【笔面复盘】

2025.03.17虾皮1.5h（a2）

2025.03.15美团1.5h（a1.5）

* 原型模式（创建型）  
  组合模式（结构型）装饰器模式（结构型）  
  模板方法模式（行为型）

设计模式可以分为三种类型：**创建型**、**结构型**和**行为型**。

**创建型设计模式**

创建型设计模式主要关注如何实例化一个对象，强调对象创建的过程。

**创建型设计模式包括：**

1. **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。
2. **工厂方法模式（Factory Method）**：定义一个创建对象的接口，让子类决定实例化哪一个类。工厂方法模式让类的实例化延迟到子类。
3. **抽象工厂模式（Abstract Factory）**：提供一个创建一系列相关或互相依赖的对象的接口，而无需指定它们具体的类。
4. **建造者模式（Builder）**：将一个复杂对象的构建过程抽象化，可以一步步地构建一个复杂的对象，而不需要暴露它的内部构建过程。
5. **原型模式（Prototype）**：通过复制现有的实例来创建新对象，而不是通过构造函数进行创建。适用于对象的创建成本较高或复杂的情况。

**结构型设计模式**

结构型设计模式关注如何将类或对象组合成更大的结构，目的是通过合理的组合来提高灵活性和可扩展性。

**结构型设计模式包括：**

1. **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，使得原本接口不兼容的类可以一起工作。
2. **桥接模式（Bridge）**：将抽象部分与实现部分分离，使得两者可以独立变化。
3. **组合模式（Composite）**：将对象组合成树形结构来表示“部分-整体”的层次结构，使得客户对单个对象和组合对象的使用具有一致性。
4. **装饰器模式（Decorator）**：动态地给一个对象添加一些额外的职责，而不影响其他对象。
5. **外观模式（Facade）**：为子系统中的一组接口提供一个统一的高层接口，使得子系统更容易使用。
6. **享元模式（Flyweight）**：运用共享技术有效地支持大量细粒度对象的共享，从而降低系统的内存消耗。
7. **代理模式（Proxy）**：为其他对象提供一种代理以控制对这个对象的访问。

**行为型设计模式**

行为型设计模式主要关注对象之间的通信、交互方式和职责分配。它们帮助定义对象的职责以及如何在对象之间传递信息。行为型模式通常处理如何协调对象之间的交互和行为。

**行为型设计模式包括：**

1. **责任链模式（Chain of Responsibility）**：将多个处理对象连成一条链，传递请求，直到链中的某个对象处理它。
2. **命令模式（Command）**：将请求封装为一个对象，从而让用户使用不同的请求、队列或日志请求，以及支持可撤销的操作。
3. **解释器模式（Interpreter）**：为语言中的句子创建解释器，该模式提供了一个解释句法的方式。
4. **迭代器模式（Iterator）**：提供一种方式来顺序访问一个集合中的元素，而不暴露集合的内部结构。
5. **中介者模式（Mediator）**：用一个中介者对象来封装一组对象的交互，使得对象之间不需要显式地引用彼此，从而降低耦合度。
6. **备忘录模式（Memento）**：在不破坏封装性的前提下，捕获一个对象的内部状态，并在对象之外保存这个状态，以便以后恢复。
7. **观察者模式（Observer）**：定义对象间的一对多依赖关系，当一个对象改变状态时，所有依赖于它的对象都会得到通知并自动更新。
8. **状态模式（State）**：允许一个对象在其内部状态改变时改变其行为，对象看起来像是改变了其类。
9. **策略模式（Strategy）**：定义一系列算法，并将每一个算法封装起来，让它们可以互换。
10. **模板方法模式（Template Method）**：定义一个操作中的算法框架，而将一些步骤的执行延迟到子类中。模板方法使得子类可以重新定义某些特定步骤而不改变算法的结构。
11. **访问者模式（Visitor）**：表示一个作用于某对象结构中的各元素的操作，它使你可以在不改变元素类的前提下定义作用于这些元素的新操作。

* 哪种垃圾回收方式按照时间分割任务、垃圾回收和增变者交错？  
  部分回收、世代垃圾回收、增量回收

**分代垃圾回收**中的**增量垃圾回收**

**增量垃圾回收**

增量垃圾回收是一种将垃圾回收任务分割成小块，并按时间片或增量的方式逐步执行的回收方式。它通常与分代垃圾回收结合使用，将垃圾回收过程切分为多个小步骤，以减少暂停时间。其主要特点是：

* **任务分割**：垃圾回收操作被划分为多个小的任务块，并在不同的时间进行执行。
* **增量式执行**：每次执行垃圾回收时，不是一次性完成所有的回收工作，而是分阶段逐步完成。
* **交替执行**：增量垃圾回收通常会在应用程序的执行过程中交替进行，每个阶段都尽量减少对应用程序执行的阻塞。

**增量垃圾回收的优点**

* **减少停顿时间**：通过将垃圾回收过程分割成小任务并交替执行，避免了长时间的停顿，这对于实时应用或需要高交互的应用非常重要。
* **提高系统响应性**：在垃圾回收期间，应用程序可以继续执行，从而提高系统的响应性。

**结合分代垃圾回收**

增量垃圾回收通常与**分代垃圾回收**策略结合使用。分代垃圾回收将堆分为多个区域（例如年轻代、老年代），并针对不同区域采用不同的回收策略。年轻代的对象由于生命周期较短，通常会频繁进行回收，而老年代的对象生命周期较长，回收频率较低。

增量垃圾回收通过交替的方式对每个代进行处理，减少了全局垃圾回收的频率和时间。

部分回收（Incremental Garbage Collection）

部分回收是一种垃圾回收策略，它将垃圾回收的过程分割成多个小的阶段，每个阶段只回收一部分内存。这样可以避免长时间的停顿，增强系统的响应性。每次回收操作只处理一小部分对象，而不是一次性处理所有对象。

优点：

减少长时间的暂停。

提高系统的响应性，适用于对实时性要求高的应用。

世代垃圾回收（Generational Garbage Collection）

世代垃圾回收是一种基于对象生命周期的垃圾回收策略。它将堆内存划分为多个区域（通常是年轻代、老年代等），并根据对象的生命周期来决定回收策略。年轻代的对象通常存活时间较短，频繁回收；老年代的对象存活时间较长，较少回收。

优点：

* 部分回收通过分阶段进行垃圾回收，减少系统停顿时间。
* 世代垃圾回收通过将内存分为多个代，对不同代的对象采用不同的回收策略，优化性能。
* 49 38 65 97 76 13 27 49  
  前三轮排序结果都是38.49.65.97.76.13.27.49  
  可能是什么排序算法？

排序算法

* 6个数据链路层帧，物理信道失败率10%，无连接服务，则发送网络层分组成功率？

90%的6次方，53%

* XYZN四个字符任意方式进栈，任意方式出栈得到字符串，由多少种入栈排列得到XYZN

卡塔兰数

C4=14

* 在create table时候，unique（a,b）语句在a，b字段语句之前之后之中，有什么影响

UNIQUE(a, b) 约束的效果不受其在 CREATE TABLE 语句中的位置影响，关键是它会确保 字段 a 和 b 的组合值唯一，不允许有重复的 (a, b) 值。

2025.03.15OPPO2h（a3、复筛未通过）

* try后面必须跟catch块，finally可以没有，对吗

try 后面 可以不跟 catch，但 必须 要跟一个 finally 块。

catch 块的顺序：

在多重 catch 块中，较具体的异常类型应该放在前面，较通用的异常类型（如 Exception）应该放在后面。

如果 catch 块顺序错误，编译器会报错（例如，catch (Exception e) 应该放在所有特定异常类型之后）。

在 catch 块中，可以使用 throw 关键字手动抛出异常：

使用 try-catch 块时，通常结合 finally 块来关闭文件、数据库连接等资源。或者，在 Java 7 及以上版本使用自动资源管理（ARM），如：

try (FileReader reader = new FileReader("file.txt")) {

// 处理文件读取

} catch (IOException e) {

// 异常处理

}

* 53842，用冒泡排序，两次完整遍历后是什么样

排序算法

* 数据表1有abbc，数据表2有aaabbcd，那么select\*from1 2 where1.x==2。x返回结果是什么

SQL练习

* HTTP缓存作用是减少流量还是增加带宽

减少流量

* 深度为3的完全二叉树节点最少多少

完全二叉树的定义是：除了最底层外，其他层的节点数都达到最大，且最底层的节点都集中在最左边。

深度为 3 的完全二叉树，最底层有 1 个节点、第二层有 2 个节点、第一层有 4 个节点。

最少的节点数是当所有层都只有一个节点时，即节点数为 1。因此，深度为 3 的完全二叉树最少 4 个节点（假设完全二叉树的深度从 1 开始计数）。

* linux通配符解释? \* [] {}

\*：匹配零个或多个字符。可以匹配任意字符，直到遇到文件系统的边界。例如，\*.txt 匹配所有 .txt 文件。

[]：匹配字符集中的单个字符。例如，[a-z] 匹配所有小写字母，[0-9] 匹配所有数字。

{}：匹配一组选项中的任意一个。例如，{a,b,c} 会匹配 a，b 或 c，例如 file{1,2,3}.txt 会匹配 file1.txt，file2.txt，file3.txt。

* 列出将-r--r--r--权限改为631的所有方法

权限表示法 r--r--r-- 表示所有者、组和其他用户的权限都为只读。

631 权限表示：

所有者：6 -> rw-（读写权限）

组：3 -> wx-（写和执行权限）

其他：1 -> --x（只执行权限）

方法 1：使用 chmod 命令

方法 1: 使用 chmod 631 filename。这将 -r--r--r-- 修改为 rw-wx--x。

方法 2：使用 chmod 的符号方式

方法 2: 使用 chmod u=rw,g=wx,o=x filename，表示为所有者添加读写权限，为组添加写和执行权限，为其他用户添加执行权限。

* 15，9，7，23，45，8，67，93，3快速排序，以第一个记录为基准，从小到大第一次划分结果.3.9.7.8.15.23.45.93.67还是3.9.7.8.15.45.67.93.23？
* 53842冒泡

排序算法

2025.03.19美的1.5h（a3）

单选

* 操作系统动态分区算法回收内存，如果回收区仅仅与空闲区链插入点前一个分区相邻接，那么需要？

空闲区（Free Space）：

空闲区是指内存中未被分配、尚未使用的区域。当操作系统或程序需要内存时，会从这些空闲区中分配合适大小的内存。

这些空闲区是可用的内存区域，等待新的进程或任务分配。

回收区（Reclaimed Space）：

回收区是指已经释放的内存空间。这些内存块曾经被某个进程或任务使用，但在任务结束或不再需要该内存时，被操作系统回收并标记为空闲。

回收区实际上是从之前的使用中回收过来的内存区域，通常是在垃圾回收或内存管理过程中释放出来的区域。

如果回收区仅与空闲区链插入点前一个分区相邻接，那么需要使用 合并 操作。也就是说，在释放内存并将其标记为空闲区时，如果释放的内存区域和其前一个内存块相邻且都是空闲的，那么它们会合并成一个更大的空闲区域。这个操作可以减少内存碎片，提高内存利用率。

* java8引用特性，names.stream().map(string.touppercase.foreachsystem.,out.println),为我修改使其语法正确

names.stream()

.map(String::toUpperCase)

.forEach(System.out::println);

后续学习

* java定义类可以不写构造函数吗

在 Java 中，如果你定义了有参构造函数，是可以不写无参构造函数的，但如果没有显式地定义无参构造函数，Java 编译器 不会自动提供默认的无参构造函数，除非类没有任何构造函数。

* 类里面的private的x属性可以被实例obj.x读取到吗

不可以。private 属性只能在定义它的类的内部访问，不能通过外部实例直接访问。如果你试图通过 obj.x 来访问一个 private 属性，编译器会报错。

* 被中断的进程可以立即获得CPU控制权恢复执行，对还是错
* 被中断的进程将进入就绪队列等待执行，对还是错

错。被中断的进程不能立即恢复执行，而是会先进入一个中断状态，并等待操作系统将其恢复到就绪队列中，直到获得 CPU 的控制权。

对。被中断的进程会被放入就绪队列，等待操作系统调度器重新调度它，并重新获取 CPU 控制权继续执行。

* 为我介绍ICMP

ICMP（Internet Control Message Protocol） 是互联网控制消息协议，主要用于发送错误消息和操作信息。它被广泛用于网络诊断工具，如 ping 和 traceroute。ICMP 可以报告网络设备或主机的可达性、路由错误、超时错误等。ICMP 消息通常由网络设备（如路由器和网关）发送，用于通知源主机或目的主机存在网络问题。

* 书写方法头，使用类名作为前缀就可以调用A.function，方法头应该写成什么样

public static void function() {

// 方法体

}

你必须将方法定义为 static，这样它就可以通过类名直接调用（即 A.function()）。

static：将方法或变量与类本身关联，而不是与实例对象关联。静态成员可以在没有实例化对象的情况下直接访问。

final：使得类、方法或变量不可更改，防止子类对方法进行重写，或防止变量的值被修改。

* linux哪个指令为脚本取消可执行权限

chmod -x script.sh

chmod 644 script.sh

多选

* mysql的四种常见日志哪些需要解析后才能查看

MySQL的四种常见日志：

错误日志（Error Log）：记录数据库启动、停止以及运行中的错误信息。

查询日志（General Query Log）：记录所有的查询语句。

慢查询日志（Slow Query Log）：记录执行时间超过设定阈值的查询。

二进制日志（Binary Log）：记录所有对数据库有更改的操作。

需要解析后才能查看的日志：

二进制日志（Binary Log）：它保存的是二进制格式，需要使用 mysqlbinlog 工具来解析并查看。

其他日志（错误日志、查询日志、慢查询日志）一般是文本格式，直接查看即可。

mysqlbinlog 是最常用的官方工具，用于直接查看 MySQL 的二进制日志。

Percona Toolkit 和 MySQL Enterprise Audit Plugin 提供了更高级的日志分析功能。

MySQL Workbench 和 ELK Stack 提供了图形化和集中化的日志分析解决方案。

* SSL协议客户端和服务器握手阶段交互的信息有？
* 使用的对称算法、不重数、MAC算法、使用的公钥算法，哪些是?
* SSL握手阶段交互MAC算法了吗？

在 SSL/TLS 握手阶段，客户端和服务器会交换一系列的信息，包括但不限于以下内容：

使用的公钥算法：例如 RSA、ECC 等。

使用的对称算法：如 AES、3DES 等。

生成的共享密钥：客户端和服务器通过协商生成对称加密密钥。

MAC算法：用于生成消息认证码（Message Authentication Code），如 HMAC-SHA256。

是否包含 MAC 算法：握手过程中，并不直接交换 MAC 算法。在后续的数据传输过程中，会使用 MAC 来保证数据的完整性和验证。

* XSS攻击可分为存储型和反射型？对吗？
* XSS攻击主要方法是什么？

是的，XSS（跨站脚本攻击） 可以分为两种常见类型：

存储型 XSS：恶意脚本存储在服务器端（如数据库），并在其他用户访问页面时执行。

反射型 XSS：恶意脚本通过 URL 传递给服务器，服务器将其原封不动地返回，执行时攻击者的脚本在浏览器中运行。

XSS 攻击的主要方法：

注入恶意脚本：攻击者通过输入框或 URL 注入 JavaScript 脚本。

DOM-based XSS：通过修改页面的 DOM 元素来执行恶意脚本。

* 对特殊字符转义处理可以预防SQL注入吗？

是的，对特殊字符进行转义（例如 ;、'、-- 等）可以防止 SQL 注入攻击。这是通过将用户输入中的特殊字符转义，使其不能作为 SQL 查询的一部分执行。

然而，更安全的方法是使用预编译语句（Prepared Statements），它能够防止 SQL 注入，并且不会依赖于手动转义。

* 为我枚举java实现多线程的方法？
* 自定义实现runnable缺点是没有返回值吗？
* 需要构造一个thread实例去启动？

Java 实现多线程的常见方法：

继承 Thread 类：创建一个类继承 Thread 类，并重写 run() 方法，然后通过 start() 方法启动线程。

实现 Runnable 接口：创建一个实现了 Runnable 接口的类，并将其传递给 Thread 构造函数，然后调用 start() 启动线程。

使用 ExecutorService：通过线程池（如 Executors.newFixedThreadPool()）来管理线程。

自定义实现 Runnable 的缺点：

没有返回值：Runnable 接口的 run() 方法没有返回值。如果需要获取线程执行的结果，可以使用 Callable 接口，Callable 可以返回结果并抛出异常。

需要构造一个 Thread 实例去启动吗？

是的，需要构造一个 Thread 实例。即使你使用了 Runnable 接口，你依然需要将 Runnable 对象传递给 Thread 构造器来创建 Thread 实例，然后调用 start() 方法来启动线程。

* 按需分配可以破坏死锁的什么条件？
* 银行家算法是死锁解除算法，正确吗？
* 资源动态分配过程防止系统安全状态，可以避免发生死锁，对吗？
* 在资源动态分配过程中防止系统进入安全状态可以避免死锁吗？

按需分配的策略通过确保每个线程在请求资源时，不会超过它所需的最大资源量，避免了死锁的发生。它破坏了死锁的 “持有并等待” 条件，即资源在分配时不会被长期持有，直到它所请求的所有资源都可以分配。

银行家算法是 死锁避免算法，而不是死锁解除算法。银行家算法通过检查资源分配请求是否会导致系统进入不安全状态来防止死锁的发生。

防止系统进入不安全状态是死锁避免的关键。

* 为我判断sql是否正确 update set id = 10 from a where id = 1？
* 为我判断sql是否正确select \* from a limit(7,10)？

UPDATE a SET id = 10 WHERE id = 1;

SELECT \* FROM a LIMIT 7, 10;

SELECT \* FROM a LIMIT 10 OFFSET 7;

