【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。

2025.04.17荣耀2h（a2.5）

2025.05.07华为2h（a0.95+1+0.95）

2025.03.15美团AI面（40min）

* TCP与UDP的区别及使用场景

**TCP**：连接 可靠、按字节、文件传输

**UDP**：无连接 不可靠、按报文、实时通信

* Linux查看文件的命令？列出至少三条

cat整个显示

less逐页显示：按空格键向下翻页、按b键向上翻页

head -n 5 example.txt查看文件前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. Atomicity原子性：要么全部执行成功，要么全部不执行
2. Consistency一致性：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. Isolation隔离性：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. Durability持久性：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equals认为相等，那么hashCode必须返回相同哈希值

HashMap、HashSet、Hashtable

依赖于hashCode来确定对象的位置

依赖于 equals来判断是否相等。

* 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：让AI决策过程更加透明和可解释

* 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
2. 文档、教程、社区、书籍
3. 动手实践、渐进式学习
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
* 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？
* HashMap源码及其扩容机制？

**底层结构**是数组+链表，Java8后引入红黑树。

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。还有当哈希冲突链表长度≥8且数组长度＜64，优先扩容而不是转为红黑树。

**Rehash**

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. **负载因子过小会频繁扩容**
2. **HashMap并非线程安全**，多线程扩容可能导致环形链表死循环

* （追问）HashMap和ConcurrentHashMap的区别？

Java 7采用Segment分段锁分成16个段，不同的线程可以同时访问不同的段。

Java 8采用 CAS + synchronized，修改时首先尝试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？

**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**
3. ​封装为复合操作 解决 **无法支持多变量的原子操作**

* SpringAOP原理

AOP面向切面编程：**切面 = 切点 + 通知**

* **切面：**封装了权限、日志通用功能
* **切点：**想要织入的方法
* **通知：**拦截后**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，通过反射机制实现
* 适合实现接口的的类

**CGLIB动态代理（**SpringBoot**）**

* 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（后来回滚）的数据
* **不可重复读：**同一事务读取两次，结果不同
* **幻读：**同一事务读取两次，结果集记录数量不同

**事务隔离级别：**

* **读未提交：**
* **读已提交：**解决脏读
* **（默认）可重复读**：解决不可重复读  
  MySQL InnoDB通过间隙锁（Next-Key Lock）解决幻读：不仅锁住已有数据，还锁住数据之间的间隙，以防止插入新数据。
* **串行化**：解决幻读
* 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
* 一种分法是，保持表结构相同，水平拆分数据行
* 另一种分法，根据业务，垂直拆分字段

需求：

* 用户要能查看自己的订单
* 支持订单号精准查询

选择订单号和用户ID作为复合分片键

每次查询时需要带着用户和订单两个字段，非常不方便：

采用基因法把 用户 ID 后六位 冗余到 订单号 里（tb订单）

按用户 ID 后六位分库分表：

分 2 个库order\_0和order\_1，以及对应业务 16 张表

分片键定义为用户 ID 和订单号：shardingColumns: user\_id,order\_sn、

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，还是需要根据索引来查找具体的记录。

只要查询中携带这两个字段，我们就取用户 ID 后六位进行查找分片表的位置：

* （追问）实际如何通过用户ID找到订单？

由于按照后六位分片，所以同一用户所有订单都在一个分片里，不会导致读扩散问题（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**自定义了分片算法**：

1. 如果SQL包含用户 ID直接取用户 ID 后六位
2. 不包含就从订单号中获取后六位，也就是用户 ID 后六位

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：   
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**

