# 面试题整理

## 一、Java基础

### 基础

#### 1. JVM、JRE和JDK的关系

> JVM
> Java Virtual Machine是Java虚拟机，Java程序需要运行在虚拟机上，不同的平台有自己的虚拟机，因此Java语言可以实现跨平台。
>
> JRE
> Java Runtime Environment包括Java虚拟机和Java程序所需的核心类库等。核心类库主要是java.lang包：包含了运行Java程序必不可少的系统类，如基本数据类型、基本数学函数、字符串处理、线程、异常处理类等，系统缺省加载这个包
>
> 如果想要运行一个开发好的Java程序，计算机中只需要安装JRE即可。
>
> JDK
> Java Development Kit是提供给Java开发人员使用的，其中包含了Java的开发工具，也包括了JRE。所以安装了JDK，就无需再单独安装JRE了。其中的开发工具：编译工具(javac.exe)，打包工具(jar.exe)等

#### 2. 什么是字节码？采用字节码的最大好处是什么

> **字节码**：Java源代码经过虚拟机编译器编译后产生的文件（即扩展为.class的文件），它不面向任何特定的处理器，只面向虚拟机。
>
> **采用字节码的好处**：
>
> Java语言通过字节码的方式，在一定程度上解决了传统解释型语言执行效率低的问题，同时又保留了解释型语言可移植的特点。所以Java程序运行时比较高效，而且，由于字节码并不专对一种特定的机器，因此，Java程序无须重新编译便可在多种不同的计算机上运行。

#### 3. 什么是值传递和引用传递?

> - 值传递，是对基本型变量而言的，传递的是该变量的一个副本，改变副本不影响原变量。
> - 引用传递，一般是对于对象型变量而言的，传递的是该对象地址的一个副本，并不是原对象本身。
>
>   一般认为，Java 内的传递都是值传递，Java 中实例对象的传递是引用传递。

#### 4. 用最有效率的方法计算 2 乘以 8

> 2 << 3（左移 3 位相当于乘以 2 的 3 次方，右移 3 位相当于除以 2 的 3 次方）

#### 5. 什么是字符串常量池?

> 字符串常量池位于堆内存中，专门用来存储字符串常量，可以提高内存的使用率，避免开辟多块空间存储相同的字符串，在创建字符串时 JVM 会首先检查字符串常量池，如果该字符串已经存在池中，则返回它的引用，如果不存在，则实例化一个字符串放到池中，并返回其引用

#### 6. String有哪些特性

> - 不变性：String 是只读字符串，是一个典型的 immutable 对象，对它进行任何操作，其实都是创建一个新的对象，再把引用指向该对象。不变模式的主要作用在于当一个对象需要被

多线程共享并频繁访问时，可以保证数据的一致性。
- 常量池优化：String 对象创建之后，会在字符串常量池中进行缓存，如果下次创建同样的对象时，会直接返回缓存的引用。
- final：使用 final 来定义 String 类，表示 String 类不能被继承，提高了系统的安全性。

**7. 自动装箱与拆箱**

> **装箱**：将基本类型用它们对应的引用类型包装起来；
>
> **拆箱**：将包装类型转换为基本数据类型；

## 进阶

**1. 在使用 HashMap 的时候，用 String 做 key 有什么好处?**

> HashMap 内部实现是通过 key 的 hashcode 来确定 value 的存储位置，因为字符串是不可变的，所以当创建字符串时，它的 hashcode 被缓存下来，不需要再次计算，所以相比于其他对象更快。

**2. String和StringBuffer、StringBuilder的区别是什么？String为什么是不可变的**

> **可变性**
>
> String类中使用字符数组保存字符串，private final char value[]，所以string对象是不可变的。StringBuilder与StringBuffer都继承自AbstractStringBuilder类，在AbstractStringBuilder中也是使用字符数组保存字符串，char[] value，这两种对象都是可变的。
>
> **线程安全性**
>
> String中的对象是不可变的，也就可以理解为常量，线程安全。AbstractStringBuilder是StringBuilder与StringBuffer的公共父类，定义了一些字符串的基本操作，如expandCapacity、append、insert、indexOf等公共方法。StringBuffer对方法加了同步锁或者对调用的方法加了同步锁，所以是线程安全的。StringBuilder并没有对方法进行加同步锁，所以是非线程安全的。
>
> **性能**
>
> 每次对String 类型进行改变的时候，都会生成一个新的String对象，然后将指针指向新的String 对象。StringBuffer每次都会对StringBuffer对象本身进行操作，而不是生成新的对象并改变对象引用。相同情况下使用StirngBuilder 相比使用StringBuffer 仅能获得10%~15% 左右的性能提升，但却要冒多线程不安全的风险。
>
> **对于三者使用的总结**
>
> 如果要操作少量的数据用 = 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

**3. error和exception有什么区别**

> error表示系统级的错误，是java运行环境内部错误或者硬件问题，不能指望程序来处理这样的问题，除了退出运行外别无选择，它是Java虚拟机抛出的。
>
> exception 表示程序需要捕捉、需要处理的异常，是由与程序设计的不完善而出现的问题，程序必须处理的问题

**4. throw和throws有什么区别?**

> throw关键字用来在程序中明确的抛出异常，相反，throws语句用来表明方法不能处理的异常。每一个方法都必须要指定哪些异常不能处理，所以方法的调用者才能够确保处理可能发生的异常，多个异常是用逗号分隔的。

**5. 异常处理的时候，finally代码块的重要性是什么?**

> 无论是否抛出异常，finally代码块总是会被执行。就算是没有catch语句同时又抛出异常的情况下，finally代码块仍然会被执行。最后要说的是，finally代码块主要用来释放资源，比如：I/O缓冲区，数据库连接。

**6. 是否可以继承 String 类**

> String 类是 final 类，不可以被继承。

**7. String str="i"与 String str=new String(“i”)一样吗?**

> 不一样，因为内存的分配方式不一样。String str="i"的方式，java 虚拟机会将其分配到常量池中；而 String str=new String(“i”) 则会被分到堆内存中。

**8. String s = new String(“xyz”);创建了几个字符串对象**

> 两个对象，一个是静态区的"xyz"，一个是用new创建在堆上的对象

# 二、多线程

## 基础

**1. 说一说自己对于 synchronized 关键字的了解**

> synchronized关键字解决的是多个线程访问共享资源的同步性问题，它可以保证被它修饰的代码块或者方法在任意时刻只能有一个线程执行
> 在早期的jdk版本重，synchronized是重量级锁，是操作系统级别的锁，效率非常低下。
> 在jdk1.6版本后，对synchronized引入了大量的优化，比如无锁、自旋锁、偏向锁、轻量级锁等技术来减少锁操作的开销。

**2. 说说自己是怎么使用 synchronized 关键字，在项目中用到了吗**

> synchronized一般用于修饰实例方法、静态方法、代码块。
>
> - 修饰实例方法的时候，作用于当前对象实例，进入同步代码块要先获取到当前实例对象的锁。
> - 修饰静态方法的时候，作用于当前类对象，进入同步代码块前要先获取当前类对象的锁。
> - 修饰代码块时，指定加锁对象，对给定对象加锁，进入代码块前要先获取给定对象的锁。
> - 双重检锁机制实现的单例模式中有用到,同时还有用到volatile

**3. 讲一下乐观锁和悲观锁的区别**

> 悲观锁，就是很悲观的认为每次拿数据的时候，数据都会被人修改，所以每次在拿数据的时候都会上锁，这样别人想拿数据就会阻塞，知道它拿到为止。
> 像synchronized和ReentranLock都是悲观锁
> 乐观锁，就是很乐观的认为，每次拿数据都不会被修改，所以不会上锁，只会在修改的时候判断一下，期间是否有其他人去修改，一般采用CAS或者版本号实现
>
> - 乐观锁的缺点有三：循环时间长开销大，只能保证一个共享变量的原子操作，会有ABA的问题

**4. 谈谈 synchronized和ReenTrantLock 的区别**

> 1、从实现层面来说，synchronized的操作系统级别的锁，而ReenTrantLock是程序级别是锁
> 2、是否可手动释放，synchronized不需要手动释放，代码执行完成后，系统会让线程自动释放对锁的占用，
> 而ReentrantLock需要手动释放（try catch finally 在finally中进行释放）
> 3、是否可中断；synchronized的不可中端，要么执行完成，要么出异常结束运行。
> reentrantlock可以中断，可以通过trylock设置超时的方法
> 4、是否公平锁，synchronized是非公平锁，reentranlock根据需要可以是公平锁，也可以是非公平锁
> 5、锁是否可以绑定条件，condition、 synchronized的不行，reentrantlock可以。并且可以通过await，singal进行精准唤醒，不像synchronized的，要么只能随机唤醒一个，要么只能唤醒所有
> 6、锁的对象，synchronized锁的是对象,锁是保存在对象头里的，根据对象头数据来标识是否有线程获得锁
> 而reentrantlock锁的是线程，，根据进入的线程和int类型的state标识锁的获得和争抢

**5. 为什么要用线程池?**

> - 复用线程，如果频繁的创建、销毁线程会消耗大量的系统资源
> - 统一对线程进行管理，可以统一分配、调优和监控
> - 控制线程的并发量，因为有最大线程、核心线程、阻塞队列的存在，如果没有线程池，一口气创建几万个线程有可能直接OOM了

**6. 实现Runnable接口和Callable接口的区别**

> 二者都能创建线程
>
> - 实现runnable创建的线程，是没有返回结果的
> - 实现callable创建的线程，是有返回结构的
> - runnable的run方法，必须在内部处理异常
> - callable的call方法支持在内部处理异常，也支持向上抛出异常

## 进阶

**1. 说说 synchronized 关键字和 volatile 关键字的区别**

> volatile的线程同步的轻量级实现，所以volatile性能肯定比synchronized的好
> 但是volatile只保证了可见性，保证不了原子性，而synchronized的都可以保证
> volatile只能修改变量，synchronized可以修饰实例方法、静态方法、代码块
> 多线程访问volatile不会发生阻塞，但是访问synchronized会发生阻塞
> volatile主要解决变量在多个线程间的可见性，而synchronized的主要解决多个线程访问资源的同步性

**2. 如何创建线程池**

> Executors里面有提供系统封装的线程池（有四种线程池），但是我们一般不用
> 我们一般通过new ThreadPoolExecutor来创建线程池
>
> - 七个参数
>   - 核心线程数
>   - 最大线程数
>   - 非核心线程空置存活时间
>   - 第三个参数的时间单位
>   - 阻塞队列
>   - 线程工厂
>   - 拒绝策略

