# 面试集

10001

**TreeSet实现定制排序的方式:**

* 需要被排序的类实现Comparable接口重写其中的 comparato
* 这种方法需要一个新的类实现Comparator接口重写其中的Compare 方法

**TreeSet去重:**

* 根据比较器设置的比较属性去重
* 如果compareTo返回0，说明是重复的，返回的是自己的某个属性和另一个对象的某个属性的差值，如果是负数，则往前面排，如果是正数，往后面排；

**LinkedList和ArrayList哪个占用的空间大?**

* 一般情况下，LinkedList的占用空间更大，因为每个节点要维护指向前后地址的两个节点，但也不是绝对，如果刚好数据量超过ArrayList默认的临时值时，ArrayList占用的空间也是不小的，因为扩容的原因会浪费将近原来数组一半的容量，不过，因为ArrayList的数组变量是用transient关键字修饰的，如果集合本身需要做序列化操作的话，ArrayList这部分多余的空间不会被序列化

**HashMap的put原理?**

* 判断数组长度是否为0或者null,如果是,则进行resize方法进行,初始化,因为new HashMap时数组不会创建.
* 进行Key值的hash运算,计算出hash,再使用数组长度-1 & hash,求出元素在数组中存储的位置,因为需要用到数组长度与hash运算,数组长度必须为2的次幂,保证计算得到的结果值离散的分布在数组范围之中.
* 判断该位置是否有元素,没有则保存,有元素则发生哈希冲突,进行再次判断(equals),如果eq相等,表示同一个值,会将原来的值保存到临时变量,用于返回,如果eq不相等则表示未重复.
* 未重复则需要进行判断当前节点,如果是首节点则直接插入
* 在判断结构,如果为树结构则生成节点并挂载在树结构中
* 如果为链表结构则进行遍历判断重复,如果未重复则直接插入链表尾部,再次判断链表长度是否大于等于8,因为涉及到泊松分布(概率性理论),我理解的就是未达到扩容临界值时链表长度达到8的概率为0.0000006(好像是这么大,源码注释有写),如果链表达到树化要求则调用treeifyBin方法进行树化,树化方法中同样会进行判断,如果当前数组长度大于64才进行树化,如果小于则进行resize.
* 如果链表中key重复,则结束循环,返回旧的值.
* 最后判断数组的长度,如果达到扩容临界值(数组长度 \* 负载因子)则进行resize.

**实现线程安全的集合:**

* (Vector/HashMap..): 加synchronize锁同步方法,效率低
* 使用Collections工具类(Collections.synchronizedList/Collections.Map): synchronize锁同步代码块
* List: 使用concurrent 包下的类(CopyOnWriteArrayList): 加Lock锁同步代码块,get方法无锁

Map: 使用ConcurrenHashMap,在Put方法的List操作中添加synchronize锁

**ArrayList和LinkedList的区别?**

可以从四个方面说:

1. 数据结构(数组/双向链表)
2. 使用场景/性能
3. 初始化容量(ArrayList: 10 扩容:原长度+原长度/2 +1 LinkedList: 0 无扩容,直接追加)
4. 相等长度时的占用空间大小

**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。

**什么是反射?**

反射是在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意一个方法和属性；这种动态获取的信息以及动态调用对象的方法的功能称为 Java 语言的反射机制。

## **什么是 Java 序列化？什么情况下需要序列化？**

Java 序列化是为了保存各种对象在内存中的状态，并且可以把保存的对象状态再读出来。

以下情况需要使用 Java 序列化：

* 想把的内存中的对象状态保存到一个文件中或者数据库中时候；
* 想用套接字在网络上传送对象的时候；
* 想通过RMI（远程方法调用）传输对象的时候。

**nginx负载均衡策略:**

1.轮询策略

2.根据权重

3.根据ip

4.根据url

5.根据服务器的响应时间

**jsp和servlet区别:**

JSP 是 servlet 技术的扩展,servlet的应用逻辑是在 Java 文件中，并且完全从表示层中的 html 里分离开来

JSP是 Java 和 html 可以组合成一个扩展名为 JSP 的文件。JSP 侧重于视图，servlet 主要用于控制逻辑

**浅拷贝和深拷贝的区别:**

浅拷贝:只是增加了一个指针指向已存在的内存地址(会复制所有的基本属性,而引用类型只是复制指针)

深拷贝: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

**页面优化方案:**

1.前后端分离

2.数据缓存(redis)