**② 表示层：**对数据编码、加密、压缩、转换

**③ 会话层**

**④ 传输层TCP协议：**端到端的可靠透明数据传输

**⑤ 网络层IP协议：**逻辑地址寻址、路由选择

**⑥ 数据链路层**

**⑦ 物理层**

TCP/IP网络四层模型

1. **应用层：**FTP、DNS
2. **传输层**
3. **网际层**
4. **网络接口层**

* TCP和UDP的区别？

TCP（Transmission Control Protocol）

面向连接 三次握手四次挥手、字节流、文件传输

UDP（User Datagram Protocol）

面向无连接 更为轻量，但不保证可靠性、数据包、视频直播

* 进程和线程的区别？

进程：资源分配基本单位

进程有独立的代码、数据、文件，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：CPU调度基本单位

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 类重写 run() 方法

2. 实现 Runnable 接口的 run() 方法，作为参数传给Thread

3. 借助 FutureTask包装 Callable 对象，作为参数传给Thread

4. 通过 ExecutorService 创建线程池，提交任务到线程池执行

* 讲一讲项目
* 为什么要分页排序？分页排序如何实现？

1. **一次性获取所有数据消耗大量的内存和时间**
2. **方便用户浏览和定位**

* （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？

当分页到比如1k页以上时，会出现**深度分页查询**，导致：

1. 需要扫描大量的数据页才能定位到指定页的数据
2. 临时结果集占用过多内存，使用磁盘存储数据，慢

**优化方案：**

1. 覆盖索引避免回表：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。
2. WHERE子句过滤数据实现基于主键的游标分页：使用上一页的最后一条记录的主键ID作为下一页查询的条件。

优化数据库的缓存大小  
在应用层使用Redis缓存   
分库分表减少单表数据量

* 外卖平台的定时任务是什么任务？（区分12306的延时消息）

**Spring Task实现定时任务**

* 超时未支付订单自动取消
  1. 待支付 && 下单时间 < 当前时间-15min
  2. @Scheduled(cron = "0 \* \* \* \* ?")
* 自动完成长时间处于派送中的订单  
  （比如tb确认收货，用户已收货但未确认）
  1. 派送中 && 派送开始时间 < 当前时间-60min
  2. 凌晨1点是为了避开日志、备份等系统维护任务
  3. @Scheduled(cron = "0 0 1 \* \* ?")

1. 启动类@EnableScheduling开启定时任务
2. 任务方法添加@Scheduled，结合Cron表达式定义执行时间  
   秒 分 时 日 月 周 [年]

**Spring Task潜在问题和解决**

1. **默认单线程执行，任务阻塞**
   1. 配置线程池 ThreadPoolTaskScheduler
   2. 添加 @Async 注解使用异步执行
2. **异常未处理导致任务停止**
   1. try-catch
   2. 实现 ErrorHandler 接口，统一处理定时任务中的异常
3. **不支持分布式环境，任务重复执行**
   1. 分布式锁

引入分布式任务调度框架，如 Quartz、XXL-Job

* AI信息？平时学习AI的工具使用？

1. Deepseek清华大学104页PPT
   1. 推理模型（DeepSeek-R1）：数学推导、逻辑分析
   2. 通用模型（GPT-4）：文本生成、创意写作
2. 腾讯云黑客松Copilot、Cursor：  
   在调试思路、架构构建、文档归纳提供智能辅助、拓展技术边界
3. 知识掌握得比人多、需求掌握得比人少：
   1. 学习提示词工程、调整提问方式改变输出质量
   2. 配合自我判断和工程思维使用AI

* （追问）大模型幻觉是什么？为什么会造成幻觉？

大模型幻觉：虚构信息、捏造细节（医疗、法律领域敏感）

原因：

* + 过拟合学习了错误数据噪声
  + 概率生成：倾向于高频出现的答案，即使它们不正确
  + 提示词不够具体，导致模型脑补
  + 缺乏真实世界反馈

解决：

* + 数据清洗
  + 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
  + 提示工程
* 手撕代码

[**7. 整数反转**](https://leetcode.cn/problems/reverse-integer/)

