## ## 初级研发可能遇到的问题

### 1. 对Spring、SpringCloud、Springboot的了解

Spring框架的核心特性是可以用于开发任何 Java 应用程序，但在 Java EE 平台上构建 web 应用程序是需要扩展的。 Spring 框架的目标是使 J2EE 开发变得更容易

Spring Boot实现了自动配置，降低了项目搭建的复杂度。因为Spring框架需要进行大量的配置，Spring Boot可以自动配置，让项目设置变得很容易。

Spring Boot本身并不提供Spring框架的核心特性以及扩展功能，只是用于快速、敏捷地开发新一代基于Spring框架的应用程序。也就是说，它并不是用来替代Spring的解决方案，而是和Spring框架紧密结合用于提升Spring开发者体验的工具。

Spring cloud是一个基于Spring Boot实现的服务治理工具包，在微服务架构中用于管理和协调服务的。Spring Cloud为分布式系统快速构建行业统一特征模式的应用提供了一个行之有效的解决工具箱，诸如配置管理、服务发现、熔断器、智能路由、微服务代理、控制总线、一次性令牌、全局锁、领导选举、分布式会话以及集群状态等，通过各分布式微服务组件的实现以及协调使用，开发者能够快速构建出运行良好的服务。

### 2. Java的异常有哪几种？具体的异常都是什么意思

系统错误（Error）：由java虚拟机抛出，这种错误很少发生，如果 发生，除了通知用户以及尽量稳妥地终止程序外，几乎什么也不能做；如OutOfMemoryError StackOverflowError

编译时异常： Exception及其除RuntimeException外的子类，在编译时期抛出的异常，会在编译期间检查程序是否可能会出现问题如 IOException

运行时异常：运行时异常是用RuntimeException类表示的，描述的是程序设计错误，如错误的类型转换，访问一个越界数组等

RuntimeException、Error 以及它们的子类都称为免检异（unchecked exception )。所有其他异常（编译时异常）都称为必检异常（checked exception),

补充：

Java异常架构

1.Throwable

Throwable 是 Java 语言中所有错误与异常的超类。

Throwable 包含两个子类：Error（错误）和 Exception（异常），它们通常用于指示发生了异常情况。

Throwable 包含了其线程创建时线程执行堆栈的快照，它提供了 printStackTrace() 等接口用于获

取堆栈跟踪数据等信息。

2. Error（错误）

定义：Error 类及其子类。程序中无法处理的错误，表示运行应用程序中出现了严重的错误。

特点：此类错误一般表示代码运行时 JVM 出现问题。通常有 Virtual MachineError（虚拟机运行错误）、NoClassDefFoundError（类定义错误）等。比如 OutOfMemoryError：内存不足错误；StackOverflowError：栈溢出错误。此类错误发生时，JVM 将终止线程。

这些错误是不受检异常，非代码性错误。因此，当此类错误发生时，应用程序不应该去处理此类错误。按照Java惯例，我们是不应该实现任何新的Error子类的！

3. Exception（异常）

程序本身可以捕获并且可以处理的异常。Exception 这种异常又分为两类：运行时异常和编译时异常。

运行时异常

定义：RuntimeException 类及其子类，表示 JVM 在运行期间可能出现的异常。

特点：Java 编译器不会检查它。也就是说，当程序中可能出现这类异常时，倘若既"没有通过

throws声明抛出它"，也"没有用try-catch语句捕获它"，还是会编译通过。比如

NullPointerException空指针异常、ArrayIndexOutBoundException数组下标越界异常、

ClassCastException类型转换异常、ArithmeticExecption算术异常。此类异常属于不受检异常，

一般是由程序逻辑错误引起的，在程序中可以选择捕获处理，也可以不处理。虽然 Java 编译器不

会检查运行时异常，但是我们也可以通过 throws 进行声明抛出，也可以通过 try-catch 对它进行

捕获处理。如果产生运行时异常，则需要通过修改代码来进行避免。例如，若会发生除数为零的情

况，则需要通过代码避免该情况的发生！

RuntimeException 异常会由 Java 虚拟机自动抛出并自动捕获（就算我们没写异常捕获语句运行

时也会抛出错误！！），此类异常的出现绝大数情况是代码本身有问题应该从逻辑上去解决并改进

代码。

编译时异常

定义: Exception 中除 RuntimeException 及其子类之外的异常。

特点: Java 编译器会检查它。如果程序中出现此类异常，比如 ClassNotFoundException（没有找

到指定的类异常），IOException（IO流异常），要么通过throws进行声明抛出，要么通过trycatch进行捕获处理，否则不能通过编译。在程序中，通常不会自定义该类异常，而是直接使用系

统提供的异常类。该异常我们必须手动在代码里添加捕获语句来处理该异常。

### 3. 日常编程用哪种Java代码规范？对代码规范的理解

主要有阿里巴巴代码规范和谷歌代码规范，

统一团队代码风格：通过遵守统一的代码规范，可以使团队代码风格保持一致，便于团队协作和代码维护。

提高代码可读性：规范的代码格式和命名规范使得代码更易读，降低了代码阅读和理解的难度。

减少 bug 和维护成本：遵循代码规范可以降低代码出错和 bug 的概率，同时减少代码维护的成本。

方便检测和自动化工具支持：许多代码静态分析工具都支持对代码规范的检查，有助于提高代码质量。

### 4. 继承、实现的区别。父类有抽象的话，子类能不能只实现部分

不同的关键字，即实现接口（implements），继承（extends）；

一个类只能继承一个抽象类；而一个类却可以实现多个接口，而一个接口也可以继承多个接口

继承是满足is - a的关系，而接口只需满足has - a的关系。接口也更加方便扩展功能，而继承更可能带来一定的冗余，比如继承后拥有一些父类有但自己又不需要的功能

子类可以只实现部分

### 5. 序列化里的常量是做什么用的？

Java的序列化机制是通过在运行时判断类的serialVersionUID来验证版本一致性的。在进行反序列化时，JVM会把传来的字节流中的serialVersionUID与本地相应实体（类）

的serialVersionUID进行比较，如果相同就认为是一致的，可以进行反序列化，否则就会出现序列化版本不一致的异常

### 6. jason的方式做序列化的话，会不会包含class信息？反序列化的话，怎么知道是哪个类？

一般不需要包含class信息，在做反序列化时，在反序列方法中指定需要反序列化的对象类型，由json组件根据对象类型生成目标实例

### 7. Spring 的cache 注解用过没？spring 的cache 注解存的值是什么样子的？

存储的是json格式的字符串

### 8. My batis 了解吗？展开叙述下。

MyBatis是一个优秀的持久层框架，它对jdbc的操作数据库的过程进行封装，使开发者只需要关注SQL本身，而不需要花费精力去处理例如注册驱动、创建connection、创建statement、手动设置参数、结果集检索等jdbc繁杂的过程代码。

Mybatis通过xml或注解的方式将要执行的各种statement（statement、preparedStatemnt）配置起来，并通过java对象和statement中的sql进行映射生成最终执行的sql语句，最后由mybatis框架执行sql并将结果映射成java对象并返回。

总之，Mybatis对JDBC访问数据库的过程进行了封装，简化了JDBC代码，解决JDBC将结果集封装为Java对象的麻烦

### 9. AOP用过吗？主要用来做什么？

AOP 采取横向抽取机制（动态代理），取代了传统纵向继承机制的重复性代码，其应用主要体现在事务处理、日志管理、权限控制、异常处理等方面。

主要作用是分离功能性需求和非功能性需求，使开发人员可以集中处理某一个关注点或者横切逻辑，减少对业务代码的侵入，增强代码的可读性和可维护性。简单的说，AOP 的作用就是保证开发者在不修改源代码的前提下，为系统中的业务组件添加某种通用功能。

### 10. 用户登录做过没？用户密码怎么存储到数据库？

用户登录根据实际情况回答

使用PBKDF2算法存储，该算法原理大致相当于在HASH算法基础上增加随机盐，并进行多次HASH运算，随机盐使得彩虹表的建表难度大幅增加，而多次HASH也使得建表和破解的难度都大幅增加

### 11. 单例如何实现？

使用静态内部类

**public** **class** **Singleton** **{**

**private** **Singleton(){}**

**private** **static** **class** **Instance** **{**

**private** **static** **final** Singleton instance **=** **new** Singleton**();**

**}**

**public** **static** Singleton **getInstance(){**

**return** Instance**.**instance**;**

**}**

**}**

枚举类型实现

public enum Singleton {

INSTANCE;

}

## ## 中级研发可能遇到的问题

### 对Spring、SpringCloud、Springboot的了解？SpringCloud的原理，主要组件？

服务注册与发现：Eureka、Consul、ZooKeeper 等，用于管理服务实例的注册和发现。

负载均衡：Ribbon、LoadBalancer 等，用于实现服务之间的负载均衡。

断路器：Hystrix、Resilience4j 等，用于处理服务之间的故障和超时。

网关：Zuul、Gateway 等，用于实现统一的访问入口和路由转发。

配置中心：Config、Spring Cloud Config 等，用于统一管理应用的配置信息。

消息总线：Spring Cloud Bus、Kafka 等，用于消息广播和配置更新。

链路追踪：Sleuth、Zipkin 等，用于实现分布式系统的请求链路追踪。

### 服务与服务之间的交互是怎样用的？

HTTP（Restful API）：一种常见的方式是使用HTTP协议进行通信，通过Restful API来实现服务之间的交互。每个服务暴露自己的API接口，其他服务可以通过发送HTTP请求来调用这些接口，实现数据的交换和通信。

RPC（远程过程调用）：RPC是一种远程调用的方式，允许一个程序调用另一个地址空间（通常是另一个机器上）的过程。常见的RPC框架有gRPC、Apache Thrift、Dubbo等，通过定义接口和调用规范，实现不同服务之间的调用和通信。

消息队列（MQ）：通过消息队列可以实现服务之间的异步通信，一个服务把消息发送到消息队列中，其他服务从消息队列中取出消息进行处理。常见的消息队列系统有RabbitMQ、Kafka、ActiveMQ等，可以实现解耦服务间的通信

### Gateway的具体作用是什么？可拓展的作用是什么？怎么透传客户的信息？

网关是整个微服务API请求的入口，负责拦截所有请求，分发到服务上去。可以实现日志拦截、权限控制、解决跨域问题、限流、熔断、负载均衡，隐藏服务端的ip，黑名单与白名单拦截、授权等。

Gateway的可拓展作用是通过添加不同的插件或模块来扩展其功能，例如增加防火墙功能、代理功能、负载均衡等服务

Gateway需要针对不同的协议进行相关设置。例如，在HTTP协议中，可以通过设置请求头中的特定信息，在TCP协议中，可以在传输层添加特定的数据字段。另外，Gateway还可以使用加密技术对客户的信息进行安全传输

### 设计模式？模板模式？适配器模式、模板模式了解吗？

工厂模式（Factory Pattern）：将对象的创建过程封装在一个工厂类中，用户只需调用工厂类的方法来创建对象，实现了对象创建和使用的分离

单例模式（Singleton Pattern）：确保一个类只有一个实例，并提供一个全局访问点

观察者模式（Observer Pattern）：定义对象间的一对多依赖关系，当一个对象状态发生变化时，所有依赖它的对象都会收到通知并自动更新

适配器模式（Adapter Pattern）：将一个类的接口转换成客户希望的另一个接口，使原本不兼容的接口能够一起工作

装饰器模式（Decorator Pattern）：动态地给一个对象添加一些额外的职责，就扩展功能而言，Decorator模式比生成子类更为灵活

策略模式（Strategy Pattern）：定义一系列的算法，把它们封装起来，并使它们可以互相替换

模板模式（Template Pattern）：定义一个算法的框架，将一些步骤延迟到子类实现，从而使子类可以不改变算法结构而重定义算法的某些步骤

模板模式是一种行为设计模式，其定义了一个算法的骨架，将一些步骤延迟到子类实现，从而使子类可以不改变算法结构而重定义算法的某些步骤。通过模板模式，可以将不变的部分抽象到父类中，将可变的部分留给子类去实现，在实现时只需要关注业务逻辑即可

### 遇到过进程占用CPU特别高的情况吗？怎样排查、定位是哪个进程引起的？按步骤说一套思路。

使用命令行工具如top（Unix/Linux）查看系统的进程和资源占用情况

使用jstack命令获取线程堆栈信息，通过jps命令获取Java进程的进程ID（PID），然后使用jstack [PID]命令获取该进程的线程堆栈信息。这将输出当前Java进程中所有线程的堆栈跟踪

分析线程堆栈信息：在获取到的线程堆栈信息中，关注那些CPU占用率较高的线程。通常，这些线程的堆栈跟踪中会包含正在执行的代码行。通过分析这些代码行，我们可以定位到导致CPU升高的具体代码段