3.动静分离

4.页面静态化

5.本地缓存(浏览器)

6.CDN加速

**限流手段:**

前端:

验证码

后端:

redis信号量限流

MQ削峰

# SpringMvc

**SpringMVC常见注解:**

1.controller

2.service

3.component

4.repository

5.ResponseBody 响应json数据

6.PathVariable 接收路劲中的参数

7.requestMapping 处理请求映射地址

8.Resource和Autowired 注入bean

**SpringMvc常见组件:**

* 前置控制器 DispatcherServlet。
* 映射控制器 HandlerMapping。
* 处理器 Controller。
* 模型和视图 ModelAndView。
* 视图解析器 ViewResolver。

**SpringMvc执行流程:**

1.用户发送请求至前端控制器 DispatcherServlet。

2.DispatcherServlet 收到请求调用 HandlerMapping 处理器映射器。

3.处理器映射器找到具体的处理器(可以根据 xml 配置、注解进行查找)，生成处理器对象

及处理器拦截器(如果有则生成)一并返回给 DispatcherServlet。

4.DispatcherServlet 调用 HandlerAdapter 处理器适配器。

5.HandlerAdapter 经过适配调用具体的处理器(Controller，也叫后端控制器)。

6.Controller 执行完成返回 ModelAndView。

7.HandlerAdapter 将 controller 执行结果 ModelAndView 返回给 DispatcherServlet。

8.DispatcherServlet 将 ModelAndView 传给 ViewReslover 视图解析器。

9.ViewReslover 解析后返回具体 View。

10.DispatcherServlet 根据 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。

**Spring**

**什么是IOC**

ioc：Inversionof Control（中文：控制反转）是 spring 的核心，对于 spring 框架来说，就是由 spring 来负责控制对象的生命周期和对象间的关系。

**什么是AOP**

aop:是面向切面编程，通过预编译方式和运行期动态代理实现程序功能的统一维护的一种技术

**Spring常见注解:**

1.controller

2.service

3.component

4.repository

5.Resource和Autowired 注入bean

6.Configuration 声明当前类为配置类

7.Aspect 声明一个切面

8.Scope 设置Spring容器新建Bean实例范围

**spring bean 的作用域**

* **singleton**：spring ioc 容器中只存在一个 bean 实例，bean 以单例模式存在，是系统默认值；
* **prototype**：每次从容器调用 bean 时都会创建一个新的示例，既每次 getBean()相当于执行 new Bean()操作；
* **request**：每次 http 请求都会创建一个 bean；
* **session**：同一个 http session 共享一个 bean 实例；
* **global-session**：用于 portlet 容器，因为每个 portlet 有单独的 session，globalsession 提供一个全局性的 http session。

**spring 自动装配 bean 的方式:**

* no：默认值，表示没有自动装配，应使用显式 bean 引用进行装配。
* byName：它根据 bean 的名称注入对象依赖项。
* byType：它根据类型注入对象依赖项。
* 构造函数：通过构造函数来注入依赖项，需要设置大量的参数。
* autodetect：容器首先通过构造函数使用 autowire 装配，如果不能，则通过 byType 自动装配。

## **spring 常用的注入方式有哪些？**

* setter属性注入
* 构造方法注入
* 注解方式注入

**spring 中的 bean 是线程安全的吗?**

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。

解决方案:

1. 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。
2. 尽量不要使用成员变量
3. 通过threadlocal，把变量放在threadlocal中，这样每个线程都有自己独立的变量副本

