1. 唯品会一面，被问麻了.....

1. 说一下Java的数据类型有哪些?
2. list和set实现类有哪些?
3. arraylist和linkedlist的区别?
4. hashset加入元素的过程，描述一下
5. hashmap线程安全吗？为什么不安全?
6. 怎么做到让hashmap线程安全?
7. concurrenthashmap怎么保证线程安全的?
8. 手撕生产者消费者模型
9. 手撕两个线程抢票代码，有没有什么其他方式保证线程安全?
10. Volitile关键字的作用有哪些?
11. atomic包用过吗?
12. 索引是什么？为什么能提高查询效率?
13. 在数据库中，我要线程安全的修改一个表里的某一列数据，要怎么做?
14. 写一下修改数据的操作语句
15. 联合索引是什么？什么是最左匹配原则?
16. 如何判断查询是否走了索引?
17. 索引失效的场景?
18. Binlog，redo，undo日志分别有什么作用?
19. springboot的启动原理?
20. springboot的核心注解有哪些？核心配置文件呢?
21. 讲一下实习的开发项目，流程是怎样的?
22. 实习开发中遇到了哪些困难？怎么解决的?
23. 愿意提前来实习吗?
24. 反问
    1. 唯品会这边的Java开发主要负责什么业务?
    2. 来了有导师带吗?
    3. 对我的评价

作者：子夏2024

1. Java的数据类型

Java的数据类型包括基本数据类型和引用数据类型：

- 基本数据类型：byte, short, int, long, float, double, char, boolean。
- 引用数据类型：class, interface, array。

2. list和set实现类

- List实现类：ArrayList, LinkedList, Vector。
- Set实现类：HashSet, TreeSet, LinkedHashSet。

3. ArrayList和LinkedList的区别

`ArrayList`和`LinkedList`都是Java集合框架中的实现类，它们分别基于数组和链表的数据结构。以下是它们之间的一些主要区别：

1. **底层数据结构：**
   - `ArrayList`使用动态数组实现。它的内部是一个数组，当数组容量不足时，会自动进行扩容。
   - `LinkedList`使用双向链表实现。每个元素都包含一个指向前一个元素和一个指向后一个元素的引用。
2. **随机访问性能：**
   - `ArrayList`支持快速的随机访问，因为它是基于数组的，可以通过索引直接访问元素。
   - `LinkedList`在随机访问时性能较差，因为必须从链表的头部或尾部开始遍历，直到找到目标元素。
3. **插入和删除操作性能：**
   - `ArrayList`在中间插入或删除元素时性能较差，因为需要移动数组中的元素。
   - `LinkedList`在插入和删除元素时性能较好，因为只需要改变相邻元素的引用。
4. **空间复杂度：**
   - `ArrayList`相对较省空间，因为它只需要存储元素值和数组容量。
   - `LinkedList`相对较耗费空间，因为每个元素都需要额外的两个引用字段。
5. **迭代器性能：**
   - `ArrayList`上的迭代器性能较好，因为它可以通过索引直接访问元素。
   - `LinkedList`上的迭代器性能较差，因为必须沿着链表一个一个地移动。
6. **适用场景：**
   - 如果需要频繁进行随机访问，使用`ArrayList`更为合适。
   - 如果需要频繁进行插入和删除操作，特别是在集合的中间位置，使用`LinkedList`更为合适。

选择使用哪个取决于具体的使用场景和操作需求。如果不确定，通常来说，`ArrayList`是一个更通用的选择，因为它在大多数常见的操作上都表现得很好。

4. HashSet加入元素的过程

`HashSet` 是基于哈希表实现的无序集合，它使用哈希算法来存储和检索元素。下面是向 `HashSet` 中加入元素的过程：

1. **计算哈希码（Hash Code）：**
   - 当你向 `HashSet` 中添加一个元素时，首先会调用该元素的 `hashCode()` 方法，得到元素的哈希码。
   - 如果元素为 `null`，则它的哈希码为 0。
2. **映射到桶位置（Bucket Position）：**
   - 哈希码经过一系列的变换和运算，被映射到哈希表中的一个桶位置（bucket position）。

- 桶位置是一个数组索引，表示存储元素的位置。

3. **处理哈希冲突：**
   - 哈希表可能存在冲突，即不同元素映射到相同的桶位置。为了解决冲突，`HashSet` 使用链表或红黑树（在JDK 8之后）来存储相同桶位置上的元素。
   - 如果桶位置上已经有一个元素，新元素会被添加到链表或红黑树的末尾。
4. **检查元素唯一性：**
   - 在添加元素的过程中，`HashSet` 会通过调用元素的 `equals()` 方法来检查元素的唯一性。
   - 如果已经存在相同的元素（根据 `equals()` 判断），新元素不会被加入。

`HashSet` 的添加过程通过哈希码和哈希表的桶来实现，确保元素的快速存储和检索。因为哈希表的桶位置是通过哈希码计算得到的，所以元素的存储位置在理想情况下是均匀分布的。这有助于在大多数情况下实现 O(1) 时间复杂度的添加、删除和查找操作。

5. HashMap线程安全吗？为什么不安全？

`HashMap` 在多线程环境下不是线程安全的。这是因为 `HashMap` 的实现是基于哈希表的，而哈希表的操作涉及到多个步骤，包括计算哈希码、定位桶位置、插入或检索元素等。在多线程环境下，多个线程同时对 `HashMap` 进行修改操作可能导致数据不一致或者丢失。

以下是一些可能导致线程不安全的情况：

1. **竞态条件（Race Condition）：** 多个线程同时尝试插入或删除元素时，可能导致竞态条件。两个线程可能同时检测到某个位置为空，然后都尝试插入元素，导致其中一个线程的操作被覆盖。
2. **扩容操作：** 当 `HashMap` 需要扩容时，会创建一个新的数组并将旧的元素重新分配到新数组中。在这个过程中，如果有其他线程同时对 `HashMap` 进行修改，可能会导致元素在扩容过程中丢失或者被重复添加。

为了在多线程环境下保证线程安全，可以使用 `ConcurrentHashMap` 类，它提供了一些并发安全的操作。`ConcurrentHashMap` 使用分段锁的机制，将哈希表分成多个段，每个段上都有一个独立的锁，从而降低了锁的粒度，提高了并发性能。这样，不同的线程可以同时修改不同的段，避免了整个数据结构的锁竞争。

总的来说，如果需要在多线程环境中使用哈希表，推荐使用 `ConcurrentHashMap` 而不是 `HashMap`，以确保线程安全性。

6. 如何做到让HashMap线程安全？

在Java中，`HashMap`本身不是线程安全的，但可以通过以下几种方式来实现线程安全的`HashMap`：

1. **使用`Collections.synchronizedMap`方法：**

   ```
   Map<K, V> synchronizedMap = Collections.synchronizedMap(new HashMap<K, V>
   ());
   ```

   这将返回一个线程安全的`Map`，它在每个方法上都使用同步机制来确保线程安全。但请注意，虽然这确保了每个方法的原子性，但在多个操作之间，仍然可能需要额外的同步。

2. **使用`ConcurrentHashMap`：**
   `ConcurrentHashMap`是Java提供的线程安全的`Map`实现。它使用分段锁机制，每个段相当于一个小的`HashMap`，不同的段之间互不影响，这样可以提高并发性能。

   ```
   Map<K, V> concurrentMap = new ConcurrentHashMap<K, V>();
   ```

3. **使用 `Collections.synchronizedMap` 包装 `HashMap` 的迭代器：**
   如果你使用 `Collections.synchronizedMap` 来创建线程安全的 `HashMap`，当你迭代 `Map` 时，仍然需要手动同步。你可以通过在迭代器上使用 `synchronized` 块来实现：

```
Map<K, V> synchronizedMap = Collections.synchronizedMap(new HashMap<K, V>
());
Set<K> keySet = synchronizedMap.keySet();
synchronized (keySet) {
    Iterator<K> iterator = keySet.iterator();
    while (iterator.hasNext()) {
        K key = iterator.next();
        // 在此处执行操作
    }
}
```

如果需要线程安全的 `HashMap`，推荐使用 `ConcurrentHashMap`，因为它在并发场景下性能更好。根据具体的需求，选择适合的方法来保证线程安全。

7. ConcurrentHashMap怎么保证线程安全的?

`ConcurrentHashMap` 是Java集合框架中的线程安全的 `Map` 实现。它采用了一些策略来确保在多线程环境中的安全性：

1. **分段锁（Segmentation）：**
   `ConcurrentHashMap` 将整个数据结构分割成多个独立的段（segments），每个段独立地管理一部分数据。每个段都类似于一个小的 `HashMap`，有自己的锁。这样，不同段的数据可以在不同的锁上进行操作，提高了并发度。当一个线程在一个段上进行操作时，其他线程可以同时在其他段上进行操作，减小了竞争范围。
2. **精细化的锁策略：**
   在 `ConcurrentHashMap` 中，只有在读写冲突的时候才会使用锁，而且只锁定与冲突相关的段，而不是整个 `Map`。这种细粒度的锁策略减小了锁的争用，提高了并发性能。
3. **读操作的无锁支持：**
   `ConcurrentHashMap` 对于读操作提供了无锁支持，允许多个线程同时进行读取操作，不会阻塞。只有在写操作发生时才需要加锁，确保写操作的原子性和可见性。
4. **CAS（Compare and Swap）操作：**
   `ConcurrentHashMap` 使用CAS操作来确保对数据的原子更新。CAS是一种无锁算法，它比传统的锁机制更轻量级。通过CAS， `ConcurrentHashMap` 可以在不加锁的情况下完成一些简单的操作。
5. **适应性自动调整：**
   `ConcurrentHashMap` 在运行时会根据负载因子、并发度等参数进行自动调整。这使得它在不同的负载和并发情况下都能够保持高效。

`ConcurrentHashMap` 通过使用分段锁、细粒度的锁策略、无锁的读操作和CAS操作等技术，以及适应性自动调整，来保证在多线程环境中的高并发性能和线程安全。这些特性使得 `ConcurrentHashMap` 成为处理高并发情况下 `Map` 操作的理想选择。

8. 手撕生产者消费者模型

```
// 生产者
class Producer implements Runnable {
    private BlockingQueue<Integer> queue;

    public Producer(BlockingQueue<Integer> queue) {
        this.queue = queue;
```

```
    }

    public void run() {
        try {
            while (true) {
                int value = produce(); // 生产数据
                queue.put(value); // 将数据放入队列
                Thread.sleep(1000); // 模拟生产过程
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    private int produce() {
        // 生产过程
        return 1;
    }
}

// 消费者
class Consumer implements Runnable {
    private BlockingQueue<Integer> queue;

    public Consumer(BlockingQueue<Integer> queue) {
        this.queue = queue;
    }

    public void run() {
        try {
            while (true) {
                int value = queue.take(); // 从队列中取出数据
                consume(value); // 消费数据
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    private void consume(int value) {
        // 消费过程
    }
}
```

9. 手撕两个线程抢票代码，有没有其他方式保证线程安全?

```
// 使用synchronized关键字保证线程安全
class TicketSystem {
    private int tickets = 100;

    public synchronized void sellTicket() {
        if (tickets > 0) {
            System.out.println(Thread.currentThread().getName() + "卖出一张票，剩余票数：" + --tickets);
        }
```

```
    }
}

// 或使用ReentrantLock
class TicketSystem {
    private int tickets = 100;
    private ReentrantLock lock = new ReentrantLock();

    public void sellTicket() {
        lock.lock();
        try {
            if (tickets > 0) {
                System.out.println(Thread.currentThread().getName() + "卖出一张
票，剩余票数：" + --tickets);
            }
        } finally {
            lock.unlock();
        }
    }
}
```

10. Volatile关键字的作用

`volatile`是Java关键字之一，它主要用于保证多线程环境下变量的可见性和禁止指令重排序。`volatile`关键字的主要作用包括：

1. **可见性（Visibility）：**
   当一个变量被声明为`volatile`时，意味着这个变量可能会被多个线程同时访问，且不同线程之间的修改操作是可见的。具体来说，如果一个线程修改了一个`volatile`变量的值，这个修改对其他线程是可见的，其他线程会立即看到这个变量的最新值。
2. **禁止指令重排序（Ordering）：**
   `volatile`关键字还有禁止指令重排序的作用。在不使用`volatile`的情况下，编译器和处理器可能会对指令进行重排序，这在多线程环境下可能导致意外的行为。通过将变量声明为`volatile`，可以防止编译器和处理器对其进行重排序，确保按照代码的顺序执行。

使用`volatile`的经典场景包括：

- **标志位：** 在多线程环境中，一个线程设置一个`volatile`标志位，另一个线程检查这个标志位，以便在某个条件满足时通知其他线程停止执行或执行某个操作。
- **单例模式中的双检锁：** 在双检锁机制中，为了避免指令重排序，需要将单例对象声明为`volatile`。

```
public class Singleton {
    private static volatile Singleton instance;

    private Singleton() {
    }

    public static Singleton getInstance() {
        if (instance == null) {
            synchronized (Singleton.class) {
                if (instance == null) {
                    instance = new Singleton();
                }
            }
        }
```

```
            return instance;
        }
    }
```

`volatile`不能保证复合操作的原子性。如果一个操作涉及到多个变量的读写，而且这些操作必须在一个原子步骤内完成，那么`volatile`就无法满足需求，此时可能需要使用其他的同步机制，例如使用`java.util.concurrent`包中的原子类。

11. Atomic包用过吗?

`java.util.concurrent.atomic`包提供了一组用于在多线程环境中进行原子操作的类。这些类通过使用硬件级别的原子性操作或者利用`sun.misc.Unsafe`提供的 CAS（Compare-And-Swap）操作来确保对变量的操作是原子的。这些类大多数都是基于原始数据类型的，例如`int`、`long`，还有一些是引用类型。

以下是`java.util.concurrent.atomic`包中一些主要的类以及它们的用途：

1. **AtomicInteger：**
   用于对整数进行原子操作，支持原子的自增（`incrementAndGet()`）、自减（`decrementAndGet()`）等操作。
2. **AtomicLong：**
   用于对长整型进行原子操作，同样支持原子的自增、自减等操作。
3. **AtomicBoolean：**
   用于对布尔类型进行原子操作，支持原子的设置和获取操作。
4. **AtomicReference：**
   用于对引用类型进行原子操作，支持原子的获取和设置引用对象。
5. **AtomicIntegerArray、AtomicLongArray、AtomicReferenceArray：**
   用于对数组中的元素进行原子操作，提供了一些原子性的数组操作。
6. **AtomicIntegerFieldUpdater、AtomicLongFieldUpdater、AtomicReferenceFieldUpdater：**
   用于对类的字段进行原子更新，允许在并发环境中对对象的字段进行原子性操作。

这些原子类提供了一种比使用`synchronized`关键字更轻量级的线程安全机制，特别适用于一些简单的计数器、状态标志等场景。在需要进行原子操作而又不需要全局的锁的情况下，这些类可以提供更好的性能。

虽然这些类提供了原子性的操作，但并不是所有的操作都可以用原子方式完成，因此在使用时仍然需要注意保证原子性的操作是否符合预期。

12. 索引是什么？为什么能提高查询效率?

- 索引是数据库中用于加速查询的一种数据结构，通过存储一定规则的索引信息，可以快速定位到符合条件的记录。
- 索引提高查询效率的原因是它减少了需要扫描的数据量，使得数据库能够更快地定位到符合条件的数据。

13. 在数据库中，我要线程安全的修改一个表里的某一列数据，要怎么做?

- 使用事务来确保操作的原子性，即要么所有修改成功，要么全部失败。
- 使用数据库的锁机制，如行级锁或表级锁，确保同时只有一个线程在修改数据。

14. 写一下修改数据的操作语句

```
UPDATE tableName
SET columnName = newValue
WHERE condition;
```

15. 联合索引是什么？什么是最左匹配原则?

- 联合索引是指在多个列上创建的索引，可以包含两个或更多列。
- 最左匹配原则是指在联合索引中，查询时必须使用索引的最左边的列，以保证索引的有效性。

16. 如何判断查询是否走了索引?

- 使用数据库查询执行计划，查看是否使用了索引。
- 可以通过Explain语句（或类似的数据库命令）查看查询的执行计划。

17. 索引失效的场景?