### Dubbo里面的nacos是做什么用的？nacos的作用？提供了什么能力？

在Dubbo框架中，Nacos是一个开源的服务发现和配置管理平台，用于实现微服务架构中的服务注册、发现、配置和管理。Nacos的主要作用包括以下几个方面：

服务注册和发现：Nacos作为服务注册中心，服务提供者在启动时将自己的服务信息注册到Nacos中心，服务消费者可以通过Nacos查询到各个服务的地址和相关信息，实现服务之间的通信与调用。

动态配置管理：Nacos提供了配置中心的功能，可以将配置信息存储在Nacos中心，并动态更新配置内容。服务可以通过Nacos实时获取最新的配置信息，实现服务配置的动态管理。

服务健康检查：Nacos可以实时监控服务的健康状态，当服务不可用时可以及时通知服务治理中心，实现自动的服务实例下线和上线。

负载均衡：Nacos提供了负载均衡能力，可以为服务提供者和消费者之间的调用提供负载均衡策略，实现服务调用的负载均衡和高可用。

服务路由与降级：Nacos支持服务路由和降级功能，可根据服务的实时状态和负载情况进行服务路由和降级操作，以提高系统的可靠性和性能

### 垃圾回收等技术原理。jvm 垃圾回收机制？回收的算法都有哪几种？

垃圾回收是程序在运行过程中自动释放不再使用的内存空间的过程，以避免内存泄漏和提高内存的利用率。在Java虚拟机（JVM）中，垃圾回收机制是通过不断地对内存进行检查和清理来实现的。

JVM的垃圾回收机制主要包括以下三个部分：

标记阶段（Mark）：遍历所有的对象，并标记哪些对象是可达的（活动对象），哪些是不可达的（垃圾对象）。

清除阶段（Sweep）：清除所有标记为不可达的对象，并释放它们占用的内存空间。

整理阶段（Compact）：将存活的对象移动到一端，释放出连续的内存空间。

JVM的垃圾回收算法主要包括以下几种：

标记-清除算法（Mark-Sweep）：最基本的垃圾回收算法，通过标记不可达对象并清除它们来释放内存。

复制算法（Copying）：把内存划分为两个区域，每次只使用其中一个区域，将存活对象复制到另一个区域，然后清除当前区域中的所有对象，交换区域角色。

标记-整理算法（Mark-Compact）：在标记阶段标记不可达的对象，然后将存活对象移动到一端，最终清除不可达对象，整理内存。

分代算法（Generational）：将内存空间划分为新生代和老年代，针对不同代的对象使用不同的垃圾回收算法，新生代通常使用复制算法，老年代通常使用标记-清除或标记-整理算法。

并发标记清除算法（CMS）：在标记和清除阶段中，允许在垃圾回收同时进行程序的执行，减少应用程序的暂停时间。

不同的算法适用于不同的场景和要求，可以根据实际情况来选择合适的算法。

### jdk的动态代理有哪几种？

JDK动态代理（基于接口）：基于接口的动态代理是通过接口来生成代理对象，代理类实现了被代理接口，对所有方法进行统一处理。在使用动态代理时，需要提供一个实现InvocationHandler接口的处理器类，实现invoke()方法来实现对被代理方法的增强。

CGLIB动态代理（基于继承）：CGLIB动态代理是通过继承被代理类来生成代理对象，代理类继承了被代理类，对所有非final方法进行复写。CGLIB代理在运行时动态生成字节码，不要求被代理类实现接口

### 关注下mangoDB，消息队列，rabbitMQ kafka，队列锁，contrller注解

MongoDB是一种NoSQL数据库，具有高可扩展性、灵活的数据模型和强大的查询语言。MongoDB存储数据的方式为文档型数据库，数据以文档的形式存储在集合中。 MongoDB支持复制和分片，适用于大规模数据存储和高并发访问场景

RabbitMQ和Kafka：RabbitMQ和Kafka都是流行的消息队列系统。RabbitMQ是一个基于AMQP协议的消息队列系统，提供了可靠的消息传递和灵活的消息路由模式。Kafka是一个分布式流处理平台，支持高吞吐量的消息处理和数据存储，常被用于日志采集、数据传输和流处理

队列锁是一种实现多线程间协作的锁机制，它基于队列的数据结构来保证资源的互斥访问。线程通过入队和出队操作来获取和释放锁，实现了对共享资源的安全访问。

在Spring框架中，Controller注解用于标识一个类为控制器类，处理用户的HTTP请求并返回响应。Controller注解通常与RequestMapping注解一起使用，用来映射URL请求路径到具体的处理方法，实现路由和请求处理的功能

### Redis 怎么清除key的？

DEL命令：DEL命令用于删除指定的Key，可以删除单个Key，也可以同时删除多个Key

EXPIRE命令：EXPIRE命令用于设置Key的过期时间，当Key到期后，会自动被删除

### kafka 消费组分区、topic都是啥意思？同一个消费组的读取到的数据会不会重复？

主题（Topic）：主题是Kafka中的基本数据单元，每条消息都会被发布到一个特定的主题中。主题是逻辑上的概念，相当于一个消息分类，类似于消息队列中的队列。

分区（Partition）：分区是主题的一个物理上的分割，用于实现数据的并行处理。每个主题都可以被划分为多个分区，每个分区可以在不同的Broker节点上进行复制和存储。分区内的消息有序存储，但不同分区之间的消息顺序不能保证。

消费组（Consumer Group）：消费组是消费者的逻辑分组，用于对同一个主题的消息进行消费。一个消费组内的消费者共同消费一个主题中的消息，每个消息只会被组内的一个消费者处理。

对于同一个消费组的消费者从同一个主题中消费消息时，它们会根据分配的分区进行消费，每个分区在同一时间点只能被一个消费者消费，确保了一条消息只会被同一个消费组内的一个消费者处理。消费组内的不同消费者会分配到不同的分区，因此不会出现数据重复消费的情况

### 多线程怎么开发？

继承Thread类：创建一个继承自Thread类的子类，重写run()方法来定义线程执行的任务，然后创建线程对象并调用start()方法启动线程

实现Runnable接口：创建一个实现Runnable接口的类，实现run()方法来定义线程执行的任务，然后创建线程对象并以该Runnable对象作为参数创建Thread对象，并调用其start()方法启动线程

使用线程池：使用线程池来管理多个线程的执行，可以减少线程的创建和销毁开销，提高性能。通过Executors工具类可以创建不同类型的线程池，如FixedThreadPool、CachedThreadPool等

使用Callable和Future：Callable接口对Runnable进行了扩展，可以返回执行结果。Future接口表示异步计算的结果，通过Future对象可以获取线程执行的结果

### Gateway需要调用子服务，怎么把当前的信息透传过去？

在使用Spring Cloud Gateway来实现API网关并调用子服务时，可以通过Gateway的过滤器（Filter）来实现透传当前请求的信息到子服务。可以在Gateway中编写自定义的过滤器，通过获取当前请求的信息并在转发请求给子服务时携带这些信息

通过自定义过滤器，可以实现在Gateway中透传当前请求的信息，包括Header、参数等到子服务，从而实现更灵活的请求处理和转发。在实际应用中，可以根据具体场景扩展和定制自己的过滤器逻辑

### 如果用哈希命令的话，怎么get到哈希里面的所有类？

在Redis中，可以使用HGETALL命令来获取哈希（Hash）数据类型中所有的字段和值。HGETALL命令会返回哈希中所有字段和对应的值的列表

使用HGETALL命令获取哈希数据类型中所有的字段和值，该命令返回一个包含所有字段和值的数组

返回结果会以字段-值对的形式存储在一个数组中，字段和对应值两两成对，可以通过遍历数组来逐一获取每个字段和值

### Spring Cache是什么的东西？缓存时间怎么设置？

Spring Cache 是 Spring 框架提供的一种缓存抽象机制，用于简化应用程序中的缓存操作。它通过缓存注解的方式，可以方便地将方法的返回结果缓存起来，避免重复调用耗时的方法。Spring Cache 支持多种缓存实现，如 ConcurrentMapCache、EhCache、Redis 等，开发者只需要通过配置即可使用相应的缓存实现。

设置缓存时间可以借助 @Cacheable 和 @CachePut 注解上的 expire 或 ttl 属性，或者在缓存配置中进行全局配置。具体设置取决于缓存实现的不同

### JVM了解吗？都有哪些东西？垃圾回收机制？

JVM包括以下主要组件：

1. 类加载器（ClassLoader）：负责将类文件加载到内存中，并生成对应的Class对象。

2. 运行时数据区（Runtime Data Areas）：包括方法区、堆、栈、程序计数器和本地方法栈等。

方法区：用于存储类的结构信息、静态变量、常量等。

堆：用于存储对象实例，是垃圾回收的主要区域。

栈：存储局部变量、方法参数和部分结果，并管理方法的调用和返回。

程序计数器：记录当前线程执行的字节码指令地址。

本地方法栈：为Java调用本地方法（Native Method）服务。

3.执行引擎（Execution Engine）：负责解释和执行Java字节码，包括解释器和即时编译器（JIT）。

4. 本地方法接口（Native Interface）：允许Java代码调用本地方法，与本地库交互。

5. 垃圾回收器（Garbage Collector）：负责自动管理堆内存，回收不再使用的对象，以防止内存泄漏。

垃圾回收器实现了不同的垃圾回收算法，包括：

1. 标记-清除算法（Mark and Sweep）：标记出所有活动对象，然后清除未被标记的对象。

2. 复制算法（Copying）：将存活对象复制到另一个区域，并清除非存活对象。

3. 标记-整理算法（Mark and Compact）：标记并移动存活对象，然后清除非存活对象，使内存空间连续。

### Mysql优化都做过哪些动作？

合适的数据库设计：

优化表结构，避免过多的关联和复杂的查询。

索引优化：

确保使用了合适的索引，避免全表扫描。

避免创建过多的索引，因为每个索引都会增加写入和存储成本。

定期检查和优化索引，删除不再需要的索引，避免冗余和重复索引。

查询优化：

编写高效的SQL查询，避免使用SELECT \*，只选择所需的列。

尽量避免使用子查询，特别是在性能敏感的场景下。

使用EXPLAIN来分析查询执行计划，优化查询性能。

### 分布式事务？CAP理论？

在分布式系统中，如果涉及多个节点（或者称为服务、数据库等）之间的操作需要保证一致性，就需要考虑分布式事务。传统的事务处理是在单个数据库中进行的，而分布式事务需要处理跨多个数据库或服务的事务操作。

CAP理论是由计算机科学家Eric Brewer提出的，用于描述分布式系统的三个基本属性：

一致性（Consistency）：在分布式系统中，所有节点看到的数据都是一致的，即在任何时刻，不同节点的数据应该是相同的。

可用性（Availability）：在分布式系统中，系统至少需要对用户的请求做出响应，即系统保持可用的状态，不会因为节点故障而无法提供服务。

分区容忍性（Partition Tolerance）：即使系统中出现网络分区，导致节点之间无法通信，系统仍然能够正常运行。

CAP理论指出，分布式系统中最多只能同时满足其中的两个属性，无法同时保证三者。这是因为在出现网络分区时，要么保证一致性和可用性，但此时需要牺牲分区容忍性；要么保证可用性和分区容忍性，但此时需要牺牲一致性。因此，在设计分布式系统时，需要根据具体业务需求和实际情况来权衡这三个属性。

### 哈希和一致性哈希有什么区别？Redis哈希命令用过没？

哈希和一致性哈希都是在分布式系统中用于数据分片和负载均衡的技术，但它们有一些不同之处。

哈希：

在哈希技术中，数据的分片是通过对数据的键（Key）进行哈希计算，然后将哈希值映射到特定的分片（或节点）上。通常，这种哈希算法是将键的哈希值对分片数量取模来确定数据所在的分片。例如，如果有3个分片，那么数据的哈希值对3取模，得到的余数就是数据所在的分片编号。

哈希的优点是简单易实现，但在节点的动态增减时，会导致大量数据的迁移，影响系统的稳定性和性能。

一致性哈希：

一致性哈希是为了解决哈希在节点动态变化时导致数据迁移的问题而提出的。它使用一个哈希环（Hash Ring）来表示所有可能的分片，每个节点在哈希环上占据一个位置。数据的哈希值通过在哈希环上顺时针查找，直到找到第一个大于等于该哈希值的节点位置，然后将数据分配给该节点。

