的是各种对象。

将对象之间的相互依赖关系交给 IOC 容器来管理，并由 IOC 容器完成对象的注入。这样可以很 大程度上简化应用的开发，把应用从复杂的依赖关系中解放出来。IOC 容器就像是一个工厂一 样，当我们需要创建一个对象的时候，只需要配置好配置文件/注解即可，完全不用考虑对象是 如何被创建出来的。在实际项目中一个 Service 类可能由几百甚至上千个类作为它的底层，假 如我们需要实例化这个 Service，可能要每次都搞清楚这个 Service 所有底层类的构造函数，这 可能会把人逼疯。如果利用 IOC 的话，你只需要配置好，然后在需要的地方引用就行了，大大增加了项目的可维护性且降低了开发难度

题目 3：IOC 原理？



解析流程： SpringBean new A()

根据路径、资源名称等方式，将 xml 文件、注解类加载到容器中

通过 BeanDefinitionReader 将对象解析成 BeanDefinition 实例

创建 BeanFactory 工厂(注册前后需要添加 bean 前置、后置处理器)

通过 BeanFactory 工厂将对象实例化、对象初始化(初始化前后执行前置、后置处理器)

总结以上步骤，核心主干主要就是五部分构成：

1. 构造 Bean 对象
2. 设置 Bean 属性
3. 初始化回调
4. Bean 调用
5. 销毁 Bean

题目 4：IOC 的使用场景?

正常情况下我们使用一个对象时都是需要 new Object() 的。而 ioc 是把需要使用的对象提前创建好，放到 spring 的容器里面。需要使用的时候直接使用就行，而且可以设置单例或多例，非常灵活。

我们在 service 层想调用另外一个 service 的方法，不需要去 new 了，直接把它交给 spring 管理，然后用注解的方式引入就能使用。

- 通过 XML 配置实例化 Bean，将 bean 的信息配置.xml 文件里，通过 Spring 加载文件为我们创建 bean 和配置 bean 属性
- 通过注解声明配置 通过在类上加注解的方式，来声明一个类交给 Spring 管理，Spring 会自动扫描带有@Component，@Controller，@Service，@Repository 这四个注解的类，然后帮我们创建并管理，前提是需要先配置 Spring 的注解扫描器。
  - –@Component：可以用于注册所有 bean
  - –@Repository：主要用于注册 dao 层的 bean
  - –@Controller：主要用于注册控制层的 bean
  - –@Service：主要用于注册服务层的 bean
- 通过配置类 将类的创建交给我们配置的 JavcConfig 类来完成，Spring 只负责维护和管理创建一个配置类
  - –添加@Configuration 注解声明为配置类
  - –方法上加上@bean，该方法用于创建实例并返回

题目 5：IOC 中 Bean 有几种注入方式？

- 构造器依赖注入：构造器依赖注入通过容器触发一个类的构造器来实现的，该类有一系列参数，每个参数代表一个对其他类的依赖。
- Setter 方法注入：Setter 方法注入是容器通过调用无参构造器或无参 static 工厂方法实例化 bean 之后，调用该 bean 的 Setter 方法，即实现了基于 Setter 的依赖注入。
- 基于注解的注入：最好的解决方案是用构造器参数实现强制依赖，Setter 方法实现可选依赖。

题目 6：SpringBean 自动装配的几种方式？

自动装配提供五种不同的模式供 Spring 容器用来自动装配 beans 之间的依赖注入:

- byName：通过参数名自动装配，Spring 容器查找 beans 的属性，这些 beans 在 XML 配置文件 中被设置为 byName。之后容器试图匹配、装配和该 bean 的属性具有相同名字的 bean。
- byType：通过参数的数据类型自动自动装配，Spring 容器查找 beans 的属性，这些 beans 在 XML 配置文件中被设置为 byType。之后容器试图匹配和装配和该 bean 的属性类型一样的 bean。如果有多个 bean 符合条件，则抛出错误。
- constructor：这个同 byType 类似，不过是应用于构造函数的参数。如果在 BeanFactory 中不 是恰好有一个 bean 与构造函数参数相同类型，则抛出一个严重的错误。
- autodetect：如果有默认的构造方法，通过 construct 的方式自动装配，否则使用 byType 的 方式自动装配。

题目 7：AOP 是什么？

- AOP：全称 Aspect Oriented Programming，即：**面向切面编程**。
- AOP（Aspect-Oriented Programming，面向切面编程）能够将那些与业务无关，却为业务 模块所共同调用的逻辑或责任（例如事务处理、日志管理、权限控制等）封装起来，便于减少 系统的重复代码，降低模块间的耦合度，并有利于未来的可扩展性和可维护性。
- Spring AOP 是基于动态代理的，如果要代理的对象实现了某个接口，那么 Spring AOP 就会使 用 JDK 动态代理去创建代理对象；而对于没有实现接口的对象，就

无法使用 JDK 动态代理，转 而使用 CGlib 动态代理生成一个被代理对象的子类来作为代理。 当然也可以使用 AspectJ，Spring AOP 中已经集成了 AspectJ，AspectJ 应该算得上是 Java 生态系统中最完整的 AOP 框架了。使用 AOP 之后我们可以把一些通用功能抽象出来，在需要用到的地方直接使用即可，这样可以大大简化代码量。我们需要增加新功能也方便，提高了系统的 扩展性。日志功能、事务管理和权限管理等场景都用到了 AOP。

题目 8：AOP 主要用在哪些场景中？

- 事务管理 比如项目中使用的事务注解@Transactional
- 日志管理 创建日志切面
- 用于全局异常处理拦截
- 性能统计 将与业务无关的代码，使用 AOP 拦截他们。
- 缓存使用 @Cacheable 和@CacheEvict
- 安全检查
  在支付行业往往对安全性要求比较高，我们在保存 / 接收 / 发送数据前先要对数据进行验签 / 签名 / 加密等操作，都需要做特殊处理。
  比如一个手机号，我们可以通过一个 " 拦截器 " 对手机号，身份证号这种敏感信息做这种特殊处理；

题目 9：SpringBean 的作用域有几种?

Spring 框架支持以下五种 **bean** 的作用域：

- singleton：唯一 bean 实例，Spring 中的 bean 默认都是单例的。
- prototype：每次请求都会创建一个新的 bean 实例。
- request：每一次 HTTP 请求都会产生一个新的 bean，该 bean 仅在当前 HTTP request 内有 效。
- session：每一次 HTTP 请求都会产生一个新的 bean，该 bean 仅在当前 HTTP session 内有 效。
- global-session：全局 session 作用域，仅仅在基于 Portlet 的 Web 应用中才有意义，Spring5 中已经没有了 Portlet。。

题目 10：Spring 事务管理方式？

- 编程式事务：在代码中硬编码。

- 声明式事务：在配置文件中配置 声明式事务又分为：
  –基于 XML 的声明式事务
  –基于注解的声明式事务

题目 11：Spring 事务传播行为有几种？

事务传播行为是为了解决业务层方法之间互相调用的事务问题。当事务方法被另一个事务方法调用时，必须指定事务应该如何传播。例如：方法可能继续在现有事务中运行，也可能开启一个新事务，并在自己的事务中运行。在 TransactionDefinition 定义中包括了如下几个表示传播行为的常量：

- **支持当前事务的情况：**

TransactionDefinition.PROPAGATION_REQUIRED(默认)：如果当前存在事务，则加入该事务；如果当前没有事务，则创建一个新的事务；

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

TransactionDefinition.PROPAGATION_MANDATORY：如果当前存在事务，则加入该事务；如果当前没有事务，则抛出异常。

- **不支持当前事务的情况：**

TransactionDefinition.PROPAGATION_REQUIRES_NEW：创建一个新的事务，如果当前存在事务，则把当前事务挂起；

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

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

- **其他情况：**

TransactionDefinition.PROPAGATION_NESTED：如果当前存在事务，则创建一个事务作为当前事务的嵌套事务来运行；如果当前没有事务，则该取值等价于 TransactionDefinition.PROPAGATION_REQUIRED。

题目 12：Spring 中的事务隔离级别？

TransactionDefinition 接口中定义了五个表示隔离级别的常量：

- TransactionDefinition.ISOLATION_DEFAULT：使用后端数据库默认的隔离级别，MySQL 默认采用的 REPEATABLE_READ（可重复读） 隔离级别，Oracle 默认采用的 READ_COMMITTED（读已提交） 隔离级别；
- TransactionDefinition.ISOLATION_READ_UNCOMMITTED：最低的隔离级别，允许读取尚未提交的数据变更，可能会导致脏读、幻读或不可重复读；
- TransactionDefinition.ISOLATION_READ_COMMITTED：允许读取并发事务已经提交的数据，可以阻止脏读，但是幻读或不可重复读仍有可能发生；
- TransactionDefinition.ISOLATION_REPEATABLE_READ：对同一字段的多次读取结果都是一致的，除非数据是被本身事务自己所修改，可以阻止脏读和不可重复读，但幻读仍有可能发生；
- TransactionDefinition.ISOLATION_SERIALIZABLE：最高的隔离级别，完全服从 ACID 的隔离级别。所有的事务依次逐个执行，这样事务之间就完全不可能产生干扰，也就是说，该级别可以防止脏读、不可重复读以及幻读。但是这将严重影响程序的性能。通常情况下也不会用到该级别。

题目 13：Spring 中的设计模式有哪些？

- **工厂模式：** Spring 使用工厂模式通过 BeanFactory 和 ApplicationContext 创建 bean 对 象。
- **单例模式：** Spring 中的 bean 默认都是单例的。
- **代理模式：** Spring 的 AOP 功能用到了 JDK 的动态代理和 CGLIB 字节码生成技术；
- **模板方法：** 用来解决代码重复的问题。比如 RestTemplate、jdbcTemplate、JpaTemplate 等以 Template 结尾的对 数据库操作的类，它们就使用到了模板模式。
- **观察者模式：** Spring 事件驱动模型就是观察者模式很经典的一个应用。定义对象键一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都会得到通知被制动更新，如 Spring 中 listener 的实现 ApplicationListener。

- **包装器设计模式：**

  我们的项目需要连接多个数据库，而且不同的客户在每次访问中根据需要 会去访问不同的数据库。这种模式让我们可以根据客户的需求能够动态切换不同的数据源。

- **适配器模式：**

  Spring AOP 的增强或通知（Advice）使用到了适配器模式、Spring MVC 中也是用到了适配器模式适配 Controller。

题目 14：Spring 的常用注解有哪些？

- @Autowired：用于有值设值方法、非设值方法、构造方法和变量。
- @Component：用于注册所有 bean
- @Repository：用于注册 dao 层的 bean
- @Controller：用于注册控制层的 bean
- @Service：用于注册服务层的 bean
- @Component: 用于实例化对象
- @Value: 简单属性的依赖注入
- @ComponentScan: 组件扫描
- @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
- @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

