首页

[Spring优点](#spring优点) [Spring核心](#spring核心) [Spring生命周期](#spring生命周期) [Spring@Resource和Autowired区别](#spring的Resource和Autowired区别)

[Spring设计模式](#spring设计模式) [Spring注解](#spring注解) [微服务,SpringBoot,SpringCloud理解](#微服务和springboot和springcloud区别) [SpringBoot自动装配](#Springboot自动配置)

[Spring-boot-Mvc区别](#Spring，boot，Mvc区别) [Spring@Transactional失效](#Transactional失效) [spring线程并发问题](#spring线程并发问题)

[MVC执行流程](#MVC执行流程) [MVC注解](#MVC注解) [过滤器-拦截器-监听器](#过滤器拦截器监听器) [它们的区别](#他们的区别)

[Spring事务自定义回滚](#自定义回滚)  [Spring事务隔离级别](#Spring事务隔离级别)

**Redis专题**

[Redis基础](#Redis基础)  [Redis过期策略](#Redis过期策略) [Redis的缓存雪崩，缓存击穿，缓存穿透](#Redis三个问题) [Redis持久化](#Redis持久化)

[RDB和AOF区别](#RDB和AOF区别) [Redis主从同步](#Redis主从同步) [全量同步-增量同步-是否第一次同步？](#全量同步增量同步) [主从同步优化](#主从同步优化)

[Redis哨兵](#Redis哨兵) [Redis分片集群](#Redis分片集群) [Redis缓冲并发](#Redis缓存并发) [Redis数据一致性](#Redis数据一致性) [**Redis集群集群**](#Redis集群基础)

**微服务专题**

[Eureka和nacos区别](#eureka和nacos区别) [nacos配置热更新](#nacos配置热更新) [nacos集群搭建步骤](#nacos集群搭建步骤) [nacos存储问题](#Nacos存储问题)

[Feign远程调用](#Feign远程调用) [Feign优化](#Feign优化) [微服务雪崩及解决](#微服务雪崩及解决) [sentinel流控-限流](#流量控制限流) [sentinel线程隔离](#隔离和降级)

[Sentinel熔断](#熔断降级) [规则持久化](#规则持久化)

[MongoDB特点](#MongoDB特点) [MongoDB数据类型](#MongoDB数据类型) [MongoDB优缺点](#MongoDB优缺点)

分布式专题

[分布式事务基础](#分布式事务基础) [Seata基础](#Seata基础) [Seata-XA模式](#SeataXA模式) [Seata-AT模式](#SeataAT模式) [AT和XA区别](#AT和XA的区别)

[Seata-TCC模式](#SeataTCC模式) [TCC空回滚和业务悬挂](#事务空回滚和业务悬挂) [四种模式对比](#四种模式对比)

[Gateway服务网关](#gateway服务网关) [filter过滤器及顺序](#fiter过滤器及顺序) [网关过滤器工厂](#网关过滤器工厂) [全局过滤器](#全局过滤器)  [跨域问题](#跨域问题)

Kafka专题

[消息中间件优点](#消息中间件优点) [Kafka如何保证消息可靠性](#Kafka保证消息可靠性) [Kafka的优点和缺点](#Kafka的优点和缺点) [如何保证消息幂等](#如何保证消息幂等)

[为什么要使用消息队列？为什么要使用kafka？](#消息队列和kafka的作用) kafka的使用

RabbitMQ专题

[RabbitMQ五种模型](#RabbitMQ中的几种模型) [注解声明队列和交换机](#注解声明队列和交换机) [为什么使用消息队列（优缺点）](#为什么使用消息队列)

[RabbitMQ持久化](#RabbitMQ的持久化) [RabbitMQ如何保证消息的可靠性？](#MQ如何保证消息的可靠性) [RabbitMQ实现延迟消息](#实现发送一个延迟消息)

[惰性队列解决消息堆积问题](#解决消息堆积问题)

**Mysql专题**

[Mysql6种索引失效](#索引失效) [sql的select语句执行顺序](#完整执行顺序) [where和having的区别](#where和having的区别)

[5种索引类型和五种约束](#索引类型和约束) [**mysql优化**](#Mysql优化) [mysql事务](#Mysql事务) [数据库设计基本步骤](#数据库的基本设计步骤)

[Mysql的引擎myisam和innoDB区别](#Mysql引擎) [主从复制](#主从复制) [mysql的mvcc](#Mysql的mvcc)

[代码实现读写分离](#代码实现读写分离) [B树，B-树，B+数，B\*数之间的关系](#B数等之间的关系) [超大分页怎么处理](#超大分页怎么处理) ?

[mysql索引的优势和劣势](#索引的优势和劣势) [如果一条SQL语句执行很慢 ,如何找到慢的原因 ?](#执行很慢)

基础知识专题

[多线程](#多线程状态) [多线程创建7大参数](#多线程创建7大参数) [悲观锁和乐观锁](#悲观锁和乐观锁) [基础集合](#基础集合) [hashMap所有](#hashmap)

[简述java垃圾回收机制](#Java垃圾回收机制) [GC是什么？为什么要GC](#GC是什么？为什么要GC) [GC的原理是什么？如何手动进行GC？](#垃圾回收器的原理)

[类加载的执行过程](#类加载的执行过程)  [JVM调优的工具](#JVM调优工具)  [什么是程序计数器](#程序计数器)  [什么双亲委派模型](#双亲委派模型)

[延迟发布审核](#延迟发布审核)

Es专题

[什么是es以及基本概念](#Elasticsearch)  [倒排索引和正排索引](#倒排索引和正牌索引) [text和keyword区别](#Text和keyword区别)  [query和filter区别](#Query和filter的区别)

[es写数据的过程(ES的写入流程：)](#es写数据的过程)  [Es在高并发下如何保证读写一致性？](#如何保证读写一致性)

[Elasticsearch的分布式原理](#Elasticsearch的分布式原理)  [ES的深度分页与滚动搜索scroll](#ES的深度分页与滚动搜索)

**Spring优点：**

Spring是一个开源的免费,轻量级,非入侵式的框架,支持事务的处理,和对其他框架整合支持

**Spring核心：**

1. 控制反转(ioc):传统的java开发模式中,当需要一个对象时,我们需要自己new或者getInstance等直接或者间接调用构造方法创建一个对象,而在Spring开发模式中,Spring容器使用工厂模式为我们创建需要的对象,直接调用Spring为我们提供即可,这就是控制反转。（通俗来讲就是将new对象的权力交给Spring，我们从Spring的IOC容器中获取对象即可使用，IOC容器负责对象的创建，初始化等一系列工作，被创建或被管理的对象在IOC容器中统称为Bean）
2. 依赖注入（DI）：Spring使用JavaBean对象的set方法或者带参数的构造方法为我们在创建所需对象时将其属性自动设置所需要的值的过程就是依赖注入（Spring容器为调用者的成员变量赋值，而不需要主动获取被依赖对象，建立bean与bean之间的依赖关系的整个过程，成为依赖注入）
3. 面向切面编程（AOP）：在面向对象编程OOP中，我们将事务纵向抽象成对象，在切面编程中，将一个个对象相似点横向抽象成切面，然后增强切面，对这个切面进行一些如，权限验证，事务管理，记录日志等公用操作（底层就是动态代理）

[返回首页](#首页)

**Spring生命周期：**

有两个重要的bean生命周期方法，第一个是setup，他是在容器加载bean的时候被调用，可以在这里打印定制代码，第二个是tearDown，容器在卸载类的时候运行

1. Spring容器从xml文件中或者注解中读取bean的定义，并实例话bean
2. Spring根据bean的定义填充所有的属性
3. 如果bean实现了beanNameAware接口，Spring传递bena的id到setBeanName方法
4. 如果bean实现了BeanFactoryAware接口，Spring传递beanfactory给setbeanfactory方法
5. 如果有任何与bean相关联的BeanPostProcessors，Spring会在postProcesserBeforeINitication方法内调用他们
6. 如果bean实现intializingBean了调用它的afterPropertyset方法，如果bean声明了初始化方法，调用此初始化方法
7. 如果有BeanPostProcessors和Bean关联，这些bean的PostProcessAfterinitialization方法将被调用
8. 如果bean实现了disposablebean，它将调用destroy方法

[返回首页](#首页)

**Spring@Resource和Autowired区别：**

@Resource和@AutoWored都是做bean的注入时使用，其实@Resource并不是Spring的注解他的包不是Spring的包，需要导入，但是Spring支持这个注解的注入

1. 共同点两者都可以写在字段和setter方法上。两者如果都写在字段上，就不需要在写setter方法
2. 不同点@AutoWwired为Spring提供的注解，导入包为Springframework。Beans，factory。Annotation。Autowired，只按照类型bytype注入，想通过byName装配可以结合@Qualifier注解一起使用
3. @resource默认按照Byname自动注入，由J2EE提供，需要导入javax。Annotition。Resource，它有两个重要的属性name和type，属性写name就按照byname来装配，属性为type就那招bytype来装配，不写默认byname

[返回首页](#首页)

**Spring设计模式：**

1. 工厂模式：在工厂模式中，我们在创建对象时不会对客户端暴露逻辑，通过使用一个共同的接口来指向新创建的对象，Spring的beanFactory就是简单工厂模式的体现，用来创建对象的实例
2. 单例模式：Bean默认为单例模式，Spring的配置文件定义的
3. 代理模式：Spring中有两种代理方式，若目标对象实现了若干接口，Spring使用JDK的Java。Lang。Reflect。Proxy类代理，如果目标对象没有实现任何接口，Spring使用CGLIB库生成目标对象的子类。Spring的AOP功能用到了JDK的动态代理和CGLIB字节码生成技术
4. 模板方法：用来解决代码重复的问题。比如RestTemplate，JmsTemplate，KafkaTemplate，mongoTemplate，RedisTemplate等
5. 观察者模式：定义对象键一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖的对象都会通知被自动更新