SQL练习

2025.03.27携程2h（a1+0.6+0.3+0.2）

2025.03.28饿了么1.5h（a3）

单选

* linux文件命名可以是空格吗

touch "my file.txt" # 使用双引号包裹文件名

touch my\ file.txt # 使用反斜杠转义空格

* ..表示当前目录吗

.. 表示父目录，即当前目录的上级目录。

.（单个点）才表示当前目录。

* /代表根目录的上级目录吗

/ 代表的是根目录，它是文件系统的最顶层目录。根目录没有上级目录。

* 大小写对文件命名有影响吗

有影响。Linux 是区分大小写的系统，因此 file.txt 和 File.txt 被认为是不同的文件。

* 系统按照单道方式运行先来先服务，两个同时到达作业A1A2，按照A1A2顺序入队，执行时间分别为B1B2则平均周转时间

A1周转时间B1

A2周转时间B1+B2

求和除以二

* 一棵二叉树有2n个节点，则它不可能有：2m个度为1的节点

多选

* 借款场景，放款成功后生产还款计划，根据不同资方生成不同计划接口，适合哪种设计模式 工厂？门面？责任链？观察者？

这个场景适合使用 工厂模式（Factory Pattern）。工厂模式通过定义一个创建对象的接口，让子类决定实例化哪一个类。根据不同的资方，工厂方法可以动态地生成不同的还款计划接口。具体来说：

放款成功后，根据资方的不同，生成不同的还款计划。

工厂模式提供了一个工厂接口，依据不同的条件（例如资方类型）创建不同的还款计划对象。

* 结构型设计模式包括？

适配器模式（Adapter）：将一个类的接口转换成客户端所期望的另一种接口。

桥接模式（Bridge）：将抽象部分与实现部分分离，使得两者可以独立变化。

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

装饰器模式（Decorator）：动态地给一个对象添加一些额外的职责。

外观模式（Facade）：为子系统中的一组接口提供一个统一的高层接口。

享元模式（Flyweight）：通过共享技术来支持大量细粒度的对象。

代理模式（Proxy）：为其他对象提供代理以控制对这个对象的访问。

* 通道技术。通道收到CPU指令后就会向CPU发送中断信号吗？

不完全对。在通道技术中，通道的作用是作为 I/O 操作的中介，它负责在计算机的 CPU 和 I/O 设备之间进行数据传输。当通道操作完成时，它会向 CPU 发送中断信号，通知 CPU 任务已完成。通道的工作机制和 CPU 的指令执行是分开独立的，CPU 发出指令给通道后，通道负责完成 I/O 操作，而不是直接向 CPU 发送指令。通道会向 CPU 发送一个中断信号，表明它的任务已经完成。

* web代理服务器在CDN中扮演重要角色？
* web代理服务器可以利用缓存机制减少响应时间对吗？

对。Web 代理服务器在 CDN（内容分发网络）中扮演了重要的角色。CDN 通过代理服务器缓存内容并在多个地理位置的服务器上存储这些内容，从而提供更快的访问速度。代理服务器可以减少源服务器的负担，提供用户请求的内容，减少延迟。

对。Web 代理服务器可以利用缓存机制存储之前请求过的内容，避免每次都请求源服务器，进而减少响应时间。通过缓存热点数据，代理服务器可以更快地响应用户请求，减轻服务器负担，提高整体效率。

* 给我枚举MySQL可以查看test表结构的SQL语句？

DESCRIBE test;

DESC test

SHOW COLUMNS FROM test;

SHOW CREATE TABLE test;

* linux的df命令可以查看什么？可以查看所有文件名称吗？

df 命令：disk free，用于查看文件系统的磁盘空间使用情况，显示各个挂载点的

容量

已用空间、可用空间

文件系统的类型

挂载点

你可以将外部存储设备（如 USB 驱动器）挂载到 /mnt/usb 目录，使得你可以通过该目录来访问 USB 驱动器中的文件。

mount /dev/sdb1 /mnt/usb

这时，/mnt/usb 就是挂载点。

df 命令不能查看文件名称，它仅显示磁盘的空间利用情况，而不是列出文件的具体内容或文件名。如果要查看文件名称，可以使用 ls 命令。

1. `cat` - 查看文件内容

2. `less` - 分页查看文件内容

3. `more` - 分页查看文件内容

4. `head` - 查看文件的前几行

5. `tail` - 查看文件的最后几行

6. `tac` - 倒序查看文件内容

7. `nl` - 查看带行号的文件内容

8. `od` - 八进制查看文件内容

9. `file` - 查看文件类型

10. `stat` - 查看文件详细信息

11. `ls` - 列出文件和目录

12. `grep` - 在文件中搜索文本

13. `find` - 查找文件

2025.03.30阿里云1.5h（a0.2+0+0）

* FTP支持断点续传吗

一般来说，FTP（文件传输协议）本身是支持断点续传的。

FTP 的断点续传功能允许在文件传输过程中出现中断（如网络故障、连接超时等情况）后，能够从上次中断的位置继续传输文件，而不是重新开始整个传输过程。这一功能主要通过 FTP 客户端和服务器之间的命令交互来实现。客户端可以向服务器发送特定的命令，告知服务器要从文件的某个特定位置开始继续传输，服务器则根据客户端的请求，从相应位置读取文件数据并发送给客户端。

然而，FTP 是否能成功实现断点续传还取决于客户端和服务器的具体配置以及它们对断点续传功能的支持程度。如果客户端或服务器一方不支持断点续传，或者在传输过程中出现了不兼容的情况，那么断点续传可能无法正常工作。

* 以82，87，10，77，83，84，28，93为例，帮我演示直接插入排序算法（第一个数字不需要插入）的过程

[82 | 87, 10, 77, 83, 84, 28, 93]

[82, 87 | 10, 77, 83, 84, 28, 93]

[10, 82, 87 | 77, 83, 84, 28, 93]

[10, 77, 82, 87 | 83, 84, 28, 93]

[10, 77, 82, 83, 87 | 84, 28, 93]

[10, 77, 82, 83, 84, 87 | 28, 93]

[10, 28, 77, 82, 83, 84, 87 | 93]

[10, 28, 77, 82, 83, 84, 87, 93]

* sleep和yeild哪个使得线程阻塞

在多线程编程里，sleep 会使线程阻塞，而 yield 一般不会让线程阻塞。下面分别进行详细介绍：

**sleep**

sleep 是一个让线程暂停执行一段时间的方法。在指定的时间内，线程会处于阻塞状态，不会去参与 CPU 时间片的竞争，也就不会执行任何任务。当指定的时间结束之后，线程才会重新进入就绪状态，等待获取 CPU 资源继续执行。

**yield**

yield 通常的作用是让当前线程放弃本次 CPU 时间片，主动让出 CPU 资源，使得其他线程有机会去执行。不过，当前线程并不会进入阻塞状态，而是马上进入就绪状态，等待下一次获取 CPU 时间片。

综上所述，sleep 会使线程阻塞，而 yield 只是让线程让出 CPU 资源，线程仍然处于就绪状态。

* 一个文件20个磁盘块，一个缓冲区和一个磁盘块大小相同，从磁盘读入缓冲区200ms，从缓冲区到用户区100ms，CPU处理一块数据120ms，请问单缓冲区和双缓冲区结构下，分析完这个文件的时间分别是？

**单缓冲区结构**

在单缓冲区结构中，磁盘块的处理过程是串行的，不过磁盘 I/O（读入缓冲区）和 CPU 处理数据可以部分重叠。我们把处理每个磁盘块的过程拆分为：从磁盘读入缓冲区、从缓冲区传输到用户区、CPU 处理数据。

* **前 19 个磁盘块的处理时间**：  
  对于每个磁盘块，从磁盘读入缓冲区需要 200ms，从缓冲区到用户区需要 100ms，这两个操作是顺序进行的，总时长为 200+100=300ms。而 CPU 处理数据的 120ms 可以和下一个磁盘块的读入操作并行。所以前 19 个磁盘块，每个磁盘块实际的耗时取决于磁盘 I/O 操作（读入缓冲区和传输到用户区），即前 19 个磁盘块的总耗时为 19×(200+100)=19×300=5700ms。
* **最后一个磁盘块的处理时间**：  
  最后一个磁盘块完成从磁盘读入缓冲区（200ms）和从缓冲区到用户区（100ms）后，CPU 对其进行处理需要 120ms，这部分无法再与后续操作并行，所以最后一个磁盘块额外需要 200+100+120=420ms。
* **总时间**：  
  将前 19 个磁盘块的处理时间和最后一个磁盘块的处理时间相加，可得单缓冲区结构下分析完这个文件的总时间为 5700+420=6120ms。

**双缓冲区结构**

双缓冲区结构可以让磁盘 I/O 和 CPU 处理更好地并行。一个缓冲区用于磁盘读入数据，另一个缓冲区可以同时进行数据从缓冲区到用户区的传输以及 CPU 处理操作。

* **前 19 个磁盘块的处理时间**：  
  磁盘读入一个缓冲区需要 200ms，与此同时，CPU 可以处理另一个缓冲区中的数据（耗时 120ms），并且将该缓冲区的数据传输到用户区（耗时 100ms）。由于磁盘读入时间最长，所以前 19 个磁盘块，每个磁盘块的处理时间以磁盘读入时间为准，即前 19 个磁盘块的总耗时为 19×200=3800ms。
* **最后一个磁盘块的处理时间**：  
  最后一个磁盘块读入缓冲区需要 200ms，接着将数据从缓冲区传输到用户区需要 100ms，然后 CPU 处理数据需要 120ms。所以最后一个磁盘块总共需要 200+100+120=420ms。
* **总时间**：  
  把前 19 个磁盘块的处理时间和最后一个磁盘块的处理时间相加，双缓冲区结构下分析完这个文件的总时间为 3800+420=4220ms。

综上所述，单缓冲区结构下分析完这个文件的时间是 6120ms，双缓冲区结构下分析完这个文件的时间是 4220ms。

* 线程共享进程的地址空间吗
* 线程拥有自己的堆栈和寄存器对吗？

在操作系统中，同一进程内的多个线程是共享该进程的地址空间的。下面为你详细介绍：

**线程和进程的概念**

* **进程**：进程是程序在操作系统中的一次执行过程，是系统进行资源分配和调度的基本单位。它拥有自己独立的内存空间、系统资源（如文件描述符、信号处理等）。
* **线程**：线程是进程中的一个执行单元，是 CPU 调度和分派的基本单位。一个进程可以包含多个线程，这些线程共享进程的大部分资源。

**线程拥有独立的堆栈**

**原理**

每个线程都有自己的栈空间，这是为了保证线程在执行过程中，局部变量和函数调用信息能够独立存储。当线程调用一个函数时，会在自己的栈上分配空间来保存函数的参数、局部变量以及返回地址等信息。不同线程的栈是相互独立的，这样可以避免线程之间的数据相互干扰。

**示例解释**

假设在一个多线程程序里，有两个线程 Thread A 和 Thread B，它们都调用了同一个函数 func。当 Thread A 调用 func 时，会在自己的栈上为 func 的局部变量和返回地址等分配空间；同样，Thread B 调用 func 时，也会在自己的栈上进行独立的分配。如果没有独立的栈，Thread A 和 Thread B 的数据就会混乱，程序无法正常运行。

* 为我介绍CSRF攻击，利用了什么，可以怎么防御

CSRF（Cross - Site Request Forgery）攻击，也被称为跨站请求伪造攻击，是一种常见的网络安全漏洞。以下是关于它的介绍、攻击利用原理以及防御方法：

**定义与概述**

CSRF 攻击是攻击者通过诱导用户在已登录目标网站的情况下，访问一个恶意链接或执行一个恶意操作，从而利用用户的身份和权限在目标网站上执行未经用户授权的操作，如转账、修改密码、发布文章等。

**攻击利用原理**

* **利用用户的登录状态**：用户在登录目标网站后，浏览器会保存用户的登录凭证（如 Cookie）。只要登录凭证未过期，用户再次访问该网站时，服务器会根据凭证识别用户身份并认为是合法用户。攻击者就是利用了这一点，通过让用户在已登录状态下访问恶意链接，使目标网站误认为请求是用户本人发起的合法操作。
* **构造伪装请求**：攻击者构造一个看似合法的请求，例如一个包含恶意操作的表单提交或者链接点击。当用户访问这个恶意链接或提交表单时，浏览器会自动带上用户在目标网站的登录凭证，向目标网站发送请求。由于请求中包含了合法的登录凭证，目标网站无法区分这是用户的真实意愿还是攻击者的恶意行为，从而执行了攻击者想要的操作。

**攻击示例**

假设用户 A 登录了银行网站进行转账操作，此时浏览器中保存了用户 A 的登录 Cookie。攻击者通过某种方式（如电子邮件、社交媒体消息等）诱骗用户 A 点击一个恶意链接。这个链接实际上是一个伪装的转账请求，指向银行网站的转账接口，但转账的目标账号是攻击者控制的账号。当用户 A 点击链接时，浏览器会自动带上用户 A 的登录 Cookie 向银行网站发送转账请求，银行网站验证 Cookie 后认为是用户 A 的合法请求，从而执行转账操作，导致用户 A 的资金被转到攻击者的账号。

**防御方法**

* **验证请求来源**
  + **同源策略**：浏览器的同源策略限制了不同源（协议、域名、端口号都相同）的网页之间的交互。服务器可以检查请求的来源是否与自己的域名相同，对于非同源的请求进行拒绝或进一步验证。
  + **Referer 字段检查**：检查请求头中的 Referer 字段，确认请求是从合法的页面发起的。但这种方法并不完全可靠，因为 Referer 字段可以被攻击者伪造。
* **使用 CSRF 令牌**
  + **生成与验证令牌**：在用户登录或访问关键页面时，服务器为用户生成一个唯一的 CSRF 令牌，并将其存储在用户的会话中。同时，将令牌通过页面表单或 JavaScript 变量传递给客户端。当用户提交表单或执行关键操作时，客户端将令牌随请求一起发送到服务器。服务器接收到请求后，验证令牌的有效性，如果令牌无效或不存在，则拒绝请求。
* **增加验证码**
  + **强制用户输入验证码**：在关键操作（如转账、修改密码等）页面增加验证码验证。用户需要输入正确的验证码才能完成操作，这样可以有效防止攻击者通过自动化脚本发起 CSRF 攻击。因为攻击者很难获取用户看到的验证码。

以上三种说法都是正确的，它们都可以用于防御 CSRF（跨站请求伪造）攻击，下面为你详细解释：

**验证 HTTP Referer 字段**

* **原理**：HTTP 请求头中的 Referer 字段会记录请求的来源页面。服务器可以通过检查该字段，判断请求是否来自合法的源。如果请求的 Referer 不是本网站的域名，那么服务器可以拒绝该请求，从而在一定程度上防御 CSRF 攻击。
* **示例**：假设用户在 example.com 网站登录后，服务器会在处理关键请求时检查 Referer 字段。如果攻击者诱导用户从恶意网站 malicious.com 发起对 example.com 的请求，example.com 服务器检测到 Referer 为 malicious.com，就会拒绝该请求。
* **局限性**：Referer 字段是可以被伪造的，并且有些浏览器或用户可能会出于隐私等原因禁用或修改 Referer 字段的发送，这会导致该方法的有效性降低。

**使用 Token 验证**

* **原理**：服务器在生成页面时，会为每个用户生成一个唯一的 CSRF 令牌（Token），并将其嵌入到 HTML 表单或 HTTP 头中。当用户提交请求时，客户端需要将该令牌一起发送给服务器。服务器接收到请求后，会验证令牌的有效性。如果令牌与服务器端存储的令牌匹配，则认为请求是合法的；否则，拒绝该请求。
* **示例**：在一个 Web 应用中，当用户登录后访问一个需要进行资金转账的页面时，服务器会生成一个 CSRF 令牌，并将其作为隐藏字段添加到转账表单中。当用户提交转账请求时，表单数据中会包含该令牌。服务器接收到请求后，会验证该令牌是否与服务器端存储的一致。
* **优点**：Token 验证是一种较为可靠的 CSRF 防御方法，因为攻击者无法预测或获取合法用户的 Token。

**设置 SameSite Cookie**

* **原理**：SameSite 是 Cookie 的一个属性，用于控制 Cookie 在跨站请求时的发送行为。通过设置 SameSite 属性，可以限制 Cookie 在不同站点之间的传递，从而减少 CSRF 攻击的风险。SameSite 属性有三个值：
  + **Strict**：严格模式，Cookie 只会在同源请求中发送，即只有当请求的源和目标站点完全相同时，才会发送 Cookie。
  + **Lax**：宽松模式，在大多数跨站请求中不会发送 Cookie，但在一些安全的顶级导航请求（如链接跳转）中会发送 Cookie。
  + **None**：允许 Cookie 在所有请求中发送，但需要同时设置 Secure 属性，即只能通过 HTTPS 协议发送。
* **示例**：如果将网站的登录 Cookie 的 SameSite 属性设置为 Strict，那么当用户从恶意网站发起对该网站的请求时，由于是跨站请求，浏览器不会发送该登录 Cookie，从而防止了 CSRF 攻击。
* **优点**：设置 SameSite Cookie 是一种简单有效的 CSRF 防御方法，浏览器会自动处理 Cookie 的发送，无需服务器端进行复杂的验证逻辑。
* 优先级调度可能导致低优先级进程饥饿？可以通过老化技术避免？
* 优先级调度不需要进程执行时间信息吗？
* 优先级调度总是非抢占式的吗？

**优先级调度导致低优先级进程饥饿**

在优先级调度算法中，系统会根据进程的优先级来分配 CPU 资源。高优先级的进程会优先获得 CPU 执行权，而低优先级的进程则需要等待。如果系统中不断有高优先级的进程进入就绪队列，那么低优先级的进程可能长时间得不到 CPU 资源，从而产生饥饿现象。例如，在一个实时系统中，实时任务具有较高的优先级，而普通的用户任务优先级较低。如果实时任务频繁出现，那么用户任务可能长时间无法执行。