一致性哈希的优点是在节点变化时，只有部分数据需要迁移，而不是全部数据，因此可以减少数据迁移的开销，提高系统的稳定性和性能。

### JVM指令重排？

指令重排是指编译器或处理器在不改变程序语义的前提下，重新排序指令的执行顺序，以提高执行效率或利用硬件并行性。在JVM中，指令重排可能会导致多线程程序的并发问题。

### kafka往ES里写数据，需要注意什么？

数据格式转换：

确保Kafka中的数据格式与Elasticsearch中的索引映射相匹配。如果数据格式不匹配，可能会导致数据写入失败或索引中的数据结构不一致。

数据批量写入：

考虑使用批量写入的方式将数据从Kafka写入Elasticsearch，以减少网络开销和提高写入性能。

数据去重：

在写入Elasticsearch之前，需要考虑如何处理重复的数据。你可以在写入数据之前进行去重处理，以确保Elasticsearch中的数据唯一性。

异常处理和重试机制：

实现适当的异常处理和重试机制，以处理由于网络问题或Elasticsearch集群故障而导致的写入失败情况。

数据一致性：

在将数据从Kafka写入Elasticsearch时，要确保数据的一致性。这包括写入数据的顺序性和正确性，以及保证数据写入后能够被查询到。

### Redis 缓存和更新，怎么样更新到数据库？

读取数据：当需要获取数据时，首先尝试从Redis缓存中获取数据。

检查缓存：如果缓存中存在数据，则直接返回数据。

从数据库获取数据：如果缓存中不存在数据，则从数据库中获取数据。

写入缓存：获取数据后，将数据写入Redis缓存中，以便下次查询时能够直接从缓存中获取数据。

更新数据库：当需要更新数据时，首先更新数据库中的数据。

清除缓存：在某些情况下，可以选择直接清除缓存中的相关数据，以便下次查询时能够重新从数据库中获取最新数据。

### 分布式锁的区别、优缺点？zk 和 Redis

ZooKeeper 分布式锁

优点：

强一致性：ZooKeeper提供了强一致性的数据模型，可以确保所有客户端在同一时间看到相同的数据视图。

可靠性：ZooKeeper是一个高可靠性的分布式协调服务，通常具有较少的故障和性能问题。

顺序性：ZooKeeper的节点是有序的，可以利用节点的序列化特性来实现分布式锁。

缺点：

复杂性：ZooKeeper的部署和维护相对复杂，需要专业的知识和经验。

性能：ZooKeeper的性能通常不如Redis等内存型数据库，尤其在高并发情况下可能会成为瓶颈。

Redis 分布式锁

优点：

简单易用：Redis提供了直观的命令和数据结构来实现分布式锁，使用起来比较简单。

性能：Redis是一个高性能的内存型数据库，能够快速处理大量的请求，适用于高并发的场景。

可扩展性：Redis支持集群模式，可以通过横向扩展来提高性能和容量。

缺点：

一致性：Redis的数据模型可能存在一致性问题，尤其是在网络分区或故障恢复时。

数据持久化：Redis的数据通常是存储在内存中的，需要通过持久化机制来确保数据不丢失，这可能会影响性能。

原子性：Redis的分布式锁实现通常需要多个命令来完成，可能存在原子性问题，需要注意处理竞态条件。

### 如何根据堆栈的内容，找到某个类？

通过异常信息里面的关键代码多少行来判断

### 说下对docker的理解？

Docker是一种用于构建、发布和运行应用程序的开源平台。它基于容器化技术，可以将应用程序及其所有依赖项打包到一个标准化的容器中，并在任何支持Docker的环境中运行

### docker镜像重启的话，还在不在？

在

### 部署K8S时用的是哪个组件？

deployment service ingress statefulset crd

### 介绍2个pod间的通信？流量是如何过去的？

在Kubernetes中，Pod之间的通信通常通过以下几种方式进行：

使用服务（Service）：

Kubernetes的Service是一种抽象，用于定义一组Pod的访问方式和网络终结点。Service提供了一个稳定的虚拟IP和端口，用于接收来自其他Pod或外部客户端的流量，并将流量负载均衡到后端的一组Pod上。

当一个Pod需要与另一个Pod通信时，它可以通过Service的虚拟IP和端口来访问目标Pod。Kubernetes会将流量路由到对应Service的后端Pod上，实现Pod之间的通信。

使用集群IP和端口：

在同一个集群内，Pod可以直接使用目标Pod的IP地址和端口进行通信。Kubernetes集群内部有一种叫做Pod间通信的网络，它通过网络插件（如Flannel、Calico等）为每个Pod分配一个唯一的IP地址，并允许Pod之间直接进行网络通信。

当一个Pod需要与另一个Pod通信时，它可以通过目标Pod的IP地址和端口来直接发送流量，不需要经过额外的中间件或服务。

在这两种通信方式下，流量的路由和负载均衡是由Kubernetes的网络插件和控制平面来实现的。流量从源Pod出发，经过Kubernetes的网络层，根据目标Pod的IP地址或Service的虚拟IP和端口，最终到达目标Pod。如果使用Service，则Service会根据负载均衡策略将流量转发到后端的多个Pod上；如果直接使用集群IP和端口，则流量会直接路由到目标Pod上。

### Mysql分库分表使用过吗？

分库分表是一种数据库水平扩展的策略，用于解决单一数据库的性能瓶颈和存储容量限制。它通过将数据分散存储在多个数据库实例或多个数据库表中，从而提高了数据库的吞吐量和并发能力。

在MySQL中，分库分表通常涉及以下几个方面：

分库（Sharding）：将数据按照一定的规则（如哈希、范围、取模等）分散存储到多个数据库实例中。每个数据库实例通常包含相同的表结构，但存储不同范围的数据。分库可以减少单个数据库的负载压力，并且能够水平扩展存储容量。

分表（Sharding）：将数据按照一定的规则（如哈希、范围、取模等）分散存储到多个数据库表中。每个数据库表通常包含相同的数据结构，但存储不同范围的数据。分表可以减少单个数据库表的数据量，提高查询性能和并发能力。

路由策略（Routing）：根据分库分表的规则，将查询请求路由到相应的数据库实例或数据库表中。通常需要在应用程序中实现路由逻辑，以确保查询能够正确地路由到目标数据库。

事务和一致性：分库分表可能会导致跨库事务和分布式事务的复杂性增加，需要特别注意事务的边界和一致性保证。

### Mysql 优化做过吗？

合适的数据类型：选择合适的数据类型可以减少存储空间的占用和提高查询性能。例如，使用INT代替VARCHAR存储数字，避免使用TEXT或BLOB类型存储大量文本数据。

索引优化：合理创建和使用索引可以加快查询速度。根据查询需求和字段的选择性，创建适当的单列索引、复合索引或覆盖索引。同时，避免创建过多的索引，以减少写操作的开销。

查询优化：编写高效的SQL查询语句是提高性能的关键。避免使用SELECT \*查询所有列，只选择需要的列；合理使用WHERE子句和JOIN操作；避免在列上进行函数操作或计算，以充分利用索引。

表结构优化：优化表结构可以提高查询性能和降低存储空间的使用。避免使用过多的垂直分割或水平分割，合理设计表的范式和反范式。

### 如何知道当前使用的索引是最优的呢？

执行计划分析：通过执行查询的执行计划来分析MySQL是如何执行该查询的。执行计划可以告诉你MySQL选择了哪个索引以及它是如何使用该索引的。可以使用EXPLAIN语句或者相关的工具（如MySQL Workbench）来获取查询的执行计划。

检查索引覆盖：如果执行计划中显示使用了索引但没有额外的查找操作（例如索引扫描或索引搜索），则说明该索引是覆盖索引，这通常是最佳的情况。因为MySQL可以直接从索引中获取所需的数据，而不必再进行额外的查找操作。

## ## 高级研发可能问到的问题

### 微服务的容错是什么，容错的目的是什么？服务熔断或请求限流机制，熔断状态有几种？

微服务的容错是指在分布式系统中，为了提高系统的稳定性和可靠性，在服务之间进行通信时，引入一些技术手段来处理服务之间的故障或异常情况。容错机制的目的是尽可能避免故障在整个系统中的传播，保障系统的正常运行

关闭状态（Closed）：系统运行正常，熔断器不会开启，请求会正常传递给服务。

打开状态（Open）：当故障达到阀值时，熔断器会打开，后续的请求不再传递给服务，直接返回熔断器配置的响应。

半开状态（Half-Open）：经过一段时间后，熔断器会进入半开状态，允许部分请求通过，检测服务是否恢复，如果恢复正常，则熔断器会关闭，否则会再次打开。

### 有无带过研发团队？团队的组成？大家的作用

### 怎样定义团队里技术比较好的人员？从专业上如何判断一个开发人员的技术能力？

技术广度：开发人员在多个领域有所涉猎并深入理解，具备较全面的技术知识，能够熟练应用多种技术和工具解决问题。

技术深度：开发人员在自己擅长的领域有深入的理解和实践经验，能够深入分析问题并提出有效的解决方案。

项目经验：开发人员具有丰富的项目经验，能够独立完成项目开发，并具备项目管理和团队协作能力。

学习能力：开发人员具备快速学习和适应新技术的能力，保持对行业发展趋势的关注，并能够持续提升自己的技术能力。

问题解决能力：开发人员能够独立解决复杂的技术难题，善于分析问题根源，提出有效的解决方案。

协作能力：开发人员具有良好的团队合作精神，能够有效地与团队成员合作，推动项目进展并解决团队遇到的问题

### 为啥最近很多人舍弃 zookeeper,用nacos了？zookeeper做注册中心，底层用什么实现？

功能更全面：Nacos 提供了服务注册与发现、配置管理和服务治理等全面的功能，包括配置中心、服务发现、负载均衡、流量管理等，而 Zookeeper 主要用于服务注册中心，功能相对较单一。

更易用：Nacos 提供了更友好的用户界面和丰富的 RESTful API，方便用户进行管理和操作，而 Zookeeper 的使用相对更复杂。

落地更方便：Nacos 支持更多编程语言的 SDK，更容易集成到不同的应用中，同时也支持多种部署方式，如单机、集群、高可用等。

活跃度和社区支持：Nacos 是阿里巴巴开源的项目，拥有活跃的开发团队和社区支持，更新迭代速度较为快速，问题得到更及时的解决。

Zookeeper 在服务注册中心中的底层实现主要包括两部分：

Zookeeper 数据模型：Zookeeper 将数据以树形结构（ZNode）的方式进行组织和存储，类似于文件系统，每个 ZNode 会存储一些数据或元信息。这种树形结构的数据模型为服务注册和发现提供了基础。

Zookeeper 服务节点：Zookeeper 服务节点是实际存储服务注册信息的节点，服务提供者在启动时会在指定路径下创建临时节点并将自身的地址信息写入，服务消费者在发现服务时会从对应路径下读取节点信息，实现服务的注册和发现

### 全链路灰度有几种方式？灰度标在什么时间点打上？

全链路灰度发布可以通过多种方式实现，常见的方式包括：

基于用户：将新版本的功能或服务仅对部分用户或用户组开放，逐步扩大到全部用户。

基于流量：将一部分流量导向新版本，逐步增加新版本的流量比例。

基于区域：将新版本在特定区域或节点进行部署，逐步扩大到其他区域。

基于实例：将新版本在特定服务实例中部署，逐步扩大到其他实例

灰度标通常在新版本部署之后、但正式上线之前打上，用于标记当前是灰度发布阶段

## --------一些反复问到的点（所有级别都需要关注）------

### java设计模式

常见的 Java 设计模式包括：

创建型模式（Creational Patterns）：

单例模式（Singleton）：确保一个类只有一个实例，并提供一个全局访问点。

工厂模式（Factory）：定义一个用于创建对象的接口，由子类决定实例化哪一个类。

抽象工厂模式（Abstract Factory）： 提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类。

原型模式（Prototype）：用原型实例指定创建对象的种类，并通过复制这个原型来创建新对象。

结构型模式（Structural Patterns）：

适配器模式（Adapter）：将一个类的接口转换成客户希望的另一个接口。

桥接模式（Bridge）：将抽象部分与它的实现部分分离，使它们可以独立地变化。

装饰者模式（Decorator）：动态地给一个对象添加一些额外的职责。

组合模式（Composite）：将对象组合成树形结构以表示“部分-整体”的层次结构。

行为型模式（Behavioral Patterns）：

观察者模式（Observer）：定义对象间的一种一对多依赖关系，当一个对象的状态发生变化时，它的所有依赖者都将得到通知并自动更新。

策略模式（Strategy）：定义一系列的算法，将每个算法封装起来，并使它们可以互相替换。

模版方法模式（Template Method）：定义一个操作中的算法的框架，而将一些步骤延迟到子类中。

责任链模式（Chain of Responsibility）：避免将请求的发送者与接收者耦合在一起，让多个对象都有机会处理这个请求。

### Spring Cloud：

#### Spring Cloud Gateway 和 Zuul 的区别和使用场景——

（1）Spring Cloud Gateway：

Spring Cloud Gateway 是 Spring Cloud 的新一代的网关组件，基于 Project Reactor 和 Spring WebFlux 开发而成。