将数字num逆序输出，注意符号和边界

输入：123

输出：321

输入：-456

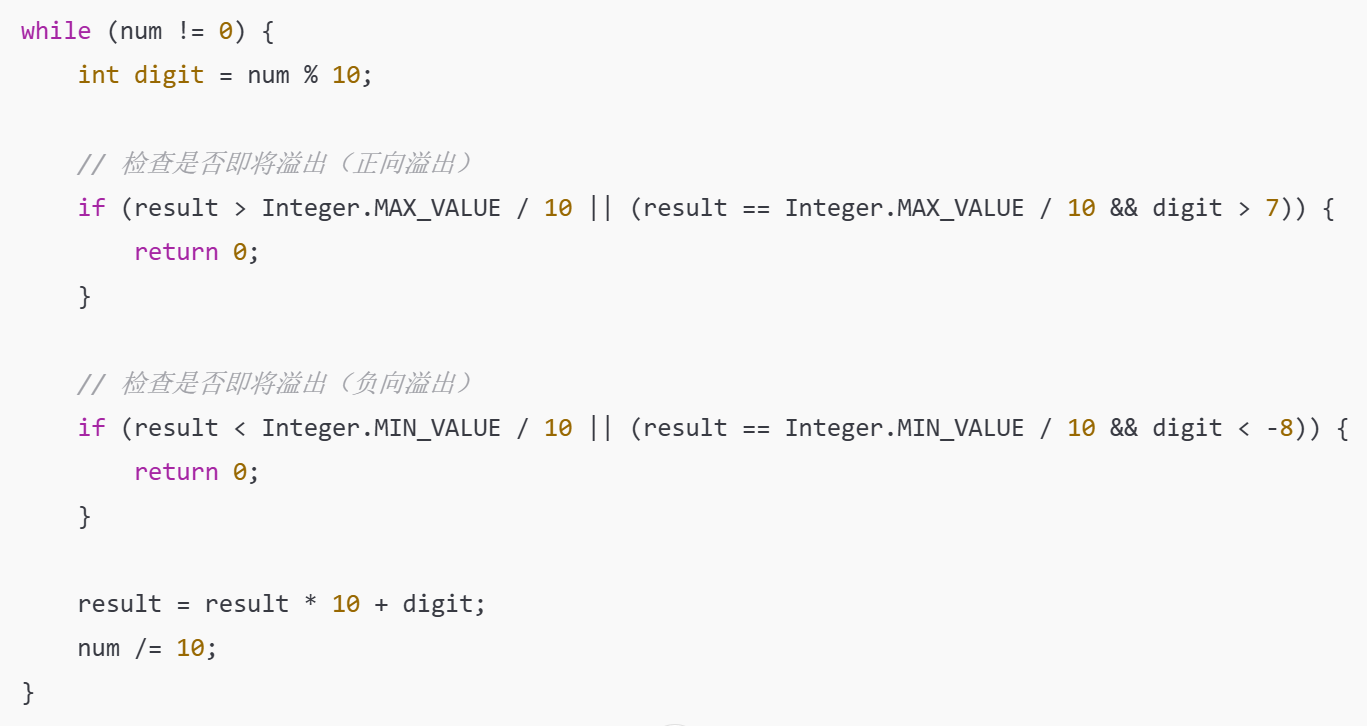
输出：-654

输入：214647483638

输出：0

Integer.MAX\_VALUE = 2147483647

Integer.MIN\_VALUE = -2147483648



* 【反问】业务内容Base地点和HC分布？

北京 核心本地商业（以前的到家到店合并）-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
* 12306 开源项目 二开 是吧？
* 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】
* （追问）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。

* （追问）获取令牌是按什么获取？
* （追问）用户下单购票时候逻辑是怎么样的？
* （追问）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：缓存双删？

极端场景：缓存余票为0，但数据库还有库存，通过当缓存余票为0时用定时任务检查是否一致

潜在风险：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：通过定时任务频繁刷新 性能消耗不可取。使用Canal Binlog 的方式更加优雅且高效。

令牌容器架构：

在用户购票先从令牌容器中扣减令牌，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。

？遇到上述问题时，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？
* 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？

前端倒计时显示：前端定时器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。

* 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
* 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

* 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 = Spring + SpringMVC + MyBatis

**SpringMVC 的请求处理流程**

1. **请求**：客户端发起请求，到达DispatcherServlet
2. **（拦截）映射、适配处理、返回（拦截）**
   * **HandlerInterceptor#preHandle()拦截**：返回true继续处理，用于权限、日志、异常、性能
   * **HandlerMapping映射**：根据请求的 URL、HTTP 方法映射合适的Controller
   * **HandlerAdapter适配处理**：用HandlerAdapter调用Controller处理
   * **返回**
     + 返回 **ModelAndView 对象（数据模型、视图名称）**
     + 返回 JSON 对象。
   * **HandlerInterceptor#postHandle()拦截**
3. **解析、渲染、响应**
   * **ViewResolver解析**
     + 对于视图：将逻辑视图名称解析为实际的视图（如 JSP、Thymeleaf ）
     + 对于 JSON ：将对象序列化为 JSON返回给客户端
   * **视图渲染引擎渲染**：根据 Model 中的数据渲染HTML页面
   * **响应**：将渲染的视图或 JSON 数据响应给客户端。