**3. 介绍一下Atomic 原子类**

> - 有四种类型
>   - 基本类型 AtomicInteger...
>   - 数组类型 AtomicIntegerArray...
>   - 引用类型 AtomicReference...
>   - 对象的属性修改类型 AtomicIntegerFieldUpdateer...
> - 在juc包下
> - 并发执行可以保证原子性，是线程安全的
> - 不需要加锁
> - 通过cas来保证原子性，通过volatile来保证可见性

**4. 讲讲 AtomicInteger 的使用**

> 并发执行可以保证原子性，是线程安全的
> 不需要加锁
> 通过cas来保证原子性，通过volatile来保证可见性
> 修改极其频繁的情况下，可能会发生很多空旋。

**5. 能不能给我简单介绍一下 AtomicInteger 类的原理**

> 对变量是用volatile进行修饰
> 直接用本地方法操作内存偏移量来
> 使用了cas来保证原子性，用到了unsafe类，大量使用本地方法
> 避免了使用synchronized的高开销
>
> 具体实现是更新前先比较期望值与目前实际值，如果结果发生了变化表示有其他线程进行了修改，于是会重新取内存中的值进行新一轮的cas判断，直至成功为止
> -- 具体阐述下cas的原理

**6. CAS的分段优化**

> 内部拆分多个数组，进行累加或者减的操作时，分散到不同数组上进行处理。
>
> 获取结果时，将各个数组的值累加进行返回。可以减少CAS的空旋。
>
> 例如：LongAdder

# 三、JVM

## 基础

**1. 解释 Java 堆空间及 GC?**

> 当通过 Java 命令启动 Java 进程的时候，会为它分配内存。内存的一部分用于创建堆空间，当程序中创建对象的时候，就从对空间中分配内存。GC 是 JVM 内部的一个进程，回收无效对象的内存用于将来的分配。

**2. Java 中堆和栈有什么区别?**

> JVM 中堆和栈属于不同的内存区域，使用目的也不同。栈常用于保存方法帧和局部变量，而对象总是在堆上分配。栈通常都比堆小，也不会在多个线程之间共享，而堆被整个 JVM 的所有线程共享。

**3. 队列和栈是什么？有什么区别?**

> 队列和栈都是被用来预存储数据的。

- 操作的名称不同。队列的插入称为入队，队列的删除称为出队。栈的插入称为进栈，栈的删除称为出栈。
- 可操作的方式不同。队列是在队尾入队，队头出队，即两边都可操作。而栈的进栈和出栈都是在栈顶进行的，无法对栈底直接进行操作。
- 操作的方法不同。队列是先进先出（FIFO），即队列的修改是依先进先出的原则进行的。新来的成员总是加入队尾（不能从中间插入），每次离开的成员总是队列头上（不允许中途离队）。而栈为后进先出（LIFO）,即每次删除（出栈）的总是当前栈中最新的元素，即最后插入（进栈）的元素，而最先插入的被放在栈的底部，要到最后才能删除。

**4. JRE、JDK、JVM 及 JIT 之间有什么不同?**

JRE 代表 Java 运行时（Java run-time），是运行 Java 引用所必须的。

JDK 代表 Java 开发工具（Java development kit），是 Java 程序的开发工具，如 Java编译器，它也包含 JRE。

JVM 代表 Java 虚拟机（Java virtual machine），它的责任是运行 Java 应用。

JIT 代表即时编译（Just In Time compilation），当代码执行的次数超过一定的阈值时，会将 Java 字节码转换为本地代码，如，主要的热点代码会被准换为本地代码，这样有利大幅度提高 Java 应用的性能。

**5.深拷贝和浅拷贝的区别?**

浅拷贝（shallowCopy）只是增加了一个指针指向已存在的内存地址，

深拷贝（deepCopy）是增加了一个指针并且申请了一个新的内存，使这个增加的指针指向这个新的内存，

使用深拷贝的情况下，释放内存的时候不会因为出现浅拷贝时释放同一个内存的错误。

浅复制：仅仅是指向被复制的内存地址，如果原地址发生改变，那么浅复制出来的对象也会相应的改变。

深复制：在计算机中开辟一块**新的内存地址**用于存放复制的对象。

**6. 说一下JVM内存模型?**

**思路：** 给面试官画一下JVM内存模型图，并描述每个模块的定义，作用

- 程序计数器（Program Counter Register）：当前线程所执行的字节码的行号指示器，字节码解析器的工作是通过改变这个计数器的值，来选取下一条需要执行的字节码指令，分支、循环、跳转、异常处理、线程恢复等基础功能，都需要依赖这个计数器来完成；
- Java 虚拟机栈（Java Virtual Machine Stacks）：用于存储局部变量表、操作数栈、动态链接、方法出口等信息；
- 本地方法栈（Native Method Stack）：与虚拟机栈的作用是一样的，只不过虚拟机栈是服务 Java 方法的，而本地方法栈是为虚拟机调用 Native 方法服务的；
- Java 堆（Java Heap）：Java 虚拟机中内存最大的一块，是被所有线程共享的，几乎所有的对象实例都在这里分配内存；
- 方法区（Methed Area）：用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译后的代码等数据。

## 进阶

**1. 简述Java垃圾回收机制**

> 在java中，程序员是不需要显示的去释放一个对象的内存的，而是由虚拟机自行执行。在JVM中，有一个垃圾回收线程，它是低优先级的，在正常情况下是不会执行的，只有在虚拟机空闲或者当前堆内存不足时，才会触发执行，扫面那些没有被任何引用的对象，并将它们添加到要回收的集合中，进行回收。

**2. GC是什么？为什么要GC**

> GC 是垃圾收集的意思（Gabage Collection）,内存处理是编程人员容易出现问题的地方，忘记或者错误的内存回收会导致程序或系统的不稳定甚至崩溃，Java 提供的 GC 功能可以自动监测对象是否超过作用域从而达到自动回收内存的目的，Java 语言没有提供释放已分配内存的显示操作方法。

**3. 垃圾回收器的基本原理是什么？垃圾回收器可以马上回收内存吗？有什么办法主动通知虚拟机进行垃圾回收？**

> 对于GC来说，当程序员创建对象时，GC就开始监控这个对象的地址、大小以及使用情况。
>
> 通常，GC采用有向图的方式记录和管理堆(heap)中的所有对象。通过这种方式确定哪些对象是"可达的"，哪些对象是"不可达的"。当GC确定一些对象为"不可达"时，GC就有责任回收这些内存空间。
>
> 可以。程序员可以手动执行System.gc()，通知GC运行，但是Java语言规范并不保证GC一定会执行。

**4. 怎么判断对象是否可以被回收？**

> 垃圾收集器在做垃圾回收的时候，首先需要判定的就是哪些内存是需要被回收的，哪些对象是「存活」的，是不可以被回收的；哪些对象已经「死掉」了，需要被回收。
>
> 一般有两种方法来判断：
>
> - 引用计数器法：为每个对象创建一个引用计数，有对象引用时计数器 +1，引用被释放时计数 -1，当计数器为 0 时就可以被回收。它有一个缺点不能解决循环引用的问题；
> - 可达性分析算法：从 GC Roots 开始向下搜索，搜索所走过的路径称为引用链。当一个对象到 GC Roots 没有任何引用链相连时，则证明此对象是可以被回收的。
>
> JVM中使用的是可达性分析算法。

**5. JVM内存为什么要分成新生代，老年代，持久代。新生代中为什么要分为Eden和Survivor。**

> 这样划分的目的是为了使 JVM 能够更好的管理堆内存中的对象，包括内存的分配以及回收。
>
> 1）共享内存区划分
>
> - 共享内存区 = 持久带 + 堆
> - 持久带 = 方法区 + 其他
> - Java堆 = 老年代 + 新生代
> - 新生代 = Eden + S0 + S1
>
> 2）一些参数的配置
>
> - 默认的，新生代 ( Young ) 与老年代 ( Old ) 的比例的值为 1:2 ，可以通过参数 –XX:NewRatio 配置。
> - 默认的，Eden : from : to = 8 : 1 : 1 ( 可以通过参数 –XX:SurvivorRatio 来设定)
> - Survivor区中的对象被复制次数为15(对应虚拟机参数 -XX:+MaxTenuringThreshold)
>
> 3)为什么要分为Eden和Survivor?为什么要设置两个Survivor区?
>
> - 如果没有Survivor，Eden区每进行一次Minor GC，存活的对象就会被送到老年代。老年代很快被填满，触发Major GC.老年代的内存空间远大于新生代，进行一次Full GC消耗的时间

比Minor GC长得多,所以需要分为Eden和Survivor。
- Survivor的存在意义，就是减少被送到老年代的对象，进而减少Full GC的发生，Survivor的预筛选保证，只有经历16次Minor GC还能在新生代中存活的对象，才会被送到老年代。
- 设置两个Survivor区最大的好处就是解决了碎片化，刚刚新建的对象在Eden中，经历一次Minor GC，Eden中的存活对象就会被移动到第一块survivor space S0，Eden被清空；等Eden区再满了，就再触发一次Minor GC，Eden和S0中的存活对象又会被复制送入第二块survivor space S1（这个过程非常重要，因为这种复制算法保证了S1中来自S0和Eden两部分的存活对象占用连续的内存空间，避免了碎片化的发生）

**6. JVM中一次完整的GC流程是怎样的，对象如何晋升到老年代**

**思路：** 先描述一下Java堆内存划分，再解释Minor GC，Major GC，full GC，描述它们之间转化流程。

**我的答案：**

- Java堆 = 老年代 + 新生代
- 新生代 = Eden + S0 + S1
- 当 Eden 区的空间满了， Java虚拟机会触发一次 Minor GC，以收集新生代的垃圾，存活下来的对象，则会转移到 Survivor区。
- **大对象**（需要大量连续内存空间的Java对象，如那种很长的字符串）**直接进入老年态**；
- 如果对象在Eden出生，并经过第一次Minor GC后仍然存活，并且被Survivor容纳的话，年龄设为1，每熬过一次Minor GC，年龄+1，**若年龄超过一定限制（15），则被晋升到老年态。**即**长期存活的对象进入老年态。**
- 老年代满了而**无法容纳更多的对象**，Minor GC 之后通常就会进行Full GC，Full GC 清理整个内存堆 – **包括年轻代和年老代。**
- Major GC **发生在老年代的GC**，**清理老年区**，经常会伴随至少一次Minor GC，**比Minor GC慢10倍以上。**

**7. JVM中的永久代中会发生垃圾回收吗**

垃圾回收不会发生在永久代，如果永久代满了或者是超过了临界值，会触发完全垃圾回收(Full GC)。如果你仔细查看垃圾收集器的输出信息，就会发现永久代也是被回收的。这就是为什么正确的永久代大小对避免Full GC是非常重要的原因。(Java8中已经移除了永久代，新加了一个叫做元数据区的native内存区)