- 不符合最左匹配原则。
- 使用了函数或表达式，导致索引无法被使用。
- 对索引列进行了类型转换。
- 使用了不等于（!=）操作符。
- 使用了OR操作符。

18. Binlog，redo，undo日志分别有什么作用?

- Binlog（二进制日志）：记录数据库的变更操作，用于数据备份和主从复制。
- Redo日志：记录事务对数据库的修改，用于恢复数据库。
- Undo日志：记录事务的撤销操作，用于事务的回滚。

19. Spring Boot的启动原理?

- Spring Boot的启动原理是通过SpringApplication类的run方法，该方法会创建并启动Spring容器。
- Spring Boot会自动扫描并加载应用中的Bean，进行自动配置，启动内嵌的Web服务器。
- 启动过程中会执行一系列的事件，通过ApplicationListener监听器来响应这些事件。

20. Spring Boot的核心注解有哪些？核心配置文件呢?

- 核心注解：@SpringBootApplication, @RestController, @Service, @Repository, @Component, @Configuration。
- 核心配置文件：application.properties或application.yml。

2. 麻了，石化盈科一面面经

- 自我介绍
- 工作地点以及工作岗位确认
- Java面向对象的三大特性，分别介绍一下
- List如何去重复
- 说一下冒泡排序的思想
- 创建线程有哪几种方式？线程池有哪几个
- SpringBoot有哪些优点
- SpringBoot常用的注解
- 数据库事务了解吗
- 小程序向后端发起请求经过了哪些网络节点
- 小程序请求接口过程中网络安全是如何保证的#面经#
- linux相关的命令
- kill和kill -9有哪些区别
- 在开发项目过程中遇到难解决的问题是如何处理的
- 项目中遇到了难点讲一下
- vue了解吗（回答了只会使用）
- 反问
  - 项目用到的技术栈

Java面向对象的三大特性，分别介绍一下

1. **封装（Encapsulation）：**
   - 封装是面向对象编程的基本原则之一。它将类的实现细节隐藏起来，只向外部暴露必要的接口和数据。通过封装，对象内部的实现细节对外部是不可见的，这提高了代码的安全性和可维护性。封装可以通过访问修饰符（如private、protected、public）来实现，同时提供公共的方法来访问或修改私有属性。
2. **继承（Inheritance）：**
   - 继承是一种通过重用现有类的属性和方法来创建新类的机制。子类继承父类的特性，可以使用父类中的方法和属性，同时可以在子类中扩展或修改这些方法和属性。继承实现了代码的重用，提高了代码的可维护性。Java中通过关键字 `extends` 来实现继承关系。
3. **多态（Polymorphism）：**
   - 多态是指同一个操作作用于不同的对象可以有不同的行为。多态提高了代码的灵活性和可扩展性。Java中有两种类型的多态：编译时多态（静态多态）和运行时多态（动态多态）。运行时多态是通过方法的重写（Override）和接口来实现的。编译时多态是通过方法的重载（Overload）来实现的。多态是面向对象编程中一个非常重要的概念，它使得程序更容易扩展和维护。

List如何去重复

在Java中，可以使用以下几种方式对List进行去重：

1. **使用Set：**
   利用Set的元素不可重复性，将List转换为Set，然后再转回List。这会自动去除重复元素。

```
List<String> list = new ArrayList<>();
// 添加元素到list
Set<String> set = new LinkedHashSet<>(list);  // 使用LinkedHashSet保持顺序
list.clear();
list.addAll(set);
```

2. **Java 8 Stream API：**
   使用Java 8引入的Stream API，通过流的distinct()方法去重。

```
List<String> list = new ArrayList<>();
// 添加元素到list
list = list.stream().distinct().collect(Collectors.toList());
```

3. **Apache Commons Collections：**
   使用Apache Commons Collections库中的ListUtils类。

```
List<String> list = new ArrayList<>();
// 添加元素到list
list = ListUtils.distinct(list);
```

4. **使用Java 8 Lambda 表达式和Collectors.toSet()：**

```
List<String> list = new ArrayList<>();
// 添加元素到list
list =
list.stream().collect(Collectors.toSet()).stream().collect(Collectors.toList
());
```

说一下冒泡排序的思想

冒泡排序是一种简单的排序算法，它的基本思想是通过多次遍历待排序的元素，比较相邻两个元素的大小，并根据需要交换它们的位置。在一次遍历中，将最大（或最小）的元素移动到最右边（或最左边），然后缩小待排序元素的范围，重复进行比较和交换直至完成排序。

具体的冒泡排序过程如下：

1. 从第一个元素开始，依次比较相邻的两个元素。
2. 如果前一个元素大于后一个元素，则交换它们的位置。
3. 继续向后比较和交换，直到最后一个元素，此时最大的元素已经被移到了最右边。
4. 缩小待排序的范围，重复上述步骤，直到所有元素都有序。

冒泡排序的特点是每一轮遍历都会确定一个最大（或最小）值的最终位置，因此总共需要进行 n-1 轮遍历，其中 n 是待排序元素的个数。

冒泡排序的时间复杂度为 O(n^2)，其中 n 是待排序元素的个数。尽管它在效率上不如快速排序等高级算法，但由于其简单易懂的原理，通常在教学和理解排序算法的过程中被介绍和使用。

创建线程有哪几种方式？线程池有哪几个

在Java中，创建线程有两种主要的方式：继承Thread类和实现Runnable接口。线程池则是通过`java.util.concurrent`包中的`Executor`框架来创建和管理线程。

创建线程的方式：

1. **继承Thread类：**
   - 创建一个类继承自Thread类，并重写run方法，然后创建该类的实例并调用start方法启动线程。

```
class MyThread extends Thread {
    public void run() {
        // 线程执行的代码
    }
}

MyThread myThread = new MyThread();
myThread.start();
```

2. **实现Runnable接口：**
   - 创建一个类实现Runnable接口，实现run方法，然后创建Thread类的实例，将实现了Runnable接口的对象作为参数传递给Thread的构造方法，并调用start方法启动线程。

```
class MyRunnable implements Runnable {
    public void run() {
        // 线程执行的代码
    }
}

Thread thread = new Thread(new MyRunnable());
thread.start();
```

线程池的方式：

Java中的线程池是通过`java.util.concurrent`包提供的Executor框架实现的。常用的线程池有以下几种：

1. `newCachedThreadPool`：

- 创建一个可缓存的线程池，线程池的大小可根据需要进行自动扩展，但在某些情况下可能会回收线程。

```
ExecutorService executor = Executors.newCachedThreadPool();
```

2. **`newFixedThreadPool`：**
   - 创建一个固定大小的线程池，线程数始终保持不变。

   ```java
   ExecutorService executor = Executors.newFixedThreadPool(10); // 10为线程池的大小
   ```

3. `newSingleThreadExecutor`：
   - 创建一个单线程的线程池，保证所有任务按照指定顺序（FIFO，LIFO，优先级）执行。

```
ExecutorService executor = Executors.newSingleThreadExecutor();
```

4. `newScheduledThreadPool`：
    - 创建一个定时执行任务的线程池。

```
ScheduledExecutorService executor = Executors.newScheduledThreadPool(5);
```

这些线程池的创建方式都是通过`Executors`工厂类提供的方法来实现的。选择不同的线程池取决于具体的业务需求和性能要求。

SpringBoot有哪些优点

Spring Boot 是一个基于 Spring 框架的快速开发脚手架，它简化了 Spring 应用的初始搭建和开发过程。以下是 Spring Boot 的一些优点：

1. **简化配置**：
   Spring Boot 通过约定大于配置的原则，采用默认配置，减少了开发人员的配置工作。同时提供了丰富的可配置项，使得配置变得更加灵活。
2. **内嵌容器**：
   Spring Boot 内置了常用的Servlet容器（如Tomcat、Jetty），使得应用程序可以以独立的方式运行，无需外部Web服务器的支持。这降低了部署和运维的难度。
3. **自动化配置**：
   Spring Boot 根据项目的依赖，自动进行应用程序上下文的配置。这意味着无需手动配置大量的 Bean，开发者可以更专注于业务逻辑的实现。
4. **快速开发**：
   Spring Boot 提供了大量的开箱即用的功能，如自动配置、热部署、可执行的 JAR 文件等，使得开发者可以更快地构建原型和开发功能。
5. **集成测试支持**：
   Spring Boot 提供了方便的测试支持，可以轻松编写集成测试和单元测试。测试的便捷性有助于确保应用程序的稳定性和可靠性。
6. **微服务支持**：
   Spring Boot 很好地支持微服务架构，可以通过 Spring Cloud 进行更进一步的微服务治理，提供服务注册、发现、负载均衡等功能。
7. **丰富的扩展生态系统**：
   Spring Boot 集成了众多 Spring 生态项目，如Spring Data、Spring Security等，可以方便地使用这些功能进行开发。
8. **大量的社区支持**：
   Spring Boot 有着庞大的社区，社区提供了大量的文档、教程和解决方案，开发者可以方便地获取帮助和分享经验。
9. **监控和管理**：
   Spring Boot 提供了许多用于监控和管理应用程序的工具，如Spring Boot Actuator，可以方便地查看应用程序的运行状态、性能指标等。
10. **生态系统**：
    Spring Boot 借助于Spring生态系统，提供了丰富的功能和解决方案，使得开发者能够更轻松地构建各类应用程序，包括企业级应用和微服务。

SpringBoot常用的注解

Spring Boot 提供了许多注解，用于简化应用程序的配置和开发。以下是一些常用的 Spring Boot 注解：

1. `@SpringBootApplication`：

    - 该注解用于标识一个主程序类，通常位于项目的入口处。它包含了`@Configuration`、`@EnableAutoConfiguration`、`@ComponentScan`三个注解，用于启用自动配置和组件扫描。
2. `@RestController`：
    - 用于标识一个类是 RESTful 风格的控制器，它的方法返回的数据直接作为响应体，不会被视图解析器解析。
3. `@RequestMapping`：
    - 用于映射 HTTP 请求的路径和方法，可以用在类和方法上，定义了请求的路径和处理方法。
4. `@Autowired`：
    - 用于进行依赖注入，可以标注在构造方法、Setter 方法、字段、方法上。
5. `@Service`、`@Repository`、`@Component`：
    - 分别用于标识一个类是业务服务层、数据访问层、通用组件，让 Spring 进行组件扫描并纳入 IoC 容器管理。
6. `@Configuration`：
    - 用于定义配置类，类似于 XML 配置文件。标注在类上，表示该类是一个配置类。
7. `@Value`：
    - 用于将属性值注入到 Bean 中，可以注入简单类型、字符串、表达式等。
8. `@PathVariable`：
    - 用于将 URL 模板中的变量绑定到方法参数中。
9. `@RequestParam`：
    - 用于将请求参数绑定到方法参数中，可以指定参数名、是否必须等。
10. `@ResponseBody`：
    - 用于标识一个方法返回的结果直接作为响应体，不进行视图解析。
11. `@GetMapping`、`@PostMapping`、`@PutMapping`、`@DeleteMapping`：
    - 这些注解用于简化常见 HTTP 方法的映射，分别对应于 GET、POST、PUT、DELETE。
12. `@ConfigurationProperties`：
    - 用于将配置文件中的属性值绑定到一个 JavaBean 中，方便统一管理配置。
13. `@EnableAutoConfiguration`：
    - 用于开启 Spring Boot 的自动配置功能，通常不需要手动添加，`@SpringBootApplication`已经包含了该注解。
14. `@Async`：
    - 用于标识一个方法是异步的，可以在方法上使用，也可以在类上使用。
15. `@EnableScheduling`：
    - 用于开启 Spring 的定时任务功能，标注在配置类上。

数据库事务了解吗

事务是一组被视为单个逻辑工作单元的数据库操作，要么全部执行成功，要么全部失败回滚，以确保数据库的一致性和可靠性。事务具有四个基本特性，通常称为 ACID 特性：

1. **原子性（Atomicity）**：
    - 事务是原子的，要么全部执行成功，要么全部失败回滚。如果事务在执行过程中发生错误，所有的修改将被撤销，数据库将会回滚到事务开始前的状态。
2. **一致性（Consistency）**：
    - 事务使数据库从一个一致的状态转变到另一个一致的状态。在事务执行过程中，数据库约束一直保持有效，不会因为事务的执行而被破坏。

3. **隔离性（Isolation）：**
    - 多个事务可以并发执行，但每个事务的操作应该与其他事务隔离。隔离性能防止多个事务之间相互影响，保证每个事务都感觉不到其他事务的存在。
4. **持久性（Durability）：**
    - 一旦事务提交，它对数据库的修改将是永久性的，即使系统发生故障。数据库系统通过将事务的修改记录到物理存储中来实现持久性。

在关系型数据库中，通常使用以下方式来管理事务：

- **显式事务管理：**
    - 开发者通过编程方式开始、提交或回滚事务。通常使用数据库连接对象的`beginTransaction()`、`commit()`和`rollback()`等方法。
- **声明式事务管理：**
    - 基于注解或配置的方式声明事务的边界。在 Spring 框架中，可以使用`@Transactional`注解来声明事务。这样的声明式事务管理更加方便，开发者无需显式编写事务管理的代码。

事务的使用场景包括需要一致性和可靠性的数据库操作，例如在银行应用中的转账操作、在线购物中的支付过程等。通过使用事务，可以确保这些操作不会在执行过程中发生错误或中断而导致数据不一致。

小程序向后端发起请求经过了哪些网络节点

当小程序向后端发起请求时，经过了一系列网络节点。以下是一般情况下的主要网络节点：

1. **小程序端：**
    - 请求发起的地方，可以是用户的手机或其他设备上运行的小程序。
2. **小程序服务器：**
    - 小程序端发送请求到小程序服务器，这是请求的第一个目标。小程序服务器负责接收请求、处理业务逻辑，然后向后端服务器发送请求。
3. **反向代理服务器（可选）：**
    - 在一些部署架构中，可能会存在反向代理服务器，用于负载均衡、安全过滤等。这个节点并不是必需的，具体是否存在取决于后端服务的架构。
4. **应用服务器：**
    - 后端应用服务器，接收小程序服务器的请求，处理业务逻辑，访问数据库等。这可能是一个单独的服务器或者一个集群。
5. **数据库服务器：**
    - 如果请求涉及到数据库操作，后端应用服务器会向数据库服务器发送请求，执行相应的数据库操作。
6. **其他第三方服务（可选）：**
    - 有时，后端服务可能还会涉及到与其他第三方服务进行交互，比如支付服务、消息推送服务等。
7. **返回路径：**
    - 从数据库或其他第三方服务获取到数据后，数据经过相同的路径返回，依次经过应用服务器、反向代理服务器（可选）、小程序服务器，最终到达小程序端。

在整个过程中，数据通过网络传输，每个网络节点都可能引入延迟，因此优化网络请求的性能是开发中需要考虑的一个重要方面。安全性也是关键问题，确保数据在传输过程中是加密的，并且服务端对请求进行了合适的身份验证和授权。

小程序请求接口过程中网络安全是如何保证的

在小程序请求接口的过程中，网络安全是非常重要的，涉及到用户的隐私和数据的保护。以下是一些主要的措施，这些措施一般也适用于其他前端应用：

1. **HTTPS加密：**
   - 小程序要求所有的网络请求必须使用HTTPS协议，确保数据在传输过程中是加密的。HTTPS通过SSL/TLS协议对通信内容进行加密，防止数据被中间人窃取或篡改。
2. **数据签名与加密：**
   - 对于重要的请求，可以使用签名机制确保请求的完整性和真实性。通常使用类似于HMAC（Hash-based Message Authentication Code）的算法，将请求参数和密钥进行签名，并将签名附加到请求中。服务端根据相同的密钥进行签名验证。
3. **小程序登录态与用户鉴权：**
   - 小程序通过调用 `wx.login` 获取登录凭证，然后将凭证发送到后端服务器。后端服务器使用这个凭证调用微信服务端接口验证，并获取到用户的唯一标识，以此来识别用户。确保在接口调用中对用户进行鉴权，只允许合法的用户访问敏感接口。
4. **接口访问限制：**
   - 限制用户的访问频率，通过防刷策略来避免恶意攻击和非法访问。这可以通过限制同一 IP 地址的访问频率或者使用令牌桶、漏桶等算法进行限流。
5. **OAuth2.0认证：**
   - 对于需要用户授权的操作，可以使用 OAuth2.0 协议，通过微信提供的用户授权接口，获取用户的授权同意后的令牌（access token）来进行接口访问。
6. **防止SQL注入和XSS攻击：**
   - 在处理用户输入时，对输入进行有效的过滤和转义，防止SQL注入和XSS攻击。不信任的输入数据应该被过滤或者进行合适的编码，确保它们不会被误认为是执行代码或者SQL语句。
7. **敏感数据保护：**
   - 对于敏感数据的存储和传输，需要进行适当的加密处理，包括数据库中的密码、个人隐私信息等。同时，确保在传输中采用安全的协议，如HTTPS。
8. **错误处理：**
   - 在接口中提供详细的错误信息是为了方便开发调试，但在生产环境中应该尽可能减少详细的错误信息的返回，以防泄漏敏感信息。

这些安全措施需要在前端小程序和后端服务器都进行合作实施，形成端到端的安全体系。前端应保护用户信息和凭证，后端则需要验证和授权请求，以确保系统的整体安全性。

linux相关的命令

以下是一些常用的Linux命令，用于在终端中执行各种任务：

1. `ls` **- 列出文件和目录：**
   - 用于列出当前目录下的文件和子目录。

```
ls
```

2. `cd` **- 切换目录：**
   - 用于更改当前工作目录。

```
cd directory_name
```

3. `pwd` **- 显示当前工作目录：**
   - 用于显示当前工作目录的路径。

```
pwd
```

4. `cp` - **复制文件或目录**：
   - 用于复制文件或目录。

```
cp source destination
```

5. `mv` - **移动/重命名文件或目录**：
   - 用于移动文件或目录，也可用于重命名。

```
mv source destination
```