**老化技术避免饥饿**

老化技术是一种用于解决进程饥饿问题的方法。它的基本思想是随着时间的推移，逐渐提高低优先级进程的优先级，使得它们最终能够获得 CPU 资源。具体实现方式可以是每隔一段时间，就将所有进程的优先级进行调整，让低优先级进程的优先级逐渐升高。例如，一个低优先级进程在就绪队列中等待了很长时间，那么它的优先级会逐渐提升，最终可能会超过一些新进入的高优先级进程，从而获得 CPU 执行权，这样就避免了低优先级进程因长时间等待而产生的饥饿问题。

**优先级调度不需要进程执行时间信息**

优先级调度并不一定需要进程执行时间信息。下面从调度的依据以及不同场景下的需求来详细说明：

**调度依据**

优先级调度算法的核心依据是进程的优先级。优先级可以基于多种因素来确定，比如进程的重要性、紧急程度、资源需求等。例如，在一个实时系统中，实时进程的优先级通常会设置得较高，以确保它们能够及时得到处理；而在一个普通的多用户操作系统中，系统进程的优先级可能会高于用户进程。

**有无执行时间信息的情况**

* **无需执行时间信息**：在很多情况下，优先级的设定与进程的执行时间并无直接关联。比如，一个数据库管理系统中，用于处理关键事务的进程会被赋予较高的优先级，无论该进程执行时间长短，它都需要优先处理。
* **可结合执行时间信息**：虽然优先级调度本身不依赖执行时间信息，但在某些场景下，执行时间也可以作为确定优先级的一个参考因素。例如，短作业优先调度（SJF）可以看作是一种特殊的优先级调度，它将执行时间短的进程赋予较高的优先级。不过这只是优先级调度的一种特殊应用，并不是优先级调度的必要条件。

**优先级调度不总是非抢占式的**

优先级调度可以分为抢占式和非抢占式两种，下面分别介绍：

**非抢占式优先级调度**

在非抢占式优先级调度中，一旦一个进程获得了 CPU 的使用权，它就会一直执行下去，直到该进程完成或者因某种原因（如等待 I/O 操作）主动放弃 CPU。即使在该进程执行期间，有更高优先级的进程进入就绪队列，也不会立即抢占当前进程的 CPU 使用权。例如，当一个低优先级进程正在执行时，一个高优先级进程进入就绪队列，它需要等待低优先级进程执行完毕或者阻塞后才能获得 CPU。

**抢占式优先级调度**

在抢占式优先级调度中，当一个更高优先级的进程进入就绪队列时，系统会立即暂停当前正在执行的低优先级进程，将 CPU 资源分配给高优先级进程。例如，一个低优先级进程正在执行，此时一个高优先级进程到达就绪队列，系统会马上剥夺低优先级进程的 CPU 使用权，让高优先级进程开始执行。这种方式能够更好地保证高优先级进程的及时性，但也会增加系统的开销，因为频繁的进程切换需要保存和恢复进程的上下文信息。

* 为我介绍二叉搜索树的特性

二叉搜索树（Binary Search Tree），也称为二叉排序树，是一种特殊的二叉树，具有以下特性：

* **节点值的有序性**：对于二叉搜索树的任意一个节点，其左子树中所有节点的值都小于该节点的值，而其右子树中所有节点的值都大于该节点的值。例如，对于根节点为 5 的二叉搜索树，其左子树中的节点值可能是 1、3 等小于 5 的值，右子树中的节点值可能是 7、9 等大于 5 的值。
* **中序遍历有序性**：对二叉搜索树进行中序遍历（先遍历左子树，再访问根节点，最后遍历右子树），可以得到一个从小到大的有序序列。例如，对于一棵包含节点值为 1、3、5、7、9 的二叉搜索树，中序遍历的结果为 1、3、5、7、9。
* **高效的查找、插入和删除操作**：基于节点值的有序性，二叉搜索树在进行查找、插入和删除操作时具有较高的效率。平均情况下，这些操作的时间复杂度为*O*(*logn*)，其中*n*是树中节点的数量。例如，在一个节点数量为 1000 的二叉搜索树中查找一个特定值，平均只需要比较约 10 次（*log*2​1000≈10）。
* **子树也是二叉搜索树**：二叉搜索树的每个子树也都是二叉搜索树，这使得二叉搜索树具有良好的递归结构，方便进行各种操作和算法设计。例如，在对二叉搜索树进行删除操作时，可以递归地在子树中进行处理。
* **不唯一**：对于给定的一组数据，可能存在多种不同形态的二叉搜索树。例如，对于数据序列 {1, 2, 3, 4, 5}，可以构建出不同高度的二叉搜索树，其形态取决于数据的插入顺序。如果按照 1、2、3、4、5 的顺序插入，得到的是一棵高度为 4 的二叉搜索树；而如果按照 3、2、4、1、5 的顺序插入，得到的是一棵高度为 2 的较为平衡的二叉搜索树。

2025.03.15美团AI面（40min）

* TCP与UDP的区别及使用场景

**TCP**：

面向连接

可靠

按字节

文件传输

**UDP**：

无连接协议

不保证可靠性

报文

实时通信

* Linux查看文件的命令？列出至少三条

cat example.txt用于显示整个文件的内容。

less example.txt逐页显示文件内容，适用于较大的文件。可以向上或向下翻页。  
按 空格键（Space）可以向下翻一页。

按 b 键可以向上翻一页。

head -n 5 example.txt查看 example.txt 文件的前5行

* 请你解释一下String对象不可变

不可变意味着一旦创建了 String 对象之后，它的值无法被改变。所有修改 String 内容的方法都会返回一个新的 String 对象，而不会改变原始对象的值。

如果 String 是可变的，可能会导致安全性问题，比如字符串在不同线程之间的共享时产生不一致的结果。

Java中的字符串常量池（String Pool）就是一个用于存储唯一字符串的池。每次创建一个字符串时，JVM会先检查常量池中是否已存在该字符串，如果存在就返回该字符串的引用，避免重复创建相同的字符串，从而节省内存。

* 什么是ACID？

1. 原子性 (Atomicity)：要么全部执行成功，要么全部不执行
2. 一致性 (Consistency)：一致性保证事务开始前和结束后，遵守数据库的所有约束（如外键、唯一性等）
3. 隔离性 (Isolation)：多个事务并发执行时，一个事务的执行不应受到其他事务的干扰
4. 持久性 (Durability)：一旦事务被提交，其对数据库的修改是永久性的，不会丢失，即使系统崩溃或重启后也能恢复。

* hashCode和equals的关系？为什么要重写hashcode和equals？

如果两个对象通过 equals 方法被认为是相等的，那么这两个对象的 hashCode 方法必须返回相同的哈希值。

HashMap、HashSet、Hashtable依赖于 hashCode 方法来确定对象的位置，依赖于 equals 方法来判断对象是否相等。

* 通过什么途径关注最新技术趋势？说一下你最近关注的新技术。

1. 技术博客（**Stack Overflow**、**GitHub**）
2. 学术资源（google scholar、arXiv）
3. 播客（声动早咖啡）
4. 白皮书/行业报告

可解释AI（Explainable AI）：随着AI系统在各行各业的应用越来越广泛，如何让AI模型的决策过程更加透明和可解释，变得至关重要。学者和研究人员正在积极开发新的可解释性算法和框架。

* 如何快速学习一门新技术？实习或项目中，快速学习的情况如何处理？具体描述一个案例。遇到不会的技术领域问题怎么解决？

1. 紧急程度、学习目标【开发一个基于 Spring Boot 的简单 RESTful API，用于处理用户信息】
2. 文档、教程、社区、书籍
3. 动手实践、渐进式学习【从简单的“Hello World”开始，逐步添加用户管理的功能，如用户的增、删、查、改操作】
4. 请教他人（技术栈、最佳实践）
5. 总结复盘

* 未来三年职业规划

1. 技能提升与基础建设：精通 Java 或 Python 编程语言，掌握常用框架（如 Spring Boot、Django），参与至少两个项目，增加开发经验。
2. 深入领域与专业化：深入学习架构设计，掌握微服务、容器化技术（Docker、Kubernetes）等，争取担任项目负责人，带领团队完成开发任务。
3. 领导职位与影响力：观察学习技术领导职位，如技术经理或架构师，负责技术路线选择、团队技术指导等，提升公司技术影响力，培养团队成员。

2025.03.20美的AI面（40min）

* 英文自我介绍
* 英文最擅长的事情

2025.03.20美团一面（50min）

* 面试官：陶伟 履约平台技术部 base北京
* 自我介绍
* 胡景峰
* 硕士就读于东南大学软件学院，成绩排名第3  
  本科毕业于合肥工业大学计算机学院，绩点排名第17，均为前5%  
  计算机网络、操作系统、数据库系统等课程成绩都是90+甚至95+成绩
* 技术栈Java，熟悉Java并发、Spring框架、MySQL和Redis
* 长期主义、注重积累：单词1500+天、读书1000+小时

对待问题有求索精神，常写博客整理反思：CSDN单篇博客3w+阅读、300+点赞、700+收藏

* 之前有过实习经历吗？实习时间大概什么时候？能实习几个月？实习地点XX有问题吗？

6月初能够到岗Landing，至少3个月，后续可以根据具体情况实习更久。

* HashMap源码及其扩容机制？

**首先，从HashMap的底层结构、扩容阈值、扩容条件说起：**

底层结构是数组+链表，Java8后引入红黑树，当链表长度≥8且数组长度≥64时，链表会转换为红黑树。

假设初始容量为 16、负载因子为 0.75的话，那么扩容阈值16\*0.75=12，当存入第13个元素时扩容为2倍。

Java7超过扩容阈值 && 发生哈希冲突时扩容。

Java8超过扩容阈值即可扩容。还有一种情况，哈希冲突链表长度≥8且数组长度＜64，优先扩容而不是转为红黑树。

**Rehash是HashMap扩容的关键步骤：**

1）新建一个2倍大小的数组。

2）条件判断e.hash & oldCap，0接在loTail后、1接在hiTail后。

HashMap数组长度是2的次方，且扩容为2倍，因此用位运算代替取模，落位是计算好的hash值 & (数组长度-1) ，这是前置基础。

举例，扩容前数组长度16，扩容后32，“数组长度-1”由“4个1”变为“5个1”，其区别相当于将扩容前长度的16的二进制表示作为Mask看hash值对应位置是否为1。

* （追问）简单说一下扩容机制存在的问题？

1. **哈希值分布不均问题**：即使扩容，部分桶中的元素数量仍然较多，影响性能。
2. **Rehash遍历开销问题**：当元素数量庞大时开销过大。
3. **频繁扩容问题**：负载因子过小会频繁扩容，可能导致内存不足。
4. **线程安全问题**：HashMap并非线程安全，Java7多线程扩容可能导致环形链表，进而导致死循环。

* （追问）HashMap和ConcurrentHashMap的区别？

Java 7采用分段锁（Segment）分成多个段，每个段有自己的锁，不同的线程可以同时访问不同的段。

Java 8采用 CAS + synchronized ，当插删时，首先尝试CAS，如果失败则使用 synchronized锁住对应的桶，锁粒度更细。

* （追问）CAS是什么机制？

**CAS（Compare And Swap）**由 CPU 硬件指令保证原子性的无锁并发​：

比较当前值与预期值，相等则将新值写入，否则不操作。

1. ​使用版本号或时间戳解决**ABA 问题**
2. ​限制自旋次数解决**不断自旋重试消耗 CPU**的问题
3. ​封装为复合操作解决**无法直接支持多个变量的原子操作**的问题

* SpringAOP原理

AOP面向切面编程：**切面 = 切点 + 通知**

* **切面（Aspect）：**封装了通用功能，比如权限、日志、事务、性能
* **切点（Pointcut）：**通过切点表达式找到想要织入weaving的方法
* **通知（Advice）：** 拦截到目标对象的连接点之后**要执行的**增强逻辑
* （追问）Spring AOP代理具体如何实现？

**JDK动态代理（**SpringFramework**）**

* 基于接口，通过Java的反射机制实现
* 适合实现接口的的类

**CGLIB动态代理（**SpringBoot2**）**

* 基于类继承，通过字节码生成工具生成子类
* 不能代理 final 类和 final 方法
* MySQL事务隔离级别？

**数据一致性问题：**

* **脏读：**一个事务读取到另一个事务未提交的数据。如果该未提交事务最终被回滚，那么第一个事务读取的数据就是脏的。
* **不可重复读：**在同一事务中，读取同一数据两次，但由于其他事务的提交，读取的结果不同。
* **幻读：**在同一事务执行相同的查询操作，返回的结果集记录数量由于其他事务的插入而发生变化。

**事务隔离级别：**

* **读未提交：**可能发生脏读、不可重复读、幻读现象
* **读已提交：**解决脏读（即不会读到未提交事务的数据）
* **可重复读（默认）**：解决不可重复读
* **串行化**：解决幻读（即数据记录数量前后一致）
* 什么情况下需要分库分表？

1. 单表数据量过大
2. 表结构复杂（字段过多或索引过多）
3. 并发访问量过高

* （追问）如何分库分表
* 水平 拆分 数据行：相同表结构复制
* 垂直 拆分 字段：根据业务划分

分 2 个库以及对应业务 16 张表：

Ds\_0、ds\_1

分片键选择：订单ID+用户ID的复合分片键

创建订单id的时候把用户id的后六位冗余到订单id的尾部，这样在查询时，既可以传用户id也可以传订单id，统一取后六位作为分片键。

分片键只是用来定位具体的表，避免读请求扩散。

当定位到具体的表之后，还是需要根据传入的完整查询条件建立的索引来查找具体的记录。

为了保证订单号生成递增，我们参考雪花算法自定义了一个 DistributedIdGenerator，生成后的分布式 ID 再拼接上用户的后六位。

* （追问）实际如何通过用户ID找到订单？

首先判断 SQL 是否包含用户 ID，如果包含直接取用户 ID 后六位

如果对订单中的 SQL 语句不包含用户 ID 那么就要从订单号中获取后六位，也就是用户 ID 后六位

由于按照后六位分片，所以同一用户自己的所有订单都在一个分片里，通过后六位找到这个分片之后，再去通过**完整的用户 ID** 在这个分片里查询同一用户的所有订单，而不会导致读扩散问题（也就是不用再到其他分片里查找）。

虽然还是拿两个字段作为了分片键，但是由于**自定义了分片算法**，所以无论 SQL 中只有用户 ID，还是只有带用户后六位的订单号，都能找到对应的分片。

Q：那既然我创建订单id的时候把用户id的后六位冗余到订单id的尾部，为什么分片键还选择了shardingColumns: user\_id,order\_sn？不能只是order\_sn？

A：确保 **数据的精确分布** 和 **查询的灵活性。**  
user\_id, order\_sn 的复合分片键可以确保分片策略能够根据这两个字段决定数据存储的位置，而不仅仅是 order\_sn。即使你在 order\_sn 的尾部冗余了用户的后六位，分片策略并不会依赖于这种冗余，而是依据 **完整的复合分片键** 来进行分片，以确保在数据分布、查询和性能优化时都能够一致且高效。**user\_id 和 order\_sn 的复合分片键** 可以确保一个用户的所有订单都在同一个分片，避免了跨分片查询的问题。

* 如何保证订单ID唯一性？

分库分表后为什么选择雪花算法作为ID？

全局唯一：时间戳 + 机器ID + 序列号

有序性：ID按时间递增

高性能：本地生成、速度快，无需依赖数据库或外部服务

如何保障雪花算法在大规模集群下生成不重复？

确保每个节点的机器ID唯一：

集中分配：使用Zookeeper或Redis集中分配和管理机器ID。

配置文件：在部署时为每个节点配置唯一的机器ID。

启动检测：在节点启动时检测机器ID是否冲突。

* （追问）分布式雪花算法如何把用户ID拼到订单ID后面？会不会超过Long最大值？

通过 DISTRIBUTED\_ID\_GENERATOR.generateId() 生成雪花算法生成的全局唯一 ID，并将 **userId % 1000000** 作为用户 ID 的后六位拼接到这个订单 ID 后。

雪花算法生成的 ID 是一个 **64 位的长整型（long）**，通常采用以下结构：

* 1 位：符号位（通常为 0）
* 41 位：时间戳（毫秒级）
* 10 位：工作机器 ID
* 12 位：序列号

通过这些字段，雪花算法可以生成唯一且递增的 ID。

**拼接用户 ID 后六位**：

* 代码中 String.valueOf(userId % 1000000) 取的是用户 ID 的 **后六位**。例如，假设 userId = 123456789，那么 userId % 1000000 的结果是 345789（即用户 ID 的后六位）。
* 然后将该六位数字拼接到雪花生成的订单 ID 后面，形成最终的订单 ID。

**是否会超过 Long 最大值？**

* **雪花算法生成的 ID** 本身是一个 **64 位的 long 类型数字**。在 Java 中，long 类型的最大值为 9223372036854775807，即约为 9.2\*10^18。
* 拼接操作是将用户 ID 的后六位（最多是 6 位数字，范围是 0 到 999,999）作为一个字符串加到雪花算法生成的数字后面。拼接后会变成一个字符串类型，而不是 long 类型。

例如，如果雪花生成的订单 ID 是 123456789012345678，而用户 ID 后六位是 345789，那么最终的订单 ID 字符串就是 123456789012345678345789，它实际上并不会超过 long 的最大值，因为拼接后是字符串类型。

**结论：不会超过 long 的最大值**，因为雪花算法生成的是 long 类型的 ID，而拼接的是用户 ID 后六位的字符串。最终生成的是一个字符串，**不涉及数值类型溢出**。

* Redis你是怎么用的？

Redis进行车次查询，在前端进行筛选

* 缓存穿透问题怎么解决？

缓存穿透 ：请求不存在于数据库的数据 造成 数据库负载、缓存内存耗尽、用户体验差

缓存穿透的常见方案

1. 对不存在的 Key 缓存并把值设为Null，设置短暂过期时间（如 60 秒）。  
   **缺点：**尝试但没注册一个不存在的用户名，该值60s内都不可被注册
2. 查询缓存不命中时使用分布式锁来保证只有一个线程访问数据库。  
   **缺点：**其他用户注册请求缓慢或超时
3. 布隆过滤器存已注册用户名，不在布隆过滤器的一定不存在-可用，如果用户名在布隆过滤器中再查询缓存或数据库。  
   **缺点：**布隆过滤器不能删除元素，注销的用户名无法再次使用。
4. Redis Set存已注册用户名，检查是否在集合内。  
   **缺点：**占用内存。