**8. Minor GC与Full GC分别在什么时候发生?**

新生代内存不够用时候发生MGC也叫YGC，JVM内存不够的时候发生FGC

**9. 垃圾收集算法有哪些类型?**

GC最基础的算法有三类： 标记 -清除算法、复制算法、标记-压缩算法，我们常用的垃圾回收器一般都采用分代收集算法。

标记 -清除算法，“标记-清除”（Mark-Sweep）算法，如它的名字一样，算法分为“标记”和“清除”两个阶段：首先标记出所有需要回收的对象，在标记完成后统一回收掉所有被标记的对象。

复制算法，“复制”（Copying）的收集算法，它将可用内存按容量划分为大小相等的两块，每次只使用其中的一块。当这一块的内存用完了，就将还存活着的对象复制到另外一块上面，然后再把已使用过的内存空间一次清理掉。

标记-压缩算法，标记过程仍然与“标记-清除”算法一样，但后续步骤不是直接对可回收对象进行清理，而是让所有存活的对象都向一端移动，然后直接清理掉端边界以外的内存

分代收集算法，“分代收集”（Generational Collection）算法，把Java堆分为新生代和老年代，这样就可以根据各个年代的特点采用最适当的收集算法

**10. 什么是类加载器，类加载器有哪些?**

> 实现通过类的权限定名获取该类的二进制字节流的代码块叫做类加载器。
>
> 主要有以下四种类加载器:
>
> 1. 启动类加载器(Bootstrap ClassLoader)用来加载java核心类库，无法被java程序直接引用。
> 2. 扩展类加载器(extensions class loader):它用来加载 Java 的扩展库。Java 虚拟机的实现会提供一个扩展库目录。该类加载器在此目录里面查找并加载 Java 类。
> 3. 系统类加载器（system class loader）：它根据 Java 应用的类路径（CLASSPATH）来加载 Java 类。一般来说，Java 应用的类都是由它来完成加载的。可以通过 ClassLoader.getSystemClassLoader()来获取它。
> 4. 用户自定义类加载器，通过继承 java.lang.ClassLoader类的方式实现。

**11. 什么是双亲委派模型?**

> 双亲委派模型：如果一个类加载器收到了类加载的请求，它首先不会自己去加载这个类，而是把这个请求委派给父类加载器去完成，每一层的类加载器都是如此，这样所有的加载请求都会被传送到顶层的启动类加载器中，只有当父加载无法完成加载请求（它的搜索范围中没找到所需的类）时，子加载器才会尝试去加载类。
>
> 当一个类收到了类加载请求时，不会自己先去加载这个类，而是将其委派给父类，由父类去加载，如果此时父类不能加载，反馈给子类，由子类去完成类的加载。

## 高级

**1. 你知道哪几种垃圾收集器，各自的优缺点，重点讲下cms和G1，包括原理，流程，优缺点。**

> **思路：** 一定要记住典型的垃圾收集器，尤其cms和G1，它们的原理与区别，涉及的垃圾回收算法。
>
> **参考答案：**
>
> 1）几种垃圾收集器：
>
> - **Serial收集器：** 单线程的收集器，收集垃圾时，必须stop the world，使用复制算法。
> - **ParNew收集器：** Serial收集器的多线程版本，也需要stop the world，复制算法。
> - **Parallel Scavenge收集器：** 新生代收集器，复制算法的收集器，并发的多线程收集器，目标是达到一个可控的吞吐量。如果虚拟机总共运行100分钟，其中垃圾花掉1分钟，吞吐量就是99%。
> - **Serial Old收集器：** 是Serial收集器的老年代版本，单线程收集器，使用标记整理算法。
> - **Parallel Old收集器：** 是Parallel Scavenge收集器的老年代版本，使用多线程，标记-整理算法。
> - **CMS(Concurrent Mark Sweep) 收集器：** 是一种以获得最短回收停顿时间为目标的收集器，**标记清除算法，运作过程：初始标记，并发标记，重新标记，并发清除**，收集结束会产生大量空间碎片。
> - **G1收集器：** 标记整理算法实现，**运作流程主要包括以下：初始标记，并发标记，最终标记，筛选标记。**不会产生空间碎片，可以精确地控制停顿。
>
> 2）CMS收集器和G1收集器的区别：
>
> - CMS收集器是老年代的收集器，可以配合新生代的Serial和ParNew收集器一起使用；
> - G1收集器收集范围是老年代和新生代，不需要结合其他收集器使用；
> - CMS收集器以最小的停顿时间为目标的收集器；
> - G1收集器可预测垃圾回收的停顿时间
> - CMS收集器是使用“标记-清除”算法进行的垃圾回收，容易产生内存碎片

- G1收集器使用的是“标记-整理”算法，进行了空间整合，降低了内存空间碎片。

**2. 简述分代垃圾回收器是怎么工作的?**

分代回收器有两个分区：老生代和新生代，新生代默认的空间占比总空间的 1/3，老生代的默认占比是 2/3。

新生代使用的是复制算法，新生代里有 3 个分区：Eden、To Survivor、From Survivor，它们的默认占比是 8:1:1，它的执行流程如下：

- 把 Eden + From Survivor 存活的对象放入 To Survivor 区；
- 清空 Eden 和 From Survivor 分区；
- From Survivor 和 To Survivor 分区交换，From Survivor 变 To Survivor，To Survivor 变 From Survivor。

每次在 From Survivor 到 To Survivor 移动时都存活的对象，年龄就 +1，当年龄到达 15（默认配置是 15）时，升级为老生代。大对象也会直接进入老生代。

老生代当空间占用到达某个值之后就会触发全局垃圾收回，一般使用标记整理的执行算法。以上这些循环往复就构成了整个分代垃圾回收的整体执行流程。

**3. 简述java内存分配与回收策率以及Minor GC和Major GC**

对象的内存分配通常是在 Java 堆上分配（随着虚拟机优化技术的诞生，某些场景下也会在栈上分配，后面会详细介绍），对象主要分配在新生代的 Eden 区，如果启动了本地线程缓冲，将按照线程优先在 TLAB 上分配。少数情况下也会直接在老年代上分配。总的来说分配规则不是百分百固定的，其细节取决于哪一种垃圾收集器组合以及虚拟机相关参数有关，但是虚拟机对于内存的分配还是会遵循以下几种「普世」规则：

## 对象优先在 Eden 区分配

多数情况，对象都在新生代 Eden 区分配。当 Eden 区分配没有足够的空间进行分配时，虚拟机将会发起一次 Minor GC。如果本次 GC 后还是没有足够的空间，则将启用分配担保机制在老年代中分配内存。

这里我们提到 Minor GC，如果你仔细观察过 GC 日常，通常我们还能从日志中发现 Major GC/Full GC。

- **Minor GC** 是指发生在新生代的 GC，因为 Java 对象大多都是朝生夕死，所有 Minor GC 非常频繁，一般回收速度也非常快；
- **Major GC/Full GC** 是指发生在老年代的 GC，出现了 Major GC 通常会伴随至少一次 Minor GC。Major GC 的速度通常会比 Minor GC 慢 10 倍以上。

## 大对象直接进入老年代

所谓大对象是指需要大量连续内存空间的对象，频繁出现大对象是致命的，会导致在内存还有不少空间的情况下提前触发 GC 以获取足够的连续空间来安置新对象。

前面我们介绍过新生代使用的是标记-清除算法来处理垃圾回收的，如果大对象直接在新生代分配就会导致 Eden 区和两个 Survivor 区之间发生大量的内存复制。因此对于大对象都会直接在老年代进行分配。

## 长期存活对象将进入老年代

虚拟机采用分代收集的思想来管理内存，那么内存回收时就必须判断哪些对象应该放在新生代，哪些对象应该放在老年代。因此虚拟机给每个对象定义了一个对象年龄的计数器，如果对象在 Eden 区出生，并且能够被 Survivor 容纳，将被移动到 Survivor 空间中，这时设置对象年龄为 1。对象在 Survivor 区中每「熬过」一次 Minor GC 年龄就加 1，当年龄达到一定程度（默认 15） 就会被晋升到老年代。

# 四、Spring

## 基础

### 1. 什么是Spring IOC 容器?

> 控制反转即IoC (Inversion of Control)，它把传统上由程序代码直接操控的对象的调用权交给容器，通过容器来实现对象组件的装配和管理。所谓的"控制反转"概念就是对组件对象控制权的转移，从程序代码本身转移到了外部容器。
>
> 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

### 2. 控制反转(IoC)有什么作用

> - 管理对象的创建和依赖关系的维护。对象的创建并不是一件简单的事，在对象关系比较复杂时，如果依赖关系需要程序猿来维护的话，那是相当头疼的
> - 解耦，由容器去维护具体的对象
> - 托管了类的产生过程，比如我们需要在类的产生过程中做一些处理，最直接的例子就是代理，如果有容器程序可以把这部分处理交给容器，应用程序则无需去关心类是如何完成代理的

### 3. IOC的优点是什么?

> - 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。

### 4. 解释Spring支持的几种bean的作用域

> Spring框架支持以下五种bean的作用域：
>
> - **singleton :** bean在每个Spring ioc 容器中只有一个实例。
> - **prototype**：一个bean的定义可以有多个实例。
> - **request**：每次http请求都会创建一个bean，该作用域仅在基于web的Spring ApplicationContext情形下有效。
> - **session**：在一个HTTP Session中，一个bean定义对应一个实例。该作用域仅在基于web的Spring ApplicationContext情形下有效。
> - **global-session**：在一个全局的HTTP Session中，一个bean定义对应一个实例。该作用域仅在基于web的Spring ApplicationContext情形下有效。
>
> **注意：** 缺省的Spring bean 的作用域是Singleton。使用 prototype 作用域需要慎重的思考，因为频繁创建和销毁 bean 会带来很大的性能开销。

### 5. 三种实现指定初始化方法的方法：

> - 使用@PostConstruct注解，该注解作用于void方法上
> - 在配置文件中配置init-method方法
> - 将类实现InitializingBean接口，实现afterPropertiesSet()方法

### 6. @Component, @Controller, @Repository, @Service 有何区别?

> @Component：这将 java 类标记为 bean。它是任何 Spring 管理组件的通用构造型。spring 的组件扫描机制现在可以将其拾取并将其拉入应用程序环境中。

> @Controller：这将一个类标记为 Spring Web MVC 控制器。标有它的 Bean 会自动导入到 IoC 容器中。
>
> @Service：此注解是组件注解的特化。它不会对 @Component 注解提供任何其他行为。您可以在服务层类中使用 @Service 而不是 @Component，因为它以更好的方式指定了意图。
>
> @Repository：这个注解是具有类似用途和功能的 @Component 注解的特化。它为 DAO 提供了额外的好处。它将 DAO 导入 IoC 容器，并使未经检查的异常有资格转换为 Spring DataAccessException。