**BeanFactory与ApplicationContext的区别:**

ApplicationContext继承BeanFactory接口，它是Spring的一各更高级的容器，提供了更多的有用的功能,

ApplicationContext是迫切加载,而BeanFactory是懒加载

**BeanFactory与FactoryBean(BeanFactory和FactoryBean其实没有什么比较性的，只是两者的名称特别接近)**

BeanFactory是IOC最基本的容器，负责生产和管理bean，它为其他具体的IOC容器提供了最基本的规范

FactoryBean可以说为IOC容器中Bean的实现提供了更加灵活的方式，在IOC容器的基础上给Bean的实现加上了一个简单工厂模式和装饰模式

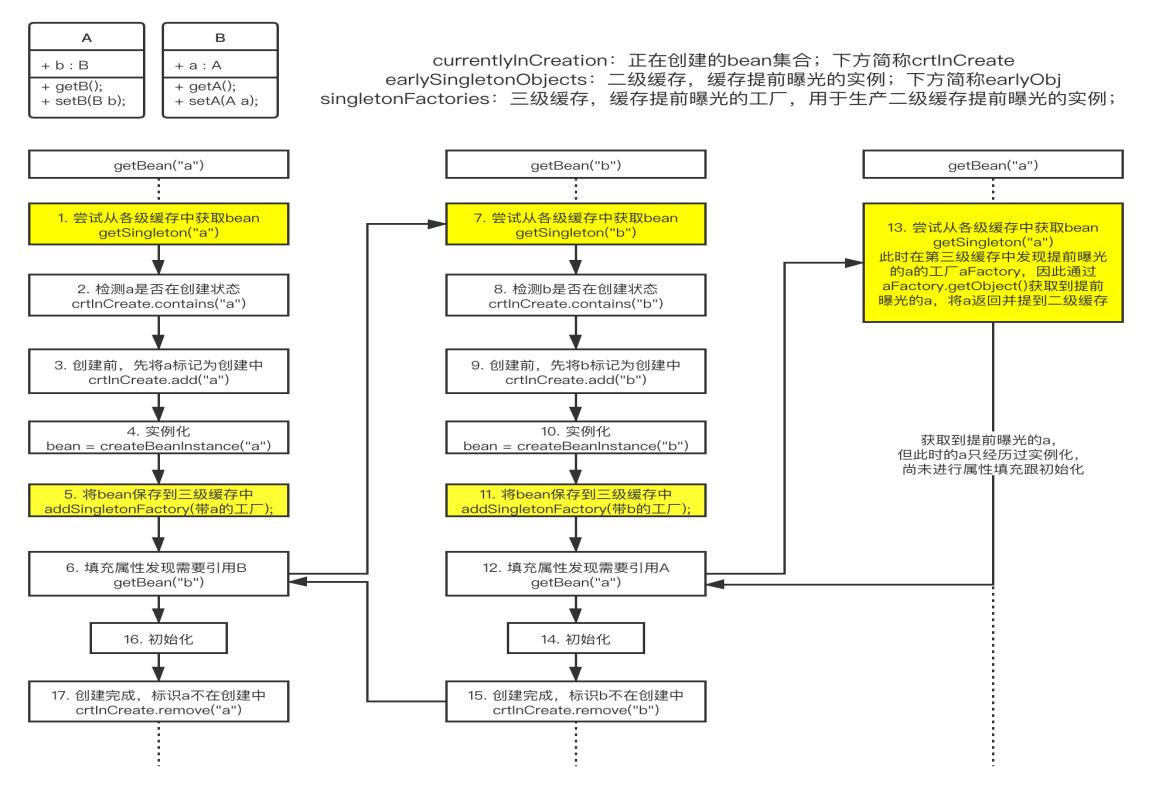
* **Sprig如何解决循环依赖问题(构造注入不能解决!):**

未等bean创建完就先将实例曝光出去，方便其他bean的引用,就是spring先将创建好的实例放到缓存中，让其他bean可以提前引用到该对象。

singletonObjects：第一级缓存，里面放置的是实例化好的单例对象；

earlySingletonObjects：第二级缓存，里面存放的是提前曝光的单例对象；

singletonFactories：第三级缓存，里面存放的是要被实例化的对象的对象工厂。



**AOP使用场景:**

1.监控性能

2.记录日志

3.事务管理

**springboot**

**Springboot常见注解:**

1.SpringBootApplication 启动类注解

2.Bean 定义一个Bean

3.Getter,Setter,Data,NoArgsConstructor,AllArgsConstructor

4.RestController 是一个结合了 @ResponseBody 和 @Controller 的注解

5.@GetMapping,@PostMapping,@PutMapping,@DeleteMapping,@PatchMapping

6.@MapperScan

7.SpringBootTest

8.runwith

9.postConstruct 初始化注解,对象创建就初始化

**SpringBoot自动装配原理:**

**启动类注解SpringbootApplication是一个复合注解,其中包含了@SpringBootConfiguration/@ComPonentSan/@EnableAutoConfiguration,其中@SpringBootConfiguration注解相当于@Configuration声明是一个配置之类,@ComPonentSan扫描组件,SpringBoot主启动类的同级路径及子路径,@EnableAutoConfiguration是主要做自动配置的注解,然而这个注解也是一个复合注解,其中包含@Import({AutoConfigurationImportSelector.class}),这里导入了这个类,这个类实现了DeferredImportSelector接口,然后复写了selectImports方法,方法中通过getAutoConfigurationEntry返回我们所需要的配置信息(Entry对象)还包含去重和检查排除类的操作,而这个方法中又调用了getCandidateConfigurations方法,返回了配置类全包名的list集合,然后这个方法通过loadFactoryName方法加载了META-INF/spring.factories文件,这个文件包含了我们所需要的配置类的全包名,然后将全路径包名返回,最终返回了一个字符串数组,spring就根据全包名自动装配上这些配置文件下定义的bean对象(@SpringBootConfiguration ?)，从而达到了自动装配的目的.**