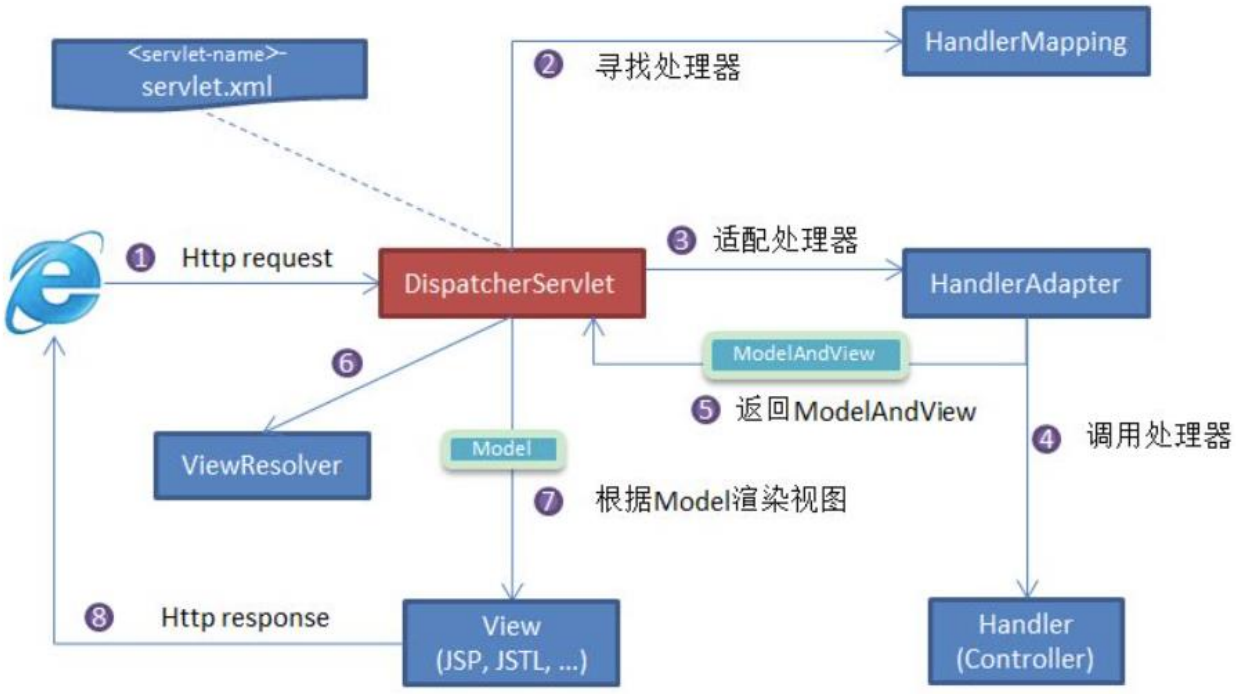
题目 15：@Resources 和@Autowired 的区别?

- 都是用来自动装配的，都可以放在属性的字段上
- @Autowired 通过 byType 的方式实现，而且必须要求这个对象存在！
- @Resource 默认通过 byName 的方式实现，如果找不到名字，则通过 byType 实现！如果两个都找不到的情况下，就报错！

题目 16：SpringMVC 执行流程（工作原理）？

MVC 是 Model — View — Controler 的简称，它是一种架构模式，它分离了表现与交互。它被分为三个核心部件：模型、视图、控制器。

- Model（模型）：是程序的主体部分，主要包含业务数据和业务逻辑。在模型层，还会涉及到用户发布的服务，在服务中会根据不同的业务需求，更新业务模型中的数据。
- View（视图）：是程序呈现给用户的部分，是用户和程序交互的接口，用户会根据具体的业务需求，在 View 视图层输入自己特定的业务数据，并通过界面的事件交互，将对应的输入参数提交给后台控制器进行处理。
- Controller（控制器）：Controller 是用来处理用户输入数据，以及更新业务模型的部分。控制器中接收了用户与界面交互时传递过来的数据，并根据数据业务逻辑来执行服务的调用和更新业务模型的数据和状态。



工作原理：

- 1.客户端（浏览器）发送请求，直接请求到 DispatcherServlet。

- 2.DispatcherServlet 根据请求信息调用 HandlerMapping，解析请求对应的 Handler。
- 3.解析到对应的 Handler（也就是我们平常说的 Controller 控制器）。
- 4.HandlerAdapter 会根据 Handler 来调用真正的处理器来处理请求和执行相对应的业务逻 辑。
- 5.处理器处理完业务后，会返回一个 ModelAndView 对象，Model 是返回的数据对象，View 是逻辑上的 View。
- 6.ViewResolver 会根据逻辑 View 去查找实际的 View。
- 7.DispatcherServlet 把返回的 Model 传给 View（视图渲染）。
- 8.把 View 返回给请求者（浏览器）。

题目 17：SpringMVC 的常用注解有哪些？

- @RequestMapping：用于处理请求 url 映射的注解，可用于类或方法上。用于类上，则表示类中的所有响应请求的方法都是以该地址作为父路径；
- @RequestBody：注解实现接收 HTTP 请求的 json 数据，将 json 转换为 Java 对象；
- @ResponseBody：注解实现将 Controller 方法返回对象转化为 json 对象响应给客户。

题目 18：SpringMVC 如何处理统一异常？

- 方式一：创建一个自定义异常处理器(实现 HandlerExceptionResolver 接口),并实现里面 的异常处理方法,然后将这个类交给 Spring 容器管理
- 方式二：在类上加注解(@ControllerAdvice)表明这是一个全局异常处理类在方法上加注解 (@ExceptionHandler), 在 ExceptionHandler 中 有 一 个 value 属性,可以指定可以处理的异常类型

题目 19：MyBatis 中#{}和${}的区别是什么？

#{}是预编译处理，${}是字符串替换。Mybatis 在处理#{}时，会将 sql 中的#{}替换为?号，调用 PreparedStatement 的 set 方法来赋值；Mybatis 在处理${}时，就是把{}替换成变量的值。

题目 20：MyBatis 是如何进行分页的？分页插件的原理是什么？

分页插件的基本原理是使用 Mybatis 提供的插件接口，实现自定义插件，在插件的拦截方法内拦截待执行的 sql，然后重写 sql，根据 dialect 方言，添加对应的物理分页语句和物理分页参数。

题目 21：MyBatis 动态 SQL 传参，注解是什么？

Xml 方式：

1. MyBatis 动态 SQL 可以让我们在 XML 映射文件内，以标签的形式编写动态 SQL，完成逻辑判断和动态拼接 SQL 的功能；

2. MyBatis 提供了 9 种动态 SQL 标签：trim、where、set、foreach、if、choose、when、otherwise、bind；

注解方式：

mybatis 注解方式的最大特点就是取消了 Mapper 的 XML 配置，具体的 SQL 脚本直接写在 Mapper 类或是 SQLProvider 中的方法动态生成 。
mybatis 提供的常用注解有： @Insert 、@Update 、@Select、 @Delete 等标签，这些注解其实就是 MyBatis 提供的来取代其 XML 配置文件的

@Results 和 @Result 注解
@Results 和 @Result 主要作用是，当有一些特殊的场景需要处理，查询的返回结果与期望的数据格式不一致时，可以将将数据库中查询到的数值自动转化为具体的属性或类型，，修饰返回的结果集。比如查询的对象返回值属性名和字段名不一致，或

者对象的属性中使用了枚举等。如果实体类属性和数据库属性名保持一致，就不需要这个属性来修饰。

```
@Select({
    "select",
    "id, company_id, username, password, nickname, age, sex, job, face_image, province, "
    "city, district, address, auth_salt, last_login_ip, last_login_time, is_delete, ",
    "regist_time",
    "from sys_user",
    "where id = #{id,jdbcType=VARCHAR}"
})
@Results({
    @Result(column="id", property="id", jdbcType=JdbcType.VARCHAR, id=true),
    @Result(column="company_id", property="companyId", jdbcType=JdbcType.VARCHAR),
    @Result(column="face_image", property="faceImage", jdbcType=JdbcType.VARCHAR),
    @Result(column="auth_salt", property="authSalt", jdbcType=JdbcType.VARCHAR),
    @Result(column="last_login_ip", property="lastLoginIp", jdbcType=JdbcType.VARCHAR),
    @Result(column="last_login_time", property="lastLoginTime", jdbcType=JdbcType.TIMESTA
    @Result(column="is_delete", property="isDelete", jdbcType=JdbcType.INTEGER),
    @Result(column="regist_time", property="registTime", jdbcType=JdbcType.TIMESTAMP)
})
User selectByPrimaryKey(String id);
```

传参方式：

1、直接传参

```
@Delete("delete from sys_user where id = #{id,jdbcType=VARCHAR}")
int deleteByPrimaryKey(String id);
```

2、使用 @Param 注解

```
@Select("SELECT * FROM sys_user WHERE username = #{username} and password = #{password}")
List<User> getListByUserSex(@Param("username") String userName, @Param("password") String pa
```

3、Map 传值

```
@Select("SELECT * FROM sys_user WHERE username=#{username} AND password = #{password}")
List<User> getListByNameAndSex(Map<String, Object> map);
```

调用时将参数依次加入到 Map 中即可。

```
Map param= new HashMap();
param.put("username","admin");
param.put("password","123456");
List<User> users = userMapper.getListByNameAndSex(param)
```

4、使用 pojo 对象

```
@Update({
    "update sys_user",
    "set company_id = #{companyId,jdbcType=VARCHAR},",
      "username = #{username,jdbcType=VARCHAR},",
      "password = #{password,jdbcType=VARCHAR},",
      "nickname = #{nickname,jdbcType=VARCHAR},",
      "age = #{age,jdbcType=INTEGER},",
      "sex = #{sex,jdbcType=INTEGER},",
      "job = #{job,jdbcType=INTEGER},",
      "face_image = #{faceImage,jdbcType=VARCHAR},",
      "province = #{province,jdbcType=VARCHAR},",
      "city = #{city,jdbcType=VARCHAR},",
      "district = #{district,jdbcType=VARCHAR},",
      "address = #{address,jdbcType=VARCHAR},",
      "auth_salt = #{authSalt,jdbcType=VARCHAR},",
      "last_login_ip = #{lastLoginIp,jdbcType=VARCHAR},",
      "last_login_time = #{lastLoginTime,jdbcType=TIMESTAMP},"
      "is_delete = #{isDelete,jdbcType=INTEGER},",
      "regist_time = #{registTime,jdbcType=TIMESTAMP}",
    "where id = #{id,jdbcType=VARCHAR}"
})
int updateByPrimaryKey(User record);
```

题目 22：实体类属性与表字段不一致解决方法？

Mybatis+xml:

Mybatis 提供了一个自动驼峰命名规则的设置，但是默认是关闭的，所以当我们没有设置的时候，这样也是对应不上的。我们就需要在 Mybatis 的配置文件中添加如下配置

方法 1:通过 sql 语句的字段起别名,别名和实体中的对象属性一致

方法 2: mybatis 最强大的地方 ：reslutMap 对象

Mybatis Plus：

@TableName（value = …）

当数据库名与实体类名不一致或不符合驼峰命名时，需要在此注解指定表名

@TableId（type = …）

1、 主要用来解决实体类的字段名与数据库中的字段名不匹配的问题（数据库 user_addr，字段 useraddr 未驼峰）

2、 实体类中的属性字段在表中不存在的问题

```
// 用来解决数据库中的字段和实体类的字段不匹配问题
@TableField(value = "age")

// 用来解决实体类中有的属性但是数据表中没有的字段
@TableField(exist = false)  // 默认为true
```

如不指定，出现以下的异常：BadSqlGrammarException

```
// 查询时不返回该字段，默认true，和@TableId作用在同一个字段上时失效
@TableField(select = false)
```

题目 23：ResultType 和 ResultMap 的区别？

- 如果数据库结果集中的列名和要封装实体的属性名完全一致的话用 resultType 属性
- 如果数据库结果集中的列名和要封装实体的属性名有不一致的情况用 resultMap 属性，通过 resultMap 手动建立对象关系映射，resultMap 要配置一下表和类的一一对应关系，所以说就算你的字段名和你的实体类的属性名不一样也没关系，都会给你映射出来