**7. @Autowired 注解有什么作用**

> @Autowired默认是按照类型装配注入的，默认情况下它要求依赖对象必须存在（可以设置它required属性为false）。
>
> @Autowired 注解提供了更细粒度的控制，包括在何处以及如何完成自动装配。它的用法和@Required一样，修饰setter方法、构造器、属性或者具有任意名称和/或多个参数的PN方法。

**8. @Autowired和@Resource之间的区别**

> @Autowired可用于：构造函数、成员变量、Setter方法
>
> @Autowired和@Resource之间的区别
>
> - @Autowired默认是按照类型装配注入的，默认情况下它要求依赖对象必须存在（可以设置它required属性为false）。
> - @Resource默认是按照名称来装配注入的，只有当找不到与名称匹配的bean才会按照类型来装配注入。

**9. @RequestMapping 注解有什么用?**

> @RequestMapping 注解用于将特定 HTTP 请求方法映射到将处理相应请求的控制器中的特定类/方法。此注释可应用于两个级别：
>
> - 类级别：映射请求的 URL
> - 方法级别：映射 URL 以及 HTTP 请求方法

**10. 什么是AOP**

> AOP(Aspect-Oriented Programming)，一般称为面向切面编程，作为面向对象的一种补充，用于将那些与业务无关，但却对多个对象产生影响的公共行为和逻辑，抽取并封装为一个可重用的模块，这个模块被命名为“切面”（Aspect），减少系统中的重复代码，降低了模块间的耦合度，同时提高了系统的可维护性。可用于权限认证、日志、事务处理等。

## 进阶

**1. Spring IoC 的实现机制**

> Spring 中的 IoC 的实现原理就是工厂模式加反射机制。

**2. ApplicationContext通常的实现是什么?**

> **FileSystemXmlApplicationContext**：此容器从一个XML文件中加载beans的定义，XML Bean配置文件的全路径名必须提供给它的构造函数。
>
> **ClassPathXmlApplicationContext**：此容器也从一个XML文件中加载beans的定义，这里，你需要正确设置classpath因为这个容器将在classpath里找bean配置。

> **WebXmlApplicationContext**：此容器加载一个XML文件，此文件定义了一个WEB应用的所有bean。

**3. Spring框架中的单例bean是线程安全的吗?**

> 不是，Spring框架中的单例bean不是线程安全的。
>
> spring 中的 bean 默认是单例模式，spring 框架并没有对单例 bean 进行多线程的封装处理。
>
> 实际上大部分时候 spring bean 无状态的（比如 dao 类），所有某种程度上来说 bean 也是安全的，但如果 bean 有状态的话（比如 view model 对象），那就要开发者自己去保证线程安全了，最简单的就是改变 bean 的作用域，把“singleton”变更为“prototype”，这样请求 bean 相当于 new Bean()了，所以就可以保证线程安全了。
>
> - 有状态就是有数据存储功能。
> - 无状态就是不会保存数据。

**4. 解释Spring框架中bean的生命周期**

> 只有四个阶段。实例化和属性赋值对应构造方法和setter方法的注入，初始化和销毁是用户能自定义扩展的两个阶段。在这四步之间穿插的各种扩展点，稍后会讲。
>
> 1. 实例化 Instantiation
> 2. 属性赋值 Populate
> 3. 初始化 Initialization
> 4. 销毁 Destruction
>
> **实例化 -> 属性赋值 -> 初始化 -> 销毁**

**5. Spring支持的事务管理类型， spring 事务实现方式有哪些?**

> Spring支持两种类型的事务管理：
>
> **编程式事务管理**：这意味你通过编程的方式管理事务，给你带来极大的灵活性，但是难维护。
>
> **声明式事务管理**：这意味着你可以将业务代码和事务管理分离，你只需用注解和XML配置来管理事务。

**6. Spring事务的实现方式和实现原理**

> Spring事务的本质其实就是数据库对事务的支持，没有数据库的事务支持，spring是无法提供事务功能的。真正的数据库层的事务提交和回滚是通过binlog或者redo log实现的

**7. 说一下Spring的事务传播行为**

> spring事务的传播行为说的是，当多个事务同时存在的时候，spring如何处理这些事务的行为。
>
> > ① PROPAGATION_REQUIRED：如果当前没有事务，就创建一个新事务，如果当前存在事务，就加入该事务，该设置是最常用的设置。
> >
> > ② PROPAGATION_SUPPORTS：支持当前事务，如果当前存在事务，就加入该事务，如果当前不存在事务，就以非事务执行。
> >
> > ③ PROPAGATION_MANDATORY：支持当前事务，如果当前存在事务，就加入该事务，如果当前不存在事务，就抛出异常。
> >
> > ④ PROPAGATION_REQUIRES_NEW：创建新事务，无论当前存不存在事务，都创建新事务。
> >
> > ⑤ PROPAGATION_NOT_SUPPORTED：以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。
> >
> > ⑥ PROPAGATION_NEVER：以非事务方式执行，如果当前存在事务，则抛出异常。

> ⑦ PROPAGATION_NESTED：如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则按REQUIRED属性执行。

**8. 请描述Spring MVC的工作流程？描述一下 DispatcherServlet 的工作流程?**

> （1）用户发送请求至前端控制器DispatcherServlet；
> （2） DispatcherServlet收到请求后，调用HandlerMapping处理器映射器，请求获取Handle；
> （3）处理器映射器根据请求url找到具体的处理器，生成处理器对象及处理器拦截器(如果有则生成)一并返回给DispatcherServlet；
> （4）DispatcherServlet 调用 HandlerAdapter处理器适配器；
> （5）HandlerAdapter 经过适配调用 具体处理器(Handler，也叫后端控制器)；
> （6）Handler执行完成返回ModelAndView；
> （7）HandlerAdapter将Handler执行结果ModelAndView返回给DispatcherServlet；
> （8）DispatcherServlet将ModelAndView传给ViewResolver视图解析器进行解析；
> （9）ViewResolver解析后返回具体View；
> （10）DispatcherServlet对View进行渲染视图（即将模型数据填充至视图中）
> （11）DispatcherServlet响应用户。

**9. 注解原理是什么**

> 注解本质是一个继承了Annotation的特殊接口，其具体实现类是Java运行时生成的动态代理类。我们通过反射获取注解时，返回的是Java运行时生成的动态代理对象。通过代理对象调用自定义注解的方法，会最终调用AnnotationInvocationHandler的invoke方法。该方法会从memberValues这个Map中索引出对应的值。而memberValues的来源是Java常量池。

## 高级

**1. BeanFactory 和 ApplicationContext有什么区别?**

> BeanFactory和ApplicationContext是Spring的两大核心接口，都可以当做Spring的容器。
>
> 其中ApplicationContext是BeanFactory的子接口。
>
> ### 依赖关系
>
> BeanFactory：是Spring里面最底层的接口，包含了各种Bean的定义，读取bean配置文档，管理bean的加载、实例化，控制bean的生命周期，维护bean之间的依赖关系。
>
> ApplicationContext接口作为BeanFactory的派生，除了提供BeanFactory所具有的功能外，还提供了更完整的框架功能：
>
> - 继承MessageSource，因此支持国际化。
> - 统一的资源文件访问方式。
> - 提供在监听器中注册bean的事件。
> - 同时加载多个配置文件。
> - 载入多个（有继承关系）上下文 ，使得每一个上下文都专注于一个特定的层次，比如应用的web层。
>
> ### 加载方式
>
> BeanFactroy采用的是延迟加载形式来注入Bean的，即只有在使用到某个Bean时(调用getBean())，才对该Bean进行加载实例化。这样，我们就不能发现一些存在的Spring的配置问题。如果Bean的某一个属性没有注入，BeanFacotry加载后，直至第一次使用调用getBean方法才会抛出异常。

ApplicationContext，它是在容器启动时，一次性创建了所有的Bean。这样，在容器启动时，我们就可以发现Spring中存在的配置错误，这样有利于检查所依赖属性是否注入。
ApplicationContext启动后预载入所有的单实例Bean，通过预载入单实例bean ,确保当你需要的时候，你就不用等待，因为它们已经创建好了。

相对于基本的BeanFactory，ApplicationContext 唯一的不足是占用内存空间。当应用程序配置Bean较多时，程序启动较慢。

### 创建方式

BeanFactory通常以编程的方式被创建，ApplicationContext还能以声明的方式创建，如使用ContextLoader。

### 注册方式

BeanFactory和ApplicationContext都支持BeanPostProcessor、BeanFactoryPostProcessor的使用，但两者之间的区别是：BeanFactory需要手动注册，而ApplicationContext则是自动注册。

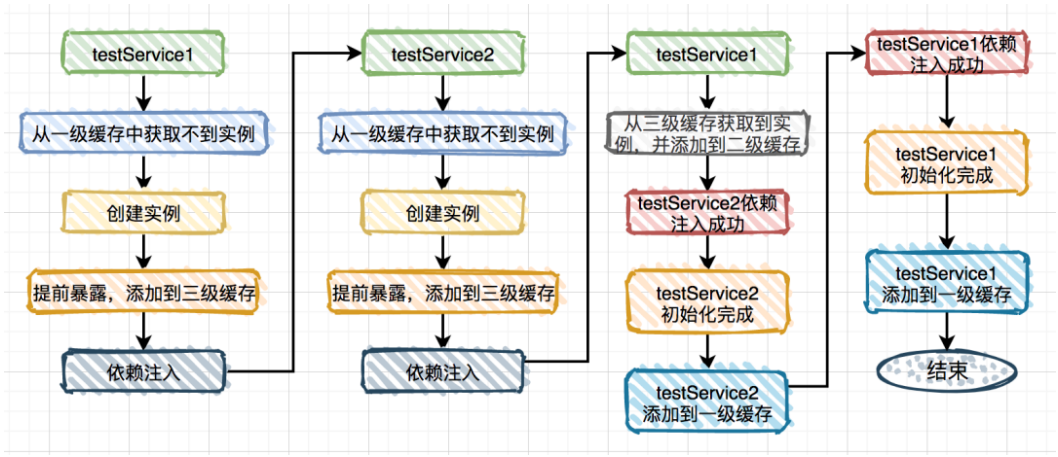
**2. Spring 如何解决循环依赖?**

循环依赖：说白是一个或多个对象实例之间存在直接或间接的依赖关系，这种依赖关系构成了构成一个环形调用。

spring通过三级缓存来解决循环依赖

spring内部有三级缓存：

- singletonObjects 一级缓存，用于保存实例化、注入、初始化完成的bean实例
- earlySingletonObjects 二级缓存，用于保存实例化完成的bean实例
- singletonFactories 三级缓存，用于保存bean创建工厂，以便于后面扩展有机会创建代理对象。
-