**导入注册bean方式3?**

**①在配置类上加@Configuration注解**

**②在配置类上加@Import注解，该注解的value属性为字节码数组，可以接收以下三种类的字节码**

**要注册的类；**

**实现了ImportSelector接口的自定义类，其实现方法的返回值为要注册的类的全限定名数组；**

**实现了ImportBeanDefinitionRegistrar的自定义类，在其实现方法中，使用BeanDefinitionRegistry手动注册，前面两种方式注册的bean的id只能是类的全限定名，只有这种方式可以自定义bean的id；**

# **spring cloud**

**spring cloud 的核心组件有哪些？**

* Eureka：服务注册于发现。
* Feign：基于动态代理机制，根据注解和选择的机器，拼接请求 url 地址，发起请求。
* Ribbon：实现负载均衡，从一个服务的多台机器中选择一台。
* Hystrix：提供线程池，不同的服务走不同的线程池，实现了不同服务调用的隔离，避免了服务雪崩的问题。
* Zuul：网关管理，由 Zuul 网关转发请求给对应的服务。
* Config:配置中心
* Bus:消息总线

**分布式事务:**

分布式环境下,不同服务之间通过网络远程协助完成的事务,无法通过本地事务解决的事务问题

**产生场景**:

微服务架构 -> 跨JVM进程

单体项目访问多个数据库 -> 跨数据库实例

多个服务访问同一个数据库 -> 跨JVM进程

**CAP理论(不能三者同时满足):**

**一致性(C):**保证数据的一致性,数据同步失败,可以返回失败信息,但一定不能返回旧数据

**可用性(A) :**必须对外提供服务,数据可以不一致,可以返回旧数据

**分区容忍性(P):**服务在不同的子网,服务之间通信可以出现异常(分区),但都必须对外提供服务 -> 分布式系统具备的基本能力

**Base理论(柔性事务):** 对CAP理论中AP的扩展,牺牲强一致性,保证最终一致性 --> 基本可用(BA) 软状态(S,允许存在中间状态) -> 最终一致性(E)

**分布式事务解决方案:**

**2PC(两阶段提交):** 整个事务由事务管理器和参与者组成,事务管理器负责决策整个分布式事务的提交和回滚,事务参与者负责自己本地事务的提交和回滚

**准备阶段**:事务管理器让每个数据库参与者在本地执行自己的事务,并记录本地的**Undo/Redo**日志,但此时**没有提交事务**

Undo: 记录修改前的数据,用于回滚

Redo:记录修改后的数据,用于提交事务

**提交阶段**:都成功的话事务管理器则发送提交请求,如果失败则给参与者发送回滚消息

# Redis

**Redis使用场景:**

1.中央缓存

2.计数器

3.设定效期应用(红包,vip,游戏道具)

4.订阅发布

5.排行榜

6.队列

**Redis淘汰策略:**

1.volatile-lru： 从已设置过期时间的数据集中挑选最近最少使用的数据淘汰

2.volatile-ttl： 从已设置过期时间的数据集中挑选将要过期的数据淘汰

3.volatile-random： 从已设置过期时间的数据集中任意选择数据淘汰

4.allkeys-lru： 从数据集中挑选最近最少使用的数据淘汰

5.allkeys-random： 从数据集中任意选择数据淘汰

6.no-enviction： 禁止驱逐数据

**Redis持久化策略:**

1. RDB(基于快照):在指定的时间间隔内将内存中的数据集快照写入磁盘,可设置指定时间操作指定次数生成快照,非正常退出不会持久化(dump.rdb)

2.AOF(追加文件):以日志的形式记录服务器所处理的每一个写、删除操作，查询操作不会记录，以文本的方式记录(更加安全)

**Redis 删除策略:**

定期删除：redis默认是每隔100ms就随机抽取一些设置了过期时间的key，检查其是否过期，如果过期就删除。

惰性删除：定期删除可能会导致很多过期key到了时间并没有被删除掉。所以就有了惰性删除。假如你的过期key,靠定期删除没有被删除掉，还停留在内存里，除非你的系统去查一下那个key，才会被redis给删除掉