6. `rm` - **删除文件或目录**：
   - 用于删除文件或目录。

```
rm file_name
```

7. `mkdir` - **创建目录**：
   - 用于创建新目录。

```
mkdir directory_name
```

8. `rmdir` - **删除空目录**：
   - 用于删除空目录。

```
rmdir directory_name
```

9. `cat` - **查看文件内容**：
   - 用于查看文件的内容。

```
cat file_name
```

10. `more` - **逐屏显示文件内容**：
    - 用于逐屏显示文件的内容。

```
more file_name
```

11. `less` - **逐屏显示文件内容，支持向前翻页**：
    - 用于逐屏显示文件的内容，支持向前翻页。

```
less file_name
```

12. `head` - **显示文件头部内容**：
    - 用于显示文件的头部内容。

```
head file_name
```

13. `tail` - **显示文件尾部内容**：
    - 用于显示文件的尾部内容，常用于查看日志。

```
tail file_name
```

14. `grep` - **在文件中查找匹配的文本**：
    - 用于在文件中查找包含指定文本的行。

```
grep pattern file_name
```

15. `find` - **在文件系统中查找文件**：
    - 用于在文件系统中递归查找文件。

```
find directory -name file_name
```

16. `chmod` - **修改文件权限**：
    - 用于修改文件或目录的权限。

```
chmod permissions file_name
```

17. `chown` - **修改文件所有者**：
    - 用于修改文件或目录的所有者。

```
chown owner:group file_name
```

18. `ps` - **显示当前运行的进程**：
    - 用于显示当前正在运行的进程信息。

```
ps
```

19. `kill` - **终止进程**：
    - 用于终止指定的进程。

```
kill process_id
```

20. `man` - **查看命令手册**：
    - 用于查看Linux命令的手册页。

```
man command_name
```

kill和kill -9有哪些区别

`kill` 和 `kill -9` 是用于终止进程的 Linux 命令，它们之间有一些重要的区别。

1. **`kill`命令：**
   - `kill`命令用于向进程发送信号，最常见的信号是 `SIGTERM`（15号信号）。这个信号请求进程正常终止，允许进程执行一些清理操作，然后退出。使用`kill`命令时，可以指定进程号（PID）或进程名。

   ```
   kill [signal] PID
   ```

   例如，终止进程号为1234的进程：

   ```
   kill 1234
   ```

2. **`kill -9`命令：**
   - `kill -9`命令也被称为强制终止，它发送 `SIGKILL`（9号信号）给进程，强制立即终止进程，不给进程执行清理和释放资源的机会。这是一种比较极端的终止方式，应该谨慎使用。使用`kill -9`命令时，同样可以指定进程号或进程名。

   ```
   kill -9 PID
   ```

   例如，强制终止进程号为5678的进程：

   ```
   kill -9 5678
   ```

区别总结：

- `kill`发送的是 `SIGTERM` 信号，允许进程进行清理工作。
- `kill -9`发送的是 `SIGKILL` 信号，立即终止进程，不允许进程进行清理工作。
- 使用 `kill` 是一种相对友好的终止方式，留给进程一些处理的机会；而使用 `kill -9` 是一种强制的方式，不管进程的状态如何，都会立即终止。
- 在正常情况下，应该首先尝试使用 `kill` 终止进程，只有在无法正常终止时，才考虑使用 `kill -9`。

3. 百度实习面经，瑟瑟发抖...

1.用什么语言多?
2.内存分区，进程内存映象，linux内存模型
3.堆是谁来分配
4.开发时碰到内存溢出，内存泄露
5.了解过检测内存泄露的工具或是方法
6.c++了解多吗，回答只会一点，面试官说不问了，部门也不用C++
7.extern c语法，具体什么作用
8.进程几种状态，切换的时机；interruptible和uninterrupted状态
9.进程和线程区别
10.内存管理的buddy，伙伴系统
11.CFS，和O(1)对比，结合实例说了一下
12.中断的基本框架，以键盘中断为例，讲了整个流程和上下部
13.IPC
14.网络，问了最基础的三次握手和四次挥手，TIMEWAIT
15.linux查看进程状态
16.线上环境服务出现问题怎么检测，挂了怎么拉起，怎么管理
17.git操作
18.项目拷打15min
19.虚拟化设备，聊了一下网卡
20.问了蓝桥杯省一，顺势引出做题，leetcode1143
21.最新学了啥，聊了聊一些paper
22.用qemu启过虚拟机吗
反问环节
1.部门和小组做什么
2.岗位需要学习什么，怎么抛开繁琐的代码实现掌握核心概念（刚从牛客上偷的问题）
3.虚拟化相关技术

---

1 内存分区，进程内存映象，linux内存模型

1. **内存分区（Memory Partitioning）**：
   内存分区是指将计算机的物理内存划分成不同的区域，每个区域用于存储不同类型的数据。常见的内存分区包括：
   - **代码段（Code Segment）**：存储程序的执行代码。
   - **数据段（Data Segment）**：存储全局变量等数据。
   - **堆（Heap）**：用于动态分配内存，由程序员进行手动管理。
   - **栈（Stack）**：用于存储函数调用的局部变量、函数参数等。
2. **进程内存映像（Process Memory Image）**：
   进程内存映像是指一个运行的进程在内存中的布局和组织。它包括以下部分：
   - **代码段**：存储进程的执行代码。
   - **数据段**：包括初始化的全局变量和静态变量。
   - **堆**：用于动态分配内存，比如通过`malloc`和`free`。

- **栈**：存储函数调用的局部变量、函数参数等。

进程内存映像使得操作系统能够有效地管理多个进程，并确保它们之间不会相互干扰。

3. **Linux内存模型**：
   Linux操作系统采用了分页机制来管理内存，其中有两个关键概念：
   - **虚拟内存（Virtual Memory）**：允许程序使用比实际物理内存更大的地址空间。这使得每个进程都有独立的地址空间，提高了系统的稳定性和安全性。
   - **分页（Paging）**：将物理内存划分成大小固定的页面，与虚拟内存中的页面相对应。当一个程序需要访问某个页面时，操作系统将其加载到物理内存中。这有助于提高内存的利用率和管理效率。

2 堆是谁来分配

在大多数编程语言中，堆内存的分配和释放通常是由程序员手动管理的，而不是由编程语言或操作系统自动处理。

1. **手动管理（Manual Management）**：
   - 在C语言中，通过使用`malloc`函数来动态分配堆内存，而使用`free`函数来释放已分配的内存。
   - 在C++中，可以使用`new`运算符来分配堆内存，而使用`delete`运算符来释放内存。
2. **自动管理（Automatic Management）**：
   - 有一些编程语言提供自动内存管理，例如Java、C#和Python。在这些语言中，垃圾回收器（Garbage Collector）负责自动识别不再被程序使用的内存，并释放它们，包括堆中的内存。

在手动管理的情况下，程序员负责确保正确地分配和释放内存，以避免内存泄漏（内存未释放）或悬挂指针（引用已释放的内存）等问题。在自动管理的情况下，垃圾回收器会自动处理内存的分配和释放，减轻了程序员的负担，但也可能引入一些性能开销。

总体而言，堆内存的分配和释放取决于编程语言和程序员的选择。

3 开发时碰到内存溢出，内存泄露

处理内存溢出：

1. **检查算法和数据结构**：
   - 优化算法和数据结构，确保它们在处理大数据量时具有良好的性能。
2. **释放不必要的资源**：
   - 在使用完资源后及时释放，包括文件句柄、网络连接等。
3. **增加堆栈大小**：
   - 对于一些递归或深度调用的情况，可能需要增加堆栈大小。
4. **使用内存池**：
   - 对于频繁的内存分配和释放，考虑使用内存池，减少内存碎片化。
5. **分析内存使用情况**：
   - 使用工具分析内存使用情况，例如内存分析器，以找出哪些部分占用了大量内存。

处理内存泄漏：

1. **使用内存分析工具**：
   - 使用专业的内存分析工具，例如Valgrind（对C/C++很有用）或内存分析器，来检测和报告内存泄漏。
2. **代码审查**：

- 定期进行代码审查，特别关注内存分配和释放的代码块，确保它们的使用是正确的。

3. **使用智能指针**：
   - 对于支持智能指针的语言，使用它们可以减少手动内存管理的错误。
4. **确保资源释放**：
   - 在对象生命周期结束时，确保释放与之关联的资源，包括关闭文件、释放网络连接等。
5. **编写单元测试**：
   - 编写测试用例，特别是针对内存管理的测试，以及时捕捉潜在的问题。
6. **日志和监控**：
   - 在应用程序中实施日志和监控机制，以及时发现和诊断内存泄漏问题。
7. **定期性能优化**：
   - 定期进行性能优化，包括内存方面的优化，以确保应用程序在长时间运行后不会因为内存泄漏而出现问题。

4 了解过检测内存泄露的工具或是方法

检测内存泄漏的工具和方法有很多，具体的选择通常取决于你所使用的编程语言和开发环境。以下是一些常见的工具和方法：

内存分析工具：

1. **Valgrind（C/C++）**：
   - Valgrind是一个强大的工具，可用于检测内存泄漏、不初始化的内存访问等问题。它能够在运行时对程序进行详细的内存分析。
2. **AddressSanitizer（C/C++）**：
   - AddressSanitizer是GCC和Clang的一个特性，用于检测内存错误，包括内存泄漏、访问已释放内存等。
3. **LeakSanitizer（C/C++）**：
   - LeakSanitizer是GCC和Clang的另一个特性，专门用于检测内存泄漏。
4. **Dr. Memory（Windows，C/C++）**：
   - Dr. Memory是一款用于检测内存泄漏和内存错误的工具，特别适用于Windows环境。
5. **Heap Profiler（Java）**：
   - Java有内置的Heap Profiler，可以用于分析Java应用程序的内存使用情况，包括泄漏检测。

日志和手动检查：

1. **内存日志**：
   - 在代码中插入日志语句，记录内存分配和释放的信息，以便分析程序的内存使用情况。
2. **手动检查**：
   - 定期审查代码，特别关注内存分配和释放的地方，确保资源的正确释放。

静态代码分析工具：

1. **Clang Static Analyzer**：
   - Clang Static Analyzer是一个用于静态代码分析的工具，可以帮助检测代码中的潜在问题，包括内存泄漏。
2. **Coverity**：
   - Coverity是一种商业静态代码分析工具，可以帮助发现和修复各种代码缺陷，包括内存泄漏。

运行时检测：

1. **自定义检测机制**：
    - 在代码中插入特殊的标记或计数，以便在运行时检测内存泄漏。
2. **使用内存管理库**：
    - 一些编程语言或框架提供了内置的内存管理库，可以用于检测内存泄漏，例如C++中的 `_CrtDumpMemoryLeaks`。

5 extern c语法，具体什么作用

`extern "C"` 是一种 C++ 中的语法特性，用于在 C++ 代码中声明使用 C 语言编写的函数或变量。它告诉编译器按照 C 语言的方式进行名称修饰和链接。

在 C++ 中，默认情况下，函数和变量的名称会经过名称修饰（name mangling）以支持函数重载和其他 C++ 特性。而在 C 语言中，不进行名称修饰。

使用 `extern "C"` 可以使 C++ 代码与 C 代码进行混合编译和链接，解决了在 C++ 中调用 C 函数或变量时名称不匹配的问题。

用法示例：

1. 函数声明：

假设有一个 C 语言的头文件 `example.h`：

```
// example.h
#ifdef __cplusplus
extern "C" {
#endif

void c_function(); // C 函数声明

#ifdef __cplusplus
}
#endif
```

在 C++ 中的源文件中可以这样使用：

```
// cpp_file.cpp
#include "example.h"

int main() {
    c_function(); // 可以在 C++ 代码中调用 C 函数
    return 0;
}
```

变量声明：

```
// example.h
#ifdef __cplusplus
extern "C" {
#endif

extern int c_variable; // C 变量声明

#ifdef __cplusplus
}
#endif
```

在 C++ 中的源文件中可以这样使用：

```
// cpp_file.cpp
#include "example.h"

int main() {
    int value = c_variable; // 可以在 C++ 代码中使用 C 变量
    return 0;
}
```

使用 `extern "C"` 时需要注意以下几点：

- `extern "C"` 应该用于 C++ 代码中，通过 `#ifdef __cplusplus` 来确保只在 C++ 环境下进行声明。
- `extern "C"` 的作用是告诉编译器使用 C 语言的命名和链接规则，因此在 `{}` 内部的函数和变量会按照 C 的方式进行处理。
- 使用 `extern "C"` 不会改变函数的实际实现，只是改变了函数的名称在符号表中的表示方式。

6 进程几种状态，切换的时机；interruptible和uninterrupted状态

在操作系统中，一个进程可以处于多个状态之一，而进程的状态转换是由操作系统内核根据进程执行的情况来管理的。以下是常见的进程状态以及状态之间的切换时机：

进程的几种状态：

1. **运行（Running）**：
   - 进程占用 CPU 运行，是当前正在执行的进程。
2. **就绪（Ready）**：
   - 进程已经准备好运行，但由于其他进程正在执行，它暂时无法获得 CPU 时间。
3. **阻塞（Blocked）**（也称为等待或睡眠）：
   - 进程暂时不能执行，因为它正在等待某个事件的发生，例如等待用户输入或等待磁盘I/O完成。

进程状态之间的切换时机：

1. **创建（Creation）**：
   - 进程被创建时，处于创建状态。
2. **就绪和运行状态切换**：
   - 进程在就绪状态等待 CPU 时间，当操作系统调度它时，切换到运行状态。
3. **运行和阻塞状态切换**：

    - 进程在执行过程中可能遇到需要等待的事件，例如等待I/O操作完成，此时它从运行状态切换到阻塞状态。
4. **阻塞和就绪状态切换**：
    - 当某个等待的事件发生，阻塞的进程切换回就绪状态，等待CPU的调度。
5. **运行和终止状态切换**：
    - 进程执行完毕或发生致命错误时，从运行状态切换到终止状态。

interruptible 和 uninterruptible 状态：

1. **interruptible 状态**：
    - 进程处于可以被中断的状态。在等待某些事件（如I/O操作）时，进程可能处于interruptible状态，这表示它可以被操作系统中断，从而响应其他事件。
2. **uninterruptible 状态**：
    - 进程处于无法被中断的状态。在某些情况下，例如正在等待磁盘I/O完成时，进程可能处于uninterruptible状态，这表示它不能被中断，必须等待事件完成才能继续执行。

这两种状态的选择通常取决于进程等待的事件的性质。例如，对于磁盘I/O，可能会选择uninterruptible状态，因为中断可能导致数据不一致。而对于网络I/O，可能选择interruptible状态，以便更及时地响应其他事件。

7 进程和线程区别

进程（Process）和线程（Thread）是操作系统中用于执行程序的两个基本的执行单位，它们之间有一些关键的区别：

1. **定义**：

- **进程**：是一个独立的执行单元，有自己的地址空间、数据栈，以及其他系统资源（文件描述符、信号处理等）。进程之间相互独立，通信需要使用进程间通信（IPC）机制。
- **线程**：是进程的一部分，共享进程的地址空间和其他资源，但拥有独立的执行栈。线程之间更容易共享数据和通信，因为它们共享相同的地址空间。

2. **资源开销**：

- **进程**：相对较大的资源开销，每个进程都有独立的内存空间、文件描述符等。
- **线程**：较小的资源开销，因为它们共享相同的资源。

3. **通信**：

- **进程**：通信需要使用IPC机制，如管道、消息队列、共享内存等。
- **线程**：通信更容易，因为它们共享相同的地址空间。

4. **独立性**：

- **进程**：相对独立，一个进程的崩溃通常不会影响其他进程。
- **线程**：一个线程的崩溃可能导致整个进程的崩溃，因为它们共享相同的地址空间和资源。

5. 创建和销毁：

- **进程**：相对较慢，涉及到分配独立的地址空间、资源初始化等。
- **线程**：相对较快，因为它们共享相同的资源，创建和销毁的开销较小。

6. 安全性：

- **进程**：由于进程有独立的地址空间，一个进程的错误通常不会直接影响其他进程。
- **线程**：由于共享相同的地址空间，一个线程的错误可能会影响其他线程。

7. **适用场景：**

- **进程**：适用于多核心系统，能够实现真正的并行计算。
- **线程**：适用于需要共享数据和通信的任务，因为线程可以更方便地共享相同的地址空间。

8 内存管理的buddy，伙伴系统

Buddy系统是一种内存管理的算法，常用于管理动态分配的内存块。它主要用于解决内存碎片化的问题，提高内存利用率。Buddy系统基于二叉树的概念，将内存分割为大小为2的幂的块，这些块的大小就像一对伙伴。