题目 24：MyBatis 中有哪些设计模式？

Mybatis 至少遇到了以下的设计模式的使用：

- **Builder 模式**，例如 SqlSessionFactoryBuilder、XMLConfigBuilder、XMLMapperBuilder、XMLStatementBuilder、CacheBuilder；
- **工厂模式**，例如 SqlSessionFactory、ObjectFactory、MapperProxyFactory；
- **单例模式**，例如 ErrorContext 和 LogFactory；
- **代理模式**，Mybatis 实现的核心，比如 MapperProxy、ConnectionLogger，用的 jdk 的动态代理；还有 executor.loader 包使用了 cglib 或者 javassist 达到延迟加载的效果；
- **组合模式**，例如 SqlNode 和各个子类 ChooseSqlNode 等；
- **模板方法模式**，例如 BaseExecutor 和 SimpleExecutor，还有 BaseTypeHandler 和所有的子类例如 IntegerTypeHandler；
- **适配器模式**，例如 Log 的 Mybatis 接口和它对 jdbc、log4j 等各种日志框架的适配实现；
- **装饰者模式**，例如 Cache 包中的 cache.decorators 子包中等各个装饰者的实现；
- **迭代器模式**，例如迭代器模式 PropertyTokenizer；

Java 热门面试题-SpringBoot

题目 1：为什么用 SpringBoot（优点）？

- 简单回答：

  Spring Boot 是 Spring 开源组织下的子项目，是 Spring 组件一站式解决方案，主要是简化了使用 Spring 的难度，简省了繁重的配置，提供了各种启动器，开发者能快速上手。

- 详细回答：

  –独立运行 Spring Boot 而且内嵌了各种 servlet 容器，Tomcat、Jetty 等，现在不再需要打成 war 包部署到容器中，Spring Boot 只要打成一个可执行的 jar 包就能独立运行，所有的依赖包都在一个 jar 包内。

  –简化配置 spring-boot-starter-web 启动器自动依赖其他组件，简少了 maven 的配置。

–自动配置 Spring Boot 能根据当前类路径下的类、jar 包来自动配置 bean，如添加一个 spring-boot-starter-web 启动器就能拥有 web 的功能，无需其他配置。

–无代码生成和 XML 配置 Spring Boot 配置过程中无代码生成，也无需 XML 配置文件就能完成所有配置工作，这一切都是借助于条件注解完成的，这也是 Spring4.x 的核心功能之一。

–避免大量的 Maven 导入和各种版本冲突

SpringBoot 优点概括起来就是简化：简化编码，简化配置，简化部署，简化监控，简化依赖坐标导入，简化整合其他技术

题目 2：SpringBoot、Spring MVC 和 Spring 有什么区别？

- **Spring** Spring 最重要的特征是依赖注入。所有 Spring Modules 不是依赖注入就是 IOC 控制反转。 当我们恰当的使用 DI 或者是 IOC 的时候，可以开发松耦合应用。
- **Spring MVC** Spring MVC 提供了一种分离式的方法来开发 Web 应用。通过运用像 DispatcherServlet，ModelAndView 和 ViewResolver 等一些简单的概念，开发 Web 应用将会变的非常简单。
- **SpringBoot** Spring 和 Spring MVC 的问题在于需要配置大量的参数。 SpringBoot 通过一个自动配置和启动的项来解决这个问题。

题目 3：SpringBoot 启动时都做了什么？

1. SpringBoot 在启动的时候从类路径下的 META-INF/spring.factories 中获取 EnableAutoConfiguration 指定的值
2. 将这些值作为自动配置类导入容器 ， 自动配置类就生效 ， 帮我们进行自动配置工作；
3. 整个 J2EE 的整体解决方案和自动配置都在 springboot-autoconfigure 的 jar 包中；
4. 它会给容器中导入非常多的自动配置类 （xxxAutoConfiguration），就是给容器中导入这个场景需要的所有组件 ， 并配置好这些组件 ；
5. 有了自动配置类 ， 免去了我们手动编写配置注入功能组件等的工作；

题目 4：SpringFactories 机制？

Spring Boot 的自动配置是基于 Spring Factories 机制实现的。Spring Factories 机制是 Spring Boot 中的一种服务发现机制，这种扩展机制与 Java SPI 机制十分相似。Spring Boot 会自动扫描所有 Jar 包类路径下 META-INF/spring.factories 文件，并读取其中的内容，进行实例化，这种机制也是 Spring Boot Starter 的基础。

题目 5：SpringBoot 自动配置原理？

注解 @EnableAutoConfiguration, @Configuration, @ConditionalOnClass 就是自动配置的核心，

@EnableAutoConfiguration 给容器导入 META-INF/spring.factories 里定义的自动配置类。

每一个自动配置类结合对应的 xxxProperties.java 读取配置文件进行自动配置功能。

题目 6：运行 SpringBoot 有哪几种方式?

- 打包用命令或者放到容器中运行
- 用 Maven/ Gradle 插件运行
- 直接执行 main 方法运行

题目 7：SpringBoot 的核心注解是哪个？由哪些注解组成？

`@SpringBootApplication`

SpringBootApplication，由 3 个注解组成：

- @SpringBootConfiguration：组合了 @Configuration 注解，实现配置文件的功能。
- @EnableAutoConfiguration：打开自动配置的功能，也可以关闭某个自动配置的选项，如关闭数据源自动配置功能。
- @ComponentScan：Spring 组件扫描。

题目 8：Spring Boot 中的 starter 到底是什么 ？

- 依赖管理对于项目至关重要。当项目足够复杂时，管理依赖项可能会变成一场噩梦，因为涉及的组件太多了。这就是 Spring Boot 的 starter 就派上用场了。每个 starter 都可以为我们提供所需要的 Spring 技术的一站式服务。并且以一致的方式传递和管理其他所需的依赖关系。 所有官方 starter 都在 org.springframework.boot 组下，其名称以 spring-boot-starter- 开头 。非官方的 starter 的名称在前，如 mybatis-spring-boot-starter。这种命名模式使得查找启动器变得很容易，尤其是在使用支持按名称搜索依赖关系的 IDE 时。但是这个不是绝对的，有些开发者可能不遵从这种契约。
- 目前大概有超过 50 种官方 starter。

在导入的 starter 之后，SpringBoot 主要帮我们完成了两件事情：

- 相关组件的自动导入
- 相关组件的自动配置

这两件事情统一称为 SpringBoot 的自动配置

题目 9：Spring Boot 中的自定义 starter?

1.XxxProperties 属性类

```
@ConfigurationProperties(prefix = "spring.datasource.c3p0")
public class C3p0DataSourceProperties {
    private String driverClassName;
    private String url;
    private String username;
    private String password;
    // 当前仅做测试，其它属性暂时
    // 提供Setter/Getter方法
    public String getDriverClassName() {
        return driverClassName;
    }
```

指定属性的前缀

```
spring:
  datasource:
    c3p0:
      driver-class-name: com.mysql.cj.jdbc.Driver
      url: jdbc:mysql://localhost:3306/study_ssm?se
      username: root
      password: 123456
```

在yml文件中使用时

2.XxxAutoConfigure 配置类

```
@Configuration 表示当前是配置类，被springboot自动扫描到
@EnableConfigurationProperties(C3p0DataSourceProperties.class) 加载属性配置对象
public class C3p0DatasourceAutoConfigure {

    @Autowired
    private C3p0DataSourceProperties dataSourceProperties; 注入属性对象
    /**
     * 提供基于c3p0配置的DataSource实例
     * @return
     */
    @Bean
    public DataSource c3p0DataSource(){
        ComboPooledDataSource dataSource = new ComboPooledDataSource();
        try {
            dataSource.setDriverClass(dataSourceProperties.getDriverClassName(
            dataSource.setJdbcUrl(dataSourceProperties.getUrl());
            dataSource.setUser(dataSourceProperties.getUsername());
            dataSource.setPassword(dataSourceProperties.getPassword());
            return dataSource;
        } catch (PropertyVetoException e) {        构建c3p0数据源实例
            e.printStackTrace();
            new RuntimeException(e.getMessage());
            return null;
        }
```