Spring Cloud Gateway 支持动态路由、断路器、过滤器等功能。

Spring Cloud Gateway 相对于 Zuul 2.x 有更高的性能和更好的扩展性。

适合于需要高性能和高可扩展性的微服务架构。

（2）Zuul：

Zuul 是 Netflix 提供的一个基于 Java 的网关服务，目前最新版本是 Zuul 2.x。

Zuul 支持动态路由、服务发现、过滤器等功能。

Zuul 在单节点上的性能较好，但在大规模集群中会有性能瓶颈。

适合于中小型项目或者对性能要求不高的项目。

因此，如果你的项目需要高性能和高扩展性，建议选择 Spring Cloud Gateway；如果你的项目规模较小或者对性能要求不高，可以选择 Zuul。在具体选择时，还要结合自身项目需求进行评估。

#### Spring Cloud 的基本概念和架构——

Spring Cloud 是一个基于 Spring Boot 的开源微服务框架，它提供了一套完整的解决方案，用于构建分布式系统中的各种微服务架构。下面是 Spring Cloud 的基本概念和架构：

1. 服务注册与发现（Service Discovery）：Spring Cloud 提供了服务注册与发现的解决方案，允许微服务应用注册到注册中心，并通过服务发现来实现服务之间的调用。

2. 负载均衡（Load Balancing）：Spring Cloud 提供了负载均衡的功能，可以自动将请求分发到服务集群中的各个实例。

3. 断路器（Circuit Breaker）：Spring Cloud 提供了断路器模式，可以实现对服务调用失败的熔断保护，防止雪崩效应。

4. 配置中心（Configuration Management）：Spring Cloud 提供了分布式配置管理的功能，可以集中管理配置信息，并实现动态刷新配置。

5. 网关（Gateway）：Spring Cloud 提供了网关服务，用于统一管理所有微服务的入口，实现路由、过滤、监控等功能。

6. 分布式消息（Messaging）：Spring Cloud 提供了消息驱动的开发模式，支持基于消息中间件的微服务之间的通信。

7. 分布式链路追踪（Distributed Tracing）：Spring Cloud 提供了分布式链路追踪的功能，可以跟踪和监控微服务调用链路。

Spring Cloud 架构是一个灵活可扩展的微服务体系，通过上述核心组件的组合和配合，可以构建复杂的分布式系统。开发人员可以根据项目需求选择合适的组件来构建自己的微服务架构。

#### zookeeper 和 Nacos 的区别和使用场景——

Zookeeper 和 Nacos 都是用于微服务架构中的注册中心和配置中心的开源软件，它们的区别和使用场景如下：

1. Zookeeper：

- Zookeeper 是一个分布式的协调服务，可以用作服务发现、分布式锁、配置管理等。

- Zookeeper 使用 Apache Zab 协议实现数据的一致性，保证数据的可靠性和一致性。

- Zookeeper 在一些大型项目中被广泛使用，例如 Hadoop、Kafka 等。

- Zookeeper 相对比较重量级，配置较为复杂，需要专门的运维人员进行管理。

2. Nacos：

- Nacos 是一个新一代的分布式服务发现和配置管理平台，支持服务发现、配置管理、动态 DNS 等功能。

- Nacos 是一个更轻量级的解决方案，易于部署和管理，并提供了更为丰富的功能。

- Nacos 支持多种注册中心协议和配置管理协议，并提供了可视化界面和 API 接口。

- Nacos 可以作为 Spring Cloud、Dubbo 等微服务框架的注册中心和配置中心。

使用场景：

- 如果你需要一个经过验证、稳定且性能优异的分布式协调服务，可以选择 Zookeeper。

- 如果你项目对性能要求不高，同时希望快速部署、易于使用、提供更丰富的功能和可视化界面，可以选择 Nacos。

需要根据实际项目需求和团队技术栈来选择合适的注册中心和配置中心。

#### Feign基本概念及使用场景，底层原理——

Feign 是一个声明式的 HTTP 客户端，用于简化服务间的调用。它基于注解和接口代理，通过定义接口的方式来实现对服务的调用，使得服务间的调用更加简洁和易于维护。下面是 Feign 的基本概念及使用场景：

1. 基本概念：

- 接口定义：通过定义接口来描述服务提供方的 API，包括请求方法、路径、参数等信息。

- 注解：使用 Feign 提供的注解来定义接口的参数、请求方法、请求路径等信息，使得接口定义更加灵活。

- 负载均衡：Feign 集成了 Ribbon 负载均衡器，可以实现服务调用时的负载均衡。

2. 使用场景：

- 用于微服务架构中的服务间调用：Feign 可以简化微服务架构中服务之间的调用，减少重复代码的编写。

- 用于传统项目的 HTTP 调用：Feign 可以作为传统项目中的 HTTP 客户端来使用，简化 HTTP 请求的发送过程，提高开发效率。

底层原理：

Feign 使用了动态代理技术，将对接口的调用转化为对 HTTP 请求的调用。当定义了一个接口并使用 Feign 注解后，Feign 会根据接口的定义创建一个代理对象，当调用接口方法时，实际上是通过 Feign 发起了一个 HTTP 请求。Feign 还集成了 Ribbon 负载均衡器，可以实现服务调用时的负载均衡。最终，Feign 将返回的 HTTP 响应转化为对应的 Java 对象，方便在业务逻辑中直接使用。

总之，Feign 是一个方便、简洁的 HTTP 客户端，可以有效简化服务间调用的编码工作，提高开发效率。

#### Ribbon 和 Hystrix 的作用和使用场景——

Ribbon 和 Hystrix 是 Netflix 提供的两个在微服务架构中常用的组件，分别用于负载均衡和熔断降级。它们的作用和使用场景如下：

1. Ribbon：

- 作用：Ribbon 是一个客户端负载均衡器，用于在服务消费方实现负载均衡，将请求均匀地分发到不同的服务实例上。

- 使用场景：

- 在微服务架构中，当有多个服务实例提供同一服务时，可以使用 Ribbon 来实现负载均衡，提高系统的可用性和性能。

- 可以与 Feign 等客户端框架配合使用，实现对服务的调用和负载均衡。

2. Hystrix：

- 作用：Hystrix 是一个熔断器框架，用于实现服务降级、隔离、熔断等策略，防止故障蔓延导致整个系统崩溃。

- 使用场景：

- 在微服务架构中，当一个服务发生故障或超时时，可以使用 Hystrix 实现服务降级，返回一个默认值或者提示信息，保证用户体验。

- 可以通过监控和配置来调整熔断策略，避免因服务不可用或异常导致整个系统的故障。

综上所述，Ribbon 主要用于实现客户端负载均衡，可以提高系统的可用性和性能；而 Hystrix 主要用于实现服务降级和熔断，保护系统免受故障影响。在微服务架构中，通常会同时使用 Ribbon 和 Hystrix 来确保系统的稳定性和可靠性。

### 微服务架构:

#### 微服务架构的优缺点——

微服务架构是一种将单个应用程序划分为一组小型、松散耦合的服务的软件架构设计。微服务架构的优缺点如下：

优点：

1. 高可伸缩性：微服务架构允许每个服务单独部署、升级和扩展，使得系统更容易实现横向扩展，提高了系统的可伸缩性。

2. 独立部署：每个微服务都可以独立部署，不会影响其他服务的正常运行，降低了发布的风险，提高了系统的灵活性。

3. 技术多样性：每个微服务可以使用不同的技术栈和编程语言，使得团队可以根据具体需求选择最适合的技术，提高了开发效率。

4. 容错性：由于微服务之间是松散耦合的，一个服务的故障不会影响到其他服务的运行，提高了系统的容错性和稳定性。

5. 易于维护：微服务的代码量较小，容易理解和维护，团队可以更快地进行问题定位和修复。

缺点：

1. 开发复杂性：微服务架构中的服务较多，服务之间的调用会增加复杂性，对团队的技术水平和协作能力有一定要求。

2. 增加网络通信成本：微服务架构中服务之间通常通过网络进行通信，可能会增加网络通信的成本和延迟。

3. 数据一致性：不同的微服务可能会拥有各自独立的数据存储，需要保证数据一致性和事务管理，增加了数据一致性的难度。

4. 系统的集成测试和部署复杂性：微服务架构中的服务较多，需要进行更多的集成测试和持续部署操作，增加了部署的复杂性和成本。

综合来看，微服务架构具有灵活、可伸缩、易于维护等优点，但也存在着开发复杂性、网络通信成本增加等缺点。在选择采用微服务架构时，需要根据具体项目需求和团队实际情况来权衡利弊。

#### 服务之间的通信方式——

在微服务架构中，不同的服务之间需要进行通信，常见的服务之间通信方式有以下几种：

1. HTTP/RESTful API：使用 HTTP 协议进行通信，通过 RESTful API 定义服务之间的接口和数据传输格式。这是最常见的微服务通信方式，适用于前后端分离的场景。

2. gRPC：gRPC 是一种高性能的远程过程调用（RPC）框架，基于 HTTP/2 协议，支持多种语言。通过 Protocol Buffers 定义服务接口和消息格式，适用于多语言、跨平台的微服务架构。

3. 消息队列：使用消息队列来实现异步通信，例如使用 Kafka、RabbitMQ 等消息中间件。服务通过发布订阅或点对点模式发送和接收消息，实现解耦和独立性。

4. RPC（Remote Procedure Call）：使用远程过程调用技术，如 Dubbo、Thrift 等框架，实现服务之间的调用。RPC 可以实现更快速的服务调用和返回，但对网络和通信效率有一定要求。

5. WebSockets：用于实现双向通信的协议，适用于需要实时性、互动性的场景。可以用于服务之间的实时通信和数据推送。

选择合适的通信方式取决于项目需求和架构设计，可以根据场景选择最适合的通信方式。常见的是使用 HTTP/RESTful API 进行服务间通信，但对于实时性要求高的场景可以考虑使用消息队列或 WebSockets。

#### 服务容错和降级——

服务容错和降级是在微服务架构中非常重要的概念，用于保护系统免受故障和故障蔓延的影响。下面分别介绍服务的容错和降级：

1. 服务容错（Service Resilience）：

- 服务容错是指系统在面对错误或异常时能够保持正常运行，并且尽可能快速地恢复到正常状态。

- 常见的服务容错机制包括断路器、重试、超时、限流等：

- 断路器（Circuit Breaker）：当服务发生故障时，断路器会打开并且短暂地停止服务调用，避免故障蔓延，起到保护系统的作用。

- 重试：在调用服务发生失败时，会进行一定次数的重试，尝试恢复服务调用。

- 超时：设置合理的超时时间，防止请求长时间等待导致资源浪费。

- 限流：对服务的请求进行限流处理，防止过多请求导致系统超载。

2. 服务降级（Service Degradation）：

- 服务降级是指在系统出现宕机或服务性能下降时，减少服务的功能或质量，以保证核心功能的可用性。

- 服务降级可以通过返回默认值、缓存数据、关闭某些功能模块等方式来实现。

- 服务降级的目的是防止系统崩溃或资源耗尽，提高系统的稳定性和可用性。

综合来看，服务容错和降级都是为了提高系统的可靠性和稳定性，及时响应错误或异常，减少故障对系统的影响。在实际应用中，可以结合断路器、重试、超时、限流等机制进行服务容错，并通过服务降级保证系统的核心功能不受影响，提升系统的容错能力。