**Redis 如何做内存优化:**

尽量使用 Redis 的散列表，把相关的信息放到散列表里面存储，而不是把每个字段单独存储。比如用户对象，应该放到散列表里面再整体存储到 Redis，而不是把用户的姓名、年龄、密码、邮箱等字段分别设置 key 进行存储。

**怎么保证缓存和数据库数据的一致性:**

1. 先删除缓存，然后在更新数据库，如果删除缓存失败，那就不要更新数据库，如果说删除缓存成功，而更新数据库失败，那查询的时候只是从数据库里查了旧的数据而已，这样就能保持数据库与缓存的一致性
2. 用队列的去解决这个问题，创建几个队列，如20个，根据商品的ID去做hash值，然后对队列个数取摸，当有数据更新请求时，先把它丢到队列里去，当更新完后在从队列里去除，如果在更新的过程中，遇到查询请求，则先去缓存里看下有没有数据，如果没有，可以先去队列里看是否有相同商品ID在做更新，如果有也把查询的请求发送到队列里去，然后同步等待缓存更新完成。

**非关系数据和关系型数据库的区别:**

关系型数据库:由二维表及其之间的联系所组成的一个数据组织

优点:

1.易于维护：表结构

2.使用方便：SQL语言通用

3.复杂操作：支持SQL，支持复杂查询

缺点:

1.读写性能比较差

2.固定的表结构，灵活度稍欠

3.硬盘I/O压力大

非系型数据库:一种数据结构化存储方法的集合

优点:

1.存储格式灵活

2.速度快：nosql可以使用硬盘或者随机存储器作为载体，而关系型数据库只能使用硬盘。

3.高扩展性

4.成本低

缺点:

1.不提供sql支持

2.无事务处理

3.数据结构相对复杂

**缓存穿透和缓存击穿:**

本质:都是绕过redis(缓存)直接命中数据库

缓存穿透:查询一个不存在的数据，由于缓存是不命中时的,这将导致这个不存在的数据每次请求都要到数据库去查询

解决方案:

1.布隆过滤器

布隆过滤器相当于一个字节数组,没每位默认值为0,存入时会将数据进行多次哈希操作,得到多个下标,将对应下标改为1

查询时同样进行哈希操作,得到下标的值都为1,则表示布隆过滤器中可能存在,其中有一个为0,则表示不存在,当然也会误判,

所以尽量布隆过滤器的容错率设置小一点

2.第一次查询后,不管是数据不存在，还是系统故障，我们仍然把这个空结果进行缓存

缓存击穿:在并发请求的情况下,查询一个不存在的数据,尽管缓存了,还是会导致所有请求都要查询数据库

解决方案:分布式锁

MySQL

**数据库三范式:**

第一范式:无重复的列,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:属性完全依赖于主键

第三范式:属性不依赖于其它非主属性

**char 和 varchar 的区别:**

**char(n)**：固定长度类型

优点：效率高；缺点：占用空间；适用场景：存储密码的 md5 值，固定长度的，使用 char 非常合适。

**varchar(n)**：可变长度，存储的值是每个值占用的字节再加上一个用来记录其长度的字节的长度

**SQL执行顺序:**

from-->on-->join-->where-->group by-->having-->select-->distinct-->order by-->limit

**union和union all的区别:**

union all:是将连接的两个查询结果表连接起来

union:是将连接的两个查询结果表连接起来并做去重处理?

对重复结果的处理:

union 是取唯一值，记录没有重复

union all是直接连接，取到得是所有值，记录可能有重复

对排序的处理:

union将会按照字段的顺序进行排序

unionall只是简单的将两个结果合并后就返回

效率:

unionall 要比union快很多

**事务特性:**

1、原子性：事务开始后所有操作，要么全部做完，要么全部不做，也就是说事务是一个不可分割的整体

　　 2、一致性：事务开始前和结束后，数据库的完整性约束没有被破坏

　　 3、隔离性：同一时间，只允许一个事务请求同一数据，不同的事务之间彼此没有任何干扰。

　　 4、持久性：事务完成后，事务对数据库的所有更新将被保存到数据库，不能回滚。

**事务隔离级别:**

1.读未提交:所有事务都可以看到其他未提交事务的执行结果(会有:脏读,不可重复读,幻读) ---脏读(能读取到其他事务未提交的数据)

