Java面试宝典

Version : V1.0

7. HashMap的扩容机制
8. HashMap链表和红黑树转化时机!
9. 什么是Hash碰撞以及常见的解决方案
10. HashMap在多线程操作中会出现什么问题!
11. 如何保证集合是线程安全的?
12. 聊一聊JUC包下的并发工具
13. ConcurrentHashMap如何保证线程安全
14. ConcurrentHashMap在1.7和1.8的底层实现
15. ConcurrentLinkedQueue和LinkedBlockingQueue有什么区别
16. 说一下 HashSet 的实现原理
17. HashSet和TreeSet有什么区别?

四、多线程 【25道】

1. 聊一聊并行和并发的区别?
2. 线程和进程的区别?
3. 说一说什么是原子性、可见性、有序性?
4. Java中实现多线程的方式
5. Java中提供的线程池种类，及其特点!
6. 线程池的工作原理以及重要参数含义
7. 线程池的阻塞队列有哪些?
8. 线程池的拒绝策略有哪些?
9. synchronized和reentranlock锁的区别?
10. synchronized底层如何实现？什么是锁升级、降级!
11. reentranlock的底层实现原理
12. volatile关键字的特点
13. Java为什么会指令重排
14. 悲观锁和乐观锁的区别
15. 什么是CAS, 以及它的ABA问题, 如何解决ABA问题
16. Atomic变量如何保证的原子性
17. ThreadLocal是什么，有什么作用
18. ThreadLocal的内存泄漏问题
19. jdk1.8对锁进行了哪些优化
20. 死锁发生的情况和如何避免死锁
21. 介绍锁的升级过程
22. 介绍锁的降级过程
23. 怎么解决多线程的大量访问时的数据同步
24. 线程的 run()和 start()有什么区别
25. JDK1.6对Synchronized底层做了哪些优化

五、IO【5道】

1. Java提供了哪些IO方式?
2. NIO, BIO, AIO区别?
3. 有哪些缓冲流？如何实现缓冲功能!
4. 实现文件拷贝的几种方式!
5. 什么Java序列化，如何实现序列化!

六、网络编程 【9道】

1. http协议和RPC协议区别
2. OSI网络模型七层都有哪些
3. 传输层上的TCP和UDP区别
4. http协议1.0和1.1版本区别
5. http是短连接还是长连接, 如何实现的
6. 简述tcp三次握手四次挥手过程
7. http有多少类响应码？分别什么含义
8. tomcat的实现原理！tomcat如何进行优化?
9. get 和 post 请求有哪些区别?

七、MySQL以及SQL面试题【20道】

1. 说一说什么是数据库事务!
2. 事务并发产生的问题和隔离级别!
3. Spring中事务的传播特性有哪些?
4. MySQL的内部有哪几部分组成?

一、Java基础面试题【24道】

1. 聊一聊Java平台的理解!

Java 本身是一种面向对象的语言，最显著的特性有两个方面，一是所谓的“书写一次，到处运行”，能够非常容易地获得跨平台能力；另外就是垃圾收集，Java 通过垃圾收集器回收分配内存，大部分情况下，程序员不需要自己操心内存的分配和回收。 一次编译、到处运行”说的是Java语言跨平台的特性，Java的跨平台特性与Java虚拟机的存在密不可分，可在不同的环境中运行。比如说Windows平台和Linux平台都有相应的JDK，安装好JDK后也就有了Java语言的运行环境。其实Java语言本身与其他的编程语言没有特别大的差异，并不是说Java语言可以跨平台，而是在不同的平台都有可以让Java语言运行的环境而已，所以才有了Java一次编译，到处运行这样的效果。 Java具有三大特性: 封装,继承,多态.利用这三大特性增加代码的复用率和灵活度.

2. String、StringBuffer、StringBuilder的区别!

String的典型的不可变类型，内部使用final关键字修饰，所以每次进行字符串操作（拼接，截取等）都会产生新的对象！在开发中，会使用到大量的字符串操作！所以字符串的临时对象，会对程序操作较大的性能开销！产生了大量的内存浪费!

StringBuilder比StringBuffer速度快, StringBuffer比String速度快

StringBuffer线程安全

3. int和Integer的区别!

int是基本数据类型，是Java的8个原始数据类型之一，直接存数值。

Integer是int对应的包装类，在拆箱和装箱中java 可以根据上下文，自动进行转换，极大地简化了相关编程。

int是基本类型，Integer是对象，用一个引用指向这个对象。由于Integer是一个对象，在JVM中对象需要一定的数据结构进行描述，相比int而言，其占用的内存更大一些。

4. == 和 equals 的区别是什么?

==是直接比较的两个对象的堆内存地址，如果相等，则说明这两个引用实际是指向同一个对象地址的。

因此基本数据类型和String常量是可以直接通过==来直接比较的。

对于引用对象对比是否相等使用equals方法,equals方法比较的是对象中的值是否相等, 但是要从写hashcode和equals

5. 聊一聊JDK1.8版本的新特性!

拉姆达表达式

函数式接口,

方法引用

新的日期类LocalDate

引入了stream类，支持链式编程

6. final关键字的作用?

final关键字可以用来修饰变量、方法和类。

被final修饰的类该类成为最终类，无法被继承。

被final修饰的方法将成为最终方法，无法被子类重写。但是，该方法仍然可以被继承。

final修饰类中的属性或者变量

无论属性是基本类型还是引用类型，final所起的作用都是变量里面存放的"值"不能变。

对于一个final变量，如果是基本数据类型的变量，则其数值一旦在初始化之后便不能更改；如果是引用类型的变量，则在对其初始化之后便不能再让其指向另一个对象。

7. 什么是内存泄漏和内存溢出!

内存泄漏(memoryleak)，是指应用程序在申请内存后，无法释放已经申请的内存空间，一次内存泄漏危害可以忽略，但如果任其发展最终会导致内存溢出（outofmemory）。如读取 文件后流要进行及时的关闭以及对数据库连接的释放。

内存溢出(outofmemory)是指应用程序在申请内存时，没有足够的内存空间供其使用。如我们在项目中对于大批量数据的导入，采用分批量提交的方式。

8. 抽象类和接口的区别!

抽象类 : 抽象类必须用 abstract 修饰，子类必须实现抽象类中的抽象方法，如果有未实现的，那么子类也必须用 abstract 修饰。抽象类默认的权限修饰符为 public，可以定义为 public 或 procted，如果定义为 private，那么子类则无法继承。抽象类不能创建对象

接口和抽象类的区别 : 抽象类只能继承一次，但是可以实现多个接口

接口和抽象类必须实现其中所有的方法，抽象类中如果有未实现的抽象方法，那么子类也需要定义为抽象类。抽象类中可以有非抽象的方法

接口中的变量必须用 public static final 修饰，并且需要给出初始值。所以实现类不能重新定义，也不能改变其值。

接口中的方法默认是 public abstract，也只能是这个类型。不能是 static，接口中的方法也不允许子类覆写，抽象类中允许有static 的方法

9. Error和Execption的区别?

Exception 和 Error 都是继承了 Throwable 类，在 Java 中只有 Throwable 类型的实例才可以被抛出（throw）或者捕获（catch），它是异常处理机制的基本组成类型。

Exception 和 Error 体现了 Java 平台设计者对不同异常情况的分类。Exception 是程序正常运行中，可以预料的意外情况，可能并且应该被捕获，进行相应处理。

Error 是指在正常情况下，不大可能出现的情况，绝大部分的 Error 都会导致程序（比如 JVM自身）处于非正常的、不可恢复状态。既然是非正常情况，所以不便于也不需要捕获，常见的比如 OutOfMemoryError 之类，都是 Error 的子类。

Exception 又分为可检查（checked）异常和不检查（unchecked）异常，可检查异常

10. 说一说常见的Execption和解决方案!

- 数组越界异常 : Java.lang.ArrayIndexOutofBoundsException

  产生的原因：访问了不存在的索引

  解决的办法：索引0到数组长度-1的范围内取值
- 空指针异常 : Java.lang.NullPointerException

  产生的原因：对象没有创建就访问了元素或者方法或者属性

  解决的办法: 先找出出现的所有引用类型，判断哪个对象是没有new的元素或者方法或者属性，如果没有就创建该对象
- 没有这样的元素异常 : Java.util.NoSuchElementException

  产生的原因：在迭代器迭代的时候没有下一个元素了

  解决的办法：在迭代器之前做相对应得判断，如果没有元素了就不迭代输出了
- 并发修改异常 : Java.util.ConcurrentModificationException

  产生的原因：在迭代器迭代的同时使用集合修改元素

  解决的办法：使用普通for循环来遍历 , 使用toArray来遍历 , 使用ListIterator来遍历
- 类型转换异常 : Java.lang.ClassCastException

  产生的原因：在向下转型的过程中，没有转换成真实的类型

  解决的方法：在向下转型之前使用instanceof关键字对所有子类做逐一判断
- 算法出错异常 : Java.lang.ArithmeticException

  产生的原因：除数不能为零

  解决的办法：改变除数的结果再进行测试
- 没有序列化异常 : Java.io.NotSerialzableException

  产生的原因：没有实现serializable接口

  解决的办法：对需要的写入到文件的类实现serializable接口，表示允许该类的该类写入到文件

11. 重写和重载区别

重写: 子类继承父类, 在子类中存在和父类中一模一样的方法, 从新编写方法中的实现业务代码.

重载: 在同一个类中存在多个方法名相同, 传参个数,顺序和类型不同, 返回值可以相同也可以不同的方法.

12. 什么是反射, 反射的好处?

Java 反射，就是在运行状态中:

- 获取任意类的名称、package信息、所有属性、方法、注解、类型、类加载器等
- 获取任意对象的属性，并且能改变对象的属性
- 调用任意对象的方法
- 判断任意一个对象所属的类

- 实例化任意一个类的对象

Java 的动态就体现在这。通过反射我们可以实现动态装配，降低代码的耦合度；动态代理等。反射的过度使用会严重消耗系统资源。

13. 事务控制在哪一层? 可否控制在dao层? 为什么?

- 事务必须控制在service层, 不可以在dao层.
- 因为dao层需要按照单一职责原则设计, 一个类对应一张表, 类中最好都是单表增删改查, 增加代码复用率而事务具有隔离性, service会调用多个dao方法组装业务, 如果事务控制在dao层就会被分隔成多个事务, 无法整体控制
- 所以事务必须控制在service层, 保证一套业务操作要么全成功, 要么全失败.

14. Cookie和session的区别和联系?

- Cookie是客户端浏览器用来保存数据的一块空间, cookie没有session安全, cookie中保存的数据量有大小限制
- Session是服务器端用来保存数据的一块空间, session比较安全
- Cookie和session联系:

  当用户第一次发送请求访问网站的时候, cookie中没有信息, 这个时候服务器就会生成一个session对象, 这个session对象有个唯一id叫做sessionID, 这个id会被保存到用户的浏览器cookie中.

  当用户再次访问服务器的时候, cookie中就会有sessionID, 会被带到服务器, 服务器会根据这个id找到对应的session对象使用.

  如果服务器没有找到对应的session对象则会新生成一个session对象, 然后将新的session对象的id再次写入到cookie中保存.

15. 一个线程两次调用start。方法会出现什么情况?

Java的线程是不允许启动两次的，第二次调用必然会抛出legalThreadStateException，这是一种运行时异常，多次调用 start 被认为是编程错误。

16. 谈谈final、finally、 finalize有什么不同!

final 可以用来修饰类、方法、变量，分别有不同的意义，final 修饰的 class 代表不可以继承扩展，final 的变量是不可以修改的，而 final 的方法也是不可以重写的（override）。

finally 则是 Java 保证重点代码一定要被执行的一种机制。我们可以使用 try-finally 或者 trycatch-finally 来进行类似关闭 JDBC 连接、保证 unlock 锁等动作。

finalize 是基础类 java.lang.Object 的一个方法，它的设计目的是保证对象在被垃圾收集前完成特定资源的回收。finalize 机制现在已经不推荐使用，并且在 JDK 9 开始被标记为deprecated。

17. 强引用、软引用、弱引用、幻象引用有什么区别!

- 所谓强引用（"Strong" Reference）：

就是我们最常见的普通对象引用，只要还有强引用指向一个对象，就能表明对象还“活着”，垃圾收集器不会碰这种对象。对于一个普通的对象，如果没有其他的引用关系，只要超过了引用的作用域或者显式地将相（强）引用赋值为 null，就是可以被垃圾收集的了，当然具体回收时机还是要看垃圾收集策略。

- 软引用（SoftReference）：

  是一种相对强引用弱化一些的引用，可以让对象豁免一些垃圾收集，只有当 JVM 认为内存不足时，才会去试图回收软引用指向的对象。JVM 会确保在抛出OutOfMemoryError 之前，清理软引用指向的对象。软引用通常用来实现内存敏感的缓存，如果还有空闲内存，就可以暂时保留缓存，当内存不足时清理掉，这样就保证了使用缓存的同时，不会耗尽内存。

- 弱引用（WeakReference）：

  并不能使对象豁免垃圾收集，仅仅是提供一种访问在弱引用状态下对象的途径。这就可以用来构建一种没有特定约束的关系，比如，维护一种非强制性的映射关系，如果试图获取时对象还在，就使用它，否则重现实例化。它同样是很多缓存实现的选择。

- 幻象引用：

  有时候也翻译成虚引用，你不能通过它访问对象。幻象引用仅仅是提供了一种确保对象被 finalize 以后，做某些事情的机制，比如，通常用来做所谓的 Post-Mortem 清理机制，我在专栏上一讲中介绍的 Java 平台自身 Cleaner 机制等，也有人利用幻象引用监控对象的创建和销毁。

18. 父子类静态代码块, 非静态代码块, 构造方法执行顺序

父类 - 静态代码块
子类 - 静态代码块
父类 - 非静态代码
父类 - 构造函数
子类 - 非静态代码
子类 - 构造函数

19. 如何实现对象克隆

实现 Cloneable 接口并重写 Object 类中的 clone()方法；

实现 Serializable 接口，通过对象的序列化和反序列化实现克隆，可以实现真正的深度克隆

20. 什么是 java 序列化，如何实现 java 序列化?

序列化就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

21. 深拷贝和浅拷贝区别是什么?

浅拷贝只是复制了对象的引用地址，两个对象指向同一个内存地址，所以修改其中任意的值，另一个值都会随之变化，这就是浅拷贝

深拷贝是将对象及值复制过来，两个对象修改其中任意的值另一个值不会改变，这就是深拷贝

22. jsp 和 servlet 有什么区别?

jsp经编译后就变成了Servlet.（JSP的本质就是Servlet，JVM只能识别Java的类，不能识别JSP的代码，Web容器将JSP的代码编译成JVM能够识别的Java类）

jsp更擅长表现于页面显示，servlet更擅长于逻辑控制。

23. 说一下 jsp 的 4 种作用域?

JSP中的四种作用域包括page、request、session和application：

- page : 代表与一个页面相关的对象和属性。
- request : 代表与Web客户机发出的一个请求相关的对象和属性。一个请求可能跨越多个页面，涉及多个Web组件；需要在页面显示的临时数据可以置于此作用域。
- session : 代表与某个用户与服务器建立的一次会话相关的对象和属性。跟某个用户相关的数据应该放在用户自己的session中。
- application : 代表与整个Web应用程序相关的对象和属性，它实质上是跨越整个Web应用程序，包括多个页面、请求和会话的一个全局作用域。

24. 请求转发（forward）和重定向（redirect）的区别?

请求转发forward比重定向redirect快

请求转发的url地址只能是本网站内部地址, 重定向的url地址可以是外部网站地址

请求转发浏览器url不发生变化，重定向浏览器url地址发生改变.

请求转发request域中的数据可以带到转发后的方法中, 重定向request域中的数据无法带到重定向后的方法中.

二、JVM虚拟机面试题【14道】

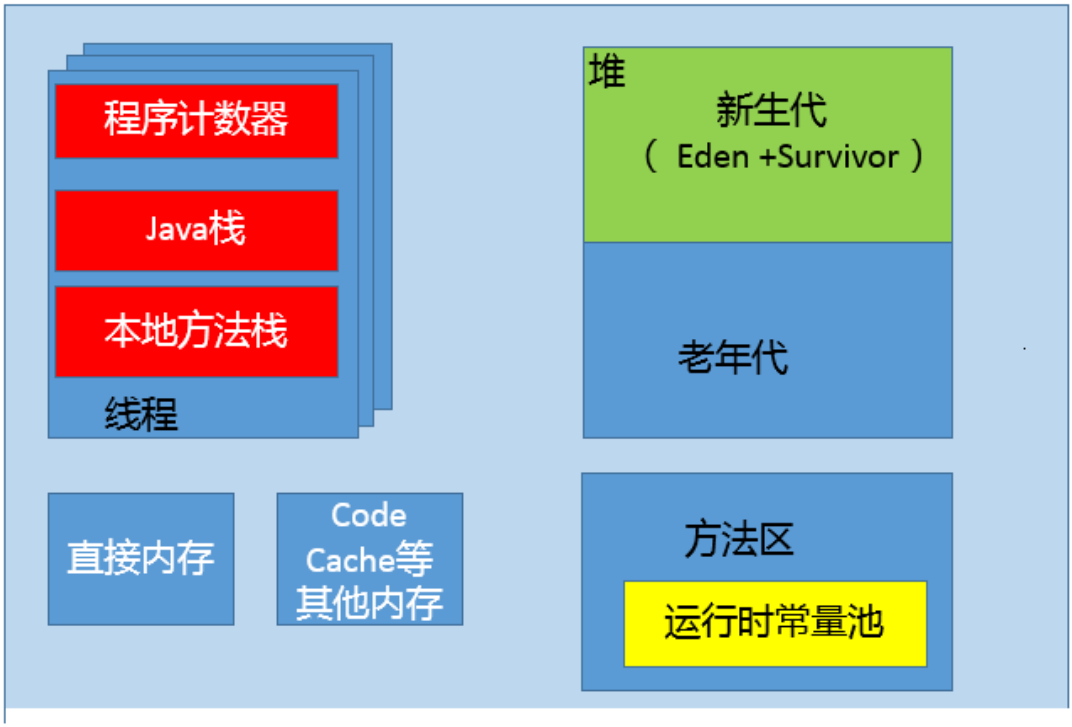
1. JDK1.7、1.8的JVM内存模型，哪些区域可能发生OOM(out of memory)内存溢出!

Out of memory： 堆内存溢出

JDK 1.7：有永久代，但已经把字符串常量池、静态变量，存放在堆上。逐渐的减少永久代的使用。

JDK 1.8：无永久代，运行时常量池、类常量池，都保存在元空间。但字符串常量池仍然存放在堆上。

JVM 内存区域划分图 :



内存溢出通俗的讲就是内存不够用了，并且 GC 通过垃圾回收也无法提供更多的内存。实际上除了程序计数器，其他区域都有可能发生 OOM, 简单总结如下：

【1】堆内存不足是最常见的 OOM 原因之一，抛出错误信息

java.lang.OutOfMemoryError:Java heap space

原因也不尽相同，可能是内存泄漏，也有可能是堆的大小设置不合理。

【2】对于虚拟机栈和本地方法栈，导致 OOM 一般为对方法自身不断的递归调用，且 没有结束点，导致不断的压栈操作。类似这种情况，JVM 实际会抛出 StackOverFlowError , 但是如果 JVM 试图去拓展栈空间的时候，就会抛出 OOM.