### 异常排查:

#### 常见异常类型和处理方法——

在软件开发过程中，常见的异常类型和处理方法如下：

1. NullPointerException（空指针异常）：

- 异常情况：当试图访问 null 对象的属性或调用 null 对象的方法时抛出。

- 处理方法：在访问对象属性或调用对象方法前，进行 null 值检查或使用 Optional 类来避免空指针异常。

2. IllegalArgumentException（非法参数异常）：

- 异常情况：当传递给方法的参数不符合方法的预期条件时抛出。

- 处理方法：在方法内部进行参数合法性检查，确保参数符合预期条件。可以使用断言（assert）或条件判断来检查参数的有效性。

3. IllegalStateException（非法状态异常）：

- 异常情况：当对象处于不允许的状态时抛出。

- 处理方法：在对象的状态发生变化时，确保状态的合法性。使用状态机或状态模式来管理对象的状态，避免出现不合法的状态异常。

4. ArithmeticException（算术异常）：

- 异常情况：当发生除零操作时抛出。

- 处理方法：在进行除法运算之前，对除数进行检查，避免出现除零操作。

5. IOException（输入输出异常）：

- 异常情况：当发生输入输出操作失败时抛出。

- 处理方法：使用 try-catch 块捕获异常，并处理或抛出异常。可以使用 Java 的异常处理机制和 IO 流操作来处理输入输出异常。

6. SQLException（数据库异常）：

- 异常情况：当数据库操作失败时抛出。

- 处理方法：在数据库操作时使用 try-catch 块捕获异常，及时释放资源、回滚事务并进行错误处理。

7. Custom Exceptions（自定义异常）：

- 异常情况：根据业务需求自定义的异常类型。

- 处理方法：定义自定义异常类，继承自 Exception 或 RuntimeException，使用合适的异常信息和错误码来传递异常信息。

在实际开发中，合理处理各种异常类型是提高软件质量和稳定性的重要环节。根据具体业务需求和场景，合理选择合适的异常处理方法，可以提高系统的可靠性和可维护性。

#### 使用堆栈跟踪和调试工具进行异常排查jstack、jstat等——

在 Java 开发中，常用的堆栈跟踪和调试工具有 jstack 和 jstat。下面介绍如何使用这些工具进行异常排查：

1. jstack：jstack 是 JDK 自带的一个命令行工具，用于生成 Java 进程的堆栈跟踪信息，可以用于定位死锁、线程异常等问题。

示例使用方法：

- 获取 Java 进程的进程号（PID）：`jps` 命令用于列出当前系统中所有 Java 进程的 PID。

- 使用 jstack 命令生成堆栈信息：`jstack <PID>` 命令可以生成指定 Java 进程的堆栈信息。例如：`jstack 1234`。

- 分析堆栈信息：分析堆栈信息，查看线程状态、死锁情况、线程占用情况等。

2. jstat：jstat 是用于监控 Java 进程内存、垃圾回收、类加载等信息的命令行工具。

示例使用方法：

- 获取 Java 进程的进程号（PID）：使用 `jps` 命令获取 Java 进程的 PID。

- 使用 jstat 命令查看内存使用情况：`jstat -gc <PID>` 命令可以查看 Java 进程的垃圾回收信息。例如：`jstat -gc 1234`。

- 分析 jstat 输出信息：通过 jstat 输出的信息，可以分析内存使用情况、垃圾回收情况等，帮助排查性能问题。

除了 jstack 和 jstat，还可以使用其他调试工具如 jconsole、VisualVM、Java Mission Control 等来辅助进行异常排查和性能优化。这些工具都可以帮助开发人员快速诊断和解决 Java 应用程序中的问题，提高系统的稳定性和性能。

### 中间件:

#### Nacos 的基本功能和使用场景——

Nacos（官方名称为 Nacos）是阿里巴巴开源的一个服务发现、配置中心和动态 DNS 服务框架。Nacos 提供了一套基于 RESTful API 的管理界面，支持多种语言，可扩展性强，具有以下基本功能和使用场景：

基本功能：

1. 服务发现：Nacos 可以注册和发现服务实例，实现微服务之间的调用和通信。

2. 配置管理：Nacos 提供配置中心的功能，可以动态管理应用程序的配置信息，支持实时更新配置。

3. 服务健康检查：Nacos 支持服务实例的健康检查机制，可以自动发现不健康的实例并进行下线处理。

4. 动态 DNS：Nacos 可以将服务实例以域名的方式暴露，支持基于服务名的负载均衡。

5. 集群管理：Nacos 支持集群部署和自动扩缩容，保证服务的高可用性和可伸缩性。

使用场景：

1. 微服务架构：Nacos 可以作为微服务架构的服务注册中心和配置中心，实现服务治理、服务发现和服务调用。

2. 集群管理：Nacos 可以用于管理集群中的节点信息、配置信息，以及实现集群中服务实例的动态发现和负载均衡。

3. 分布式系统：Nacos 可以作为分布式系统的配置中心，协调系统的配置信息，实时同步和更新配置。

4. 云原生应用：Nacos 可以用于部署和管理云原生应用，支持容器化部署和动态拓扑调整，满足云原生应用的需求。

5. 动态 DNS：Nacos 可以管理服务实例的域名信息，动态解析域名实现负载均衡和服务调用。

总之，Nacos 是一个功能强大且易于集成的服务发现和配置中心框架，可以帮助开发团队构建高可用、分布式、微服务化的系统。通过使用 Nacos，可以实现服务的自动注册、发现、配置管理和负载均衡，提高系统的灵活性和可靠性。

#### RabbitMQ 的基本功能和使用场景——

RabbitMQ 是一个开源的消息代理软件，实现了高级的消息队列协议 AMQP（高级消息队列协议），具有以下基本功能和使用场景：

基本功能：

1. 消息队列：RabbitMQ 提供了消息队列的功能，允许生产者将消息发送到队列，并允许消费者从队列中接收消息。

2. 发布/订阅模式：RabbitMQ 支持发布/订阅模式，一个生产者可以向多个消费者发送消息。

3. 消息路由：RabbitMQ 提供了灵活的消息路由机制，通过交换机和路由键来将消息路由到指定的队列。

4. 消息确认：RabbitMQ 支持消息确认机制，保证消息的可靠性传递。

5. 可靠性：RabbitMQ 具有高可靠性和高可用性，支持持久化队列和消息，保证消息不丢失。

使用场景：

1. 异步通信：RabbitMQ 可以作为生产者和消费者之间的异步通信中间件，实现解耦和提高系统响应速度。

2. 任务队列：RabbitMQ 可以用于实现任务队列，将耗时的任务放入队列，通过多个消费者并发处理任务，提高系统性能和吞吐量。

3. 微服务架构：RabbitMQ 可以作为微服务架构中服务间通信的消息中间件，实现服务之间的解耦和高可扩展性。

4. 日志收集：RabbitMQ 可以用于日志收集系统，将日志消息发送到队列中，并消费者处理日志记录。

5. 实时数据处理：RabbitMQ 可以用于实时数据处理和事件驱动架构，实现实时数据传输和处理。

总之，RabbitMQ 是一个功能强大、可靠性高的消息队列中间件，适合用于异步通信、任务队列、微服务架构、日志收集等场景下。通过使用 RabbitMQ，可以实现应用系统的解耦、异步通信、提高系统的可伸缩性和弹性，提升系统的性能和效率。

6.RabnitMQ如何保证不重复消费（幂等性）

保证消息等幂性；

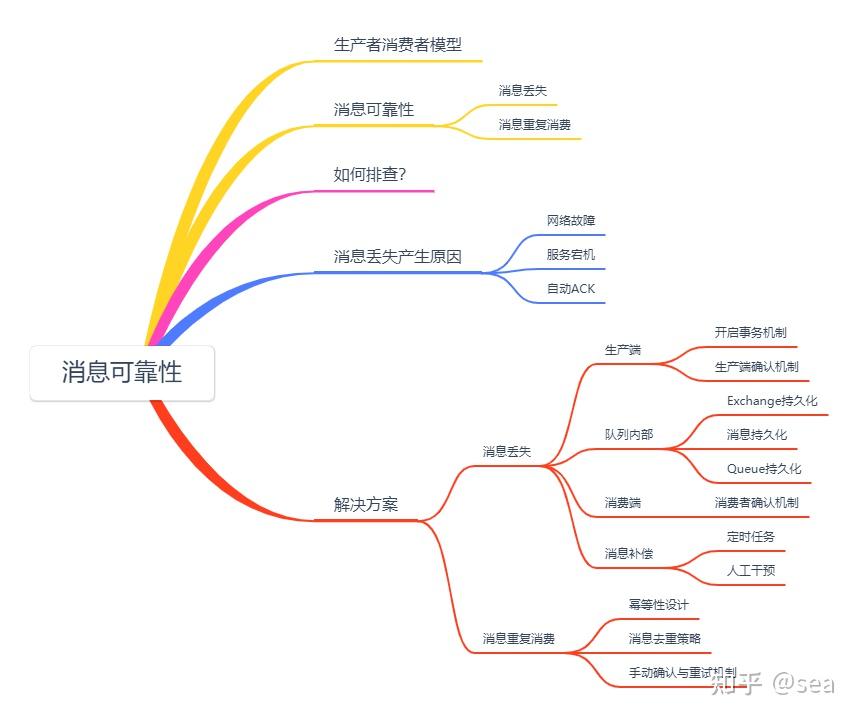
比如：在写入消息队列的数据做唯一标示，消费消息时，根据唯一标识判断是否消费过；

假设你有个系统，消费一条消息就往数据库里插入一条数据，要是你一个消息重复两次，你

不就插入了两条，这数据不就错了？但是你要是消费到第二次的时候，自己判断一下是否已

经消费过了，若是就直接扔了，这样不就保留了一条数据，从而保证了数据的正确性。

7.RabnitMQ消息可靠性（消息不丢失）



#### Kafka 的基本功能和使用场景——

Apache Kafka 是一个开源的分布式流处理平台，具有高吞吐量、低延迟等特点，主要用于构建实时数据管道和流处理应用。以下是 Kafka 的基本功能和使用场景：

基本功能：

1. 消息传递：Kafka 是一个分布式消息系统，可以存储和传递大量消息流。

2. 分布式存储：Kafka 以分布式存储的方式保存消息，支持消息持久化和高可靠性。

3. 发布/订阅模式：Kafka 支持发布/订阅模式，生产者向主题（topic）发送消息，消费者从主题订阅消息。

4. 消息流处理：Kafka 可以实现实时流处理，支持处理大规模数据流。

5. 数据复制和容错机制：Kafka 通过分区复制和分布式存储实现数据冗余和容错机制，保证消息不丢失。

使用场景：

1. 日志收集与处理：Kafka 可以用于实时的日志收集和处理，将不同来源的日志消息传输到 Kafka 集群，然后进行处理、分析等操作。

2. 流式处理：Kafka 适合用于构建实时数据管道，支持实时数据处理、流式计算等场景。

3. 事件驱动架构：Kafka 可以作为事件驱动的消息中间件，用于构建事件驱动架构的应用系统。

4. 实时数据分析：Kafka 支持实时数据流处理，可以与流处理框架（如 Apache Flink、Spark Streaming）结合，用于实时数据分析和处理。

5. 数据异步传输：Kafka 可以实现不同系统之间的数据异步传输、解耦，实现系统之间的松耦合。

总的来说，Kafka 是一个适用于构建实时数据管道和流处理应用的高性能消息系统，适用于日志收集、流式处理、事件驱动架构、实时数据分析等多种应用场景。通过使用 Kafka，可以构建高性能、可靠性高的实时数据处理系统，提升系统的可扩展性和性能。

#### Redis常问问题

1. redis集群模式。

哨兵和集群。哨兵使用在数据量不大，并发性不高。只要保证高可用的场景。

集群使用在数据量大，并发性高的场景。

1. redis cluster的hash slot算法

　　redis cluster有固定的16384个hash slot，对每个key计算CRC16值，然后对16384取模，可以获取key对应的hash slot

　　redis cluster中每个master都会持有部分slot，比如有3个master，那么可能每个master持有5000多个hash slot

　　hash slot让node的增加和移除很简单，增加一个master，就将其他master的hash slot移动部分过去，减少一个master，就将它的hash slot移动到其他master上去

　　移动hash slot的成本是非常低的

　　客户端的api，可以对指定的数据，让他们走同一个hash slot，通过hash tag来实现

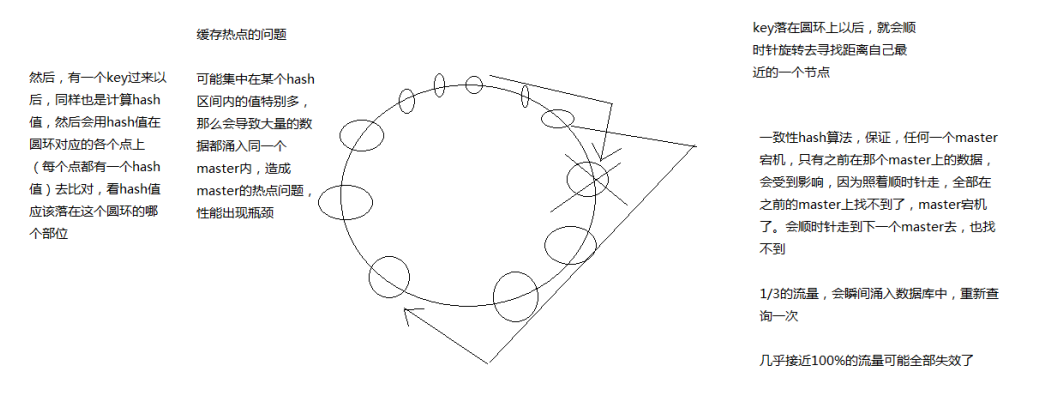
　　127.0.0.1:7000>CLUSTER ADDSLOTS 0 1 2 3 4 ... 5000 可以将槽0-5000指派给节点7000负责。