[返回首页](#首页)

**Spring的注解：**

1. Controller作用在控制器上
2. Service作用在业务层类上
3. Repository作用在数据访问层类上
4. Componet作用在类上，泛指组件，当组件不好归类时，可以使用这个注解
5. Bean作用在方法上声明Bean对象
6. Scope声明bean的作用范围单例多例
7. autowired按类型注入bean
8. resource按名称注入bean
9. PropertiesSource加载配置文件
10. import导入其他的配置类
11. Configration标注在配置类上
12. requestMapping作用在类或者方法上,用在类上,表示类中的所有响应的方法都是以该地址作为父类路径来访问,用在方法上表示表示方法的访问路径
13. RestController作用在类上

[返回首页](#首页)

**微服务，SpringBoot，SpringCloud的理解：**

微服务实际是一种分布式,因为先有了单体应用,再有集群,然后才有的分布式,他的优点在于他的各个模块耦合度比较低,另外分布式的性能比较高,相对一台机器来说缺点1.难以管理,因为不在是单体应用了,部署也比较麻烦,它需要网路之间的不同机器,相互进行一些数据交流,网络的一些阻塞就相对来说比较大,网络不好影响很大

SpringBoot可以快速构建基于Spring的项目 内嵌web服务器,提供了starter简化maven配置以及依赖管理,自动配置SpringBean,使得使用更加简单,更关注快速方便的开发单个个体微服务

使编码变简单

使配置变简单

使部署变简单

使监控变简单

SpringCloud是微服务的一个框架关注全局微服务的协调,整理,治理,它将SpringBoot开发的单体整合并管理起来,Spring'boot可以离开Springcloud独立开发项目,但是Springcloud离不开Springboot

[返回首页](#首页)

**SpringBoot的自动装配原理：**

Springboot启动的时候通过enableAutoConfiguration注解找到META—INF/spring.factories配置文件中所有的自动配置类并对其加载，这些配置类的类名都是以AutoConfiguration结尾来命名的，实际上就是一个javaConfig形式的Spring容器配置类，他们都有一个@EnableConfigurationProperties的注解，通过这个注解启动XXXProperties命名的类去加载全局配置中的属性，然后将所有自动配置类加载到Spring容器中

[返回首页](#首页)

**Spring-boot-MVC的区别：**

Spring的完整名字应该是SpringFramework。它提供了多个模块，ioc，aop，mvc等所以mvc是spring众多模块中的一个而springboot是构造在spring之上的boot启动器，意在更容易的配置一个spring项目

[返回首页](#首页)

因为Spring事务是基于代理来实现的，所以某个加了@Transactional的发方法只有是被代理对象调用时，这个注解才会生效，所有如果是被代理对象来调用这个方法，那么这个@Transactional是不会失效的。同时如果某个方法是protected，private修饰 事务就会失效，并且还不会有任何报错，很容易忽视

[返回首页](#首页)

一般情况下，只有无状态的bean才可以在多线程环境下共享，在spring中绝大部分bean都可以声明为singleton单例作用域，spring对一些bean中非线程安全状态采用threadLocal进行处理，解决线程安全问题

[返回首页](#首页)

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返回给DispatcherServlet
4. DispatcherServlet通过HandlerAdapter处理器适配器调用处理器
5. 执行处理器（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ViewReslover传给ViewReslover视图解析器
9. ViewReslover视图解析器解析后返回具体view
10. DispatcherServlet对view进行渲染试图（将模型数据填充至视图中）
11. DispatcherServlet响应用户

[返回首页](#首页)

**MVC注解**

1. RestController：Controller和responseBody组合注解 处理请求地址映射
2. RequestMapping：接口请求映射
3. GetMapping：Get请求方式映射
4. PutMapping：Put请求方式映射
5. PostMapping：Put请求方式映射
6. DeleteMapping：Delete请求方式映射
7. ResponseBody：响应数据自动转JSON
8. RequestBody：接受请求体数据，JSON会自动转对象
9. RequestParam: 接受请求参数
10. RequestHeader：接受请求头
11. PathVariable: 接受路径参数 声明路径变量
12. Resource和Autowired：做bean的注入

[返回首页](#首页)

**过滤器**：

做一些过滤操作，获取我们想要获取的数据：比如在过滤器中修改字符编码，修改httpServletRequest的一些参数，AddRequestHeader添加请求头数据

**监听器**：

主要就是Spring容器启动的时候加载一些数据，最常用的功能就是开发权限系统的时候，当监听器启动的时候，从数据库加载权限url。感知到包括request请求域，session会话域和application的初始化和属性的变化

**拦截器：**

SpringMVC中的Interceptor拦截器主要作用就是拦截用户的url请求，并在执行handler方法的前中后加入某些特殊请求，类似与servlet里面的过滤器，依赖与web框架的调用，因此可以使用Spring的依赖注入进行一些业务操作，同时一个拦截器实例在一个controller生命周期之类可以多次调用，但是缺点是只能对controller请求进行拦截，对其他的一些比如直接访问静态资源的请求没办法进行拦截处理

[返回首页](#首页)

**过滤器和拦截器区别：**

1. 拦截器是基于java的反射机制的，而过滤器是基于函数回调
2. 拦截器不依赖servlet容器，过滤器依赖servlet容器
3. 拦截器只能对action请求起作用，而过滤器则可以对几乎所有的请求起作用
4. 拦截器可以访问action上下文，值栈里的对象，而过滤器不能访问
5. 在action的生命周期中，拦截器可以多次被调用，而过滤器只能在容器初始化是被调用一次
6. 拦截器可以获得IOC容器中的各个bean，而过滤器就不行，这点很重要，在拦截器里注入一个service，可以调用业务逻辑

[返回首页](#首页)

**什么是Redis：**

Redis是一个基于内存的key-value结构数据库。是互联网技术领域使用最广泛的存储中间件

性能极高，还有丰富的数据类型，并且所有的操作都是原子性的

**Redis的五种数据类型：**

1. String：key-value 常用命令：get，set，incr加一，decr减一
2. Hash: largeKey-key-values 常用命令：hset，hget
3. List: 有顺序可重复 常用命令：lpush/rpush，lrange list，lpop/rpop
4. Set：无顺序，不能重复 常用命令：sadd， smembers set，srem set
5. Zset：有顺序，不能重复 常用命令：zadd zset，zrange zset ，zrem zset

[返回首页](#首页)

**Redis过期策略：**

1. 定时删除：创建定时器对key进行删除 定时器创建耗时还影响性能 设置key的过期时间
2. 惰性删除：key过期的时候不删除，每次从数据库获取key时检查是否过期，过期删除，返回null
3. 定期删除：每隔一段时间执行一次删除过期key操作

[返回首页](#首页)

**Redis的缓存雪崩，缓存击穿，缓存穿透：**

缓存雪崩：（缓存失效）指的时大量的缓存在同一时间失效，到时DB的压力了瞬间飙升

造成这种现象的原因时key的过期时间都设置成一样的了

如何避免？

1. 不同的key设置不同的过期时间，让缓存失效的时间点尽量均匀
2. 做二级缓存，a为原始缓存，b为拷贝缓存，a失效时，可以访问b，a失效时间为短期，b设置为长期
3. 跑定时任务，定时去刷缓存，在失效前就刷进去
4. 缓存永不过期
5. 部署集群，将热点的key平均分布在不同的redis节点点

缓存穿透：

缓存穿透指查询一条数据库和缓存都没有的一条数据，就是一直查询数据库，对数据库的方访问压力就会增大

如何解决？

1. 对查询结果为空的情况也进行缓存，缓存时间设置短一点，有该key对应数据了就删除
2. 布置过滤器，对一定不存在的key进行过滤
3. 拉黑ip，一般出现这种的都是恶意请求
4. 对参数进行效验，不合法直接return

缓存击穿：

缓存击穿是指缓存中没有但数据库中有的数据（一般是缓存时间到期），这时由于并发用户特别多，同时读缓存没读到，又同时去数据库去取数据，引器数据库压力瞬间增大，造成过大压力

如何解决？

1. 热点数据提前预热
2. 设置热点数据永不过期
3. 加锁限流

[返回首页](#首页)

**Redis持久化：**

Reids提供了两种数据持久化方式，分别是RDB和AOF，默认是RDB

**RDB持久化**是指在指定的时间间隔内将内存中数据集快照写入磁盘（简单来说就是把内存中的所有数据记录到磁盘中，当redis实例故障重启后，从磁盘读取快照文件，恢复数据，快照文件就是rdb文件，默认保存在当前运行目录）

**优点**：只有一个rdb文件，随时备份，比aof文件小，加载效率高，提供fork子进程，共享内存空间，不阻塞主进程，io操作比较少

**执行RBD持久化的四种情况**

1. 执行save命令
2. 执行bgsave命令
3. Redis停机时 默认时redis停机时
4. 触发RDB条件时

**缺点**：RDB执行间隔时间长，两次RDB之间写入数据有丢失的风险，子进程压缩，写出RDB文件都比较耗时

**RDB快照基本原理**，RDB持久化主要时通过save和bgsave两个命令对redis数据库中当前数据做snapshot并生成rdb文件来实现的，其中save时阻塞的bgsave时非阻塞的，通过fork一个子进程来完成，并在redis启动的时候会检测rbd文件，然后载入rdb中未过期的数据到服务器中

**AOF持久化**就是将redis处理的每一个写命令都会记录在AOF文件，可以看做命令日志文件，在redis服务器启动之初会读取该文件来重新构建数据库，以保证启动后数据库中的数据时完整的

默认关闭，可以通过修改redisfonf配置文件来开启AOF 记录的频率也是通过这个文件

优点：每次改动同步数据安全性好，APPEND方式追加日志，不会对旧日志文件产生影响

基本原理：

AOF持久化时通过将储存每次执行的客户端命令，然后由一个伪客户端来执行这些命令将数据写入服务器的方式实现的，一共分为命令追加（append）文件写入，文件同步（sync）三个步骤完成的，当有修改，删除操作时，服务器在执行完之后以协议格式将执行的写命令追加到服务器状态的aofbuf缓冲区的末尾