【3】对于老版的 JDK, 因为永久代大小是有限的，并且 JVM 对老年代的内存回收非常不积极，所以当我们添加新的对象，老年代发生 OOM 的情况也非常常见。

【4】随着元数据区的引入，方法区内存已经不再那么窘迫，所以相应的 OOM 有所改 观，出现 OOM，异常信息则变成了：“java.lang.OutOfMemoryError: Metaspace”。

2. 请介绍类加载过程和双亲委派模型！

类加载过程 : 加载 -> 连接 -> 验证 -> 准备 -> 解析 -> 初始化

双亲委派模型: 先找父类加载器, 让父类加载器加载, 如果父类加载器无法加载就让子类加载器加载.这个加载过程叫做双亲委派模型或者双亲委派机制.

3. 谈一谈堆和栈的区别！

栈: 存储局部变量, 基本类型数据, 动态链接(堆中对象的内存地址)

堆: 凡是new出来的对象都在堆中, 也就是存储数组和对象,垃圾回收器不定期到堆中收取垃圾.

4. 说一说jvm常见垃圾回收器和特点！

按照内存空间来划分，年轻代的垃圾回收器有：

- Serial：是一个串行的垃圾回收器
- ParNew：是Serial的并行版本
- Parallel Scavenge：也是一个并行的垃圾回收器，区别在于它注重吞吐量

年轻代的三个垃圾回收器采用的垃圾回收算法都是复制算法，所以在进行垃圾回收时都会暂停所有的用户线程；

然后是老年代的垃圾回收器：

- Serial Old：是Serial的老年代版本，串行，采用的是标记-整理算法，同样会STW
- Parallel Old：是Parallel Scavenge的老年代版本，并行，采用的是标记-整理算法，会STW，注重吞吐量
- CMS：注重低延迟，采用的是标记-清除算法，分为四个阶段：初始标记、并发标记、重新标记、并发清除；在初始化和重新标记阶段中是并行的，会STW，其余两个阶段都是并发执行与用户线程同时执行；由于采用标记清理算法，会产生空间碎片

最后是整堆收集器：G1收集器，G1收集器的特点有：能够独立管理整个堆空间、可利用多CPU、多核的硬件优势缩短STW的时间、采用分代收集算法不会产生空间碎片

5. Java创建对象的过程！

对象的创建过程一般是从new指令开始的，JVM首先会对符号引用进行解析，解析完毕后JVM会为对象在堆中分配内存，之后，JVM会将该内存进行零值初始化。最后，JVM会调用对象的构造函数。此时，一般会有一个引用指向这个对象，将对象的地址值赋值给变量。在Java对象初始化过程中，主要涉及三种执行对象初始化的结构，分别是 实例变量初始化、实例代码块初始化 以及 构造函数初始化。

6. Java中垃圾回收机制

- 垃圾回收机制用到finalize。当程序创建对象、数组等引用类型实体时，系统都会在堆内存中为之分配一块内存区，对象就保存在这块内存中，当这块内存不再被任何引用变量引用时，这块内存就会变成垃圾，等待垃圾回收机制进行回收。
- 分析对象是否为垃圾的方法 : 可达性分析法，引用计数法两种
- 强制垃圾回收 : 当一个对象失去引用后，系统何时调用它的finalize ()方法对它进行资源清理，何时它会变成不可达状态，系统何时回收它所占有的内存。对于系统程序完全透明。程序只能控制一个对象任何不再被任何引用变量引用，绝不能控制它何时被回收。强制只是建议系统立即进行垃圾回收，系统完全有可能并不立即进行垃圾回收，垃圾回收机制也不会对程序的建议置之不理:垃圾回收机制会在收到通知后，尽快进行垃圾回收。
- 强制垃圾回收的两个方法 : 调用System类的gc()静态方法System.gc(), 调用Runtime对象的gc()实例方法:Runtime.getRuntime().gc()方法
- 垃圾回收基本算法有四种 : 引用计数法，标记清除法，标记压缩法，复制算法
- 垃圾回收复合算法 – 分代收集算法:

- 当前虚拟机的垃圾收集都采用分代收集算法，这种算法就是根据具体的情况选择具体的垃圾回收算法。一般将java 堆分为新生代和老年代，这样我们就可以根据各个年代的特点选择合适的垃圾收集算法。
- 比如在新生代中，每次收集都会有大量对象死去，所以可以选择复制算法，只需要付出少量对象的复制成本就可以完成每次垃圾收集。而老年代的对象存活几率是比较高的，而且没有额外的空间对它进行分配担保，所以我们必须选择“标记-清除”或“标记-压缩”算法进行垃圾收集。

7. Java中垃圾回收算法

（1）标记-清除算法：使用可达性分析算法标记内存中的垃圾对象，并直接清理，容易造成空间碎片

（2）标记-压缩算法：在标记-清除算法的基础上，将存活的对象整理在一起保证空间的连续性

（3）复制算法：将内存空间划分成等大的两块区域，from和to只用其中的一块区域，收集垃圾时将存活的对象复制到另一块区域中，并清空使用的区域；解决了空间碎片的问题，但是空间的利用率低

（4）复合算法 - 分代收集算法：是对前三种算法的整合，将内存空间分为老年代和新生代，新生代中存储一些生命周期短的对象，使用复制算法；而老年代中对象的生命周期长，则使用标记-清除或标记-整理算法。

8. YoungGC和FullGC触发时机?

- Young GC ： Young GC其实一般就是在新生代的Eden区域满了之后就会触发，采用复制算法来回收新生代的垃圾。
- Full GC ： 调用System.gc()时。系统建议执行Full GC，但是不必然执行
- 触发时机 :
    - 老年代空间不足
    - 方法区空间不足
    - 进入老年代的平均大小大于老年代的可用内存
    - 由Eden区，幸存者0区向幸存者1区复制时，对象大小大于1区可用内存，则把该对象转存到老年代，且老年代的可用内存大小小于该对象大小。
- 注意：full GC是开发或调优中尽量要避免的，这样STW会短一些

9. FullGC触发可能产生什么问题

fullgc的时候除gc线程外的所有用户线程处于暂停状态,也就是不会有响应了。一般fullgc速度很快,毫秒级的,用户无感知。除非内存特别大上百G的,或者fullgc也无法收集到足够内存导致一直fullgc,应用的外在表现就是程序卡死了.

10. jvm调优工具和性能调优思路?

Jvm中调优使用的分析命令如下, 先试用下面命令分析jvm中的问题 :

- Jps : 用于查询正在运行的JVM进程。 直接获取进程的pid
- Jstat : 可以实时显示本地或远程JVM进程中装载，内存，垃圾信息，JIT编译等数据
- Jinfo : 用于查询当前运行着的JVM属性和参数的值
- Jmap : 用于显示当前java堆和永生代的详细信息
- Jhat : 用于分析使用jmap生成的dump文件，是JDK自带的工具
- Jstack: 用于生成当JVM所有线程快照，线程快照是虚拟机每一条线程正在执行的方法，目的是定位线程出现长时间停顿的原因。

下面是JVM常见的调优参数, 经过上面命令分析定位问题后使用下面参数优化:

- -Xmx 指定java程序的最大堆内存
- -Xms 指定最小堆内存
- -Xmn 设置年轻代大小 , 整个堆大小=年轻代大小+年老代大小。所以增大年轻代后，会减小年老代大小。此值对系统影响较大，Sun官方推荐为整个堆的3/8
- -Xss 指定线程的最大栈空间，此参数决定了java函数调用的深度，值越大说明调用深度越深，若值太小则容易栈溢出错误（StackOverflowError）
- -XX: PermSize 指定方法区(永久区)的初始值默认是物理内存的1/64。
- -XX:MetaspaceSize指定元数据区大小, 在Java8中，永久区被移除，代之的是元数据区
- -XX:NewRatio=n 年老代与年轻代的比值，-XX:NewRatio=2,表示年老代和年轻代的比值为2:1
- -XX:SurvivorRatio=n Eden区与Survivor区的大小比值，-XX:SurvivorRatio=8表示Eden区与Survivor区的大小比值是8:1:1,因为Survivor区有两个(from,to);

11. Jvm内存模型?

注意 : 大多数面试官混淆了内存模型和内存结构的区别, 如果面试官这么问, 可以咨询下面试官是否问的是jvm内存的结构都包含哪些东西, 还是问JMM(Java memory model)内存模型原理

Jvm内存结构由9块组成:

- 类加载器, 垃圾回收器, 执行引擎, 方法区，堆，栈，直接内存, 本地方法栈, 程序计数器(pc寄存器)
- JMM(java memory model)java内存模型:

  JMM规定了所有变量（除了方法参数和本地变量，包括实例变量和静态变量）都放在主内存中。每个线程都有自己的工作内存，工作内存保存了该线程使用的主内存的变量副本，所有的操作都在工作内存中进行，线程不能直接操作主内存。线程之间通过将数据刷回主内存的方式进行通信。
- JMM定义了原子性，可见性和有序性。

  原子性：一个操作不可分割，不可中断，不可被其他线程干扰。JMM提供moniterenter和monitereixt俩个字节码指令保证代码块的原子性
  可见性：当一个变量被修改后，其他线程能够立即看到修改的结果
  有序性：禁止指令重排序

12. GC如何识别垃圾?

- 引用计数算法(已被淘汰的算法)

  给对象中添加一个引用计数器,每当有一个地方引用它时,计数器值就加1;当引用失效时,计数器值就减1;任何时刻计数器为0的对象就是不可能再被使用的。
- 可达性分析算法

  目前主流的编程语言(java,C#等)的主流实现中,都是称通过可达性分析(Reachability Analysis)来判定对象是否存活的。这个算法的基本思路就是通过一系列的称为“GC Roots”的对象作为起始点,从这些节点开始向下搜索,搜索所走过的路径称为引用链(Reference Chain),当一个对象到GC Roots没有任何引用链相连, 就是从GC Roots到这个对象不可达时,则证明此对象是不可用的。

13. 详细介绍一下 CMS 垃圾回收器?

CMS 是英文 Concurrent Mark-Sweep 的简称，是以牺牲吞吐量为代价来获得最短回收停顿时间的垃圾回收器。对于要求服务器响应速度的应用上，这种垃圾回收器非常适合。

在启动 JVM 的参数加上“-XX:+UseConcMarkSweepGC”来指定使用 CMS 垃圾回收器。

CMS 使用的是标记-清除的算法实现的，所以在 gc 的时候会产生大量的内存碎片，当剩余内存不能满足程序运行要求时，系统将会出现 Concurrent Mode Failure，临时 CMS 会采用 Serial Old 回收器进行垃圾清除，此时的性能将会被降低。

CMS 工作机制相比其他的垃圾收集器来说更复杂，整个过程分为以下 4 个阶段：

（1）初始标记

只是标记一下 GC Roots 能直接关联的对象，速度很快，仍然需要暂停所有的工作线程。

（2）并发标记

进行 GC Roots 跟踪的过程，和用户线程一起工作，不需要暂停工作线程。

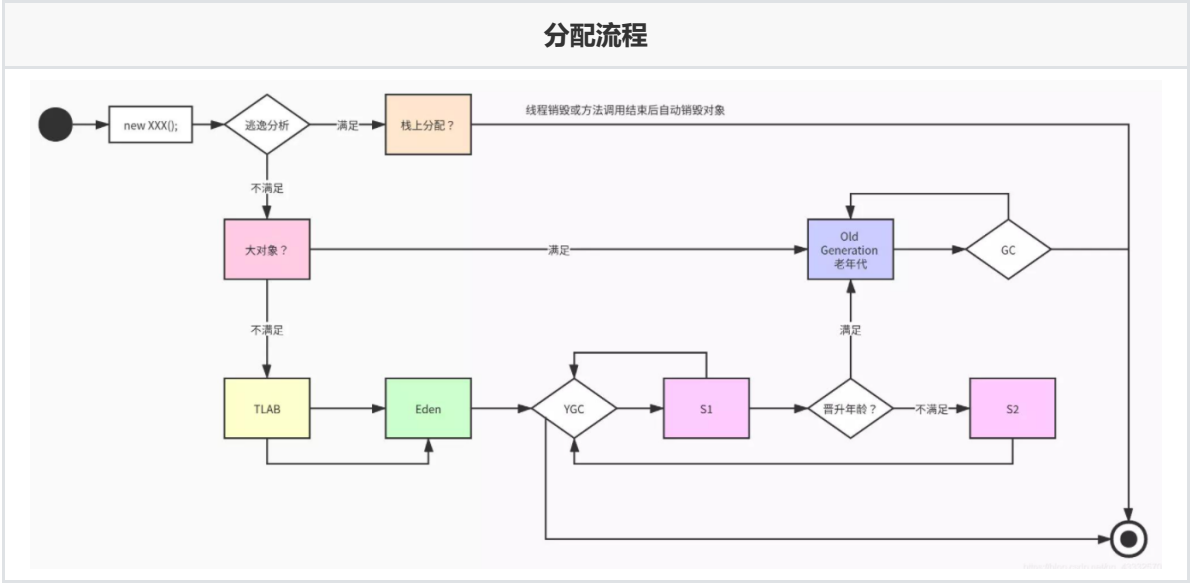
（3）重新标记

为了修正在并发标记期间，因用户程序继续运行而导致标记产生变动的那一部分对象的标记记录，仍然需要暂停所有的工作线程。

（4）并发清除

清除 GC Roots 不可达对象，和用户线程一起工作，不需要暂停工作线程。由于耗时最长的并发标记和并发清除过程中，垃圾收集线程可以和用户现在一起并发工作， 所以总体上来看CMS 收集器的内存回收和用户线程是一起并发地执行。

14. Java创建对象的内存分配流程?



三、集合相关面试题【17道】

1. 聊一聊Java中容器体系结构！

- Collection：集合接口
  - List：是Collection的子接口，用来存储有序的数据集合
  - ArrayList:是List集合的实现类，底层采用动态数组进行存储数据，默认是空数组，如果存值则扩容至10，如果不够则以1.5倍进行扩容。
  - LinkedList:是List集合的实现类，底层采用双向链表结构进行存储数据，增删数据比较方便，速度快。
  - Vector :Vector类实现了可扩展的对象数组, 像数组一样，它包含可以使用整数索引访问的组件。但是，Vector的大小可以根据需要增长或缩小，以适应在创建Vector之后添加和删除。

  【注】Vector是同步的（线程安全）。 如果不需要线程安全的实现，建议使用ArrayList代替Vector.

- Set: 是Collection的子接口，用来存储无序的数据集合
  - HashSet:底层采用哈希表(HashMap)的方式存储数据，数据无序且唯一
  - TreeSet：采用有序二叉树进行存储数据，遵循了自然排序。

- Map: 与Collection并列，用来存储具有映射关系(key-value)的数据
  - HashMap:哈希表结构存储数据，key值可以为null
  - TreeMap:红黑树算法的实现
  - HashTable:哈希表实现，key值不可以为null
  - Properties:HashTable的子类，键值对存储数据均为String类型，主要用来操作以.properties结尾的配置文件。

【特点】存储数据是以key，value对进行存储，其中key值是以Set集合形式存储，唯一且不可重复。value是以Collection形式存储，可以重复。

2. List、Set和Map的区别，以及各自的优缺点

- List :
  - 可以允许重复的对象。可以插入多个null元素。是一个有序容器，保持了每个元素的插入顺序，输出的顺序就是插入的顺序。
  - 常用的实现类有 ArrayList、LinkedList 和 Vector。ArrayList 最为流行，它提供了使用索引的随意访问，而 LinkedList 则对于经常需要从 List 中添加或删除元素的场合更为合适。
- Set :
  - 不允许重复对象。无序容器，你无法保证每个元素的存储顺序，TreeSet通过 Comparator 或者 Comparable 维护了一个排序顺序。只允许一个 null 元素。
  - Set 接口最流行的几个实现类是 HashSet、LinkedHashSet 以及 TreeSet。最流行的是基于 HashMap 实现的 HashSet；TreeSet 还实现了 SortedSet 接口，因此 TreeSet 是一个根据其 compare() 和 compareTo() 的定义进行排序的有序容器。而且可以重复。
- Map :
  - Map不是collection的子接口或者实现类。Map是一个接口。
  - Map 的 每个 Entry 都持有两个对象，也就是一个键一个值，Map 可能会持有相同的值对象但键对象必须是唯一的。
  - TreeMap 也通过 Comparator 或者 Comparable 维护了一个排序顺序。
  - Map 里你可以拥有随意个 null 值但最多只能有一个 null 键。
  - Map 接口最流行的几个实现类是 HashMap、LinkedHashMap、Hashtable 和 TreeMap。（HashMap、TreeMap最常用）

3. ArrayList，LinkedList，Vector的区别！

- ArrayList和Vector是基于数组实现的。 ArrayList线程不安全,速度快, Vector线程安全,速度慢, 因为底层数组实现所以查询快, 修改快, 添加和删除慢,
- LinkedList是基于双向链表实现的, 线程不安全, 添加, 删除快, 查询和修改慢.
- ArrayList和Vector都是使用Object的数组形式来存储的，当向这两种类型中增加元素的时候，若容量不够，需要进行扩容。ArrayList扩容后的容量是之前的1.5倍，然后把之前的数据拷贝到新建的数组中去。而Vector默认情况下扩容后的容量是之前的2倍。
- Vector可以设置容量增量，而ArrayList不可以。

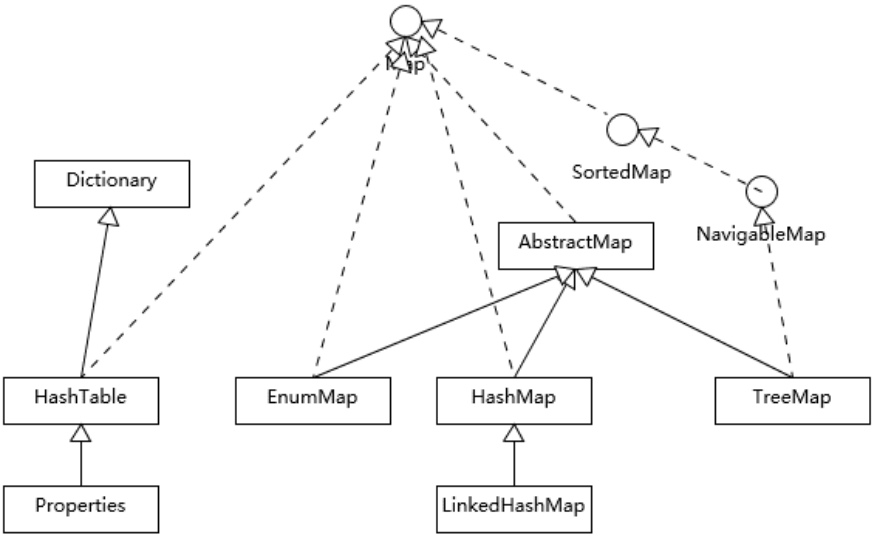
4. TreeSet如何保证对象的有序性！

- TreeSet是有序不可重复集，具有以下特点：
- 数据会按自然排序（可以理解为从小到大排序）
- 不可存储null
- 数据不可重复
- 非线程安全
- TreeSet底层依赖于TreeMap，TreeMap的数据结构是二叉树（红黑树），由底层数据结构决定了TreeSet中元素有序且唯一。
- 我们使用TreeSet时，如果添加对象的话，我们就必须去实现Comparable接口，当然也可以实现Comparator接口来保证对象的有序性。

