1. 分布式ID的生成策略

- UUID：UUID是一种常见的本地生成ID的方法，它有着全球唯一的特性。但UUID过长，往往用字符串表示，作为主键建立索引查询效率低
- 数据库自增ID：基于数据库的auto_increment自增ID完全可以充当分布式ID，但访问量激增时MySQL本身就是系统的瓶颈，用它来实现分布式服务风险比较大
  数据库多主模式：对上述数据库自增ID方式做一些高可用优化，换成主从模式集群。但如果集群后的性能还是扛不住高并发，就需要进行MySQL扩容增加节点
- Redis：Redis也同样可以实现，原理就是利用redis的incr命令实现ID的原子性自增
- Twitter的Snowflake算法：Snowflake是Twitter开源的分布式ID生成算法，其核心思想为，一个long型的ID：41 bit作为毫秒数，10 bit作为机器编号，12 bit作为毫秒内序列号

2. 线程池有哪些问题?

- **资源消耗**：线程池需要维护一组线程，这些线程在空闲时仍然占用系统资源。如果线程池中的线程数量过多，可能会导致内存和CPU资源的浪费。
- **线程泄漏**：如果线程池中的线程没有正确地释放，可能会导致线程泄漏。这会增加系统的负担并降低性能。
- **死锁**：线程池中的任务可能会相互等待，导致死锁。例如，如果一个任务等待另一个任务完成，而后者又在等待第一个任务完成，就会出现死锁。
- **拒绝策略不当**：线程池满载时，如果没有合适的拒绝策略，可能会导致任务被丢弃或系统崩溃。
- **任务执行时间不均衡**：如果线程池中的某些任务执行时间过长，可能会影响其他任务的执行速度。
- **线程安全问题**：线程池中的共享资源可能会出现竞争条件，需要谨慎处理同步和互斥。

3. Spring Security的优缺点

优点：

- 易于使用：Spring Security 提供了许多现成的功能和配置选项，使其易于集成到应用程序中。
- Java 集成：作为 Java 生态系统的一部分，Spring Security与其他 Spring 框架（如 Spring Boot、Spring MVC）无缝集成。
- 灵活性：Spring Security 可以轻松扩展以满足自定义需求。您可以自定义身份验证和授权逻辑。
- 全面的 Servlet API 集成：Spring Security 可以与 Servlet API 集成，包括基于 URL 的安全性、会话管理等。
- 防御攻击：Spring Security 提供了保护机制，防止跨站点请求伪造（CSRF）、跨站点脚本（XSS）等攻击。

缺点：

- 文档不足：Spring Security 的文档可能不足以处理某些安全威胁，例如跨站点脚本（XSS）。开发人员需要额外的资源来了解和应用最佳实践。
- 复杂性：尽管 Spring Security 提供了许多功能，但它的配置和使用可能会变得复杂，特别是对于初学者。
- 学习曲线：对于没有经验的开发人员，学习 Spring Security 可能需要一些时间。

4. java8新特性

- **Lambda 表达式**：Lambda 允许将函数作为方法的参数传递，使代码更简洁、紧凑。
- **函数式接口**：函数式接口指的是只有一个抽象方法的接口，可以隐式转换为 Lambda 表达式。
- **方法引用**：方法引用允许直接引用已有 Java 类或对象的方法或构造器，使代码更简洁。
- **默认方法**：接口中可以有默认实现的方法，方便接口的扩展。
- **Stream API**：Stream API 引入了函数式编程风格，用于处理集合数据。

5. 用于实现功能扩展的设计模式

- **装饰者模式**：装饰者模式允许我们在运行时动态地为对象添加新的行为。通过将装饰器包裹在原始对象周围，我们可以逐步添加功能，而不影响原始类的结构。这对于实现功能扩展非常有用
- **策略模式**：策略模式允许我们定义一系列算法，并将其封装成独立的类。通过在运行时选择不同的策略，我们可以实现不同的功能扩展。例如，根据不同的需求选择不同的排序算法
- **观察者模式**：观察者模式用于实现对象之间的一对多依赖关系。当一个对象的状态发生变化时，所有依赖于它的对象都会得到通知。这对于实现功能扩展和事件处理非常有用