[返回首页](#首页)

**RDB和AOF的区别：**

1. 持久化方式：RDB定时对整个内存做快照，aof记录每一次执行的命令
2. 数据完整性：rdb不完整，两次备份之间会丢失，aof相对完整，取决于刷盘celue
3. 宕机恢复速度：rdb很快，aof很慢
4. 系统资源占用：rdb高，大量cpu和内存消耗，低，主要是磁盘io资源但aof重写时会占用大量cpu和内存资源
5. Rdb可以理解为一种全量数据更新机制，aof可以理解为是一种增量的更新机制
6. Rdb适合数据量大，写命令少的场景

[返回首页](#首页)

**Redis集群的主从同步：**

Redis的主从同步机制，允许Slave从master哪里通过网络传输拷贝到完整的数据备份，从而达到主从机制，

1. 主数据库可以进行读写操作，当发生写操作的时候自动将数据同步到数据库，而从数据库一般是只读的，并接受主数据库同步过来的数据
2. 一个主数据库可以有多个从数据库，而一个从数据库只能有一个主数据库
3. 第一次同步时，主节点做一次bgsave操作，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，赋值节点接受完成后将rdb镜像加载到内存，加载完成后，在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

[返回首页](#首页)

**全量同步：**

Master判断一个节点是否是第一次同步的依据，就是看replicationId是否一致。

主从第一次建立连接时，会执行全量同步，将Master节点的所有数据都拷贝给Slave节点。流程：

1. Slave节点请求增量同步
2. Master节点判断replicationId是否一致，发现不一致，拒绝增量同步
3. Master将完整内存数据生成rdb，发送rdb到slave
4. Slave清空本地数据，加载maseter的Rdb
5. Master将Rdb期间的命令记录在replication-baklog中，并持续将log中的命令发送给slave
6. Slave执行接受到的命令，保持master之间的同步

**增量同步：**

只更新slave和master存在差异的部分数据，依靠全量同步时的repl-baklog文件，这个文件时一个固定带线啊哦的数组，只不过数组时环形，角标到达数组末尾后，会再次从0开始读写，数组头部的数据就会被覆盖

注意，如果slave断开时间过久，导致尚未备份的数据被覆盖，则无法基于log做增量同步，只能再次做全量同步了

[返回首页](#首页)

**主从同步优化：**

主从同步可以保证主从数据的一致性，非常重要，可以从以下几个方面来优化Redis主从集群

1. 在master中配置repl-diskless-sync yes启用无磁盘复制，避免全量同步时的磁盘io
2. Redis单节点上的内存占用不要太大，减少Rdb导致的过多磁盘io
3. 适当提高repl-baklog的大小，发现slave宕机时尽快实现故障恢复，尽可能避免全量同步
4. 限制一个master上的slave节点数量，如果实在是太多slave，则可以采用主-从-从链式结构，减少master压力

[返回首页](#首页)

**Redis哨兵：**

哨兵的作用：

1. 监控：sentinel会不断检查master和slave是否按预期工作
2. 自动故障恢复：如果master故障，sentinel会将一个slave提升为master。当故障实例恢复后以新的master为主
3. 通知：sentinel充当redis客户端的服务来源，当集群发生故障转移，会将最新信息推送redis的客户端

原理：

如何判断一个redis实例是否健康？

1. 每隔一秒发送一次ping命令，如果超过一定时间没有相向则认为主观下线
2. 如果大多数sentinel都认为实例主管下线，则判定服务下线 指定数量最好超过sentinel实例数量的一半

故障转移的步骤？

1. 首先选定一个slave作为新的master，执行slaveof one
2. 然后让所有节点都执行slaveof新master
3. 修改故障节点配置，添加slaveof新master

[返回首页](#首页)

**Redis分片集群：**

主从和哨兵只能解决高可用，高并发读的问题，但是还有两个问题没有解决：

海量数据存储问题

高并发写的问题

特征：

1. 集群中有多个master，每个master保存不同数据
2. 每个master都可以有多个slave节点
3. Master之间通过ping监测彼此健康状态
4. 客户端请求可以访问集群任意节点，最终都会被转发到正确节点

Redis如何判断某个key在那个实例？

1. 将16384个插槽分配到不同的实例
2. 根据key的有效部分计算哈希值，对16384取余
3. 余数作为插槽，寻找插槽所在实例即可

[返回首页](#首页)

**Redis缓存并发：**

有时候网站并发访问高，一个缓存如果失效，可能出现多个进程同时查询DB，同时设置缓存的情况，如果并发确实很大，这也可能造成DB压力过大，还有缓存频繁更新的问题。一般处理方案实在查DB的时候进行加锁，如果key不存在，就加锁，然后查DB入缓存，然后解锁，其他进程如果发现有锁就等待，然后等解锁后在查缓存或者进如DB查询

[返回首页](#首页)

**Redis缓存数据与数据库一致性问题的解决：**

单点情况：选择先更新数据库，在删除缓存，加上更新后延迟删除缓存完美解决

更新的四种

1. 先更新缓存，在更新数据库，出问题了数据库回滚，redis怎么回滚？
2. 多线程情况下数据库出现脏写，但是redis可能更新的就是脏写的数据
3. 多线程情况下会产生数据不一致的问题
4. 并发下即使a请求将旧的值写入redis，但是请求b还是会将缓存redis删除掉的

非常小的可能也会出现，更新比查询b更新了就删除缓存，但是a又把旧值缓存了

解决办法，延迟删除，更新数据后，睡眠几百毫秒，然后在去删除缓存

[返回首页](#首页)

**Redis集群集群：**

集群是指将几台服务器集中在一起，实现同一业务：

目的：高可用，负载均衡，易扩展，数据安全，性能提升

技术：集群地址（虚拟id),网络通信（监控信息）

功能：负载均衡，读写分离，故障转移

[返回首页](#首页)

**Eureka注册中心和nacos注册配置中心:**

nacos和eureka整体结构类似，服务注册，服务拉取，心跳等待，但是也存在一些差异

共同点：都支持服务注册和服务拉取，都支持服务提供者心跳方式做健康检查

不同点：

1. nacos支持服务端主动检测提供者状态，临时实例采用心跳模式，非临时实例采用主动检查模式
2. 临时实例心跳不正常会被剔除，非临时实例则不会被剔除
3. Nacos支持服务列表变更的消息推送模式，服务列表更新更及时
4. Nacos集群默认采用AP方式，当集群中存在非临时实例时，采用cp模式，eureka采用ap方式

[返回首页](#首页)

**nacos配置热更新：**

有两种方式

1. 在@value注入的变量所在类添加注解@RefreshScope
2. 使用@ConfigurationProperties注解代替@Value注解

@ConfigurationProperties（prefix=“pattern”）

[返回首页](#首页)

**nacos集群搭建步骤：**

1. 搭建数据库，初始化数据库表结构
2. 下载nacos安装包
3. 配置nacos
4. 启动nacos集群
5. Nginx反向代理

[返回首页](#首页)

Nacos存储问题：

Nacos默认数据存储在内嵌数据库Derby中，不属于生产可用的数据库，官方推荐的最佳实践是使用带有主从的高可用数据库集群，比如搭建mysql主从复制集群

优化：实际部署时，需要给做反向代理的nginx服务器设置一个域名，这样后续如果有服务器迁移nacos的客户端也无需要更改配置，nacos的各个节点应该部署到多个不同服务器，做好容灾和隔离

[返回首页](#首页)

**Feign远程调用：**

feign是一个声明式的http客户端，作用就是帮助我们优雅的实现http请求的发送

在启动类添加注解开启feign的功能：

@enableFeignClients（basepackages=com.heima.）

包扫描问题解决:

1. 指定Feign应该扫描的包，EnableFeignClients（basePackages=cn.itcast.feign.clients
2. 指定需要加载的client接口enableFeignClient（client=userclient.Class)

在远程调用方法上添加FeignClient（服务名）

[返回首页](#首页)

Feign优化：

1. 日志级别尽量用basic
2. 使用HttpClient或OKHttp代替URLConnection，feign底层发起http请求，依赖于其他的框架，起底层客户端实现包括，URLconnection默认实现不支持连接池，ApacheHttpClient支持连接池，OKHttp支持连接
3. 因此提高feign的性能主要手段就是使用连接池代替默认的URLConnection，加依赖feign-httpclient配置连接池
4. feign可以将client抽取为独立模块，并且把接口有关的pojo默认的feign配置都放在这个模块中，提供给所有消费者使用

[返回首页](#首页)

**微服务雪崩及解决方案：**

微服务之间相互调用，因为调用链中的一个服务故障，引起整个链路都无法访问的情况

解决方案：

超时处理，线程隔离，降级熔断是在部分服务故障时，将故障控制在一定范围，避免雪崩，是一种补救措施。限流是对服务的保护，避免因瞬间高并发流量而导致服务故障，进而避免雪崩，是一种预防措施

[返回首页](#首页)

**流量控制--限流：**

直接：对当前资源限流

关联：高优先级资源触发阈值，对低优先级资源限流

链路：阈值统计时，只统计从指定资源进入当前资源的请求，是对请求来源的限流

**流空效果有那些？**

快速失败：QPS超过阈值时，拒绝新的请求

Warm up:QPS超过阈值，拒绝新的请求，QPS阈值是逐渐提升的，可以避免冷启动时高并发导致服务宕机

排队等待：请求会进入队列，按照阈值允许的时间间隔依次执行请求；如果请求预期等待时大于超时时间，直接拒绝

什么是热点参数限流？

之前的限流是统计访问某个资源的所有请求，判断是否超过QPS阈值，而热点参数限流是分别统计参数值相同的请求，判断是否超过QPS阈值

注意：热点参数限流对默认的SpringMVC资源无效，需要利用sentinelResource注解标记资源

[返回首页](#首页)