**MyBatis 的数据持久化：**通过 MyBatis 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 来增删改查。

* 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
* 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

1. **环境一致性**：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。，避免了“在我电脑上能跑”的问题。
2. **隔离性与安全性：**每个容器相互隔离，故障不会轻易扩散。
3. **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
4. **易于扩展和维护**：配合Kubernetes容器编排工具进行应用扩展、负载均衡
5. **CI/CD支持**：自动化构建、测试、部署流程

* 容器一定比虚拟机占用资源少吗？

**一般应用场景下，容器比虚拟机轻量。**

1. 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。
2. 容器镜像通常更小，启动更快，内存、CPU开销更低。

**极端高负载、强隔离需求场景下，虚拟机更适合**

1. **高密度负载**：当单机容器数量极多时，容器自身的调度开销（如cgroups、namespace管理）也会堆积
2. **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. **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。不同内核版本需求、严苛安全要求下必须使用虚拟机

* 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？

容器利用了 **Linux 的内核特性**来实现资源控制和隔离：

* **cgroups（Control Groups）限制、隔离、监控**CPU、内存、网络资源
* **namespaces**用来**隔离**容器的进程、网络、挂载点、IPC等

控制 CPU 的方式主要有两种：

1. **绑核cpuset**
   1. docker run --cpuset-cpus="0,1" my\_container
   2. 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。
2. **CPU配额与周期cpu.cfs**：设置容器在单位时间内能使用多少 CPU 时间
   1. cpu.cfs\_period\_us = 100000（100ms）
   2. cpu.cfs\_quota\_us = 200000
   3. 允许200ms的时间量，那就相当于 2 核的 CPU 资源。

* HTTP在网络分层的哪一层

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的网页传输。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

1. **解耦、模块化：**每一层只关注自身的功能：传输层关注可靠传输，应用层关注数据内容。各层之间接口标准化，可以独立开发、更新和优化，互不干扰。故障调试可以定位到应用层超时、传输层丢包，还是网络层路由问题
2. **标准化、兼容、互联：**采用统一标准，各厂家容易兼容和通信，不同品牌设备也能互联

* HTTP长连接和短连接区别？

1. HTTP短连接（1.0 Connection: close）：每次请求-响应完成后，立刻关闭TCP连接
2. HTTP长连接（1.1 Connection: keep-alive）：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。
3. 在HTTP/2中，引入了多路复用，同一连接并发多个请求

这个连接是指传输层的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，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。

* 钉钉/邮件 手撕：双线程交替打印1-100（奇偶分开）

Thread最传统的写法：

* new Runnable() {} 创建的**匿名内部类**
* 重写 run() 方法
* 作为参数传给给 Thread

Java 8新特性：Lambda表达式可以替代函数式接口的实例

* Lambda的基本结构：(参数列表) -> { 方法体 }
* 函数式接口Runnable只有一个抽象方法 run()
* // 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
* 如果binlog特别多，消息堆积了怎么办？

1. **短期缓解措施**：
   * 增加消费线程或消费实例数
   * 一次拉取更多消息批量处理
2. **中期优化措施**：
   * 热点数据优先消费
   * 使用分区机制：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**：
   * 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。
   * 引入消息中间层（比如变更过滤服务）缓冲高峰压力。
   * 允许小时间窗口内短暂不一致

* 运维方面，Linux命令如何查看Java进程

1. **ps -ef | grep java** 通用，进程ID、JAR包、类名、JVM参数
2. **jps -l** 快速精准显示完整的类名或者JAR包路径
3. **top -c | grep java** 实时资源监控CPU、内存使用率、PID

**查到PID后进一步操作：**

1. top -Hp <pid>：查看某个进程下所有线程的资源占用情况
2. kill <pid>：发送信号 15（SIGTERM）。先执行钩子函数（比如保存数据、关闭连接等）
3. kill -9 <pid>：强制终止