3.编写 META-INF/spring.factories 文件

在resource目录下创建META-INF文件夹并创建spring.factories

注意："\ "是换行使用的

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
  com.itheima.c3p0.autoconfigure.C3p0DatasourceAutoConfigure
```

题目 10：bootstrap.yml 和 application.yml 有何区别 ？

SpringBoot 两个核心的配置文件：

- **bootstrap**(.yml 或者 .properties)：boostrap 由父 ApplicationContext 加载的，比 applicaton 优先加载，配置在应用程序上下文的引导阶段生效。一般来说我们在 SpringCloud Config 或者 Nacos 中会用到它。且 boostrap 里面的属性不能被覆盖；
- **application** (.yml 或者.properties)：由 ApplicatonContext 加载，用于 SpringBoot 项目的自动化配置。

题目 11：SpringBoot 配置文件加载顺序?

如果在不同的目录中存在多个配置文件，它的读取顺序是：

1. config/application.properties（项目根目录中 config 目录下）
2. config/application.yml
3. application.properties（项目根目录下）
4. application.yml
5. resources/config/application.properties（项目 resources 目录中 config 目录下
6. resources/config/application.yml
7. resources/application.properties（项目的 resources 目录下）
8. resources/application.yml

题目 12：SpringBoot 可以有哪些方式加载配置？

- properties 文件；
- YAML 文件；
- 系统环境变量；
- 命令行参数；

题目 13：SpringBoot 读取配置文件内容的方式有几种？

1. 使用 @Value 注解直接注入对应的值，这能获取到 Spring 中 Environment 的值；
2. 使用 @ConfigurationProperties 注解把对应的值绑定到一个对象；
3. 直接获取注入 Environment 进行获取；
4. @Value PropertySource 和@Value 配合使用（注意不支持操作 yml 格式配置文件）

题目 14：SpringBoot 打成的 jar 和普通的 jar 有什么区别 ？

- SpringBoot 项目最终打包成的 jar 是可执行 jar ，这种 jar 可以直接通过 java -jar xxx.jar 命令来运行，这种 jar 不可以作为普通的 jar 被其他项目依赖，即使依赖了也无法使用其中的类。
- SpringBoot 的 jar 无法被其他项目依赖，主要还是他和普通 jar 的结构不同。普通的 jar 包，解压后直接就是包名，包里就是我们的代码，而 Spring Boot 打包成的可执行 jar 解压后，在 \BOOT-INF\classes 目录下才是我们的代码，因此无法被直接引用。如果非要引用，可以在 pom.xml 文件中增加配置，将 Spring Boot 项目打包成两个 jar ，一个可执行，一个可引用。

题目 15：SpringBoot 命令行启动的参数?

java -jar -Xmx4096m -Xms4096m xxxx.jar --server.port=8088 --spring.profiles.active=prod

通过 Xmx 和 Xms 两个参数指定最大内存 4096M，最小内存 4096M

--server.port 指定端口

--spring.profiles.active 指定 yaml 文件

题目 16：SpringBoot 启动流程?

https://baijiahao.baidu.com/s?id=1666047746833809358&wfr=spider&for=pc

1.SpringApplication 静态调用 run 方法，从静态 run 方法中 new 一个自己的实例，并调用实例的 run 方法。

实例化 SpringApplication 做哪些事情？

1. 推断 WebApplicationType，主要思想就是在当前的 classpath 下搜索特定的类
2. 搜索 META-INF\spring.factories 文件配置的 ApplicationContextInitializer 的实现类
3. 搜索 META-INF\spring.factories 文件配置的 ApplicationListener 的实现类
4. 推断 MainApplication 的 Class

Run 方法做了什么？

1. 创建一个StopWatch并执行start方法，这个类主要记录任务的执行时间
2. 配置Headless属性，Headless模式是在缺少显示屏、键盘或者鼠标时候的系统配置
3. 在文件META-INF\spring.factories中获取SpringApplicationRunListener接口的实现类EventPublishingRunListener，主要发布SpringApplicationEvent
4. 把输入参数转成DefaultApplicationArguments类
5. 创建Environment并设置比如环境信息，系统熟悉，输入参数和profile信息
6. 打印Banner信息
7. 创建Application的上下文，根据WebApplicationTyp来创建Context类，如果非web项目则创建AnnotationConfigApplicationContext，在构造方法中初始化AnnotatedBeanDefinitionReader和ClassPathBeanDefinitionScanner
8. 在文件META-INF\spring.factories中获取SpringBootExceptionReporter接口的实现类FailureAnalyzers
9. 准备application的上下文
10. 刷新上下文，在这里真正加载 bean 到容器中。

Java 热门面试题-微服务框架

题目 1：SpringCloud 和 SpringBoot 的区别和关系？

- Spring Boot 专注于快速方便的开发单个个体微服务。
- Spring Cloud 是关注全局的微服务协调整理治理框架以及一整套的落地解决方案，它 将 Spring Boot 开发的 一个个单体微服务整合并管理起来，为各个微服务之间提供：配置管理，服务发现，断路器，路由，微代理，事件总线等的集成服务。
- Spring Boot 可以离开 Spring Cloud 独立使用，但是 Spring Cloud 离不开 SpringBoot ，属于依赖的关系。

总结： Spring Boot 专注于快速，方便的开发单个微服务个体。 SpringCloud 关注全局的服务治理框架。

题目 2：SpringCloud 由哪些组件组成？

- Eureka 或 Nacos：服务注册与发现
- Zuul 或 SpringCloudGateway：服务网关
- Ribbon：客户端负载均衡
- Feign：声明性的 Web 服务客户端
- Hystrix：断路器
- SpringCloudConfig 或 Nacos：分布式统一配置管理

题目 3：SpringCloud 与 Dubbo 的区别？

	Dubbo	Spring Cloud
服务注册中心	Zookeeper	Spring Cloud Netflix Eureka
服务调用方式	RPC	REST API
服务监控	Dubbo-monitor	Spring Boot Admin
断路器	不完善	Spring Cloud Netflix Hystrix
服务网关	无	Spring Cloud Netflix Zuul
分布式配置	无	Spring Cloud Config
服务跟踪	无	Spring Cloud Sleuth
消息总线	无	Spring Cloud Bus
数据流	无	Spring Cloud Stream
批量任务	无	Spring Cloud Task

服务调用方式：

- dubbo 是 RPC
- springcloud 是 Rest Api

注册中心：

- dubbo 是 zookeeper；
- springcloud 是 eureka，也可以是 zookeeper、nacos

服务网关：

- dubbo 本身没有实现，只能通过其他第三方技术整合；

- springcloud 有 Zuul 路由网关，作为路由服务器，进行消费者的请求分发；springcloud 支持断路器，与 git 完美集成配置文件支持版本控制，事物总线实现配置文件的更新与服务自动装配等等一系列的微服务架构要素。

题目 4：Eureka 和 ZooKeeper 的区别？

- ZooKeeper 中的节点服务挂了就要选举，在选举期间注册服务瘫痪，虽然服务最终会恢复，但是选举期间不可用的， 选举就是改微服务做了集群，必须有一台主其他的都是从。
- Eureka 各个节点是平等关系，服务器挂了没关系，只要有一台 Eureka 就可以保证服务可用，数据都是最新的。如果查询到的数据并不是最新的，就是因为 Eureka 的自我保护模式导致的。
- Eureka 本质上是一个工程，而 ZooKeeper 只是一个进程。
- Eureka 可以很好的应对因网络故障导致部分节点失去联系的情况，而不会像 ZooKeeper 一样使得整个注册系统瘫痪。
- ZooKeeper 保证的是 CP，Eureka 保证的是 AP

题目 5：Eureka 工作原理？

- EurekaServer：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，提供什么服务）
- 消费者：向 Eureka 订阅服务，Eureka 会将对应服务的所有提供者地址列表发送给消费者，并且定期更新
- 心跳(续约)：提供者定期通过 http 方式向 Eureka 刷新自己的状态（每 30s 定时向 EurekaServer 发起请求）

题目 6：Feign 工作原理？

1. 主程序入口添加了@EnableFeignClients 注解开启对 FeignClient 扫描加载处理。根据 FeignClient 的开发规范，定义接口并加@FeignClientd 注解。

b) 当程序启动时，会进行包扫描，扫描所有@FeignClients 的注解的类，并且将这些信息注入 Spring IOC 容器中，当定义的 Feign 接口中的方法被调用时，通过 JDK 的代理方式，来生成具体的 RequestTemplate.

c) 当生成代理时，Feign 会为每个接口方法创建一个 RequestTemplate 对象，改对象封装可 HTTP 请求需要的全部信息，如请求参数名，请求方法等信息都是在这个过程中确定的。

4. 然后 RequestTemplate 生成 Request,然后把 Request 交给 Client 去处理，这里的 Client 可以是 JDK 原生的 URLConnection、Apache 的 HttpClient、也可以是 OKhttp，最后 Client 被封装到 LoadBalanceClient 类，这个类结合 Ribbon 负载均衡发起服务之间的调用。

题目 7：什么是 Hystrix？

在分布式系统,我们一定会依赖各种服务,那么这些个服务一定会出现失败的情况，就会导致雪崩，Hystrix 就是这样的一个工具，防雪崩利器，它具有服务降级，服务熔断，服务隔离，监控等一些防止雪崩的技术。 Hystrix 有四种防雪崩方式：

- 服务降级：接口调用失败就调用本地的方法返回一个空
- 服务熔断：接口调用失败就会进入调用接口提前定义好的一个熔断的方法，返回错误信息
- 服务隔离：隔离服务之间相互影响
- 服务监控：在服务发生调用时,会将每秒请求数、成功请求数等运行指标记录下来。

题目 8：什么是服务熔断？什么是服务降级？

- **熔断机制：** 是应对雪崩效应的一种微服务链路保护机制。当某个微服务不可用或者响应时间太长时，会进行服务降级，进而熔断该节点微服务的调用，快速返回“错误”的响应信息。当检测到该节点微服务调用响应正常后恢复调用链路。在 SpringCloud 框架里熔断机制通过 Hystrix 实现，Hystrix 会监控微服务间调用的状况，当失败的调用到一定阈值，缺省是 5 秒内调用 20 次，如果失败，就会启动熔断机制。
- **服务降级：** 一般是从整体负荷考虑。就是当某个服务熔断之后，服务器将不再被调用，此时客户端可以自己准备一个本地的 fallback 回调，返回一个缺省值

题目 9：什么是服务雪崩效应？

雪崩效应是在大型互联网项目中，当某个服务发生宕机时，调用这个服务的其他服务也会发生宕机，大型项目的微服务之间的调用是互通的，这样就会将服务的不可用逐步扩大到各个其他服务中，从而使整个项目的服务宕机崩溃

题目 10：微服务之间如何独立通讯?

- 同步通信：Dubbo 通过 **RPC 远程过程调用、springcloud 通过 REST 接口 json 调用** 等。
- 异步通信：消息队列，如：**RabbitMq、ActiveM、Kafka** 等。

Java 热门面试题-Redis 专题

题目 1：为什么使用 Redis

1. 速度快，因为数据存在内存中，类似于 HashMap，HashMap 的优势就是查找和操作的时间复杂度都是 O1)
b) 支持丰富数据类型，支持 string，list，set，Zset，hash 等