**3. JDK动态代理和CGLIB动态代理的区别**

Spring AOP中的动态代理主要有两种方式，JDK动态代理和CGLIB动态代理：

- JDK动态代理只提供接口的代理，不支持类的代理。核心InvocationHandler接口和Proxy类，InvocationHandler 通过invoke()方法反射来调用目标类中的代码，动态地将横切逻辑和业务编织在一起；接着，Proxy利用 InvocationHandler动态创建一个符合某一接口的的实例, 生成目标类的代理对象。
- 如果代理类没有实现 InvocationHandler 接口，那么Spring AOP会选择使用CGLIB来动态代理目标类。CGLIB（Code Generation Library），是一个代码生成的类库，可以在运行时动态的生成指定类的一个子类对象，并覆盖其中特定方法并添加增强代码，从而实现AOP。CGLIB是通过继承的方式做的动态代理，因此如果某个类被标记为final，那么它是无法使用CGLIB做动态代理的。

> 静态代理与动态代理区别在于生成AOP代理对象的时机不同，相对来说AspectJ的静态代理方式具有更好的性能，但是AspectJ需要特定的编译器进行处理，而Spring AOP则无需特定的编译器处理。

# 五、Spring Boot

## 基础

### 1. 什么是 Spring Boot?

> 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。

### 2. Spring Boot、Spring MVC 和 Spring 有什么区别?

> **SpringFrame**
>
> > SpringFramework 最重要的特征是依赖注入。所有 SpringModules 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，我们可以开发松耦合应用。松耦合应用的单元测试可以很容易的进行。
>
> **SpringMVC**
>
> > Spring MVC 提供了一种分离式的方法来开发 Web 应用。通过运用像 DispatcherServelet，MoudlAndView 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 SpringMVC 的问题在于：需要配置大量的参数。
>
> Spring Boot 通过一个自动配置和启动的项来目解决这个问题。为了更快的构建产品就绪应用程序，Spring Boot 提供了一些非功能性特征。

### 3. Spring Boot 有哪些优点?

> Spring Boot 主要有如下优点：
>
> 1. 容易上手，提升开发效率，为 Spring 开发提供一个更快、更广泛的入门体验。
> 2. 开箱即用，远离繁琐的配置。
> 3. 提供了一系列大型项目通用的非业务性功能，例如：内嵌服务器、安全管理、运行数据监控、运行状况检查和外部化配置等。
> 4. 没有代码生成，也不需要XML配置。
> 5. 避免大量的 Maven 导入和各种版本冲突。

### 4. Spring Boot 的核心注解是哪个？它主要由哪几个注解组成的?

> 启动类上面的注解是@SpringBootApplication，它也是 Spring Boot 的核心注解，主要组合包含了以下 3 个注解：
>
> @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： @SpringBootApplication(exclude = { DataSourceAutoConfiguration.class })。
>
> @ComponentScan：Spring组件扫描。

**5. 如何在自定义端口上运行 Spring Boot 应用程序?**

> 可以在application.properties 中指定端口。
>
> server.port = 8090

## 进阶

**1. Spring Boot 自动配置原理是什么?**

> 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。
>
> 筛选有效的自动配置类。
>
> 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

**2. 如何理解 Spring Boot 配置加载顺序?**

> 在 Spring Boot 里面，可以使用以下几种方式来加载配置。
>
> 1）properties文件；
>
> 2）YAML文件；
>
> 3）系统环境变量；
>
> 4）命令行参数；

**3. spring boot 核心配置文件是什么？bootstrap.properties 和 application.properties 有何区别？**

> - 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 Spring Cloud 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 spring boot 项目的自动化配置。

**4. Spring Boot 中如何解决跨域问题？**

> - 通过实现WebMvcConfigurer接口然后重写addCorsMappings方法解决跨域问题

**5. 微服务中如何实现 session 共享？**

> 在微服务中，一个完整的项目被拆分成多个不相同的独立的服务，各个服务独立部署在不同的服务器上，各自的 session 被从物理空间上隔离开了，但是经常，我们需要在不同微服务之间共享 session ，常见的方案就是 Spring Session + Redis 来实现 session 共享。将所有微服务的 session 统一保存在 Redis 上，当各个微服务对 session 有相关的读写操作时，都去操作 Redis 上的 session 。这样就实现了 session 共享，Spring Session 基于 Spring 中的代理过滤器实现，使得 session 的同步操作对开发人员而言是透明的，非常简便。

# 六、Spring Cloud

## 基础

**1. 什么是Spring Cloud**

> Spring Cloud是一系列框架的有序集合， 它利用Spring Boot的开发便利性巧妙地简化了分布式系统基础设施的开发。Spring Cloud并没有重复制造轮子，它只是将各家公司开发的比较成熟、经得起实际考验的服务框架组合起来，通过Spring Boot风格进行再封装屏蔽掉了复杂的配置和实现原理，最终给开发者留出了一套简单易懂、易部署和易维护的分布式系统开发工具包。

**2. SpringBoot和SpringCloud的区别?**

> SpringBoot专注于快速方便的开发单个个体微服务。
>
> SpringCloud是关注全局的微服务协调整理治理框架，它将SpringBoot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、全局锁、决策竞选、分布式会话等等集成服务
>
> SpringBoot可以离开SpringCloud独立使用开发项目， 但是SpringCloud离不开SpringBoot ，属于依赖的关系
>
> SpringBoot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。

**3. Spring Cloud 和dubbo区别?**

> （1）服务调用方式：dubbo是RPC，Spring Cloud是Rest Api。
>
> （2）注册中心：dubbo 是zookeeper，Spring Cloud可以是zookeeper或其他。
>
> （3）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。
>
> （4）架构完整度：Spring Cloud包含诸多微服务组件要素，完整度上比Dubbo高。

**4. Spring Cloud主要的子项目（Spring Cloud主要的组件）有哪些**

> Spring Cloud Config：配置中心，集中配置管理工具
>
> Spring Cloud Bus： 用于传播集群状态变化的消息总线
>
> Spring Cloud Consul：服务注册与发现
>
> Spring Cloud Stream：轻量级事件驱动微服务框架，可以使用简单的声明式模型来发送及接收消息，主要实现为Apache Kafka及RabbitMQ
>
> Spring Cloud Gateway：API网关组件，对请求提供路由及过滤功能。
>
> Spring Cloud OpenFeign：基于Ribbon和Hystrix的声明式服务调用组件，可以动态创建基于Spring MVC注解的接口实现用于服务调用。

## 进阶

**1. 负载均衡的意义什么?**

> 在计算中，负载均衡可以改善跨计算机，计算机集群，网络链接，中央处理单元或磁盘驱动器等多种计算资源的工作负载分布。负载平衡旨在优化资源使用，最大化吞吐量，最小化响应时间并避免任何单一资源的过载。使用多个组件进行负载均衡而不是单个组件可能会通过冗余来提高可靠性和可用性。负载均衡通常涉及专用软件或硬件，例如多层交换机或域名系统服务器进程。

**2. 什么是 Hystrix？它如何实现容错?**

> Hystrix 是一个延迟和容错库，旨在隔离远程系统，服务和第三方库的访问点，当出现故障是不可避免的故障时，停止级联故障并在复杂的分布式系统中实现弹性。

**3. 什么是 Hystrix 断路器？我们需要它吗?**

> 在一个分布式系统里，许多依赖不可避免的会调用失败，比如超时、异常等。如果没有采取任何措施的话就会导致雪崩效应。
>
> 服务雪崩效应是一种因“服务提供者的不可用”（原因）导致“服务调用者不可用”（结果），并将不可用逐渐放大的现象
>
> 如何能够保证在一个依赖出问题的情况下，不会导致整体服务失败，这个就是Hystrix需要做的事情。Hystrix提供了熔断、隔离、Fallback、cache、监控等功能，能够在一个、或多个依赖同时出现问题时保证系统依然可用。

**4. Hystrix断路器的状态以及如何切换**

> 完全打开状态: 一段时间内 达到一定的次数无法调用 并且多次监测没有恢复的迹象 断路器完全打开 那么下次请求就不会请求到该服务
>
> 半开: 短时间内 有恢复迹象 断路器会将部分请求发给该服务，正常调用时 断路器关闭
>
> 关闭：当服务一直处于正常状态 能正常调用

## 高级

**1. 微服务的优缺点分别是什么?**

> **优点**
>
> - 每一个服务足够内聚,代码容易理解
> - 开发效率提高,一个服务只做一件事
> - 微服务能够被小团队单独开发
> - 微服务是松耦合的,是有功能意义的服务
> - 可以用不同的语言开发,面向接口编程
> - 易于与第三方集成
> - 微服务只是业务逻辑的代码,不会和HTML,CSS或者其他界面组合
>
> 开发中,两种开发模式
> 前后端分离
> 全栈工程师
>
> - 可以灵活搭配,连接公共库/连接独立库
>
> **缺点**
>
> - 分布式系统的负责性
> - 多服务运维难度,随着服务的增加,运维的压力也在增大
> - 系统部署依赖
> - 服务间通信成本
> - 数据一致性
> - 系统集成测试
> - 性能监控

**2. Eureka和ZooKeeper都可以提供服务注册与发现的功能,请说说两个的区别**

> 1.ZooKeeper保证的是CP,Eureka保证的是AP
>
> ZooKeeper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

> Eureka各个节点是平等关系,只要有一台Eureka就可以保证服务可用,而查询到的数据并不是最新的
>
> 自我保护机制会导致
>
> Eureka不再从注册列表移除因长时间没收到心跳而应该过期的服务
>
> Eureka仍然能够接受新服务的注册和查询请求,但是不会被同步到其他节点(高可用)
>
> 当网络稳定时,当前实例新的注册信息会被同步到其他节点中(最终一致性)
>
> 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
>
> 2.ZooKeeper有Leader和Follower角色,Eureka各个节点平等
>
> 3.ZooKeeper采用过半数存活原则,Eureka采用自我保护机制解决分区问题
>
> 4.Eureka本质上是一个工程,而ZooKeeper只是一个进程

**3. eureka自我保护机制是什么?**

> 当Eureka Server 节点在短时间内丢失了过多实例的连接时（比如网络故障或频繁启动关闭客户端）节点会进入自我保护模式，保护注册信息，不再删除注册数据，故障恢复时，自动退出自我保护模式。

# 七、Redis

## 基础

**1. Redis有哪些数据结构?**

> 字符串String、字典Hash、列表List、集合Set、有序集合SortedSet。
> 如果你是Redis中高级用户，还需要加上下面几种数据结构HyperLogLog、Geo、Pub/Sub。
> 如果你说还玩过Redis Module，像BloomFilter，RedisSearch，Redis-ML，面试官得眼睛就开始发亮了。

**2. 使用过Redis分布式锁么，它是什么回事?**

> 先拿setnx来争抢锁，抢到之后，再用expire给锁加一个过期时间防止锁忘记了释放。
>
> set指令有非常复杂的参数，可以同时把setnx和expire合成一条指令来用的，从而保证原子性

**3. Redis里面有1亿个key，其中有10w个key是以某个固定的已知的前缀开头的，如何将它们全部找出来?**

> 有两种方式：keys指令和scan指令，建议使用scan指令。
>
> 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。
>
> 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

**4. 如果有大量的key需要设置同一时间过期，一般需要注意什么?**

> 如果大量的key过期时间设置的过于集中，到过期的那个时间点，redis可能会出现短暂的卡顿现象。一般需要在时间上加一个随机值，使得过期时间分散一些。

**5. Redis如何做持久化的?**

> bgsave做镜像全量持久化，aof做增量持久化。因为bgsave会耗费较长时间，不够实时，在停机的时候会导致大量丢失数据，所以需要aof来配合使用。
>
> 在redis实例重启时，优先使用aof来恢复内存的状态，如果没有aof日志，就会使用rdb文件来恢复。
>
> Redis4.0之后有了混合持久化的功能，将bgsave的全量和aof的增量做了融合处理，这样既保证了恢复的效率又兼顾了数据的安全性.