* 说一说Linux的内核态用户态以及零拷贝

**用户态**：访问受限的内存区域，不能直接操作硬件、不能直接调用内核代码。如果要访问硬件、管理内存、控制进程，需要发起系统调用请求内核来完成

**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。

**传统IO操作**需要多次数据拷贝和态切换：磁盘、内核缓冲区、用户态缓冲区、内核缓冲区、通过网络发送

**零拷贝（Zero-Copy）：避免不必要的数据拷贝和态切换，提高IO性能**。

1. **mmap：**内存映射文件，应用直接访问内核缓冲区
2. **sendfile：**直接在内核里把文件数据发到Socket，不经过用户态
3. splice、vmsplice：高级零拷贝接口

2025.04.10联想一面（30min）

* 自我介绍
* 你对于测试岗位和行业如何看待

**测试岗位在软件开发生命周期中**

发现Bug 、**质量保障**、**用户体验守护**、**业务价值交付**

* **测试设计能力**：理解需求，设计合理的测试用例和场景。
* **测试开发能力**：能够开发自动化测试脚本，提高测试效率。
* **协作与沟通**：与开发、产品、运维等多部门高效沟通，共同推动问题解决。

**测试行业现状与发展趋势**

* **测试左移、测试右移**：测试已经从开发完成后的验证，提前到了设计阶段（左移），也延伸到了上线后的监控与反馈（右移）。
* **自动化测试、AI+测试**：自动化程度不断提高，回归测试、接口测试、性能测试越来越依赖自动化体系。越来越多企业探索用机器学习、智能算法辅助测试生成、测试优化、缺陷预测。
* **DevOps和测试工程**：测试不再是孤立环节，而是DevOps闭环中持续集成、持续交付的重要一环，强调测试的持续性和自动化。
* 对Git提交使用流程
* 场景：如何设计测试用例测试网页登录功能？思路？从功能角度，小白用该网站登录？

**（1）基本功能测试**

**（2）输入边界与异常值测试**

* 空提交
* 超长
* 特殊字符（!@#$%^&\*）输入
* SQL注入测试（比如输入 ' OR '1'='1）

**（3）安全性测试**

* 多次连续登录失败后（比如5次），账户是否锁定或者触发验证码验证
* 登录接口是否有防止暴力破解（如加时间间隔、IP限制）
* 登录过程中密码是否明文传输？（应加密，如HTTPS）
* 前端页面密码框是否禁止复制/粘贴（根据业务要求）

**（4）兼容性测试**

* 不同浏览器（Chrome、Firefox、Safari、Edge）测试
* 不同设备（PC端、手机端自适应页面）测试
* 浏览器禁止Cookie/JavaScript时，是否正常提示或处理

**（5）易用性测试（小白用户角度）**

* 登录失败后，错误提示是否清楚易懂（比如“用户名或密码错误”，而不是晦涩代码）
* 忘记密码？注册链接是否明显可见
* 是否支持“记住我”选项？勾选后，下次是否自动填充
* 是否有合理的Loading提示？防止慢网速时小白误操作多次点击
* UI测试自动化工具和实际尝试
* **选择工具要看项目特点**：
  + Web系统首选Selenium/Playwright；移动端首选Appium；
  + 若是资源有限的小项目，Katalon Studio也能快速起步。
* 定期更新脚本、脚本结构化设计（POM模式）降低维护成本
* **自动化的ROI评估**：
  + 自动化回归测试适合高频、重复、变化少的功能；
  + 复杂的初期投入换取后期节省人力，不能盲目全自动化。
* 什么场景适合自动化测试？什么场景适合手工测试验证？

新功能 变化大 不适合自动化

旧功能 会不会有影响 适合自动化

* 作为测试人员发现bug提交给研发，研发认为这个场景不常见不算bug，你该如何沟通？

1. 冷静看待：确认自己立场

作为测试人员，发现问题并客观记录问题是职责。

2. 沟通准备：补充必要信息

* 在正式沟通前，我会准备好：
* 清晰的问题描述
* 出现问题的具体步骤、环境、数据
* 出现频率（是必现？偶发？）
* 具体影响评估
* 如果小概率出现，会导致什么？比如：数据错误？页面异常？安全隐患？
* 用户视角解释
* 站在终端用户角度，说明即便是小概率，也可能造成信任感下降或体验损伤。
* 参考标准或规范
* 如果有产品需求文档、设计稿、行业标准（比如电商支付安全、登录异常处理规范）可以引用，支撑自己Bug定义的合理性。