12306解决注册穿透：布隆过滤器 + Redis Set缓存

1. 布隆过滤器不存在，说明数据库没有-可用
2. Redis Set缓存存在，说明已注销-可用
3. 查询数据库到底有没有

Q：用户频繁申请后注销导致Redis Set。

Redis大Key问题是指在Redis中存在单个Key对应的Value数据量过大。

**A：**

1. 限制1个证件号最多注销5次
2. 对缓存进行分片处理，根据用户名的HashCode进行取模操作，将数据分散存储在多个Set结构中。

Q：现在redis中有一个大key，如何平滑拆分？

**A：**

双写：大的redis key正常写入，同时还要写一个拆分后的。在某一个时机两边数据能够对齐的情况下，从大 Key迁移到新Key上。

* （追问）什么是缓存击穿？如何避免？

Redis缓存穿透

穿透不存在：指查询一个不存在的数据，缓存中没有相应的记录，每次请求都会去数据库查询，造成数据库负载激增。

解决：

* 使用布隆过滤器，过滤掉不存在的请求，避免直接访问数据库。
* 对查询结果进行缓存，即使是不存在的数据，也可以缓存一个标识，以减少对数据库的请求。

Redis缓存击穿

击穿热点：指某个热点数据在缓存中过期，导致大量请求同时访问数据库，造成数据库负载激增。

解决：

* 使用互斥锁，确保同一时间只有一个请求可以去数据库查询并更新缓存。
* 热点数据永不过期。

Redis 缓存雪崩

多个雪崩：指多个缓存数据在同一时间过期，导致大量请求同时访问数据库，造成数据库负载激增。

解决：

* 采用随机过期时间策略，避免多个数据同时过期。
* 使用双缓存策略，将数据同时存储在两层缓存中，减少数据库直接请求。
  + 步骤1：读取数据时的逻辑
    - 首先尝试从主缓存中读取数据。
    - 如果主缓存中没有数据，则尝试从备份缓存中读取数据。
    - 如果备份缓存也没有数据，则从数据库中读取数据，并同时更新主缓存和备份缓存。
  + 步骤2：写入数据时的逻辑
    - 当有新的数据写入缓存时，同时更新主缓存和备份缓存。
    - 为主缓存设置一个短的过期时间，为备份缓存设置一个较长的过期时间。
    - 当主缓存失效时，备份缓存可以继续提供数据，减少对数据库的直接访问。
* OSI七层模型？

OSI/RM七层模型

① 应用层：在应用与网络间作为接口

② 表示层：对数据编码、加密、压缩、转换

③ 会话层

④ 传输层（通信子网与资源子网的分界层）：提供端到端的可靠透明数据传输

TCP协议

⑤ 网络层（通信子网的最高层）：逻辑地址寻址、路由选择

IP协议

⑥ 数据链路层：提供可靠数据传输链路

⑦ 物理层：屏蔽具体设备的差异，使链路层不必关心具体传输介质。

TCP/IP网络四层模型

应用层：提供网络服务，比如 FTP、DNS等

传输层：提供端到端可靠数据流传输服务

网际层：负责异构网或同构网的进程间通信,

将传输层分组封装为数据报格式进行传送,

每个数据报必须包含目的地址和源地址。

网络接口层：是网络访问层，负责与物理网络的连接。

* TCP和UDP的区别？

TCP（Transmission Control Protocol）和UDP（User Datagram Protocol）是两种常用的传输层协议，它们在网络通信中有一些重要的区别：

1. 连接性：

TCP： 提供面向连接的通信。在数据传输之前，必须先建立连接，然后进行可靠的数据传输，最后再释放连接。

UDP： 是面向无连接的通信。每个数据包都是独立的，不需要先建立连接，直接发送数据。因此，UDP通信更为轻量，但不保证可靠性。

2. 可靠性：

TCP： 提供可靠的数据传输，通过序列号、确认和重传机制确保数据的完整性和顺序性。如果数据包丢失，TCP会进行重传。

UDP： 不提供可靠性保证，数据包可能会丢失，也不保证数据包的顺序性。对于某些实时性要求高的应用，可以容忍少量的丢包。

6. 连接和断开：

TCP： 建立连接时有三次握手的过程，断开连接有四次挥手的过程。

UDP： 无连接的特性，不需要建立和断开连接的过程。

5. 适用场景：

TCP： 适用于对数据完整性和顺序性有要求的应用，如文件传输、网页访问等。

UDP： 适用于实时性要求高、可以容忍少量数据丢失的应用，如语音通话、视频直播等。

3. 数据传输方式：

TCP： 数据以流的方式进行传输，被划分为小的数据段，然后通过序列号进行重新组装。数据传输是可靠而有序的。

UDP： 数据以数据包的形式传输，每个数据包都是独立的。UDP更适用于需要快速传输的场景，如实时音视频。

4. 头部开销：

TCP： TCP头部相对较大，包含序列号、确认号、窗口大小等字段，用于保证可靠性和流控制。

UDP： UDP头部较小，只包含源端口、目标端口、长度和校验和等基本信息，开销较小。

总体来说，选择TCP还是UDP取决于具体的应用需求。TCP适用于要求可靠性和有序性的场景，而UDP适用于实时性要求高、可以容忍少量数据丢失的场景。在某些应用中，也可以结合使用TCP和UDP，根据不同的数据传输需求选择合适的协议。

* 进程和线程的区别？

进程：是资源分配的基本单位，一个进程可以包含多个线程

进程有独立的上下文（代码、数据、文件）等，创建和切换开销大

进程之间是相互独立的，需要进程间通信（IPC）机制，如管道、消息队列、共享内存、套接字等

线程：是CPU调度的基本单位，属于进程

线程组内共享资源，但有独立的PC和堆栈，创建和切换开销较小

线程直接读写内存即可，但需要同步机制以避免数据竞争

* （追问）如何创建多个线程？

1. 继承 Thread 类

继承 Thread 类，并重写 run() 方法，该方法包含了线程要执行的任务。之后创建该类的实例，调用 start() 方法启动线程。

2. 实现 Runnable 接口

实现 Runnable 接口的 run() 方法，把任务逻辑写在其中。接着创建 Thread 类的实例，将实现了 Runnable 接口的对象作为参数传入，最后调用 start() 方法启动线程。

3. 实现 Callable 接口并结合 FutureTask

Callable 接口的 call() 方法有返回值，能抛出异常。借助 FutureTask 包装 Callable 对象，再将 FutureTask 对象作为参数创建 Thread 对象，最后调用 start() 方法启动线程。

4. 使用线程池

通过 ExecutorService 创建线程池，提交任务到线程池执行。

* 讲一讲项目

外卖平台主要是为了在实践中深入理解Spring IoC、AOP思想

12306主要是为了学习缓存、消息队列、分库分表等技术

* 为什么要分页排序？分页排序如何实现？

1. **提升性能**：当数据量较大时，如果一次性获取所有数据并进行排序，会消耗大量的内存和时间。分页排序可以每次只获取部分数据并进行排序，减少内存占用和提高查询效率。
2. **优化用户体验**：在网页或应用程序中展示数据时，将数据分页展示可以避免一次性加载过多数据导致页面加载缓慢。同时，用户可以更方便地浏览和定位到自己需要的数据，提高数据的可读性和可操作性。

* （追问）从SQL底层原理和深度分页问题回答分页到1k页2k页会面临什么问题？如何优化和解决？
* **性能问题**
  + **数据扫描量大**：LIMIT 子句是先从数据表中读取数据，然后根据偏移量和行数进行筛选。随着页数增加，偏移量增大，数据库需要扫描大量的数据页才能定位到指定页的数据。例如，每页 10 条数据，1k 页的偏移量就是 10000，数据库需要先扫描前面 10000 条记录，这会消耗大量的时间和资源。
  + **索引失效风险**：在深度分页时，由于需要扫描大量数据，可能会导致索引无法有效发挥作用。数据库可能会选择全表扫描来获取数据，而不是通过索引快速定位，进一步降低查询性能。
* **内存占用问题**：数据库处理查询时，会在内存中存储临时结果集。深度分页查询需要获取大量数据，可能使临时结果集占用过多内存。若内存不足，数据库可能会使用磁盘临时文件来存储数据，这会导致查询性能大幅下降，因为磁盘 I/O 操作比内存操作慢得多。

**优化和解决方法**

* **优化查询语句**
  + **利用覆盖索引**：确保查询语句中使用的列都在索引中，这样数据库可以直接从索引中获取数据，而无需回表查询，减少数据扫描量。例如，如果查询只需要 id、name 列，且这两列在一个联合索引中，那么可以创建覆盖索引来优化查询。
  + **合理使用索引**：分析查询条件，确保在经常用于过滤和排序的列上创建索引。对于深度分页查询，索引可以帮助快速定位到指定页的数据，减少全表扫描的可能性。
* **采用书签式分页**：可以使用上一页的最后一条记录的某个唯一标识（如主键）作为下一页查询的条件，通过 WHERE 子句过滤数据。例如，假设主键是 id，上一页最后一条记录的 id 是 100，每页显示 10 条记录，那么下一页的查询语句可以是 SELECT \* FROM your\_table\_name WHERE id > 100 ORDER BY id LIMIT 10。这样可以避免使用大的偏移量，提高查询性能。
* **优化数据库配置**：适当调整数据库的缓存大小、查询缓存等配置参数，以提高数据库的性能。例如，增加 InnoDB 存储引擎的缓冲池大小，使更多的数据和索引可以缓存在内存中，减少磁盘 I/O 操作。
* **数据分库分表**：如果数据量非常大，可以考虑将数据按照一定的规则进行分库分表。例如，按照时间范围、业务模块等进行划分。这样可以将数据分散到多个数据库或表中，减少单个表的数据量，提高查询性能。在查询时，可以根据具体的查询条件，只查询相关的数据库或表。
* **应用层缓存**：在应用层使用缓存技术，如 Redis 等，将常用的数据缓存起来。对于分页查询，可以将查询结果缓存起来，当用户再次访问相同页面时，直接从缓存中获取数据，减少数据库查询次数。同时，可以设置缓存的过期时间，以保证数据的及时性。
* 外卖平台的定时任务是什么任务？（区分12306的延时消息）

定时任务，Chron表达式

* AI信息？平时学习AI的工具使用？

腾讯云黑客松

Deepseek

清华大学PPT

推理模型 vs 模型

知识掌握得比人多

需求掌握得比人少

提示词工程、编程思路、自我判断

* （追问）大模型幻觉是什么？为什么会造成幻觉？

大模型的“幻觉”（Hallucination）是指模型生成与输入无关、不符合事实或逻辑上不合理的内容。这种现象在文本生成、问答等任务中尤为常见，表现为模型“虚构”信息、捏造细节或给出错误答案。以下是关于幻觉的详细解析：

---

### \*\*一、幻觉的常见表现\*\*

1. \*\*虚构事实\*\*

- 例如：当被问及“谁在2025年赢得了诺贝尔奖？”时，模型可能编造一个不存在的获奖者。

2. \*\*逻辑矛盾\*\*

- 例如：回答“水的沸点是多少？”时，可能同时给出“50°C”和“100°C”两个矛盾答案。

3. \*\*过度补充细节\*\*

- 例如：要求总结一篇新闻，模型可能添加原文中未提及的人物或事件。

---

### \*\*二、幻觉产生的原因\*\*

#### 1. \*\*训练数据的局限性\*\*

- \*\*数据噪声\*\*：训练数据中可能包含错误或矛盾的信息，模型学习后复现这些错误。

- \*\*知识覆盖不全\*\*：模型对某些领域（如最新事件、小众知识）缺乏足够数据，只能“猜测”。

- \*\*统计偏见\*\*：模型倾向于高频出现的答案，即使它们不正确（例如“莎士比亚写了《哈利波特》”）。

#### 2. \*\*概率生成的本质\*\*

- 大模型通过预测“下一个词”生成内容，选择的是\*\*概率高\*\*而非\*\*正确\*\*的词汇。这种机制容易在长文本中累积错误。

- 例如：生成“拿破仑登月”的故事，因为“登月”与“人类成就”在训练数据中高频关联。

#### 3. \*\*提示（Prompt）设计问题\*\*

- 模糊或矛盾的提示可能导致模型“脑补”。例如：“描述一个科学家的生平”，未指定具体人物时，模型可能虚构。

#### 4. \*\*缺乏真实世界反馈\*\*

- 模型无法像人类一样通过实践验证答案（例如：无法真正计算“1+1”是否等于2），只能依赖训练数据的统计规律。

#### 5. \*\*过参数化与过拟合\*\*

- 大模型参数规模庞大，可能记住训练数据中的噪声或异常模式，并在生成时复现。

---

### \*\*三、技术层面的解释\*\*

1. \*\*自回归生成的误差传播\*\*

生成每个词时的小错误会逐步放大（如时间、地点、人物名称的微小偏差导致整体错误）。

2. \*\*注意力机制的局限\*\*

模型可能错误分配注意力权重，忽略关键输入信息，转而依赖内部记忆的关联。

3. \*\*缺乏显式知识验证\*\*

模型没有内置的“事实核查”模块，无法在生成过程中验证逻辑一致性。

---

### \*\*四、缓解幻觉的方法\*\*

1. \*\*数据优化\*\*

- 清洗训练数据，减少噪声；引入高质量知识库（如维基百科、学术文献）。

2. \*\*模型改进\*\*

- 使用检索增强生成（RAG），实时检索外部知识辅助回答；通过强化学习对齐人类反馈（RLHF）。

3. \*\*提示工程\*\*

- 明确约束生成范围（如“仅基于以下文本回答”）；要求模型标注不确定性（如“根据我的知识，可能是…”）。

4. \*\*后处理校验\*\*

- 通过外部工具验证事实（如调用搜索引擎或计算器）；多模型交叉验证。

---

### \*\*五、为什么难以彻底消除？\*\*

- \*\*知识动态性\*\*：现实世界持续变化，模型无法实时更新（如2024年后的事件）。

- \*\*语义复杂性\*\*：某些问题没有明确答案（如哲学讨论），模型可能被迫“创造”。

- \*\*权衡创造性\*\*：完全避免幻觉可能让模型过于保守（例如拒绝回答不确定的问题）。

---

### \*\*总结\*\*

大模型幻觉是概率驱动生成的固有缺陷，根源在于数据、算法与真实世界的脱节。尽管可通过技术手段缓解，但完全消除需突破当前范式（如结合符号推理或动态知识更新）。理解幻觉的成因有助于更谨慎地使用大模型，尤其在医疗、法律等高风险领域。

* 手撕代码

将数字num逆序输出，注意符号和边界

输入：123

输出：321

输入：-456

输出：-654

输入：214647483638

输出：0

* （追问）简单说一下思路

214748364？/10

* 【反问】业务内容Base地点和HC分布？

北京

核心本地商业-业务研发平台

以前的到家到店合并成立的业务研发平台

我们属于到家下面的履约平台技术部

* 【反问】笔试过后的面试部门分配是怎么来的？

每天早上10点，各个部门去抢笔试出来的简历，看机缘

* 【反问】纸质简历会处理吗？

不清楚，猜测应该会录入系统

线上投递了，纸质简历意义不大

* 【反问】实习职责

Mentor带领了解整个履约平台技术部的工作、在前一周介绍组内业务、代码规范、开发流程

小的、非核心项目，尝试写一些技术方案设计、代码、单元测试、上线

一般实习3个月的都会写1-2个需求上线

美团学习资料还是挺多的

* 【反问】面试改进点？哪些好的地方继续保持？

项目清晰度、不熟悉

大模型开发：AI兴趣、全拥抱AI、学习积极度

八股基础还可以

2025.03.27美的一面（20min）

* 自我介绍

我叫胡景峰，本科毕业于合肥工业大学计算机学院，硕士就读于东南大学软件学院，成绩排名均为前5%。

技术栈Java，熟悉Spring、MySQL、Redis。

长期主义、注重积累。单词1500+天、读书1000+小时、单篇博客3w阅读、300点赞、700收藏。

* 12306 开源项目 二开 是吧？

GitHub开源项目12306，大规模的在线火车票订购平台，涉及复杂业务逻辑和高并发处理

在开发过程中遇到了一些挑战：

* 如何处理并发访问和保证数据的一致性
* RocketMQ 消息防止重复消费

完成了一个基本功能完善的 12306 项目。

**回答核心**

注意在回答时，强调你对项目的兴趣和学习动力，以及你在项目中所学到的技术和经验，这样能更好地展现你的积极性和学习能力。

同时，可以适当强调你在解决问题的能力，这些都是面试官非常关注的点。如果你有小组的话，可以说和团队合作方面的事项，都是加分项。

* 挑一个复杂、改进效果非常明显的点介绍一下？【超时订单延时取消】  
  其他难度复杂度更高的点？【余票扣减】
* 后端一次查询，通过网络抓取发现是前端做的筛选。

Q：为什么用Redis不用ElasticSearch

A：

1. 实时性要求高。

2. 内存占用低、部署扩展性好。

3. 列车查询不涉及全文搜索和模糊搜索，查询时后端给前端所有符合出发地\_目的地\_日期的车次，其他条件由前端筛选。

* 获取令牌是按什么获取？  
  用户下单购票时候逻辑是怎么样的？  
  获取余票令牌在用户操作界面上来讲有什么体现吗？你了解现在12306的做法吗？

传统秒杀架构：在用户购买车票时，系统会先从余票缓存中扣减库存，然后再更新数据库，完成整个流程。

遇到极端场景时——例如扣减完余票缓存后，系统宕机，导致数据库未能成功更新——就可能出现前端展示余票为 0，但数据库中实际还有库存的情况。在这种情况下，就像电商平台的处理方式一样，余票为 0 的记录是不允许用户继续提交订单的。