**线程隔离：**

限流是一种预防措施，虽然限流可以尽量避免因高并发而引起的服务故障，但服务还会因为其他原因而故障，而要将这些故障控制在一定范围，避免雪崩，就要靠线程隔离和熔断降级手段了

线程隔离就是在调用者在调用服务提供者时，给每个调用的请求分配独立线程池，出现故障时，最多消耗这个线程池内资源，避免把调用者的所有的资源耗尽

线程池隔离：给每个服务调用业务分配一个线程池，利用线程池本身实现隔离效果

特点：基于线程池模式，有额外开销，但隔离控制更强

信号量隔离：不创建线程池，而是计数器模式，记录业务使用的线程数量，达到信号量上限时，禁止新的请求

特点：基于计数器模式，简单，开销小

[返回首页](#首页)

**熔断降级：**

熔断降级是解决雪崩问题的重要手段，其思路是由断路器，统计服务调用的异常比例，慢请求比例，如果超过阈值则会熔断该服务，即拦截访问该服务的一切请求，而当服务恢复时，断路器会放行访问该服务的请求

断路器控制熔断和放行是通过**状态机**来完成的

状态机包括三个状态：

1. Closed：关闭状态，断路器放行所有请求，并开始统计异常比例，慢请求比例，超过阈值则切换到open状态
2. Open：打开状态，服务调用被熔断，访问被熔断服务的请求会被拒绝，快速失败，直接走降级逻辑，Open状态5秒后会进入half-open状态
3. Half-open：半开状态，会放行一次请求，根据执行结果来判断接下来的操作，请求成功则切换到closed状态，请求失败，则切换到open状态

熔断降级策略：

断路器熔断策略由三种：慢调用，异常比例，异常数

[返回首页](#首页)

**规则持久化：**

现在sentinel的所有规则都是内存存储，重启后所有规则都会丢失，在生产环境下，我们必须确保这些规则的持久化，避免丢失。

规则管理模式

规则能否持久化，取决于规则管理模式，sentinel支持规则管理模式

1. 原始模式：sentinel的默认模式，将规则保存在内存，重启服务会丢失
2. Pull模式：控制台将配置的规则推送到sentinel客户端，而客户端会将配置规则保存在本地文件或数据库中，以后对定时去本地文件或数据库中查询，更新本地规则
3. Push模式；sentinelDashboard默认不支持nacos的持久化，需要修改源码，借助nacos实现持久化

[返回首页](#首页)

**MongoDB的特点：**

1. 面向集合，文档存储，易于存储对象类型的数据
2. 模式自由
3. 支持动态查询
4. 支持完全索引，包含内部对象
5. 支持复制和故障恢复
6. 使用高效的二进制数据存储，包括大型对象（如视频等）
7. 自动处理碎片，以支持云计算层次的扩展性
8. 支持PHP，java，c，c++等语言驱动程序
9. 文件存储格式为BSON（一种JSON的扩展）

[返回首页](#首页)

**MongoDB数据类型：**

1. 数据格式：BOSN（aa:bb）
2. Null:用于表示控制或者不存在的字段
3. 布尔型：true或者false
4. 数值：shell默认使用64为浮点型数值
5. 字符串：UTF—8字符串都可以表示为字符串类型的数据
6. 日期：日期被存储为自新纪元依赖经过的毫秒数，不存储时区
7. 正则表达式：查询时，使用正则表达式作为限定条件，语法与js的正则表达式相同
8. 数组：数据列表或数据集可以表示为数组
9. 内嵌文档：文档可以嵌套其他文档，被嵌套的文档作为值来处理
10. 对象id：对象id是一个12字节的字符串，是文档的唯一标识
11. 二进制数据：二进制数据是一个任意字节的字符串，他不能直接在shell中使用

[返回首页](#首页)

**MongoDB优缺点：**

1. 弱一致性（最终一致），更能保证用户的访问速度
2. 文档结构的存储方式，能够更快捷的获取
3. 内置GridFS，高效存储二进制大对象（照片视频）
4. 支持复制集，主备，互为主备，自动分片等特性
5. 动态查询
6. 全索引支持，扩展到内部对象和内嵌数组

缺点：

1. 不支持事务
2. MongDB占用空间过大，维护工具不够成熟

**注意:** Mongo的BSON数据格式非常适合文档格式化的存储和查询

重要数据：mysql，一般数据：mongodb

[返回首页](#首页)

**分布式事务基础：**

本地事务就是传统的单机事务，在传统数据库中，必须要满足四个原子：ACID

1. 原子性：事务中的所有操作，要么全部成功，要么全部失败
2. 一致性：要保证数据库内部完整性约束，声明性约束
3. 隔离性：对同一资源操作的事务不能同时发生
4. 持久性：对数据库做的一切修改将永久保存

分布式事务就是指不是在单个服务或单个数据库架构下，产生的事务，例如跨数据源的分布式事务，跨服务的分布式事务，综合情况

分布式事务最大的问题就是各个子事务的一致性问题，因此可以借鉴CAP定理和BASE理论，由两种解决思路

ap模式：各子事务分别执行和提交，允许出现结果不一致，然后采用弥补措施恢复数据即可，实现最终一致

Cp模式：各个子事务执行后互相等待，同时提交，同时回滚，达成强一致，但事务等待过程中，处于弱可用状态

但不管那一种模式，都需要在子系统事务之间相互通讯，协调事务状态，也就是需要一个事务协调者

[返回首页](#首页)

**Seata基础：**

Seata事务管理中有三个重要的角色：

TC：事务协调者，维护全局和分支事务的状态，协调全局事务提交或回滚

TM：事务管理器，定义全局事务的范围，开始全局事务，提交或回滚全局事务

RM：资源管理器，管理分支事务处理的资源，与TC交谈以注册分支事务和报告分支事务的状态，并驱动分支事务提交或回滚

Seata基于上述架构提供了四种不同的分布式事务解决方案：

XA模式：强一致性分阶段事务模式，牺牲了一定的可用性，无业务侵入

TCC模式：最终一致的分阶段事务模式，有业务侵入

AT模式：最终一致的分阶段事务模式，无业务侵入，也是Seata的默认模式

SAGA模式：长事务模式，有业务侵入  
无论哪种方案，都离不开TC，也就是事务的协调者

[返回首页](#首页)

**Seata-XA模式：**

**一阶段：**

1. 事务协调者通过每个事务参与者执行本地事务
2. 本地事务执行完成以后报告事务执行状态给事务协调者，此时事务不提交，继续持有数据库锁

**二阶段：**

事务协调者基于一阶段的报告来判断下一步操作

如果一阶段都成功，则通知所有事务参与者提交事务

如果一阶段任何一个参与者失败，则通知所有事务参与者回滚事务

**优点：**

事务的强一致性，满足ACID原则

常用数据库都支持，实现简单，并且没有代码侵入

**缺点：**

因为一阶段需要锁定数据库资源，等待二阶段结束才释放，性能较差

依赖关系型数据库实现事务

**XA模式实现：**

Seata的starter已经完成了XA模式的自动装配，实现非常简单

1. 修改application.yml文件（每个参与事务的微服务），开启XA模式，添加属性seata：data-source-proxy-mode：XA
2. 给发起全局事务的入口方法添加GlabalTRansactional注解

[返回首页](#首页)

**Seata-AT模式：**

AT模式下，当前分支事务执行流程如下

**一阶段：**

1. TM发起并注册全局事务到TC
2. TM调用分支事务
3. 分支事务准备执行业务SQL
4. RM拦截业务SQL，根据where条件查询原始数据，形成快照
5. RM执行业务SQL，提交本地事务，释放数据库锁
6. RM报告本地事务状态给TC

**二阶段：**

1. TM通知TC事务结束
2. TC检查分支事务状态

如果都成功，则立即删除快照

如果有分支事务失败，需要回滚，读取快照数据，将快照恢复到数据库

**优点：**

一阶段完成直接提交事务，释放数据库资源，性能比较好

利用全局锁实现读写分离

没有代码侵入，框架自动完成回滚和提交

缺点：

两阶段之间属于软状态，属于最终一致

框架的快照功能会影响性能，但比XA模式要好很多

**AT模式实现：**

1. AT模式中的快照生成，回滚等动作都是有框架自动完成，没有任何代码侵入，因此实现非常简单，只不过，AT模式需要一个表来记录全局锁，另一张表来记录数据快照undo-log
2. 修改application.yml文件，将事务模式修改为AT模式即可 默认模式就是AT

[返回首页](#首页)

**AT和XA的区别：**

AT模式与XA模式最大的区别是什么？

XA模式一阶段不提交事务，锁定资源，AT模式一阶段直接提交，不锁定资源

XA模式依赖数据库机制实现回滚，AT模式利用数据快照实现数据回滚

XA模式强一致，AT模式最终一致

[返回首页](#首页)

**Seata-TCC模式：**

TCC模式性能最强，与AT模式非常相似，每阶段都是独立事务，不同的是TCC通过人工编码来实现数据恢复

优点:

1. 一阶段直接提交，释放数据库资源，性能好
2. 相比AT模型，无需生成快照，无需使用全局锁，性能最强
3. 不依赖数据库事务，而是依赖补偿操作，可以用与非事务型数据库

缺点：

1. 有代码侵入，需要人为编写try，confirm和cancel接口，太麻烦
2. 软状态，事务是最终一致
3. 需要考虑confirm和cancel的失败情况，做好幂等处理

实现的单个方法 每个阶段的工作

1. Try：资源的检测和预留
2. Confirm：完成资源操作业务，要求try成功confirm一定要能成功
3. Cancle：预留资源释放，可以理解为try的反向操作

[返回首页](#首页)