3. 沟通方式

使用合作型、专业型话术，而不是对抗型语言。

4. 如果研发同意是Bug → 正常跟进修复流程。

如果研发仍坚持不算Bug →向测试leader或项目经理同步此问题，让产品经理/项目经理参与判断。

可以将问题标记为“Known Issue”，在测试报告/发布说明中列明，做到记录留痕。

* 以往独立开发系统功能，经历软件的整个生命周期，包括需求设计、框架开发、最终验证，你能讲一讲对整个过程的理解吗？

**1. 软件生命周期**

1. 需求分析
2. 系统设计
3. 开发实现
4. 测试验证
5. 部署发布
6. 运维与优化

**2. 结合我以往独立开发经历具体讲每一阶段**

**（1）需求分析阶段**

* 不仅仅是"做功能"，而是理解"解决什么问题"。
* 《需求规格说明书》（SRS），记录功能范围、非功能性要求、异常场景等。

**（2）系统设计阶段**

**（3）开发实现阶段**

* **模块化开发**：
  + 按照设计文档开发，每个功能点独立实现并单元测试。
* **版本管理**：
  + 通过Git进行分支管理和提交记录，保持每次提交可回溯。

**（4）测试验证阶段**

**（5）部署发布阶段**

* **准备部署方案**：服务器选型、网络配置、备份机制。
* **灰度发布**：先小范围内测，确认稳定后再大规模上线。
* **上线回滚机制**：确保出现问题可以快速回滚，降低上线风险。

**（6）运维与优化阶段**

* **日志监控**：搭建日志收集系统，实时监控异常。
* **性能优化**：发现某些查询慢，后续增加缓存、SQL优化。
* **版本迭代**：根据用户反馈持续打磨功能，比如增加筛选条件、报警通知细化等。
* ChatGPT、Deepseek，谈谈你对AI的使用和未来发展的理解？
* 对软件工程开发、测试有一定了解，相对其他土木工程建筑区别是什么，测试在软件工程里属于哪个环节？

**1. 软件工程 vs 土木建筑工程的核心区别**

| **维度** | **软件工程** | **土木工程/建筑工程** |
| --- | --- | --- |
| 产品特性 | 无形、可持续演进（代码、逻辑系统） | 有形、一次成型（建筑物、桥梁） |
| 变更成本 | 变更便捷，开发后期仍可快速调整 | 变更代价极高，一旦施工完成，改动非常困难 |
| 开发节奏 | 快速迭代，版本持续更新 | 严格分阶段推进，每阶段完结后较难回头 |
| 测试验证 | 可以频繁回归测试，自动化覆盖，左移/右移测试 | 需要在建材、结构完成后做物理验收检测 |
| 市场驱动力 | 快速响应市场需求，先上线、后优化，占领市场是优先级 | 主要以安全、耐久性为底线，遵循严格标准、法规 |
| 失败后果 | 小规模失败可快速修复（软件补丁、热更新） | 结构性失误代价极大，甚至危及人身安全 |

**2. 现代软件工程开发模式特点（敏捷开发、DevOps、快速迭代）**

* **敏捷开发（Agile Development）**
  + 小步迭代，需求可以动态调整。
  + 强调**快速交付可工作的产品（Working Software）**，而不是长周期大交付。
  + 每一轮迭代（Sprint）都以用户价值为中心，快速验证市场反馈。
* **DevOps理念**
  + 将开发（Development）和运维（Operations）打通，形成一条\*\*持续集成、持续交付（CI/CD）\*\*的快速流水线。
  + 让代码变更可以**快速上线、快速回滚**，提高系统发布频率和稳定性。
* **快速占领市场的重要性**
  + 在激烈竞争环境下，谁能**更快上线、验证市场、占领用户心智**，谁就能取得先发优势。
  + 完美主义要让位于**快速迭代+持续优化**，不断通过版本更新打磨产品。
* 模块上线后，出现重大问题，无论如何你也复现不出来，但是用户又很生气你怎么做？