c) 支持事务，操作都是原子性，所谓的原子性就是对数据的更改要么全部执行，要么全部不执行
4. 丰富的特性：可用于缓存，消息，按 key 设置过期时间，过期后将会自动删除

题目 2：Redis 常见数据结构以及使用场景？

- String
  - –介绍 ：string 数据结构是简单的 key-value 类型。虽然 Redis 是用 C 语言写的，但是 Redis 并没有使用 C 的字符串表示，而是自己构建了一种 **简单动态字符串**（simple dynamic string，**SDS**）。最大能存储 512MB。
  - –常用命令： set,get,strlen,exists,decr,incr,setex 等等。
  - –应用场景： 计数、缓存文章标题、微博内容等。
- List
  - –介绍 ：**list** 即是 **链表**。链表是一种非常常见的数据结构，特点是易于数据元素的插入和删除并且可以灵活调整链表长度，但是链表的随机访问困难。最多可存储 232 - 1 元素(4294967295, 每个列表可存储 40 亿)
  - –常用命令: rpush,lpop,lpush,rpop,lrange,llen 等。
  - –应用场景: 发布与订阅或者说消息队列。

- Hash

  –介绍 ：hash 类似于 JDK1.8 前的 HashMap，内部实现也差不多(数组 + 链表)。不过，Redis 的 hash 做了更多优化。另外，hash 是一个 string 类型的 field 和 value 的映射表，**特别适合用于存储对象**，后续操作的时候，你可以直接仅仅修改这个对象中的某个字段的值。 比如我们可以 hash 数据结构来存储用户信息，商品信息等等。每个 hash 可以存储 232 -1 键值对（40 多亿）

  –常用命令：hset,hmset,hexists,hget,hgetall,hkeys,hvals 等。

  –应用场景: 系统中对象数据的存储。

- Set

  –介绍 ： set 类似于 Java 中的 HashSet 。Redis 中的 set 类型是一种无序集合，集合中的元素没有先后顺序。当你需要存储一个列表数据，又不希望出现重复数据时，set 是一个很好的选择，并且 set 提供了判断某个成员是否在一个 set 集合内的重要接口，这个也是 list 所不能提供的。可以基于 set 轻易实现交集、并集、差集的操作。比如：你可以将一个用户所有的关注人存在一个集合中，将其所有粉丝存在一个集合。Redis 可以非常方便的实现如共同关注、共同粉丝、共同喜好等功能。这个过程也就是求交集的过程。最大的成员数为 232 -1(4294967295, 每个 集合可存储 40 多亿个成员)。

  –常用命令：sadd,spop,smembers,sismember,scard,sinterstore,sunion 等。

  –应用场景: 需要存放的数据不能重复以及需要获取多个数据源交集和并集等场景

- SortedSet(zset)

  –介绍：SortedSet 和 set 相比，SortedSet 增加了一个权重参数 score，使得集合中的元素能够按 score 进行有序排列，还可以通过 score 的范围来获取元素的列表。有点像是 Java 中 HashMap 和 TreeSet 的结合体。

  –常用命令：zadd,zcard,zscore,zrange,zrevrange,zrem 等。

应用场景： 需要对数据根据某个权重进行排序的场景。比如在直播系统中，实时排行信息包含直播间在线用户列表，各种礼物排行榜，弹幕消息（可以理解为按消息维度的消息排行榜）等信息。

- 缓存
- 排行榜

- 分布式计数器
- 分布式锁
- 消息队列
- 分布式 token
- 限流

题目 3：Redis 为什么快？

- （内存操作）完全基于内存，绝大部分请求是纯粹的内存操作，非常快速。
- （单线程，省去线程切换、锁竞争的开销）采用单线程，避免了不必要的上下文切换和竞争条件，也不存在多进程或者多线程导致的切换而消耗 CPU，不用去考虑各种锁的问题，不存在加锁释放锁操作，没有因为可能出现死锁而导致的性能消耗；
- （NIO 的 IO 多路复用模型）使用多路 I/O 复用模型，非阻塞 IO；这里“多路”指的是多个网络连接，“复用”指的是复用同一个线程

题目 4：Redis 过期删除策略？

常用的过期数据的删除策略就两个：

- **惰性删除** ：只会在取出 key 的时候才对数据进行过期检查。这样对 CPU 最友好，但是可能会造成太多过期 key 没有被删除。
- **定期删除** ： 每隔一段时间抽取一批 key 执行删除过期 key 操作。并且，Redis 底层会通过限制删除操作执行的时长和频率来减少删除操作对 CPU 时间的影响。

定期删除对内存更加友好，惰性删除对 CPU 更加友好。两者各有千秋，所以 Redis 采用的是 **定期删除+惰性/懒汉式删除** 。

但是，仅仅通过给 key 设置过期时间还是有问题的。因为还是可能存在定期删除和惰性删除漏掉了很多过期 key 的情况。这样就导致大量过期 key 堆积在内存里，然后就 Out of memory 了。

怎么解决这个问题呢？答案就是：**Redis 内存淘汰机制。**

题目 5：Redis 内存淘汰策略？

Redis 提供 6 种数据淘汰策略：

- **volatile-lru（least recently used）**：从已设置过期时间的数据集（server.db[i].expires）中挑选最近最少使用的数据淘汰
- **volatile-ttl**：从已设置过期时间的数据集（server.db[i].expires）中挑选将要过期的数据淘汰
- **volatile-random**：从已设置过期时间的数据集（server.db[i].expires）中任意选择数据淘汰
- **allkeys-lru（least recently used）**：当内存不足以容纳新写入数据时，在键空间中，移除最近最少使用的 key（这个是最常用的）
- **allkeys-random**：从数据集（server.db[i].dict）中任意选择数据淘汰
- **no-eviction**：禁止驱逐数据，也就是说当内存不足以容纳新写入数据时，新写入操作会报错。这个应该没人使用吧！

4.0 版本后增加以下两种：

- **volatile-lfu（least frequently used）**：从已设置过期时间的数据集（server.db[i].expires）中挑选最不经常使用的数据淘汰
- **allkeys-lfu（least frequently used）**：当内存不足以容纳新写入数据时，在键空间中，移除最不经常使用的 key

题目 6：Redis 持久化机制 RDB 和 AOF 区别？

持久化方式	RDB	AOF
占用存储空间	小（数据级：压缩）	大（指令级：重写）
存储速度	慢	快
恢复速度	快	慢
数据安全性	会丢失数据	依据策略决定
资源消耗	高/重量级	低/轻量级
启动优先级	低	高

image-20220121165750009

题目 7：Redis 如何选择合适的持久化方式

- 如果是数据不那么敏感，且可以从其他地方重新生成补回的，那么可以关闭持久化。
- 如果是数据比较重要，不想再从其他地方获取，且可以承受数分钟的数据丢失，比如缓存等，那么可以只使用 RDB。
- 如果是用做内存数据库，要使用 Redis 的持久化，建议是 RDB 和 AOF 都开启，或者定期执行 bgsave 做快照备份，RDB 方式更适合做数据的备份，AOF 可以保证数据的不丢失。

补充：Redis4.0 对于持久化机制的优化

Redis4.0 相对与 3.X 版本其中一个比较大的变化是 4.0 添加了新的混合持久化方式。

简单的说：新的 AOF 文件前半段是 RDB 格式的全量数据后半段是 AOF 格式的增量数据

- **优势**：混合持久化结合了 RDB 持久化 和 AOF 持久化的优点， 由于绝大部分都是 RDB 格式，加载速度快，同时结合 AOF，增量的数据以 AOF 方式保存了，数据更少的丢失。
- **劣势**：兼容性差，一旦开启了混合持久化，在 4.0 之前版本都不识别该 aof 文件，同时由于前部分是 RDB 格式，阅读性较差。

题目 8：在生成 RDB 期间，Redis 可以同时处理写请求么？

可以的，Redis 使用操作系统的多进程**写时复制技术 COW(Copy On Write)** 来实现快照持久化，保证数据一致性。Redis 在持久化时会调用 glibc 的函数 fork 产生一个子进程，快照持久化完全交给子进程来处理，父进程继续处理客户端请求。当主线程执行写指令修改数据的时候，这个数据就会复制一份副本， bgsave 子进程读取这个副本数据写到 RDB 文件。这既保证了快照的完整性，也允许主线程同时对数据进行修改，避免了对正常业务的影响。

题目 9：如何保存 Redis 数据与 DB 一致？

- 方案 1：同步双写，即更新完 DB 后立即同步更新 redis
- 方案 2：异步监听，即通过 Canal 监听 MySQL 变化的表，同步更新数据到 Redis
- 方案 3：MQ 异步，即更新完 DB 后生产消息到 MQ，MQ 消费者更新数据到 Redis

题目 10：Redis 的什么是缓存预热?

缓存预热是指系统上线后，提前将相关的缓存数据加载到缓存系统。避免在用户请求的时候，先查询数据库，然后再将数据缓存的问题，用户直接查询事先被预热的缓存数据。

如果不进行预热，那么 Redis 初始状态数据为空，系统上线初期，对于高并发的流量，都会访问到数据库中， 对数据库造成流量的压力。

缓存预热解决方案：

- 数据量不大的时候，工程启动的时候进行加载缓存动作；
- 数据量大的时候，设置一个定时任务脚本，进行缓存的刷新；
- 数据量太大的时候，优先保证热点数据进行提前加载到缓存。

题目 11：什么是缓存降级？

缓存降级是指缓存失效或缓存服务器挂掉的情况下，不去访问数据库，直接返回默认数据或访问服务的内存数据。降级一般是有损的操作，所以尽量减少降级对于业务的影响程度。

在进行降级之前要对系统进行梳理，看看系统是不是可以丢卒保帅；从而梳理出哪些必须誓死保护，哪些可降级；比如可以参考日志级别设置预案：

- **一般**：比如有些服务偶尔因为网络抖动或者服务正在上线而超时，可以自动降级；
- **警告**：有些服务在一段时间内成功率有波动（如在 95~100%之间），可以自动降级或人工降级，并发送告警；
- **错误**：比如可用率低于 90%，或者数据库连接池被打爆了，或者访问量突然猛增到系统能承受的最大阀值，此时可以根据情况自动降级或者人工降级；
- **严重错误**：比如因为特殊原因数据错误了，此时需要紧急人工降级。

题目 12：Redis 的缓存雪崩、缓存穿透、缓存击穿

- 缓存穿透

  –缓存穿透：缓存中和数据库中都没有所查询的东西，从而使数据库崩掉。

–解决方案：将一条数据库不存在的数据也放入缓存中这样即使数据库不存在但缓存中有，还可以使用布隆过滤器。

- 缓存击穿

–缓存击穿：缓存中没有但数据库中有，如果同一时间访问量过大会使数据崩掉。

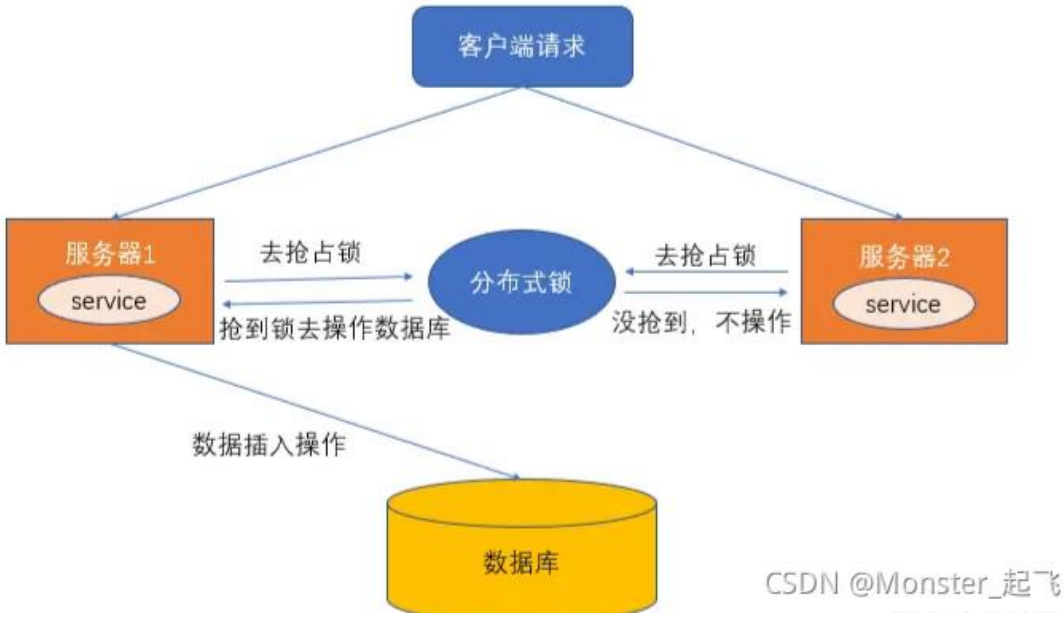
–解决方案：加分布式锁，一条数据访问数据库；将数据存储到缓存中，其他线程从缓存中拿。

- 缓存雪崩

–缓存雪崩：缓存中的数据正好在一个时间删除，当请求来时穿过缓存访问数据库。

–解决方案：1.提前预热；2.设置随机缓存中数据的过期时间；3.做缓存备份

题目 13：常见的分布式锁有哪些解决方案？



分布式锁一般有如下的特点：

- 互斥性： 同一时刻只能有一个线程持有锁
- 可重入性： 同一节点上的同一个线程如果获取了锁之后能够再次获取锁
- 锁超时：和J.U.C中的锁一样支持锁超时，防止死锁
- 高性能和高可用： 加锁和解锁需要高效，同时也需要保证高可用，防止分布式锁失效
- 具备阻塞和非阻塞性：能够及时从阻塞状态中被唤醒

实现分布式锁目前有三种流行方案，即基于关系型数据库、Redis、ZooKeeper 的方案

1、基于关系型数据库，如 MySQL

基于关系型数据库实现分布式锁，是依赖数据库的唯一性来实现资源锁定，比如主键和唯一索引等。

- 缺点：
  - –这把锁强依赖数据库的可用性，数据库是一个单点，一旦数据库挂掉，会导致业务系统不可用。
  - –这把锁没有失效时间，一旦解锁操作失败，就会导致锁记录一直在数据库中，其他线程无法再获得到锁。
  - –这把锁只能是非阻塞的，因为数据的 insert 操作，一旦插入失败就会直接报错。没有获得锁的线程并不会进入排队队列，要想再次获得锁就要再次触发获得锁操作。
  - –这把锁是非重入的，同一个线程在没有释放锁之前无法再次获得该锁。因为数据中数据已经存在了。

2、基于 Redis 实现

https://blog.csdn.net/Monsterof/article/details/120529385

使用 set key value [EX seconds][PX milliseconds][NX|XX] 命令 (正确做法)

- EX seconds: 设定过期时间，单位为秒
- PX milliseconds: 设定过期时间，单位为毫秒
- NX: 仅当key不存在时设置值
- XX: 仅当key存在时设置值

在 Redis 集群的时候也会出现问题，比如说 A 客户端在 Redis 的 master 节点上拿到了锁，但是这个加锁的 key 还没有同步到 slave 节点，master 故障，发生故障转移，一个 slave 节点升级为 master 节点，B 客户端也可以获取同个 key 的锁，但客户端 A 也已经拿到锁了，这就导致多个客户端都拿到锁。

Redisson 实现简单分布式锁，Redisson 底层使用 Netty 可以实现非阻塞 I/O，该客户端封装了锁的，继承了 J.U.C 的 Lock 接口，所以我们可以像使用 ReentrantLock 一样使用 Redisson。