**6. Pipeline有什么好处，为什么要用pipeline?**

> 可以将多次IO往返的时间缩减为一次，前提是pipeline执行的指令之间没有因果相关性。使用redis-benchmark进行压测的时候可以发现影响redis的QPS峰值的一个重要因素是pipeline批次指令的数目。

**7. Redis的同步机制了解么?**

> 主从同步。第一次同步时，主节点做一次bgsave，并同时将后续修改操作记录到内存buffer，待完成后将rdb文件全量同步到复制节点，复制节点接受完成后将rdb镜像加载到内存。加载完成后，再通知主节点将期间修改的操作记录同步到复制节点进行重放就完成了同步过程。

**8. 是否使用过Redis集群，集群的原理是什么?**

> Redis Sentinal着眼于高可用，在master宕机时会自动将slave提升为master，继续提供服务。
> Redis Cluster着眼于扩展性，在单个redis内存不足时，使用Cluster进行分片存储。

## 进阶

**1、在项目中缓存是如何使用的？为什么要用缓存？缓存使用不当会造成什么后果?**

> 项目中缓存是如何使用的?
>
> 这个，需要结合自己项目的业务来。
>
> 为什么要用缓存?
>
> 用缓存，主要有两个用途：高性能、高并发。

**2. redis 的线程模型**

> redis 内部使用文件事件处理器 file event handler，这个文件事件处理器是单线程的，所以 redis 才叫做单线程的模型。它采用 IO 多路复用机制同时监听多个 socket，将产生事件的 socket 压入内存队列中，事件分派器根据 socket 上的事件类型来选择对应的事件处理器进行处理。
>
> 文件事件处理器的结构包含 4 个部分：
>
> - 多个 socket
> - IO 多路复用程序
> - 文件事件分派器
> - 事件处理器（连接应答处理器、命令请求处理器、命令回复处理器）
>
> 多个 socket 可能会并发产生不同的操作，每个操作对应不同的文件事件，但是 IO 多路复用程序会监听多个 socket，会将产生事件的 socket 放入队列中排队，事件分派器每次从队列中取出一个 socket，根据 socket 的事件类型交给对应的事件处理器进行处理。

**3. 为啥 redis 单线程模型也能效率这么高?**

> - 纯内存操作
> - 核心是基于非阻塞的 IO 多路复用机制

> - 单线程反而避免了多线程的频繁上下文切换问题

**4. 如何保证缓存与数据库的双写一致性?**

> 延迟双删

## 高级

**1. 了解什么是 Redis 的雪崩、穿透和击穿？Redis 崩溃之后会怎么样？系统该如何应对这种情况？如何处理 Redis 的穿透?**

> 缓存雪崩的事前事中事后的解决方案如下：
>
> - 事前：Redis 高可用，主从+哨兵，Redis cluster，避免全盘崩溃。
> - 事中：本地 ehcache 缓存 + hystrix 限流&降级，避免 MySQL 被打死。
> - 事后：Redis 持久化，一旦重启，自动从磁盘上加载数据，快速恢复缓存数据。
>
> 缓存穿透：布隆过滤器
>
> 缓存击穿：就是说某个 key 非常热点，访问非常频繁，处于集中式高并发访问的情况，当这个 key 在失效的瞬间，大量的请求就击穿了缓存，直接请求数据库，就像是在一道屏障上凿开了一个洞。
>
> 不同场景下的解决方式可如下：
>
> - 若缓存的数据是基本不会发生更新的，则可尝试将该热点数据设置为永不过期。
> - 若缓存的数据更新不频繁，且缓存刷新的整个流程耗时较少的情况下，则可以采用基于 Redis、zookeeper 等分布式中间件的分布式互斥锁，或者本地互斥锁以保证仅少量的请求能请求数据库并重新构建缓存，其余线程则在锁释放后能访问到新缓存。
> - 若缓存的数据更新频繁或者在缓存刷新的流程耗时较长的情况下，可以利用定时线程在缓存过期前主动地重新构建缓存或者延后缓存的过期时间，以保证所有的请求能一直访问到对应的缓存。

**2. Redis 集群模式的工作原理能说一下么？在集群模式下，Redis 的 key 是如何寻址的？分布式寻址都有哪些算法？了解一致性 hash 算法吗?**

> ### Redis cluster 介绍
>
> - 自动将数据进行分片，每个 master 上放一部分数据
> - 提供内置的高可用支持，部分 master 不可用时，还是可以继续工作的
>
> ### 分布式寻址算法
>
> - hash 算法（大量缓存重建）
> - 一致性 hash 算法（自动缓存迁移）+ 虚拟节点（自动负载均衡）
> - Redis cluster 的 hash slot 算法

# 八、MQ

## 基础

**1. 为什么使用MQ？MQ的优点**

> - 异步处理 - 相比于传统的串行、并行方式，提高了系统吞吐量。

- 应用解耦 - 系统间通过消息通信，不用关心其他系统的处理。
- 流量削锋 - 可以通过消息队列长度控制请求量；可以缓解短时间内的高并发请求。
- 日志处理 - 解决大量日志传输。
- 消息通讯 - 消息队列一般都内置了高效的通信机制，因此也可以用在纯的消息通讯。比如实现点对点消息队列，或者聊天室等。

**2. 消息队列有什么优缺点？RabbitMQ有什么优缺点?**

**在特殊场景下有其对应的好处，解耦、异步、削峰。**

缺点有以下几个：

**系统可用性降低**

本来系统运行好好的，现在你非要加入个消息队列进去，那消息队列挂了，你的系统不是呵呵了。因此，系统可用性会降低；

**系统复杂度提高**

加入了消息队列，要多考虑很多方面的问题，比如：一致性问题、如何保证消息不被重复消费、如何保证消息可靠性传输等。因此，需要考虑的东西更多，复杂性增大。

**一致性问题**

A 系统处理完了直接返回成功了，人都以为你这个请求就成功了；但是问题是，要是 BCD 三个系统那里，BD 两个系统写库成功了，结果 C 系统写库失败了，咋整？你这数据就不一致了。

**3. MQ 有哪些常见问题？如何解决这些问题?**

MQ 的常见问题有：

1. 消息的顺序问题
2. 消息的重复问题

**消息的顺序问题**

消息有序指的是可以按照消息的发送顺序来消费。

假如生产者产生了 2 条消息：M1、M2，假定 M1 发送到 S1，M2 发送到 S2，如果要保证 M1 先于 M2 被消费，怎么做?

解决方案：

（1）保证生产者 - MQServer - 消费者是一对一对一的关系

缺陷：

- 并行度就会成为消息系统的瓶颈（吞吐量不够）
- 更多的异常处理，比如：只要消费端出现问题，就会导致整个处理流程阻塞，我们不得不花费更多的精力来解决阻塞的问题。 （2）通过合理的设计或者将问题分解来规避。
- 不关注乱序的应用实际大量存在
- 队列无序并不意味着消息无序 所以从业务层面来保证消息的顺序而不仅仅是依赖于消息系统，是一种更合理的方式。

**消息的重复问题**

造成消息重复的根本原因是：网络不可达。

所以解决这个问题的办法就是绕过这个问题。那么问题就变成了：如果消费端收到两条一样的消息，应该怎样处理?

> 消费端处理消息的业务逻辑保持幂等性。只要保持幂等性，不管来多少条重复消息，最后处理的结果都一样。保证每条消息都有唯一编号且保证消息处理成功与去重表的日志同时出现。利用一张日志表来记录已经处理成功的消息的 ID，如果新到的消息 ID 已经在日志表中，那么就不再处理这条消息。

## RabbitMQ

### 1. RabbitMQ基本概念

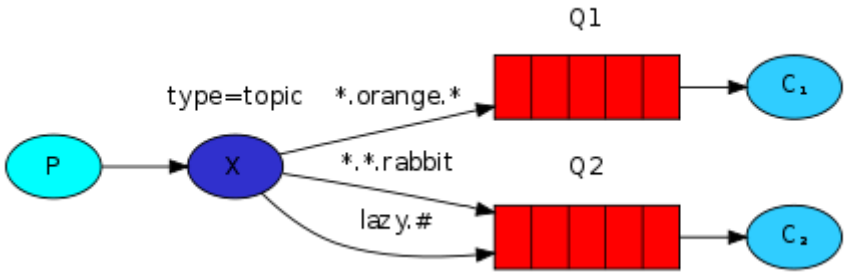
> - Broker： 简单来说就是消息队列服务器实体
> - Exchange： 消息交换机，它指定消息按什么规则，路由到哪个队列
> - Queue： 消息队列载体，每个消息都会被投入到一个或多个队列
> - Binding： 绑定，它的作用就是把exchange和queue按照路由规则绑定起来
> - Routing Key： 路由关键字，exchange根据这个关键字进行消息投递
> - VHost： vhost 可以理解为虚拟 broker ，即 mini-RabbitMQ server。其内部均含有独立的 queue、exchange 和 binding 等，但最最重要的是，其拥有独立的权限系统，可以做到 vhost 范围的用户控制。当然，从 RabbitMQ 的全局角度，vhost 可以作为不同权限隔离的手段（一个典型的例子就是不同的应用可以跑在不同的 vhost 中）。
> - Producer： 消息生产者，就是投递消息的程序
> - Consumer： 消息消费者，就是接受消息的程序
> - Channel： 消息通道，在客户端的每个连接里，可建立多个channel，每个channel代表一个会话任务
>
> 由Exchange、Queue、RoutingKey三个才能决定一个从Exchange到Queue的唯一的线路。