**1. 第一步：冷静面对，重视用户反馈**

**2. 第二步：系统性展开调查（即使无法直接复现）**

**（1）信息收集**

* 详细询问用户操作步骤、数据输入、具体页面、时间节点。
* 询问设备环境（系统版本、浏览器、客户端版本等）。
* 获取错误截图、录屏（如果有）或日志文件。
* 确认网络状况（比如是否弱网、断网情况）。

**（2）后台检查**

* 查看服务器日志（按用户ID/时间戳精确排查）。
* 检查相关模块的异常报警记录（Apm监控、错误率监控）。
* 检查CDN缓存、第三方服务、配置中心变化等外部依赖情况。
* 核查版本发布日志，是否有灰度、蓝绿切换等环境异动。

**（3）模拟复现**

* 多环境测试（不同系统、浏览器、网络状态下尝试）
* 输入相同参数、使用相同操作路径进行复测
* 异常场景测试（断网、异常中断、边界值数据）

**3. 第三步：即使无法复现，也要给出可执行的处理方案**

* **提交调查报告**

整理目前已排查信息，列出可能原因、已排除项、后续跟踪计划。

* **升级保护措施**
  + 增加日志埋点、增加客户端异常捕获机制
  + 增强容错处理（比如即使数据异常，也不会直接崩溃）
* **临时兜底**
  + 例如异常出现时，自动重试、回滚、给用户友好提示，而不是直接报错退出。

**4. 第四步：持续沟通与信任修复**

在调查过程中，保持透明沟通：

* 每天或每阶段定时更新给用户当前调查进度（即使没有新进展也要同步）
* 告知已做的改进措施，让用户知道系统在持续变好
* 如果确实短期无法定位，要真诚道歉，同时承诺：
  + 增加异常监控
  + 优先保障后续稳定性
  + 持续追踪直到彻底解决

**后续动作**：

* 将此次事件归档，纳入**问题库和复盘总结**。
* 补充相关测试用例，防止未来同类问题再次出现。
* 联想测开介绍

联想 PC 智能体客户端

写文章、翻译、电脑性能分析优化（清理）

性能测试、功能测试

联想拯救者 售后客服 快速定位问题（改进：自动化配置和关机，重新建立连接）

ThinkBook14+ （软件下载、F区 AI智能体、联想电脑管家）

联想电脑管家 开机加速也有 是否加入开机启动，做得智能化、人性化 降温除尘

2025.05.12饿了么一面（30min）

* 天楼 饿了么零售商家
* 自我介绍
* 项目介绍
* 项目架构
* Synchronized和锁的区别
* CAS
* ConcurrentHashMap
* HashMap什么时候转红黑树？为什么需要红黑树？
* HashMap长度为什么是2^n？（扩容相关）
* MySQL日志、RedoLog和UndoLog？（Binlog、数据一致性）
* 平时了解新技术？
* 大模型？检索增强生成？智能体（Coze、火山引擎、Dify）？
* 职业规划？管理 / 架构 / 码农？
* 入职时间？实习多久？
* 反问

上海：薪酬、工作氛围较好

饿了么 即时零售 + 餐饮外卖

部门负责商家、客户，零售 商家端 （1）基础账号（2）AI经营助手

2025.05.13饿了么二面（50min）

* 苍平 零售业务
* 自我介绍
* 今天就当聊聊天
* “长期主义”有哪些实践？出发点？保研？英语单词量？
* 读书哪些书？印象较深的书？讲的什么内容？
* 美团成功的因素？

1. **高效的即时配送体系**

美团构建了庞大的即时配送网络，拥有超过336万的月均有单骑手数量，日订单量峰值达到9800万单。这一体系确保了快速、稳定的配送服务，提升了用户体验和平台粘性。

2. **多元化的本地生活服务生态**

美团不仅提供外卖服务，还涵盖了到店餐饮、酒店预订、旅游、闪购等多个领域，形成了一个全面的本地生活服务平台。这种多元化战略增强了用户粘性，提升了平台的盈利能力和抗风险能力。