下面是Buddy系统的基本原理和操作：

1. **内存分配**：
2. 初始时，整个可用内存看作是一个大小为2的幂的块。
3. 当需要分配一块大小为2^k的内存时，系统在2^k的空闲块链表中查找可用块。如果找到，则分配该块；否则，向上合并块，直到找到合适大小的块。
4. **内存释放**：
5. 当一块内存释放时，将其加入到对应大小的空闲块链表。
6. 检查其伙伴块是否也是空闲的，如果是，就合并这两个块，并将合并后的块加入到上一级的空闲块链表中。这一过程一直持续，直到合并到最大块或者找到一个非空闲的块。
7. **优点**：

- 解决了内存碎片问题，因为大小为2的幂的块容易合并和分割。
- 分配和释放操作的时间复杂度为O(1)。

4. **缺点**：

- 内存浪费：由于每个块的大小必须是2的幂，因此可能出现一些内存浪费。
- 外部碎片：释放的内存块可能不能直接合并到相邻的块中，导致外部碎片。

5. **应用场景**：

- 适用于需要频繁分配和释放小块内存的场景，例如操作系统中的内核内存管理。
- 不适用于大对象的分配，因为可能导致较大的内存浪费。

9 CFS，和O(1)对比，结合实例说了一下

CFS（Completely Fair Scheduler）和O(1)调度器是Linux内核中用于进程调度的两种不同的调度算法。

O(1)调度器：

O(1)调度器是早期Linux内核中使用的一种调度算法，它的设计目标是使调度的时间复杂度保持为O(1)。这是通过维护一个任务数组和一个就绪队列实现的，通过简单的数据结构，O(1)调度器可以在常数时间内找到下一个要执行的任务。

实例：

```
#include <linux/sched.h>

void some_function(void) {
    while (1) {
        // 进行一些工作
        schedule();  // 使用O(1)调度器进行进程切换
    }
}
```

在上述示例中，`schedule()` 函数用于调用O(1)调度器进行任务切换。O(1)调度器在实时性能方面表现较好，但在处理大量任务时，可能导致某些任务饥饿。

CFS调度器：

CFS调度器是Linux内核中较新的调度算法，引入了红黑树的概念，通过对任务运行时间的虚拟时钟进行动态的调整，实现对任务的公平调度。CFS调度器的设计目标是提供更好的公平性和负载均衡。

实例：

```
#include <linux/sched.h>

void some_function(void) {
    while (1) {
        // 进行一些工作
        schedule();  // 使用CFS调度器进行进程切换
    }
}
```

在这个示例中，`schedule()` 调用用于调用CFS调度器进行任务切换。CFS调度器会根据任务的虚拟运行时间来选择下一个要运行的任务，以实现公平性。

对比：

1. **时间复杂度**：
   - O(1)调度器的时间复杂度为O(1)。
   - CFS调度器的时间复杂度相对较高，但它更注重于提供公平性和负载均衡。
2. **公平性**：
   - O(1)调度器可能在任务的公平性方面表现较差，容易导致某些任务长时间无法获得CPU时间。
   - CFS调度器被设计为更公平，它通过动态调整虚拟运行时间来确保任务相对公平地分享CPU时间。
3. **适用场景**：
   - O(1)调度器适用于需要较低调度延迟的场景。
   - CFS调度器适用于更注重公平性和负载均衡的场景。

10 中断的基本框架，以键盘中断为例

中断是计算机系统中一种异步事件处理的机制，允许系统在执行当前任务的同时响应外部事件。以下是处理中断的基本框架，以键盘中断为例：

1. **中断发生**：

- 用户按下键盘上的某个键触发中断，键盘控制器产生中断请求（IRQ）。

2. **硬件层处理**：

- CPU检测到中断请求，停止当前执行的任务。
- CPU保存当前执行任务的上下文（程序计数器、寄存器等）。

3. **中断向量表**：

- 硬件通过中断向量表确定中断的类型和处理程序的入口地址。
- 在键盘中断的情况下，中断向量表会指向处理键盘中断的中断服务程序。

4. **中断服务程序（Interrupt Service Routine，ISR）**：

- 控制权转移到相应中断的处理程序。
- 在键盘中断的情况下，键盘中断服务程序负责处理键盘输入。

```
keyboard_isr:
    ; 处理键盘中断的汇编代码
    ; 读取键盘输入，更新相应的数据结构或触发相应的事件
    ; 恢复寄存器状态等

    ; 中断服务程序执行完毕后，执行中断返回指令
    iret
```

5. **软件层处理**：

- 中断服务程序执行完毕后，CPU从中断返回指令（iret）返回到之前被中断的任务。
- 恢复之前保存的任务上下文。

6. **继续执行任务**：

- 控制权返回到之前被中断的任务，任务继续执行。

在这个基本框架下，中断服务程序是中断处理的核心。键盘中断服务程序将负责读取键盘输入、更新相应的数据结构、触发事件等。整个中断处理流程实现了异步事件的响应，使得系统能够在执行任务的同时处理来自外部的事件，提高了系统的响应性。

11 IPC

IPC（Inter-Process Communication）是指不同进程之间进行数据交换和通信的机制。在多任务和多进程的操作系统中，不同的进程可能需要互相通信，共享数据或者协同完成某些任务。以下是几种常见的IPC方式：

1. **管道（Pipe）**：

- **描述：** 管道是一种半双工的通信机制，数据流只能单向流动。通常用于具有父子关系的进程之间的通信。
- **示例：** 在Shell中，通过管道可以将一个进程的输出连接到另一个进程的输入，实现两者之间的通信。

```
$ ps aux | grep "process_name"
```

2. **消息队列（Message Queue）**：

- **描述：** 消息队列是一种通过消息进行通信的机制，可以实现进程之间的异步通信。
- **示例：** 进程A通过消息队列向进程B发送消息，进程B从消息队列中读取消息并作出相应处理。

3. **共享内存（Shared Memory）**：

- **描述：** 共享内存允许多个进程访问同一块物理内存区域，进程可以直接读写这块内存区域。
- **示例：** 进程A将数据写入共享内存，进程B可以直接从共享内存读取这些数据，实现了高效的进程间通信。

4. **信号（Signal）**：

- **描述：** 信号是一种异步通信方式，用于通知进程发生了某个事件。每个信号都有一个唯一的数字标识，例如SIGINT表示中断信号。
- **示例：** 进程A通过发送信号SIGUSR1通知进程B执行某个特定操作。

5. **套接字（Socket）**：

- **描述：** 套接字是一种提供网络通信的机制，也可用于同一台主机上不同进程之间的通信。
- **示例：** 在客户端-服务器模型中，服务器通过套接字监听客户端的连接请求，实现进程之间的通信。

6. **信号量（Semaphore）**：

- **描述：** 信号量是一种用于控制多个进程对共享资源访问的机制，可用于解决进程同步和互斥问题。
- **示例：** 多个进程需要同时访问一个共享资源时，可以使用信号量来进行同步和协调。

这些IPC机制提供了不同层次的抽象和复杂性，选择合适的IPC方式取决于具体的应用需求和设计考虑。

12 linux查看进程状态

在Linux系统中，可以使用一系列命令来查看进程的状态。以下是一些常用的命令：

1. **ps命令**：

`ps` 命令用于显示当前运行在系统上的进程。下面是一些常见的用法：

- 查看所有进程：

```
ps aux
```

- 查看指定用户的进程：

```
ps -u username
```

2. **top命令**：

`top` 命令以动态的方式显示系统的活动进程。它提供了一个实时更新的任务列表，并显示各种系统资源使用情况。

```
top
```

3. **htop命令**：

`htop` 是 `top` 的一个增强版本，提供了更多的交互式功能和更直观的界面。

```
htop
```

4. **pgrep命令**：

`pgrep` 命令用于通过进程名查找进程的PID（进程标识符）。

```
pgrep process_name
```

5. **pkill命令**：

`pkill` 命令用于通过进程名终止进程。

```
pkill process_name
```

6. **kill命令**：

`kill` 命令用于向指定的进程发送信号，例如终止进程。

```
kill -9 PID
```

以上命令中的 PID 是进程的标识符，可以通过 `ps` 命令或其他进程查看命令获取。

13 线上环境服务出现问题怎么检测，挂了怎么拉起，怎么管理

线上环境服务出现问题时，检测、拉起和管理服务都是关键的运维任务。以下是一些建议和常见的做法：

1. **检测服务问题**：
2. **监控系统**：
   - 使用监控系统实时监测关键指标，如 CPU 使用率、内存使用率、网络流量等。
   - 设置阈值并配置警报，以便在达到或超过阈值时及时通知运维人员。
3. **日志分析**：
   - 定期分析服务日志，查找异常信息，警报或记录异常情况。
   - 使用日志聚合工具，如ELK Stack（Elasticsearch、Logstash、Kibana），以更方便地分析和监控日志。
4. **心跳检测**：
   - 实现心跳检测机制，定期向服务发送请求并检查响应，确保服务正常运行。
5. **综合健康检查**：
   - 定期进行综合健康检查，包括服务端口的可用性、依赖服务的连通性等。
6. **服务挂了怎么拉起**：
7. **自动恢复机制**：
   - 实现自动恢复机制，例如使用进程管理工具（如systemd、supervisord）监控服务状态，当服务挂掉时自动拉起。
8. **容器编排工具**：
   - 如果使用容器化技术，可使用容器编排工具（如Docker Compose、Kubernetes）来管理服务的自动重启和伸缩。
9. **自动扩展**：
   - 在云环境中，通过自动扩展组件，如Auto Scaling（AWS）、Instance Group（Google Cloud）来动态调整实例数量，确保服务可用性。
10. **服务管理**：
11. **版本控制**：
    - 使用版本控制系统（如Git）来管理服务代码和配置，确保可以回滚到稳定的版本。
12. **灰度发布**：
    - 实施灰度发布策略，逐步将新版本服务引入线上环境，减少潜在问题的影响范围。
13. **自动化运维工具**：
    - 使用自动化运维工具（如Ansible、Chef、Puppet）来部署和管理服务，确保环境一致性。
14. **备份与恢复**：
    - 定期进行数据备份，确保在服务问题导致数据丢失时能够迅速恢复。
15. **紧急手动操作**：
    - 提前定义好应急手动操作流程，以防止自动化机制失效时进行紧急修复。

通过以上措施，可以有效地检测服务问题、自动拉起服务、并管理服务的生命周期，提高服务的可靠性和稳定性。

14 git操作

Git 是一分布式版本控制系统，用于跟踪文件的变化和协作开发。以下是一些基本的 Git 操作：

1. **初始化仓库**：

```
git init
```

在当前目录下初始化一个新的 Git 仓库。

2. **克隆仓库**：

```
git clone <repository_url>
```

从远程仓库克隆一个本地副本。

3. **添加文件**：

```
git add <filename>
```

将文件添加到暂存区。

4. **提交更改**：

```
git commit -m "Commit message"
```

将暂存区的更改提交到本地仓库。

5. **查看状态**：

```
git status
```

查看工作区、暂存区和本地仓库的状态。

6. **查看提交历史**：

```
git log
```

查看提交历史记录，包括提交者、提交时间和提交信息。

7. **创建分支**：

```
git branch <branch_name>
```

创建一个新的分支。

8. **切换分支**：

```
git checkout <branch_name>
```

切换到指定分支。

9. **合并分支**：

```
git merge <branch_name>
```

将指定分支的更改合并到当前分支。

10. **远程操作**：

- **添加远程仓库**：

```
git remote add origin <repository_url>
```

- **推送到远程仓库**：

```
git push -u origin <branch_name>
```

- **拉取远程仓库的更改**：

```
git pull origin <branch_name>
```

11. **撤销更改**：

- **撤销工作区的更改**：

```
git checkout -- <filename>
```

- **撤销暂存区的更改**：

```
git reset HEAD <filename>
```

- **撤销最后一次提交**：

```
git reset --soft HEAD^
```

12. **标签操作**：

- **创建标签**：

```
git tag -a <tag_name> -m "Tag message"
```

- **推送标签到远程仓库**：

```
git push origin --tags
```

这些是 Git 的一些基本操作，涵盖了从初始化仓库到推送到远程仓库的基本流程。在实际使用中，你可能还会遇到其他一些高级的 Git 操作，如 rebase、stash、cherry-pick 等，具体使用取决于项目的需求和团队的工作流程。

15 虚拟化设备

虚拟化设备是指通过虚拟化技术模拟或抽象的硬件设备，使得多个虚拟机（VM）能够在同一物理主机上并发运行，各自拥有自己独立的虚拟硬件资源。这种虚拟化的方式使得多个操作系统和应用程序能够在一个物理主机上同时运行，而不需要物理硬件的完全隔离。

以下是一些常见的虚拟化设备：

1. **虚拟中央处理单元（Virtual CPU，vCPU）：** 每个虚拟机都分配有一个或多个虚拟 CPU，这些虚拟 CPU 由宿主机的物理 CPU 进行调度和执行。
2. **虚拟内存（Virtual Memory）：** 虚拟机使用虚拟内存来管理其内存空间，而实际的物理内存由宿主机管理。
3. **虚拟磁盘（Virtual Disk）：** 虚拟机使用虚拟磁盘来存储其操作系统和应用程序，这些虚拟磁盘可能是文件（如VMDK、VHD）或者直接访问物理存储设备。
4. **虚拟网络适配器（Virtual Network Adapter）：** 虚拟机通过虚拟网络适配器与物理网络通信，这样可以使得虚拟机之间和虚拟机与宿主机进行网络通信。
5. **虚拟图形处理单元（Virtual GPU，vGPU）：** 允许虚拟机共享物理主机的图形处理能力。
6. **虚拟输入/输出设备（Virtual I/O Devices）：** 包括键盘、鼠标、打印机等，使得虚拟机可以与这些虚拟设备进行交互。

虚拟化设备的出现使得资源可以更加灵活地分配和共享，提高了硬件的利用率，并简化了系统管理。主流的虚拟化平台包括VMware、Microsoft Hyper-V、KVM（Kernel-based Virtual Machine）等。这些平台提供了完整的虚拟化解决方案，包括虚拟化设备的模拟、管理、监控等功能。

16 用qemu启过虚拟机吗

QEMU（Quick Emulator）是一款开源的虚拟化工具，支持多种硬件体系结构。以下是使用 QEMU 启动虚拟机的一般步骤：

1. **安装 QEMU:**

   在大多数 Linux 发行版中，你可以使用包管理器安装 QEMU。例如，在 Ubuntu 中可以运行：

   ```
   sudo apt-get install qemu
   ```

2. **准备虚拟机镜像:**

   你需要一个虚拟机镜像，这是一个包含操作系统和文件系统的磁盘镜像文件。你可以使用 QEMU 自带的工具（如 `qemu-img`）创建一个虚拟机镜像。

3. **启动虚拟机:**

   使用以下命令启动虚拟机，其中 `your_image.img` 是虚拟机镜像文件：

   ```
   qemu-system-x86_64 -hda your_image.img
   ```

   上述命令中的 `-hda` 参数用于指定虚拟机的硬盘镜像。

4. **附加其他参数:**

   你可能需要根据你的需求附加其他参数，例如：

   - `-m`：设置虚拟机内存大小。
   - `-cpu`：指定虚拟机使用的 CPU 类型。
   - `-net`：配置虚拟机的网络。

   例如：

   ```
   qemu-system-x86_64 -hda your_image.img -m 512 -cpu host -net nic -net user
   ```

   以上命令将虚拟机的内存设置为512MB，使用主机的CPU，同时配置了一个网络接口。

4. 字节测开面经，瑟瑟发抖

字节测开一面

2024/1/3 一面

1.深挖实习经历

2.java的set、map、list的区别，map中key是否能为空的问题

3.java中进程和线程的区别，

4.java的锁机制

5.数据库的索引如何使用

6.SQL题，有两张表，用户表和订单表，查找每个用户购买商品的最大值

7.coding：合并两个有序数组

8.你的职业规划

9.测开中你是偏业务测试还是测试工具开发

总34分钟，有点短

古希腊掌管面试的神啊，保佑我面试必过！

2.java的set、map、list的区别，map中key是否能为空的问题

在Java中，Set、Map和List是常见的集合框架，它们分别用于不同的目的。

1. **Set（集合）**：
   - Set是一种不允许包含重复元素的集合。
   - 实现了Set接口的类通常包括HashSet、TreeSet等。
   - 不保证元素的顺序，因为它主要关心元素是否存在。

```
Set<String> stringSet = new HashSet<>();
```

2. **Map（映射）**：
   - Map是一种键值对的集合，每个键都映射到一个值。
   - 不允许重复的键，但可以有重复的值。
   - 常见的实现包括HashMap、TreeMap等。

```
Map<String, Integer> stringIntegerMap = new HashMap<>();
```