### 2. RabbitMQ的工作模式

> **一.simple模式（即最简单的收发模式）**
>
> 1.消息产生消息，将消息放入队列
>
> 2.消息的消费者(consumer) 监听 消息队列,如果队列中有消息,就消费掉,消息被拿走后,自动从队列中删除(隐患 消息可能没有被消费者正确处理,已经从队列中消失了,造成消息的丢失，这里可以设置成手动的ack,但如果设置成手动ack，处理完后要及时发送ack消息给队列，否则会造成内存溢出)。
>
> **二.work工作模式(资源的竞争)**
>
> 1.消息产生者将消息放入队列消费者可以有多个,消费者1,消费者2同时监听同一个队列,消息被消费。C1 C2共同争抢当前的消息队列内容,谁先拿到谁负责消费消息(隐患：高并发情况下,默认会产生某一个消息被多个消费者共同使用,可以设置一个开关(syncronize) 保证一条消息只能被一个消费者使用)。
>
> **三.publish/subscribe发布订阅(共享资源)**
>
> 1、每个消费者监听自己的队列；
>
> 2、生产者将消息发给broker，由交换机将消息转发到绑定此交换机的每个队列，每个绑定交换机的队列都将接收到消息。
>
> **四.routing路由模式**
>
> 1.消息生产者将消息发送给交换机按照路由判断,路由是字符串(info) 当前产生的消息携带路由字符(对象的方法),交换机根据路由的key,只能匹配上路由key对应的消息队列,对应的消费者才能消费消息;
>
> 2.根据业务功能定义路由字符串
>
> 3.从系统的代码逻辑中获取对应的功能字符串,将消息任务扔到对应的队列中。

4.业务场景:error 通知;EXCEPTION;错误通知的功能;传统意义的错误通知;客户通知;利用key路由,可以将程序中的错误封装成消息传入到消息队列中,开发者可以自定义消费者,实时接收错误;

**五.topic 主题模式(路由模式的一种)**




1.星号井号代表通配符

2.星号代表多个单词,井号代表一个单词

3.路由功能添加模糊匹配

4.消息产生者产生消息,把消息交给交换机

5.交换机根据key的规则模糊匹配到对应的队列,由队列的监听消费者接收消息消费

**3. 如何保证RabbitMQ消息的顺序性?**

拆分多个 queue，每个 queue 一个 consumer；或者就一个 queue 但是对应一个 consumer，然后这个 consumer 内部用内存队列做排队，然后分发给底层不同的 worker 来处理。

**4. 消息如何分发?**

若该队列至少有一个消费者订阅，消息将以循环（round-robin）的方式发送给消费者。每条消息只会分发给一个订阅的消费者（前提是消费者能够正常处理消息并进行确认）。通过路由可实现多消费的功能

**5. 消息基于什么传输?**

由于 TCP 连接的创建和销毁开销较大，且并发数受系统资源限制，会造成性能瓶颈。RabbitMQ 使用信道的方式来传输数据。信道是建立在真实的 TCP 连接内的虚拟连接，且每条 TCP 连接上的信道数量没有限制。

**6. 如何保证RabbitMQ消息的可靠传输?**

消息不可靠的情况可能是消息丢失，劫持等原因；

丢失又分为：生产者丢失消息、消息列表丢失消息、消费者丢失消息；

**生产者丢失消息**：从生产者弄丢数据这个角度来看，RabbitMQ提供transaction和confirm模式来确保生产者不丢消息；

transaction机制就是说：发送消息前，开启事务（channel.txSelect()）,然后发送消息，如果发送过程中出现什么异常，事务就会回滚（channel.txRollback()）,如果发送成功则提交事务（channel.txCommit()）。然而，这种方式有个缺点：吞吐量下降；

confirm模式用的居多：一旦channel进入confirm模式，所有在该信道上发布的消息都将会被指派一个唯一的ID（从1开始），一旦消息被投递到所有匹配的队列之后；

rabbitMQ就会发送一个ACK给生产者（包含消息的唯一ID），这就使得生产者知道消息已经正确到达目的队列了；

如果rabbitMQ没能处理该消息，则会发送一个Nack消息给你，你可以进行重试操作。

> **消息队列丢数据**：消息持久化。
>
> 处理消息队列丢数据的情况，一般是开启持久化磁盘的配置。
>
> 这个持久化配置可以和confirm机制配合使用，你可以在消息持久化磁盘后，再给生产者发送一个Ack信号。
>
> 这样，如果消息持久化磁盘之前，rabbitMQ阵亡了，那么生产者收不到Ack信号，生产者会自动重发。
>
> 那么如何持久化呢?
>
> 这里顺便说一下吧，其实也很容易，就下面两步
>
> 1. 将queue的持久化标识durable设置为true,则代表是一个持久的队列
> 2. 发送消息的时候将deliveryMode=2
>
> 这样设置以后，即使rabbitMQ挂了，重启后也能恢复数据
>
> **消费者丢失消息**：消费者丢数据一般是因为采用了自动确认消息模式，改为手动确认消息即可!
>
> 消费者在收到消息之后，处理消息之前，会自动回复RabbitMQ已收到消息；
>
> 如果这时处理消息失败，就会丢失该消息；
>
> 解决方案：处理消息成功后，手动回复确认消息。

**7. 如何保证高可用的？RabbitMQ 的集群**

> RabbitMQ 是比较有代表性的，因为是基于主从（非分布式）做高可用性的，我们就以 RabbitMQ 为例子讲解第一种 MQ 的高可用性怎么实现。RabbitMQ 有三种模式：单机模式、普通集群模式、镜像集群模式。
>
> **单机模式**，就是 Demo 级别的，一般就是你本地启动了玩玩儿的?，没人生产用单机模式
>
> **普通集群模式**，意思就是在多台机器上启动多个 RabbitMQ 实例，每个机器启动一个。你创建的 queue，只会放在一个 RabbitMQ 实例上，但是每个实例都同步 queue 的元数据（元数据可以认为是 queue 的一些配置信息，通过元数据，可以找到 queue 所在实例）。你消费的时候，实际上如果连接到了另外一个实例，那么那个实例会从 queue 所在实例上拉取数据过来。这方案主要是提高吞吐量的，就是说让集群中多个节点来服务某个 queue 的读写操作。
>
> **镜像集群模式**：这种模式，才是所谓的 RabbitMQ 的高可用模式。跟普通集群模式不一样的是，在镜像集群模式下，你创建的 queue，无论元数据还是 queue 里的消息都会存在于多个实例上，就是说，每个 RabbitMQ 节点都有这个 queue 的一个完整镜像，包含 queue 的全部数据的意思。然后每次你写消息到 queue 的时候，都会自动把消息同步到多个实例的 queue 上。RabbitMQ 有很好的管理控制台，就是在后台新增一个策略，这个策略是镜像集群模式的策略，指定的时候是可以要求数据同步到所有节点的，也可以要求同步到指定数量的节点，再次创建 queue 的时候，应用这个策略，就会自动将数据同步到其他的节点上去了。这样的话，好处在于，你任何一个机器宕机了，没事儿，其它机器（节点）还包含了这个 queue 的完整数据，别的 consumer 都可以到其它节点上去消费数据。坏处在于，第一，这个性能开销也太大了吧，消息需要同步到所有机器上，导致网络带宽压力和消耗很重！RabbitMQ 一个 queue 的数据都是放在一个节点里的，镜像集群下，也是每个节点都放这个 queue 的完整数据。

## Kafka

**1. 什么是消费者组?**

> 1、定义：即消费者组是 Kafka 提供的可扩展且具有容错性的消费者机制。
> 2、原理：在 Kafka 中，消费者组是一个由多个消费者实例 构成的组。多个实例共同订阅若干个主题，实现共同消费。同一个组下的每个实例都配置有 相同的组 ID，被分配不同的订阅分区。当某个实例挂掉的时候，其他实例会自动地承担起 它负责消费的分区。

**2. 在 Kafka 中，ZooKeeper 的作用是什么?**

> 目前，Kafka 使用 ZooKeeper 存放集群元数据、成员管理、Controller 选举，以及其他一些管理类任务。之后，等 KIP-500 提案完成后，Kafka 将完全不再依赖 于 ZooKeeper。
>
> “存放元数据”是指主题 分区的所有数据都保存在 ZooKeeper 中，且以它保存的数据为权威，其他“人”都要与它 保持对齐。“成员管理”是指 Broker 节点的注册、注销以及属性变更，等 等。“Controller 选举”是指选举集群 Controller，而其他管理类任务包括但不限于主题 删除、参数配置等。
>
> **KIP-500 思想，是使用社区自研的基于 Raft 的共识算法， 替代 ZooKeeper，实现 Controller 自选举。**

**3. 解释下 Kafka 中位移(offset)的作用**

> 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。

**4. 阐述下 Kafka 中的领导者副本(Leader Replica)和追随者副本 (Follower Replica)的区别**

> 这道题表面上是考核你对 Leader 和 Follower 区别的理解，但很容易引申到 Kafka 的同步 机制上。因此，我建议你主动出击，一次性地把隐含的考点也答出来，也许能够暂时把面试 官“唬住”，并体现你的专业性。
>
> 你可以这么回答:**Kafka 副本当前分为领导者副本和追随者副本。只有 Leader 副本才能 对外提供读写服务，响应 Clients 端的请求。Follower 副本只是采用拉(PULL)的方 式，被动地同步 Leader 副本中的数据，并且在 Leader 副本所在的 Broker 宕机后，随时 准备应聘 Leader 副本。**
>
> 通常来说，回答到这个程度，其实才只说了 60%，因此，我建议你再回答两个额外的加分 项。
>
> - **强调 Follower 副本也能对外提供读服务。**自 Kafka 2.4 版本开始，社区通过引入新的 Broker 端参数，允许 Follower 副本有限度地提供读服务。
> - **强调 Leader 和 Follower 的消息序列在实际场景中不一致。**很多原因都可能造成 Leader 和 Follower 保存的消息序列不一致，比如程序 Bug、网络问题等。这是很严重 的错误，必须要完全规避。你可以补充下，之前确保一致性的主要手段是高水位机制， 但高水位值无法保证 Leader 连续变更场景下的数据一致性，因此，社区引入了 Leader Epoch 机制，来修复高水位值的弊端。关于“Leader Epoch 机制”，国内的资料不是 很多，它的普及度远不如高水位，不妨大胆地把这个概念秀出来，力求惊艳一把。