对于传统秒杀架构，如果发生类似情况，当某个商品库存为 0 时，可以通过定时任务或异步任务检查数据库与缓存的库存是否一致，不一致时同步更新缓存。但这也带来了一些潜在风险。例如，在宕机时，如果商品库存充足（例如 100），扣减缓存后变为 99，发生宕机时，数据库和缓存中的库存会不一致。直到库存余量为 0 之前，这种不一致可能会持续存在。虽然这不会引发超卖，但商品显示不足也是一个问题。解决方案之一是通过定时任务频繁刷新商品余量，但对于大量商品而言，这样的做法会消耗较大的性能，而且在刷新过程中，数据库和缓存的状态可能不一致（例如缓存为 99，数据库仍为 100，这时需要将缓存库存恢复为 100？），这容易导致更多问题。

使用 Canal Binlog 的方式更加优雅且高效，能够避免这些潜在问题。

令牌容器架构：在用户购买车票时，首先从令牌容器（它代表了余票库存）中扣减令牌，然后再扣减数据库，并通过 Binlog 同步更新余票缓存。用户看到的是余票缓存，而令牌容器缓存则属于技术实现层面的内容，对用户透明。

而如果我们使用令牌容器架构，遇到类似问题时，令牌余量为 0，余票缓存正常，数据库也正常。此时，只要余票缓存不为 0，哪怕令牌容器中的令牌已耗尽，也可以允许用户下单，并通过该操作触发令牌容器的刷新。

* 给用户发送通知是等十分钟后吗？

【技术选型、为什么不用定时任务】

通过定时任务是一种常见的订单延迟关闭解决方案。

1. 延迟时间不精确
2. 高并发场景下，可能导致大量的定时任务同时执行
3. 订单表按照用户标识和订单号进行了分库分表，那这样的话，和上面说的根据订单创建时间去扫描一批订单进行关闭，自然就行不通。因为根据创建时间查询没有携带分片键，存在读扩散问题。

读扩散问题：在数据库分库分表场景下，当查询条件不是分片键时，无法定位到具体要查询的分表，就需要对所有分表都执行查询语句。

* 不用RabbitMQ是因为RabbitMQ 的延时消息特性是基于消息的 TTL（Time-To-Live）来实现的，因此消息的延时时间并不是完全准确的，可能会有一定的误差。
* 不用Redis 过期监听是因为如果 Redis 宕机或重启，那些已经设置了过期时间但还未过期的订单信息将会丢失，导致这部分订单无法正确关闭。
* 不用Redisson是因为Redis 过期监听消息存在的问题，RDelayedQueue 也都会有，因为 RDelayedQueue 本质上也是依赖 Redis 实现。

在订单生成时，我们将订单关闭消息发送到 RocketMQ，并设置消息的延迟时间为十分钟。RocketMQ 支持设置消息的延迟时间，可以通过设置消息的 delayLevel 来指定延迟级别，每个级别对应一种延迟时间。这样，订单关闭消息将在十分钟后自动被消费者接收到。

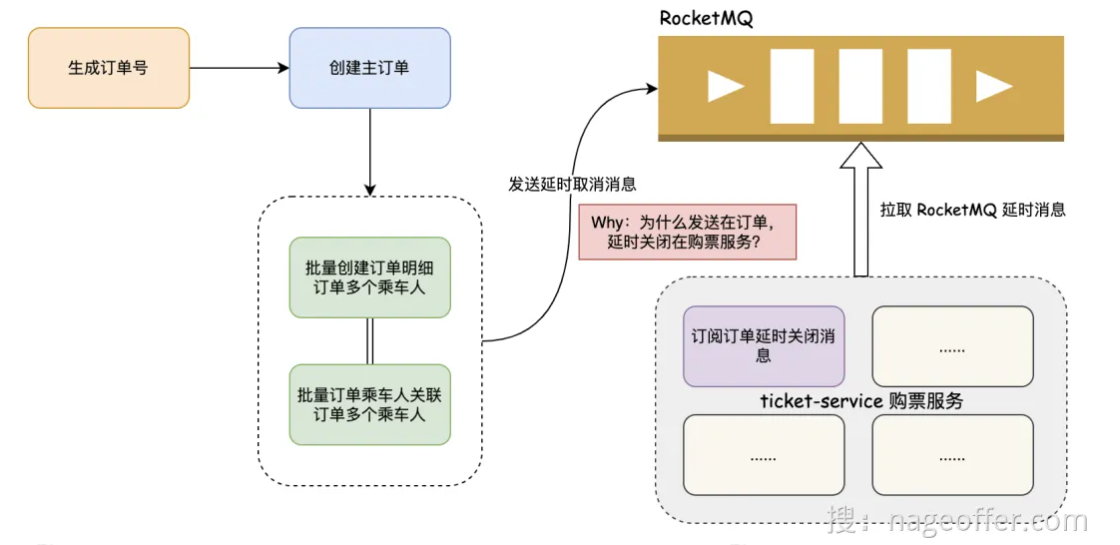
为了处理订单关闭消息，我们需要在消费者端创建一个消息监听器。当消息监听器接收到订单关闭消息时，触发订单关闭操作，将订单状态设置为关闭状态。

RocketMQ 的消息传递机制保证了消息的可靠性传递，因此消息可能会进行多次重试。为了确保订单关闭操作的幂等性，即多次执行不会产生副作用，我们需要在订单关闭逻辑中进行幂等性的处理。

【延时消息不能删除】

Q：创建订单时会发送一个十分钟未支付则关闭订单的 RocketMQ 消息，如果说创建订单并且支付了，这个消息能删除么？

A：延时消息一旦发出去，就一定会执行。所以，咱们要从业务逻辑上去判断，避免已支付的订单再被延时关闭。



订单延迟关闭的发送入口在订单服务的创建订单接口。

https://www.yuque.com/magestack/12306/ldw8nxp96yfg7cgx

考虑到项目依赖关系的关系，延迟关闭订单消息的消费者并没有放到订单服务，而是放在了购票服务。先来一波简单的项目调用关系梳理，然后会和大家说我为什么这么设计。

大家通过调用关系得知，用户购买车票是先调用购票服务，然后购票服务调用订单服务创建订单。那么，此时已知购票服务依赖订单服务。

这个时候，咱们再来看看延时取消订单业务逻辑都在做什么事情：

* 修改订单相关状态，比如将待支付变更为已取消。
* 解锁订单相关的座位状态，咱们下单时锁定了用户提交的座位状态为锁定状态，需要解锁。
* 回退订单中乘车人购买车座类型的缓存余票数量。
* 回退订单中乘车人购买车座类型的令牌限流数量。

延时关闭订单同时需要操作购票服务和订单服务，已知购票服务依赖订单服务，从微服务架构设计上，应尽量避免循环依赖问题。

所以我们将延时关闭订单的消费者放到购票服务，再通过购票服务远程调用订单服务修改订单状态。

订单取消前进行了状态判断，如果是非待支付的状态直接返回 false，购票服务判断返回值为 false 就不进行接下来的座位解锁、余票更新等操作。

* 抖音下单后会有倒计时，到了后会释放，你是这么做的吗？  
  用户下单时候会知道十分钟后释放吗？

为了实现这一功能，常见的技术方案包括使用消息队列和定时任务。​例如，可以将未支付订单的信息发送到一个延迟队列，并设置特定的过期时间。当订单超过设定的支付时间未完成支付，系统会自动处理这些超时订单，释放库存和资源。​

倒计时的实现通常依赖于后端技术，如使用延迟队列来管理定时任务。​例如，RabbitMQ等消息队列系统支持设置消息的过期时间，通过死信队列机制，将过期的消息转发到指定队列，供消费者处理。这种方式能够确保系统在高并发情况下仍能高效、可靠地处理订单超时逻辑。​

抖音下单后的倒计时支付功能主要通过**定时任务调度、状态检查机制和分布式系统技术**实现，其核心原理是**在订单生成时设定超时时间，并通过后台定时器或延迟队列触发状态检查，最终自动释放库存或取消订单**。以下是具体实现技术及流程：

一、倒计时功能的技术实现原理

​定时任务调度

订单生成时，系统记录创建时间并设定超时阈值（例如15分钟）。

后端通过以下技术启动倒计时：

​ 延迟队列（DelayQueue）​：将订单对象存入队列，设定延迟时间为超时阈值，到期后自动取出并处理。

前端倒计时显示

用户端通过以下方式展示剩余时间：

​前端定时器（如JavaScript/Vue的setInterval）​：每隔1秒更新剩余时间，计算方式为：超时时间戳 - 当前时间戳，并将结果转换为分秒格式。

​时间同步：前端需依赖服务器时间而非本地时间，防止用户篡改设备时间绕过限制。

* 缓存、消息队列具体的使用场景
* 缓存余票信息的数据结构

Key：Region\_train\_station\_起始城市\_终点城市\_日期

Value：

Key：列车ID\_起始站\_终点站

Val：列车详细信息

* Binlog是数据库底层日志信息，如何抓取

Binlog工具拿到log，处理，发送消息

* Redis缓存更新数据是如何同步的|  
  先更新数据库，再删缓存，再缓存吗？  
  同时写回Redis如何避免冲突

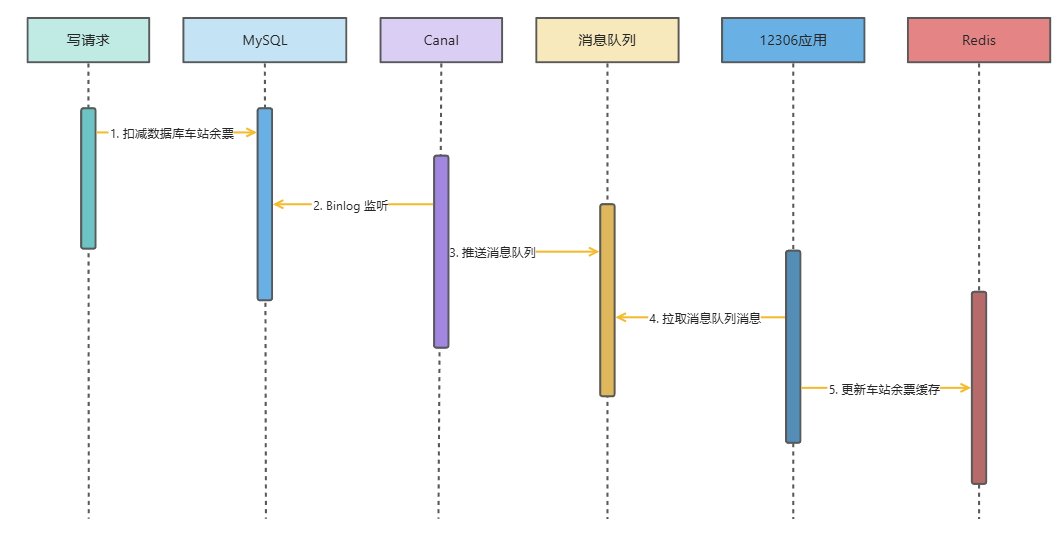
先写数据库

然后删掉缓存

等下一个用户查询和写，保证写Redis的只有一个地方

多个地方，并发会造成报错等一堆问题

实践过程中多练习练习



* 微服务框架组件构成

（1）注册中心 Nacos

服务名称、IP 地址、端口号等

注册中心定期检查，剔除故障实例

（2）网关Spring Cloud Gateway

所有外部请求都先经过网关，根据请求的 URL、请求方法、请求头信息路由到相应的微服务实例上

对请求进行过滤和拦截，比如可以在网关层对请求进行 JWT 令牌验证，只有验证通过的请求才能被转发到后端微服务

（3）熔断限流Sentinel

**熔断**：微服务故障时，为了防止故障扩散，暂时切断对该服务的调用。根据服务的调用失败率、响应时间等指标触发熔断。定期尝试恢复对服务的调用。

**限流：**

1. 令牌桶算法：每秒生成一些令牌放入桶中，请求只有获取到令牌才能被处理，如果桶中没有令牌则拒绝请求。
2. 漏桶算法：将请求放入一个固定容量的桶中，以固定的速率从桶中取出请求进行处理，当桶满时，新的请求将被丢弃。

（4）配置中心Spring Cloud Config

1. **统一配置管理**：包括数据库连接配置、缓存配置、日志级别配置等。
2. 动态更新：变化时，配置中心能够实时推送，使微服务能够在运行时动态地更新配置，而无需重启服务。
3. **环境隔离**：可以为不同的环境（如开发环境、测试环境、生产环境）提供不同的配置信息。

* SSM框架熟悉得多吗

SSM 框架是 Spring、SpringMVC 和 MyBatis 三个框架的整合，以下是其最重要最核心的三个知识点：

**Spring 的 IOC（控制反转）容器**

* **概念**：IOC 是一种设计模式，它将对象的创建和依赖关系的管理从应用程序代码中分离出来，交给 Spring 容器来负责。在传统的开发中，对象之间的依赖关系是在代码中通过直接实例化或查找来实现的，而在 Spring 框架中，通过 IOC 容器，对象的创建和依赖关系的注入由容器来完成，对象只需要被动地接受依赖对象，而不需要主动去获取。
* **作用**：实现了对象之间的解耦，提高了代码的可维护性和可扩展性。例如，一个业务逻辑类需要依赖一个数据访问类，如果没有 IOC，那么业务逻辑类中就会直接实例化数据访问类，当数据访问类的实现发生变化时，业务逻辑类的代码也需要修改。而使用 IOC，业务逻辑类只需要声明对数据访问类的依赖，具体的实例化和注入由 Spring 容器来完成，业务逻辑类不需要关心数据访问类的具体实现，这样当数据访问类的实现发生变化时，只需要在 Spring 配置文件中修改相关配置，而不需要修改业务逻辑类的代码。

**SpringMVC 的请求处理流程**

* **流程**：用户发送请求到前端控制器 DispatcherServlet，DispatcherServlet 接收到请求后，根据请求的 URL 调用处理器映射器 HandlerMapping，HandlerMapping 根据 URL 找到对应的处理器 Handler，并将其返回给 DispatcherServlet。DispatcherServlet 再调用处理器适配器 HandlerAdapter 来执行 Handler，Handler 执行完成后返回一个 ModelAndView 对象给 DispatcherServlet，DispatcherServlet 最后根据 ModelAndView 中的视图信息，调用视图解析器 ViewResolver 来解析视图，将模型数据填充到视图中，并将最终的响应返回给用户。
* **作用**：通过这种分层的请求处理流程，实现了请求的统一处理和视图的统一渲染，使得 Web 应用的开发更加规范和易于维护。不同的组件负责不同的功能，如处理器映射器负责根据 URL 找到处理器，处理器适配器负责执行处理器，视图解析器负责解析视图，各个组件之间相互协作，共同完成请求的处理和响应。

**MyBatis 的数据持久化**

* **功能**：MyBatis 是一个持久层框架，它主要用于实现 Java 对象与数据库表之间的映射关系，以及对数据库的操作。通过 MyBatis 的配置文件和 SQL 映射文件，可以将 Java 对象的属性与数据库表的列进行映射，定义各种 SQL 语句来实现对数据库的增删改查操作。
* **优势**：MyBatis 在 SQL 语句的编写上非常灵活，开发人员可以根据具体的业务需求编写复杂的 SQL 语句，同时它也提供了很好的缓存机制，能够提高数据访问的性能。例如，在查询数据时，可以根据配置启用一级缓存和二级缓存，当多次查询相同的数据时，可以直接从缓存中获取，而不需要再次访问数据库，从而提高了查询效率。
* 复杂查询慢的SQL从哪些方面考虑优化这个SQL

SQL 语句本身

1. 避免使用 SELECT \*，只选择需要的列
2. **子查询 ->**  JOIN连接查询，子查询的性能通常不如连接查询
3. 避免在索引列上使用函数或表达式，因为这可能导致索引失效

索引优化

1. 创建合适的索引，过多的索引会增加数据插删改的开销
2. **EXPLAIN 分析**执行计划

* select\_type：查询类型
  1. SIMPLE简单查询
  2. PRIMARY主查询
  3. SUBQUERY子查询
* type：访问类型
  1. range范围扫描
  2. index索引扫描
  3. ALL全表扫描
* key：实际用到的索引
* rows：扫描行数，越小越好
* Extra：额外信息
  1. Using index表示使用覆盖索引，覆盖索引
  2. Using where表示使用 WHERE 条件进行过滤，索引下推
  3. Using temporary表示使用临时表，通常不好 临时表开销大
  4. Using filesort表示需要额外的排序步骤，很不好 尤其数据量大时