```
//2. 构造RedissonClient
RedissonClient redissonClient = Redisson.create(config);

//3. 设置锁定资源名称
RLock lock = redissonClient.getLock("redlock");
lock.lock();
try {
    System.out.println("获取锁成功，实现业务逻辑");
    Thread.sleep(10000);
} catch (InterruptedException e) {
    e.printStackTrace();
} finally {
    lock.unlock();
}
```

优点：

- Redis 锁实现简单，理解逻辑简单，性能好，可以支撑高并发的获取、释放锁操作。

缺点：

- Redis 容易单点故障，集群部署，并不是强一致性的，锁的不够健壮；
- key 的过期时间设置多少不明确，只能根据实际情况调整；
- 需要自己不断去尝试获取锁，比较消耗性能。

3、基于 zookeeper

优点：

- zookeeper 天生设计定位就是分布式协调，强一致性，锁很健壮。如果获取不到锁，只需要添加一个监听器就可以了，不用一直轮询，性能消耗较小。

缺点：

- 在高请求高并发下，系统疯狂的加锁释放锁，最后 zk 承受不住这么大的压力可能会存在宕机的风险。

题目 14：Redis 主从架构数据会丢失吗，为什么？

有两种数据丢失的情况：

- 异步复制导致的数据丢失：因为 master -> slave 的复制是异步的，所以可能有部分数据还没复制到 slave，master 就宕机了，此时这些部分数据就丢失了。
- 脑裂导致的数据丢失：某个 master 所在机器突然脱离了正常的网络，跟其他 slave 机器不能连接，但是实际上 master 还运行着，此时哨兵可能就会认为 master 宕机了，然后开启选举，将其他 slave 切换成了 master。这个时候，集群里就会有两个 master，也就是所谓的脑裂。此时虽然某个 slave 被切换成了 master，但是可能 client 还没来得及切换到新的 master，还继续写向旧 master 的数据可能也丢失了。因此旧 master 再次恢复的时候，会被作为一个 slave 挂到新的 master 上去，自己的数据会清空，重新从新的 master 复制数据。

题目 15：Redis 如何做内存优化？

- **控制 key 的数量**：当使用 Redis 存储大量数据时，通常会存在大量键，过多的键同样会消耗大量内存。Redis 本质是一个数据结构服务器，它为我们提供多种数据结构，如 hash，list，set，zset 等结构。使用 Redis 时不要进入一个误区，大量使用 get/set 这样的 API，把 Redis 当成 Memcached 使用。对于存储相同的数据内容利用 Redis 的数据结构降低外层键的数量，也可以节省大量内存。
- **缩减键值对象**，降低 Redis 内存使用最直接的方式就是缩减键（key）和值（value）的长度。
  - –key 长度：如在设计键时，在完整描述业务情况下，键值越短越好。
  - –value 长度：值对象缩减比较复杂，常见需求是把业务对象序列化成二进制数组放入 Redis。首先应该在业务上精简业务对象，去掉不必要的属性避免存储无效数据。其次在序列化工具选择上，应该选择更高效的序列化工具来降低字节数组大小。
- **编码优化**。Redis 对外提供了 string,list,hash,set,zet 等类型，但是 Redis 内部针对不同类型存在编码的概念，所谓编码就是具体使用哪种底层数据结构来实现。编码不同将直接影响数据的内存占用和读写效率。

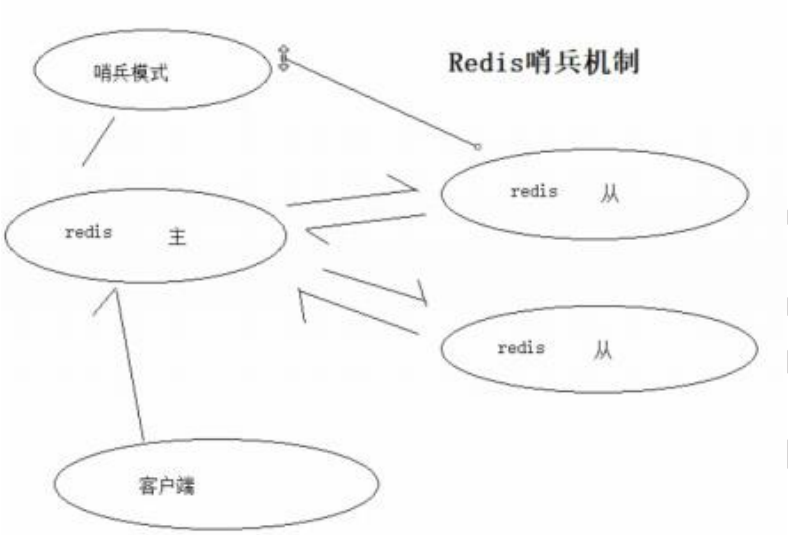
题目 16：Redis 集群

1.主从模式

主从复制,读写分离,,主挂了则不提供写的服务.

1. 主可以进行读写,新数据会自动同步给从
2. 从主要负责读
3. 一主多从
4. 从挂了,不影响读写,主挂了,redis将不提供写的服务,不会新推选主,知道主恢复了,才提供写服务

2.哨兵模式



1. 主从模式基础上增加选机制,主挂了,从 从中新推选主

   保证redis高可用,客户端连接主redis,,然后主redis会与从redis进行同步,如果主redis挂了,,则会从redis从中推选出一个主,然后客户端的连接会自动的连接新的主,,

2. 主如果发现自己无法连接redis从,则会拒绝写的操作

   如果主发现自己的新数据无法给redis从同步时(最大10秒连接不上redis从),那么他会拒绝客户端的写的操作,..只允许读.(解决,新的数据无法同步到从,然后主宕机了,,丢失数据的问题,因为新推选的主会没有那部分数据)

3. Cluster(分片模式)

在哨兵模式的基础上增加数据分片,就是根据key的hash值平均分配的每个redis服务节点上,每个服务节点都是一主一从(或多从),其中从 不提供服务,仅作为备用

1. sentinel和主从模式的结合体+数据分片
2. redis将key平均分配储存到每个redis服务节点上
3. 每个服务节点都是一主一从(或多从),其中从 不提供服务,仅作为备用
4. 根据hash值来分.

Java 热门面试题- Monngo&ES 专题

题目 1：MongoDB 的优势有哪些？

- 面向文档的存储：以 JSON 格式的文档保存数据
- 任何属性都可以建立索引
- 高性能及高可扩展性
- 自动分片
- 丰富的查询功能
- 快速的即时更新

题目 2：MongoDB 与 MySQL 的区别是什么？

mongodb 的本质还是一个数据库产品，3.0 以上版本其稳定性和健壮性有很大提升。它与 mysql 的区别在于它不会遵循一些约束，比如：sql 标准、ACID 属性，表结构等。其主要特性如下：

- 面向集合文档的存储：适合存储 Bson（json 的扩展）形式的数据；
- 格式自由，数据格式不固定，生产环境下修改结构都可以不影响程序运行；
- 强大的查询语句，面向对象的查询语言，基本覆盖 sql 语言所有能力；
- 完整的索引支持，支持查询计划；
- 支持复制和自动故障转移；
- 支持二进制数据及大型对象（文件）的高效存储；
- 使用分片集群提升系统扩展性；
- 使用内存映射存储引擎，把磁盘的 IO 操作转换成为内存的操作；

题目 3：MongoDB 中概念有哪里与 MySQL 不一样的？

比较	MySQL	MongoDB
库	database	database
表	table	collection
行	row	document
列	column	field
索引	index	index
表关联	table joins	$lookup
主键	primary key	primary key
聚合	aggregation	aggregation pipeline

image-20220121202453240

题目 4：MongoDB 文档的 Objectld 由什么构成？

- 时间戳
- 客户机 ID
- 客户端进程 ID
- 3 字节递增计数器

题目 5：mongodb 数据为什么会丢失?

之前版本是这样的,现在默认开启Journal日志

1. 数据->内存->60秒刷到硬盘上
   a. 数据在写入内存之后即刻返回给应用程序。而数据刷盘动作则在后台由操作系统来进行。MongoDB会每隔60秒强制把数据刷到磁盘上
2. 解决方案:开启Journal日志
   a. 数据->Journal Buffer->内存数据
      a. ->100ms到磁盘上
      i. MongoDB会先把数据更新写入到Journal Buffer里面然后再更新内存数据，然后再返回给应用端。Journal会以100ms的间隔批量刷到盘上

题目 6：ES 索引体系包含哪些内容?

Index 索引	Database 数据库
Type 文档类型	Table 表
Document 文档	Row 记录
Field 字段	Column 属性
Mapping 映射	Schema 模型
Query DSL	SQL

题目 7：ES 为什么这么快（什么是倒排索引）？

- 倒排索引是搜索引擎的核心。搜索引擎的主要目标是在查找发生搜索条件的文档时提供快速搜索。倒排索引是一种像数据结构一样的散列图，可将用户从单词导向文档或网页。它是搜索引擎的核心。其主要目标是快速搜索从数百万文件中查找数据。
- 传统的我们的检索是通过文章，逐个遍历找到对应关键词的位置。而倒排索引，是通过分词策略，形成了词和文章的映射关系表，这种词典+映射表即为倒排索引。有了倒排索引，就能实现 o（1）时间复杂度的效率检索文章了，极大的提高了检索效率。

要注意倒排索引的两个重要细节：

- 倒排索引中的所有词项对应一个或多个文档

- 倒排索引中的词项 根据字典顺序升序排列

题目 8：ES 的索引是什么？

- 索引（名词）： 一个索引(index)就像是传统关系数据库中的数据库，它是相关文档存储的地方，index 的复数是 indices 或 indexes。
- 索引（动词）：「索引一个文档」表示把一个文档存储到索引（名词）里，以便它可以被检索或者查询。这很像 SQL 中的 INSERT 关键字，差别是，如果文档已经存在，新的文档将覆盖旧的文档。

题目 9：ES 中字符串类型有几个？区别是什么？

有两个 keyword 和 Text，两个的区别主要分词的区别：

- keyword 类型是不会分词的，直接根据字符串内容建立倒排索引，keyword 类型的字段只能通过精确值搜索
- Text 类型在存入 Elasticsearch 的时候，会先分词，然后根据分词后的内容建立倒排索引

题目 10：ES 中 query 和 filter 的区别？

- query：查询操作不仅仅会进行查询，还会计算分值，用于确定相关度；
- filter：查询操作仅判断是否满足查询条件，不会计算任何分值，也不会关心返回的排序问题，同时，filter 查询的结果可以被缓存，提高性能。

题目 11：如何解决 ES 集群的脑裂问题

所谓集群脑裂，是指 Elasticsearch 集群中的节点（比如共 20 个），其中的 10 个选了一个 master，另外 10 个选了另一个 master 的情况。

- 当集群中 master 候选节点数量不小于 3 个时（node.master: true），可以通过设置最少投票通过数量（discovery.zen.minimum_master_nodes），设置超过所有候选节点一半以上来解决脑裂问题，即设置为 (N/2)+1；
- 当集群 master 候选节点 只有两个时，这种情况是不合理的，最好把另外一个 node.master 改成 false。如果我们不改节点设置，还是套上面的(N/2)+1 公式，