3. **精细化的运营与技术驱动**

美团通过大数据和人工智能技术优化运营，包括智能派单、路径规划和用户推荐等，提升了运营效率和用户满意度。此外，美团还注重骑手管理和服务质量，增强了平台的整体竞争力。

* 怎么看京东入局外卖行业？

1. **战略防御与业务延伸**

面对美团等竞争对手在即时零售领域的扩张，京东选择主动出击，进军外卖市场，以防御其核心电商业务受到侵蚀。通过外卖业务，京东不仅可以拓展高频消费场景，还能促进用户活跃度，带动其他品类的销售，实现业务的协同发展。

2. **差异化定位与品质服务**

京东外卖主打“品质外卖”，重点招募连锁品牌和堂食餐厅，强调食品安全和服务质量。此外，京东为全职骑手缴纳五险一金，提升了配送服务的稳定性和专业性。这种差异化策略有助于吸引注重品质的消费者，形成与美团、饿了么的竞争优势。

3. **价格策略与市场冲击**

京东推出“全年免佣金”政策，并投入百亿级别的补贴，吸引商家和用户入驻。这一激进的价格策略对现有平台构成压力，迫使其调整佣金政策和服务策略，可能引发行业的价格战，重塑市场格局。

* 即时零售品类

应急：医药、日用品

保鲜：酒水、鲜花

其他：衣服

* 分库分表索引设计？
* 分片后怎么解决查询问题？有哪些查询场景？
* 查某一时间段内的订单如何解决？
* 外卖平台/用户购票下单整个流程/服务调用链路？

1. HTTP防止重复提交——幂等性处理
2. 责任链模式——过滤
3. 获取分布式锁——扣减余票

* （追问）去重表里面什么内容？消息的哈希值？消息具体是什么？
* （追问）分布式锁怎么设计
* （追问）为什么用Lua？为什么不用Redis的事务？
* Java的8个基本类型
* Java什么时候会把对象分配在堆内存以外的地方
* （追问）临时变量都是存在栈上的吗
* 介绍一下反射的底层原理？怎么获取的？能做哪些处理？
* 反问

即时配送：

餐饮：增速趋缓

零售：即时、电商、线下

AI赋能：

ToC

ToB（大模型）：

* 经营状况分析
* 供给调整
* 商品信息规格调整

2025.05.16华为一面技术面（40min）

共享屏幕、本地IDE手撕代码

提前下载WeMeeting

会议号会在准点时在签到的网页中刷新，“进入会议”按钮可以跳转WeMeeting

* 你也是在会议室里
* 自我介绍
* Java和虚拟现实
* 就业考虑虚拟现实方向吗？国内机会多吗？
* 虚拟现实对终端设备有要求吗？
* 虚拟现实对视力有影响吗？
* 项目介绍？实习or 自学
* 购票是怎么实现高并发效果？令牌限流用什么开源框架了吗？
* 令牌是在内存的操作还是在缓存的操作？
* 你有看过美团、淘宝关于并发处理实现的技术吗？
* 优化后有没有测试并发量
* 微服务组件 业界常用架构
* 购票分段问题
* 其他NoSQL数据库？
* Redis只做一些数据存储相关工作吗？
* Redis可靠性保证
* RocketMQ的其他应用场景？什么情况下需要消息队列？
* 手撕：岛屿数量

2025.05.16华为二面主管面（30min）

* xiadingyin
* 考研保研？保研比例？学硕专硕？几年毕业？

18人

* 项目介绍？是否上线？实习经历？
* 说说分量高/影响力大的奖项？
* 工作方向虚拟现实？
* 什么途径获得华为的实习信息的？
* 项目学习过程中比较有影响力的事情？
* 最近在学一些什么新东西？
* 学习工作做项目中，挫折失败的经历？
* 聊不了解华为的价值观和文化？对华为的印象？
* 部门工作地点有没有什么特殊要求？
* 反问

终端-终端云服务-应用生态（应用市场）

终端：手机、电脑

终端云服务：后端服务、能力、解决方案

ToC：

1. 质量功能
2. 分发个性化
3. 商业变现

涉及高并发、高下载

对部门、对公司的印象：

大平台、10-20万人、规范

包含东西多、机会多、技术、自研

【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