TCC模式的问题

事务空回滚，当某个分支事务的try阶段阻塞时，可能导致全局事务超时而触发二阶段的cancel操作，在未执行try的操作时先执行了cancel操作，这时cancel不能做回滚，就是空回滚

业务悬挂，对与已经空回滚的业务，之前被阻塞的try操作恢复，继续执行try，就永远不可能confirm或cancel，事务一直处于中间状态，这就是业务悬挂，执行try操作时，应当判断cancel是否已经执行过了，如果已经执行，应当阻止空回滚后的try操作，避免悬挂

解决：

**解决空回滚和业务悬挂问题，必须要记录当前事务状态，是在try还是cancel**

[返回首页](#首页)

四种模式对比：



[返回首页](#首页)

Gateway服务网关：

在SpringCloud中的网关的实现包括两种gateway和zuul

Gateway是基于spring中提供的webflux属于响应式编程的实现

Zuul：基于servlet的实现，属于阻塞式编程，性能没有gateway好

SpringCloudGateway是springcloud的一个全新项目，该项目是基于响应式编程和事件流技术开发的网关，它意在为微服务架构提供一种简单有效的统一的API路由管理方式

Gateway网关是我们服务的守门神，所有微服务的统一入口

核心功能：

1. 请求路由，路由转发
2. 权限控制，权限效验
3. 限流

权限控制：网关作为微服务入口，需要效验用户是否有请求资格，如果没有则进行拦截

路由和负载均衡：一切请求都必须先经过gateway，但网关不处理业务，而是根据某种规则，把请求转发到某个微服务，这个过程就叫做路由，当然路由的目标服务有多个时，还需要做负载均衡

限流：当请求流量过高时，在网关中按照下限的微服务能够接受的速度来放行请求，避免服务压力过大

路由配置：

1. 路由id：路由的唯一标示
2. 路由目标：路由的目标地址，http代表固定地址，ib代表根据服务名负载均衡
3. 路由断言：判断路由的规则
4. 路由过滤器：对请求或响应做处理

[返回首页](#首页)

Filter过滤器：

请求进入网关会碰到三类过滤器：当前过滤器，defaultFilter，glabalFilter

请求路由后，会将当前路由过滤器和defaultFilter，glabalFilter合并到一个过滤器链中，排序后，排序后依次执行每个过滤器

排序规则是什么呢？

1. 每一个过滤器都必须指定一个int类型的order 值，order值越小，优先级越高，执行顺序越靠前
2. GlobalFilter通过实现ordered接口，或者添加order注解来指定order值，有我们自己指定
3. 路由过滤器和defaultFilter的order由spring指定，默认时按照顺序从1递增
4. 当过滤器的order值一样时，会按照defauleFilter》路由过滤器》globalfilter的顺序执行

[返回首页](#首页)

网关过滤器工厂：

过滤器的作用是什么？

1. 对路由的请求或响应做加工处理，比如添加请求头
2. 配置在路由下的过滤器只对当前路由的请求生效

Defaultfileters的作用是什么？

对所有路由都生效的过滤器

[返回首页](#首页)

全局过滤器：

全局过滤器的作用也是处理一切进入网关的请求和微服务响应，与gatewayfilter的作用一样，区别在于gatewayfilter通过配置定义，处理逻辑是固定的，而globalfilter的逻辑需要自己写代码实现，定义的方式是实现glabafilter接口

在filter中编写自定义逻辑，可以实现下列功能

1. 登录状态判断
2. 权限效验
3. 请求限流

[返回首页](#首页)

跨域问题：

跨域：域名不一致就是跨域，比如域名不同或者端口不同

跨域问题？浏览器禁止请求的发起者与服务端发生跨域ajax请求，请求被浏览器拦截的问题

解决跨域问题：

Globalcors：全局的跨域处理。在gateway服务的application.yml文件中，添加配置：



[返回首页](#首页)

消息中间件优点：

系统解耦：解耦消息生产者和消费者之间的关系

异步调用：用户调用接口时，由于接口之间调用导致用时时间比较久，用户体验不好，调用接口后将消息放入到MQ后就返回，用户体验好，最终一致性由 MQ来保证

流量削峰：减少高峰时期对服务器压力，先把请求放到MQ中，系统根据实际能处理的并发量来消费

[返回首页](#首页)

Kafka如何保证消息可靠性：

消费端会不会弄丢数据？不会

唯一可能导致消费者弄丢数据的情况就是说，你拉取了这个消息，然后消费者那边自动提交了offset,让kafka以为你已经消费好了这个消息，但其实你才刚准备处理这个消息，你还没有处理，你自己就挂了，此时这条消息就丢了

都知道kafka会自动提交offset，那么只要关闭自动提交offset，在处理完之后自己手动提交offset，就可以保证数据不会丢，但是此时确实还是可能会有重复消费，比如你刚处理完，还没提交offset，结果自己挂了，此时肯定会重复消费一次，出现了消息重复消费的问题，这个时候可以通过一些其他方式保证消息幂等就可以了

**消息消费时要求消息体中必须要由一个全局唯一ID，作为去重和幂等的依据，避免同一条消息被重复消费**

另外消费者会在metaData中自动记录消息的偏移+1，因此kafka可以确保消息不被重复消费，kafkabroker只用保证一条消息只能被一个消费者消费就行了

Kafka会为每一个consumer group保留一些metadata消息，当前消费的消息的position，也即offset，这个offset由consumer控制，正常情况下consumer会在消费完一条消息后线性增加这个offset，当然，consumer也可将offset设成一个较小的值，重新消费一些消息，因为offset由consumer控制，所有kafka broker是无状态的，它不需要标记那些消息被那些consumer过，不需要通过broker去保证同一个consumer group只有consumer能消费某一条消息，因此也就不需要锁机制，这也为kafka的高吞吐提供了有力保障

手动提交方案：同步提交，异步提交，同步异步结合

[返回首页](#首页)

Kafka的优点和缺点：

优点：

1. 吞吐量十万级
2. 可用性非常高（一个数据多个副本，少数机器宕机，不会丢失数据，不会导致不可用）
3. 耐久性：由于kafka支持消息复制，因此消息永远不会丢失，这个耐久性好的原因之一
4. 可扩展性：kafka可以扩展，而不需要通过添加额外的节点而在运行中造成任何停机

缺点：

Topic增多会导致吞吐量大幅下降（几百个topic），如果要支持大规模topic，需要更多的机器资源

依赖zooleeper进行元数据管理（额外的复杂性）

[返回首页](#首页)

如何保证消息幂等:

1. 乐观锁机制
2. 基于redis的原子操作
3. 增加消息状态表，通俗来说就是一个账本，用来记录消息的处理状态，每次处理消息之前，都去状态表中查询一次，如果已经有相同的消息存在，那么不处理，可以防止重复发送

[返回首页](#首页)

为什么要使用消息队列？为什么要使用kafka？

1. 缓冲和削峰：上游数据时有突发流量，下游可能扛不住，或者下游没有足够多的机器来保证冗余，kafka在中间可以起到一个缓冲的作用，把消息暂存在kafka中，下游服务就可以按照自己的节奏进行慢慢处理
2. 解耦和扩展性：项目开始的时候，并不能确定具体需求，消息队列可以作为一个接口层，解耦重要的业务流程。只需要遵守约定，针对数据编程即可获取扩展能力
3. 冗余：可以采用一对多的方式，一个生产者发布消息，可以被多个订阅topic的服务消费到，供多个毫无关联的业务使用
4. 健壮性：消息队列可以堆积请求，所以消费端业务即使短时间死掉，也不会影响主要业务的正常进行。
5. 异步通信：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他们

[返回首页](#首页)

RabbitMQ中的几种模型：

1. BasicQueue简单队列模型，消费者和生产者监听同一个队列就可以直接进行发送接受消息
2. WorkQueue也被称为任务模型，简单来说就是让多个消费者绑定到一个队列，共同消费队列中的消息。默认是将消息平均分配给每个消费者，我们可以进行设置。多个消费者绑定到一个队列，同一条消息只会被一个消费者处理，处理完了才能处理下一个，通过设置prefetch来控制消费者预取的消息数量，提高队列消息的处理速度避免消息堆积

第二大类就是发布订阅模式，在订阅模式中，多了一个exchange交换机，而且过程略有变化：pulisher生产者，也就是要发送消息的程序，但是不在发送到队列中，而是发送给交换机。Exchange交换机，一方面接受生产者发送的消息，另一方面，知道如何处理消息，例如递交给某个特别队列，递交给所有队列，或是将消息丢弃，到底如何操作取决于exchange的类型，exchange有三种类型

1. Fanou：广播，将消息交给所有绑定到交换机的队列，可以有多个队列，相当于是广播消息，只要在一条线上，都能接受到消息。 生产者发送的消息只能发送到交换机，消息有交换机决定发给那个队列，生产者是无法决定的，
2. Direct：定向路由，把消息交给符合指定routing key的队列，类似对暗号，暗号对上了才能进入队列接受到消息了，队列与交换机绑定，就不能是任意绑定了，而是要指定一个RoutingKey，消息的发送方在exchange发送消息时，也必须指定消息routingkey，exchange不在把消息交给每一个绑定的队列，而是根据消息的routingkey进行判断，只有当队列routingkey与消息的routingkey完全一致，才会接受到消息
3. Topic：主题通配符模式，把消息交给符合routing pattern路由模式的队列和direct相比都是可以根据routingkey把消息路由到不同的队列，只不过topic类型让exchange可以让队列在绑定routingkey的时候使用通配符。#匹配一个或多个词 \*匹配一个词

Exchange交换机只负责转发消息，不具备存储消息的能力，因此如果没有任何队列与exchange绑定或者没有符合路由规则的队列，那么消息会丢失

[返回首页](#首页)