2.读已提交:一个事务只能看见已经提交事务所做的改变(会有:不可重复读,幻读) ---不可重复读(事务a读取事务b已提交的数据,事务b进行回滚导致数据不一致)

3.可重读(MySql默认隔离级别):它确保同一事务的多个实例在并发读取数据时，会看到同样的数据行(会有:幻读) ---幻读(幻读指当用户读取某一范围的数据行时，另一个事务又在该范围内插入了新行)

4.可串行化:这是最高的隔离级别，它通过强制事务排序，使之不可能相互冲突

**如何做 MySQL 的性能优化:**

* 为搜索字段创建索引。
* 避免使用 select \*，列出需要查询的字段。
* 分库分表。
* 选择正确的存储引擎。

**MyISAM和INNODB的区别:**

1.InnoDB支持事务，MyISAM不支持

2.InnoDB支持外键，而MyISAM不支持

3.InnoDB是聚集索引，数据文件是和索引绑在一起的，必须要有主键,而MyISAM是非聚集索引，数据文件是分离的，索引保存的是数据文件的指针

4.Innodb不支持全文索引，而MyISAM支持全文索引

5.MyISAM锁的粒度是表级，而InnoDB支持行级锁定

**MyISAM和INNODB的使用场景:**

INMODB: 由于锁的粒度小，写操作是不会锁定全表的,所以在并发度较高的场景下使用会提升效率的

MySAM: 如果表的读操作远远多于写操作时，并且不需要事务的支持的，可以将 MyIASM 作为数据库引擎的首选

**MySQL 的行锁和表锁:**

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

**乐观锁和悲观锁:**

* 乐观锁：每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在提交更新的时候会判断一下在此期间别人有没有去更新这个数据。
* 悲观锁：每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会阻止，直到这个锁被释放。

数据库的乐观锁需要自己实现，在表里面添加一个 version 字段，每次修改成功值加 1，这样每次修改的时候先对比一下，自己拥有的 version 和数据库现在的 version 是否一致，如果不一致就不修改，这样就实现了乐观锁。

**聚簇索引和非聚簇索引:**

聚簇索引：索引的叶节点就是数据节点。

非聚簇索引: 索引的叶节点仍然是索引节点，只不过有一个指针指向对应的数据块。

(不要当真,随便说说..个人理解)聚集索引: 索引数据放一个文件中，创建表时有主键则在主键生成聚集索引，没有则选择不重复并且不为null的列生产聚集索引，都没有则生成隐藏列生成聚集索引，聚集索引在B+树的叶子节点保存完整数据，在其他字段建立的索引叫二级索引或者辅助索引，会产生回表查询，叶子结点存储的是索引字段和主键,查找\*时找到叶子节点的主键,再去主键树中寻找完整数据  不使用selet \* 就是因为避免回表，使用覆盖索引，覆盖索引就是在二级索引中能拿到数据，不需要再回表

## **什么是存储过程？用什么来调用？**

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

**调用:**

1）可以用一个命令对象来调用存储过程。

2）可以供外部程序调用，比如：java程序。

**优点：**

1）存储过程是预编译过的，执行效率高。

2）存储过程的代码直接存放于数据库中，通过存储过程名直接调用，减少网络通讯。

3）安全性高，执行存储过程需要有一定权限的用户。

4）存储过程可以重复使用，可减少数据库开发人员的工作量。

# Mybatis

**Mybatis:**

1.与JDBC相比，减少了50%以上的代码量

2.最简单的持久化框架、小巧简单易学

3.SQL代码从程序代码中彻底分离出来，可重用

4.提供XML标签，支持编写动态SQL

5.提供映射标签，支持对象与数据库的ORM字段关系映射,省去结果集手动转换

**Mybatis工作流程:**

1.通过SqlSessionFactoryBuilder创建SqlSessionFactory对象

2.通过SqlSessionFactory创建SqlSession对象

3.通过SqlSession拿到Mapper代理对象

4.通过MapperProxy调用Mapper中增删改查的方法

**Mynatis分页插件实现原理:**

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

**MyBatis 逻辑分页和物理分页的区别:**

* 逻辑分页是一次性查询很多数据，然后再在结果中检索分页的数据。这样做弊端是需要消耗大量的内存、有内存溢出的风险、对数据库压力较大。
* 物理分页是从数据库查询指定条数的数据，弥补了一次性全部查出的所有数据的种种缺点，比如需要大量的内存，对数据库查询压力较大等问题。