数据库配置

1. **增加缓冲池大小**，使更多的数据和索引可以缓存在内存中
2. 将**数据文件和日志文件**分布在不同的磁盘上，减少磁盘 I/O 冲突

数据库服务器性能

1. **读写分离**
2. **集群**负载均衡

* 反问

企业数字平台 = IT部门

事业部：空调、洗衣机、冰箱、营销、国际物流、供应链

事业部所用的操作系统都是由IT部门提供产品开发测试团队

Base 武汉、佛山

技术栈：Java、微服务框架

2025.04.09阿里云一面（50min）

HR 曹梦

TAM 枫桥Garvin

技术服务经理（Technical Account Management, TAM）

* 做过大模型项目吗
* K8s、网络相关知识有了解吗
* Docker的优势是什么

 **环境一致性**

* 开发、测试、生产环境完全一致，避免了“在我电脑上能跑”的问题。
* 容器打包了应用及其依赖，部署时不再受制于操作系统或硬件环境。

 **资源高效利用**

* 相比传统虚拟机，Docker 容器更轻量，不需要额外的操作系统开销。
* 启动速度快，内存占用小，可以在同样的物理资源上运行更多实例。

 **易于扩展和维护**

* 配合容器编排工具（如 Kubernetes），可以方便地进行应用的扩展、负载均衡和自动恢复。
* 镜像版本管理清晰，回滚和更新非常方便。

 **持续集成与持续部署（CI/CD）支持**

* 容器非常适合自动化构建、测试、部署流程，提高开发效率，缩短交付周期。

 **隔离性与安全性**

* 每个容器相互隔离，故障不会轻易扩散。
* 可以结合权限控制和安全加固措施，增强整体系统的安全性。
* 容器一定比虚拟机占用资源少吗？

**容器通常资源占用更少，但**  
**并不是在所有情况下都绝对比虚拟机资源占用少。**

具体来说：

1. **正常情况下：容器资源占用更少**
   * 容器共享宿主机的操作系统内核，不需要像虚拟机那样每个实例都运行一套完整的操作系统。
   * 容器镜像通常更小，启动更快，内存、CPU开销更低。
   * 因此在**轻量应用、多实例部署、快速弹性伸缩**场景下，容器明显优于虚拟机。
2. **但在以下特殊情况下，容器并不一定更省资源**：
   * **高密度负载**：当单机容器数量极多时，如果调度、网络、存储管理不当，容器自身的调度开销（如cgroups、namespace管理）也会堆积，导致系统性能下降。
   * **需要强隔离和独立内核功能时**：虚拟机的隔离是硬件级别的，容器是内核级别的。有些场景（如不同内核版本需求、严苛安全要求）下必须使用虚拟机，即使多占资源。
   * **容器膨胀问题**：如果容器中运行的是重量级应用，或者镜像管理不规范（比如频繁堆积镜像层），也会导致占用增大，失去轻量优势。
3. **总结一句话**
   * **一般应用场景下，容器比虚拟机轻量。**
   * **极端高负载、强隔离需求场景下，虚拟机更适合。**

* 一台4核的虚拟机跑了2个2核的容器？容器用了什么技术（Linux相关）如何实现保证2核而不是3核的？

容器通过 **cgroups** 的 **cpuset**（硬绑核） 或 **cpu配额**（软限制）机制，确保容器**只能用固定的 CPU 核心数量或者使用率**，从而实现了“2核而不是3核”的资源保障。

**容器用到的核心技术（Linux相关）**

容器利用了 **Linux 的内核特性**来实现资源控制和隔离，主要涉及：

* **cgroups（Control Groups）**
  + 用来**限制、隔离、监控**容器使用的 CPU、内存、磁盘、网络资源。
  + 在 CPU 资源方面，cgroups 可以设置每个容器可以使用的 CPU 核心数、使用率配额等。
* **namespaces**
  + 用来**隔离**容器的进程、网络、挂载点、IPC等，但**资源限制主要靠 cgroups**。

**如何具体限制 CPU数量？**

在容器运行时，Docker（或底层的 containerd）通过调用 Linux 的 cgroups 来控制 CPU。

控制 CPU 的方式主要有两种：

* **绑核（CPU Affinity，cpuset）**
  + 用 cpuset 子系统，直接指定容器只能运行在哪几个具体的 CPU 核心上，比如 CPU0、CPU1。
  + docker run --cpuset-cpus="0,1" my\_container
  + 这样容器内的所有进程只会被调度到 CPU0 和 CPU1 上，不可能占用第3、4核。
* **CPU配额与周期（cpu.cfs\_quota\_us / cpu.cfs\_period\_us）**
  + 用 cpu 子系统，设置容器在单位时间内能使用多少 CPU 时间。
  + 比如：
    - cpu.cfs\_period\_us = 100000（100ms）
    - cpu.cfs\_quota\_us = 200000（允许200ms的时间量）
    - 那就相当于 2 核的 CPU 资源。
  + 示例命令：docker run --cpus="2.0" my\_container
  + 注意，这种方法是**软性限制**，允许进程跨核，但总计算能力受控。
* HTTP在网络分层的哪一层

**HTTP在网络分层的位置**

* 在 **OSI七层模型** 中，HTTP属于**应用层**（第七层，Application Layer）。
* 在 **TCP/IP四层模型** 中，HTTP同样属于**应用层**（Application Layer）。

HTTP是**应用层协议**，主要负责**数据的表示、编码、会话管理**，比如浏览器和服务器之间的网页传输。

它依赖于下层的\*\*传输层（TCP）\*\*来保证数据可靠传输，但本身不负责数据的传输控制。

* 网络分层的好处是什么

**网络分层的好处主要有以下几点：**

1. **简化设计与实现**
   * 每一层只关注自身的功能，比如传输层关注可靠传输，应用层关注数据内容，降低了设计复杂度。
2. **模块化、灵活性高**
   * 各层之间接口标准化，可以**独立开发、更新和优化**，互不干扰。
   * 例如可以改进传输层（如TCP优化）而不用修改应用层（如HTTP协议）。
3. **促进互操作性**
   * 采用统一标准，各厂家、各系统之间容易兼容和通信，比如不同品牌的设备也能互联。
4. **便于故障定位与调试**
   * 问题可以定位到具体层次，比如是应用层超时、传输层丢包，还是网络层路由问题，快速排查。
5. **支持标准化和推广**
   * 网络协议的发展可以遵循分层标准（如ISO、IETF标准化组织制定的协议），加速技术推广。

* HTTP长连接和短连接区别？

短连接开销大但简单；长连接提高性能但需要资源管理。选择依据是通信频率和系统设计需要。

 **HTTP短连接**（默认HTTP/1.0行为）

* 每次请求-响应完成后，**立刻关闭TCP连接**。
* 下次需要通信时，**重新建立TCP连接**（包含三次握手）。

 **HTTP长连接**（Keep-Alive机制，HTTP/1.1默认支持）

* 建立一次TCP连接后，**可以复用同一个连接发送多个请求-响应**。
* 减少了频繁建立和断开连接的开销，提高了通信效率。



**长连接实现方式**：在HTTP头部加上：Connection: keep-alive

**短连接实现方式**：显式指定：Connection: close

* 长连接虽然减少了TCP建立开销，但需要**服务器资源保持连接状态**，如果连接数过多，需要结合**连接池**、**超时机制**管理。
* 在HTTP/2中，长连接还进一步演进，引入了**多路复用**（同一连接并发多个请求）。
* 这个连接是指什么连接？

这里的“连接”指的是——

**TCP连接**，也就是**客户端和服务器之间建立的一条可靠的双向通信通道**。

具体来说：

* 当你发送HTTP请求时，浏览器（客户端）需要**通过TCP协议**，先和服务器建立一条稳定的通信链路（即TCP连接）。
* 这个过程需要经历**TCP三次握手**，保证双方可以正常、可靠地收发数据。

所以，不管是HTTP短连接还是长连接，  
它们的“连接”指的都是**底层的TCP连接**，而不是HTTP协议本身的某种连接。

因为HTTP协议本身是**基于TCP之上的应用层协议**，它**依赖TCP来传输数据**。

而TCP连接建立、维持、关闭都要消耗资源（比如CPU、内存、端口资源），  
这就是为什么长连接（保持一个TCP连接）可以减少开销，提高性能。

* TCP属于哪一层的连接

**1. 这个“连接”（TCP连接）属于哪一层？**

* 在**OSI七层模型**中，TCP连接属于**传输层**（Transport Layer，第4层）。
* 在**TCP/IP四层模型**中，也是属于**传输层**（Transport Layer）。

**2. 为什么属于传输层？**

* 传输层负责**端到端的通信**，也就是说，它要确保数据从客户端正确、可靠地传到服务器。
* **TCP协议**是传输层最典型的协议，它通过：
  + **三次握手建立连接**
  + **数据传输中的确认应答（ACK）**
  + **超时重传、流量控制、拥塞控制** 来保证数据的可靠、有序传输。



* 钉钉聊天 / 邮件收到 笔试题目

// 双线程交替打印1-100（奇偶分开）

// 数组拼凑最大数的字符串

/\*

输入：nums = [3, 10]

输出："310"

输入：nums = [5, 9, 30, 3, 34]

输出："9534330"

\*/

* Binlog配合MQ如何保证数据库和Redis一致性

方案1：先写缓存再写数据库（×）

有可能 更新缓存顺序正确，但更新数据库顺序错误

方案2：先写数据库再写缓存（×）

有可能 更新数据库顺序正确，但更新缓存顺序错误

方案3：先删缓存再写数据库（×）

有可能 数据库更新前，缓存被其他人再次写入了旧数据

方案4：先删缓存再写数据库再删缓存（缓存双删）

需要 保证第二次删除要在请求回写缓存之后（通过消息队列）

方案5：先写数据库再删缓存

有可能 第一次查询缓存得到旧数据

方案6：先写数据库再通过Binlog异步更新缓存

理由：高并发场景删除缓存可能并不合适

通常的做法是**基于Binlog订阅+异步消费更新Redis**，核心步骤如下：

1. **数据库作为唯一数据源（真源）**，正常业务操作只写数据库，不直接写Redis。
2. **开启Binlog监听**（比如用 Canal、Debezium等工具）：
   * 订阅数据库的Binlog（记录了数据变更，比如Insert/Update/Delete）。
   * 捕获到变更后，生成对应的消息推送到消息队列（MQ，比如Kafka、RocketMQ等）。
3. **下游消费者异步处理**：
   * 业务服务消费MQ消息，根据消息内容同步更新或失效Redis中的缓存。
4. **如何保证一致性？重点措施：**
   * **顺序消费**：同一条数据（如同一条主键ID的数据变更）必须按顺序处理，避免出现脏读。
   * **幂等性处理**：消费端对重复消费、消息重放具备幂等保护，防止数据错误。
   * **补偿机制**：如果消息消费失败，比如Redis更新失败，要有重试机制或者异常记录、人工兜底处理。
   * **及时失效策略**：如果有些变更难以及时更新，可以选择直接让对应Redis Key失效，强制下次从数据库重建缓存。

* 如果binlog特别多，消息堆积了怎么办？

遇到消息堆积，处理思路是分层的：

1. **短期缓解措施**（快速止血）：
   * **扩容消费者数量**：增加消费线程或消费实例数，提高消费速度。
   * **调整消费批量和并发度**：一次拉取更多消息，批量处理，提升吞吐量。
2. **中期优化措施**（系统调整）：
   * **优化Redis更新逻辑**：减少不必要的更新操作，比如只在真正变更时同步。
   * **热点数据优先处理**：可以对重要/热点数据优先消费，降低用户体验受损。
   * **使用分区（Partition）机制**：比如Kafka可以把数据按业务主键Hash分区，多个消费者并行拉取不同分区，加速处理。
3. **长期根本性优化**：
   * **Binlog筛选**：只订阅、处理真正需要同步到Redis的表和字段，减少冗余同步量。
   * **架构分层设计**：引入消息中间层（比如变更过滤服务），进一步缓冲高峰压力。
   * **异步一致性容忍**：允许小时间窗口内Redis与数据库存在短暂不一致，降低强同步压力。

* 运维方面，Linux命令如何查看Java进程

**ps -ef | grep java** 最通用，

* ps -ef 列出所有进程的详细信息。
* grep java 过滤出所有包含"java"字样的进程。

可以快速看到：

* Java进程的PID（进程ID）
* 启动参数（包括运行的JAR包、类名、JVM参数等）

**jps -l** 最快速精准，

 jps列出所有Java进程。

 -l参数会显示**完整的类名或者JAR包路径**，更容易定位是哪一个应用。

**top -c | grep java** 适合实时资源监控。

 -c选项可以显示完整命令行，帮助你快速识别各个Java进程的具体信息。

 可以看到CPU、内存使用率、PID。

**查到PID后进一步操作**

top -Hp <pid>

kill <pid>

kill -9 <pid>

* 说一说Linux的内核态用户态以及零拷贝

**1. 什么是内核态（Kernel Mode）和用户态（User Mode）？**

* **用户态（User Mode）**
  + 应用程序运行在用户态。
  + 在这个模式下，程序只能访问受限的内存区域，不能直接操作硬件，也不能直接调用内核代码。
  + 如果要访问硬件或者做敏感操作（如读写磁盘、网络收发），需要\*\*发起系统调用（System Call）\*\*请求内核来完成。
* **内核态（Kernel Mode）**
  + 操作系统内核运行在内核态。
  + 拥有**最高权限**，可以直接访问硬件、管理内存、控制进程调度等。
  + 当应用程序通过系统调用请求服务时，CPU会从用户态切换到内核态执行。

小图理解：

[用户态]

应用程序（如Java、Nginx、MySQL）

↓（系统调用）

[内核态]

Linux内核（管理文件系统、网络协议栈、磁盘IO等）

2. 什么是零拷贝（Zero-Copy）？

* **传统IO操作**（以文件发送到Socket为例）需要多次数据拷贝和态切换：
  1. 内核从磁盘读取数据到内核缓冲区。
  2. 用户态程序再从内核缓冲区拷贝数据到用户态缓冲区。
  3. 程序发起发送请求，又把数据从用户态缓冲区拷贝回内核缓冲区。
  4. 最后内核通过网络发送出去。

➔ 过程有**两次数据拷贝**、**两次态切换**，非常耗CPU和内存带宽。

* **零拷贝（Zero-Copy）**
  1. 核心目标：**避免不必要的数据拷贝和态切换，提高IO性能**。
  2. 典型机制如：
     + mmap（内存映射文件，应用直接访问内核缓冲区）
     + sendfile（直接在内核里把文件数据发到Socket，不经过用户态）
     + splice、vmsplice（高级零拷贝接口）

➔ 通过零拷贝技术，可以大大减少CPU开销，提高数据传输速度，尤其是大文件传输、网络服务中效果明显。

【自我介绍】

* 本科合工大计科，硕士东南软件，成绩5%
* 技术栈Java，熟悉Spring、MySQL、Redis
* 反思总结 单篇博客3w阅读，长期主义 微信读书1000小时
* 科研方向 虚拟现实人机交互 渲染+边缘计算

【英文介绍】

【简历项目】

分库分表

通过订单号和用户信息复合分片算法完成订单数据分库分表，支持订单号和用户查询维度。

* [用户分库分表](https://www.yuque.com/magestack/12306/pb98neetmww1rr9y)
* [乘车人分库分表](https://www.yuque.com/magestack/12306/zhsauz6ksng8wvgf)
* [订单分库分表](https://www.yuque.com/magestack/12306/dyr1d4r3me19gg7l)
* [分布式雪花算法](https://www.yuque.com/magestack/12306/ciigw9ctq0v90u3w)
* [手摸手实现分布式ID组件库](https://www.yuque.com/magestack/12306/lc2yb8gxtvfdt7rp)
* 1. 选择哪个字段作为分片键分库分表？

订单分库分表选择**订单号**和**用户ID**作为复合分片键。支持通过订单号和用户ID两个维度进行查询。

* 2. 基因算法保障分片键易用性：订单号字段同时支持订单号和用户ID查询

基因算法通过将用户ID的哈希值嵌入到订单号中，使订单号既包含订单的唯一标识，也包含用户ID的信息。

* 3. 系统支持用户名/手机/邮箱登录，无法确定分片键造成“读请求扩散”？

采用映射表将用户名/手机/邮箱与用户ID建立映射关系，存储在单独的索引表中。登录时先通过索引表查询到用户ID，再根据用户ID定位到具体的分片，从而避免全表扫描。

* 4. 分库分表后为什么选择雪花算法作为ID？
* **全局唯一**：时间戳 + 机器ID + 序列号
* **有序性**：ID按时间递增
* **高性能**：本地生成、速度快，无需依赖数据库或外部服务
* 5. 如何保障雪花算法在大规模集群下生成不重复？

确保每个节点的**机器ID**唯一：

* **集中分配**：使用Zookeeper或Redis集中分配和管理机器ID。
* **配置文件**：在部署时为每个节点配置唯一的机器ID。
* **启动检测**：在节点启动时检测机器ID是否冲突。

用户注册

封装缓存组件库避免注册用户时用户名全局唯一带来的缓存穿透问题，减轻数据库访问压力。

* [手摸手之注册用户如何防止缓存穿透？](https://www.yuque.com/magestack/12306/go6vg8whk9g1lyhp) 
  + [用户注册布隆过滤器容量设置以及碰撞率问题](https://www.yuque.com/magestack/12306/fr3zurztaq3xwfkc)
* [手摸手之实现敏感信息加密存储](https://www.yuque.com/magestack/12306/vhf4i3c604t2qex7)
  + [核心技术文档-如何防止用户敏感数据泄露](https://www.yuque.com/magestack/12306/cd9zbuugg663qsu4)
* [手摸手之用户敏感信息展示脱敏](https://www.yuque.com/magestack/12306/myl4gqx84bxyxmay)
* [如何防止用户敏感数据泄露](https://www.yuque.com/magestack/12306/cd9zbuugg663qsu4)
* 1. 如何解决判断用户名是否已被注册带来的缓存穿透？
* 缓存穿透

请求不存在于数据库的数据 造成 数据库负载、缓存内存耗尽、用户体验差

* 缓存穿透的常见方案

1. 对不存在的 Key 缓存并把值设为Null，设置短暂过期时间（如 60 秒）。  
   **缺点：**尝试但没注册一个不存在的用户名，该值60s内都不可被注册
2. 查询缓存不命中时使用分布式锁来保证只有一个线程访问数据库。  
   **缺点：**其他用户注册请求缓慢或超时
3. 布隆过滤器存已注册用户名，不在布隆过滤器的一定不存在-可用，如果用户名在布隆过滤器中再查询缓存或数据库，。  
   **缺点：**布隆过滤器不能删除元素，注销的用户名无法再次使用。
4. Redis Set存已注册用户名，检查是否在集合内。  
   **缺点：**占用内存。