5. HashTable、HashMap、TreeMap的区别！

- Hashtable、HashMap、TreeMap都实现了Map接口，使用键值对的形式存储数据和操作数据。
- Hashtable是java早期提供的，方法是同步的（加了synchronized）。key和value都不能是null值。
- HashMap的方法不是同步的，支持key和value为null的情况。行为上基本和Hashtable一致。由于Hashtable是同步的，性能开销比较大，一般不推荐使用Hashtable。通常会选择使用HashMap。
- HashMap进行put和get操作，基本上可以达到常数时间的性能
- TreeMap是基于红黑树的一种提供顺序访问的Map，和HashMap不同，它的get或put操作的时间复杂度是O(log(n))。具体的顺序由指定的Comparator来决定，或者根据键key的具体顺序来决定。

关系图：



6. HashMap的底层数据存储结构

JDK1.7采用 : 数组 + 链表结构

JDK1.8采用 : 数组 + 链表 + 红黑树

红黑树：jdk1.8最重要的就是引入了红黑树的设计，当hash表的单一链表长度超过 8 个的时候，链表结构就会转为红黑树结构。

HashMap的底层数据存储结构为哈希表加红黑树。实现过程如下：

- 调用HashMap的无参构造方法，加载因子loadFactor赋值0.75，table数组是空。
- 当添加第一个元素时，创建长度16的数组，threshold=12。
- 当'链表长度大于等于8时，并且数组长度大于等于64时，链表调整红黑树
- 当红黑树的节点个数小于6时，调整为链表
- 当HashMap的容量超出阈值时，扩容为原来大小的2倍，减少元素的移动，提高效率。

7. HashMap的扩容机制

- 为什么会需要扩容:
    - 当它在初始化时会创建一个容量为16的数组，当元素数量超过阈值（当前容量X加载因子(通常为0.75)=扩容阈值）时，会将数组大小扩展为当前容量的两倍。
    - 但是HashMap的容量是有上限的。如果hashmap的数组当前容量达到了1073741824，则该数组不会再增长。且阈值将被设置为Integer.MAX_VALUE（2^31-1）即永远不会超出阈值。
- Hashmap的扩容机制在JDK8中，容量变化通常有以下集中情况：
    - 无参构造：实例化的hashmap默认内部数组是null，也就是无实例化。第一次调用put方法时，会开始第一次初始化扩容，长度为16.
    - 有参构造：用于指定容量。会根据指定的正整数找到不小于指定容量的2的幂数，将这个数设置赋值给阈值。第一次调用put方法时，会将阈值赋值给容量，然后让 阈值=容量X负载因子。因此并不是手动指定了容量就一定不会触发扩容，超过阈值后一样扩容。
- 如果不是第一次扩容，则容量变为原来的2倍，阈值也变为原来的2倍。但负载因子不变。
- 补充：

    - 首次put时会以扩容的方式初始化，然后存入数据，之后判断是否需要扩容
    - 如果不是首次put，则不咋需要初始化，会直接存入数据，然后判断是否需要扩容。
- 元素迁移:
    - 在JDK8中，由于数组的容量是以2的幂次方扩容的，那么一个数组在扩容时，一个元素要么在原位置要么在原长度+原位置的位置。
    - 数组长度变为原来的2倍表现在二进制上多了一个高位参与数组下标确定。此时，一个元素通过hash转换坐标的方法计算后会出现一个现象：
    - 最高位是0，则坐标不变，最高位是1则坐标变为10000+原坐标，即原长度+原坐标。

8. HashMap链表和红黑树转化时机！

HashMap中维护有一个Node类型的数组table，当新添加的元素的hash值(hashCode & (length - 1))所指向的位置已有元素时就会被挂在该位置的最后面而形成链表。当在插入一个元素后某一位置的链表长度大于等于树化的阈值（TREEIFY_THRESHOLD = 8）时，会先去检查table的size，如果此时table数组的长度小于最小树化容量（MIN_TREEIFY_CAPACITY = 64），会先对数组进行扩容，扩容大小为原来的二倍。如果此时数组的长度已经大于64，则会将链表转化为红黑树。在扩容时，如果发现发现某处已经变为红黑树的树上的节点数量总和小于等于红黑树转化为链表的阈值（UNTREEIFY_THRESHOLD = 6），则会将此处的红黑树转化为链表。

简而言之，当链表长度超过8且哈希表容量大于64，会将链表转化为红黑树。而当红黑树的大小小于6，则由树转化为链表。

9. 什么是Hash碰撞以及常见的解决方案

Hash碰撞 : 对象Hash的前提是实现equals()和hashCode()两个方法，那么HashCode()的作用就是保证对象返回唯一hash值，但当两个不同对象计算值一样时，这就发生了碰撞冲突。

解决方案：

- 开放地址法(再散列法)开放地执法有一个公式:Hi=(H(key)+di) MOD m i=1,2,...,k(k<=m-1)

  其中，m为哈希表的表长。di 是产生冲突的时候的增量序列。如果di值可能为1,2,3,...m-1，称线性探测再散列。

  如果di取1，则每次冲突之后，向后移动1个位置.如果di取值可能为1,-1,2,-2,4,-4,9,-9,16,-16,...kk,-kk(k<=m/2)，称二次探测再散列。如果di取值可能为伪随机数列。称伪随机探测再散列。

- 再哈希法Rehash当发生冲突时，使用第二个、第三个、哈希函数计算地址，直到无冲突时。

  缺点：计算时间增加。

  比如上面第一次按照姓首字母进行哈希，如果产生冲突可以按照姓字母首字母第二位进行哈希，再冲突，第三位，直到不冲突为止.这种方法不易产生聚集，但增加了计算时间。

- 链地址法（拉链法）将所有关键字为同义词的记录存储在同一线性链表中.基本思想:将所有哈希地址为i的元素构成一个称为同义词

  链的单链表，并将单链表的头指针存在哈希表的第i个单元中，因而查找、插入和删除主要在同义词链中进行。链地址法适用于经常进行插入和删除的情况。

10. HashMap在多线程操作中会出现什么问题!

会出现线程安全问题, 可以使用加锁解决, 但是速度慢, 所以建议使用juc包下的ConcurrentHashMap替代HashMap, 速度快且线程安全.

11. 如何保证集合是线程安全的?

第一: 使用JUC包下的ConcurrentHashMap, 线程安全, 并且速度也快

第二: 选用synchronize或者Lock加锁解决, 速度慢.

12. 聊一聊JUC包下的并发工具

CountDownLatch 闭锁,使用时需要传入一个int类型的参数，一个线程等待一组线程执行完毕后再恢复执行；await() 等待其他线程都执行完毕，通过计数器来判断等待的线程是否全部执行完毕。计数器：countDown方法，被等待线程执行完毕后将计数器值-1，当CountDownLatch的值减为0时无法恢复，这就是叫做闭锁的原因。

CyclicBarrier 循环栅栏, 一组线程 同时到达临界点后再同时恢复执行(先到达临界点的线程会阻塞，直到所有线程都到达临界点)，当多个线程同时到达临界点时，随机挑一个线程执行barrierAction后再同时恢复执行。await()：调用该方法时表示线程已经到达屏障，随即阻塞。模拟：多线程向磁盘写入数据，计数器的值可以恢复。

SemaPhore-信号量：SemaPhore是synchronized或Lock的加强版，作用是控制线程的并发数量。作用：用来控制同时访问特定资源的线程数量，通过协调保证合理的使用公共资源。acquire():尝试占用一个信号量，失败的线程会阻塞，直达有新的信号量，再恢复执行release():释放一个信号量；acquire(n):尝试占用n信号量，失败的线程会阻塞，直达有新的信号量，再恢复执行，release(n):释放n信号量。

Exchanger 线程交换器，Exchange类似于一个交换器，Exchange类允许在两个线程之间定义同步点。当两个线程都到达同步点时，他们交换数据，因此第一个线程的数据进入到第二个线程中，第二个线程的数据进入到第一个线程中。

13. ConcurrentHashMap如何保证线程安全

底层采用CAS(内存比较交换技术) + volatile + synchronize实现, 初始化为无锁状态也就是使用CAS + volatile解决安全问题, 但是会涉及ABA问题. 但是几率很小, 如果涉及ABA问题, 底层自动切换成使用synchronize实现.

14. ConcurrentHashMap在1.7和1.8的底层实现

JDK1.7底层采用 : 数组 + 链表，采用Segement保证安全

JDK1.8底层采用: 数据 + 链表 + 红黑树，采用CAS+synchronized代码块保证安全

15. ConcurrentLinkedQueue和LinkedBlockingQueue有什么区别

首先二者都是线程安全的得队列，都可以用于生产与消费模型的场景。

LinkedBlockingQueue是阻塞队列，其好处是：多线程操作共同的队列时不需要额外的同步，由于具有插入与移除的双重阻塞功能，对插入与移除进行阻塞，队列会自动平衡负载，从而减少生产与消费的处理速度差距。

由于LinkedBlockingQueue有阻塞功能，其阻塞是基于锁机制实现的，当有多个线程消费时候，队列为空时消费线程被阻塞，有元素时需要再唤醒消费线程，队列元素可能时有时无，导致用户态与内核态切换频繁，消耗系统资源。从此方面来讲，LinkedBlockingQueue更适用于多线程插入，单线程取出，即多个生产者与单个消费者。

ConcurrentLinkedQueue非阻塞队列，采用 CAS+自旋操作，解决多线程之间的竞争，多写操作增加冲突几率，增加自选次数，并不适合多写入的场景。当许多线程共享访问一个公共集合时，ConcurrentLinkedQueue 是一个恰当的选择。从此方面来讲，ConcurrentLinkedQueue更适用于单线程插入，多线程取出，即单个生产者与多个消费者。

总之，对于几个线程生产与几个线程消费，二者并没有严格的规定， 只有谁更适合。

16. 说一下 HashSet 的实现原理

HashSet底层使用了哈希表来支持的，特点：存储快, 底层由HashMap实现

往HashSet添加元素的时候，HashSet会先调用元素的hashCode方法得到元素的哈希值 ，然后通过元素 的哈希值经过移位等运算，就可以算出该元素在哈希表中的存储位置。

如果算出的元素存储的位置目前没有任何元素存储，那么该元素可以直接存储在该位置上

如果算出的元素的存储位置目前已经存在有其他的元素了，那么还会调用该元素的equals方法与该位置的元素再比较一次，如果equals方法返回的是true，那么该位置上的元素视为重复元素，不允许添加，如果返回的是false，则允许添加

17. HashSet和TreeSet有什么区别?

HashSet是由一个hash表来实现的，因此，它的元素是无序的。add()，remove()，contains()方法的时间复杂度是O(1)。

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)。

四、多线程 【25道】

1. 聊一聊并行和并发的区别?

- 并发是指同一时间一起发生的.

  例如: 一个处理器同时处理多个任务叫并发处理, 同一时间好多个请求一起访问你的网站叫做并发请求等

- 并行是指多个任务在同一时间一起运行.

  例如: 多个处理器或者是多核的处理器同时处理多个不同的任务, 这叫并行执行.

2. 线程和进程的区别?

进程是执行中的一个程序, 而一个进程中执行的一个任务即为一个线程

一个线程只属于一个进程, 但一个进程能包含多个线程

线程无地址空间, 它包括在进程的地址空间中

线程的开销比进程小

3. 说一说什么是原子性、可见性、有序性?

原子性：提供互斥访问，同一时刻只能有一个线程对数据进行操作（Atomic、CAS算法、synchronized、Lock）

可见性：一个主内存的线程如果进行了修改，可以及时被其他线程观察到（synchronized、volatile）

有序性：如果两个线程不能从 happens-before原则 观察出来，那么就不能观察他们的有序性，虚拟机可以随意的对他们进行重排序，导致其观察结果杂乱无序（happens-before原则）

4. Java中实现多线程的方式

- 通过扩展Thread类来创建多线程:

  定义Thread类的子类，并重写该类的run方法，该run方法的方法体就代表了线程要完成的任务。

- 通过实现Runnable接口来创建多线程

  定义runnable接口的实现类，并重写该接口的run()方法，该run()方法的方法体同样是该线程的线程执行体

- 通过线程池实现

  定义线程池实例, 使用的时候从线程池中获取线程使用.

- 通过Callable和Future创建线程

  创建Callable接口的实现类，并实现call()方法，该call()方法将作为线程执行体，并且有返回值

  创建Callable实现类的实例，使用FutureTask类来包装Callable对象，该FutureTask对象封装了该Callable对象的call()方法的返回值。

  使用FutureTask对象作为Thread对象的target创建并启动新线程。

  调用FutureTask对象的get()方法来获得子线程执行结束后的返回值

5. Java中提供的线程池种类，及其特点!

五种线程池

- Single Thread Executor : 这是一个单线程的Executor，它创建单个工作线程来执行任务，如果这个线程异常结束，会创建一个新的来替代它；它的特点是能确保依照任务在队列中的顺序来串行执行

  代码： Executors.newSingleThreadExecutor()

- Cached Thread Pool : 创建一个可缓存的线程池，如果线程池的规模超过了处理需求，将自动回收空闲线程，而当需求增加时，则可以自动添加新线程，线程池的规模不存在任何限制。

  代码：Executors.newCachedThreadPool()

- Fixed Thread Pool : 拥有固定线程数的线程池，如果没有任务执行，那么线程会一直等待， 代码： Executors.newFixedThreadPool(4) 在构造函数中的参数4是线程池的大小，你可以随意设置，也可以和cpu的核数量保持一致，获取cpu的核数量

  代码 : int cpuNums = Runtime.getRuntime().availableProcessors();

- Scheduled Thread Pool : 创建一个固定长度的线程池，而且以延迟或定时的方式来执行任务，类似于Timer。

  代码：Executors.newScheduledThreadPool()

6. 线程池的工作原理以及重要参数含义

- 原理:

  线程池会初始化一定数量线程, 每次使用线程从线程池中拿一个使用, 省去了创建线程的开销和时间, 使用完毕, 放回线程池中, 不销毁, 省去了销毁的开销和时间.

- 重要参数:
  - corePoolSize：线程池核心线程数量
  - maximumPoolSize:线程池最大线程数量
  - keepAliverTime：当活跃线程数大于核心线程数时，空闲的多余线程最大存活时间
  - unit：存活时间的单位
  - workQueue：存放任务的队列
  - threadFactory：创建线程的工厂
  - handler：拒绝策略

7. 线程池的阻塞队列有哪些?

ArrayBlockingQueue : 数组有界阻塞队列FIFO, 按照阻塞的先后顺序访问队列，默认情况下不保证线程公平的访问队列, 如果要保证公平性，会降低一定的吞吐量

LinkedBlockingQueue : 链表有界阻塞队列，默认最大长度为Integer.MAX_VALUE。此队列按照先进先出的原则对元素进行排序SynchronousQueue : 不存储元素的阻塞队列

DelayedWorkQueue : 延迟获取元素队列，指定时间后获取，为无界阻塞队列。

8. 线程池的拒绝策略有哪些?

- 拒绝策略核心接口:

  java.util.concurrent.RejectedExecutionHandler

- 实现:
  - CallerRunsPolicy : java.util.concurrent.ThreadPoolExecutor.CallerRunsPolicy

    当有新任务提交后，如果线程池没被关闭且没有能力执行，则把这个任务交于提交任务的线程执行，也就是谁提交任务，谁就负责执行任务。

  - AbortPolicy : java.util.concurrent.ThreadPoolExecutor.AbortPolicy

    拒绝策略在拒绝任务时，会直接抛出一个类型为 RejectedExecutionException 的 RuntimeException，让你感知到任务被拒绝了，于是你便可以根据业务逻辑选择重试或者放弃提交等策略。

  - DiscardPolicy : java.util.concurrent.ThreadPoolExecutor.DiscardPolicy

    当新任务被提交后直接被丢弃掉，也不会给你任何的通知，相对而言存在一定的风险，因为我们提交的时候根本不知道这个任务会被丢弃，可能造成数据丢失。

  - DiscardOldestPolicy : java.util.concurrent.ThreadPoolExecutor.DiscardOldestPolicy

    如果线程池没被关闭且没有能力执行，则会丢弃任务队列中的头结点，通常是存活时间最长的任务，这种策略与第二种不同之处在于它丢弃的不是最新提交的，而是队列中存活时间最长的，这样就可以腾出空间给新提交的任务，但同理它也存在一定的数据丢失风险。

9. synchronized和reentranlock锁的区别?

Synchronized属于重量级锁, JDK早期版本使用了线程的状态变化来实现, JDK1.8中使用了自适应自旋锁实现. 总体来说安全, 但是效率慢

Reentranlock是JUC包下的锁, 底层使用CAS + Volatile关键字实现. 效率高.

10. synchronized底层如何实现？什么是锁升级、降级！

jdk 中做了优化，提供了三种不同的 Monitor实现，分别是：

- 偏斜锁 （Biased Locking）
- 轻量级锁
- 重量级锁

所谓锁的升级，降级，实际上是 JVM 对 synchronized优化的一种策略，JVM 会检测不同的竞争状态，然后自动切换到合适的锁实现，这种切换就是锁的升级，降级。

当没有出现锁的竞争时，默认使用的是偏斜锁。JVM 会利用 CAS 实现.

如果有另一个线程试图锁定某个以被加持偏斜锁的对象时，JVM 就需要撤销偏斜锁，并切换到轻量级锁实现。如果获取成功，就使用轻量级锁，否者，进一步升级到重量级锁

11. reentranlock的底层实现原理

reentranlock 是基于AQS(AbstractQueueSynchronizer) 采用FIFO的队列表示排队等待的线程.

AQS底层采用CAS(内存比较交换技术) + Volatile关键字实现.

12. volatile关键字的特点

- 保证线程之间的可见性，当一个线程对共享的变量进行了修改，其他的线程能够通过此关键字发现这个修改
- 禁止指令重排序，编译器在编译的过程中会对程序进行优化，在保证结果不变的前提下，调整指令执行的顺序，提高执行效率，如果加了volatile关键字，则会禁止指令重排序。
- 不能保证原子性

13. Java为什么会指令重排

java中源代码文件会被编译成.class的字节码文件, 字节码指定在执行之前, jvm底层有内置的对字节码的优化策略, 也就是指令重排机制, 会调整指令执行顺序. 目的是加快执行速度.