3. **List（列表）**：
   - List是一个有序的集合，允许重复元素。
   - 实现了List接口的类通常包括ArrayList、LinkedList等。
   - 可以通过索引访问元素，元素按照插入顺序排列。

```
List<String> stringList = new ArrayList<>();
```

一般来说，Map的实现对于key是不允许为null的，因为在Map中，每个键都应该是唯一的。如果允许key为null，就无法保证唯一性了。但是，在某些特殊的情况下，也有一些实现允许null作为key，例如HashMap。在使用时，要注意特定实现的规定。

3.java中进程和线程的区别，

在Java中，进程和线程是多任务处理的两个关键概念，它们有一些本质的区别。下面是它们的主要区别：

1. **定义**：
   - **进程（Process）**：是一个独立的执行单元，有独立的内存空间。一个进程包括一个或多个线程。
   - **线程（Thread）**：是一个轻量级的执行单元，是进程中的一个实体，执行任务的基本单位。
2. **资源占用**：
   - **进程**：拥有独立的内存空间，相互之间不会直接共享内存，通信需要使用特殊的机制。
   - **线程**：共享所属进程的内存空间，因此线程间的通信更加方便，但也需要考虑同步和互斥问题。
3. **启动速度和资源开销**：
   - **进程**：启动速度相对较慢，占用的资源较多。
   - **线程**：启动速度快，占用的资源相对较少。
4. **独立性**：
   - **进程**：相互独立，一个进程的崩溃不会影响其他进程。
   - **线程**：线程是进程的一部分，彼此之间共享进程的地址空间和资源，一个线程的问题可能会影响整个进程。
5. **通信机制**：
   - **进程**：通信需要使用进程间通信（Inter-Process Communication，IPC）的机制，如管道、消息队列等。
   - **线程**：线程间可以通过共享内存等轻量级的方式进行通信。
6. **切换开销**：
   - **进程**：进程切换的开销相对较大。
   - **线程**：线程切换的开销相对较小，因为它们共享了相同的地址空间。

在Java中，线程是通过`java.lang.Thread`类来实现的，而进程则是由操作系统管理的。Java也提供了`java.lang.Process`类用于处理进程相关的操作，但在Java中，通常更多地关注于线程的使用，因为线程更轻量、更容易管理。在多核处理器的情况下，线程的并行执行可以提高程序的性能。、

4.java的锁机制

在Java中，锁（Lock）是一种用于控制多线程对共享资源访问的机制，以确保在同一时刻只有一个线程可以访问共享资源，从而避免竞争条件和数据不一致性问题。Java提供了多种锁机制，其中最常见的是synchronized关键字和Java.util.concurrent包中的锁。

以下是Java中常见的锁机制：

1. **synchronized关键字**：
   - `synchronized`关键字用于创建同步方法和同步块，确保在同一时刻只有一个线程可以执行被保护的代码。
   - 它可以用于实例方法、静态方法和代码块。
   - 通过在方法上使用`synchronized`关键字，可以实现对整个方法的同步。
   - 通过在代码块内使用`synchronized`关键字，可以实现对指定的代码段的同步。

```
public synchronized void synchronizedMethod() {
    // 同步方法的代码块
}

public void someMethod() {
    synchronized (lockObject) {
        // 同步代码块
    }
}
```

2. **ReentrantLock**:
   - `ReentrantLock`是Java.util.concurrent包中提供的显式锁。
   - 与`synchronized`不同，`ReentrantLock`允许重入，即同一个线程可以多次获得同一个锁。
   - 它提供了更灵活的锁定机制，例如可中断锁、定时锁等。

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class Example {
    private final Lock lock = new ReentrantLock();

    public void someMethod() {
        lock.lock();
        try {
            // 保护的代码块
        } finally {
            lock.unlock();
        }
    }
}
```

3. **Read/Write Locks**:
   - `ReadWriteLock`接口定义了读写锁，其中包括`ReentrantReadWriteLock`的实现。
   - 读写锁允许多个线程同时读取共享资源，但只允许一个线程写入共享资源。

```
import java.util.concurrent.locks.ReadWriteLock;
import java.util.concurrent.locks.ReentrantReadWriteLock;

public class Example {
    private final ReadWriteLock readWriteLock = new
ReentrantReadWriteLock();

    public void readMethod() {
        readWriteLock.readLock().lock();
        try {
            // 读取共享资源的代码块
        } finally {
            readWriteLock.readLock().unlock();
        }
    }

    public void writeMethod() {
        readWriteLock.writeLock().lock();
        try {
```

```
            // 写入共享资源的代码块
        } finally {
            readWriteLock.writeLock().unlock();
        }
    }
}
```

这些锁机制都用于解决多线程并发访问共享资源时可能出现的问题，选择哪种锁取决于具体的需求和情境。

5.数据库的索引如何使用

数据库索引是一种优化数据库查询性能的关键工具。通过使用索引，可以快速定位和检索数据库表中的数据，降低查询的时间复杂度。以下是数据库索引的基本使用和一些建议：

1. **创建索引**：
    - 在数据库表中，可以通过在某一列或一组列上创建索引来加速查询。
    - 在大型表中，对于经常用于过滤和排序的列，索引的作用更为显著。
    - 常见的索引类型包括单列索引、组合索引和唯一索引。

```
-- 创建单列索引
CREATE INDEX index_name ON table_name(column_name);

-- 创建组合索引
CREATE INDEX index_name ON table_name(column1, column2);

-- 创建唯一索引
CREATE UNIQUE INDEX index_name ON table_name(column_name);
```

2. **选择合适的列**：
    - 选择适当的列进行索引是很重要的，通常选择经常用于查询条件的列。
    - 避免对经常修改的列创建过多索引，因为每次修改都会涉及到索引的更新。
3. **避免创建过多的索引**：
    - 虽然索引可以提高查询性能，但过多的索引也会增加插入、更新和删除操作的开销。
    - 在创建索引时要权衡查询性能和维护成本。
4. **使用覆盖索引**：
    - 覆盖索引是指索引包含了查询所需的所有数据，而不需要回表到实际数据行。
    - 这样可以减少IO操作，提高查询效率。
5. **定期维护和优化索引**：
    - 定期检查数据库中的索引性能，删除不再需要的索引，重新构建或重新组织现有索引。
    - 数据库系统通常提供了优化工具和命令，如`ANALYZE TABLE`、`OPTIMIZE TABLE`等。

```
-- 分析表的索引信息
ANALYZE TABLE table_name;

-- 优化表的索引
OPTIMIZE TABLE table_name;
```

6. **理解查询优化器**：
    - 数据库查询优化器会根据查询条件、表结构和索引情况来选择最优的执行计划。
    - 使用`EXPLAIN`语句可以查看查询执行计划，帮助理解优化器的决策过程。

```
EXPLAIN SELECT * FROM table_name WHERE column_name = 'value';
```

7. **考虑使用全文索引**：
    - 对于包含文本数据的列，可以考虑使用全文索引（Full-Text Indexing）来支持全文搜索。
    - 全文索引可以更有效地处理文本搜索查询。

```
CREATE FULLTEXT INDEX index_name ON table_name(column_name);
```

数据库索引的使用需要根据具体的业务场景和查询需求来进行优化。合理的索引设计可以明显提高数据库查询性能，但不当使用可能会导致性能下降。在设计和维护索引时，需要深入了解数据库引擎的特性和查询优化原理。

6.SQL题，有两张表，用户表和订单表，查找每个用户购买商品的最大值

假设有两张表，一张是用户表（`users`），包含用户的信息，另一张是订单表（`orders`），记录了用户的订单信息。假设订单表中有两个字段：`user_id`表示用户ID，`amount`表示订单金额。

要查找每个用户购买商品的最大值，可以使用以下SQL查询：

```
SELECT
    u.user_id,
    u.username,
    MAX(o.amount) AS max_purchase_amount
FROM
    users u
JOIN
    orders o ON u.user_id = o.user_id
GROUP BY
    u.user_id, u.username;
```

这个查询使用了`JOIN`操作将用户表和订单表关联在一起，然后使用`GROUP BY`按用户分组。最后，使用`MAX(o.amount)`计算每个用户购买商品的最大值，并将结果呈现在查询的结果集中。

7.coding：合并两个有序数组

合并两个有序数组是一个常见的算法问题。以下是一个Java示例代码，演示了如何合并两个有序数组：

```
public class MergeSortedArrays {

    public static void merge(int[] nums1, int m, int[] nums2, int n) {
        // 初始化两个数组的末尾索引
        int i = m - 1;  // nums1的末尾
        int j = n - 1;  // nums2的末尾

        // 合并后的数组的末尾
        int mergeIndex = m + n - 1;

        // 从后往前比较并合并两个数组的元素
        while (i >= 0 && j >= 0) {
            if (nums1[i] > nums2[j]) {
                nums1[mergeIndex] = nums1[i];
                i--;
            } else {
                nums1[mergeIndex] = nums2[j];
                j--;
            }
            mergeIndex--;
        }

        // 将nums2中剩余的元素复制到nums1中
        while (j >= 0) {
            nums1[mergeIndex] = nums2[j];
            j--;
            mergeIndex--;
        }
    }

    public static void main(String[] args) {
        // 示例
        int[] nums1 = {1, 2, 3, 0, 0, 0};
        int m = 3;
        int[] nums2 = {2, 5, 6};
        int n = 3;

        merge(nums1, m, nums2, n);

        // 输出合并后的结果
        for (int num : nums1) {
            System.out.print(num + " ");
        }
    }
}
```

在这个例子中，使用双指针的方法，从两个数组的末尾开始比较元素，并将较大的元素放在合并后的数组的末尾。最后，如果第二个数组中还有元素未处理，将其复制到合并后的数组中。这样可以实现在不使用额外空间的情况下合并两个有序数组。

分享者：小牛哥永不退缩

5. 京东实习后端面经，被问麻了...

1.mysql联合索引失效（简历和项目上没用mysql还是问了）
2.索引底层
3.说说分库分表
4.mysql事务
5.MQ幂等性
6.了解分布式吗
7.spring类如何使用的
8.让我选一个掌握深的知识 java基础、中间件
选了基础的集合，让我换redis

反问
第一次面试 #面经#
不到30分钟，寄

1.mysql联合索引失效

MySQL联合索引失效通常发生在以下情况：

1. **未使用最左前缀原则：** 联合索引的最左前缀原则指的是在查询中使用索引的时候，必须从联合索引的最左边列开始使用。如果查询不按照最左前缀的顺序使用索引，MySQL 可能无法充分利用这个索引。

   例如，如果有一个联合索引 (col1, col2)，而你的查询中只使用了 col2，那么这个索引就不会被用到。

2. **不符合索引顺序的查询：** 联合索引只在查询中按照索引的顺序使用时才会生效。如果你的查询中的条件不按照索引的顺序，MySQL 可能无法使用这个索引。

   例如，如果有一个联合索引 (col1, col2)，而你的查询中使用了 col2，col1 进行筛选，那么这个索引也可能无法生效。

3. **数据分布不均匀：** 如果索引列的数据分布不均匀，即某些值的出现频率远高于其他值，那么 MySQL 可能会选择不使用索引，而是进行全表扫描。

4. **数据类型不匹配：** 联合索引的列如果包含了不同的数据类型，可能会导致索引失效。例如，如果一个列是字符串类型，另一个列是数字类型，MySQL 在执行查询时可能无法有效使用这个索引。

5. **使用了函数或运算符：** 如果在查询条件中使用了函数或运算符，可能会导致索引失效。MySQL 无法直接在索引上执行函数或运算符，因此可能会选择不使用索引。

为了避免这些问题，设计索引时应根据实际查询需求考虑最佳的索引顺序，并确保查询条件与索引一致。同时，确保索引列的数据类型和函数的使用方式符合索引的要求。在优化查询性能时，可以使用 `EXPLAIN` 来查看 MySQL 的执行计划，以确定是否正确使用了索引。

2.索引底层

MySQL索引的底层实现主要依赖于B+树和哈希索引两种结构。

1. **B+树索引：**

- **B+树结构：** B+树是B树的一种变体，它在B树的基础上进行了优化。在B+树中，只有叶子节点存储实际的数据，而非叶子节点仅包含键值信息用于导航。这样的结构使得范围查询更加高效。InnoDB使用的索引即为B+树索引。

2. **哈希索引：**
   - **哈希结构：** 哈希索引是将键的值通过哈希函数转换成一个固定大小的哈希值，然后将哈希值与存储实际数据的位置相对应。这种结构使得单个键的查找效率很高，但不支持范围查询。InnoDB不直接支持哈希索引，但可以通过Memory存储引擎来创建哈希索引。

在InnoDB存储引擎中，使用的是B+树索引。每个InnoDB表都有一个聚簇索引（clustered index），这是数据行的物理排序。聚簇索引在InnoDB中实际上就是一颗B+树，因此表的数据行实际上是按照聚簇索引的排序存储的。

此外，InnoDB中的辅助索引也是B+树结构，这些辅助索引的叶子节点包含了指向聚簇索引中对应行的引用。

MySQL通过使用不同的索引结构，使得它在处理各种类型的查询和数据操作时能够取得较好的性能。选择适当的索引结构取决于具体的应用场景和查询需求。

3.说说分库分表

分库分表是一种数据库水平拆分的策略，旨在应对数据库规模不断增长导致的性能瓶颈和存储容量限制。这种策略将一个大型数据库拆分成多个较小的数据库实例（分库）和表（分表），从而提高并行处理能力、降低单个数据库实例的负载，提升整体系统的性能和可伸缩性。

以下是关于分库分表的一些要点：

1. **水平拆分：** 分库分表是水平拆分的一种形式，与垂直拆分（按照业务模块划分表）不同。水平拆分指的是将一个表中的数据按照某个规则分散存储在多个数据库实例或表中。
2. **分库：** 将原来的单个数据库拆分成多个独立的数据库。每个数据库实例都可以独立运行在不同的物理服务器上，从而提高并发处理能力。分库通常采用数据划分的规则，比如按照用户ID的取模或哈希值进行分库。
3. **分表：** 在每个数据库中，原来的大表被拆分成多个较小的表。分表的常见策略包括按照时间范围、按照业务模块或按照数据范围等。分表可以减小每个表的数据量，提高查询性能。
4. **数据一致性：** 在分库分表的架构中，需要考虑数据一致性的问题。因为数据分散在不同的数据库和表中，应用程序需要确保对跨库、跨表的事务有良好的处理机制。
5. **全局唯一标识：** 在分库分表的情况下，通常需要一个全局唯一的标识符，以便在多个数据库和表中唯一标识一条记录。这可以是通过分布式ID生成器、UUID等方式实现的。
6. **路由策略：** 应用程序需要实现合适的路由策略，确保查询时能够正确定位到需要的数据库和表。路由策略可以基于数据划分的规则，例如用户ID的哈希值。
7. **动态扩展：** 分库分表的优势之一是能够相对容易地进行水平扩展。当数据量增加时，可以通过增加新的数据库实例或表来进行扩展，而不必改变整体系统架构。

分库分表的实现需要谨慎考虑业务需求、数据一致性、查询性能和扩展性等因素。这样的架构适用于大型、高并发的应用，但也带来了管理和维护的挑战。

4.mysql事务

MySQL事务和InnoDB引擎的实现方式涉及到ACID属性（原子性、一致性、隔离性、持久性）的保障。

MySQL事务：

1. **事务的开始和结束：** 事务是由 `BEGIN`、`COMMIT` 和 `ROLLBACK` 语句来标识的。`BEGIN` 标志事务的开始，`COMMIT` 标志事务的结束并提交，而 `ROLLBACK` 则用于事务的回滚。
2. **隔离级别：** MySQL支持不同的事务隔离级别，包括READ UNCOMMITTED、READ COMMITTED、REPEATABLE READ和SERIALIZABLE。这些级别决定了在事务执行过程中，对数据

的读写是否会受到其他事务的影响。

InnoDB引擎：

1. **事务日志（Redo Log）：** InnoDB引擎使用事务日志（redo log）来实现事务的持久性。当事务对数据库进行更改时，这些更改首先被写入事务日志，然后再写入磁盘上的数据页。这样即使在事务提交之前系统崩溃，通过重放事务日志，可以将数据库恢复到崩溃前的状态。
2. **两阶段提交（Two-Phase Commit）：** InnoDB引擎使用两阶段提交来确保事务的原子性。在第一阶段，事务的更改被写入事务日志，但尚未将其应用到磁盘上的数据页。在第二阶段，如果事务成功完成，所有的更改都被写入磁盘；如果事务失败，所有的更改都会被回滚。
3. **MVCC（多版本并发控制）：** InnoDB使用MVCC来提供不同的事务隔离级别。MVCC通过在每行数据上维护多个版本，使得事务可以在不互相干扰的情况下并发执行。每个事务可以看到数据库在它开始之前已经提交的版本，但不会看到其他事务正在执行的版本。
4. **行级锁：** InnoDB引擎支持行级锁，这意味着事务在修改数据时只锁定涉及的行，而不是锁定整个表。这提高了并发性，减少了锁竞争，但也引入了一些额外的开销。

总体来说，InnoDB引擎通过事务日志、两阶段提交、MVCC和行级锁等机制，实现了对事务的可靠支持，保证了ACID属性的满足。这使得InnoDB成为MySQL中常用的事务型存储引擎。

5.MQ幂等性

消息队列（MQ）中的幂等性是指无论消息被消费多少次，最终的效果应该是一致的。在分布式系统中，由于网络延迟、重试机制等原因，消息可能被处理多次，而幂等性可以确保系统的一致性。

实现MQ幂等性的方法：

1. **唯一标识符：** 每条消息都应该包含一个唯一的标识符（Message ID），该标识符可以用来检测消息的重复。当消息处理完成后，可以记录已处理的消息标识符，下次收到相同标识符的消息时，直接判断为重复消息，不再处理。
2. **幂等性检测逻辑：** 在消息的处理逻辑中加入幂等性检测的逻辑，确保同一消息在多次处理时不会产生不一致的结果。例如，在数据库中存储已处理消息的信息，每次处理消息前检查是否已经处理过。
3. **版本号：** 每个消息可以包含一个版本号，表示消息的版本信息。消费者在处理消息时，可以检查消息的版本号，如果版本号相同，即使消息重复到达，也可以认为是幂等的。
4. **幂等性接口设计：** 对于消息的消费者，可以设计幂等性接口，确保相同输入产生相同的输出。通过合理设计接口，消费者可以在处理相同消息时返回相同结果，保证了幂等性。
5. **事务性操作：** 如果消息的处理涉及到事务性操作，可以利用数据库事务的特性来保证幂等性。通过将消息的处理和数据库的更新放在同一个事务中，可以确保即使消息多次到达，数据库的状态也能正确更新。
6. **分布式锁：** 使用分布式锁机制，确保在同一时间只有一个消费者能够处理消息。这可以避免多个消费者同时处理同一消息，从而保证幂等性。

实现MQ的幂等性需要结合业务场景和具体的消息处理逻辑，采用合适的方式来防止重复处理相同的消息。选择合适的幂等性策略可以提高系统的可靠性和一致性。

6.了解分布式吗

分布式系统是由多台计算机（或节点）协同工作，共同完成某个任务或提供某种服务。分布式系统常用于处理大规模数据、提高系统的可扩展性和可靠性。以下是一些常用的分布式技术和框架：

1. **分布式存储系统：**
   - **Hadoop Distributed File System (HDFS):** 用于存储和处理大规模数据的分布式文件系统，是Apache Hadoop的核心组件。
   - **Amazon S3:** 亚马逊提供的分布式对象存储服务，适用于在云环境中存储和检索数据。
2. **分布式计算框架：**

- **Apache Spark:** 用于大规模数据处理的分布式计算框架，支持内存计算，适用于迭代式算法和复杂的数据流处理。
- **Apache Flink:** 分布式流处理引擎，支持事件时间处理和状态管理，适用于实时数据处理和分析。

3. **分布式消息队列：**
   - **Apache Kafka:** 高吞吐量的分布式消息系统，用于构建实时数据流平台，支持发布/订阅模型。
   - **RabbitMQ:** 开源的消息代理，实现了高级消息队列协议（AMQP），用于在分布式系统中传递消息。
4. **分布式数据库：**
   - **Apache Cassandra:** 高度可扩展的分布式 NoSQL 数据库，适用于处理大规模的分布式数据。
   - **MongoDB:** 非关系型的分布式数据库，支持水平扩展，适用于处理大量文档型数据。
5. **分布式缓存：**
   - **Redis:** 高性能的内存键值存储系统，用于缓存和数据存储。
   - **Memcached:** 分布式内存对象缓存系统，用于加速动态Web应用程序。
6. **容器与编排：**
   - **Docker:** 轻量级容器技术，用于打包、分发和运行应用程序。
   - **Kubernetes:** 开源的容器编排引擎，用于自动化容器的部署、扩展和管理。
7. **分布式事务：**
   - **XA协议:** 提供了一种分布式事务的协议，可以在不同数据库管理系统之间确保事务的一致性。
   - **TCC（Try-Confirm-Cancel）：**一种补偿型分布式事务的设计模式，通过预留资源、确认和取消操作来保证事务的一致性。
8. **服务发现与治理：**
   - **Consul:** 分布式服务发现和治理工具，用于服务注册、健康检查和动态路由。
   - **etcd:** 分布式键值存储系统，常用于服务发现和配置管理。

这些技术和框架在不同领域解决了分布式系统面临的各种问题，提供了可靠的基础设施和工具，帮助开发者构建高性能、可扩展和可靠的分布式应用。

7.spring类如何使用的

Spring框架提供了众多的类和功能，用于开发各种类型的应用程序，包括依赖注入、AOP（面向切面编程）、事务管理、数据访问、Web开发等。以下是Spring中一些核心类的使用方式：

1. **ApplicationContext：** ApplicationContext 是 Spring 中用于管理Bean的上下文对象。它可以通过配置文件或Java配置类加载并管理应用程序中的所有Bean。