6. Mysql的分布式事务

> 在MySQL数据库中，分布式事务是一种用于管理多个数据库之间的数据一致性和事务处理的技术。它主要通过两种方式实现：**两阶段提交**（Two-Phase Commit，2PC）和**基于消息的事务性**（X/Open XA）

- **两阶段提交**：这是一种基于协调者和参与者的分布式事务处理协议。在第一阶段，协调者会向所有参与者发送准备命令，参与者在接收到准备命令后，会执行事务操作，并将操作结果保存在本地，然后向协调者报告准备就绪。在第二阶段，如果协调者收到所有参与者的准备就绪消息，那么它会向所有参与者发送提交命令，参与者在接收到提交命令后，会提交事务，并释放在第一阶段中锁定的资源
- **基于消息的事务性**：这是一种通过消息队列来实现分布式事务的方法。在这种方法中，一个服务会在本地数据库中执行事务操作，并将操作结果发送到消息队列中。其他服务可以从消息队列中读取

消息，并根据消息内容在本地数据库中执行相应的事务操作。这种方法的优点是可以确保所有服务的数据一致性，缺点是需要维护一个消息队列，增加了系统的复杂性。

7. 一致性哈希

一致性哈希（Consistent Hashing）是一种特殊的哈希算法，用于解决分布式系统中的数据分片问题。它的主要目标是在系统扩容和缩容时，最小化数据迁移量，从而减轻系统负载，保证系统的高可用性。

一致性哈希的基本思想是将哈希值域组成一个虚拟的环（也称为哈希环），然后将数据和节点（机器）映射到这个环上。当需要查找某个键（key）对应的值（value）时，可以沿着环的方向查找，直到遇到的第一个节点，该节点就是该键对应的值所在的位置。

一致性哈希的一个重要特性是，当增加或删除节点时，只需要对环上的一小部分数据进行重新映射，而不需要对所有数据进行重新映射。这就大大减少了在节点动态变化时所需的数据迁移量，从而提高了系统的稳定性和效率

8. 如何实现全局异常处理器

在Spring Boot中，全局异常处理可以通过使用 `@ControllerAdvice` 注解来实现。这个注解可以让我们在一个独立的类中定义全局的异常处理方法

`@ExceptionHandler(value = Exception.class)` 注解表示这个方法用于处理所有类型的异常。当发生异常时，Spring会自动调用这个方法，并将异常对象作为参数传入，然后我们可以在这个方法中处理异常

9. 线程池的拒绝策略

- **AbortPolicy**：这是线程池的默认拒绝策略。当任务被拒绝时，AbortPolicy会直接抛出一个类型为RejectedExecutionException的RuntimeException
- DiscardPolicy：这种拒绝策略会直接丢弃被拒绝的任务，但是不会抛出异常
- DiscardOldestPolicy：这种策略会丢弃任务队列中最旧的任务（即最先进入队列的，最久未处理的任务），然后尝试重新提交新的任务
- CallerRunsPolicy：这种策略下，如果线程池未关闭，对于新提交的任务，它是由调用者所在的线程来处理

10. Spring Security的实现原理

11. 常见异常类

- ArithmeticException：在数值计算过程中发生的异常，例如除零错误
- NullPointerException：当应用程序试图在需要对象的地方使用 null 时引发的异常
- ArrayIndexOutOfBoundsException：当对数组的索引值为负数或大于等于数组大小时抛出
- ClassCastException：当一个不是该类的实例转换成这个类就会抛出这个异常
- ClassNotFoundException：未能找到指定的类，比如使用 Class.forName () 时指定的类名不正确
- IOException：输入输出操作异常，比如文件读写错误
- SQLException：SQL 操作异常，比如连接数据库错误
- FileNotFoundException：找不到文件异常，如果文件不存在就会抛出这种异常
- NoSuchMethodException：方法不存在异常，当应用试图调用某类的某个方法，而该类的定义中没有该方法的定义时抛出该错误