14. 悲观锁和乐观锁的区别

悲观锁 : 无论读还是更改对象都要加锁, 所以慢, 但是安全.

乐观锁 : 读不加锁, 更改对象加锁, 所以快, 但是没有悲观锁安全

15. 什么是CAS, 以及它的ABA问题, 如何解决ABA问题

- CAS的含义：

  CAS是compare and swap的缩写，即我们所说的比较交换。

  CAS 操作包含三个操作数 —— 内存位置（V）、预期原值（A）和新值(B)。如果内存地址里面的值和A的值是一样的，那么就将内存里面的值更新成B。CAS是通过无限循环来获取数据的，若果在第一轮循环中，a线程获取地址里面的值被b线程修改了，那么a线程需要自旋，到下次循环才有可能机会执行。

- CAS的ABA问题：

  CAS容易造成ABA问题。一个线程a将数值改成了b，接着又改成了a，此时CAS认为是没有变化，其实是已经变化过了，而这个问题的解决方案可以使用版本号标识，每操作一次version加1。在java5中，已经提供了AtomicStampedReference来解决问题。

  CAS造成CPU利用率增加。之前说过了CAS里面是一个循环判断的过程，如果线程一直没有获取到状态，cpu资源会一直被占用。

16. Atomic变量如何保证的原子性

它底层是使用CAS + volatile关键字来保证原子性操作，从而达到线程安全的目的.

17. ThreadLocal是什么，有什么作用

ThreadLocal是一个本地线程副本变量工具类。主要用于将私有线程和该线程存放的副本对象做一个映射，各个线程之间的变量互不干扰，在高并发场景下可以实现无状态的调用，特别适用于各个线程依赖不通 的变量值完成操作的场景。

简单说ThreadLocal就是一种以空间换时间的做法，在每个Thread里面维护了一个以开地址法实现的ThreadLocal.ThreadLocalMap,把数据进行隔离，数据不共享，自然就没有线程安全的问题了。

18. ThreadLocal的内存泄漏问题

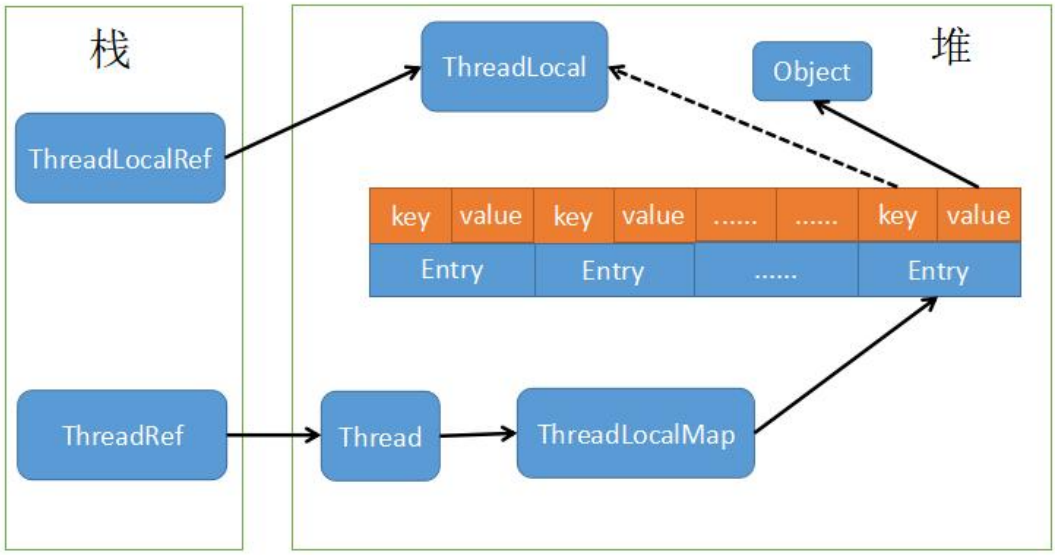
```
static class ThreadLocalMap {
    static class Entry extends WeakReference<ThreadLocal<?>> {
        /** The value associated with this ThreadLocal. */
        Object value;

        Entry(ThreadLocal<?> k, Object v) {
            super(k);
            value = v;
        }
    }
    // 。。。。
}
```

`ThreadLocalMap` 中的 `key` 是一个 `ThreadLocal` 对象，且是一个弱引用，而 `value` 却是一个强引用。

存在一种情况，可能导致内存泄漏。如果在某一时刻，将 `ThreadLocal` 实例设置为 `null`，即 `ThreadLocal` 没有强引用了，如果发生 GC 时，由于 `ThreadLocal` 实例只存在弱引用，所以被回收了，但是 `value` 仍然存在一个当前线程连接过来的强引用，其不会被回收，只有等到线程结束死亡或者手动清空 `value` 或者等到另一个 `ThreadLocal` 对象进行 `get` 或 `set` 操作时刚好触发

`expungeStaleEntry` 函数并且刚好能够检查到本 `ThreadLocal` 对象 `key` 为空（概率太小），这样才不会发生内存泄漏。否则，`value` 始终有引用指向它，它也不会被 GC 回收，那么就会导致内存泄漏。虽然发生内存泄漏的概率比较小，但是为了保险起见，也建议在使用完 `ThreadLocal` 对象后调用一下 `remove` 方法清理一下值。
作者：guozhchun

ThreadLocal的内存泄漏问题



19. jdk1.8对锁进行了哪些优化

LongAdder 类似automicLong, 但是提供了“热点分离”。过程如下：如果并发不激烈，则与automicLong 一样，cas赋值。如果出现并发操作，则使用数组，数组的各元素之和为真实value值，让操作分散在数组各个元素上，把并发操作压力分散，一遇到并发就扩容数组，最后达到高效率。一般cas如果遇到高并发，可能一直赋值失败导致不断循环，热点分离可以解决这个问题

stampedLock 改进读写锁，读不阻塞写。

completableFuture 对Future进行增强，支持函数式编程的流式调用

20. 死锁发生的情况和如何避免死锁

死锁就是多个线程同时等待其他线程释放锁，导致被无限期阻塞的现象。

- 发生死锁的四个必要条件：
  - 互斥条件
  - 不可抢占条件
  - 占有且申请条件
  - 循环等待条件
- 避免死锁：
  - 尽量避免一个线程同时获取多个锁
- 尽量避免一个线程在锁内部占用多个资源，尽量保证每个锁只占用一个资源
  - 顺序加锁
- 加锁时限

- 死锁检验

21. 介绍锁的升级过程

锁状态一种有四种 : 从级别由低到高依次是：无锁、偏向锁，轻量级锁，重量级锁，

锁状态只能升级，不能降级

升级过程 :

无锁状态 - > 当一个线程访问同步代码块时升级成偏向锁 - > 偏向锁 -> 有锁竞争时升级成轻量级锁 - >

轻量级锁 - > 自旋N次失败, 锁膨胀, 升级成重量级锁 - > 重量级锁

22. 介绍锁的降级过程

- 锁降级

  锁降级发生在读写锁中，写锁降级读锁的过程
- 读写锁 : ReentrantReadWriteLock
  读写锁，既可以获取读锁，也可以获取写锁
  - 写锁是独占锁，所谓独占即为独自占有，别的线程既不能获取到该锁的写锁，也不能获取到对应的读锁。
  - 读锁是共享锁，所谓共享即是所有线程都可以共同持有该读锁
- 锁降级过程 :

  锁降级指的是写锁降级为读锁的过程，他的过程是持有写锁，获取读锁，然后释放写锁

23. 怎么解决多线程的大量访问时的数据同步

可以加锁解决, 至于锁可以使用synchronized重量级锁, 也可以使用Lock轻量级锁

还可以使用线程安全的对象作为多线程共用的数据操作对象, 比如ConcurrentHashMap, 或者Atomic原子操作类等

速度快慢则是, 线程安全操作对象ConcurrentHashMap或者原子操作类比轻量级锁快, 轻量级锁比重量级锁要快.

24. 线程的 run()和 start()有什么区别

每个线程都是通过某个特定Thread对象所对应的run()方法来完成其操作的，方法run()称为线程体。也就是run方法中是线程具体要执行的任务或者业务.

Start方法是启动线程, 调用线程类的Start方法线程开始运行. 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, 真正实现了多线程运行

25. JDK1.6对Synchronized底层做了哪些优化

Synchronized底层早期JDK版本是采用线程的状态转换实现的, 主要使用了线程的阻塞和唤醒来实现.

JDK1.6中使用了自适应自旋锁实现, 还加入的锁消除, 锁粗化等优化策略

自适应自旋锁 : JDK 1.6引入了更加聪明的自旋锁，即自适应自旋锁。所谓自适应就意味着自旋的次数不再是固定的，它是由前一次在同一个锁上的自旋时间及锁的拥有者的状态来决定。

锁消除 : 为了保证数据的完整性，我们在进行操作时需要对这部分操作进行同步控制，但是在有些情况下，JVM检测到不可能存在共享数据竞争，这是JVM会对这些同步锁进行锁消除

锁粗化 : 就是将多个连续的加锁、解锁操作连接在一起，扩展成一个范围更大的锁

五、IO【5道】

1. Java提供了哪些IO方式?

传统BIO同步阻塞IO, 这里面又分为字节流和字符流

JDK1.4引入NIO, 同步非阻塞IO, 速度比传统BIO快, 并且更节省资源

JDK1.7引入NIO2也叫做AIO, 异步非阻塞IO, 基于事件和回调机制实现, 速度更快

2. NIO, BIO, AIO区别?

NIO：在JDK1.4以前，Java的IO模型一直是BIO，从JDK1.4开始，JDK引入的新的IO模型NIO，它是同步非阻塞的。而服务器的实现模式是多个请求一个线程，即请求会注册到多路复用器Selecter上，多路复用器轮询到连接有IO请求时才启动一个线程处理

BIO：同步阻塞，服务器的实现模式是一个连接一个线程，这样的模式很明显的一个缺陷是：优于客户端连接数与服务器线程数成正比关系，可能造成不必要的线程开销，严重的还会导致服务器内存溢出，当然，这种情况可以通过线程池改善，但并不能从本质上消除这个弊端

AIO：JDK1.7发布了NIO2.0，这就是真正意义上的异步非阻塞，服务器的实现模式为多个有效请求一个线程，客户端的IO请求都是由OS完成再通知服务器应用去启动线程处理（回调）

3. 有哪些缓冲流？如何实现缓冲功能!

缓冲流也叫高效流，是对四个基本的FileXxx流的增强，按照数据类型分类：

- 字节缓冲流 ： BufferedInputStream，BufferedOutputStream
- 字符缓冲流：BufferedReader，BufferedWriter

基本原理：

- 是在创建流对象的时候，会创建一个内置默认大小的缓冲区数组，减少系统IO次数，从而提高读写效率

字节缓冲流

- public BufferedInputStream(InputStream in)： 创建一个 新的缓冲输入流。
- public BufferedOutputStream(OutputStream out)： 创建一个新的缓冲输出流。

4. 实现文件拷贝的几种方式!

通过字节流实现文件拷贝

通过字符流实现文件拷贝

通过字节缓冲流实现文件拷贝

通过字符缓冲流实现文件拷贝

通过JAVA NIO非直接缓冲区拷贝文件

通过JAVA NIO直接缓冲区拷贝文件

通过JAVA NIO通道传输拷贝文件

5. 什么Java序列化，如何实现序列化！

序列化：就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。

六、网络编程 【9道】

1. http协议和RPC协议区别

RPC是远程过程调用, 是JDK底层定义的规范, 它的实现既可以使用TCP也可以使用http, 底层可以使用二进制传输, 效率高.

Http是协议, http协议底层位于传输层是tcp协议. 通用性好可以传输json数据, 效率稍微慢一些

2. OSI网络模型七层都有哪些

OSI参考模型分为7层，分别是物理层，数据链路层，网络层，传输层，会话层，表示层和应用层。

3. 传输层上的TCP和UDP区别

TCP与UDP的特点：

UDP：无连接、不可靠、传输速度快. 适合传输语音, 视频等

TCP：面向连接、可靠、传输速度没有UDP快, 例如: http, smtp等协议底层就是tcp

4. http协议1.0和1.1版本区别

主要有4个方面：缓存处理，带宽优化和网络连接使用，Host请求头的区别，长连接

● 缓存处理：HTTP1.1则引入了更多的缓存控制策略

● 长连接：HTTP1.1默认使用长连接，可有效减少TCP的三次握手开销，在一个TCP连接上可以传送多个HTTP请求和响应，减少了建立和关闭连接的消耗和延迟

● Host请求头的区别：HTTP1.0是没有host域的，HTTP1.1才支持这个参数

● 带宽优化和网络连接使用：在1.1后，现在只会发送head信息，果服务器认为客户端有权限请求服务器，则返回100，当接收到100后才会将剩下的信息发送到服务器

5. http是短连接还是长连接, 如何实现的

- HTTP协议与TCP/IP协议的关系

  HTTP的长连接和短连接本质上是TCP长连接和短连接。HTTP属于应用层协议，在传输层使用TCP协议，在网络层使用IP协议。IP协议主要解决网络路由和寻址问题，TCP协议主要解决如何在IP层之上可靠的传递数据包，使在网络上的另一端收到发端发出的所有包，并且顺序与发出顺序一致。TCP有可靠，面向连接的特点。

- 如何理解HTTP协议是无状态的

  HTTP协议是无状态的，指的是协议对于事务处理没有记忆能力，服务器不知道客户端是什么状态。也就是说，打开一个服务器上的网页和你之前打开这个服务器上的网页之间没有任何联系。HTTP是一个无状态的面向连接的协议，无状态不代表HTTP不能保持TCP连接，更不能代表HTTP使用的是UDP协议（无连接）。

- 什么是长连接、短连接?

  在HTTP/1.0中，默认使用的是短连接。也就是说，浏览器和服务器每进行一次HTTP操作，就建立一次连接，但任务结束就中断连接。如果客户端浏览器访问的某个HTML或其他类型的 Web页中包含有其他的Web资源，如JavaScript文件、图像文件、CSS文件等；当浏览器每遇到这样一个Web资源，就会建立一个HTTP会话。

  但从 HTTP/1.1起，默认使用长连接，用以保持连接特性。使用长连接的HTTP协议，会在响应头有加入这行代码：Connection:keep-alive

  在使用长连接的情况下，当一个网页打开完成后，客户端和服务器之间用于传输HTTP数据的 TCP连接不会关闭，如果客户端再次访问这个服务器上的网页，会继续使用这一条已经建立的连接。Keep-Alive不会永久保持连接，它有一个保持时间，可以在不同的服务器软件（如Apache）中设定这个时间。实现长连接要客户端和服务端都支持长连接。

- HTTP协议的长连接和短连接，实质上是TCP协议的长连接和短连接。
  - TCP连接

    当网络通信时采用TCP协议时，在真正的读写操作之前，server与client之间必须建立一个连接，当读写操作完成后，双方不再需要这个连接 时它们可以释放这个连接，连接的建立是需要三次握手的，而释放则需要4次握手，所以说每个连接的建立都是需要资源消耗和时间消耗

- 三次握手建立连接 , 四次挥手关闭连接
  - 短连接的操作步骤是：

    建立连接——数据传输——关闭连接...建立连接——数据传输——关闭连接
  - 长连接的操作步骤是：

    建立连接——数据传输...（保持连接）...数据传输——关闭连接

6. 简述tcp三次握手四次挥手过程

先向HTTP服务器发起TCP的确认请求(三次握手)

- 客户端 --> SYN --> 服务器
- 服务器 --> SYN+ACK --->客户端
- 客户端 --> ACK --> 服务器

客户端要和服务器断开TCP连接(四次挥手)

- 客户端 --> FIN +ACK ---> 服务器

- 服务器 --> FIN ---> 客户端
- 服务器 --> ACK --> 客户端
- 客户端 --> ACK ---> 服务器

7. http有多少类响应码？分别什么含义

响应码由三位十进制数字组成，它们出现在由HTTP服务器发送的响应的第一行。

响应码分五种类型，由它们的第一位数字表示：

1xx：信息，请求收到，继续处理

2xx：成功，行为被成功地接受、理解和采纳

3xx：重定向，为了完成请求，必须进一步执行的动作

4xx：客户端错误，请求包含语法错误或者请求无法实现

5xx：服务器错误，服务器不能实现一种明显无效的请求

8. tomcat的实现原理！tomcat如何进行优化?

tomcat是一个基于JAVA的WEB容器，其实现了JAVA EE中的 Servlet 与 jsp 规范，与Nginx Apache 服务器不同在于一般用于动态请求处理。在架构设计上采用面向组件的方式设计。即整体功能是通过组件的方式拼装完成。另外每个组件都可以被替换以保证灵活性。

- 实现原理：

Tomcat是运行在JVM中的一个进程。它定义为“中间件”，顾名思义是一个在Java项目与JVM之间的中间容器。

Web项目的本质，是一大堆的资源文件和方法。Web项目没有入口方法（即main方法），这意味着Web项目中的方法不会自动运行起来。Web项目部署进Tomcat的webapp中的目的是很明确的，那就是希望Tomcat去调用写好的方法去为客户端返回需要的资源和数据。

Tomcat可以运行起来，并调用写好的方法。那么，Tomcat有一个main方法。对于Tomcat而言，它并不知道用户会有什么样的方法，这些都只是在项目被部署进webapp下后才确定的。由此，可知Tomcat用到了Java的反射来实现类的动态加载、实例化、获取方法、调用方法。但是部署到Tomcat的中的Web项目必须是按照规定好的接口来进行编写，以便进行调用。

- 优化：
    - 修改内存的相关配置

      修改TOMCAT_HOME/bin/catalina.sh，在其中加入，也可以放在 CLASSPATH=下面, 加一些对内存调优的参数
    - 优化连接器Connector

      Connector是连接器，负责接收客户的请求，以及向客户端回送响应的消息。所以 Connector 的优化是重要部分。默认情况下 Tomcat只支持200线程访问，超过这个数量的连接将被等待甚至超时放弃，所以我们需要提高这方面的处理能力。

      在TOMCAT_HOME/conf/server.xml添加最大线程数量和最小空闲线程数，请求的数量
    - 配置线程池

      Executor代表了一个线程池，可以在Tomcat组件之间共享。使用线程池的好处在于减少了创建销毁线程的相关消耗，而且可以提高线程的使用效率。

9. get 和 post 请求有哪些区别?