* 12306解决注册穿透：布隆过滤器 + Redis Set缓存

1. 布隆过滤器不存在，说明数据库没有-可用
2. Redis Set缓存存在，说明已注销-可用
3. 查询数据库到底有没有

* Q：用户频繁申请后注销导致Redis Set。

Redis大Key问题是指在Redis中存在单个Key对应的Value数据量过大。

**A：**

1. 限制1个证件号最多注销5次
2. 对缓存进行分片处理，根据用户名的HashCode进行取模操作，将数据分散存储在多个Set结构中。

* Q：现在redis中有一个大key，如何平滑拆分？

**A：**

双写：大的redis key正常写入，同时还要写一个拆分后的。在某一个时机两边数据能够对齐的情况下，从大 Key迁移到新Key上。

* 2. 用户敏感数据（如手机号、证件号）存储到数据库如何脱敏？

Apache ShardingSphere是一个开源分布式数据库解决方案，其中的JDBC支持数据分片、分布式、读写分离、数据加密

1. 数据源配置
   1. 加密前，数据源（如druid或hikari）直接通过Spring加载到IOC 容器
   2. 加密后，由 ShardingSphere包装一层再交给 Spring 管理
   3. 原理：拦截、解析、改写、发送
2. 加密器配置
   1. ShardingSphere 中内置了 AES 和 MD5 两种加密算法
3. 脱敏表配置
   1. plainColumn明文列
   2. cipherColumn密文列
   3. logicColumn逻辑列：面向开发者，和明文以及密文保持映射关系
4. 查询属性配置
   1. 是查询明文列的数据直接返回
   2. 还是查询密文列再通过 ShardingSphere 解密后返回。

车票查询

* [**手摸手之车票搜索为什么用Redis而不是ES？**](https://www.yuque.com/magestack/12306/zd9wok8w0dn8eig5)
* [**手摸手之如何完成列车数据检索**](https://www.yuque.com/magestack/12306/tygc8hs113al2c2z)
* 众多查询条件如何满足？

TicketController#pageListTicketQuery

1. 责任链验证数据
   1. 空
   2. 出发日期≥当前日期，毕竟不可能买上一天的车票
   3. 出发地和目的地存在、出发站点和到达站点存在
2. 加载城市数据
   1. 搜索北京南到杭州东，列出北京到杭州所有的列车车次
   2. 通过站点关联到城市，通过城市查询列车
3. 数据结构  
   Key：Region\_train\_station\_起始城市\_终点城市\_日期  
   Value：  
    Key：列车ID\_起始站\_终点站  
    Val：列车详细信息
4. 单独存储余票信息  
   列车余票数据是实时变更的，如果在存储到基本信息中就没办法变更了
5. 构建返回数据  
   12306 列车查询页可知，会存在不同的查询条件，这些查询条件都是通过本次查询所有列车数据构建出来的。页面上的大部分查询条件（如车次类型、席别、出发车站等）由前端进行筛选采用构建者模式构建

* 车票购买
* 通过 Redis Lua 脚本原子特性，完成用户购票令牌分配，通过令牌限流以应对海量用户购票请求。
* 通过 Redis Lua 脚本原子特性，完成用户购票时票数检验、高铁座位分配以及扣减库存等功能。
* 使用 BinLog 配合 RocketMQ 消息队列完成 MySQL 数据库与 Redis 缓存之间的数据最终一致性。
* [**节假日高并发购票Redis能扛得住么？**](https://www.yuque.com/magestack/12306/eq6v9p9dfre117mg)
* [**手摸手之实现列车购票流程**](https://www.yuque.com/magestack/12306/nmmgqkgbfxb2bwl0)
  + [**手摸手之实现用户购票责任链验证**](https://www.yuque.com/magestack/12306/ggg2txzbfgfqp6tm)
  + [核心技术文档-从根上理解Redis分布式锁演进架构](https://www.yuque.com/magestack/12306/ag5pffwexihshe2s)
* [**手摸手之实现v2版本列车购票流程**](https://www.yuque.com/magestack/12306/ov3u6lpgartx0mst)（Lua脚本不太会）
  + [**缓存击穿之双重判定锁如何优化性能？**](https://www.yuque.com/magestack/12306/xrtg5mibquardvvi)（逻辑清晰、简明易懂）
  + [**高并发库存扣减为什么需要令牌限流？**](https://www.yuque.com/magestack/12306/gdqnbhp10xx9g9kv)
* [**购买列车中间站点余票如何更新？**](https://www.yuque.com/magestack/12306/efhhtogdr8ouz6t2)
  + [**缓存与数据库一致性如何解决？**](https://www.yuque.com/magestack/12306/wocbrht50ctg14nv)（加星，很详尽）
  + [**手摸手之列车余票如何保障缓存数据库一致性**](https://www.yuque.com/magestack/12306/glv5e0785b2d7oag)
* [**余票Binlog更新延迟问题如何解决？**](https://www.yuque.com/magestack/12306/kxuug8l6zslfyz21)
  + [**购买列车余票如何防止库存超卖？**](https://www.yuque.com/magestack/12306/pemih6p8h7xaw2b3)

买一张北京南到南京南的车票实际上会扣减北京南-南京南，北京南-济南西，济南西-南京南的三趟车票。如果其中有任意条件不满足都不会购买成功。

* 1. 如何限流？

通过**Redis Lua 脚本的原子特性**实现**令牌限流机制**：

* **令牌容器**：将未出售的座位作为令牌放入 Redis 容器中，用户购票时需要从容器中获取令牌，成功获取令牌的用户才能进入后续流程，否则直接返回失败。
* **Lua 脚本**：原子性地检查令牌容器中的令牌数量并扣减，确保不超卖。
* **分布式锁优化**：仅允许少量用户竞争锁，减少 Redis 的压力。
* 2. 如何落库？
* **订单创建**：通过分布式锁保障座位分配的唯一性，生成订单号（基因法）并创建订单记录。
* **延迟关闭订单**：RocketMQ的延时消息，实现订单超时未支付自动取消。
* **分库分表**：通过订单号和用户信息的复合分片算法，完成订单数据的分库分表，支持订单号和用户维度的查询。
* 3. 如何保障余票缓存和数据库一致性？
* **BinLog + RocketMQ**：通过 BinLog 监听 MySQL 的数据变更，将变更消息发送到 RocketMQ，由消费者更新 Redis 缓存，实现最终一致性。
* **Lua 脚本原子操作**：在购票时，通过 Lua 脚本原子性地完成余票校验、座位分配和库存扣减，确保缓存和数据库的数据一致性。
* **双重校验**：在扣减库存时，先检查缓存中的余票数量，再通过 Lua 脚本原子性地扣减，防止超卖。

支付超时

通过 RocketMQ 延时消息特性，完成用户购票 10 分钟后未支付情况下取消订单功能。

* [**订单延时关闭功能技术选型**](https://www.yuque.com/magestack/12306/sazb1bn2a1okwtlu)（最后选择RocketMQ实现）
* [**创建订单并支付后延时关闭订单消息怎么办？**](https://www.yuque.com/magestack/12306/ldw8nxp96yfg7cgx)
  + [手摸手之消息队列正确使用姿势](https://www.yuque.com/magestack/12306/dhm3lr598gtt23od)（看不大懂，要补）
* 1. 如何取消十分钟未支付的订单？

**RocketMQ延时消息**：

* 订单生成时，向 RocketMQ 发送一条延时消息，设置延时时间为十分钟。
* 十分钟后，RocketMQ 将消息投递给消费者，检查订单状态。
* 如果订单状态仍为未支付，则释放库存并将订单状态标记为已取消。
* 2. 如何避免已支付错误取消？

在消费者处理延时消息时进行**状态校验**：消费者接收到延时消息后，首先检查订单的支付状态。如果订单已支付，则直接忽略消息，不执行取消操作。

12306优化

* [缓存击穿之双重判定锁如何优化性能？](https://www.yuque.com/magestack/12306/xrtg5mibquardvvi)（逻辑清晰、简明易懂）
* [12306核心接口性能优化都做了什么？](https://www.yuque.com/magestack/12306/bh7c4x3i4sn682bn)（有点难噢）
* 缓存击穿之双重判定锁优化性能

1. **缓存击穿**：热点数据缓存失效的瞬间，大量并发请求直接访问数据库，导致数据库压力骤增。
2. **双重判定锁**：

* 第一次检查缓存，若数据不存在，获取分布式锁。
* 获取锁后再次检查缓存，若数据仍不存在，才访问数据库并更新缓存。

1. **Redis Lua脚本：**保证原子性操作，避免锁竞争和数据不一致问题。

* 12306核心接口性能优化

1. **全面使用缓存**：用NoSQL数据库取代传统数据库
2. **队列削峰**：构建交易处理排队系统，接收用户下单请求并异步处理
3. **分库分表**：将订单/电子客票数据从1个节点拆分为3个节点30个库30张表，提升处理能力并支持横向扩展。
4. **读写分离**：用NoSQL数据库存储订单数据，避免高并发查询影响交易。
5. **异地双活架构**：构建双活数据中心，支持高峰期弹性扩容。

Nacos

服务注册发现、配置集中管理

1. 引入依赖nacos-discovery、nacos-config
2. Application.yml配置server-address、file-extension
3. 服务发现@FeignClient(name = “xx”)
4. 动态刷新@RefreshScope

延迟消息

Msg.setDelayTimeLevel

他山之石

美团二面

链接：<https://www.bilibili.com/video/BV1Dn4y1X7aD?vd_source=7341e06df79108c355b8df461a20c071>

* 背景

选择开源项目练手，模仿12306完成了核心功能。

* 五大模块

1. 网关模块：JWT令牌校验、请求路由转发
2. 用户模块：登陆注册、增删改查
3. \*购票模块：查票、购票
4. 订单模块
5. 支付模块

* 难点：查询，南京-北京，需要展示南京南-北京x的一系列

解决：地区映射表，每个地区对应哪些车站，放入缓存

* 难点：购票时的参数校验

解决：责任链设计模式

1. 非空判断
2. 购票日期小于当前
3. 是否和已有车票冲突

* Q：车票时间冲突如何解决？

A：针对用户建立bitmap，用户一天24h，以10min为单位， 用户购买一张票后，举例说0点到1点，就需要将0-5这6个比特位填充。再次下单时候，查看有无重叠。以10min为单位的情况下：

* 一个用户一天144bit = 18B
* 日活跃用户2kw = 2 x 104
* 相乘得到300+ MB

Redis部署了很多集群，为了给用户更高的响应速度，内存消耗可以接受。

* Q：如果有一亿用户，如何快速检索用户对应的bitmap？

A：针对车次建立bitmap，看用户下单了哪些车次

* Q：如果存长文本/复杂文本，比如头条场景/论文查重率，如何解决重复校验问题？

A：瓦片算法通过滑动窗口分组哈希将长文本转换为哈希集合，结合Jaccard相似度计算，核心优势：

1. ​抗局部修改：重叠窗口设计可容忍部分内容变动。
2. ​高效性：哈希化与MinHash压缩大幅减少数据量。
3. ​可扩展性：适合分布式处理，应对海量文本查重。

* 难点：传统秒杀可以Redis库存扣减，12306提前15天放票均匀分布

解决：

* 对于非节假日情况，数据库+分布式锁，保证同一用户同一时间只下一单；
  + 分布式锁的粒度优化
    - 如果根据train id设置锁key，会导致同一辆车次只会让一个用户操作
    - 优化，多拼接一个座位类型（商务座、一等座、二等座）
    - 优化，a->b->c，a->b和b->c可以并发，受到MySQL间隙锁启发，间隙有重叠再进行阻塞
  + 十万个请求争夺分布式锁 -> Redis的big key问题
    - 微服务有很多JVM实例，比如5个，通过网管模块进行负载均衡，每个实例2w个请求，先竞争本地的Reentrantlock锁，再竞争分布式锁，因此最终只有5个线程竞争分布式锁
* 对于节假日抢票，TOKEN令牌限流，1比1映射票数不保证强一致性，取到令牌才可以购票，没票设置为-1挡住后续无效请求
  + Q：为什么获取TOKEN不需要强一致性？
  + A：Redis中还有余票缓存，这个是保证强一致的。Token令牌是隐式的，用户看不见，是用于限流的，所以保有一定冗余
    - 令牌有、余票无：拿到Token还是会进行兜底查询的
    - 令牌0：查数据库、更新Redis缓存。如果还是没有，设置为-1，打回无效请求
  + Q：余票缓存和数据库如何保持一致？
  + A：canal监听binlog发给MQ消费消息实现。
    - 保证强一致性的情况：购票后删除Redis缓存，会过于频繁而且还是要查数据库
    - 保证最终一致的情况：canal监听binlog，会存在1-2s的延迟，但是可以接受。12306官网并不会因为我下单一张票就立即扣减
* 项目独立思考部分

锁粒度降低

车票时间冲突

1.3w行源码

* 认知提升思路
* 改良已有项目
* 理清楚已有代码
* 针对具体模块，根据git提交记录，了解具体职能和与调用关系
* 交流探讨代码风格相关
* 反问

Q：实习生会接收到哪些任务？

A：

（1）代码测试，辅助提升性能质量。

（2）在已有项目迭代需求

Q：我的面试哪里可以改进。美团是我第一个比较正式的面试，其他面试也就问了30min，拿了offer不太想去。

【简历八股】

MySQL

B+树索引的查询过程？为什么不用哈希索引？

B+树索引的查询过程：​

从**根**节点自上而下逐层比较确定下一层子节点的位置，最终定位到**叶子**节点

如果是范围查询，叶子节点通过链表连接，可以高效地遍历满足条件的数据。

为什么不用哈希索引？

B+树**有序 范围查询**、哈希索引**随机分布 等值查询**

分库分表后全局ID生成方案有哪些？

1. ​UUID：无序。  
   String uuid = UUID.randomUUID().toString();  
   Generated UUID: e3e70682-f3a1-11e9-8f0b-362b9e155667
2. ​Snowflake：时间戳 + 机器ID + 序列号。 时钟回拨可能引发问题。  
   0 | 41-bit 时间戳 | 10-bit 机器 ID | 12-bit 序列号
3. 数据库自增ID：依赖数据库。  
   
4. ​Redis自增ID：存在单点故障风险。  
   
5. ​号段模式​：从数据库预分配一批ID号段，本地缓存使用。号段用尽需重新申请，可能短暂阻塞。

主从同步延迟的解决方案？



事务隔离级别和MVCC的实现原理？

**事务隔离级别：​**

1. ​读未提交：最低级别
2. ​读已提交：解决脏读
3. ​可重复读​：默认级别，解决不可重复读
4. ​串行化：最高级别，解决幻读

**MVCC实现原理：​**

1. ​根据事务ID和版本时间戳判断数据可见性
2. ​写操作生成新版本，旧版本仍可被读取
3. ​定期清理旧数据，释放空间

死锁的产生条件及排查方法

死锁产生四大条件：​

1. ​互斥：资源一次只能被一个事务占用。
2. ​持等：事务持有资源的同时，等待其他资源。
3. ​不夺：已分配的资源不能被强制剥夺，只能由持有者释放。
4. ​循等：事务之间形成资源等待的环路。

**排查方法：​**

1. ​SHOW ENGINE INNODB STATUS;
2. ​查看LATEST DETECTED DEADLOCK：

* 查看涉事务ID
* 分析事务持有的锁和等待的锁
* 定位造成循环等待的SQL语句

1. ​调整锁顺序、事务粒度、重试机制

覆盖索引和索引下推的区别？

覆盖索引**：​**查询字段都包含在索引中，无需回表

索引下推**：​**在引擎层利用索引过滤数据，减少回表，适用于WHERE包含索引和非索引的查询



Redo Log和Undo Log的作用及写入机制？

**Redo Log**持久性、崩溃恢复：将**修改**写Buffer刷**磁盘**

**Undo Log**原子性、MVCC读一致性、回滚：将**旧数据**写**段**中刷磁盘

Change Buffer对写操作的优化原理？

如果数据库直接修改磁盘上的索引页，那么需要：将索引页从磁盘加载到内存，在内存中修改数据，最后写回磁盘。

当修改非唯一 二级索引时，先将修改记录到Change Buffer中，当相关索引页被加载到内存时，再将Change Buffer中的修改合并到索引页。

适用于写多读少

Redis

Redis单线程模型为什么能高效处理请求？

1. ​**内存**
2. ​**跳表**
3. **单线程 事件驱动模型 + I/O多路复用**

缓存穿透/击穿/雪崩的解决方案？

缓存穿透不存在：查询一个不存在的数据，缓存中没有相应的记录，每次请求都会去数据库查询，造成数据库负载激增。

* 布隆过滤器
* 缓存不存在标识

Redis缓存击穿热点：某个热点数据在缓存中过期，导致大量请求同时访问数据库，造成数据库负载激增。

* 互斥锁
* 热点数据永不过期

Redis 缓存雪崩多个：多个缓存数据在同一时间过期，导致大量请求同时访问数据库，造成数据库负载激增。

* 双缓存策略：主缓存设置短过期时间，备份缓存设置较长过期时间
* 随机过期时间

RDB和AOF的优缺点及混合持久化机制？

RDB快照：更快、文件体积小（便于传输）

AOF追加写：更完整、文本格式（便于人工检查修复）

Redis7混合持久化机制：先用RDB生成快照，然后追加AOF日志

Redis Cluster的slot分配算法？

Redis Cluster将整个键空间划分为16384个哈希槽，每个槽对应一个范围。

分配算法**：​**

1. ​对每个键计算CRC16哈希值，然后对16384取模，得到对应的slot编号。
   1. **适配65536的1/4，让哈希值均匀映射到槽**
   2. 优化心跳包的网络传输（16384b大约2KB）在节点之间传递关于哈希槽分配的信息时，用较少的比特位就可以表示一个哈希槽的状态
2. ​每个节点负责一部分slot，节点增减时重新分配，确保负载均衡。

大Key问题如何定位和解决？

1. **拆分**
2. gzip**压缩**
3. 冷数据存储到**磁盘**

热Key问题的解决方案？

1. ​**拆分**到多个节点：通过哈希算法或引入随机前缀。
2. **缓存**：CDN或本地缓存
3. ​**读写分离**：将读请求分散到从节点，减轻主节点压力。
4. ​**限流降级**