12. 分治法 vs 动态规划

相同点：两者都要求原问题具有最优子结构性质，都是将原问题分而治之，分解成若干个规模较小的子问题，然后将子问题的解合并，形成原问题的解

不同点：

- 分治法将分解后的子问题看成相互独立的，通过用递归实现

- 动态规划将分解后的子问题理解为相互间有联系，有重叠部分，需要记忆，通常用迭代实现

13. 消息队列的使用场景

- **异步处理**：例如，用户注册后，需要发送注册邮件和注册短信。传统的做法是将注册信息写入数据库成功后，发送注册邮件，再发送注册短信。使用了消息队列，可以将这些非必须的业务逻辑异步处理，提高系统响应速度
- **应用解耦**：例如，用户下单后，订单系统需要通知库存系统。传统的做法是，订单系统调用库存系统的接口。引入消息队列后，可以实现订单系统与库存系统的应用解耦，提高系统稳定性
- **流量削峰**：例如，在秒杀或团抢活动中，由于流量过大，导致流量暴增，应用挂掉。引入消息队列，可以控制活动的人数，缓解短时间内高流量压垮应用
- **日志处理**：例如，分布式系统的日志数据庞大且分散，消息队列可用于集中日志信息。各服务将日志发送到队列，专门的日志收集服务负责从队列中读取并进行存储、分析和报警
- **消息通讯**：例如，消息队列一般都内置了高效的通信机制，因此也可以用在纯的消息通讯。比如实现点对点消息队列，或者聊天室等

13. 如何保证消息队列的有序性

消息队列的有序性是指消息在生产者发送到消费者的过程中，消息的顺序保持不变。以下是一些常见的方法来保证消息队列的有序性：

1. **单一队列**：将所有相关消息都发送到同一个队列中，并且确保消费者按顺序处理这些消息。这种方式简单直接，但可能在高负载情况下成为瓶颈
2. **创建多个队列**：对于RabbitMQ，我们可以给RabbitMQ创建多个queue，每个消费者固定消费一个queue的消息，生产者发送消息的时候，同一个订单号的消息发送到同一个queue中，由于同一个queue的消息是一定会保证有序的，那么同一个订单号的消息就只会被一个消费者顺序消费，从而保证了消息的顺序性

14. 使用线程池需要考虑的因素

- 任务的性质：CPU密集型任务，IO密集型任务，混合型任务。不同类型的任务对线程池的使用和配置有不同的影响。
- 任务的优先级：某些任务可能比其他任务更重要，可能需要优先执行。
- 任务的执行时间：一些任务可能需要很长时间才能完成，而其他任务可能会很快完成。这可能会影响到如何配置线程池，以及如何调度任务。
- 任务的依赖性：一些任务可能需要等待其他任务完成才能开始。

- 资源限制：应用程序可用的最大线程数，以及系统的总体负载，都可能影响到线程池的大小和配置。
- 线程池的管理：包括创建线程的策略、线程如何结束、线程如何排队等。
- 结果处理：你需要考虑如何处理线程执行的结果，是直接在子线程中处理还是传递到主线程中处理。
- 错误处理：你需要考虑如何处理线程执行过程中的错误。

> **CPU密集型**（原生线程池实现）：当工作线程数大于核心线程数，小于最大线程数时，对于后续的任务是先存放到工作队列中，等到工作队列满了之后才创建新线程去执行
> **IO密集型**（Tomcat，Dubbo中主要用于执行网络的IO任务）：当工作线程数大于核心线程数，小于最大线程数时，对于后续的任务是先创建新线程直接执行，等到工作线程数等于最大线程数后，才会将后续的任务存放到队列中

15. 项目中遇到了哪些问题?