GET在浏览器回退时是无害的，而POST会再次提交请求。

GET请求会被浏览器主动cache，而POST不会，除非手动设置。

GET请求只能进行url编码，而POST支持多种编码方式。

GET请求参数会被完整保留在浏览器历史记录里，而POST中的参数不会被保留。

GET请求在URL中传送的参数是有长度限制的，而POST没有。

参数的数据类型，GET只接受ASCII字符，而POST没有限制。

GET比POST更不安全，因为参数直接暴露在URL上，所以不能用来传递敏感信息。

GET参数通过URL传递，POST放在Request body中

七、MySQL以及SQL面试题【20道】

1. 说一说什么是数据库事务!

数据库事务就是在一套业务操作的多条sql语句执行中要么全成功, 要么全失败. 保证了数据的一致性.

事务的四个属性：原子性，一致性，隔离性，持久性。

- 原子性：在事务中进行的修改，要么全部执行，要么全不执行。如果在事务完成之前系统出现故障，SQLServer会撤销在事务中的修改。
- 一致性：为了事务在查询和修改时数据不发生冲突。
- 隔离性：隔离性是一种用于控制数据访问的机制，能够确保事务只能访问处于期望的一致性级别下的数据。SQLServer使用锁对各个事务之间正在修改和查询的数据进行隔离。
- 持久性：在将数据修改写入到磁盘之前，总是先把这些修改写入到事务日志中。这样子，即使数据还没有写入到磁盘中，也可以认为事务是持久化的。这是如果系统重新启动，SQL Server也会检查数据库日志，进行恢复处理。

隔离级别：读未提交、读已提交、可重复读、串行化

2. 事务并发产生的问题和隔离级别!

事务并发产生的问题:

- 脏读：一个事务读取另一个事务的未提交数据！ 真错误！read UNCOMMITTED；
- 不可重复读：一个事务读取；另一个事务的提交的修改数据！read committed;
- 虚读（）幻读：一个事务读取了另一个数的提交的插入数据！repeatable read

隔离级别：读未提交、读已提交、可重复读、串行化

隔离级别选择:

- 数据库隔离级别越高！数据越安全，性能越低！
- 数据库隔离级别越低！数据越不安全，性能越高

建议：设置为第二个隔离级别 read committed;

3. Spring中事务的传播特性有哪些?

七种传播特性:

propagation_requierd：如果当前没有事务，就新建一个事务，如果已存在一个事务中，加入到这个事务中，这是最常见的选择。

propagation_supports：支持当前事务，如果没有当前事务，就以非事务方法执行。

propagation_mandatory：使用当前事务，如果没有当前事务，就抛出异常。

propagation_required_new：新建事务，如果当前存在事务，把当前事务挂起。

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。

propagation_never：以非事务方式执行操作，如果当前事务存在则抛出异常。

propagation_nested：如果有事务在运行，当前的方法就应该在这个事务的嵌套事务内运行，否则就启动一个新的事务，并在它自己的事务内运行。

Spring 默认的事务传播行为是 PROPAGATION_REQUIRED

4. MySQL的内部有哪几部分组成?

- 连接器

  MYSQL拥有非常多的客户端比如：navicat，jdbc，SQLyog等客户端，这些客户端都需要向MYSQL通信都必须要建立连接，这个建立连接的工作就是由连接器完成的

- 查询缓存(5.8版本开始取消了这个)

  连接建立之后，你就可以执行select语句了 这个时候首先会去查询下缓存，缓存的存储形式类似于key-value，key保存的是sql语句value 保存的是结果集，如果查询不在缓存中就会执行sql语句执行完成后把执行结果放入缓存中，如果是能在缓存中查询到那么直接返回

- 词法分析器

  解析sql语句中的词法, 语法

- 优化器

  对解析后生成的执行计划进行自动优化, 提升查询效率

- 执行器

  执行sql语句分析后生成的执行计划, 执行后返回结果

5. MySQL如何实现乐观锁和悲观锁

①乐观锁实现：版本号控制及时间戳控制。

- 版本号控制：表中加一个 version 字段；当读取数据时，连同这个 version 字段一起读出；数据每更新一次就将此值加一；当提交更新时，判断数据库表中对应记录的当前版本号是否与之前取出来的版本号一致，如果一致则可以直接更新，如果不一致则表示是过期数据需要重试或者做其它操作。
- 时间戳控制：其原理和版本号控制差不多，也是在表中添加一个 timestamp 的时间戳字段，然后提交更新时判断数据库中对应记录的当前时间戳是否与之前取出来的时间戳一致，一致就更新，不一致就重试。

②悲观锁实现：必须关闭mysql数据库的自动提交属性，开启事务，再进行数据库操作，最后提交事务。

6. MySQL常用的存储引擎及区别

InnoDB默认的存储引擎, 不需要任何配置, 默认使用的就是它, 支持事务, 支持主外键连接, 速度一般

MyISAM不支持事务, 速度快, 以读写插入为主的应用程序，比如博客系统、新闻门户网站。

Memory存储的数据都放在内存中, 服务器断电, 数据丢失, 速度最快, 现基本被redis取代.

7. 说一说对MySQL索引的理解

- 什么是数据库索引?

  数据库索引，是数据库管理系统中一个排序的数据结构，索引实现通常使用B树及变种的B+树。在数据之外，数据库系统还维护着满足特定查找算法的数据结构，这些数据结构以某种方式引用（指向）数据，这样就可以在这些数据结构上实现高级查找算法。这种数据结构就是索引。

- 索引的作用?

  协助快速查询、更新数据库表中数据。

- 副作用:

  增加了数据库的存储空间插入和修改数据时要花费较多的时间（因为索引也会随之变动）

8. MySQL中什么字段适合添加索引

表的主键、外键必须有索引;

经常与其他表进行连接的表，在连接字段上应该建立索引;

经常出现在Where子句中的字段，特别是大表的字段，应该建立索引;

索引应该建在选择性高的字段上;

索引应该建在小字段上，对于大的文本字段甚至超长字段，不要建索引;

复合索引的建立需要进行仔细分析；尽量考虑用单字段索引代替:

9. MySQL中常见的索引类型

普通索引：是最基本的索引，它没有任何限制

唯一索引：与前面的普通索引类似，不同的就是：索引列的值必须唯一，但允许有空值。如果是组合索引，则列值的组合必须唯一

主键索引：是一种特殊的唯一索引，一个表只能有一个主键，不允许有空值。一般是在建表的时候同时创建主键索引

组合索引：指多个字段上创建的索引，只有在查询条件中使用了创建索引时的第一个字段，索引才会被使用。使用组合索引时遵循最左前缀集合

10. MySQL中索引失效的场景！

WHERE字句的查询条件里有不等于号（WHERE column!=...），MYSQL将无法使用索引，类似地，如果WHERE字句的查询条件里使用了函数（如：WHERE DAY(column)=...），MYSQL将无法使用索引

在JOIN操作中（需要从多个数据表提取数据时），MYSQL只有在主键和外键的数据类型相同时才能使用索引，否则即使建立了索引也不会使用

如果WHERE子句的查询条件里使用了比较操作符LIKE和REGEXP，MYSQL只有在搜索模板的第一个字符不是通配符的情况下才能使用索引。比如说，如果查询条件是LIKE 'abc%',MYSQL将使用索引；如果条件是LIKE '%abc'，MYSQL将不使用索引。

11. 说一说Hash和B+树结构索引区别?

hash索引 : 等值查询效率高，不能排序，不能进行范围查询；Hash索引仅仅能满足"=","IN"和"<=>"查询，不能使用范围查询

B+树索引 : 数据有序，支持范围查询, 查询效率没有hash索引高.

12. B+比B树索引的区别?

B树，每个节点都储存key和dta，所有节点组成这棵树，并且叶子节点指针为null，叶子节点不包含任何关键字的信息。

B+树，所有叶子节点中包含了全部关键字的信息，及指向含有这些关键字记录的指针，且叶子节点本身依关键字的大小自小尔达的顺序链接，所有的非终端节点可以看成是索引部分，节点中仅含有其子树根节点中最大的（或最小的）关键字。

13. 聚集（集中）索引和非聚集（稀疏）索引的区别

【聚集索引】：也称 Clustered Index。是指关系表记录的物理顺序与索引的逻辑顺序相同。由于一张表只能按照一种物理顺序存放，一张表最多也只能存在一个聚集索引。与非聚集索引相比，聚集索引有着更快的检索速度。

MySQL 里只有 INNODB 表支持聚集索引，INNODB 表数据本身就是聚集索引，也就是常说 IOT，索引组织表。非叶子节点按照主键顺序存放，叶子节点存放主键以及对应的行记录。所以对 INNODB 表进行全表顺序扫描会非常快。

【非聚集索引】：也叫 Secondary Index。指的是非叶子节点按照索引的键值顺序存放，叶子节点存放索引键值以及对应的主键键值。MySQL 里除了 INNODB 表主键外，其他的都是二级索引。MYISAM，memory 等引擎的表索引都是非聚集索引。简单点说，就是索引与行数据分开存储。一张表可以有多个二级索引。

14. 什么是索引倒排!

倒排索引源于实际应用中需要根据属性的值来查找记录。这种索引表中的每一项都包括一个属性值和具有该属性值的各记录的地址。由于不是由记录来确定属性值，而是由属性值来确定记录的位置，因而称为倒排索引(inverted index)。带有倒排索引的文件我们称为倒排索引文件，简称倒排文件(inverted file)。

15. 如何快速的复制一张表!

将表结构和数据导出成sql脚本文件, 可以根据要求修改脚本中的表名字防止重名, 然后再导入到mysql中

在库中创建新表后, 使用insert into 表名(字段名) select 字段名 from 表名

16. SQL语句优化的方案?

第一, 开启慢查询日志,让系统运行一段时间, 打开慢查询日志找到具体需要优化的sql语句

第二, 使用explain执行计划分析sql语句中问题出现的哪里

第三, 根据sql语句编写规范, 进行优化, 例如: 不要使用like模糊查询, 因为索引会失效; 尽量避免使用select*, 因为会返回无用字段; 条件中不要使用or关键字, 因为会全表扫描等.

17. 组合索引的最左原则?

MySQL 的联合索引会首先根据联合索引中最左边的、也就是第一个字段进行排序，在第一个字段排序的基础上，再对联合索引中后面的第二个字段进行排序，依此类推。联合索引当遇到范围查询(>、<、between、like)就会停止匹配，建立索引时匹配度高的字段在前，匹配度低的字段在后

举例 : a = 1 and b = 2 and c > 3 and d = 4 如果建立(a,b,c,d)顺序的索引，d是用不到索引的，如果建立(a,b,d,c)的索引则都可以用到，a,b,d的顺序可以任意调整。

=和in可以乱序，比如a = 1 and b = 2 and c = 3 建立(a,b,c)索引可以任意顺序，mysql的查询优化器会帮你优化成索引可以识别的形式

18. 什么是行锁, 表锁, 页锁?

- 行锁 : 开销大，加锁慢；会出现死锁；锁定粒度小，发生锁冲突的概率低

  行锁就是针对数据表中行记录的锁。这很好理解，比如事务A更新了一行，而这时候事务B也要更新同一行，则必须等事务A的操作完成后才能进行更新。
- 表锁 : 开销小，加锁快；不会出现死锁；锁定粒度大，发生锁冲突概率高

  表锁的语法是 lock tables ... read/write。可以用unlock tables主动释放锁，也可以在客户端断开的时候自动释放。需要注意，lock tables语法除了会限制别的线程的读写外，也限定了本线程接下来的操作对象。
- 页级锁

  页级锁是MySQL中锁定粒度介于行级锁和表级锁中间的一种锁。表级锁速度快，但冲突多，行级冲突少，但速度慢。所以取了折衷的页级，一次锁定相邻的一组记录。

  特点：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

19. 数据库的三范式是什么

第一范式：1NF 是对属性的原子性约束，强调的引擎是列的原子性，即数据库表的每一列都是不可分割的原子数据项。

第二范式：2NF 是对记录的惟一性约束，要求实体的属性完全依赖于主关键字。所谓完全依赖是指不能存在仅依赖主关键字一部分的属性。

第三范式：3NF 是对字段冗余性的约束，任何非主属性不依赖于其它非主属性。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。

20. 什么情况下设置了索引但无法使用

- 以“%”开头的 LIKE 语句，模糊匹配
- OR 语句前后没有同时使用索引
- 数据类型出现隐式转化(如 varchar 不加单引号的话可能会自动转换为 int 型)

八、常用框架【19道】

1. MyBatis$和#的区别

#{ }可以防止Sql 注入，它会将所有传入的参数作为一个字符串来处理。

Mybatis在处理#{}时，会将SQL语句中的#{}替换为?号，调用PrepaerdStatement的set方法来赋值。

$ {} 则将传入的参数拼接到Sql上去执行，一般用于表名和字段名参数，$ 所对应的参数应该由服务器端提供，前端可以用参数进行选择，避免 Sql 注入的风险

Mybatis在处理${}时，就是把${}换成变量的值。

2. MyBatis中一级缓存和二级缓存的区别

- 一级缓存 :

  一级缓存是基于sqlsession默认开启的，在操作数据库时需要构造SqlSession对象，在对象中有一个HashMap用于存储缓存数据。不同的SqlSession之间的缓存数据区域是互相不影响的。

  一级缓存作用是sqlsession范围的，在同一个sqlsession中执行两次相同的sql时，第一次得到的数据会缓存放在内存中，第二次不再去数据库获取，而是直接在缓存中获取，提高效率。

  如果执行了增删改并提交到数据库，mybatis是会把sqlsession中的一级缓存清空的，这样是为了数据的准确性，避免脏读现象。

- 二级缓存 :

  二级缓存是基于mapper的namespace作用域，但多个sqlsession操作同一个namespace下的sql时，并且传入的参数也相同，执行相同的sql语句，第一次执行完毕后会将数据缓存，这就是二级缓存。

  二级缓存同样是使用HashMap进行数据存储。相比一级缓存SqlSession，二级缓存的范围更大，多个Sqlsession可以共用二级缓存，二级缓存它是可以跨越多个sqlsession的。

3. MyBatis和数据库交互的原理

详细流程如下：

加载mybatis全局配置文件（数据源、mapper映射文件等），解析配置文件，MyBatis基于XML配置文件生成Configuration，和一个个MappedStatement（包括了参数映射配置、动态SQL语句、结果映射配置），其对应着<select | update | delete | insert>标签项。

SqlSessionFactoryBuilder通过Configuration对象生成SqlSessionFactory，用来开启SqlSession。

SqlSession对象完成和数据库的交互, 通过Executor执行器将MappedStatement对象进行解析，最后通过JDBC执行sql。

借助MappedStatement中的结果映射关系，将返回结果转化成HashMap、JavaBean等存储结构并返回。

4. Spring的IOC是什么

SpringIOC是控制反转, 负责创建对象，管理对象，装配对象，配置对象，并且管理对象的整个生命周期, 也就是以前创建对象需要new, 那么在使用了spring的IOC后, 所有这样的工作都交给spring进行管理.

5. Spring的常用注解

@Autowired 用于自动注入bean

@Resource 根据名字注入bean

@Controller 用于声明控制层类

@Repository 用于声明Dao持久层类

@Service 用于声明Service业务层类

@Component 用于声明一个普通JavaBean

@Configuration 用于声明初始化类相当于spring核心配置文件的标签

@Bean 声明一个配置, 相当于spring核心配置文件的标签

6. Spring的IOC中创建对象的整体流程

通过ClassPathXmlApplicationContext对象加载spring核心配置文件创建ApplicationContext对象

通过applicationContext对象的getBean(beanName)方法创建所需类对象

转换beanName，因为传入的参数可能是别名，也可能是FactoryBean，所以需要解析

如果是beanName，不需要处理

如果是 “&beanName”,则去掉&

如果是别名，则转换为beanName

7. Spring的Bean的三级缓存

一级缓存singletonObjects是完整的bean，它可以被外界任意使用，并且不会有歧义。

二级缓存earlySingletonObjects是不完整的bean，没有完成初始化，它与singletonObjects的分离主要是职责的分离以及边界划分，可以试想一个Map缓存里既有完整可使用的bean，也有不完整的，只能持有引用的bean，在复杂度很高的架构中，很容易出现歧义，并带来一些不可预知的错误。

三级缓存singletonFactories，其职责就是包装一个bean，有回调逻辑，所以它的作用非常清晰，并且只能处于第三层。

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

8. Spring如何处理循环依赖

- 构造器参数循环依赖：

  通过构造器注入构成的循环依赖，此依赖是无法解决的，只能抛出BeanCurrentlyIn CreationException异常表示循环依赖。

  如在创建TestA类时，构造器需要TestB类，那将去创建TestB，在创建TestB类时又发现需要TestC类，则又去创建TestC，最终在创建TestC时发现又需要TestA，从而形成一个环，没办法创建。

  Spring容器会将每一个正在创建的Bean 标识符放在一个“当前创建Bean池”中，Bean标识符在创建过程中将一直保持在这个池中，因此如果在创建Bean过程中发现自己已经在“当前创建Bean池”里时将抛出BeanCurrentlyInCreationException异常表示循环依赖；而对于创建完毕的Bean将从“当前创建Bean池”中清除掉。

- setter方式单例，默认方式：

  Spring先是用构造实例化Bean对象 ，此时Spring会将这个实例化结束的对象放到一个Map中，并且Spring提供了获取这个未设置属性的实例化对象引用的方法。当Spring实例化了StudentA、StudentB、StudentC后，紧接着会去设置对象的属性，此时StudentA依赖StudentB，就会去Map中取出存在里面的单例StudentB对象，以此类推，不会出来循环的问题。

- prototype作用域bean的循环依赖：

  这种循环依赖同样无法解决，因为spring不会缓存prototype作用域的bean，而spring中循环依赖的解决方式正是通过缓存来实现的。

  Spring解决这个问题主要靠巧妙的三层缓存，Spring首先从singleTonObjects（一级缓存）中尝试获取，如果获取不到并且对象在创建中，则尝试从earlySingleTonObjects（二级缓存）中获取，如果还是获取不到并且允许从singleTonFactories通过getObject获取，则通过singleTonFactory.getObject()(三级缓存)获取。

  如果获取到了则移除相对应的singleTonFactory，将singleTonObject放入到earlySingleTonObjects,其实就是将三级缓存提升到二级缓存，这个就是缓存升级。Spring在进行对象创建的时候，会依次从一级、二级、三级缓存中寻找对象，如果找到直接返回。

- 拓展：

  一级缓存：singletonObjects，存放完全实例化属性赋值完成的单例对象的cache，直接可以使用。

  二级缓存：earlySingletonObjects，存放提前曝光的单例对象的cache，尚未进行属性封装的Bean。

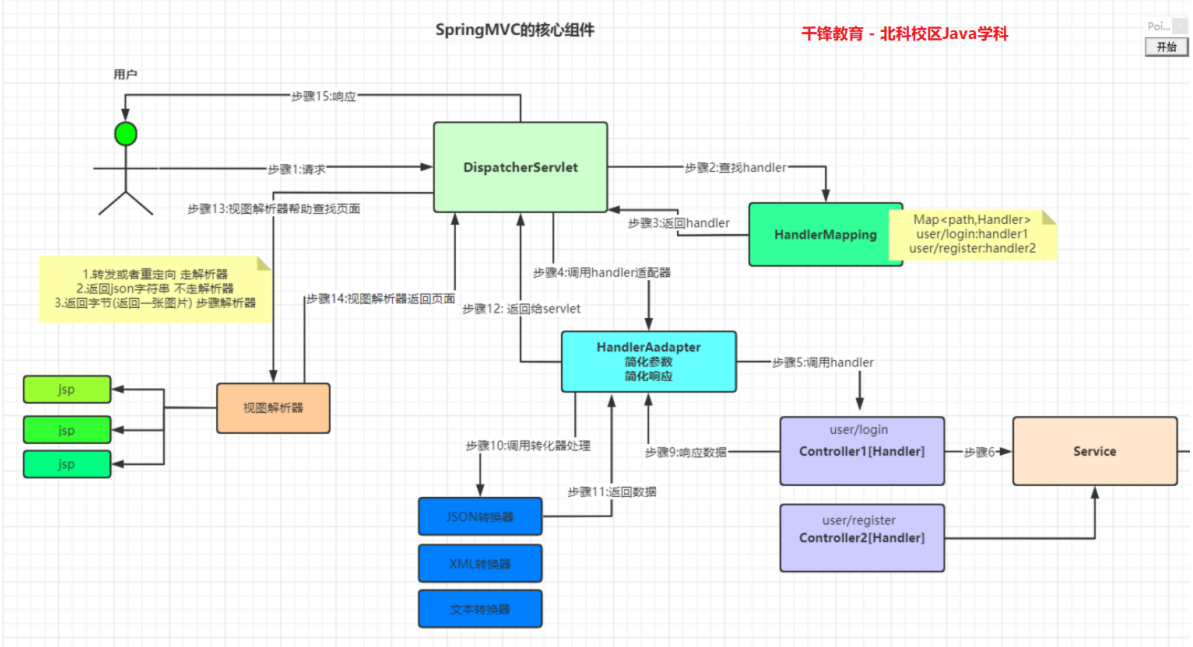
  三级缓存：singletonFactories，存放单例对象工厂的cache。

9. @Autowired和@Resource的区别

@Autowired是自动注入, 一个接口只有一个实现类的时候使用

@Resource是根据bean的名字注入, 一个接口有多个实现类的时候使用, @Resource(实现类bean名字)根据实现类的名字注入指定实现类.