　　每个节点都会记录哪些槽指派给了自己，哪些槽指派给了其他节点。

　　客户端向节点发送键命令，节点要计算这个键属于哪个槽。

如果是自己负责这个槽，那么直接执行命令，如果不是，向客户端返回一个MOVED错误，指引客户端转向正确的节点。

1. 一致性hash算法



1. 用过或者了解Redis Zset数据结构吗？

有序集合zset保留了set集合不能有重复成员的特点，但与set集合不同的是，zset的每个member都有一个唯一的浮点数类型的分数score与之关联。依据每个member的score进行排序，有序集合中的元素是不能重复的，但分数允许重复；

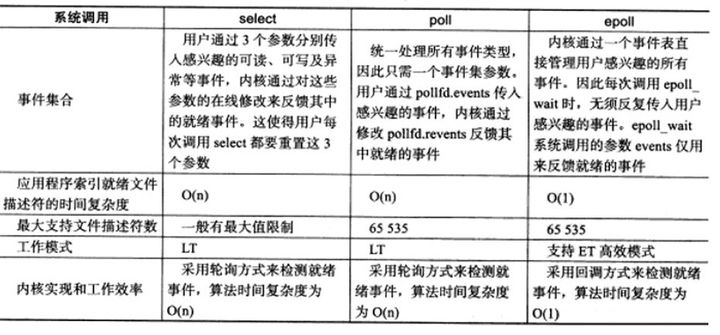
ZSET 的底层数据结构主要由两部分组成：跳跃表（Skip List）和哈希表（Hash Table）。

跳跃表（Skip List）：跳跃表是一种随机化数据结构，用于有序元素的快速查找和插入。它是通过在元素的层级之间建立索引来实现快速查找的。跳跃表中的每个节点都包含一个成员（Member）和一个分数（Score），以及若干个指向其他节点的指针，这些指针用于在不同层级之间快速跳跃。通过跳跃表，ZSET 可以实现基于分数的范围查询、快速的成员查找和插入操作。

哈希表（Hash Table）：除了跳跃表外，ZSET 还使用了一个哈希表，用于存储成员到分数的映射关系。在哈希表中，键是成员，值是分数。通过哈希表，可以快速

使用ZSET做延迟队列，或者 排行榜。

1. IO多路复用机制——select、poll、epoll的原理和区别



1. Redis集群写入/查询数据原理

redis集群数据存储原理：

在redis cluster中，如果想要存入一个key-value，

这个key首先会通过CRC16算法取余（和16384取余），

结果会对应上0-16383之间的哈希槽（hash slot）

最后，redis cluster会将key-value放置在对应的哈希槽中。

redis集群数据获取原理：

当client向redis cluster中的任意一个节点发送与数据库key有关的命令时，

接收命令的节点会计算出要处理的key属于哪个哈希槽（hash slot），

并且先检查这个hash slot是否属于自己（管辖）：

如果key所在的槽正好属于自己（管辖），节点会直接执行这个key相关命令。

如果key所在的槽不属于自己（管辖），那么节点会给client返回一个MOVED错误，

指引client转向负责对应槽的节点，并客户端需要再次发送想要执行的和key相关的命令。

### 数据库

#### MySQL 的优化——

MySQL 是一个流行的关系型数据库管理系统，为了提升数据库性能和效率，可以进行一系列的优化工作。以下是一些常见的 MySQL 优化方法：

1. 设计合适的数据库结构：

- 使用合适的数据类型，避免占用过多存储空间。

- 设计合适的索引，提升查询性能。

- 避免过度范式化，避免 JOIN 操作过多。

2. 优化查询语句：

- 使用合适的 WHERE 子句条件，尽可能减少查询返回的数据量。

- 尽量避免使用 SELECT \*，明确指定需要查询的字段。

- 避免在 WHERE 子句中对列进行计算，可以优先进行计算再进行查询。

3. 使用合适的存储引擎：

- InnoDB 是 MySQL 默认的存储引擎，适合读写频繁、事务处理的场景。

- MyISAM 适合读操作频繁的场景，不支持事务处理。

4. 配置合适的缓存：

- 使用查询缓存需要谨慎，可能会导致性能下降。

- 可以通过配置 query\_cache\_type、query\_cache\_size 来优化查询缓存。

5. 优化数据库参数配置：

- 根据数据库使用情况，调整相关参数如 innodb\_buffer\_pool\_size、innodb\_log\_file\_size 等。

- 合理配置连接数、线程池大小等，避免资源浪费和性能瓶颈。

6. 定期优化表：

- 使用 OPTIMIZE TABLE 命令来优化表，减少碎片，提升性能。

- 使用 ANALYZE TABLE 命令来更新表的统计信息，帮助优化查询计划。

7. 监控和调优：

- 使用 MySQL 自带的性能监控工具，如 EXPLAIN、SHOW STATUS 等，分析 SQL 语句执行情况。

- 使用性能监控工具如 MySQL Workbench、Percona Toolkit 等进行性能调优。

通过以上一系列的优化措施，可以显著提升 MySQL 数据库的性能和效率，同时也可以提升应用系统的整体性能和用户体验。

#### 熟悉常见的 NoSQL 数据库 (MongoDB, Redis, Elasticsearch)——

下面是关于常见的 NoSQL 数据库 MongoDB、Redis 和 Elasticsearch 的简要介绍：

1. MongoDB：

- MongoDB 是一个面向文档的 NoSQL 数据库，采用文档存储模型，数据以 BSON（一种类似 JSON 的二进制形式）格式存储。

- MongoDB 支持复制、分片和事务功能，适合于需要高性能、高可用性和弹性扩展的场景。

- MongoDB 可以存储复杂的结构化数据、非结构化数据，并支持灵活的查询语言，适合用于大数据、Web 应用、实时分析等场景。

2. Redis：

- Redis 是一个开源的内存数据库，也称为缓存数据库，支持多种数据类型（如字符串、哈希、列表、集合、有序集合）。

- Redis 可以将数据存储在内存中，提供快速的读写速度，适合于高并发读写、缓存、消息队列等场景。

- Redis 支持持久化存储、主从复制、数据过期、事务等功能，提供了丰富的命令集合，适合于构建实时应用、计数器、实时聊天等场景。

3. Elasticsearch：

- Elasticsearch 是一个开源的分布式搜索和分析引擎，基于 Apache Lucene 构建，具有高性能的全文搜索和分析功能。

- Elasticsearch 支持实时索引和搜索，可以对大规模数据进行快速检索、聚合和分析。

- Elasticsearch 提供了强大的搜索 API、聚合功能、地理信息搜索等特性，适合用于日志分析、文档检索、监控分析等场景。

以上所述是一些常见的 NoSQL 数据库 MongoDB、Redis 和 Elasticsearch 的简要介绍。它们各自具备不同的特性和适用场景，根据具体的业务需求和场景选择合适的 NoSQL 数据库可以帮助提升系统性能和扩展性。

#### Mysql索引

1. [Hash](https://so.csdn.net/so/search?q=Hash&spm=1001.2101.3001.7020" \t "_blank) 索引和 B+ 树索引有什么区别或者说优劣呢？

首先要知道 Hash 索引和 B+ 树索引的底层实现原理：

hash 索引底层就是 hash 表，进行查找时，调用一次 hash 函数就可以获取到相应的键值，之后进行回表查询获得实际数据。B+ 树底层实现是多路平衡查找树。对于每一次的查询都是从根节点出发，查找到叶子节点方可以获得所查键值，然后根据查询判断是否需要回表查询数据。

那么可以看出他们有以下的不同：

\* hash 索引进行等值查询更快(一般情况下)，但是却无法进行范围查询。

因为在 hash 索引中经过 hash 函数建立索引之后，索引的顺序与原顺序无法保持一致，不能支持范围查询。而 B+ 树的的所有节点皆遵循(左节点小于父节点，右节点大于父节点，多叉树也类似)，天然支持范围。

\* hash 索引不支持使用索引进行排序，原理同上。

\* hash 索引不支持模糊查询以及多列索引的最左前缀匹配。原理也是因为 hash 函数的不可预测。

\* hash索引任何时候都避免不了回表查询数据,而B+树在符合某些条件(聚簇索引,覆盖索引等)的时候可以只通过索引完成查询

\* hash 索引虽然在等值查询上较快，但是不稳定。性能不可预测，当某个键值存在大量重复的时候，发生 hash 碰撞，此时效率可能极差。而 B+ 树的查询效率比较稳定，对于所有的查询都是从根节点到叶子节点，且树的高度较低。

因此，在大多数情况下，直接选择 B+ 树索引可以获得稳定且较好的查询速度。而不需要使用 hash 索引。

### 高并发处理:

#### 高并发场景下的性能优化方法——

在高并发场景下，为了确保系统的稳定性和性能，需要进行一系列的性能优化措施。以下是一些常见的针对高并发场景的性能优化方法：

1. 缓存：使用缓存技术可以减轻数据库压力，提高系统的响应速度。可以使用内存缓存（如 Redis、Memcached）或者 CDN 等来缓存静态资源和热点数据。

2. 编程优化：优化代码逻辑，减少不必要的计算、循环和调用，尽量减少锁竞争。避免使用全局锁和同步块，使用乐观锁或 CAS（Compare and Swap）等机制。

3. 负载均衡：通过负载均衡技术将流量均匀分发到多台服务器上，提高系统的可扩展性和负载能力。常用的负载均衡器有 Nginx、HAProxy 等。

4. 高可用性架构：采用集群、分布式部署方式，提高系统的稳定性和可用性。通过主从复制、分区、副本等方式保证系统的高可用性。

5. 数据库优化：优化数据库查询语句，设计合适的索引，尽量减少数据库的访问和查询次数。可以采用数据库分片、数据分区、缓存等方式优化数据库性能。

6. 异步处理：将一些耗时的操作异步化，如日志记录、邮件发送等，减少线程等待时间，提高系统的并发处理能力。

7. 消息队列：使用消息队列系统来解耦系统各模块的耦合度，提高系统的处理能力和可靠性。常用的消息队列系统有 RabbitMQ、Kafka 等。

8. 系统监控与调优：实时监控系统的性能指标，通过监控工具（如 Prometheus、Grafana）分析系统负载、性能瓶颈，并进行针对性调优。

通过以上一系列性能优化措施，可以有效应对高并发场景下的挑战，提升系统的并发能力、性能和稳定性，保证系统能够在高负载下运行稳定并高效。

#### 常见的分布式缓存技术——

在分布式系统中，使用缓存可以显著提高系统的性能和吞吐量。常见的分布式缓存技术包括：

1. Redis：Redis 是一个高性能的开源分布式缓存数据库，支持多种数据结构（字符串、哈希表、列表、集合、有序集合），适合于高并发读写、数据存储和缓存。

2. Memcached：Memcached 是一个高性能的分布式内存对象缓存系统，适用于存储键值对数据，支持多节点部署和数据分片，提供快速的数据访问。

3. Hazelcast：Hazelcast 是一个开源的内存数据网格（In-Memory Data Grid），提供分布式缓存和计算功能，支持持久化、分布式部署和高可用性。

4. Apache Ignite：Apache Ignite 是一个内存计算平台，提供分布式缓存、计算和流处理功能，支持读写分离、事务等特性，适用于大数据处理和实时分析。

5. Couchbase：Couchbase 是一个面向文档的 NoSQL 数据库，具有分布式缓存功能，支持高性能、高可用性、水平扩展性，适合于低延迟的数据访问和存储。

6. Amazon ElastiCache：Amazon ElastiCache 是亚马逊提供的托管缓存服务，支持 Redis 和 Memcached，可以快速部署、扩展和管理分布式缓存服务。

7. GigaSpaces：GigaSpaces 是一个内存计算平台，提供分布式缓存和计算功能，支持数据分片、数据备份、事务等特性。

这些分布式缓存技术都有各自特点和适用场景，根据具体的业务需求和系统架构，选择合适的分布式缓存技术可以有效提高系统的性能、可伸缩性和可靠性。

#### 分布式锁——

分布式锁是一种用于在分布式系统中实现同步和互斥的机制，确保在不同节点上的多个进程或线程能够安全地访问共享资源。下面是一些常见的分布式锁实现方式：

1. 基于数据库的分布式锁：使用数据库的事务和唯一性约束来实现分布式锁。可以在数据库中创建一个特定的表，利用唯一约束（如主键或唯一索引）来确保同一时间只有一个节点获得锁。

2. 基于缓存的分布式锁：使用分布式缓存（如 Redis）来实现分布式锁。可以将锁信息存储在 Redis 的键值对中，并利用 Redis 的原子性操作（如 SETNX、SET EX）来实现互斥锁。