```
// 通过XML配置文件加载ApplicationContext
ApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml");

// 或者通过Java配置类加载ApplicationContext
AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext(AppConfig.class);

// 通过ApplicationContext获取Bean
MyService myService = context.getBean(MyService.class);
```

2. **Bean：** 在Spring中，Bean是由Spring容器管理的对象。通过配置文件或注解标记类为Bean，Spring容器负责实例化、配置和管理Bean。

```
// 在XML配置文件中定义Bean
<bean id="myBean" class="com.example.MyBean"/>

// 在Java配置类中定义Bean
@Configuration
public class AppConfig {
    @Bean
    public MyBean myBean() {
        return new MyBean();
    }
}

// 使用@Component注解定义Bean
@Component
public class MyComponent {
    // ...
}
```

3. **依赖注入：** Spring通过依赖注入来管理对象之间的关系。可以通过构造函数、setter方法或字段注入的方式将依赖关系注入到Bean中。

```
// 通过构造函数注入
public class MyClass {
    private MyDependency myDependency;

    public MyClass(MyDependency myDependency) {
        this.myDependency = myDependency;
    }
}

// 通过setter方法注入
public class MyClass {
    private MyDependency myDependency;

    public void setMyDependency(MyDependency myDependency) {
        this.myDependency = myDependency;
    }
}
```

4. **AOP（面向切面编程）：** Spring的AOP模块提供了面向切面编程的能力，通过定义切面和切点，可以在应用程序中插入横切关注点。

```
// 定义切面
@Aspect
public class MyAspect {
    @Before("execution(* com.example.MyService.*(..))")
    public void beforeMethod() {
        // 在方法执行前执行的逻辑
    }
}
```

5. **事务管理：** Spring提供了对声明式事务的支持，通过配置可以简化事务管理的过程。

```
// 在XML配置文件中声明事务
<tx:advice id="txAdvice" transaction-manager="transactionManager">
    <tx:attributes>
        <tx:method name="*" propagation="REQUIRED" />
    </tx:attributes>
</tx:advice>

<aop:config>
    <aop:pointcut id="serviceMethods" expression="execution(*
com.example.*Service.*(..))" />
    <aop:advisor advice-ref="txAdvice" pointcut-ref="serviceMethods" />
</aop:config>
```

以上是一些Spring中核心类的基本使用方式。Spring框架提供了广泛的文档和示例，可以根据具体应用场景和需求深入学习和使用不同的功能。Spring框架提供了众多的类和功能，用于开发各种类型的应用程序，包括依赖注入、AOP（面向切面编程）、事务管理、数据访问、Web开发等。以下是Spring中一些核心类的使用方式：

1. **ApplicationContext：** ApplicationContext 是 Spring 中用于管理Bean的上下文对象。它可以通过配置文件或Java配置类加载并管理应用程序中的所有Bean。

```
// 通过XML配置文件加载ApplicationContext
ApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml");

// 或者通过Java配置类加载ApplicationContext
AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext(AppConfig.class);

// 通过ApplicationContext获取Bean
MyService myService = context.getBean(MyService.class);
```

2. **Bean：** 在Spring中，Bean是由Spring容器管理的对象。通过配置文件或注解标记类为Bean，Spring容器负责实例化、配置和管理Bean。

```
// 在XML配置文件中定义Bean
<bean id="myBean" class="com.example.MyBean"/>

// 在Java配置类中定义Bean
@Configuration
public class AppConfig {
    @Bean
    public MyBean myBean() {
        return new MyBean();
    }
}

// 使用@Component注解定义Bean
@Component
public class MyComponent {
    // ...
}
```

3. **依赖注入：** Spring通过依赖注入来管理对象之间的关系。可以通过构造函数、setter方法或字段注入的方式将依赖关系注入到Bean中。

```
// 通过构造函数注入
public class MyClass {
    private MyDependency myDependency;

    public MyClass(MyDependency myDependency) {
        this.myDependency = myDependency;
    }
}

// 通过setter方法注入
public class MyClass {
    private MyDependency myDependency;

    public void setMyDependency(MyDependency myDependency) {
        this.myDependency = myDependency;
    }
}
```

4. **AOP（面向切面编程）：** Spring的AOP模块提供了面向切面编程的能力，通过定义切面和切点，可以在应用程序中插入横切关注点。

```
// 定义切面
@Aspect
public class MyAspect {
    @Before("execution(* com.example.MyService.*(..))")
    public void beforeMethod() {
        // 在方法执行前执行的逻辑
    }
}
```

5. **事务管理：** Spring提供了对声明式事务的支持，通过配置可以简化事务管理的过程。

```
// 在XML配置文件中声明事务
<tx:advice id="txAdvice" transaction-manager="transactionManager">
    <tx:attributes>
        <tx:method name="*" propagation="REQUIRED" />
    </tx:attributes>
</tx:advice>

<aop:config>
    <aop:pointcut id="serviceMethods" expression="execution(*
com.example.*Service.*(..))" />
    <aop:advisor advice-ref="txAdvice" pointcut-ref="serviceMethods" />
</aop:config>
```

[原文连接]

6. 阿里国际二面面经，八股文好难呀

阿里国际二面面经

还以为一面凉了，居然收到了二面的通知，总结记录一下

1.自我介绍
2.hashcode和equals
3.hashmap扩容，线程是否安全
4.怎么样的hashmap线程安全，怎么实现，currenthashmap和Hashtable的区别
5.什么是死锁，怎么发现死锁了，怎么避免死锁
6.new怎么回收，垃圾回收机制
7.怎么判断是否还在引用
8.什么情况下回收到老年代
9.快排的原理
10.在10亿条数据里找出最大的一百条用什么方法（我说了堆排序的实现过程）
11.mysql的事物隔离级别和区别
12.索引的底层，为什么用B+树（讲了一下从二叉树开始进化到B+树的过程）
12.回表查询（说了一大堆的场景，其实就是问这个）
13.redis为什么快，怎么持久化
14.拷打项目
15.做一道中等难度的算法题，做完之后优化时间复杂度，我把用链表实现变成了用树实现

2.hashcode和equals的区别

`hashCode` 和 `equals` 是 Java 中用于处理对象相等性的两个方法：

1. **equals方法：**
   - `equals` 方法是用来比较两个对象是否在逻辑上相等。根据类的设计，你可以重写 `equals` 方法来指定自定义的相等性判断规则。
   - 一般来说，如果你重写了 `equals` 方法，就应该同时重写 `hashCode` 方法，以保持一致性。两个对象在调用 `equals` 方法返回 true 时，它们的 `hashCode` 值应该相等。

```
@Override
public boolean equals(Object obj) {
    // 实现相等性判断的逻辑
}
```

2. **hashCode方法：**
   - `hashCode` 方法用于返回对象的哈希码，它是一个整数值。哈希码的作用是在进行集合操作时提高查找效率，例如在哈希表中存储对象。
   - 如果两个对象使用 `equals` 方法返回 true，它们的 `hashCode` 值应该相等。但是反过来并不成立，即相等的对象不一定具有相等的哈希码。
   - 如果不重写 `equals` 方法，Java 默认使用 `Object` 类的实现，它比较的是对象的引用地址，而不是逻辑相等性。

```
@Override
public int hashCode() {
    // 实现生成哈希码的逻辑
}
```

总体来说，`equals` 主要用于判断两个对象是否在逻辑上相等，而 `hashCode` 用于支持对对象的高效存储和检索。在使用自定义类作为集合的键时，确保正确实现这两个方法是很重要的。

3.hashmap扩容，线程是否安全

`HashMap` 是 Java 中常用的集合类之一，用于存储键值对。

扩容机制：

1. **初始容量和负载因子：**
   - `HashMap` 有一个初始容量和一个负载因子。初始容量是哈希表在创建时的容量，默认为 16。负载因子是一个在哈希表大小超过其容量乘以负载因子时，哈希表将进行扩容的阈值，默认为 0.75。
2. **扩容操作：**
   - 当哈希表的元素个数达到了负载因子所定义的阈值，`HashMap` 将进行扩容操作。
   - 扩容操作包括创建一个新的哈希表，其容量为原容量的两倍，然后将所有元素重新分配到新的哈希表中。

线程安全性：

1. **非线程安全：**
   - `HashMap` 不是线程安全的，即在多线程环境下，如果有一个线程在修改 `HashMap`，而另一个线程同时访问它，可能会导致不确定的结果。
2. **线程安全解决方案：**
   - 如果需要在多线程环境中使用线程安全的 `HashMap`，可以考虑使用 `ConcurrentHashMap`。`ConcurrentHashMap` 提供了一些线程安全的操作，例如 `putIfAbsent`、`replace` 等，以及分段锁的机制，以提高并发性能。

`HashMap` 的扩容机制是为了保持哈希表的性能， `HashMap` 本身不是线程安全的。如果需要在多线程环境中使用，可以使用 `ConcurrentHashMap` 或者在并发访问时进行适当的同步处理。

4.怎么样的hashmap线程安全，怎么实现，currenthashmap和Hashtable的区别

在 Java 中，有几种实现线程安全的 `HashMap` 的方式。其中，`ConcurrentHashMap` 和 `Hashtable` 是两种常见的线程安全的 `Map` 实现：

1. ConcurrentHashMap:

- **分段锁机制：**
  - `ConcurrentHashMap` 使用了分段锁机制。它将整个数据集分成多个段（segment），每个段拥有自己的锁。这样在多线程环境中，不同的线程可以同时访问不同的段，提高并发性能。
- **并发度高：**
  - 因为不同的线程可以同时操作不同的段，`ConcurrentHashMap` 具有较高的并发度，适合在高并发的场景中使用。
- **不支持 null 键和值：**

- `ConcurrentHashMap` 不支持存储 null 键和值。

2. Hashtable:

- **同步整个数据结构：**
  - `Hashtable` 使用同步方法或同步块来确保线程安全。所有的读写操作都是同步的，这意味着在多线程环境下，只能有一个线程访问 `Hashtable`，可能导致性能瓶颈。
- **支持 null 键和值：**
  - `Hashtable` 支持存储 null 键和值。
- **遗留类：**
  - `Hashtable` 是一个遗留的类，而 `ConcurrentHashMap` 是 Java 5 引入的，更加现代化和高效。

区别总结：

- `ConcurrentHashMap` 使用了分段锁机制，具有更好的并发性能，适合高并发场景。
- `Hashtable` 使用同步方法，整个数据结构同步，可能导致性能瓶颈。
- `ConcurrentHashMap` 不支持存储 null 键和值，而 `Hashtable` 支持。

通常推荐使用 `ConcurrentHashMap`，因为它相对于 `Hashtable` 具有更好的性能。如果需要支持 null 键或值，可以考虑使用 `HashMap` 并在访问时进行适当的同步处理。

5.什么是死锁，怎么发现死锁了，怎么避免死锁

死锁是指两个或多个线程在互相等待对方释放资源而无法继续执行的状态。当线程之间存在循环等待资源的情况，且每个线程都持有对方所需资源的锁时，就可能发生死锁。

如何发现死锁：

1. **线程转储（Thread Dump）：**
   - 通过获取应用程序的线程转储，可以查看每个线程的状态以及持有的锁信息。死锁通常表现为一组线程互相等待。
2. **监控工具：**
   - 使用监控工具，如 Java VisualVM 或类似的性能分析工具，可以观察线程的状态、锁信息等。

如何避免死锁：

1. **按序获取锁：**
   - 确保所有线程按照相同的顺序获取锁。这样可以避免循环等待的情况。
2. **使用尝试获取锁：**
   - 在获取锁的时候，使用尝试获取锁的方式，而不是一直等待。可以设置一个超时时间，在超时后放弃获取锁。
3. **使用锁的层次性：**
   - 设计锁的层次性，确保所有线程按照一定的层次获取锁，避免发生循环等待。
4. **避免长时间持有锁：**
   - 尽量减小持有锁的时间，避免在持有锁的时候执行复杂、耗时的操作。
5. **定期检查：**
   - 定期检查系统中是否存在死锁，及时发现并处理。
6. **使用工具进行分析：**
   - 利用死锁检测工具，如 `jstack`、`jconsole` 等，来监控和分析系统中的死锁情况。
7. **合理设计并发结构：**

- 在设计并发结构时，考虑线程安全性，避免设计上的瑕疵导致死锁的发生。

6.new怎么回收，垃圾回收机制

在 Java 中，通过 `new` 关键字创建的对象，其内存的管理和回收是由Java虚拟机（JVM）的垃圾回收机制来负责的。

1. 垃圾回收的原理：

- Java 使用自动内存管理，通过垃圾回收机制来释放不再被引用的对象占用的内存。
- 当对象不再被任何引用指向时，它就成为垃圾，垃圾回收机制将识别并释放这部分内存。

2. 垃圾回收的方式：

- **标记-清除算法：**
  - 垃圾回收器会通过根对象开始遍历整个对象图，标记所有被引用的对象。然后清除所有未标记的对象，释放它们的内存。
- **复制算法：**
  - 将内存分为两块，每次只使用其中一块。当一块内存用尽时，将存活的对象复制到另一块内存中，然后清除当前内存中的所有对象。
- **标记-整理算法：**
  - 结合了标记-清除和复制算法的优点。首先标记所有存活的对象，然后将它们向一端移动，清理掉端边界以外的对象，最后释放内存。

3. 垃圾回收器的类型：

- **Serial 收集器：**
  - 单线程执行，适用于小型应用或者客户端应用。
- **Parallel 收集器：**
  - 多线程执行，适用于多核处理器，关注吞吐量。
- **CMS（Concurrent Mark-Sweep）收集器：**
  - 以最短停顿时间为目标，主要用于对响应时间有较高要求的应用。
- **G1（Garbage First）收集器：**
  - 面向服务端应用，将堆内存划分为多个区域，进行垃圾回收。

4. 手动内存管理和 `new` 的回收：

- Java 中，开发者无需手动释放通过 `new` 创建的对象。垃圾回收器会自动检测不再被引用的对象并回收它们所占用的内存。
- 开发者需要注意的是，及时释放对对象的引用可以加速垃圾回收的进行，提高程序的性能。

Java的垃圾回收机制为开发者提供了方便的内存管理手段，通过不同的垃圾回收算法和收集器，可以满足不同应用场景的需求。

7.怎么判断是否还在引用

在Java中，可以使用垃圾回收机制来判断一个对象是否还在引用。当一个对象没有任何引用指向它时，垃圾回收机制会将其标记为可回收的垃圾对象，并在适当的时候进行回收。

然而，我们也可以通过代码来判断一个对象是否还在引用。以下是几种常见的方法：

1. 引用计数法：为对象添加一个引用计数器，每当有一个引用指向该对象时，计数器加1；当引用失效时，计数器减1。当计数器为0时，表示该对象没有任何引用指向它，即可判断对象不再被引用。

2. 可达性分析算法：通过判断对象是否可达来确定是否还有引用指向它。Java中的垃圾回收器使用的就是这种算法。当一个对象不再被任何根对象（如静态变量、局部变量等）引用时，即可判断对象不再被引用。

下面是一个示例代码，演示了如何判断一个对象是否还在引用：