10. SpringMVC的执行流程



11. Spring和SpringMVC的父子工厂关系

Spring容器是父容器，包含的对象有dao，service等

Springmvc是子容器，包括controller

子容器可以访问父容器的对象

父容器不能访问子容器的对象

12. 阐述SpringBoot启动类中注解

@SpringBootAplication：这个注解是启动springboot，从源码我们会发现这个注解被@Configuration、@EnableAutoConfiguration、 @ComponentScan三个注解修饰。换言之，springboot提供了统一的注解来替代这三个注解。

@EnableEurekaClient:这个注解是必须的，表示注册到某个Eureka服务（注册中心）中，相当于给这个服务加了一个通行证，通行证的具体的内容需要在application.yml中配置。一般配置里面会有：eureka、client、serviceurl、defaultZone：http://，代表注册到上面的注册中心，这个注册中心里面的服务包括所有的接口都可以通过协商对方暴露的接口之后直接按照规则进行调用

@MapperScan：这个注解是用来扫描到dao的范围的。如果使用的是mybatis的话，需要通过配置文件来之指定Mapper和主要的配置文件的位置。

@EnableRedisHttpSession：这个注解是用来获取session中缓存的内容， 还需要配合配置文件来完成。

13. SpringBoot如何实现的自动装配

Spring Boot 通过@EnableAutoConfiguration开启自动装配，通过 SpringFactoriesLoader 最终加载META-INF/spring.factories中的自动配置类实现自动装配，自动配置类其实就是通过@Conditional按需加载的配置类，想要其生效必须引入spring-boot-starter-xxx包实现起步依赖

14. Spring事务的传播行为?

事务传播行为（propagation behavior）指的就是当一个事务方法被另一个事务方法调用时，这个事务方法应该如何进行。是为自己开启一个新事务运行，还是由原来的事务执行, 这是由传播行为决定的

传播行为	含义
PROPAGATION_REQUIRED	表示当前方法必须运行在事务中。如果当前事务存在，方法将会在该事务中运行。否则，会启动一个新的事务
PROPAGATION_SUPPORTS	表示当前方法不需要事务上下文，但是如果存在当前事务的话，那么该方法会在这个事务中运行
PROPAGATION_MANDATORY	表示该方法必须在事务中运行，如果当前事务不存在，则会抛出一个异常
PROPAGATION_REQUIRED_NEW	表示当前方法必须运行在它自己的事务中。一个新的事务将被启动。如果存在当前事务，在该方法执行期间，当前事务会被挂起。如果使用JTATransactionManager的话，则需要访问TransactionManager
PROPAGATION_NOT_SUPPORTED	表示该方法不应该运行在事务中。如果存在当前事务，在该方法运行期间，当前事务将被挂起。如果使用JTATransactionManager的话，则需要访问TransactionManager
PROPAGATION_NEVER	表示当前方法不应该运行在事务上下文中。如果当前正有一个事务在运行，则会抛出异常
PROPAGATION_NESTED	表示如果当前已经存在一个事务，那么该方法将会在嵌套事务中运行。嵌套的事务可以独立于当前事务进行单独地提交或回滚。如果当前事务不存在，那么其行为与PROPAGATION_REQUIRED一样。注意各厂商对这种传播行为的支持是有所差异的。可以参考资源管理器的文档来确认它们是否支持嵌套事务

15. Spring中AOP底层实现原理

AOP底层实现使用了动态代理

在spring中有两种实现方式, 一种是JDK动态代理, 一种是CJlib动态代理

JDK动态代理针对实现了接口的类使用

CJlib动态代理针对只有类的时候使用

16. Spring中ApplicationContext对象和BeanFactory对象区别?

这两个对象都可以创建JavaBean对象

ApplicationContext对象 : 这个对象底层是通过BeanFactory对象实现的, 但是ApplicationContext创建JavaBean的时机是调用了这个对象的getBean()方法, 就会立即创建JavaBean

BeanFactory对象 : 它创建JavaBean的时机是调用getBean方法不进行创建JavaBean而是等到用到了JavaBean的时候才去创建.

17. Spring中的注入方式有哪些

常用的注入方式主要有三种:

- 构造方法注入 : 构造方法注入会有不支持的情况发生，因为在调用构造方法中必须传入正确的构造参数，否则报错。
- setter注入 : 注入支持大部分的依赖注入
- 基于注解的注入 : 可以在类中使用@Autowired或是@Resource注解进行注入.

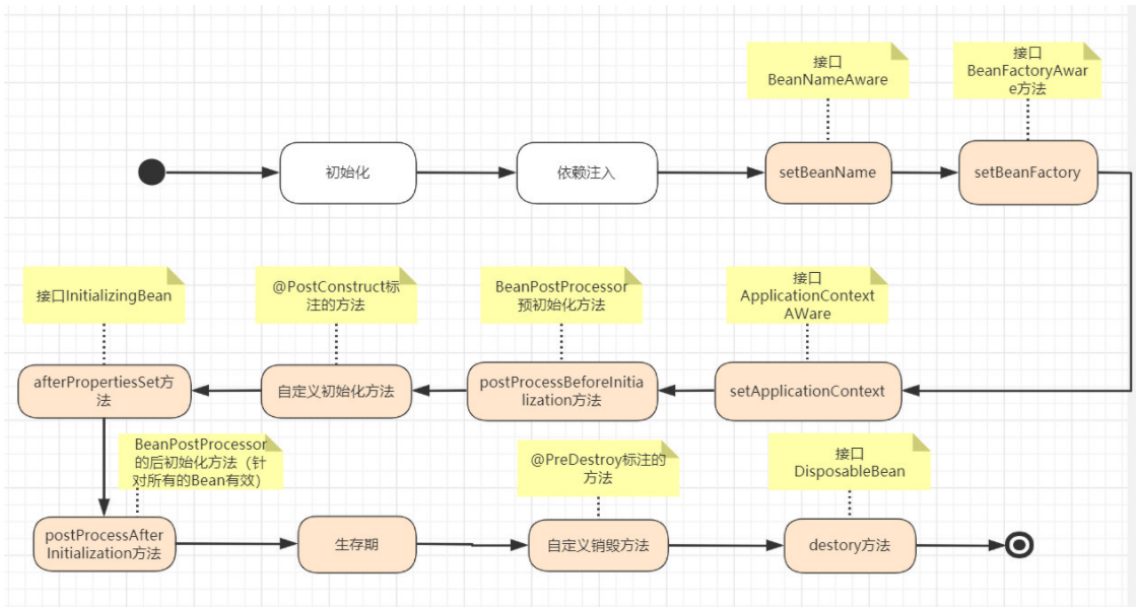
18. Spring 支持几种 bean 的作用域

支持如下5种作用域：

- singleton(默认的)：单例模式，在整个Spring IoC容器中，使用singleton定义的Bean将只有一个实例
- prototype：原型模式，每次通过容器的getBean方法获取prototype定义的Bean时，都将产生一个新的Bean实例
- request：对于每次HTTP请求，使用request定义的Bean都将产生一个新实例，即每次HTTP请求将会产生不同的Bean实例。只有在Web应用中使用Spring时，该作用域才有效
- session：对于每次HTTP Session，使用session定义的Bean豆浆产生一个新实例。同样只有在Web应用中使用Spring时，该作用域才有效
- globalsession：每个全局的HTTP Session，使用session定义的Bean都将产生一个新实例。典型情况下，仅在使用portlet context的时候有效。同样只有在Web应用中使用Spring时，该作用域才有效

19. Spring中Bean的生命周期

Spring中Bean的生命周期



九、中间件和分布式 【54道】

1. Nginx的应用场景有哪些?

负载均衡, 反向代理, 静态资源服务器

2. Nginx负载均衡的策略有哪些?

轮询(默认), 指定权重, IP hash

3. Nginx的进程模型？

nginx模型有两种进程，master进程(相当于管理进程)和worker进程（实际工作进程）。

master进程主要用来管理worker进程，管理包含：接收来自外界的信号，向各worker进程发送信号，监控worker进程的运行状态，当worker进程退出后(异常情况下)，会自动重新启动新的worker进程。

而基本的网络事件，则是放在worker进程中来处理了。多个worker进程之间是对等的，他们同等竞争来自客户端的请求，各进程互相之间是独立的。一个请求，只可能在一个worker进程中处理，一个worker进程，不可能处理其它进程的请求。worker进程的个数是可以设置的，一般我们会设置与机器cpu核数一致，这里面的原因与nginx的进程模型以及事件处理模型是分不开的。

4. Nginx常用命令！

启动 ./nginx

关闭 ./nginx -s stop

查看nginx进程 ps -ef | grep nginx

重新加载nginx ./nginx -s reload

5. Nginx的优化方案！

nginx 进程数，建议按照cpu 数目来指定，一般为它的倍数 (如,2个四核的cpu计为8)。

为每个进程分配cpu，上例中将8 个进程分配到8 个cpu，当然可以写多个，或者将一个进程分配到多个cpu。

配置每个进程允许的最多连接数， 理论上每台nginx 服务器的最大连接数为 : worker进程数 * worker连接数。

6. Redis的应用场景！

热点数据的缓存, 限时业务的运用, 计数器相关问题, 排行榜相关问题, 分布式锁, 点赞、好友等相互关系的存储等

7. Redis存储数据的结构！

Redis的Key-Value格式中Key只能是String类型

Redis的Value类型有5种：

String, List, Set(无序Set), ZSet(有序Set), Hash类型(Map)

8. Redis持久化策略！

redis的持久化策策略有2种：默认是RDB

RDB（数据快照模式），定期存储，保存的是数据本身，操作磁盘频率低, 速度快, 服务器突然断电数据可能丢失一部分.

AOF（追加模式），每次修改数据时，同步到硬盘(写操作日志)，保存的是数据的变更记录, 频繁操作磁盘, 速度慢, 但是数据可靠不容易丢失.

9. Redis集群架构!

哨兵模式:

需要三台服务器, 一台哨兵服务器, 一台主机, 一台备机

哨兵使用心跳检测技术每隔一段时间向主机发送ping命令, 主机返回pong命令, 认为主机存活, 如果主机不返回pong命令, 认为主机宕机

哨兵服务器就会通知备机进行切换, 备机充当主机, 备机会最后ping一次主机, 如果主机返回pong命令不进行切换, 如果主机没有返回pong命令则认为主机确实宕机, 备机切换成主机替代主机工作

10. Redis的淘汰策略

volatile-lru：从设置过期时间的数据集中挑选出最近最少使用的数据淘汰。

volatile-ttl：淘汰机制采用LRU，策略基本上与volatile-lru相似，从设置过期时间的数据集中挑选将要过期的数据淘汰，ttl值越大越优先被淘汰。

volatile-random：随机淘汰, 从已设置过期时间的数据集中任意选择数据淘汰。

allkeys-lru：从所有数据集中挑选最近最少使用的数据淘汰，该策略要淘汰的key面向的是全体key集合，而非过期的key集合。

allkeys-random：从所有数据集中选择任意数据淘汰。

11. Redis缓存的击穿，穿透，雪崩，倾斜问题!

- 缓存击穿:

  就是某一个热点数据，缓存中某一时刻失效了，因而大量并发请求打到数据库上，就像被击穿了一样。说白了，就是某个数据，数据库有，但是缓存中没有。那么，缓存击穿，就会使得因为这一个热点数据，将大量并发请求打击到数据库上，从而导致数据库被打垮。

- 缓存击穿解决方案:

  去掉热点数据的生存时间

  热点数据访问到数据库的时候加个锁, 这样同一时间只能有一个访问热点数据的请求访问数据库, 防止数据库宕机.

- 缓存穿透 :

  缓存穿透与击穿的区别就是，

  击穿：redis中没有数据, 数据库里"有"数据；

  穿透：redis中没有数据, 数据库里也"没"数据。

- 缓存穿透解决方案:

  用户查询会有查询参数, 使用查询参数作为key, value值设置为null, 存入redis并设置生存时间, 如果这样的请求多了, redis抗压,但是不会造成数据库宕机.

  使用布隆过滤器, 直接过滤这样的查询请求

- 缓存雪崩 : 指的是大面积的 key 同时过期，导致大量并发打到我们的数据库。
- 雪崩解决方案 : 将key的过期时间设置为随机, 这样不会在同一时间大量过期, 不会在同一时间造成大量数据库访问, 防止数据库宕机

12. Redis的缓存和数据库的双写一致性!

数据库和redis同步数据流程 : 写数据库, 删除redis对应缓存数据, 再将数据库最新数据写入redis中

- 将不一致分为三种情况:

  数据库有数据，缓存没有数据;

  数据库有数据，缓存也有数据，数据不相等;

  数据库没有数据，缓存有数据。

- 解决方案大概有以下几种:

  对删除缓存进行重试，数据的一致性要求越高，我越是重试得快。

  定期全量更新，简单地说，就是我定期把缓存全部清掉，然后再全量加载。

  给所有的缓存一个失效期

13. Redis如何实现的分布式锁!

- 获取锁:

  使用setnx命令向redis中存入一个键值对并设置过期时间

  setnx命令在存入之前会进行判断, 如果redis中存在这个键值对, 则无法存入并返回状态0, 相当于锁被占用, 无法获取锁可以进入等待或再一次尝试, 如果redis中不存在这个键值对, 则可以存入并返回状态1

- 释放锁:

  使用setnx命令存入数据后, 相当于获取到了锁, 执行完业务后一定要使用delete命令删除这个键值对, 就相当于释放锁,其他任务就可以继续以上面方式获取锁, 因为设置了键值对的超时时间,在规定时间内没有释放锁会redis会自动让这个键值对失效,防止死锁

14. Redis的线程模型

从redis的性能上进行考虑，单线程避免了上下文频繁切换问题，效率高;

从redis的内部结构设计原理进行考虑，redis是基于Reactor模式开发了自己的网络事件处理器： 这个处理器被称为文件事件处理器（file event handler）。而这个文件事件处理器是单线程的，所以才叫redis的单线程模型，这也决定了redis是单线程的

15. Elasticsearch的应用场景

互联网全文检: 像百度, 谷歌等

站内全文检索: 贴吧内帖子搜索, 京东, 淘宝商品搜索等

总之就是大数据量, 海量数据的搜索功能都可以用ElasticSearch

16. 什么是倒排索引

搜索前, 根据被搜索的内容创建文档对象, 文档对象有唯一id, 对被搜索内容进行切分词, 也就是将被搜索的内容中的一句句话切分成一个个词, 组成索引, 索引记录了文档id, 也就是索引和文档有关联关系, 查询的时候先查询索引, 根据索引找到文档id, 根据文档id找到文档内容, 这个过程叫做倒排索引算法

17. Elasticsearch在5.x，6.x，7.x版本的区别

5.x Lucene 6.x 的支持，磁盘空间少一半；索引时间少一半；查询性能提升25%；支持IPV6。

6.x 开始不支持一个 index 里面存在多个 type

7.x TransportClient被废弃以至于es7的java代码，只能使用restclient。对于java编程，建议采用 High-level-rest-client 的方式操作ES集群

18. Elasticsearch中分片的概念

单个节点由于物理机硬件限制，存储的文档是有限的，如果一个索引包含海量文档，则不能在单个节点存储。ES提供分 片机制，同一个索引可以存储在不同分片（数据容器）中，这些分片又可以存储在集群中不同节点上

分片分为 主分片(primary shard) 以及 从分片(replica shard)

主分片与从分片关系：从分片只是主分片的一个副本，它用于提供数据的冗余副本

从分片应用：在硬件故障时提供数据保护，同时服务于搜索和检索这种只读请求

是否可变：索引中的主分片的数量在索引创建后就固定下来了，但是从分片的数量可以随时改变

索引默认创建的分片：默认设置5个主分片和一组从分片（即每个主分片有一个从分片对应），但是从分片没有被启用（主从分片在同一个节点上没有意义），因此集群健康值显示为黄色(yellow)

19. Elasticsearch中refresh和flush是什么

- Refresh:

  当我们向ES发送请求的时候，我们发现es貌似可以在我们发请求的同时进行搜索。而这个实时建索引并可以被搜索的过程实际上是一次es 索引提交（commit）的过程，如果这个提交的过程直接将数据写入磁盘（fsync）必然会影响性能，所以es中设计了一种机制，即：先将index-buffer中文档（document）解析完成的segment写到filesystem cache之中，这样避免了比较损耗性能的io操作，又可以使document可以被搜索。以上从index-buffer中取数据到filesystem cache中的过程叫做refresh。

  es默认的refresh间隔时间是1s，这也是为什么ES可以进行近乎实时的搜索

- Flush :

  我们可能已经意识到如果数据在filesystem cache之中是很有可能在意外的故障中丢失。这个时候就需要一种机制，可以将对es的操作记录下来，来确保当出现故障的时候，保留在filesystem的数据不会丢失，并在重启的时候可以从这个记录中将数据恢复过来。elasticsearch提供了translog来记录这些操作。当向elasticsearch发送创建document索引请求的时候，document数据会先进入到index buffer之后，与此同时会将操作记录在translog之中，当发生refresh时（数据从index buffer中进入filesystem cache的过程）translog中的操作记录并不会被清除，而是当数据从filesystem cache中被写入磁盘之后才会将translog中清空。而从filesystem cache写入磁盘的过程就是flush。

20. 如何提升Elasticsearch的查询效率

- 调优Filesystem Cache性能的方法：

  要让 ES 性能好，最佳的情况下，就是机器的内存，至少可以容纳总数据量的一半。索引数据占用的内存最好控制在留给Filesystem Cache的内存容量中。尽量将所有索引数据加载到内存中, 这样搜索的时候可以从内存中搜索索引速度极快.