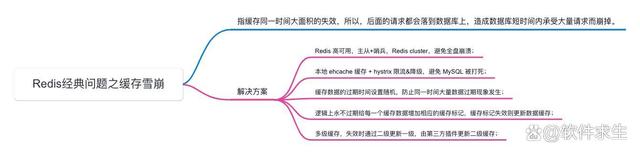
3. 基于分布式锁服务的分布式锁：使用专门的分布式锁服务（如 Zookeeper、etcd）来管理分布式锁。这些分布式锁服务提供了高可用、一致性和可靠性，并可用于分布式系统中的锁管理。

4. 基于乐观锁的分布式锁：使用版本号或时间戳等机制实现乐观锁，在更新数据时通过比较版本号来判断是否加锁成功。适用于并发量不高的场景。

5. 基于独立节点的协调锁：使用独立的节点来作为协调者（类似于选举器），协调节点负责管理和分发锁，其他节点通过与协调节点的通信来获取锁。

使用分布式锁可以帮助解决在分布式系统中的数据并发访问问题，避免数据竞争和脏数据的出现。但在使用分布式锁时需要注意避免死锁、锁超时等问题，同时需要考虑锁的粒度和性能影响。选择合适的分布式锁实现方式，并根据具体场景进行适当优化，可以有效保证系统的数据一致性和可靠性。

6. Redis 缓存雪崩（即 千万级访问量，缓存到redis，如何redis崩了，如何防止 数据都打到数据库上。）



### java基础

#### java线程池——

Java 线程池是用于管理和控制线程的工具，通过线程池可以实现线程的复用、线程的管理和调度，从而提高程序的性能和效率。在 Java 中，线程池主要由以下几个关键类和接口组成：

1. Executor 接口：是线程池的顶层接口，定义了一些提交任务的方法。

2. ExecutorService 接口：是 Executor 接口的子接口，提供了更丰富的功能，如提交任务、关闭线程池等。

3. Executors 类：是线程池的工厂类，提供了一系列静态方法用于创建不同类型的线程池。

4. ThreadPoolExecutor 类：是 ExecutorService 接口的实现类，也是 Java 标准库中默认的线程池实现。

常用的线程池参数包括：

1. corePoolSize：核心线程数，线程池中保持存活的线程数量。

2. maximumPoolSize：最大线程数，线程池中允许的最大线程数量。

3. keepAliveTime：线程空闲时间，当线程空闲时间超过这个值时，多余的线程会被销毁。

4. workQueue：任务队列，用于存储待执行的任务，常见的队列包括 ArrayBlockingQueue、LinkedBlockingQueue、SynchronousQueue 等。

常见的线程池类型包括：

1. FixedThreadPool：固定大小的线程池，核心线程数和最大线程数相等，适用于任务数量是一定的情况。

2. CachedThreadPool：可缓存的线程池，没有核心线程，线程数量根据需要动态调整，适用于处理大量临时性任务的情况。

3. ScheduledThreadPool：定时任务线程池，用于定时执行任务和周期性任务。

4. SingleThreadPool：单线程线程池，只有一个线程在执行任务，适用于需要按顺序执行任务的情况。

Java 线程池提供了一种灵活、高效的线程管理方式，可以有效控制线程的创建与销毁，减少线程的开销，提高系统的性能和稳定性。合理使用线程池，可以避免线程创建和销毁的开销，提高程序的性能并降低资源的消耗。

5.项目里如何创建线程池。（加分题。）

使用ThreadPoolExecutor对象的方法来使用线程池，因为这样能使人明确各个参数的具体含义。各个参数属性文件配置话。

面试过程中你能说出核心线程数的设置方法能大大提高的面试得分。

CPU密集型：核心线程数 = CPU核数 + 1

IO密集型：核心线程数 = CPU核数 \* 2 + 1

6. 线程池七个参数详解

从源码中可以看出，线程池的构造函数有7个参数，分别是corePoolSize、maximumPoolSize、keepAliveTime、unit、workQueue、threadFactory、handler。下面会对这7个参数一一解释。

一、corePoolSize 线程池核心线程大小

线程池中会维护一个最小的线程数量，即使这些线程处理空闲状态，他们也不会 被销毁，除非设置了allowCoreThreadTimeOut。这里的最小线程数量即是corePoolSize。

二、maximumPoolSize 线程池最大线程数量

一个任务被提交到线程池后，首先会缓存到工作队列（后面会介绍）中，如果工作队列满了，则会创建一个新线程，然后从工作队列中的取出一个任务交由新线程来处理，而将刚提交的任务放入工作队列。线程池不会无限制的去创建新线程，它会有一个最大线程数量的限制，这个数量即由maximunPoolSize来指定。

三、keepAliveTime 空闲线程存活时间

一个线程如果处于空闲状态，并且当前的线程数量大于corePoolSize，那么在指定时间后，这个空闲线程会被销毁，这里的指定时间由keepAliveTime来设定

四、unit 空间线程存活时间单位

keepAliveTime的计量单位

五、workQueue 工作队列

新任务被提交后，会先进入到此工作队列中，任务调度时再从队列中取出任务。jdk中提供了四种工作队列：

①ArrayBlockingQueue

基于数组的有界阻塞队列，按FIFO排序。新任务进来后，会放到该队列的队尾，有界的数组可以防止资源耗尽问题。当线程池中线程数量达到corePoolSize后，再有新任务进来，则会将任务放入该队列的队尾，等待被调度。如果队列已经是满的，则创建一个新线程，如果线程数量已经达到maxPoolSize，则会执行拒绝策略。

②LinkedBlockingQuene

基于链表的无界阻塞队列（其实最大容量为Interger.MAX），按照FIFO排序。由于该队列的近似无界性，当线程池中线程数量达到corePoolSize后，再有新任务进来，会一直存入该队列，而不会去创建新线程直到maxPoolSize，因此使用该工作队列时，参数maxPoolSize其实是不起作用的。

③SynchronousQuene

一个不缓存任务的阻塞队列，生产者放入一个任务必须等到消费者取出这个任务。也就是说新任务进来时，不会缓存，而是直接被调度执行该任务，如果没有可用线程，则创建新线程，如果线程数量达到maxPoolSize，则执行拒绝策略。

④PriorityBlockingQueue

具有优先级的无界阻塞队列，优先级通过参数Comparator实现。

六、threadFactory 线程工厂

创建一个新线程时使用的工厂，可以用来设定线程名、是否为daemon线程等等

七、handler 拒绝策略

当工作队列中的任务已到达最大限制，并且线程池中的线程数量也达到最大限制，这时如果有新任务提交进来，该如何处理呢。这里的拒绝策略，就是解决这个问题的，jdk中提供了4中拒绝策略：

①CallerRunsPolicy

该策略下，在调用者线程中直接执行被拒绝任务的run方法，除非线程池已经shutdown，则直接抛弃任务。

②AbortPolicy

该策略下，直接丢弃任务，并抛出RejectedExecutionException异常。

③DiscardPolicy

该策略下，直接丢弃任务，什么都不做。

④DiscardOldestPolicy

该策略下，抛弃进入队列最早的那个任务，然后尝试把这次拒绝的任务放入队列

#### JVM

* 有哪些垃圾回收算法

复制算法

标记--复制算法就是把Java堆分成两块，每次垃圾回收时只使用其中一块，然后把存活的对象全部移动到另一块区域。

标记--整理算法

标记--整理算法算是一种折中的垃圾收集算法，在对象标记的过程，和前面两个执行的是一样步骤。但是，进行标记之后，存活的对象会移动到堆的一端，然后直接清理存活对象以外的区域就可以了。这样，既避免了内存碎片，也不存在堆空间浪费的说法了。但是，每次进行垃圾回收的时候，都要暂停所有的用户线程，特别是对老年代的对象回收，则需要更长的回收时间，这对用户体验是非常不好的。

标记--清除算法

见名知义，标记--清除算法就是对无效的对象进行标记，然后清除。

* 都有哪些垃圾回收器



CPU单核，那么毫无疑问Serial 垃圾收集器是你唯一的选择。

CPU多核，关注吞吐量 ，那么选择PS+PO组合。

CPU多核，关注用户停顿时间，JDK版本1.6或者1.7，那么选择CMS。

CPU多核，关注用户停顿时间，JDK1.8及以上，JVM可用内存6G以上，那么选择G1。

#### 并发锁

1. AQS了解过吗

Java的AQS是JDK自带的锁机制，是JUC（java.util.concurrent）的基础，很多并发类都是基于它实现的，例如：ReentrantLock、CountDownLatch、Semaphore、ReadWriteLock，CyclicBarrier。

AQS：全名为AbstractQuenedSynchronizer，是抽象的队列式同步器。AQS是除了synchronized关键字之外的锁机制，全限定类名为：java.util.concurrent.locks.AbstractQueuedSynchronizer.java

**AQS的原理**

        AQS原理：基于CLH队列，用volatile修饰共享变量state，线程通过CAS去改变状态，成功则获取锁成功，失败则进入等待队列，等待被唤醒。

        CLH（Craig，Landin，and Hagersten）队列是一个虚拟的双向队列，虚拟的双向队列即不存在队列实例，仅存在节点之间的关联关系。**AQS将每一条请求共享资源的线程封装成一个CLH锁队列的一个结点（Node），来实现锁的分配。**

### 多线程

#### 多线程在具体项目中的应用场景

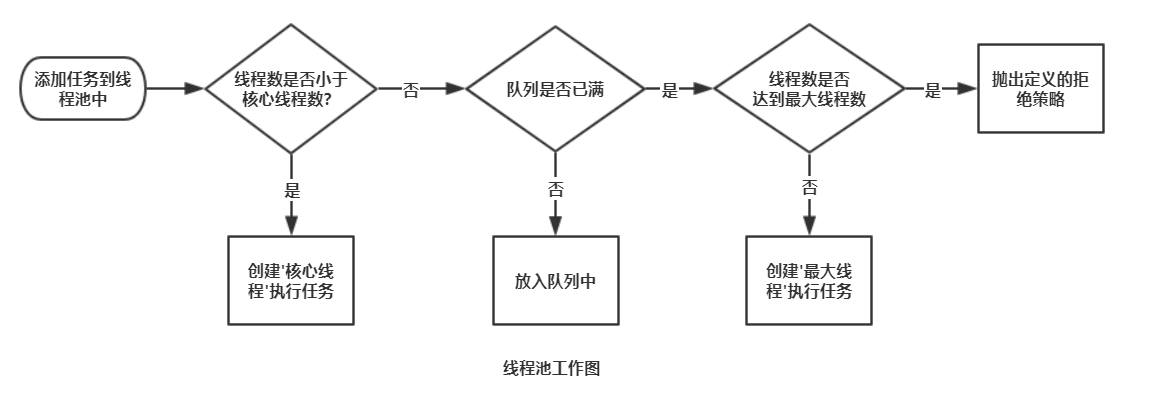
根据自己项目经历，举一个场景来说明。

举例：数据文件导入数据库中，优化性能，可以每5000条开一个线程去处理。

实现的效果：充分理由CPU多核能力。缩减数据入库速度。

#### 线程池什么时候会进行拒绝策略

当工作队列中的任务已到达最大限制，并且线程池中的线程数量也达到最大限制，这时如果有新任务提交进来，就会执行拒绝策略。默认拒绝策略为AbortPolicy。即：不执行此任务，而且抛出一个运行时异常。



### 云原生

1. Docker File都有哪些关键字及含义



https://blog.csdn.net/qq\_43456605/article/details/139510173

1. 云原生的三个可观测性都是哪些？解决方案是什么

可观测性的三大支柱：Metrics（指标）、Trace（追踪）、Log (日志)