```
复制public class ObjectReferenceExample {
    public static void main(String[] args) {
        Object obj = new Object(); // 创建一个对象
        Object ref = obj; // 引用指向该对象

        obj = null; // 将对象的引用置为null

        if (ref != null) {
            System.out.println("对象还在引用中");
        } else {
            System.out.println("对象已不再引用");
        }
    }
}
```

在上述示例中，当将`obj`的引用置为null后，通过判断`ref`是否为null，可以确定对象是否还在引用。如果输出结果为"对象已不再引用"，则表示对象不再被引用。

需要注意的是，Java的垃圾回收机制是自动进行的，我们无法手动控制对象的回收时机。因此，判断一个对象是否还在引用只是一种判断手段，具体的对象回收由垃圾回收机制负责。

8.什么情况下回收到老年代

在Java中，对象的内存分配和回收是由垃圾回收器（Garbage Collector）负责的。垃圾回收器会根据对象的存活时间和内存分配情况，将对象分配到不同的内存区域，包括新生代（Young Generation）和老年代（Old Generation）。

一般情况下，新创建的对象会被分配到新生代的Eden区域。当Eden区域满时，会触发一次Minor GC（新生代垃圾回收），将存活的对象复制到Survivor区域。经过多次Minor GC后，仍然存活的对象会被晋升到老年代。

除了新生代中的对象晋升到老年代外，还有以下情况会导致对象直接分配到老年代：

1. 大对象直接分配：如果需要分配的对象大小超过了老年代的阈值（通过参数-XX:PretenureSizeThreshold设置），则该对象会直接在老年代分配。
2. 长期存活的对象：如果一个对象经过多次Minor GC后仍然存活，那么它会被晋升到老年代。这是因为老年代的对象存活时间更长，可以容纳那些长期存活的对象。
3. 动态年龄判定：在Survivor区域中，对象经过多次Minor GC后仍然存活的，会根据其年龄（通过参数-XX:MaxTenuringThreshold设置）决定是否晋升到老年代。

对象被分配到老年代并不意味着它会立即被回收。老年代的垃圾回收通常是通过Major GC（Full GC）来进行的，它会对整个堆内存进行回收，包括新生代和老年代。Major GC的触发条件和具体行为会根据不同的垃圾回收器和参数设置而有所不同。

总之，对象在Java中被分配到老年代的情况包括：大对象直接分配、长期存活的对象和经过多次Minor GC后仍然存活的对象。

9.快排的原理

- 快速排序（Quicksort）是一种常用的排序算法，它的原理基于分治法（Divide and Conquer）。快速排序的基本思想是选择一个基准元素，通过一趟排序将待排序的序列分割成独立的两部分，其中一部分的所有元素都比基准元素小，另一部分的所有元素都比基准元素大，然后对这两部分继续进行排序，最终得到一个有序序列。

  具体的快速排序算法步骤如下：

  1. 选择一个基准元素（通常选择序列的第一个元素）。
  2. 设定两个指针，一个指向序列的起始位置，称为左指针（left），另一个指向序列的末尾位置，称为右指针（right）。
  3. 左指针向右移动，直到找到一个大于等于基准元素的元素。
  4. 右指针向左移动，直到找到一个小于等于基准元素的元素。
  5. 如果左指针小于等于右指针，则交换左右指针所指向的元素。
  6. 重复步骤3到步骤5，直到左指针大于右指针。
  7. 将基准元素与右指针所指向的元素进行交换，此时基准元素左边的元素都小于基准元素，右边的元素都大于基准元素。
  8. 对基准元素左边的子序列和右边的子序列分别进行递归调用快速排序算法。

  通过不断地划分子序列并对子序列进行排序，最终可以得到整个序列的有序排列。

  快速排序的时间复杂度为O(nlogn)，其中n为待排序序列的长度。它是一种原地排序算法，不需要额外的辅助空间，但是在最坏情况下（序列已经有序或逆序），时间复杂度可能达到O(n^2)。为了避免最坏情况的发生，可以采用随机选择基准元素或者三数取中法来选择基准元素，以提高算法的性能。

10.在10亿条数据里找出最大的一百条用什么方法

在10亿条数据中找出最大的一百条数据，可以使用堆排序（Heap Sort）的方法来解决。堆排序是一种基于二叉堆的排序算法，它可以在O(nlogk)的时间复杂度内找出最大的k个元素。

具体步骤如下：

1. 创建一个大小为100的最小堆（Min Heap）。
2. 遍历10亿条数据，对于每个数据项，执行以下操作：
   - 如果堆的大小小于100，将数据项插入堆中。
   - 如果堆的大小已经达到100，比较当前数据项与堆顶元素的大小：
     - 如果当前数据项大于堆顶元素，将堆顶元素替换为当前数据项，并进行堆的调整操作，以保持最小堆的性质。
     - 如果当前数据项小于等于堆顶元素，则忽略该数据项。
3. 遍历完所有数据后，最小堆中的100个元素即为最大的一百条数据。

通过使用最小堆，我们可以始终保持堆中的元素是当前最大的k个元素。每次插入新的数据项时，如果堆已满且新数据项比堆顶元素大，则替换堆顶元素，并进行堆的调整操作。这样，最终堆中的元素就是最大的k个元素。

需要注意的是，由于数据量非常大，内存可能无法一次性加载所有数据。因此，可以采用分批加载数据的方式，每次加载一部分数据进行处理，直到遍历完所有数据。

另外，为了进一步提高性能，可以使用多线程或分布式处理的方式来并行处理数据，加快查找最大的一百条数据的速度。

11.mysql的事物隔离级别和区别

MySQL支持多个事务隔离级别，用于控制并发事务的隔离程度。不同的隔离级别提供了不同的数据一致性和并发性保证。以下是MySQL中常见的四个事务隔离级别及其区别：

1. 读未提交（Read Uncommitted）：
    - 最低的隔离级别，事务中的修改可以被其他事务立即读取。
    - 存在脏读（Dirty Read）问题，即一个事务读取到了另一个事务未提交的数据。
    - 可能导致不可重复读和幻读问题。
2. 读已提交（Read Committed）：
    - 保证一个事务提交后，其他事务才能读取到其修改的数据。
    - 解决了脏读问题，但仍可能出现不可重复读和幻读问题。
3. 可重复读（Repeatable Read）：
    - 默认的隔离级别，保证在同一个事务中多次读取同一数据时，结果始终一致。
    - 通过多版本并发控制（MVCC）机制实现，读取的是事务开始时的快照数据。
    - 解决了脏读和不可重复读问题，但仍可能出现幻读问题。
4. 串行化（Serializable）：
    - 最高的隔离级别，确保事务串行执行，避免了脏读、不可重复读和幻读问题。
    - 通过对读取的数据加锁实现，保证了最高的数据一致性，但并发性较差。

需要注意的是，隔离级别越高，数据一致性越好，但并发性能也会降低。选择合适的隔离级别需要根据具体业务需求和并发访问情况进行权衡。

在MySQL中，可以通过以下方式设置事务隔离级别：

- 在连接时设置：`SET SESSION TRANSACTION ISOLATION LEVEL <隔离级别>`
- 在事务开始时设置：`SET TRANSACTION ISOLATION LEVEL <隔离级别>`
- 在创建表时设置默认隔离级别：`CREATE TABLE ... ENGINE=<存储引擎> DEFAULT TRANSACTION ISOLATION LEVEL <隔离级别>`

总结：MySQL的事务隔离级别包括读未提交、读已提交、可重复读和串行化。不同的隔离级别提供了不同的数据一致性和并发性保证，开发者需要根据具体需求选择合适的隔离级别。

12.索引的底层，为什么用B+树

在MySQL中，索引的底层数据结构主要有以下几种：

1. B+树（B+ Tree）：B+树是最常用的索引数据结构，它具有有序性、平衡性和磁盘访问优化等特点，适用于范围查询、排序和高效的插入、删除操作。
2. 哈希索引（Hash Index）：哈希索引使用哈希函数将键值映射到索引位置，适用于等值查询，具有快速的查找速度。但是，哈希索引不支持范围查询和排序操作，且对于键值的插入和删除操作相对较慢。
3. 全文索引（Full-Text Index）：全文索引用于对文本内容进行搜索，支持关键词的模糊匹配和全文检索。MySQL中的全文索引使用倒排索引（Inverted Index）来实现。
4. 位图索引（Bitmap Index）：位图索引使用位图来表示每个键值的存在与否，适用于低基数（Cardinality）的列，如性别、状态等。位图索引可以进行位运算来进行快速的多条件查询。
5. R树（R-Tree）：R树是一种用于空间数据的索引结构，适用于地理信息系统（GIS）等场景，可以高效地支持空间范围查询。

B+树是一种平衡的多路搜索树，它具有以下特点，使其成为数据库索引的理想选择：

1. 有序性：B+树的所有节点都按照键值的大小顺序排列，这使得B+树在范围查询和排序操作上具有良好的性能。
2. 平衡性：B+树通过自平衡的方式保持树的平衡，使得每个节点的高度相对较小，从而减少了查询的IO次数。
3. 磁盘访问优化：B+树的节点通常比内存页的大小要小，一个节点可以存储多个键值对，这样可以减少磁盘IO的次数。
4. 支持快速查找：B+树的查找操作只需要进行一次从根节点到叶子节点的遍历，而不需要回溯。
5. 支持范围查询：B+树的有序性使得范围查询变得简单，只需要在叶子节点上进行遍历即可。

6. 支持高效的插入和删除：B+树的自平衡特性使得插入和删除操作相对高效，只需要进行少量的节点分裂和合并操作。

由于数据库中的索引通常需要支持高效的查找、范围查询和排序操作，而B+树恰好具备这些特点，因此被广泛应用于数据库索引的实现中。

12.回表查询

回表查询是指在使用非聚集索引进行查询时，需要通过索引中的指针回到主键索引或者聚集索引中获取完整的数据行的过程。回表查询通常发生在以下场景中：

1. 需要查询的字段不在非聚集索引中：当查询的字段不在非聚集索引中时，数据库引擎无法直接从索引中获取完整的数据行，而是需要通过回表操作到主键索引或聚集索引中获取完整的数据行。
2. 需要返回的数据超过了非聚集索引的覆盖索引能力：覆盖索引是指索引中包含了查询所需的所有字段，可以直接从索引中获取查询结果，而无需回表操作。但是，如果需要返回的数据超过了非聚集索引的覆盖索引能力，仍然需要进行回表查询。
3. 使用了索引优化的查询：有些查询语句可能会使用到索引优化，例如使用了索引的覆盖扫描、索引合并等技术，这些优化可能会导致回表查询的发生。

回表查询会增加额外的IO操作，因为需要通过指针再次访问主键索引或聚集索引。在一些对查询性能要求较高的场景中，可以考虑使用覆盖索引或者调整查询语句的优化方式，以减少回表查询的次数。

回表查询并非一定是性能问题，有时候回表查询是必要的，特别是在需要返回完整数据行的情况下。在设计数据库表和索引时，需要根据具体的业务需求和查询场景来选择合适的索引策略，以达到最佳的查询性能。

13.redis为什么快，怎么持久化

Redis之所以快速，主要有以下几个原因：

1. 内存存储：Redis将数据存储在内存中，相比于传统的磁盘存储，内存的读写速度更快，可以达到微秒级的响应时间。
2. 单线程模型：Redis采用单线程模型，避免了多线程之间的竞争和锁的开销，简化了并发控制，提高了处理请求的效率。
3. 非阻塞IO：Redis使用了异步的非阻塞IO模型，通过IO多路复用技术（如epoll、kqueue等）实现高效的网络通信，提高了并发处理能力。
4. 简单的数据结构：Redis支持多种简单的数据结构，如字符串、哈希表、列表、集合和有序集合等，这些数据结构的实现都经过了优化，使得Redis在处理这些数据结构时更加高效。

至于Redis的持久化机制，它提供了两种持久化方式：

1. RDB（Redis Database）持久化：RDB是Redis默认的持久化方式，它通过将内存中的数据快照保存到磁盘上的二进制文件中。可以手动触发或定期自动触发RDB持久化。RDB持久化适用于数据备份和恢复，但可能会有一定的数据丢失。
2. AOF（Append-Only File）持久化：AOF持久化通过将写操作追加到文件末尾的方式来记录数据的变化，以保证数据的持久性。AOF持久化可以通过配置不同的策略来控制写入频率，包括每个写命令、每秒钟同步一次或者按照一定的时间间隔同步。AOF持久化适用于数据的持久性要求较高的场景，但相比RDB持久化，AOF持久化的文件更大，恢复速度较慢。

可以根据实际需求选择适合的持久化方式，或者同时使用RDB和AOF持久化来提高数据的安全性和可靠性。

出处飞机

7. 快手 Java 面经，根据项目一顿拷打 | 0302

1. 用到了哪些微服务组件
2. 负载均衡是怎么做的，如何实现一直均衡给一个用户
3. 服务熔断
4. 服务降级
5. Spring的IOC
6. 为什么依赖注入不适合使用字段注入
7. Spring的aop
8. Spring的事务，使用this调用是否生效
9. Spring的循环依赖
10. Spring MVC的工作流程；如果是传输文件呢
11. Kafka如何保证消息不丢失
12. Kafka如何保证消息不重复消费
13. MySQL的三大日志
14. ElasticSearch如何进行全文检索
15. 除了IK分词器还用到哪些分词器
16. 分词器的底层实现
17. 线性的数据结构（链表和数组），有什么区别
18. 如何使用两个栈实现队列
19. Dns基于什么协议实现，为什么是udp
    答：开销小、效率高、低延迟
20. http的特点
21. http无状态体现在哪
22. Cookie和session的区别

1. 负载均衡是怎么做的，如何实现一直均衡给一个用户

负载均衡实现的一般步骤：

1. **识别负载均衡的需求：** 首先需要确定在网络中的哪些资源需要进行负载均衡，如 Web 服务器、应用服务器或数据库服务器等。
2. **选择负载均衡算法：** 根据具体的需求选择合适的负载均衡算法，常见的算法包括轮询（Round Robin）、加权轮询（Weighted Round Robin）、最小连接数（Least Connection）等。
3. **配置负载均衡器：** 配置负载均衡器，设置负载均衡的规则和算法。这包括指定要均衡的服务器、配置健康检查（Health Check）机制以及设置负载均衡策略等。
4. **请求分发：** 当用户发送请求时，负载均衡器会根据选定的算法将请求分发到相应的服务器上。

5. **监控和调整：** 监控服务器的负载情况，根据需要动态调整负载均衡策略，以确保各服务器的负载保持均衡。

实现一直均衡给一个用户可能涉及不同的需求和场景，可以通过一些特定的策略来实现：

- **基于 IP 地址的持续连接（IP Stickiness）：** 通过将用户的请求路由到同一台服务器，以确保用户的会话状态在同一服务器上保持。这通常通过在负载均衡器上配置 IP 地址和服务器的映射关系来实现。
- **会话保持（Session Affinity）：** 类似于 IP Stickiness，但是更加灵活，可以基于用户的会话信息（如 Cookie）来实现，以确保用户的会话状态在同一服务器上保持。
- **动态负载均衡策略：** 除了静态配置的负载均衡策略外，还可以根据用户的负载情况和服务器的负载情况动态调整负载均衡策略，以实现持续的均衡。

2. 服务熔断