此时 discovery.zen.minimum_master_nodes 应该设置为 2。这就出现一个问题，两个 master 备选节点，只要有一个挂，就选不出 master 了

题目 12：ES 索引数据多了怎么办，如何调优，部署？

索引数据的规划，应在前期做好规划，正所谓“设计先行，编码在后”，这样才能有效的避免突如其来的数据激增导致集群处理能力不足引发的线上客户检索或者其他业务受到影响。 如何调优，正如问题 1 所说，这里细化一下：

- 动态索引层面

  基于模板+时间+rollover api 滚动创建索引，举例：设计阶段定义：blog 索引的模板格式为：blog_index_时间戳的形式，每天递增数据。

  这样做的好处：不至于数据量激增导致单个索引数据量非常大，接近于上线 2 的 32 次幂-1，索引存储达到了 TB+甚至更大。

  一旦单个索引很大，存储等各种风险也随之而来，所以要提前考虑+及早避免。

- 存储层面

  冷热数据分离存储，热数据（比如最近 3 天或者一周的数据），其余为冷数据。对于冷数据不会再写入新数据，可以考虑定期 force_merge 加 shrink 压缩操作，节省存储空间和检索效率。

- 部署层面

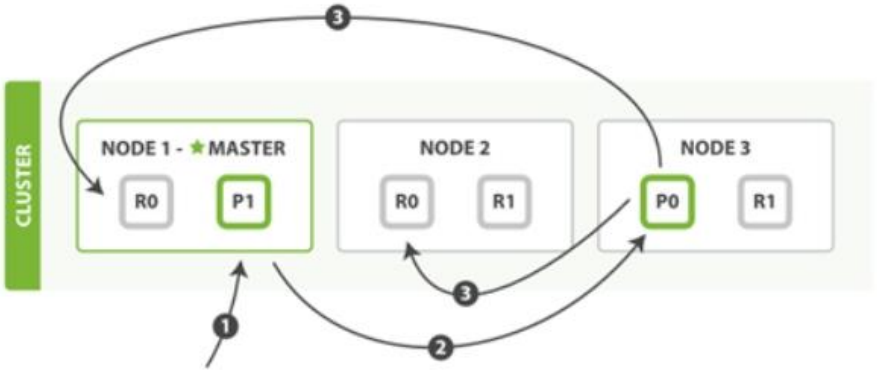
  一旦之前没有规划，这里就属于应急策略。 结合 ES 自身的支持动态扩展的特点，动态新增机器的方式可以缓解集群压力，注意：如果之前主节点等规划合理，不需要重启集群也能完成动态新增的。

题目 13：详细描述一下 ES 搜索的过程？

搜索拆解为“query then fetch” 两个阶段。

- **query 阶段的目的**：定位到位置，但不取。 步骤拆解如下：
  - –1）假设一个索引数据有 5 主+1 副本 共 10 分片，一次请求会命中（主或者副本分片中）的一个。
  - –2）每个分片在本地进行查询，结果返回到本地有序的优先队列中。
  - –3）第 2）步骤的结果发送到协调节点，协调节点产生一个全局的排序列表。
- **fetch 阶段的目的**：取数据。 路由节点获取所有文档，返回给客户端。

题目 14：详细描述一下 ES 索引文档的过程？



这里的索引文档应该理解为文档写入 ES，创建索引的过程。文档写入包含：单文档写入和批量 bulk 写入，这里只解释一下：单文档写入流程。

- 第一步：客户写集群某节点写入数据，发送请求。（如果没有指定路由/协调节点，请求的节点扮演路由节点的角色。）
- 第二步：节点 1 接受到请求后，使用文档_id 来确定文档属于分片 0。请求会被转到另外的节点，假定节点 3。因此分片 0 的主分片分配到节点 3 上。
- 第三步：节点 3 在主分片上执行写操作，如果成功，则将请求并行转发到节点 1 和节点 2 的副本分片上，等待结果返回。所有的副本分片都报告成功，节点 3 将向协调节点（节点 1）报告成功，节点 1 向请求客户端报告写入成功。

第二步中的文档获取分片的过程？

- 借助路由算法获取，路由算法就是根据路由和文档 id 计算目标的分片 id 的过程。
- 。

Java 热门面试题- MQ&Kafka 专题

题目 1：哪些场景下会选择 Kafka？

- 日志收集：一个公司可以用 Kafka 可以收集各种服务的 log，通过 kafka 以统一接口服务的方式开放给各种 consumer，例如 hadoop、HBase、Solr 等。
- 消息系统：解耦和生产者和消费者、缓存消息等。
- 用户活动跟踪：Kafka 经常被用来记录 web 用户或者 app 用户的各种活动，如浏览网页、搜索、点击等活动，这些活动信息被各个服务器发布到 kafka 的 topic

中，然后订阅者通过订阅这些 topic 来做实时的监控分析，或者装载到 hadoop、数据仓库中做离线分析和挖掘。

- 运营指标：Kafka 也经常用来记录运营监控数据。包括收集各种分布式应用的数据，生产各种操作的集中反馈，比如报警和报告。
- 流式处理：比如 Spark Streaming 和 Flink

题目 2：Kafka 架构包含哪些内容？

Kafka 架构分为以下几个部分

- Producer ：消息生产者，就是向 kafka broker 发消息的客户端。
- Consumer ：消息消费者，向 kafka broker 取消息的客户端。
- Topic ：可以理解为一个队列，一个 Topic 又分为一个或多个分区，
- Consumer Group：这是 kafka 用来实现一个 topic 消息的广播（发给所有的 consumer）和单播（发给任意一个 consumer）的手段。一个 topic 可以有多个 Consumer Group。
- Broker ：一台 kafka 服务器就是一个 broker。一个集群由多个 broker 组成。一个 broker 可以容纳多个 topic。
- Partition：为了实现扩展性，一个非常大的 topic 可以分布到多个 broker 上，每个 partition 是一个有序的队列。partition 中的每条消息都会被分配一个有序的 id（offset）。将消息发给 consumer，kafka 只保证按一个 partition 中的消息的顺序，不保证一个 topic 的整体（多个 partition 间）的顺序。
- Offset：kafka 的存储文件都是按照 offset.kafka 来命名，用 offset 做名字的好处是方便查找。例如你想找位于 2049 的位置，只要找到 2048.kafka 的文件即可。当然 the first offset 就是 00000000000.kafka。

题目 3：Kafka 分区的目的？

分区对于 Kafka 集群的好处是：实现负载均衡。分区对于消费者来说，可以提高并发度，提高效率。

题目 4：Kafka 是如何做到消息的有序性？

kafka 中的每个 partition 中的消息在写入时都是有序的，而且单独一个 partition 只能由一个消费者去消费，可以在里面保证消息的顺序性。但是分区之间的消息是不保证有序的。

题目 5：Kafka Producer 如何提升系统发送消息的效率？

- 增加 partition 数
- 提高 batch.size
- 压缩消息
- 异步发送

题目 6：Kafka 发送数据，ack 为 0，1，-1 分别是什么意思？

- 1（默认） 数据发送到 Kafka 后，经过 leader 成功接收消息的的确认，就算是发送成功了。在这种情况下，如果 leader 宕机了，则会丢失数据。
- 0 生产者将数据发送出去就不管了，不去等待任何返回。这种情况下数据传输效率最高，但是数据可靠性确是最低的。
- -1producer 需要等待 ISR 中的所有 follower 都确认接收到数据后才算一次发送完成，可靠性最高。当 ISR 中所有 Replica 都向 Leader 发送 ACK 时，leader 才 commit，这时候 producer 才能认为一个请求中的消息都 commit 了。

题目 7：Kafka 中 consumer group 是什么概念？

同样是逻辑上的概念，是 Kafka 实现单播和广播两种消息模型的手段。

- 同一个 topic 的数据，会广播给不同的 group；
- 同一个 group 中的 worker，只有一个 worker 能拿到这个数据。
- 换句话说，对于同一个 topic，每个 group 都可以拿到同样的所有数据，但是数据进入 group 后只能被其中的一个 worker 消费。group 内的 worker 可以使用多线程或多进程来实现，也可以将进程分散在多台机器上，worker 的数量通常不超过 partition 的数量，且二者最好保持整数倍关系，因为 Kafka 在设计时假定了一个 partition 只能被一个 worker 消费（同一 group 内）。

题目 8：Kafka 消息丢失和重复消费怎么处理？

- **针对消息丢失**：同步模式下，确认机制设置为-1，即让消息写入 Leader 和 Follower 之后再确认消息发送成功；异步模式下，为防止缓冲区满，可以在配置文件设置不限制阻塞超时时间，当缓冲区满时让生产者一直处于阻塞状态；
- **针对消息重复**：

  –将消息的唯一标识保存到外部介质中，每次消费时判断是否处理过即可。

  –消费者处理业务添加分布式锁控制保证业务幂等性

题目 9：Kafka follower 如何与 leader 同步数据?

kafka 的复制机制既不是完全的同步复制，也不是单纯的异步复制。

- 完全同步复制要求 All Alive Follower 都复制完，这条消息才会被认为 commit，这种复制方式极大的影响了吞吐率。
- 异步复制方式下，Follower 异步的从 Leader 复制数据，数据只要被 Leader 写入 log 就被认为已经 commit，这种情况下，如果 leader 挂掉，会丢失数据；
- kafka 使用 ISR 的方式很好的均衡了确保数据不丢失以及吞吐率。Follower 可以批量的从 Leader 复制数据，而且 Leader 充分利用磁盘顺序读以及 send file(zero copy)机制，这样极大的提高复制性能，内部批量写磁盘，大幅减少了 Follower 与 Leader 的消息量差。

题目 10：Kafka 什么情况下一个 broker 会从 ISR 中被踢出？

leader 会维护一个与其基本保持同步的 Replica 列表，该列表称为 ISR(in-sync Replica)，每 个 Partition 都会有一个 ISR，而且是由 leader 动态维护 ，如果一个 follower 比一个 leader 落后太多，或者超过一定时间未发起数据复制请求，则 leader 将其重 ISR 中移除 。

题目 11：为什么 Kafka 不支持读写分离？

在 Kafka 中，生产者写入消息、消费者读取消息的操作都是与 leader 副本进行交互的，从 而 实现的是一种主写主读的生产消费模型。

Kafka 并不支持主写从读，因为主写从读有 2 个很明显的缺点:

- 数据一致性问题。数据从主节点转到从节点必然会有一个延时的时间窗口，这个时间 窗口 会导致主从节点之间的数据不一致。某一时刻，在主节点和从节点中 A 数据的值都为 X， 之后 将主节点中 A 的值修改为 Y，那么在这个变更通知到从节点之前，应用读取从节点中的 A 数据 的值并不为最新的 Y，由此便产生了数据不一致的问题。

- 延时问题。类似 Redis 这种组件，数据从写入主节点到同步至从节点中的过程需要经 历网 络→主节点内存→网络→从节点内存这几个阶段，整个过程会耗费一定的时间。而在 Kafka 中，主从同步会比 Redis 更加耗时，它需要经历网络→主节点内存→主节点磁盘→网络→从节 点内存→从节点磁盘这几个阶段。对延时敏感的应用而言，主写从读的功能并不太适用。