**Metrics（指标）---Prometheus**

Metrics 是 Counter（计数器）、Gauge（仪表）和 Histogram（直方图），它们被快速、实时地分析聚合后，用于仪表板显示和告警。

系统层面的 Metrics：CPU、内存、I/O 等信息，一般由运维人员关注

应用层面的 Metrics：调用量、出错率、请求延迟等，一般由开发人员关注

业务层 Metrics ：转化率、下单率，一般由产品与运营人员关注

**Log（日志）-ELK**

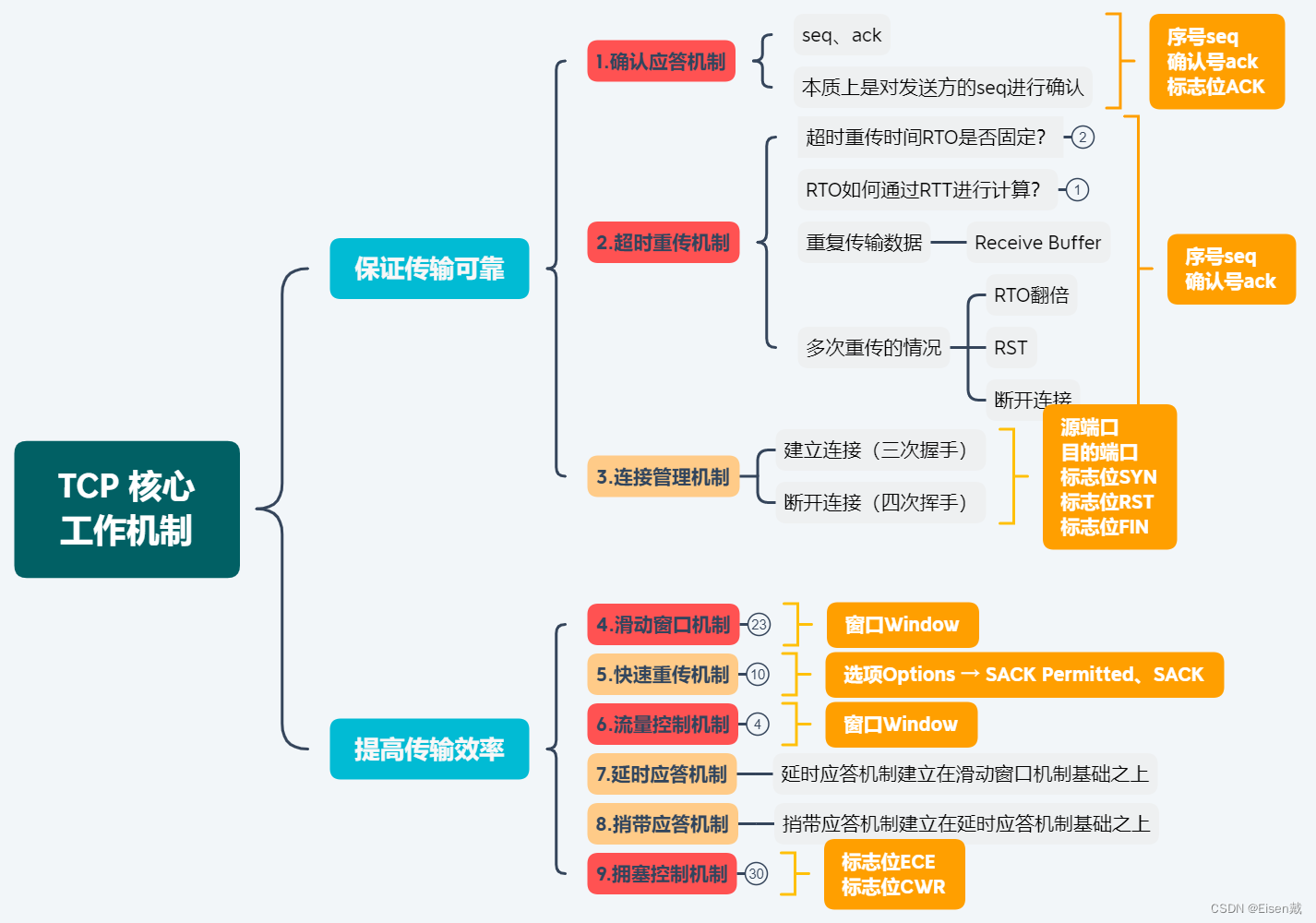
Log 是离散的事件，用于事后的分析、报告和调试；高质量的日志一定是结构化的，可以在创建之后被灵活的编排操控。

**Trace（追踪）-Skywalking**

Trace 是请求链路中的全部性能数据，尤其是在分布式系统中，Trace 链路必须是全的，正确、完整地实施跟踪的成本非常高，并且只有在分布式系统规模很大时才有意义。

### 网络编程

TCP 核心工作机制



### 接口防重放，防串改如果做

防篡改

数据加签

为了解决这类问题，前端可以给数据加上一个签名，比如采用MD5最简单的签名方式，可以把请求参数当作待签名的数据，计算一个MD5值，然后传给服务端，服务端同样用参数生成MD5值进行比对，发现不一样，那说明数据被篡改过了

防重放

为了防止接口重放，我们给每一个请求加上一个时间戳，但是这样并没有完全地解决防重放问题，有效期内还是可以多次调用。

所以为了解决这一问题呢，我们可以在每一个请求上，再加上一个随机串nonce，客户端每个请求的nonce我都记录下来，那么下一次再次请求的时候，就先根据nonce查一下，有没有请求过，请求过就丢弃，没有就正常处理。

12，分布式使用redis实现（常用）

* 使用setnx+expire命令实现

setnx是set if not exists 的缩写,也就是只有不存在的时候才设置rediskey 设置成功时返回 1 , 设置失败时返回 0

缺点：

加锁操作和设置超时时间是分开的。假设在执行完setnx加锁后，正要执行expire设置过期时间时，出现问题，则锁永远无法释放。所以需要使用lua脚本来使setnx+expire成为原子操作

如果超时时间过短，会出现业务还未结束，锁就被释放的情况。

Redisson就是使用watch dog解决了**锁过期释放，业务没执行完问题**

当线程加锁成功后，会启动一个后台线程，会每隔10秒检查一下，还持有锁，那么就会不断的延长锁key的30秒生存时间

* 使用zookeeper实现

Zookeeper的节点Znode有四种类型：

持久节点：默认的节点类型。创建节点的客户端与zookeeper断开连接后，该节点依旧存在。

持久顺序节点：所谓顺序节点，就是在创建节点时，Zookeeper根据创建的时间顺序给该节点名称进行编号，持久节点顺序节点就是有顺序的持久节点。

临时节点：和持久节点相反，当创建节点的客户端与zookeeper断开连接后，临时节点会被删除。

临时顺序节点：有顺序的临时节点（用于实现分布式锁）。

zookeeper分布式锁原理步骤：

zookeeper首先创建一个/lock节点

当有节点获取锁时，先为这个节点创建临时节点，例如lock-702564158761685-000001，序列号按创建顺序递增。

zookeeper会检查 lock-702564158761685-000001 是否/lock下的最小节点，如果是该节点得到锁，否则监听 lock-702564158761685-000001 前一个节点状态

当前一个节点的释放锁时会通知（唤醒）后一个节点，（这样只监听前一个节点的方式，避免了线程“惊群效应”）。锁如何实现

### 数据sql调优

1. SQL优化

第一步查看慢SQL日志。

show variables like '%slow\_query\_log\_file%';

使用mysqldumpslow命令分析慢SQL日志

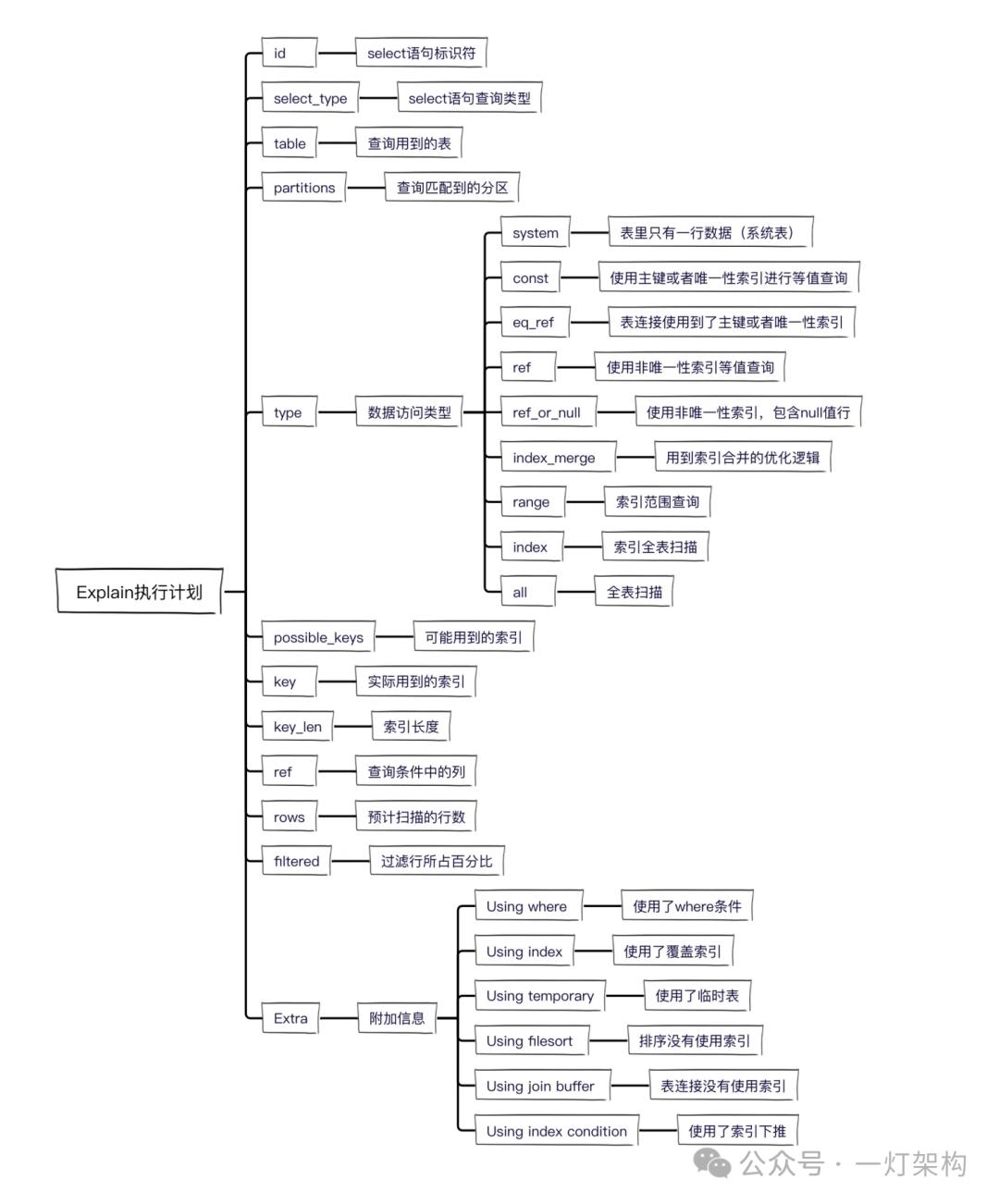
mysqldumpslow -s r -t 10 /usr/local/mysql/data/localhost\_slow.log

查询耗时最长的10条SQL：

mysqldumpslow -s t -t 10 /usr/local/mysql/data/localhost\_slow.log

第二步优化慢SQL

explain执行计划



### Cap理论

cap定理（分布式系统的三个指标）：

Consistency(一致性)：用户访问分布式系统中的任意节点，得到的数据必须一致。

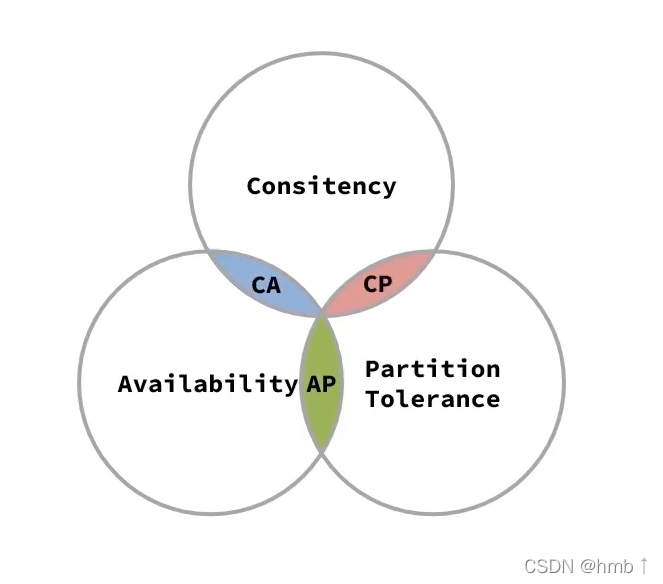
Avallabiry(可用性)：用户访问集群中的任意健康节点，必须能得到响应，而不是超时或拒绝。

Partition tolerance(分区容错性)：

分区：因为网络故障或其他原因导致分布式系统中的部分节点与其他节点失去连接，形成独立分区。

容错：在集群出现分区时，整个系统也要持续对外提供服务。

分布式系统无法同时满足这三个指标,这个结论叫做CAP定理。



2、BASE理论

BASE理论是对CAP的一种解决思路，包含三个思想：

Basically Available(基本可用):分布式系统在出现故障时，允许损失部分的可用性，即保证核心可用。

Soft State(软状态)：在一定时间内，允许出现中间状态，比如临时的不一致状态。

Eventually Consistent(最终一致性)：虽然无法保证一致性，但是在软状态结束后，最终达到数据一致。

而分布式事务最大的问题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：

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。