> 围绕分布式任务：执行视频转码任务，进行展开

- **分布式锁**解决因为网络波动造成一些机器下线，任务被重复处理
- 使用**Executors**实现线程池时遇到的问题？工作队列最大上限是Integer.MAX_VALUE,最大线程数是Integer.MAX_VALUE，引发**OOM**
- 使用**CountDownLatch**来允许一个或多个线程等待其他线程完成操作之后再关闭线程池，但无法解决由于spring关闭造成任务丢失问题
- 使用**ThreadPoolTaskExecutor**实现线程池：在Spring容器关闭时会调用shutdown方法，这个方法会等待所有已提交（submit与execute）的任务（包括正在执行的和队列中等待的）执行完毕，然后再关闭线程池，这就确保了没有任务会在Spring容器关闭时丢失。

16. 如何解决慢查询

- 优化SQL语句：根据慢查询日志定位需要优化的SQL语句，然后使用EXPLAIN等工具分析SQL执行计划，修改SQL或者尽量让SQL走索引
- 索引优化：慢查询非常多的情况是由于SQL没有走适当的索引导致的。但是，即使加上索引，如果使用不正确，SQL语句在执行时依然不会走索引
- 分解关联查询：将一个大的查询分解为多个小查询是很有必要的。很多高性能的应用都会对关联查询进行分解，就是可以对每一个表进行一次单表查询，然后将查询结果在应用程序中进行关联
- 使用视图：将复杂查询封装为视图，以简化查询过程

17. 保证消息队列的消息不被重复消费

在消息队列系统中，确保消息不被重复消费通常需要以下几个步骤：

- **消息确认**：当消费者成功处理一个消息后，它需要**向消息队列发送一个确认信号**。只有收到确认信号后，消息队列才会将该消息从队列中移除
- **消息标识**：为每个消息分配一个全局唯一的ID。消费者在处理消息前，先检查这个消息ID是否已经被处理过。如果已经处理过，那么就跳过这个消息。这需要**消费者能够持久化已处理的消息ID**，例如保存在数据库或Redis中
- 幂等操作：设计你的系统使得对同一个消息的多次处理具有相同的效果。这样，即使一个消息被重复消费，也不会对系统的状态产生影响
- **事务支持**：一些消息队列系统，如Kafka，支持事务，可以确保**消息的生产和消费都在同一个事务中完成**，从而避免消息的重复消费

18. 为什么Redis Cluster没有采用一致性哈希

- 一致性哈希的节点分布基于圆环，无法很好的**手动设置数据分布**，比如有些节点的硬件差，希望少存一点数据，这种很难操作。而哈希槽可以很**灵活的配置每个节点占用哈希槽的数量**
- 一致性哈希的某个节点宕机或者掉线后，当该机器上原本缓存的数据被请求时，会从数据源重新获取数据，并将数据添加到失效机器后面的机器，这个过程被称为 "缓存抖动" ，而使用哈希槽的节点宕机，会导致一定范围内的槽不可用，只能通过主从复制加哨兵模式保证高可用。
- 真是基于一致性哈希的特点，当某台机器宕机时，极易引起雪崩，如上述介绍中删除节点。
- 相对于哈希槽，一致性哈希算法更复杂

19. Redission的分布式锁原理?

> 依靠lua脚本和看门狗机制实现，生成一个hash结构，而不是String

```
if (redis.call('exists', KEYS[1]) == 0) then " +
   "redis.call('hincrby', KEYS[1], ARGV[2], 1); " +
   "redis.call('pexpire', KEYS[1], ARGV[1]); " +
   "return nil; " +
   "end; " +
"if (redis.call('hexists', KEYS[1], ARGV[2]) == 1) then " +
    "redis.call('hincrby', KEYS[1], ARGV[2], 1); " +
    "redis.call('pexpire', KEYS[1], ARGV[1]); " +
    "return nil; " +
    "end; " +
"return redis.call('pttl', KEYS[1]);"
```

Redission的分布式锁原理

20. 分布式事务解决方案

Mysql分布式事务解决方案

Seata的四种模式

21. ConcurrentHashMap的实现原理?

ConcurrentHashMap的实现原理