- 冷热分离方法：

  将热门数据写一个索引，冷门数据单独写一个索引，这样在热门数据被预热（即写入Filesystem Cache）后，不会被冷数据给冲刷掉。

- 数据预热方法：

  对于热门的搜索词，后台系统可以定时自己去搜索一下，这样，热门数据就会刷到Filesystem Cache中，后面用户实际来搜索热词时就是直接从内存中搜索了

21. 如何提高Elasticsearch的查询命中率

- 加入扩展词词典ext.dic

  主要有两项：生词扩展和同义词扩展

  生词扩展：把网络用语、公司名称等词语放入词典中

  同义词扩展：如我们平常用的iphone，当输入“苹果”的时候，我们也是期待能查出手机的。

- 加入停止词词典stop.dic

  把 了、啊、哦、的、之类的语气词放入停止词词典；把数据筛选过滤

22. RabbitMQ的应用场景

- 异步处理

  很多时候，用户不想也不需要立即处理消息。消息队列提供了异步处理机制，允许用户把一个消息放入队列，但并不立即处理它。想向队列中放入多少消息就放多少，然后在需要的时候再去处理它们。

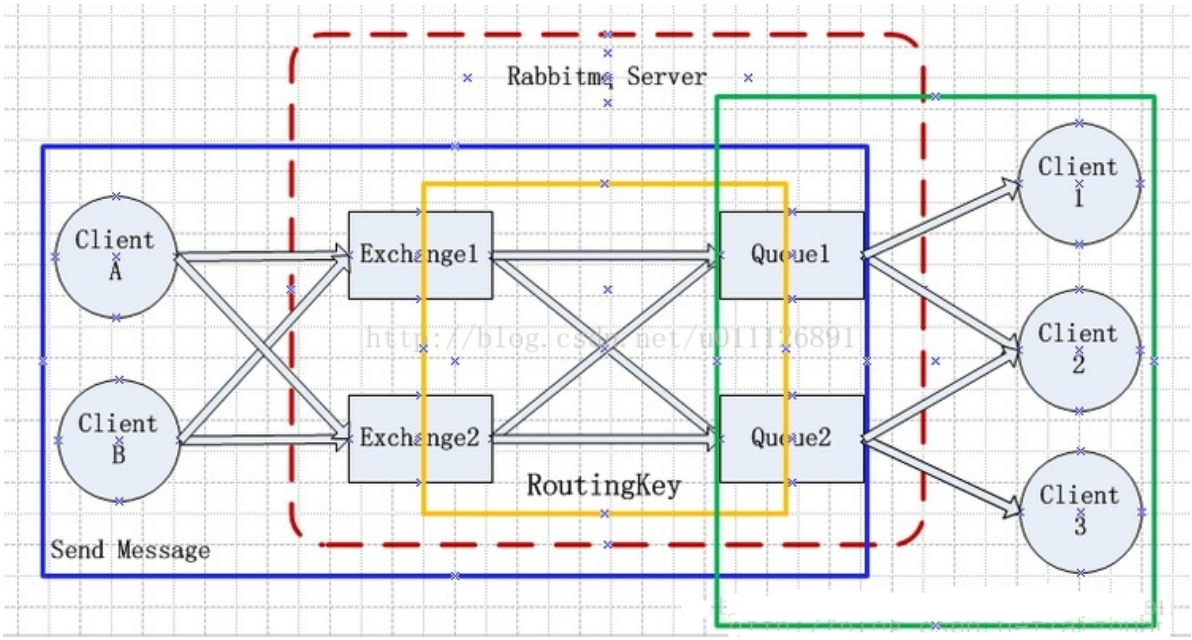
- 应用解耦

  在项目启动之初来预测将来项目会碰到什么需求，是极其困难的。消息系统在处理过程中间插入了一个隐含的、基于数据的接口层，两边的处理过程都要实现这一接口。这允许你独立的扩展或修改两边的处理过程，只要确保它们遵守同样的接口约束

- 流量消峰

  主要可以用来抗高并发的写入, 防止数据库因为高并发写入宕机

23. RabbitMQ的底层架构



Broker:它提供一种传输服务,它的角色就是维护一条从生产者到消费者的路线，保证数据能按照指定的方式进行传输,

Exchange：消息交换机,它指定消息按什么规则,路由到哪个队列。

Queue:消息的载体,每个消息都会被投到一个或多个队列。

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:虚拟主机,一个broker里可以有多个vhost，用作不同用户的权限分离。

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.

24. RabbitMQ如何保证消息的可靠性

可靠性是保证消息不丢失,分为发送消息不丢失和消费消息不丢失两种 :

- 发送不丢失有confirm和return机制
- 消费不丢失可以用ack手动确认机制

25. RabbitMQ如何保证避免消息重复消费

让每个消息携带一个全局的唯一ID，即可保证消息的幂等性，具体消费过程为:

消费者获取到消息后先根据id去查询redis是否存在该消息。

如果不存在，则正常消费，消费完毕后写入redis。

如果存在，则证明消息被消费过，直接丢弃。

26. RabbitMQ的死信队列

死信：是RabbitMQ中的一种消息机制，当消息在队列的存活时间超过设置的TTL时间或者消息队列的消息数量已经超过最大队列长度。这样的消息被认为是死亡的信息也就是死信.

死信消息，会被RabbitMQ自动重新投递到另一个交换机上（Exchange）,这个交换机往往被称为DLX(dead-letter-exchange)“死信交换机”，然后交换机根据绑定规则转发到对应的队列上，监听该队列就可以被重新消费。

27. RabbitMQ如何基于死信队列实现延迟队列以及存在的问题

RabbitMQ中没有延迟队列的功能, 但是可以借助延时消息, 死信, 死信队列来实现延迟队列的功能

第一. 给队列发送消息给消息设置一个超时时间, 超过这个时间没有被消费掉这个消息就会被认为是死信, 这个队列不设置消费方.

第二. 队列中的所有消息, 到达超时时间都会成为死信, 会被RabbitMQ发送到死信交换器

第三. 死信交换器中的数据会被RabbitMQ发送到死信队列

第四. 接收方监听器监听死信队列. 这样从里面消费的消息都是超时后的消息, 也就是先了延迟队列的功能.

28. Zookeeper的应用场景

可以做为dubbo的注册中心

可以做分布式锁

可以为dubbo提供负载均衡功能

分布式协调/通知

29. Zookeeper的存储数据的结构

zookeeper的数据存储结构是一个DataTree数据结构。

其内部是一个Map<String, DataNode> nodes的数据结构，其中key是path，DataNode是真正保存数据的核心数据结构。

30. Zookeeper的节点的类型

永久节点 : 无序但是客户端和zookeeper服务器断开连接后不会被自动删除

永久有序节点 : 有序, 并且客户端和zookeeper服务器断开连接后不会被自动删除

临时节点 : 无序, 客户端和zookeeper服务器断开连接后会被自动删除

临时有序节点 : 有序, 客户端和zookeeper服务器断开连接后会被自动删除

31. Zookeeper的集群架构

Zookeeper集群采用选举机制, 也就是多台zookeeper之间会互相进行投票, 从而选出主机leader, 其他的都是备机flower.

主备之间使用心跳检测技术, 备机每隔一段时间会ping主机, 主机返回pong给备机, 认为主机存活. 如果主机不返回pong,则认为主机宕机, 这个时候其他备机会再次执行选举机制, 选举出新的主机leader来替代老主机工作.

Zookeeper集群一旦超过半数机器宕机, 则整个集群不提供服务, 将失去作用, 所以建议zookeeper集群服务器数量为奇数台.

32. Zookeeper如何实现分布式锁

使用zookeeper创建临时序列节点来实现分布式锁，适用于顺序执行的程序，大体思路就是创建临时序列节点，找出最小的序列节点，获取分布式锁，程序执行完成之后此序列节点消失，通过watch来监控节点的变化，从剩下的节点的找到最小的序列节点，获取分布式锁，执行相应处理，依次类推……

33. 阐述一下你理解的微服务架构

微服务架构是一种架构模式，它提倡将单一应用程序划分成一组小的服务，服务之间相互协调、互相配合，为用户提供最终价值。每个服务运行在其独立的进程中，服务和服务之间采用轻量级的通信机制相互沟通（通常是基于HTTP的Restful API).每个服务都围绕着具体的业务进行构建，并且能够被独立的部署到生产环境、类生产环境等。另外，应尽量避免统一的、集中的服务管理机制.这样实现了易扩展,易维护, 松耦合的特点.

34. 阐述一下SpringCloud中常用的组件

注册中心Eureka : 所有微服务启动后都要注册到Eureka中

远程调用Feign : 底层使用http协议, 发送请求给被调用方, 执行后返回结果

接口负载均衡Ribbon : 如果被调用的微服务是集群, Ribbion起到了负载均衡的效果

服务网关Gateway : 所有外部访问微服务的请求都要经过网关, 网关根据访问的url路径来转发请求到具体服务.

配置中心Config : 统一管理所有微服务的配置文件.

熔断器Hystrix : 并发量超过微服务可以承受的极限, 可以进行熔断或者降级.

35. Eureka的工作机制

服务启动后向Eureka注册，Eureka Server会将注册信息向其他Eureka Server进行同步，当服务消费者要调用服务提供者，则向服务注册中心获取服务提供者地址，然后会将服务提供者地址缓存在本地，下次再调用时，则直接从本地缓存中取，完成一次调用。

当服务注册中心Eureka Server检测到服务提供者因为宕机、网络原因不可用时，则在服务注册中心将服务置为DOWN状态，并把当前服务提供者状态向订阅者发布，订阅过的服务消费者更新本地缓存。

服务提供者在启动后，周期性（默认30秒）向Eureka Server发送心跳，以证明当前服务是可用状态。Eureka Server在一定的时间（默认90秒）未收到客户端的心跳，则认为服务宕机，注销该实例。

36. Ribbon如何实现负载均衡的

调用方微服务到Eureka中根据被调用的服务名获取被调用服务器地址ip和端口号列表

在调用方根据返回的地址列表Rbbion默认使用轮询的策略, 分配请求进行逐个调用

37. Hystrix的断路器以及实现原理

Hystrix的断路器实现原理采用, 马丁福勒断路器原理如下:

一段时间内，失败率达到一定阈值（比如50%失败，或者失败了50次），断路器打开，此时不再请求服务提供者，而是只是快速失败的方法（断路方法）

断路器打开一段时间，自动进入“半开”状态，此时，断路器可允许一个请求方法服务提供者，如果请求调用成功，则关闭断路器，否则继续保持断路器打开状态。

断路器hystrix是保证了局部发生的错误，不会扩展到整个系统，从而保证系统的即使出现局部问题也不会造成系统雪崩。

38. Eureka和Zookeeper实现注册中心的区别

Zookeeper保证的是CP(一致性和分区容错性)

zookeeper对一致性的要求要高于可用性。

Eureka保证的是AP(可用性和分区容错性)

Eureka看明白了这一点, 因此在设计时就优先保证可用性。Eureka各个节点都是平等的，几个节点挂掉不会影响正常节点的工作，剩余的节点依然可以提供注册和查询服务。而Eureka的客户端在向某个Eureka注册时，如果发现连接失败，则会自动切换至其他节点，只要有一台Eureka还在, 就能保住注册服务的可用性,只不过查到的信息可能不是最新的

39. 分布式项目中如何解决分布式事务的问题

首先尽量不要使用分布式事务, 因为会降低代码执行效率, 尽量保证业务的原子性和幂等性, 不使用分布式事务最好.

某些业务必须用, 则可以使用分布式事务框架LCN或者阿里的Seata解决

40. 阐述一下SpringCloud Alibaba中常用的组件

- Nacos(配置中心 + 注册中心)

  Nacos实现了服务的配置中心与服务注册发现的功能，Nacos可以通过可视化的配置降低相关的学习与维护成本，实现动态的配置管理与分环境的配置中心控制。 同时Nacos提供了基于http/RCP的服务注册与发现功能

- Sentinel (分布式流控)

  Sentinel是面向分布式微服务架构的轻量级高可用的流控组件，以流量作为切入点，从流量控制，熔断降级，系统负载保护等维度帮助用户保证服务的稳定性。常用与实现限流、熔断降级等策略

- Dubbo (远程过程调用)

  Dubbo已经在圈内很火了,SpringCloud Alibaba基于上面提到的Nacos服务注册中心也同样整合了Dubbo。

- RocketMQ (消息队列)

  RocketMQ基于Java的高性能、高吞吐量的消息队列，在SpringCloud Alibaba生态用于实现消息驱动的业务开发，常见的消息队列有Kafka、RocketMQ、RabbitMQ等

- Seata (分布式事物)

  既然是微服务的产品，那么肯定会用到分布式事物。Seata就是阿里巴巴开源的一个高性能分布式事物的解决方案。

41. zuul和gateway区别?

目前zuul已经被gateway取代

Gateway比zuul功能更加强大, gateway内部实现了限流, 负载均衡等,但是也限制了仅适用于SpringCould套件, zuul在其他微服务架构下也可以使用, 但是内部没有实现限流, 负载均衡等功能

Zuul仅支持同步, gateway支持异步, 所以gateway性能更好

42. springCloud和Dubbo区别?

没有可比性, dubbo只是底层使用了RPC远程过程调用规范的一个远程调用技术. 而SpringCloud框架集里面有很多子项目, SpringCloud中有注册中心, 配置中心, 远程调用, 接口间负载均衡, 熔断器, 网关等, 可以对服务进行监控, 治理, 配置等有着全套解决方案.

43. 分布式事务和分布式任务如何实现?

分布式事务遵循CAP定理, 可以使用2pc两阶段提交, 或者TCC基于补偿机制实现, 又或者可以使用基于消息最终一致性解决方案等.

具体使用起来也可以使用LCN或者阿里巴巴的Seata框架实现

分布式任务可以使用当当网的ElasticJob分布式任务框架实现

44. redis与mysql怎么保证数据的一致?

- 采用延时双删策略:
  - 先删除缓存;
  - 再写数据库;
  - 休眠500毫秒（根据具体的业务时间来定）;
  - 再次删除缓存。

  那么，这个500毫秒怎么确定的，具体该休眠多久呢?

  需要评估自己的项目的读数据业务逻辑的耗时。这么做的目的，就是确保读请求结束，写请求可以删除读请求造成的缓存脏数据。

  当然，这种策略还要考虑 redis 和数据库主从同步的耗时。最后的写数据的休眠时间：则在读数据业务逻辑的耗时的基础上，加上几百ms即可。比如：休眠1秒。

- 双删失败如何处理?

  设置缓存数据的过期时间

45. 延迟删除是怎么解决数据一致性

延时双删的方案思路是，为了避免更新数据库的时候，其他线程从缓存中读取不到数据，就在更新完数据库之后，在sleep一段时间，然后再删除缓存。

sleep的时间要对业务读写缓存的时间做出评估，sleep时间大于读写缓存的时间即可。

46. 什么是跨域问题? SpringBoot如何解决跨域问题?

- 跨域问题:

  浏览器厂商在生产浏览器的时候, 浏览器内部已经内置了同源策略, 也就是要求当前页面所在的url地址和发送请求目标的url地址中, 协议, 域名, 端口号不可以有变化, 任意一项发生改变, 服务器虽然可以接收请求返回响应, 但是浏览器认为不安全, 不接受响应的数据. 这就是跨域问题.

- 解决方案:

  SpringMvc已经内置了跨域解决方案, 在controller类上加入@CrossOrigin注解就可以解决

  底层原理就是在响应头中加入Access-Control-Allow-Origin: * 设置

47. Zookeeper 集群节点为什么要部署成奇数

Zookeeper集群中只要有超过半数机器不工作，那么整个集群都是不可用的. 当宕掉几个zookeeper节点服务器之后，剩下的个数必须大于宕掉的个数，也就是剩下的节点服务数必须大于n/2，这样zookeeper集群才可以继续使用，无论奇偶数都可以选举leader。例如 : 5台zookeeper节点机器最多宕掉2台，还可以继续使用，因为剩下3台大于5/2。至于为什么最好为奇数个节点？这样是为了以最大容错服务器个数的条件下，能节省资源。比如，最大容错为2的情况下，对应的zookeeper服务数，奇数为5，而偶数为6，也就是6个zookeeper服务的情况下最多能宕掉2个服务，所以从节约资源的角度看，没必要部署6（偶数）个zookeeper服务节点

48. Zookeeper脑裂问题和解决方案?

- 什么是脑裂:

  脑裂(Split-Brain) 就是比如当你的集群里面有3个节点，它们都知道在这个 cluster 里需要选举出一个 master。那么当它们3个之间的通信完全没有问题的时候，就会达成共识，选出其中一个作为 master。但是如果它们之间的通信出了问题，其中一个节点网络抖动断开好长时间, 那么剩下的两个结点都会觉得现在没有 master，所以又会选举出一个 master，当网络恢复后集群里面就会有两个 master。两个master都会争抢资源, 集群就会混乱出现问题.

- 解决Split-Brain脑裂的问题方案：
  - 法定人数方式：比如3个节点的集群，集群可以容忍1个节点失效，这时候还能选举出1个lead，集群还可用。这是zookeeper防止"脑裂"默认采用的方法。
  - 采用Redundant communications (冗余通信)方式：集群中采用多种通信方式，防止一种通信方式失效导致集群中的节点无法通信。
  - Fencing (共享资源) 方式：比如能看到共享资源就表示在集群中，能够获得共享资源的锁的就是Leader，看不到共享资源的，就不在集群中。

49. 什么是幂等性（Idempotence）及用在那里?

幂等性是能够以同样的方式做两次，而最终结果将保持不变，就好像它只做了一次的特性。

用法：在远程服务或数据源中使用幂等性，以便当它多次接收指令时，只处理一次。

50. 什么是熔断？什么是服务降级?