布隆过滤器的实现原理及误差率计算？

布隆过滤器是一个初始全0的m位数组、k个哈希函数、n个元素映射置1。

查询元素时任意位置为0一定不存在



Redis事务和Lua脚本的原子性区别？

Redis事务

* 通过**MULTI、EXEC、DISCARD**批量执行
* 事务执行期间其他客户端命令可能插入执行
* 如果某条命令失败后续命令仍会执行，**不支持回滚**

Lua脚本​

* 通过EVAL或EVALSHA真正完全原子
* 执行期间其他客户端命令会被阻塞
* 如果Lua 脚本中有错误直接抛出错误不会执行，**隐式“回滚”**

Java并发

CAS的原理是什么？存在哪些缺陷？

**CAS（Compare And Swap）**由 CPU 硬件指令保证原子性的无锁并发​：

比较当前值（V）与预期值（E），相等则将新值（N）写入，否则不操作。

1. ​ABA 问题：值从 A 变为 B 再变回 A，CAS 无法感知中间变化

* ​解决：版本号或时间戳

1. ​自旋开销：如果 CAS 失败，线程会不断自旋重试、消耗 CPU

* ​解决：限制自旋次数。

1. ​单变量限制：无法直接支持多个变量的原子操作。

* ​解决：封装为复合操作。

AQS的核心机制是什么？举一个基于AQS实现的工具类。

AQS（Abstract Queued Synchronizer）抽象队列锁

1. ​通过 state表示**资源状态（ReentrantLock通过 state 记录锁的重入次数）**
2. 在​CLH 队列中，线程按请求顺序排队，后面请求锁的线程会自旋等待，直到它前面的线程释放锁，保证公平性。线程通过 acquire() 和 release() 方法进入或离开队列。提供 tryAcquire() 和 tryRelease() 等模板方法，由子类实现具体逻辑。

ConcurrentHashMap如何实现线程安全？Java7和Java8的区别？

​

Java7 分段锁（默认16个段）：锁粒度大（段级别），性能低

Java8 CAS + synchronized：锁粒度小（桶级别），性能高

synchronized和ReentrantLock的性能对比？

**synchronized**：

JVM内置优化，适合低竞争

隐式锁 自动获取释放

**ReentrantLock**：

基于AQS，适合高竞争

显式锁 手动获取释放

支持可中断锁、尝试锁等特性

SpringBoot / SSM

Spring AOP的实现原理？JDK动态代理和CGLIB的区别？

Spring AOP​通过动态代理实现

* JDK动态代理​
  + 基于接口、通过 Proxy 类生成代理对象
  + 无法代理未实现接口的类
* CGLIB代理​
  + 基于继承、通过字节码生成子类代理
  + 无法代理 final 类或方法

如何解决循环依赖问题？三级缓存机制详解。

循环依赖**：**两个或多个Bean相互依赖，导致无法完成初始化。

三级缓存**：**

1. ​一级缓存（Singleton Objects Map 单例对象映射表）：  
   存放已经实例化、属性注入、初始化的 Bean。
2. 二级缓存（Early Singleton Objects Map 早期单例对象映射表）：  
   存放已实例化，但未完成属性注入和初始化的 Bean。  
   防止 AOP每次调用ObjectFactory.getObject()都产生新代理对象。
3. 三级缓存（Singleton Factories Map 单例工厂映射表）：  
   ObjectFactory的getObject()调用getEarlyBeanReference()创建早期Bean。

解决循环依赖的流程**：**

1. ​**Bean实例化**：

* 创建Bean实例，但未注入属性和初始化。
* 将Bean工厂对象（ObjectFactory）放入三级缓存。

1. ​**属性注入**：

* 发现依赖的Bean，尝试从一级缓存获取。
* 如果一级缓存不存在，从二级缓存获取。
* 如果二级缓存不存在，从三级缓存获取工厂对象，将这个 ObjectFactory 从三级缓存中移除，生成的Bean并放入二级缓存。

1. ​**Bean初始化**：

* 完成属性注入和初始化。
* 将Bean从二级缓存移除，放入一级缓存。

Spring Bean的生命周期和作用域有哪些？

**Bean生命周期：**

1. ​实例化
2. ​属性赋值
3. ​初始化

* ​**前置处理**postProcessBeforeInitialization()。
* ​Initialization**()**：
  + @PostConstruct注解方法
  + InitializingBean的afterPropertiesSet()
  + init-method
* ​**后置处理**postProcessAfterInitialization()

1. ​使用
2. ​销毁

* @PreDestroy注解方法
* DisposableBean的destroy()
* destroy-method

**Bean的作用域：**

Singleton 默认单例

Prototype 每次请求创建一个新的Bean实例。

Request 每个HTTP请求创建一个Bean实例

Session 每个HTTP会话创建一个Bean实例

Application 每个ServletContext生命周期内创建一个Bean实例

WebSocket 每个WebSocket会话创建一个Bean实例

SpringMVC请求处理流程？

1. DispatcherServlet接收请求
2. HandlerMapping根据请求URL找到对应Controller
3. HandlerAdapter适配并调用处理器、通过handle()执行业务逻辑
4. Controller 执行业务逻辑，返回ModelAndView或数据对象
5. HandlerAdapter将ModelAndView或数据对象转换为适合的格式。
6. ViewResolver视图解析器解析
7. DispatcherServlet将渲染视图或JSON数据返回给客户端。

SpringBoot自动配置的实现？

1. ​@SpringBootApplication =   
    @SpringBootConfiguration

+ @EnableAutoConfiguration // 关键：开启自动配置

+ @ComponentScan  
Spring Boot启动时ImportSelector加载spring.factories 中的自动配置类

1. ​条件匹配@Conditional：  
   根据类路径、Bean是否存在判断是否加载配置类。
   1. @ConditionalOnClass
   2. @ConditionalOnMissingBean

Spring事务失效的常见场景有哪些？

1. ​异常不匹配  
   默认只回滚RuntimeException和Error，如果抛出比如IOException且未配置rollbackFor，不会回滚。
2. ​异常未抛出
3. ​同类方法调用  
   同一个类中，方法A调用方法B（B有@Transactional），实际调用的是原始对象的方法，而不是代理对象的方法，而事务管理是通过代理机制实现的，B的事务不会生效。
4. ​非public方法、final方法、static 方法
5. ​事务基于ThreadLocal实现，多线程事务无法传递。
6. ​其他
   * **事务传播行为**PROPAGATION\_NOT\_SUPPORTED会挂起事务
   * 未正确配置数据源DataSourceTransactionManager
   * MyISAM引擎本身不支持事务

MyBatis的#和$的区别及SQL注入问题。

* #：预编译、更安全
* $：直接拼接、存在SQL注入风险

动态表名、排序字段必须用$：

* 因为表名、字段名、ORDER BY 排序字段不能使用 ? 作为参数绑定。
* 只有值可以用 # 作为参数占位符。

严格控制用户输入，确保只能传递合法的表名和字段名

* 白名单校验
* 使用 Enum 限制可选字段

SpringBoot如何集成Tomcat容器？

默认内嵌Tomcat容器

* **端口**：8080
* **最大1w连接**：有1w客户端可以TCP连接到服务器
* **最大200线程：**服务器可以同时处理200个请求

计算机网络

TCP三次握手的详细过程和状态变化？

三次握手

* 第一次: 客户端发出同步请求报文。
* 第二次：服务器发出同步确认报文。
* 第三次: 客户端发出确认报文。



四次挥手

* 第一次：客户端发出释放报文。
* 第二次：服务器发出确认报文。
* 第三次：服务器发出释放确认报文。
* 第四次：客户端发出确认报文。



**TIME\_WAIT状态的作用：​**

1. 防止旧连接的延迟数据包干扰新连接。
2. 保证对端收到最后的ACK，避免连接关闭异常。

**TIME\_WAIT过多的解决方案：​**

1. 开启TCP 连接复用：sudo sysctl -w net.ipv4.tcp\_tw\_reuse=1
2. 增加可用端口：sudo sysctl -w net.ipv4.ip\_local\_port\_range="32768 60999"
3. 使用连接池复用连接
4. 减少短连接的使用

Sudo: Superuser Do

Sysctl: System Control

HTTPS和HTTP有哪些区别？

**1**. 端口号

* **HTTP**：80
* **HTTPS**：443

2. 加密

* **HTTP**
* **HTTPS**：通过SSL/TLS协议加密

HTTPS工作原理是什么？它是如何实现数据加密的？

SSL/TLS: 对称加密、非对称加密、哈希算法的结合

1. 客户端Hello：SSL/TLS版本、**加密算法列表**、**随机数**
2. 服务器Hello：SSL/TLS版本、**加密算法、随机数**
3. 服务器发送**公钥和数字证书**
4. 客户端验证证书，并生成会话密钥使用服务器公钥加密发送
5. 服务器使用私钥解密获得**会话密钥**。
6. 交换握手消息摘要、开始加密通信。

TCP快速重传和超时重传的区别？

* 快速重传：收到3个重复ACK快速重传，适用部分包丢失但网络较好
* 超时重传：未收到ACK且计时器超时重传，适用严重丢包或连接中断

TCP拥塞控制？



MSS：**Maximum Segment Size**

Ssthresh：**slow start threshold**

HTTP1.1新特性？

1. 持久连接keep-alive复用同一连接（串行请求响应）

2. 支持请求管道化：同时发送多个请求

3. 更细粒度的缓存控制：新增Cache-Control头部

* no-cache：必须向服务器验证后才能使用缓存。
* no-store：不允许缓存（敏感数据）。
* max-age=60：缓存内容在 60 秒内有效。
* public：响应可以被任何缓存（比如代理服务器）保存。
* private：响应只能被单个用户缓存，不能被共享缓存（代理）保存。

4. Host头部：允许同一IP托管多个域名  
这样即使同一个服务器只有一个 IP 地址，它也可以根据 Host 头部区分请求，托管多个不同的网站（比如 example.com 和 test.com 都指向同一个 IP，但服务器能正确响应不同内容）。

5. 新增错误代码：410资源已永久删除

HTTP2新特性？

1. 多路复用 应用层消除队头阻塞（并行请求响应）

2. 二进制协议 以帧传输

* **1.1基于文本**

3. 独立流量控制

* **1.1依赖TCP的拥塞控制**

4. HPACK算法头部压缩

* **1.1请求头冗长，尤其对于重复的请求头，如Cookies等**

5. 服务器主动推送

HTTP3新特性？

1. 基于QUIC协议

* **1和 2**依赖TCP协议
* Quick UDP Internet Connections是Google提出，建立在UDP上

2. 零RTT连接建立

* 往返时延（Round-Trip Time, RTT）
* **1和2**基于TCP的三次握手
* **3**首次请求时可以立即开始数据传输

3. 传输层 消除队头阻塞

* **2**虽然支持多路复用，但仍然受TCP队头阻塞的影响：当一个流的数据包丢失时，整个连接的延迟会增加，影响其他流的传输。
* QUIC基于UDP、采用多路复用 + 独立流量控制和快速重传

4.内建TLS加密

操作系统

进程和线程的区别？协程的优势是什么？

进程：

* 资源分配基本单位
* 进程创建和切换开销大：有独立的代码、数据、文件等，
* 进程之间是相互独立的，需要**进程间通信机制**（管道、消息队列、共享内存、套接字）

线程：

* CPU调度基本单位
* 线程创建和切换开销小：组内共享资源，但有独立的PC和堆栈，
* 线程直接读写内存即可，但需要**同步机制**

Select、poll、epoll的区别？

1. ​**select**：

* 使用 file descriptor set ，支持的文件描述符数量1024。
* 每次调用需要遍历所有文件描述符，效率低。时间复杂度：O(n)。

1. ​**poll**：

* 使用 ​pollfd 数组，支持的文件描述符数量无限。
* 每次调用需要遍历所有文件描述符，效率低。时间复杂度：O(n)。

1. ​**epoll**：

* 使用事件驱动机制，只关注活跃的文件描述符，效率高。
* 支持边缘触发（ET）和水平触发（LT）模式。时间复杂度：O(1)。  
  边缘触发模式只在状态变化时通知，需一次性处理所有数据，否则会丢失事件。适用高并发网络编程。

用户态和内核态切换的开销来源？

1. ​寄存器、程序计数器状态
2. ​权限**切换**
3. ​缓存**失效**
4. ​系统调用需要内核验证
5. ​切换可能伴随中断处理

进程间通信 共享内存和消息队列的对比？

**消息队列**：

* ​内核维护消息队列，进程发送/接收消息通信
* 自带同步机制
* ​有数据拷贝 速度慢，适用小数据量异步通信

**共享内存**：

* ​多个进程映射同一块物理内存
* 需要同步机制（如信号量）避免竞争。
* ​无数据拷贝 速度快，适用高性能大数据量通信。

零拷贝技术（sendfile）的实现原理？

零拷贝技术（sendfile）：

在文件传输中，让内核直接将文件数据从磁盘读取到网络缓冲区，无需经过用户态，避免数据在用户态和内核态之间的拷贝，提升性能。

* 传统方式：磁盘 -> 内核缓冲区 -> 用户缓冲区 -> 网络缓冲区
* sendfile ：磁盘 -> 网络缓冲区

内存映射（memory map, mmap）系统调用的实现原理？

mmap系统调用将文件/设备映射到进程的虚拟地址空间，实现文件读写或共享内存。减少用户态和内核态的数据拷贝，提高性能。

* ​**创建映射**：内核在进程的虚拟地址空间中分配一段区域，建立虚拟内存与文件/设备的映射关系。
* ​**延迟加载**：映射时不加载文件内容，通过缺页中断按需加载。
* ​**读写操作**：进程通过指针直接访问映射区域，内核负责将数据同步到文件或设备。

虚拟内存的作用？

虚拟内存：扩展、隔离、管理、共享

* ​允许程序使用比物理内存更大的地址空间
* ​每个进程拥有独立的虚拟地址空间，提高安全性
* ​简化内存分配和回收，支持动态内存需求
* ​多个进程可以共享同一块内存区域

Java集合

详细描述HashMap的扩容机制（触发条件、rehash过程）。

**Java7扩容条件：**

元素数量超过阈值 && 发生哈希冲突

**Java8扩容条件：**

1. 情况一：元素数量超过阈值
2. 情况二：链表长度≥8 && 数组长度＜64，优先扩容而不是转为红黑树

Rehash过程**：**

1. 新建一个2倍大小的数组
2. 只需部分移动：if e.hash & oldCap==0
   1. 0接在loTail后面
   2. 1接在hiTail后面

基础：

数组长度是2的次方，且扩容为2倍。

oldCap：

位运算代替取模提高效率、保证hash均匀分布：hash & (数组长度-1)

旧长16即010000，16-1=15即001111

新长32即100000，32-1=31即011111

rehashing时用**旧长16**即010000作为一个Mask看对应该位是否为1

源码**if e.hash & oldCap==0**（0接在loTail后面、1接在hiTail后面）

HashMap底层结构是什么？如何解决哈希冲突？

**HashMap底层结构**是数组+链表，Java8后引入红黑树。

当链表长度≥8且 数组容量≥64时，链表会转换为红黑树  
（链表长度≤8时，链表性能足够且内存占用低）

当红黑树节点数≤6时，红黑树会退化为链表

**解决哈希冲突**：

1. **拉链法**
2. **红黑树**
3. **二次扰动函数：**如果哈希值低位变化较小，计算索引时，低位相同的键会映射到同一位置。**将哈希值高位低位异或、优化索引分布、减少哈希冲突**
4. **动态扩容机制：**
   1. 超过阈值（容量×负载因子，默认0.75）扩容2倍并重新计算索引。
   2. 0.75是空间利用与哈希冲突的最佳平衡  
      负载因子过高会增加哈希冲突  
      **负载因子**过低则导致频繁扩容

为什么HashMap线程不安全？举例说明并发问题场景。

**HashMap线程不安全的原因：**

数组数据覆盖、头插环形链表、红黑树树化结构异常

**解决方案：**

1. Collections.synchronizedMap(new HashMap<>())：在方法级别加锁，性能低
2. ConcurrentHashMap：采用分段锁（7）或CAS+ synchronized（8），性能高。

HashMap的遍历方式有哪几种？哪种效率更高？

1. ​entrySet() 效率最高、推荐使用。
2. forEach()+ Lambda 底层实现基于entrySet()。
3. ​keySet() 返回所有键的集合，通过 map.get(key)获取值
4. values() 返回所有值的集合。由于无法通过值获取键

for (Map.Entry<String, Integer> entry : map.entrySet()) {

String key = entry.getKey();

Integer value = entry.getValue();

System.out.println(key + ": " + value);

}

map.forEach((key, value) -> {

System.out.println(key + ": " + value);

});

Linux命令

如何用grep查找包含"error"的日志并统计次数？

* grep "error" logfile.log：查找 logfile.log 中包含 "error" 的所有行
* grep -c "error" logfile.log：统计 logfile.log 中包含 "error" 的行数

awk如何实现按列求和？

awk '{sum+=$3} END{print sum}' filename

* {sum+=$3}：逐行累加第3列的值。
* END{print sum}：处理完所有行后，输出累加结果。

如何用sed批量替换文件中的字符串？

sed -i 's/old\_string/new\_string/g' filename

* sed：Stream Editor 流编辑器
* -i：in-place，直接原地修改文件、不输出到终端
* s：**替换命令**，格式为 s/查找内容/替换内容/
* g ：全局替换

strace和perf工具的作用及使用场景？

strace：跟踪进程系统调用

strace -p <pid> 监视进程 <pid> 的所有系统调用。

perf：分析 CPU 性能瓶颈、函数调用链、硬件事件

perf top 实时查看占用 CPU 最高的函数

如何用tcpdump抓取指定端口的SYN包？

tcpdump -i eth0 'tcp[tcpflags] & tcp-syn != 0 and port 80'

* -i eth0：指定网卡接口
* tcp[tcpflags] & tcp-syn != 0：匹配 SYN标志位
* port 80：指定端口

【手撕代码】

本地环境

文件夹放桌面

直接Main旁边运行

Junit

Comparator

泛型

十大排序

【反问环节】