# **RabbitMQ**

**RabbitMQ持久化:**

1.交换机持久化

durable(true)持久化

2.队列持久化

new Queue(QUEUE\_INFORM\_SMS,true);

3.消息持久化:

channel.basicPublish -> MessageProperties.PERSISTENT\_TEXT\_PLAIN

**RabbitMQ 怎么避免消息丢失:**

* 把消息持久化磁盘，保证服务器重启消息不丢失。
* 每个集群中至少有一个物理磁盘，保证消息落入磁盘。

## **RabbitMQ 有哪些重要的组件:**

* ConnectionFactory（连接管理器）：应用程序与Rabbit之间建立连接的管理器，程序代码中使用。
* Channel（信道）：消息推送使用的通道。
* Exchange（交换器）：用于接受、分配消息。
* Queue（队列）：用于存储生产者的消息。
* RoutingKey（路由键）：用于把生成者的数据分配到交换器上。
* BindingKey（绑定键）：用于把交换器的消息绑定到队列上。

# **多线程**

**进程和线程的区别:**

* 线程是程序执行的最小单位，而进程是操作系统分配资源的最小单位
* 一个进程由一个或多个线程组成，线程是一个进程中代码的不同执行路线。
* 多个线程资源可以共享,进程反之

**创建线程有哪几种方式？**

* 继承 Thread 重写 run 方法；
* 实现 Runnable 接口,重写 run 方法；
* 实现 Callable 接口,重写call();
* 线程池方式创建
* 定时任务也算...

runnable 没有返回值，callable 可以拿到有返回值，runnable不能抛出异常,callable 可以。

**线程的几种状态:**

* 创建状态（New）
* 就绪状态（Runnable）:调用start方法
* 运行状态（Running）:也就是获取时间片执行
* 阻塞状态（Blocked）:锁/sleep/wait/网络延时
* 死亡状态（Dead）

**sleep() 和 wait() 有什么区别？**

* 类的不同：sleep() 来自 Thread，wait() 来自 Object。
* 释放锁：sleep() 不释放锁；wait() 释放锁。
* 用法不同：sleep() 时间到会自动恢复；wait() 可以使用 notify()/notifyAll()直接唤醒。

**notify()和 notifyAll()有什么区别？**

notifyAll()会唤醒所有的线程，notify()之后唤醒一个线程。notifyAll() 调用后，会将全部线程由等待池移到锁池，然后参与锁的竞争，竞争成功则继续执行，如果不成功则留在锁池等待锁被释放后再次参与竞争

**线程通信**

**join():** 等待调用线程执行完之后再执行任务

**yield():** 暂停当前正在执行的线程对象,礼貌的让出时间片，再次获取

**notify / notifyall**

**系统提供的四个线程池(阿里规范不建议使用,造成OOM)**

* 指定长度线程池
* 单个线程池
* 可缓存线程池
* 定时线程

**自定义线程池?**

**new ThreadPoolExecutor ,并指定参数:**

* int corePoolSize: 核心线程数。最小线程数
* int maximumPoolSize: 最大线程数
* long keepAliveTime: 线程连接时间
* TimeUnit unit: 时间单位
* BlockingQueue<Runnable> workQueue: 存储线程的队列
* ThreadFactory threadFactory: 创建线程的工厂
* RejectedExecutionHandler handler: 线程的拒绝策略

**拒绝策略**

* ①丢弃任务并抛出RejectedExecutionException异常；
* ②丢弃任务，但是不抛出异常；
* ③丢弃队列最前面的任务，然后重新尝试执行任务；
* ④由调用线程处理该任务

**在 Java 程序中怎么保证多线程的运行安全?**

* 方法一：使用安全类，比如 Java. util. concurrent 下的类。
* 方法二：使用自动锁 synchronized。
* 方法三：使用手动锁 Lock。

**什么是死锁？**

当线程 A 持有独占锁a，并尝试去获取独占锁 b 的同时，线程 B 持有独占锁 b，并尝试获取独占锁 a 的情况下，就会发生 AB 两个线程由于互相持有对方需要的锁，而发生的阻塞现象，我们称为死锁。

**怎么防止死锁？**

* 尽量使用 tryLock(long timeout, TimeUnit unit)的方法(ReentrantLock、ReentrantReadWriteLock)，设置超时时间，超时可以退出防止死锁。
* 尽量使用 Java. util. concurrent 并发类代替自己手写锁。
* 尽量降低锁的使用粒度，尽量不要几个功能用同一把锁。
* 尽量减少同步的代码块。