- 熔断

  服务熔断的作用类似于我们家用的保险丝，当某服务出现不可用或响应超时的情况时，为了防止整个系统出现雪崩，暂时停止对该服务的调用。

- 降级

  服务降级是从整个系统的负荷情况出发和考虑的，对某些负荷会比较高的情况，为了预防某些功能（业务场景）出现负荷过载或者响应慢的情况，在其内部暂时舍弃对一些非核心的接口和数据的请求，而直接返回一个提前准备好的fallback（退路）错误处理信息。这样，虽然提供的是一个有损的服务，但却保证了整个系统的稳定性和可用性。

51. dubbo支持的协议有哪些

支持 : dubbo协议, rmi协议, hessian协议, http协议, webservice协议, thrift协议, memcached协议, redis协议

- dubbo协议(默认)
  - 连接个数：单连接
  - 连接方式：长连接
  - 传输协议：TCP
  - 传输方式：NIO 异步传输
  - 序列化：Hessian 二进制序列化
  - 适用范围：传入传出参数数据包较小（建议小于100K），消费者比提供者个数多，单一消费者无法压满提供者，尽量不要用 dubbo 协议传输大文件或超大字符串。
  - 适用场景：常规远程服务方法调用
- rmi协议
  - 连接个数：多连接
  - 连接方式：短连接
  - 传输协议：TCP
  - 传输方式：同步传输
  - 序列化：Java 标准二进制序列化
  - 适用范围：传入传出参数数据包大小混合，消费者与提供者个数差不多，可传文件。
  - 适用场景：常规远程服务方法调用，与原生RMI服务互操作
- hessian协议
  - 连接个数：多连接
  - 连接方式：短连接
  - 传输协议：HTTP
  - 传输方式：同步传输
  - 序列化：Hessian二进制序列化
  - 适用范围：传入传出参数数据包较大，提供者比消费者个数多，提供者压力较大，可传文件。
  - 适用场景：页面传输，文件传输，或与原生hessian服务互操作
- http协议
  - 连接个数：多连接

    - 连接方式：短连接
    - 传输协议：HTTP
    - 传输方式：同步传输
    - 序列化：表单序列化
    - 适用范围：传入传出参数数据包大小混合，提供者比消费者个数多，可用浏览器查看，可用表单或URL传入参数，暂不支持传文件。
    - 适用场景：需同时给应用程序和浏览器 JS 使用的服务。
- WebService协议
    - 连接个数：多连接
    - 连接方式：短连接
    - 传输协议：HTTP
    - 传输方式：同步传输
    - 序列化：SOAP 文本序列化
    - 适用场景：系统集成，跨语言调用

52. dubbo的连接方式有几种

- 无注册中心, 直接连接

  在开发阶段, 服务提供方没有必要部署集群, 所以采用服务调用方直接连接服务提供方更方便测试与开发

- 采用Zookeeper作为注册中心连接

  在线上部署阶段使用, 对于某些并发访问压力大的服务器节点可以部署集群，这时dubbo的服务提供方服务器集群可以使用zookeeper来管理.

- 分组连接

  当一个接口有多种实现时，可以用 group 区分。

53. dubbo的负载均衡策略有哪些

Random LoadBalance 随机, 按权重设置随机概率(默认)

RoundRobin LoadBalance 轮询, 按权重设置轮询比率

LeastActive LoadBalance 最少活跃调用数, 相同活跃数的随机

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

54. dubbo的序列化有哪些, 推荐哪种

- Hessian 序列化：是修改过的 hessian lite，默认启用, 推荐使用.
- json 序列化：使用 FastJson 库
- java 序列化：JDK 提供的序列化，性能不理想
- dubbo 序列化：未成熟的高效 java 序列化实现，不建议在生产环境使用

十、设计模式面试 【2道】

1. 谈一谈你了解的设计模式有哪些?

大致按照模式的应用目标分类，设计模式可以分为创建型模式、结构型模式和行为型模式。

创建型模式 : 是对对象创建过程的各种问题和解决方案的总结，包括各种工厂模式（Factory、Abstract Factory）、单例模式（Singleton）、构建器模式（Builder）、原型模式（ProtoType）。

结构型模式 : 是针对软件设计结构的总结，关注于类、对象继承、组合方式的实践经验。常见的结构型模式，包括桥接模式（Bridge）、适配器模式（Adapter）、装饰者模式（Decorator）、代理模式（Proxy）、组合模式（Composite）、外观模式（Facade）、享元模式（Flyweight）等。

行为型模式 : 是从类或对象之间交互、职责划分等角度总结的模式。比较常见的行为型模式有策略模式（Strategy）、解释器模式（Interpreter）、命令模式（Command）、观察者模式（Observer）、迭代器模式（Iterator）、模板方法模式（Template Method）、访问者模式（Visitor）

2. 设计模式开发中的应用!

单例模式: 例如: 配置类config的对象只能有一个

工厂模式: 例如: 使用BeanFactory工厂创建对象, 不用自己去new对象

代理模式: 例如: spring中的aop底层实现就是动态代理

适配器模式: 例如: SpringMvc中的HandlerInterceptorAdapter就是适配器模式

建造者模式: 例如: mybatis中的SqlsessionFactoryBuilder就是建造者模式, 遵循了单一职责原则, 只做一件事.

十一、 数据结构 【19道】

1. 什么是空间和时间复杂度?

时间复杂度 :

（1）时间频度 一个算法执行所耗费的时间，从理论上是不能算出来的，必须上机运行测试才能知道。但我们不可能也没有必要对每个算法都上机测试，只需知道哪个算法花费的时间多，哪个算法花费的时间少就可以了。并且一个算法花费的时间与算法中语句的执行次数成正比例，哪个算法中语句执行次数多，它花费时间就多。一个算法中的语句执行次数称为语句频度或时间频度。记为T(n)。

（2）时间复杂度在刚才提到的时间频度中，n称为问题的规模，当n不断变化时，时间频度T(n)也会不断变化。但有时我们想知道它变化时呈现什么规律。为此，我们引入时间复杂度概念。 一般情况下，算法中基本操作重复执行的次数是问题规模n的某个函数，用T(n)表示，若有某个辅助函数f(n),使得当n趋近于无穷大时，T（n)/f(n)的极限值为不等于零的常数，则称f(n)是T(n)的同数量级函数。记作T(n)=O(f(n)),称O(f(n)) 为算法的渐进时间复杂度，简称时间复杂度。

空间复杂度 :

一个程序的空间复杂度是指运行完一个程序所需内存的大小。利用程序的空间复杂度，可以对程序的运行所需要的内存多少有个预先估计。一个程序执行时除了需要存储空间和存储本身所使用的指令、常数、变量和输入数据外，还需要一些对数据进行操作的工作单元和存储一些为现实计算所需信息的辅助空间。程序执行时所需存储空间包括以下两部分。

（1）固定部分。这部分空间的大小与输入/输出的数据的个数多少、数值无关。主要包括指令空间（即代码空间）、数据空间（常量、简单变量）等所占的空间。这部分属于静态空间。

（2）可变空间，这部分空间的主要包括动态分配的空间，以及递归栈所需的空间等。这部分的空间大小与算法有关。

一个算法所需的存储空间用f(n)表示。S(n)=O(f(n)) 其中n为问题的规模，S(n)表示空间复杂度。

2. 常见的数据结构有哪些?

数组, 链表, 队列, 堆, 栈, 树, 散列表, 图

3. 链表的数据结构的特点

链表使用不连续内存空间, 空间利用率高

链表插入删除效率高, 查询修改效率慢

链表占用空间大小不固定, 可以扩展

4. 栈数据结构的特点

栈是一种操作受限的线性表，只允许从一端插入和删除数据。拥有后进先出的特点

栈的插入和删除只能在一个位置上进行的表，栈只有进栈、出栈两种操作。前者相当于插入，后者相当于删除最后的元素。

从底层看，栈就是CPU寄存器里的某个指针所指向的一片内存区域

5. 队列数据结构的特点

队列也是一种操作受限的数据结构，只能从队尾的一端进行插入，队头的一端进行删除。先进先出FIFO。双端队列不受这种限制，两端都能进行插入和删除。

6. 说一说什么是跳表？Redis为什么用跳表实现有序集合?

Redis中的有序集合是通过跳表来实现的，还用到了散列表，它支持的核心操作有：插入一个数据、删除一个数据、查找一个数据、按照区间查找数据、迭代输出有序序列。

按照区间来查找数据，这个操作红黑树的效率没有跳表高，其他几个操作红黑树都可以完成，时间复杂度都一样。按照区间来查找，跳表可以做到O(logn)的时间复杂度定位区间的起点，然后在原始链表中的顺序往后遍历即可，非常高效。

跳表使用空间换时间，通过构建多级索引来提高查询的效率，实现基于链表的二分查找，跳表是一个动态数据结构，支持快速的插入、删除、查找操作，时间复杂度都是O(logn)。

跳表的空间复杂度是O(n),通过改变索引构建策略

7. 散列表的数据结构特点

哈希表的查找效率主要取决于构造哈希表时选取的哈希函数和处理冲突的方法。

在各种查找方法中，平均查找长度与结点个数n无关的查找方法是哈希表查找法。

哈希函数取值是否均匀是评价哈希函数好坏的标准。

哈希存储方法只能存储数据元素的值，不能存储数据元素之间的关系。

8. 二叉树数据数据结构特点

每个结点最多有两棵子树，所以二叉树中不存在度大于2的结点。注意不是只有两棵子树，而是最多有。没有子树或者有一棵子树都是可以的。

左子树和右子树是有顺序的，次序不能任意颠倒

即使树中某结点只有一棵子树，也要区分它是左子树还是右子树。

9. 图数据结构特点

无向图 : 每个节点都没有方向，边上可能有权重

有向图 : 每个节点是有方向的的

有向无环图 : 可以描述任务之间的关系

10. 堆数据结构特点

将根节点最大的堆叫大顶堆或者大根堆，根节点最小的堆叫小顶堆或小根堆。

常见的堆有二叉堆，斐波拉契堆。

如果是大顶堆，常见操作及时间复杂度：
查找最大值：o（1）
删除最大值：o(logn)
添加值：o(1)或o(logn)

11. 大顶堆和小顶堆的区别?

大顶堆 : 根结点（亦称为堆顶）的关键字是堆里所有结点关键字中最大者，称为大顶堆。大根堆要求根节点的关键字既大于或等于左子树的关键字值，又大于或等于右子树的关键字值。

小顶堆 : 根结点（亦称为堆顶）的关键字是堆里所有结点关键字中最小者，称为小顶堆。小根堆要求根节点的关键字既小于或等于左子树的关键字值，又小于或等于右子树的关键字值。

12. 说一说常见的排序算法和对应的时间复杂度

类别	排序方法	时间复杂度			空间复杂度	稳定性
		平均情况	最好情况	最坏情况	辅助存储	
插入排序	直接插入	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
	Shell排序	$O(n^{1.3})$	$O(n)$	$O(n^2)$	$O(1)$	不稳定
选择排序	直接选择	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
	堆排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(1)$	不稳定
交换排序	冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
	快速排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n^2)$	$O(n\log_2 n)$	不稳定
归并排序		$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n)$	稳定
基数排序		$O(d(r+n))$	$O(d(n+rd))$	$O(d(r+n))$	$O(rd+n)$	稳定

注：基数排序的复杂度中，r代表关键字的基数，d代表长度，n代表关键字的个数

13. 用Java代码实现冒泡排序和快速排序

```
package 冒泡排序;
import java.util.Arrays;

/**
 * 冒泡排序
 * @author mmz
 */
public class BubbleSort {
    public static void BubbleSort(int[] arr) {
        int temp;//定义一个临时变量
        for(int i=0;i<arr.length-1;i++){//冒泡趟数
            for(int j=0;j<arr.length-i-1;j++){
                if(arr[j+1]<arr[j]){
                    temp = arr[j];
                    arr[j] = arr[j+1];
                    arr[j+1] = temp;
                }
            }
        }
    }

    public static void main(String[] args) {
        int arr[] = new int[]{1,6,2,2,5};
        BubbleSort.BubbleSort(arr);
        System.out.println(Arrays.toString(arr));
    }
}
```

```java
// 快速排序
public class QuickSort{
    public static void quickSort(int[] arr,int low,int high){
    int i,j,temp,t;
    if(low>high){
        return;
    }

    i=low;
    j=high;
    //temp就是基准位
    temp = arr[low];

    while (i<j) {
        //先看右边，依次往左递减
        while (temp<=arr[j]&&i<j) {
           j--;
        }
        //再看左边，依次往右递增
        while (temp>=arr[i]&&i<j) {
           i++;
        }
        //如果满足条件则交换
        if (i<j) {
            t = arr[j];
            arr[j] = arr[i];
            arr[i] = t;
         }
    }

    //最后将基准为与i和j相等位置的数字交换
    arr[low] = arr[i];
    arr[i] = temp;

    //递归调用左半数组
    quickSort(arr, low, j-1);

    //递归调用右半数组
    quickSort(arr, j+1, high);
}

public static void main(String[] args){
    int[] arr = {10,7,2,4,7,62,3,4,2,1,8,9,19};
    quickSort(arr, 0, arr.length-1);
    for (int i = 0; i < arr.length; i++) {
            System.out.println(arr[i]);
        }
    }
}
```

14. 100万用户如何根据年龄排序?

可以通过桶排序，基数排序，计数排序

桶排序：桶排序要把数据进行划分到m个桶内，希望的是桶内数据是均匀的，并且桶与桶之间有着天然的大小顺序。极端情况下时间复杂度会退化为O(nlog n); 比较适合外部排序。在进行划分桶数据的时候，可能存在桶数据不均匀的情况，可以选择在多的数据桶进行继续划分桶，直到桶数据可以加载到内存中为止。

计数排序：把一系列的数字统计个数放在数组内A。 依次累加，统计结果还是在同一个数组中存放A。临时数组C存放排序后的结果。 遍历最初时的数据B，从A中找到该值。 A数组中的值-1就是数据B在临时数组C存放的位置。 把A数组中的值减1. 循环这个过程。该计数规则只适合数据范围不大的情景

15. 深度优先和广度优先搜索算法?

广度优先搜索（Breadth-First-Search），一般简称为 BFS。直观地讲，它其实就是一种地毯式层层推进的搜索策略，即先查找离起始顶点最近的，然后是次近的，依次往外搜索。

visited，布尔数组，记录顶点是否已经被访问过，访问过则为真，没有访问过则为假，这里用 0 和 1 表示。

vertex，记录上一层的顶点，也即已经被访问但其相连的顶点还没有被访问的顶点。当一层的顶点搜索完成后，我们还需要通过这一层的顶点来遍历与其相连的下一层顶点，这里我们用队列来记录上一层的顶点。

prev，记录搜索路径，保存的是当前顶点是从哪个顶点遍历过来的，比如 prev[4] = 1，说明顶点 4 是通过顶点 1 而被访问到的。

深度优先搜索（Depth-First-Search），简称 DFS，最直观的例子就是走迷宫。

假设你站在迷宫的某个分岔路口，你想找到出口。你随意选择一个岔路口来走，走着走着发现走不通的时候就原路返回到上一个分岔路口，再选择另一条路继续走，直到找到出口，这种走法就是深度优先搜索的策略。

深度优先搜索用的是一种比较著名的思想——回溯思想，这种思想非常适合用递归来实现。深度优先搜索的代码里面有几个和广度优先搜索一样的部分 visited、prev 和 Print() 函数，它们的作用也都是一样的。此外，还有一个特殊的 found 变量，标记是否找到终止顶点，找到之后我们就可以停止递归不用再继续查找了。

16. 如何快速获取Top10热门搜索关键词?

使用优先级队列实现 : 优先级队列，顾名思义，它首先应该是一个队列。队列最大的特性就是先进先出。不过，在优先级队列中，数据的出队顺序不是先进先出，而是按照优先级来，优先级最高的，最先出队

17. 单向链表反转如何实现!

①. 迭代反转链表：从当前链表的首元节点开始，一直遍历至链表的最后一个节点，这期间会逐个改变所遍历到的节点的指针域，另其指向前一个节点。

```
public static Node reverse2(Node head) {
    if (head == null || head.getNext() == null) {
        return head;
    }
```

```
    // 上一结点等于头节点
    Node pre = head;
    // 当前结点
    Node cur = head.getNext();
    // 临时结点，用于保存当前结点的指针域（即下一结点）
    Node tmp;

    // 当前结点为null，说明位于尾结点
    while (cur != null) {
        //临时节点，保存当前节点下一个节点
        tmp = cur.getNext();
        // 反转指针域的指向
        cur.setNext(pre);
        // 指针往下移动
        pre = cur;
        cur = tmp;
    }
    // 最后将原链表的头节点的指针域置为null，还回新链表的头结点，即原链表的尾结点
    head.setNext(null);
    return pre;
}
```

②. 递归反转链表：从链表的尾节点开始，依次向前遍历，遍历过程依次改变各节点的指向，即另其指向前一个节点。

```
public static Node Reverse1(Node head) {
    // head看作是前一结点，head.getNext()是当前结点，reHead是反转后新链表的头结点
    if (head == null || head.getNext() == null) {
        // 若为空链或者当前结点在尾结点，则直接还回
        return head;
    }
    // 先反转后续节点head.getNext()
    Node reHead = Reverse1(head.getNext());
    // 将当前结点的指针域指向前一结点
    head.getNext().setNext(head);
    // 前一结点的指针域令为null;
    head.setNext(null);
    // 反转后新链表的头结点
    return reHead;
}
```

③. 头插法反转链表：在原有链表的基础上，依次将位于链表头部的节点摘下，然后采用从头部插入的方式生成一个新链表，则此链表即为原链表的反转。

```
public ListNode reverseList(ListNode head) {
    ListNode dum = new ListNode();
    ListNode pre = head;
    while (pre!=null){
        ListNode pNext = pre.next;
        pre.next = dum.next;
        dum.next = pre;
        pre =  pNext;
    }
    head = dum.next;
    return head;
}
```

18. 如何判断链表是否有环?

使用追赶的方法，设定两个指针slow、fast

从头指针开始，每次fast指针前进1步, slow指针前进2步。

如存在环，则两者相遇；如不存在环，fast遇到NULL退出

19. 如何找到单向链表的中间元素

设定两个指针slow、fast

从头指针开始，每次fast指针前进1步, slow指针前进2步

当fast指针走到链表结尾位置, slow指针所处位置的元素就是中间元素.

十二、 工具使用 【3道】

1. 在Linux中如何查看tomcat日志, 命令是什么?

tail -f tomcat日志所在路径

2. maven推送自己写的工具包项目到公司maven私服供其他组员使用的命令?

mvn deploy 可以推送到公司内部的私服服务器,

其他组员maven的settings.xml中可以配置公司私服的地址,

然后项目的pom文件中写上这个jar包的坐标, 就会自动从公司私服下载这个依赖包, 项目中就可以直接使用.

3. 使用maven项目如何打包部署

使用mvn package命令进行打包

War包项目打包后将打包好的文件上传到服务器tomcat的webapps目录, 重启tomcat

Springboot项目打包后是jar包, 将打包好的文件上传到服务器, 执行java -jar xxx.jar启动运行项目.