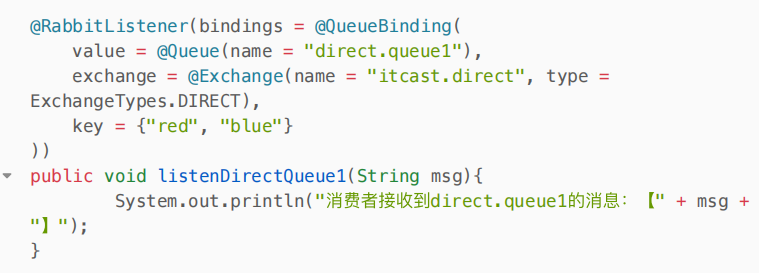
基于注解声明队列和交换机：

@RabbitListener注解声明队列和交换机

@Queue指定队列

@Exchange指定交换机

基于bean的方式声明队列和交换机比较麻烦，Spring还提供了基于注解方式来声明



[返回首页](#首页)

为什么使用消息队列（优点）

1. 系统解耦：解耦消息生产者和消费者之间的关系

异步调用：用户调用接口时，由于接口之间调⽤导致⽤时时间⽐较久，⽤户体验不好。调⽤接⼝后将消息放⼊到MQ后就返回，⽤户体验好，最终⼀致性由MQ来保证

1. 流量削峰：减少⾼峰时期对服务器压⼒，先把请求放到MQ中，系统根据实际能处理的并发量 来消费请求

会带来什么问题（缺点）？

1.系统可⽤性降低。如果MQ挂了，整个系统就不能服务了

2.系统复杂性提⾼。消息丢失、消息重复消费、消息重复发送、消息顺序错乱等问题

3.⼀致性问题。将消息放到MQ后就返回给⽤户成功的信息，但是其他系统消费消息时，若某个

系统失败了，导致数据不⼀致

[返回首页](#首页)

RabbitMQ的持久化：

RabbitMQ的持久化分为3个部分:交换器的持久化、队列的持久化和消息的持久化。但是如果将所

有消息持久化，将会严重影响rabbitmq的性能。

即使将交换器、队列和消息都设置了持久化，也不能保证消息100%不丢失。

1. 从消费者端，防⽌消费者收到消息还没来得及处理就宕机的情况，需要将autoAck设置为

false

2. 从⽣产者断，防⽌发送到rabbitmq后还没来的及落盘rabbitmq就宕机的情况，可以在⽣产者

端引⼊事务机制或者发送⽅确认机制来保证消息⼰经正确地发送并存储RabbitMQ 中（前提

还要保证在调⽤ channel.basicPublish ⽅法的时候交换器能够将消息正确路由到相应的队列

之中）或者引⼊镜像队列来保证⾼可⽤

[返回首页](#首页)

RabbitMQ如何保证消息的可靠性？

● ⽣产者到队列：开去生产者确认机制，事务机制和Confirm机制，注意：事务机制和Confirm机制是互斥的，两者不 能共存，会导致RabbitMQ报错。

● 队列⾃身：持久化、集群、普通模式、镜像模式。

● 队列到消费者：basicAck机制、死信队列、消息补偿机制

RabbitMq如何避免重复重复消费

重复投递的原因：等待超时后，需要重试。

避免重复投递：消息⽣产时，⽣产者发送的消息携带⼀个Message ID（全局唯⼀ID），作为去重

和幂等的依据，避免重复的消息进⼊队列

重复消费的原因：消费者接收消息后，在确认之前断开了连接或者取消订阅，消息会被重新分发

给下⼀个订阅的消费者。

避免重复消费：消息消费时，要求消息体中必须要有⼀个全局唯⼀ID，作为去重和幂等的依据，

避免同⼀条消息被重复消费

[返回首页](#首页)

如果实现发送一个延迟消息？

● 给消息的⽬标队列指定死信交换机

● 将消费者监听的队列绑定到死信交换机

● 发送消息时给消息设置超时时间为20秒

[返回首页](#首页)

惰性队列解决消息堆积问题：

当⽣产者发送消息的速度超过了消费者处理消息的速度，就会导致队列中的消息堆积，直到队列存储消 息达到上限。之后发送的消息就会成为死信，可能会被丢弃，这就是消息堆积问题

从RabbitMQ的3.6.0版本开始，就增加了Lazy Queues的概念，也就是惰性队列。惰性队列的特征如下：

● 接收到消息后直接存⼊磁盘⽽⾮内存

● 消费者要消费消息时才会从磁盘中读取并加载到内存

● ⽀持数百万条的消息存储

惰性队列的优点有哪些？

● 基于磁盘存储，消息上限⾼

● 没有间歇性的page-out，性能⽐较稳定

惰性队列的缺点有哪些？

● 基于磁盘存储，消息时效性会降低

● 性能受限于磁盘的IO

其他：

● 队列上绑定多个消费者，提⾼消费速度

● 使⽤惰性队列，可以再mq中保存更多消息

[返回首页](#首页)

Mysql6种会发生索引失效的情况：

1.当我们使⽤左或者左右模糊匹配的时候，也就是 like %xx 或者 like %xx% 这两种⽅式都会造成索引失效；

2.当我们在查询条件中对索引列使⽤函数，就会导致索引失效。

3.当我们在查询条件中对索引列进⾏表达式计算，也是⽆法⾛索引的。

4.MySQL 在遇到字符串和数字⽐较的时候，会⾃动把字符串转为数字，然后再进⾏⽐较。如果字符串是 索引列，⽽条件语句中的输⼊参数是数字的话，那么索引列会发⽣隐式类型转换，由于隐式类型转换是 通过 CAST 函数实现的，等同于对索引列使⽤了函数，所以就会导致索引失效。

5.联合索引要能正确使⽤需要遵循最左匹配原则，也就是按照最左优先的⽅式进⾏索引的匹配，否则就会导致索引失效。

6.在 WHERE ⼦句中，如果在 OR 前的条件列是索引列，⽽在 OR 后的条件列不是索引列，那么索引会失效。

不在索引上做任何操作

[返回首页](#首页)

Sql的select语句完整执行顺序：

from->where->group by->having->计算表达式->order by->select输出

[返回首页](#首页)

Where:过滤：分组之前的过滤：对数据源进⾏过滤：

Wher后⾯不可以使⽤聚合函数

Having：过滤：分组之后的过滤

Having后⾯可以使⽤聚合函数

在查询过程中执⾏顺序：from>where>group（含聚合）>having>order>select。

[返回首页](#首页)

索引类型和约束：

索引类型：

主键索引、唯⼀索引、普通索引、组合索引、全⽂索引

五⼤约束：

主键约束、唯⼀性约束、默认值约束、⾮空约束、外键约束

[返回首页](#首页)

Mysql优化

MYSQL优化主要分为以下四⼤⽅⾯：

设计：存储引擎，字段类型，范式与逆范式

功能：索引，缓存，分库分表。

架构：主从复制，读写分离，负载均衡。

合理SQL：测试，经验

1. 设计良好的数据库结构，允许部分数据冗余，尽量避免join查询，提⾼效率。
2. 选择合适的表字段数据类型和存储引擎，适当的添加索引。
3. mysql库主从读写分离。
4. 找规律分表，减少单表中的数据量提⾼查询速度。
5. 添加缓存机制，⽐如memcached，apc等。
6. 不经常改动的⻚⾯，⽣成静态⻚⾯。
7. 书写⾼效率的SQL

对查询进⾏优化，应尽量避免全表扫描，⾸先应考虑在 where 及 order by 涉及的列上建⽴索 引。 这个可能会出现⽂件内排序，⾮常影响性能

应尽量避免在 where ⼦句中对字段进⾏ null 值判断，否则将导致引擎放弃使⽤索引⽽进⾏全表 扫描

应尽量避免在 where ⼦句中使⽤!=或<>操作符，否则将引擎放弃使⽤索引⽽进⾏全表扫描。 这个可能造成间隙锁，锁住范围的数据

应尽量避免在 where ⼦句中使⽤ or 来连接条件，否则将导致引擎放弃使⽤索引⽽进⾏全表扫 描 这个还可能造成锁升级⼜⾏锁升级为表锁

in 和 not in 也要慎⽤，否则会导致全表扫描，

下⾯的查询也将导致全表扫描

应尽量避免在 where ⼦句中对字段进⾏表达式操作，这将导致引擎放弃使⽤索引⽽进⾏全表扫 描

应尽量避免在where⼦句中对字段进⾏函数操作，这将导致引擎放弃使⽤索引⽽进⾏全表扫描

不要在 where ⼦句中的“=”左边进⾏函数、算术运算或其他表达式运算，否则系统将可能⽆法 正确使⽤索引。

在使⽤索引字段作为条件时，如果该索引是复合索引，那么必须使⽤到该索引中的第⼀个字段 作为条件时才能保证系统使⽤该索引，否则该索引将不会被使⽤，并且应尽可能的让字段顺序与 索引顺序相⼀致。

不要写⼀些没有意义的查询，这类代码不会返回任何结果集，但是会消耗系统资源的， 很多时候⽤ exists 代替 in 是⼀个好的选择：

select num from a where num in(select num from b)

并不是所有索引对查询都有效，SQL是根据表中数据来进⾏查询优化的，当索引列有⼤量数据

重复时，SQL查询可能不会去利⽤索引，如⼀表中有字段sex，male、female⼏乎各⼀半，那么 即使在sex上建了索引也对查询效率起不了作⽤。

[返回首页](#首页)

Mysql事务：

原⼦性A：通过undo log ⽇志实现 事务回滚就反操作

⼀致性C：指事务执⾏完毕后，数据库的完整性约束没有被破坏，事务追求的最终⽬标

隔离性I：写写操作 是通过锁去实现的 读写模式就是mvcc去实现的

持久性D：就是通过redo log去实现的

[返回首页](#首页)

Mysql数据库的基本设计步骤：

需求分析：分析⽤户的需求，包括数据、功能和性能需求

概念结构设计：主要采⽤E-R模型进⾏设计，包括画E-R图；

逻辑结构设计：通过将E-R图转换成表，实现从E-R模型到关系模型的转换；

物理结构设计：主要是为所设计的数据库选择合适的存储结构和存取路径；

数据库实施：包括编程、测试和试运⾏；

数据库的运⾏和维护：系统的运⾏与数据库的⽇常维护。,主要讨论其中的第3个阶段,即逻辑设计。

[返回首页](#首页)