**5. 如何设置 Kafka 能接收的最大消息的大小?**

> **这道题除了要回答消费者端的参数设置之外，一定要加上 Broker 端的设置，这样才算完整。**毕竟，如果 Producer 都不能向 Broker 端发送数据很大的消息，又何来消费一说呢? 因此，你需要同时设置 Broker 端参数和 Consumer 端参数。
>
> - Broker 端参数:message.max.bytes、max.message.bytes(主题级别)和 replica.fetch.max.bytes。
> - Consumer 端参数:fetch.message.max.bytes。
>
> Broker 端的最后一个参数比较容易遗漏。我们必须调整 Follower 副本能够接收的最大消 息的大小，否则，副本同步就会失败。因此，把这个答出来的话，就是一个加分项。

**6. 如何估算 Kafka 集群的机器数量?**

> 这道题目考查的是**机器数量和所用资源之间的关联关系**。所谓资源，也就是 CPU、内存、磁盘和带宽。
>
> 通常来说，CPU 和内存资源的充足是比较容易保证的，因此，你需要从磁盘空间和带宽占用两个维度去评估机器数量。
>
> 在预估磁盘的占用时，你一定不要忘记计算副本同步的开销。如果一条消息占用 1KB 的磁 盘空间，那么，在有 3 个副本的主题中，你就需要 3KB 的总空间来保存这条消息。显式地 将这些考虑因素答出来，能够彰显你考虑问题的全面性，是一个难得的加分项。
>
> 对于评估带宽来说，常见的带宽有 1Gbps 和 10Gbps，但你要切记，**这两个数字仅仅是最大值**。因此，你最好和面试官确认一下给定的带宽是多少。然后，明确阐述出当带宽占用接 近总带宽的 90% 时，丢包情形就会发生。这样能显示出你的网络基本功。

**7. kafka 的 ack 机制**

> request.required.acks 有三个值 0 1 -1
>
> 0:生产者不会等待 broker 的 ack，这个延迟最低但是存储的保证最弱当 server 挂掉的时候
>
> 就会丢数据
>
> 1：服务端会等待 ack 值 leader 副本确认接收到消息后发送 ack 但是如果 leader 挂掉后他
>
> 不确保是否复制完成新 leader 也会导致数据丢失
>
> -1：同样在 1 的基础上 服务端会等所有的 follower 的副本受到数据后才会受到 leader 发出
>
> 的 ack，这样数据不会丢失

# 九、Mysql

## 基础：

**1. MySQL 中有哪几种锁?**

> （1）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，并发度最低。
>
> （2）行级锁：开销大，加锁慢；会出现死锁；锁定粒度最小，发生锁冲突的概率最 低，并发度也最高。
>
> （3）页面锁：开销和加锁时间界于表锁和行锁之间；会出现死锁；锁定粒度界于表 锁和行锁之间，并发度一般。

**2. MySQL 中有哪些不同的存储引擎?**

> （1）MyISAM
>
> （2）Heap
>
> （3）Merge
>
> （4）INNODB
>
> （5）ISAM

**3. 简述在 MySQL 数据库中 MyISAM 和 InnoDB 的区别**

> **MyISAM：**
>
> （1）不支持事务，但是每次查询都是原子的；
>
> （2）支持表级锁，即每次操作是对整个表加锁；
>
> （3）存储表的总行数；
>
> （4）一个 MYISAM 表有三个文件：索引文件、表结构文件、数据文件；
>
> （5）采用菲聚集索引，索引文件的数据域存储指向数据文件的指针。辅索引与主索引基本一致，但是辅索引不用保证唯一性。
>
> **InnoDb：**
>
> （1）支持 ACID 的事务，支持事务的四种隔离级别；
>
> （2）支持行级锁及外键约束：因此可以支持写并发；
>
> （3）不存储总行数：
>
> （4）一个 InnoDb 引擎存储在一个文件空间（共享表空间，表大小不受操作系统控制，一个表可能分布在多个文件里），也有可能为多个（设置为独立表空，表大小受操作系统文件大小限制，一般为 2G），受操作系统文件大小的限制；
>
> （5）主键索引采用聚集索引（索引的数据域存储数据文件本身），辅索引的数据域存储主键的值；因此从辅索引查找数据，需要先通过辅索引找到主键值，再访问辅索引；最好使用自增主键，防止插入数据时，为维持 B+树结构，文件的大调整。

**4. MySQL 中 InnoDB 支持的四种事务隔离级别名称**

> SQL 标准定义的四个隔离级别为：
>
> （1）read uncommited ： 读到未提交数据
>
> （2）read committed：脏读，不可重复读
>
> （3）repeatable read：可重读
>
> （4）serializable ： 串行事物

**5. NOW（）和 CURRENT_DATE（）有什么区别?**

> NOW（）命令用于显示当前年份，月份，日期，小时，分钟和秒。
>
> CURRENT_DATE（）仅显示当前年份，月份和日期。

**6. MySQL 支持事务吗?**

> 在缺省模式下，MySQL 是 autocommit 模式的，所有的数据库更新操作都会即时提交，所以在缺省情况下，MySQL 是不支持事务的。
>
> 但是如果你的 MySQL 表类型是使用 InnoDB Tables 或 BDB tables 的话，你的MySQL 就可以使用事务处理,使用 SETAUTOCOMMIT=0 就可以使 MySQL 允许在非 autocommit 模式，在非 autocommit 模式下，你必须使用 COMMIT 来提交你的更改，或者用 ROLLBACK来回滚你的更改。

**7. 数据库中的事务是什么?**

> 事务（transaction）是作为一个单元的一组有序的数据库操作。如果组中的所有操作都成功，则认为事务成功，即使只有一个操作失败，事务也不成功。如果所有操作完成，事务则提交，其修改将作用于所有其他数据库进程。如果一个操作失败，则事务将回滚，该事务所有操作的影响都将取消。

> 事务特性：
>
> （1）原子性：即不可分割性，事务要么全部被执行，要么就全部不被执行。
>
> （2）一致性或可串性。事务的执行使得数据库从一种正确状态转换成另一种正确状态。
>
> （3）隔离性。在事务正确提交之前，不允许把该事务对数据的任何改变提供给任何其他事务。
>
> （4）持久性。事务正确提交后，其结果将永久保存在数据库中，即使在事务提交后有了其他故障，事务的处理结果也会得到保存。

## 进阶

**1. 锁的优化策略**

> （1）读写分离
>
> （2）分段加锁
>
> （3）减少锁持有的时间
>
> （4）多个线程尽量以相同的顺序去获取资源
>
> 不能将锁的粒度过于细化，不然可能会出现线程的加锁和释放次数过多，反而效率不如一次加一把大锁。

**2. 索引的底层实现原理和优化**

> B+树，经过优化的 B+树
>
> 主要是在所有的叶子结点中增加了指向下一个叶子节点的指针，因此 InnoDB 建议为大部分表使用默认自增的主键作为主索引。

**3. 什么情况下设置了索引但无法使用**

> - 对列用了函数
> - 联合索引不符合最左前缀法则的时候
> - 双%like
> - is null 或者 is not null
> - or 前后不是同一个字段索引
> - 使用了!=

**4. 简单描述 MySQL 中，索引，主键，唯一索引，联合索引的区别，对数据库的性能有什么影响（从读写两方面）**

> 索引是一种特殊的文件(InnoDB 数据表上的索引是表空间的一个组成部分)，它们包含着对数据表里所有记录的引用指针。
>
> 普通索引(由关键字 KEY 或 INDEX 定义的索引)的唯一任务是加快对数据的访问速度。
>
> 普通索引允许被索引的数据列包含重复的值。如果能确定某个数据列将只包含彼此各不相同的值，在为这个数据列创建索引的时候就应该用关键字 UNIQUE 把它定义为一个唯一索引。也就是说，唯一索引可以保证数据记录的唯一性。
>
> 主键，是一种特殊的唯一索引，在一张表中只能定义一个主键索引，主键用于唯一标识一条记录，使用关键字 PRIMARY KEY 来创建。
>
> 索引可以覆盖多个数据列，如像 INDEX(columnA, columnB)索引，这就是联合索引。
>
> 索引可以极大的提高数据的查询速度，但是会降低插入、删除、更新表的速度，因为在执行这些写操作时，还要操作索引文件。

**5. 如何通俗地理解三个范式?**

> 第一范式：1NF 是对属性的原子性约束，要求属性具有原子性，不可再分解；
>
> 第二范式：2NF 是对记录的惟一性约束，要求记录有惟一标识，即实体的惟一性；
>
> 第三范式：3NF 是对字段冗余性的约束，即任何字段不能由其他字段派生出来，它要求字段没有冗余。

**6. 描述下查询sql的执行过程**

> 当执行一条查询的SQl的时候大概发生了一下的步骤：
>
> 1. 客户端发送查询语句给服务器。
> 2. 服务器首先检查缓存中是否存在该查询，若存在，返回缓存中存在的结果。若是不存在就进行下一步。
> 3. 服务器进行SQl的解析、语法检测和预处理，再由优化器生成对应的执行计划。
> 4. Mysql的执行器根据优化器生成的执行计划执行，调用存储引擎的接口进行查询。
> 5. 服务器将查询的结果返回客户端。

**7. 谈谈 SQL 优化的经验**

> - 查询语句无论是使用哪种判断条件 **等于、小于、大于**， `WHERE` 左侧的条件查询字段不要使用函数或者表达式
> - 使用 `EXPLAIN` 命令优化你的 SELECT 查询，对于复杂、效率低的 sql 语句，我们通常是使用 explain sql 来分析这条 sql 语句，这样方便我们分析，进行优化。
> - 当你的 SELECT 查询语句只需要使用一条记录时，要使用 `LIMIT 1`
> - 不要直接使用 `SELECT *`，而应该使用具体需要查询的表字段，因为使用 EXPLAIN 进行分析时，SELECT * 使用的是全表扫描，也就是 `type = all`。
> - 为每一张表设置一个 ID 属性
> - 避免在 `WHERE` 字句中对字段进行 `NULL` 判断
> - 避免在 `WHERE` 中使用 `!=` 或 `<>` 操作符
> - 使用 `BETWEEN AND` 替代 `IN`
> - 为搜索字段创建索引
> - 选择正确的存储引擎，InnoDB 、MyISAM 、MEMORY 等
> - 使用 `LIKE %abc%` 不会走索引，而使用 `LIKE abc%` 会走索引
> - 对于枚举类型的字段(即有固定罗列值的字段)，建议使用`ENUM`而不是`VARCHAR`，如性别、星期、类型、类别等
> - 拆分大的 DELETE 或 INSERT 语句
> - 选择合适的字段类型，选择标准是 **尽可能小、尽可能定长、尽可能使用整数。**
> - 字段设计尽可能使用 `NOT NULL`
> - 进行水平切割或者垂直分割