## ThreadLocal 是什么？有哪些使用场景？

ThreadLocal 为每个使用该变量的线程提供独立的变量副本，所以每一个线程都可以独立地改变自己的副本，而不会影响其它线程所对应的副本。

ThreadLocal 的经典使用场景是数据库连接和 session 管理等。

**synchronized 和 volatile 的区别是什么？**

* volatile 是变量修饰符；synchronized 是修饰类、方法、代码段。
* volatile 仅能实现变量的修改可见性，不能保证原子性；而 synchronized 则可以保证变量的修改可见性和原子性。
* volatile 不会造成线程的阻塞；synchronized 可能会造成线程的阻塞。

**synchronized 和 Lock 有什么区别？**

* synchronized 可以给类、方法、代码块加锁；而 lock 只能给代码块加锁。
* synchronized 不需要手动获取锁和释放锁，使用简单，发生异常会自动释放锁，不会造成死锁；而 lock 需要自己加锁和释放锁，如果使用不当没有 unLock()去释放锁就会造成死锁。
* 通过 Lock 可以知道有没有成功获取锁，而 synchronized 却无法办到。

**Synchronize的锁升级**

锁的升级：是由偏向锁到轻量级锁，再到重量级锁。

首先一个线程如果需要锁，这时候JVM会给一个互斥锁，该锁其实就是一个偏向锁，它偏向第一个使用它的线程。并且会给定一个threadid，以后通过threadid来确定这个线程。当有第二个线程也需要使用这个锁的时候，会首先去判断上一个线程是否需要偏向锁，如果不需要，则将之改为无所状态，并给新的线程加上偏向锁。如果判断该锁还需要偏向锁，则会去判断上一个线程的内部执行的逻辑是否需要偏向锁，如果不需要就可以释放，然后给新的线程加上偏向锁。其实在判断的时候如果上一个线程还需要偏向锁，当前线程也需要，这时候其实偏向锁就升级成了轻量级锁，轻量级锁将两个线程使用的时间进行分开，一个在使用，另一个就需要进行自旋，然后进行更替。当轻量级锁中一个线程在自旋，自旋多次之后，还是没有分配锁，这时候当前的轻量级锁会膨胀成为重量级锁，该重量级锁会导致除了当前线程以外的线程全部阻塞，会导致降低性能，但是不会使CPU进行空转，资源浪费的情况出现。

# **JVM**

**JVM垃圾回收算法:**

* 标记-清除算法：标记无用对象，然后进行清除回收。缺点：效率不高，无法清除垃圾碎片。
* 标记-整理算法：标记无用对象，让所有存活的对象都向一端移动，然后直接清除掉端边界以外的内存。
* 复制算法：按照容量划分二个大小相等的内存区域，当一块用完的时候将活着的对象复制到另一块上，然后再把已使用的内存空间一次清理掉。缺点：内存使用率不高，只有原来的一半。

新生代采用标记清除算法，老年代采用标记整理算法,守护区采用复制算法。

**如何判断一个对象是否可回收:**

* 引用计数器(不用了?)：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：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组成部分:**

* 类加载器（ClassLoader）
* 运行时数据区（Runtime Data Area）
* 执行引擎（Execution Engine）
* 本地库接口（Native Interface）

  首先通过类加载器（ClassLoader）会把 Java 代码转换成字节码，运行时数据区（Runtime Data Area）再把字节码加载到内存中，而字节码文件只是 JVM 的一套指令集规范，并不能直接交给底层操作系统去执行，因此需要特定的命令解析器执行引擎（Execution Engine），将字节码翻译成底层系统指令，再交由 CPU 去执行，而这个过程中需要调用其他语言的本地库接口（Native Interface）来实现整个程序的功能。

**类加载的执行过程:**

* 加载：根据查找路径找到相应的 class 文件然后导入；
* 检查：检查加载的 class 文件的正确性；
* 准备：给类中的静态变量分配内存空间；
* 解析：虚拟机将常量池中的符号引用替换成直接引用的过程。符号引用就理解为一个标示，而在直接引用直接指向内存中的地址；
* 初始化：对静态变量和静态代码块执行初始化工作。

**类加载器分类:**

启动类加载器（Bootstrap ClassLoader）

扩展类加载器（Extension ClassLoader）

应用程序类加载器（Application ClassLoader）

**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。

**JVM 运行时数据区:**

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。