Mysql引擎myisam和innoDB的区别：

myisam 只有表锁没有⾏锁 不⽀持事务

innoDB 有表锁和⾏锁 有事务隔离级别 可重复度



[返回首页](#首页)

主从复制：

*Binary log：主数据库的⼆进制⽇志。*

*Relay log：从服务器的中继⽇志。*

第⼀步：master在每个事务更新数据完成之前，将该操作记录串⾏地写⼊到binlog⽂件中。

第⼆步：salve开启⼀个I/O Thread，该线程在master打开⼀个普通连接，主要⼯作是binlog dump process。如果读取

的进度已经跟上了master，就进⼊睡眠状态并等待master产⽣新的事件。I/O线程最终的⽬的是将这些事件写⼊到中继

⽇志中。

第三步：SQL Thread会读取中继⽇志，并顺序执⾏该⽇志中的SQL事件，从⽽与主数据库中的数据保持⼀致。

[返回首页](#首页)

Mysql的mvcc是什么？

多版本并发控制，主要是为了提⾼数据库读写性能，让数据库在读写的时候不⽤去加锁。

mvcc主要是处理读请求的。这个读指的是快照读[select读]，⽽不是当前读[悲观锁]。

加锁的都是当前读在进⾏ instert 和 update 都是需要先将数据读出来的 在进⾏操作的。

快照读就是mvcc实现的 ⽬的就是在读写数据的时候不⽤去竞争锁。

读已提交RC 和 可重复读RR 他们的快照读 都是基于mvcc实现的

版本链 是 undo log和回滚指针将他们链接起来的

[返回首页](#首页)

代码层⾯如何实现mysql的读写分离

默认⼤家对MyBatis和Spring都有所了解，最简单的实现⽅式：

1.Spring注⼊数据源，包括Master-Slave；

2.⼿写MyBatis拦截器，判断⼀个SQL语句是读还是写；

3.选择对应的数据源进⾏执⾏

[返回首页](#首页)

B数等之间的关系

⼆叉搜索树：⼆叉树，每个结点只存储⼀个关键字，等于则命中，⼩于⾛左结点，⼤于

⾛右结点；

B（B-）树：多路搜索树，每个结点存储M/2到M个关键字，⾮叶⼦结点存储指向关键

字范围的⼦结点；

所有关键字在整颗树中出现，且只出现⼀次，⾮叶⼦结点可以命中；

B+树：在B-树基础上，为叶⼦结点增加链表指针，所有关键字都在叶⼦结点

中出现，⾮叶⼦结点作为叶⼦结点的索引；B+树总是到叶⼦结点才命中；

B\*树：在B+树基础上，为⾮叶⼦结点也增加链表指针，将结点的最低利⽤率

从1/2提⾼到2/3；

[返回首页](#首页)

多线程状态：

1. 新建，new thread
2. 就绪，start获得锁，或者睡眠时间到了
3. 阻塞，获取不到锁
4. 无线等待，wait
5. 计时等待
6. 终止，run方法执行结束，润抛出的异常未被铺货

如何设置线程数

提供一个公式：

最佳线程数⽬ = （（线程等待时间+线程CPU时间）/线程CPU时间 ）\* CPU数⽬

怎么使用？

使⽤步骤:

创建线程池 Executors.newCachedThreadPool()

提交任务到池,submit(Runnable)

创建固定上线的线程池 Executors.newFixedThreadPool(5);

好处: 60秒不⽤的线程会⾃动销毁

缺点:创建的线程数是int的最⼤值,可能耗光系统资源.导致服务器宕机

不设置线程数量上限就会导致资源耗尽。【阿⾥巴巴规范不让⽤，虽然简单，容易实现】

[返回首页](#首页)

多线程创建7大参数：

1. 核心线程数量
2. 最大线程数量
3. 空闲线程最大存活时间
4. 时间单位
5. 任务队列
6. 创建线程工厂
7. 任务的拒绝策略

任务的拒绝策略

RejectedExecutionHandler是jdk提供的⼀个任务拒绝策略接⼝，它下⾯存在4个⼦类。

何时触发拒绝策略??

任务数 > 池的最⼤容量 + 队列容量

[返回首页](#首页)

悲观锁和乐观锁：

相同点：在多线程情况下，都可以保证共享数据的安全性

不同点：

Synchronized总是从最坏的角度出发，认为每次获取数据的时候，别人都有可能修改，所以在每次操作共享数据之前都会上锁（悲观锁）

Cas是从乐观的角度出发，假设每次获取数据别人都不会修改，所有不会上锁，只不过在修改共享数据的时候，会检测一下，别人有没有修改过这个数据，如果别人修改过，那么我再次获取现在最新的值，如果别人没有修改过，那么我现在就直接修改共享数据的值（乐观锁）

[返回首页](#首页)

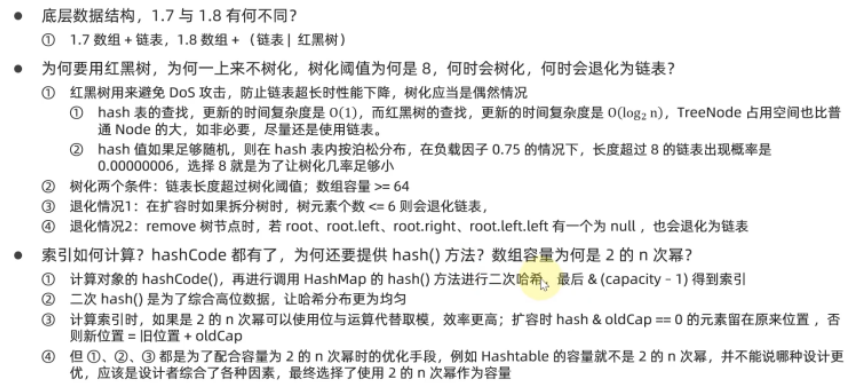
单列集合 collection

双列集合 map



[返回首页](#首页)

HashMap1.7和1.8的区别：

hashmap扩容 ：

hashMap 底层是1.7之前数组+链表

hashMap 底层是1.8之后数组+链表+红⿊树

红黑数触发条件：

链表长度大于8

数组长度一句扩容到64，而且要大于64链表才会优先树化

数组长度在64之前都会优先扩容，导致链表变短，减少链表长度

为什么必须要64之后才能扩容 ？

因为在64之前 树化效率不怎么明显 也就是查询效率也不是很快

HashMap

多线程下会有什么问题？

扩容死链1.7 数据错乱1.7+1.8

Key能否为null 作为key的对象有什么要求？

Hashmap的key可以为null，但map的其他实现不行

作为key的对象，必须实现hashcode和equels，并且key的内容不能修改

Hashcode是如何设计的 为啥每次都乘31？

31带进去有很好的散列型，每个hashcode足够独特

31\*h可以被优化 31=2^5-1=32-1

41的散列特性也可以，但是41不能计算优化

[返回首页](#首页)

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。

[返回首页](#首页)

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。

[返回首页](#首页)

垃圾回收器的原理是什么？有什么办法⼿动进⾏垃圾回收？

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。

164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。

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。

[返回首页](#首页)

类加载的执行过程：

类装载分为以下 5 个步骤：

加载：根据查找路径找到相应的 class ⽂件然后导⼊；

验证：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。

初始化：对静态变量和静态代码块执⾏初始化⼯作。

[返回首页](#首页)

JVM调优工具：

JDK ⾃带了很多监控⼯具，都位于 JDK 的 bin ⽬录下，其中最常⽤的是 jconsole 和 jvisualvm

这 两款视图监控⼯具。

● jconsole：⽤于对 JVM 中的内存、线程和类等进⾏监控；

● jvisualvm：JDK ⾃带的全能分析⼯具，可以分析：内存快照、线程快照、程序死锁、监控内

存的变化、gc 变化等。

[返回首页](#首页)

什么是程序计数器？

程序计数器是⼀块较⼩的内存空间，它可以看作是：保存当前线程所正在执⾏的字节码指令 的地址 (⾏号)

由于Java虚拟机的多线程是通过线程轮流切换并分配处理器执⾏时间的⽅式来实现的，⼀个 处理器 都只会执⾏⼀条线程中的指令。因此，为了线程切换后能恢复到正确的执⾏位置，每 条线程都有⼀ 个独⽴的程序计数器，各个线程之间计数器互不影响，独⽴存储。称之为“线程 私有”的内存。

[返回首页](#首页)

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。

总结就是： 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。

[返回首页](#首页)

延迟发布审核：

1.请求过来了，先保存数据到数据库，此时这条数据状态为待审核状态，是不能展示到列表的。

2.然后添加了延迟发布任务，到延迟队列，⽤定时任务去拉去队列中的任务进⾏消费，

3.再调⽤异步审核⽅法对数据中的⽂字和图⽚ocr识别进⾏提取⽂本，

4.可以调⽤⾃定义的敏感词过滤，然后再调⽤阿⾥云的内容审核，对图⽚和⽂本进⾏审核

5.审核失败就直接返回，再将评价的状态改为审核失败，然后审核成功就直接修改状态发布展示。

亮点：

如果问延迟队列怎么实现了的话就说本来是打算⽤rabbitMq的死信交换机和死信队列实现的

然后考虑到 rabbitMq发送消息的消息被重复消费问题和消息的丢失问题还要在做处理。就去百度学习了

⽤Redis来实现的延迟消息队列 主要是⽤ List+Zset 进⾏实现的。

（zset里面有个score分值排序就直接将用户传过来的时间戳当score排序实现）

[返回首页](#首页)