题目 12：解释下 Kafka 中偏移量(offset)的是什么？

在 Kafka 中，每个 主题分区下的每条消息都被赋予了一个唯一的 ID 数值，用于标识它在分区中的位置。这个 ID 数值，就被称为位移，或者叫偏移量。一旦消息被写入到分区日志，它的位移值将不能 被修改。

题目 13：Kafka 消费消息是采用 Pull 模式，还是 Push 模式？

Kafka 还是选取了传统的 pull 模式

- Pull 模式的好处是 consumer 可以自主决定是否批量的从 broker 拉取数据。Push 模式必须在不知道下游 consumer 消费能力和消费策略的情况下决定是立即推送每条消息还是缓存之后批量推送。如果为了避免 consumer 崩溃而采用较低的推送速率，将可能导致一次只推送较少的消息而造成浪费。Pull 模式下，consumer 就可以根据自己的消费能力去决定这些策略
- Pull 有个缺点是，如果 broker 没有可供消费的消息，将导致 consumer 不断在循环中轮询，直到新消息到 t 达。为了避免这点，Kafka 有个参数可以让 consumer 阻塞知道新消息到达(当然也可以阻塞知道消息的数量达到某个特定的量这样就可以批量发

题目 14：Kafka 创建 Topic 时如何将分区放置到不同的 Broker 中？

- 副本因子不能大于 Broker 的个数；
- 第一个分区（编号为 0）的第一个副本放置位置是随机从 brokerList 选择的；
- 其他分区的第一个副本放置位置相对于第 0 个分区依次往后移。也就是如果我们有 5 个 Broker，5 个分区，假设第一个分区放在第四个 Broker 上，那么第二个分区将会放在第五个 Broker 上；第三个分区将会放在第一个 Broker 上；第四个分区将会放在第二个 Broker 上，依次类推；
- 剩余的副本相对于第一个副本放置位置其实是由 nextReplicaShift 决定的，而这个数也是随机产生的

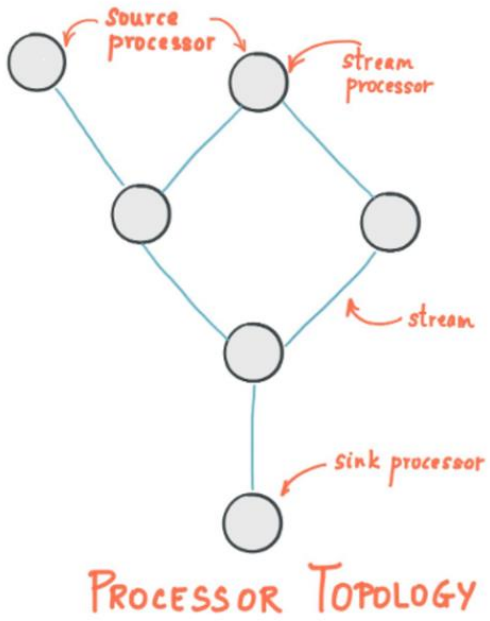
题目 15：Kafka 中 partition 的数据如何保存到硬盘？

topic 中的多个 partition 以文件夹的形式保存到 broker，每个分区序号从 0 递增，且消息有序 Partition 文件下有多个 segment（xxx.index，xxx.log）segment 文件里的 大小和配置文件大小一致可以根据要求修改 默认为 1g 如果大小大于 1g 时，会滚动一个新的 segment 并且以上一个 segment 最后一条消息的偏移量命名

题目 16：Kafka 什么时候会触发 Rebalance

- Topic 分区数量变更时
- 消费者数量变更时

题目 17：KafkaStream 的工作原理？



实时流式计算包含了 **无界数据 近实时 一致性 可重复结果** 等等特征。**a type of data processing engine that is designed with infinite data sets in mind** 一种考虑了无线数据集的数据处理引擎。

1、无限数据：一种不断增长的，基本上无限的数据集。这些通常被称为“流式数据”。无限的流式数据集可以称为无界数据，相对而言有限的批量数据就是有界数据。

2、无界数据处理：一种持续的数据处理模式，应用于上面的无界数据。批量处理数据（离线计算）也可以重复运行来处理数据，但是会有性能的瓶颈。

3、低延迟，近实时的结果：相对于离线计算而言，离线计算并没有考虑延迟的问题。

Kafka Streams 被认为是开发实时应用程序的最简单方法。它是一个 Kafka 的客户端 API 库，编写简单的 java 和 scala 代码就可以实现流式处理。流式计算工作原理：

- 生产者 生产原始数据到入口 topic
- 流式计算（拓扑图）
  - –第一阶段：从入口 topic 获取原始数据，返回列表数据给第二阶段
  - –第二阶段：遍历第一阶段的列表数据，返回需要聚合的 key
  - –第三阶段：聚合计数，将最终数据转发给出口 topic

- 消费者 订阅出口 topic 消费数据处理业务

题目 18：Kafka 中的幂等是怎么实现的？

kafka 幂等性指的是消费者处理的幂等性，一旦出现重复消费消息，那么我们要避免重复消息对业务带来的影响。

解决方案是：生产的消息都要带上业务唯一 ID，消费消息时，可以根据唯一 ID 添加基于 Redis 的分布式锁防止消息重复进行业务处理。

题目 19：KafkaConsumer 是非线程安全的，那么怎么样实现多线程消费？

kafka 消费者多线程消费不应该是一个消费者主线程拉取到消息后，将消息放线程池交给多个 worker 子线程处理，这种是没法保证线程安全的。

正确的用法应该是同一消费者群组内多个消费者订阅同一主题不同分区的数据，进行拉取消息消费，消费者彼此之间没有影响，我们可以理解多个消费者就是多个主线程，不存在 worker 子线程，所以这种用法不存在线程安全问题。

项目相关

1. 你们的数据库大概有多少数据？

目前处于早期运营阶段，已上架文章大概 20 万条左右

2. 你们的敏感词表（DFA 算法要用到的表）有多大？

大概有 1 万条

3. 你们的 Redis 集群用了多少节点？

目前 redis 集群是 3 主 3 从

4. 项目开发当中碰到了哪些难题？说三个

比如之前文章审核的效率就不是很高，我们之前审核时用到的是同步审核，多个自媒体用户同时发布文章后，发现每次点击发布文章到页面响应得到审核结果都需要好几秒时间。后来我研究后发现审核的过程其实是个耗时过程，就想到了用线程池技术异步来解决，这样审核效率问题就得到解决了。

比如之前文章精准发布不能做到很及时，每次的方案式定时扫描数据库的方案，这种方案每次扫描的文章数比较多对数据库造成了压力，我们扫描的频道也不能太快，扫描慢了，又发现文章最终发布的时间又远大于与设定的发布时间，不够精准。所有就设计了基于 Redis 的 ZSET 和 LIST 的精准发布方案，这样效率和实效性都得到了解决。

比如之前热点文章计算的结果都存入了数据库，页面查询文章列表太频道导致数据库压力很大，所以后面改成了存入 Redis 的技术方案，这样就提升了查询性能。

比如之前的行为数和评论数据都是直接操作数据库，发现用户多的时候，操作起来响应比较慢，数据库压力非常大，后来我就将行为数存入了 Redis 做好了持久化，将评论数据存入到了 MongoDB。最终效率就得到了大的提升。

5. 网关这块日志怎么处理的？

网关主要记录了用户请求的 IP、设备型号登录信息存储在 mongo 中，方便后期进行数据分析。

6. 项目中需要多少台服务器，如何计算的

项目用到的服务器数量具体多少，我们是一线开发人员不太清楚，是由开发负责人和运维负责人来决定的。由于我们是容器化部署。

8 .项目中线程池是一个服务一个还是整体使用一个，为什么

每个微服务用一个线程池，避免不通过业务模块的线程池彼此之间互相影响。不至于出现某个服务的线程池出现问题导致其他服务的线程池不可用。

7. 什么场景下用到了分布式事务，从事务开始到事务结束，经历了什么，结合项目进行讲解。

跨服务调用基本都会用到分布式事务，以文章自动审核业务为例，自动审核通过会修改自媒体微服务，还会调用文章微服务的 feign 接口创建 APP 文章，这个时候就涉及到跨服务调用。就需要使用分布式事务技术 Seata 保证这次审核业务对应事务的原子性和一致性，要么全部操作成功，要么就全部操作失败。

8. 如何设计一个秒杀系统；

秒杀的特点是瞬间高并发抢购商品。除了高并发的常规优化手段，秒杀还要防止商品超卖和防止重复下单问题，可以利用分布式锁和 Redis 查询双重校验。

9. miniIO 使用什么协议

默认是 https 协议

10.注册中心搭几个服务器？

双节点

11.文章篇幅的大小是多少

文章内容我们系统限制在 1 万字以内

12.项目如何设计日活是多少？实际日活是多少？

日活多少如何设计这个目前不太清楚，由于项目在早期运营阶段，实际目前的日活用户可能也就几千个用户

13.15 项目部署打包类型

项目最终的部署包都是 JAR 包形式

14.16 Git 使用中的注意事项

每次提交代码前先拉取最新代码，然后解决冲突；另外就是提交代码是 commit 描述信息一定要写的比较规范，描述明白是新功能、修复 BUG 还是进行代码优化。

15.18 文章数据什么时候放在 es 里？

在我们的业务里，文章会在审核通过后创建完 APP 文章后，将数据实时导入到 ES

16.开发改一些字段，怎么同步到你的数据库中

每位开发人员都是基于同一个开发测试数据库进行修改的，每个人都能改，每个人都能看见变化。不过修改数据库结构，我们有规范，必须把修改的哪部分内容以文档的方式提交到 GIT 说明文档里。

17.你们用的不是同一个数据库，你们的 sql 怎么同步

我们用的同一个开发数据库的，只是每个服务连接的 database 名称不一样而已，有变化都能看见。

18.搜索的业务流程是怎么样的

前端输入搜索关键词，请求到服务端，服务端根据搜索关键词进行多字段搜索，搜索时还要组合根据时间范围搜索目的是能上拉翻页；另外搜索结果我们会要求限制

每页查询 10 条，按照文章发布时间倒排序，还要求针对文章标题这个字段里出现的分词结果进行高亮。

19.分布式数据一致性(分布式事务)

分布式系统中，每个节点都能知道自己的事务操作是否成功，但是没法知道系统中的其他节点的事务是否成功。这就有可能会造成分布式系统中的各节点的状态出现不一致。

1. 阿里巴巴的seata
2. Seata-TCC模式
   a. 当事务的发起者和参与者都执行成功时,事务协调管理器(TC)调用每个微服务的Confirm()方法,否则,调用每个微服务的Cancel()方法对全局事务进行回滚

20.高并发如何解决

1.1. 服务器

增加服务器和配置(ssd 内存 cpu)

2.2. 部署

负载均衡 ngxin 网关 gateway

限流 网关 gateway

集群部署

java 微服务

基础服务

1.1. redis mysql 等

3.3. 代码

进入消息中间件 流量削峰 rabbitmq

用 redis 内存存储,增加速度

等等

4.4. mysql 优化

实践中如何优化 MySQL 顺序优化：

SQL 语句及索引的优化

数据库表结构的优化

1.1. 纵向分库,横向分表

数据库配置的优化

1.1. 选择正确的存储引擎。

innerdb 有事务,行锁 默认使用,可靠性更高

myLsam 没有事务,适合插入不频繁,查询频繁的

memory 内存存储,用的少

1.1. 集群部署

读写分离

硬件的优化

5.5. jvm 优化->最后手段,主要是降低 Gc 垃圾回收频率