优势：

其实，Redis⽤来进⾏实现延时队列是具有这些优势的：

Redis zset⽀持⾼性能的 score 排序。

Redis是在内存上进⾏操作的，速度⾮常快。

Redis可以搭建集群，当消息很多时候，我们可以⽤集群来提⾼消息处理的速度，提⾼可⽤性。

Redis具有持久化机制，当出现故障的时候，可以通过AOF和RDB⽅式来对数据进⾏恢复，保证了数据的可靠性

然后在讲讲并发线程下有可能需要加个分布式锁 解决消息被重复消费问题

分布式锁 --三种解决⽅案

1.数据库乐观锁；【这个⼀般是mysql的】

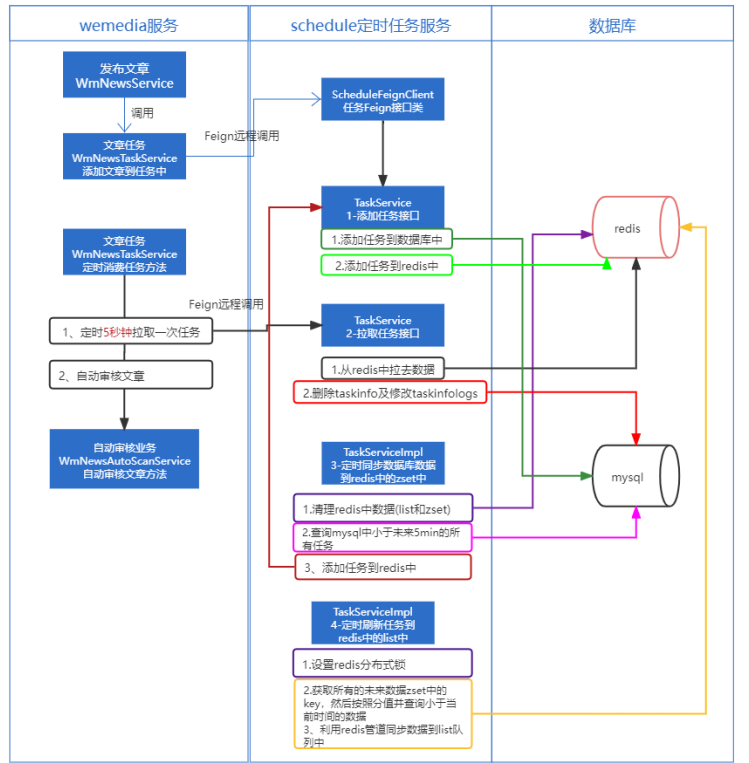
2.基于Redis的分布式锁；【我们⽤的就是这个】

redis⾃带的 setNX命令实现 加锁，通过LUA脚本实现解锁，特点是可以设置锁存时间，时间到了自动解锁

3.基于ZooKeeper的分布式锁。【就说不了解】

如果问zset的底层数据结构 ？

只知道zset里面是一个跳跃表的结构，详细的倒是没怎么了解



[返回首页](#首页)

在Spring中事务如何自定义回滚？

context⾥⾯可以拿到事务 上下⽂中有

然后 ⾃定义过滤器 抓取异常 进⾏事务回滚

[返回首页](#首页)

Elasticsearch 是基于 Lucene 的 Restful 的分布式实时全文搜索引擎，

每个字段都被索引并可被搜索，可以快速存储、搜索、分析海量的数据。

基本概念：

（1）index 索引：索引类似于mysql 中的数据库，Elasticesearch 中的索引是存在数据的地方，包含了一堆有相似结构的文档数据。

（2）type 类型：类型是用来定义数据结构，可以认为是 mysql 中的一张表，type 是 index 中的一个逻辑数据分类

（3）document 文档：类似于 MySQL 中的一行，不同之处在于 ES 中的每个文档可以有不同的字段，但是对于通用字段应该具有相同的数据类型，文档是es中的最小数据单元，可以认为一个文档就是一条记录。

（4）Field 字段：Field是Elasticsearch的最小单位，一个document里面有多个field

（5）shard 分片：单台机器无法存储大量数据，es可以将一个索引中的数据切分为多个shard，分布在多台服务器上存储。有了shard就可以横向扩展，存储更多数据，让搜索和分析等操作分布到多台服务器上去执行，提升吞吐量和性能。

（6）replica 副本：任何一个服务器随时可能故障或宕机，此时 shard 可能会丢失，因此可以为每个 shard 创建多个 replica 副本。replica可以在shard故障时提供备用服务，保证数据不丢失，多个replica还可以提升搜索操作的吞吐量和性能。primary shard（建立索引时一次设置，不能修改，默认5个），replica shard（随时修改数量，默认1个），默认每个索引10个 shard，5个primary shard，5个replica shard，最小的高可用配置，是2台服务器。

[返回首页](#首页)

倒排索引和正牌索引：  
'- 正向索引是最传统的，根据id索引的方式。

但根据词条查询时，必须先逐条获取每个文档，

然后判断文档中是否包含所需要的词条，是根据文档找词条的过程。

- 而倒排索引则相反，是先找到用户要搜索的词条，

根据词条得到保护词条的文档的id，然后根据id获取文档。是根据词条找文档的过程。

\*\*正向索引\*\*：

- 优点：

- 可以给多个字段创建索引

- 根据索引字段搜索、排序速度非常快

- 缺点：

- 根据非索引字段，或者索引字段中的部分词条查找时，只能全表扫描。

\*\*倒排索引\*\*：

- 优点：

- 根据词条搜索、模糊搜索时，速度非常快

- 缺点：

- 只能给词条创建索引，而不是字段

- 无法根据字段做排序

[返回首页](#首页)

Text和keyword区别：  
两个的区别主要分词的区别：keyword 类型是不会分词的，

直接根据字符串内容建立倒排索引，keyword类型的字段只能通过精确值搜索到；

Text 类型在存入 Elasticsearch 的时候，会先分词，然后根据分词后的内容建立倒排索引

[返回首页](#首页)

Query和filter的区别：

\*\*（1）query：\*\*查询操作不仅仅会进行查询，还会计算分值，用于确定相关度；

\*\*（2）filter：\*\*查询操作仅判断是否满足查询条件，不会计算任何分值，

也不会关心返回的排序问题，同时，filter 查询的结果可以被缓存，提高性能。

[返回首页](#首页)

es写数据的过程(ES的写入流程：)

（1）客户端选择一个 node 发送请求过去，这个 node 就是 coordinating node （协调节点）

（2）coordinating node 对 document 进行路由，将请求转发给对应的 node（有 primary shard）

（3）实际的 node 上的 primary shard 处理请求，然后将数据同步到 replica node

（4）coordinating node 等到 primary node 和所有 replica node 都执行成功之后，就返回响应结果给客户端。

[返回首页](#首页)

8 ES在高并发下如何保证读写一致性？

（1）对于更新操作：可以通过版本号使用乐观并发控制，以确保新版本不会被旧版本覆盖

（2）对于写操作，一致性级别支持 quorum/one/all，默认为 quorum，即只有当大多数分片可用时才允许写操作。

（3）对于读操作，可以设置 replication 为 sync(默认)，这使得操作在主分片和副本分片都完成后才会返回

[返回首页](#首页)

Elasticsearch的分布式原理：

Elasticsearch 会对存储的数据进行切分，将数据划分到不同的分片上，

同时每一个分片会保存多个副本，主要是为了保证分布式环境的高可用。

在 Elasticsearch 中，节点是对等的，节点间会选取集群的 Master，

由 Master 会负责集群状态信息的改变，并同步给其他节点。

[返回首页](#首页)

ES的深度分页与滚动搜索：

1）深度分页：\*\*

深度分页其实就是搜索的深浅度，(比如第1页，第2页，第10页，第20页，是比较浅的；第10000页，第20000页就是很深了。)

搜索得太深，就会造成性能问题，会耗费内存和占用cpu。

而且es为了性能，他不支持超过一万条数据以上的分页查询。

（2）滚动搜索：\*\*

一次性查询1万+数据，往往会造成性能影响，因为数据量太多了。

这个时候可以使用滚动搜索，也就是 scroll。

滚动搜索可以先查询出一些数据，然后再紧接着依次往下查询。

在第一次查询的时候会有一个滚动id，相当于一个锚标记 ，随后再次滚动搜索会需要上一次搜索滚动id

每次搜索都是基于一个历史的数据快照，查询数据的期间，如果有数据变更，那么和搜索是没有关系的。

[返回首页](#首页)

Spring事务隔离级别：

其实就是事务隔离级别，

1. Default使用数据库默认的事务隔离级别
2. 读未提交，脏读不可重复读，虚读都有可能发生
3. 以提交读，避免脏读，但是不可重读和虚读有可能发生
4. 可重复读，避免脏读和不可重复读，但是虚读有可能发生
5. 串行化，所有事务请求串行执行

[返回首页](#首页)

超大分页怎么处理 ?

MYSQL 不是跳过offset行, 而是取offset+N行, 然后放弃前offset行 , 返回N行, 所以当offset比较法的情况下分页效率很低

正确的处理方法是 : 先快速定位需要获取的id再关联查询获取数据

[返回首页](#首页)

mysql索引的优势和劣势？

优势：

可以提高数据检索的效率，降低数据库的IO成本，类似于书的目录。

通过索引列对数据进行行排序，降低数据排序的成本，降低了了CPU的消耗

劣势：

索引会占据磁盘空间

索引虽然会提高查询效率，但是会降低更新表的效率。比如每次对表进行增删改操作，MySQL不仅要保存数据，还要保存或者更新对应的索引文件

[返回首页](#首页)

如果一条SQL语句执行很慢 , 如何找到慢的原因 ?

1 开启慢查询日志或者使用监控工具监控慢查询 , 例如 : skywalking

2 为常用的查询字段或者条件建立索引

3 使用Explain关键字查看SQL执行计划 , 看索引是否命中 , 以及命中的级别

4 根据情况修改SQL语句

[返回首页](#首页)