服务熔断（Circuit Breaker）是一种用于构建分布式系统的设计模式，用于增强系统的稳定性和可靠性。服务熔断的核心思想是在出现服务故障或异常时，及时地中断对该服务的请求，防止故障进一步扩散，并且允许系统在出现问题时快速失败而不是无限期地等待响应。

服务熔断的用途和优势包括：

1. **防止级联故障：** 当某个服务或组件出现故障时，服务熔断可以快速地停止对该服务的请求，防止故障扩散到其他部分，从而保护整个系统的稳定性。
2. **快速失败：** 服务熔断允许系统在出现问题时快速失败，而不是等待超时，这可以减少用户等待时间，并快速释放资源以减轻系统负担。
3. **降级处理：** 当服务熔断触发时，可以采取降级处理策略，例如返回预先定义的默认值、执行备用逻辑或者从缓存中获取数据，以保证系统的基本功能继续可用。
4. **自我修复：** 当服务熔断一段时间后，可以尝试重新发起请求，如果服务恢复正常，则关闭熔断器，继续正常提供服务。

常见问题和挑战包括：

1. **熔断器状态管理：** 需要有效地管理熔断器的状态，包括打开、关闭和半开状态的切换，以及对状态的监控和调整，确保熔断器的行为符合预期。
2. **故障诊断和处理：** 需要及时地检测和诊断服务的故障，并采取相应的措施进行处理，例如记录错误日志、发送警报通知等。
3. **降级策略设计：** 需要设计合适的降级处理策略，以保证在服务熔断时系统依然能够提供基本的功能，而不影响用户体验。
4. **性能影响：** 在服务熔断时，可能会增加系统的负载和响应时间，因此需要合理评估熔断器的触发条件和恢复机制，以减少性能影响。

总的来说，服务熔断是一种重要的分布式系统设计模式，通过及时地中断对故障服务的请求，可以提高系统的稳定性和可靠性，但同时也需要综合考虑各种因素，合理设计和管理熔断策略，以确保系统的正常运行。

3. 服务降级

服务降级（Service Degradation）是一种在面对系统资源不足或者系统负载过大时，为了保证系统的核心功能或者关键服务的可用性而采取的一种策略。服务降级的核心思想是在系统压力过大时，有选择性地降低非核心或次要功能的服务质量，以确保系统的核心功能仍然能够正常运行。

服务降级的目的在于保证系统在遇到异常或高负载情况下仍能够提供基本的服务能力，从而提高系统的稳定性和可用性。在实际应用中，服务降级通常伴随着一些特定的策略和实践，例如：

1. **优先级划分：** 将系统中的功能和服务按照其重要性和优先级进行划分，确保核心功能拥有更高的优先级，并在资源不足时优先保证核心功能的运行。

2. **限流和限速：** 通过限制对系统的访问速率或者并发请求数量，以防止系统过载，从而减轻系统负载压力。
3. **降级策略设计：** 设计合适的降级策略，当系统资源不足或者系统负载过大时，有选择性地降低非核心功能或次要功能的服务质量，例如降低服务响应时间、减少服务数据的返回量等。
4. **实时监控和调整：** 实时监控系统的负载情况和性能指标，根据实际情况动态调整降级策略，以确保系统的稳定性和性能表现。

服务降级的优点包括：

- **提高系统的稳定性：** 在面对异常情况或者高负载情况时，通过降级非核心功能或次要功能的服务质量，可以确保系统的核心功能仍能够正常运行，从而提高系统的稳定性。
- **保证核心功能的可用性：** 通过有选择性地降级非核心功能或次要功能，可以确保系统的核心功能在异常或高负载情况下仍能够提供基本的服务能力，从而保证核心功能的可用性。
- **减少系统压力：** 通过降级非核心功能或次要功能的服务质量，可以减少系统的负载压力，从而提高系统的整体性能和响应速度。

服务降级也存在一些潜在的缺点

- 可能影响用户体验
- 需要精细的策略设计和实时监控
- 可能引起业务方面的不满等。
  因此，在实施服务降级策略时，需要综合考虑各种因素，并设计合理的降级策略，以确保系统的整体性能和用户体验。

4. Spring的IOC

Spring框架中的IOC（Inverse of Control，控制反转）是一种设计原则，也是Spring框架的核心之一。IOC的概念是指控制权的转移，即将对象的创建、依赖注入和生命周期管理等控制权交给了容器来管理，而不是由对象自己来控制。这种控制反转的思想使得应用程序的组件之间的关系变得更加灵活、可维护和可测试。

在Spring框架中，IOC的实现主要通过依赖注入（Dependency Injection，DI）来实现。依赖注入是指容器负责在对象创建的同时，自动将对象所依赖的其他对象注入到它们之中，而不是由对象自己来创建或查找依赖的对象。

Spring框架通过XML配置文件、注解或Java代码来描述对象之间的依赖关系，然后由Spring容器在运行时根据这些配置来实现依赖注入。通过IOC容器管理对象之间的依赖关系，使得应用程序的组件解耦，提高了代码的灵活性和可维护性。

Spring的IOC容器通过控制对象之间的依赖关系，实现了对象之间的解耦，提高了代码的灵活性和可测试性，是Spring框架的核心特性之一。

5. 为什么依赖注入不适合使用字段注入

依赖注入有三种主要的方式：构造函数注入、Setter方法注入和字段注入。虽然字段注入是最简洁的一种方式，但它也存在一些问题，这就是为什么在某些情况下不推荐使用字段注入的原因。

1. **可测试性差：** 字段注入使得依赖关系被硬编码到了类的字段中，这导致在进行单元测试时很难对这些依赖进行模拟或替换。因为字段注入的依赖是直接赋值给类的字段，而不是通过构造函数或Setter方法来传递，所以在测试时很难通过传入不同的依赖实例来进行测试。
2. **难以追踪依赖关系：** 使用字段注入时，依赖关系是通过字段直接注入到类中的，这使得依赖关系不够明确，很难追踪类与其依赖之间的关系。这可能导致代码的可读性和可维护性下降。
3. **耦合性高：** 字段注入使得依赖关系与类的实现紧密耦合在一起，从而增加了类与其依赖之间的耦合性。当需要修改依赖关系时，可能需要修改类的字段定义，这违反了开闭原则。

相比之下，构造函数注入和Setter方法注入提供了更好的解耦性和可测试性。构造函数注入将依赖关系通过构造函数传入，而Setter方法注入则通过Setter方法设置依赖。这两种方式都能够清晰地标识出类与其依赖之间的关系，提高了代码的可读性和可维护性，同时也更易于进行单元测试和模拟依赖。因此，在使用依赖注入时，推荐优先考虑使用构造函数注入或Setter方法注入，而尽量避免使用字段注入。

6. Spring的aop

Spring框架中的AOP（Aspect-Oriented Programming，面向切面编程）是一种编程范式，它允许开发者通过将横切关注点（Cross-cutting Concerns）从核心业务逻辑中分离出来，从而提高了代码的模块化程度和可维护性。横切关注点包括日志记录、事务管理、安全检查等与核心业务逻辑无关的功能。

AOP的核心概念是切面（Aspect），切面是一个模块化单元，它封装了横切关注点，并定义了在何处、何时应用这些关注点。在Spring框架中，切面通常以Java类的形式存在，并且通过特定的注解或XML配置来定义切面。

Spring框架提供了几种实现AOP的方式：

1. **基于代理的AOP：** Spring使用动态代理来实现AOP。当一个类被AOP代理后，方法调用会被拦截，从而允许切面在方法调用前、后或者出现异常时执行额外的逻辑。Spring中的基于代理的AOP主要使用JDK动态代理和CGLIB动态代理来实现。
2. **切点和通知：** 切点（Pointcut）用于定义在哪些连接点（Join Point）应用切面逻辑，通知（Advice）用于定义切面的行为。Spring框架提供了几种通知类型，包括前置通知（Before Advice）、后置通知（After Advice）、返回通知（After Returning Advice）、异常通知（After Throwing Advice）和环绕通知（Around Advice）。
3. **切面表达式：** Spring提供了切面表达式语言（AspectJ Expression Language），通过切面表达式可以更精确地定义切点，并在切面中应用通知。切面表达式支持基于方法签名、参数、注解等方式来定义切点。

通过AOP，开发者可以将横切关注点模块化，避免代码重复和耦合度增加，从而提高代码的可维护性和可重用性。在Spring框架中，AOP常用于日志记录、事务管理、安全检查等方面，使得这些与核心业务逻辑无关的功能可以被统一管理和应用。

7. Spring的事务，使用this调用是否生效

在Spring中，使用`this`调用方式调用类的方法时，事务是生效的。Spring的事务是基于代理实现的，当一个类被Spring的事务代理包装后，所有对该类的方法的调用，包括通过`this`调用方式，都会被代理拦截。因此，即使使用`this`调用方式，Spring的事务管理仍然能够生效。

这是因为Spring的事务是通过AOP实现的，通常使用基于代理的AOP来处理事务。在基于代理的AOP中，Spring会为目标对象创建一个代理对象，而调用该代理对象的方法时，会触发AOP的通知，从而实现事务的管理。因此，无论是直接调用目标对象的方法，还是通过`this`调用方式调用目标对象的方法，Spring都能够正确地拦截并管理事务。

总之，Spring的事务是基于代理实现的，因此无论通过何种方式调用目标对象的方法，Spring的事务管理都会生效。

8. Spring的循环依赖

在 Spring 中，循环依赖是指两个或多个 Bean 之间互相依赖的情况，导致 Bean 的创建发生循环，从而无法完成依赖注入的过程。这种情况可能会导致应用程序启动失败或者出现不可预测的行为。

Spring 框架提供了一些解决循环依赖的机制：

1. **提前曝光（Early Exposure）：** Spring 容器在创建 Bean 的过程中，会提前暴露正在创建的 Bean 实例，以解决循环依赖的问题。当一个 Bean A 依赖另一个 Bean B，而 Bean B 又依赖 Bean A 时，Spring 在创建 Bean A 的过程中，会提前暴露一个代理对象，用于处理 Bean B 对 Bean A 的依赖。这样，Bean A 可以在被完全创建之前，通过代理对象来访问 Bean B。这种方式需要使用 CGLIB 来创建代理对象。
2. **构造函数注入：** 使用构造函数注入来解决循环依赖问题。Spring 容器在创建 Bean 的过程中，会首先将依赖项通过构造函数传递进去，从而避免了循环依赖的问题。这种方式需要谨慎使用，因为构造函数注入会将循环依赖暴露在类的构造函数中，可能导致代码不够清晰。
3. **Setter 方法注入：** 使用 Setter 方法注入来解决循环依赖问题。与构造函数注入类似，通过将依赖项通过 Setter 方法注入，可以避免循环依赖的问题。与构造函数注入相比，Setter 方法注入更加灵活，可以在 Bean 创建完成后再进行依赖注入，但也需要注意循环依赖可能带来的问题。

虽然 Spring 提供了这些解决循环依赖的机制，但是在设计应用程序时，尽量避免出现循环依赖是更好的选择。循环依赖会导致代码的复杂性增加，降低程序的可维护性和可读性。

9. Spring MVC的工作流程；如果是传输文件呢

Spring MVC（Model-View-Controller）是一个基于Java的Web框架，用于开发Web应用程序。它遵循经典的MVC设计模式，将应用程序划分为模型（Model）、视图（View）和控制器（Controller）三个部分，以实现松耦合和模块化开发。

Spring MVC的工作流程：

1. **客户端发起请求：** 客户端（通常是浏览器）向服务器发起请求，请求由Servlet容器（如Tomcat）接收。
2. **DispatcherServlet拦截请求：** 请求被DispatcherServlet拦截，DispatcherServlet是Spring MVC的核心控制器，负责统一管理请求的调度和分发。
3. **HandlerMapping确定处理器：** DispatcherServlet通过HandlerMapping将请求映射到对应的处理器（Controller）上。
4. **处理器执行请求：** 处理器根据请求的映射，执行相应的业务逻辑，并返回一个ModelAndView对象，其中包含了处理结果数据和视图名称。
5. **ViewResolver解析视图：** ModelAndView中的视图名称被ViewResolver解析为具体的视图对象，视图对象可以是JSP页面、Thymeleaf模板等。
6. **视图渲染：** 视图对象将处理结果数据渲染到具体的视图中。
7. **响应返回客户端：** 渲染后的视图最终以响应的形式返回给客户端。

传输文件时的工作流程：

如果在Spring MVC中需要传输文件，通常使用multipart/form-data类型的表单来上传文件。工作流程与上述流程类似，但有一些不同之处：

1. **客户端发送包含文件的请求：** 客户端通过表单提交包含文件的请求。
2. **DispatcherServlet拦截请求：** 请求被DispatcherServlet拦截。
3. **MultipartResolver解析请求：** DispatcherServlet使用MultipartResolver解析包含文件的请求，将请求中的文件内容解析为MultipartFile对象。
4. **HandlerMapping确定处理器：** 请求被映射到相应的处理器（Controller）上。
5. **处理器处理文件上传：** 处理器接收到请求后，处理文件上传操作，通常会将文件保存到服务器本地或进行其他操作。
6. **处理器返回响应：** 处理器返回处理结果。
7. **视图渲染或直接返回响应：** 如果需要，视图渲染将处理结果渲染到视图中；如果不需要，直接返回响应给客户端。

10. Kafka如何保证消息不丢失

Kafka 是一个分布式流处理平台，它使用一些机制来确保消息不丢失：

1. **持久化到磁盘：** Kafka 将消息持久化到磁盘上的文件中，这样即使在消息被处理之前，它们也可以被持久化和存储。因此，即使发生系统故障或者重启，消息也不会丢失。
2. **复制机制：** Kafka 通过副本机制来提供高可用性和容错性。在 Kafka 集群中，每个分区的数据都会被复制到多个 Broker 上。这些副本中的一个被选为 Leader，负责处理所有的读写请求，其他副本作为 Followers，用于备份数据。如果 Leader 失效，Kafka 将自动从 Followers 中选举出新的 Leader，从而保证数据的可用性和一致性。
3. **消息确认机制：** Kafka 提供了消息确认机制，可以确保消息被成功写入到 Broker 中。生产者可以选择等待 Broker 的确认信息，以确保消息被成功复制到指定数量的副本中后再发送下一条消息。这样可以避免消息丢失，但可能会降低消息的发送速度。
4. **持久化日志：** Kafka 使用持久化日志来存储消息，这使得即使在发生异常情况下，消息也不会丢失。Kafka 的存储机制允许在消息被消费之前将其保留在日志中。

Kafka 使用持久化、复制机制、消息确认和持久化日志等多种机制来确保消息不丢失。这些机制使得 Kafka 成为一个可靠的消息系统，适用于处理关键业务数据和高吞吐量的场景。

11. Kafka如何保证消息不重复消费

Kafka 使用消费者组（Consumer Group）来确保消息不重复消费的机制，具体方式如下：

1. **分区和偏移量：** Kafka 将每个主题（Topic）划分为多个分区（Partition），每个分区包含一个有序的消息序列。消费者在消费消息时，会按照分区的顺序消费消息。而每个分区中的消息都有一个唯一的偏移量（Offset），用来标识消息在分区中的位置。
2. **消费者组：** 消费者可以组成消费者组，一个消费者组中可以包含一个或多个消费者。每个消费者组对于一个主题的每个分区都可以有一个或多个消费者，但是一个分区只能被同一个消费者组中的一个消费者消费。消费者组中的消费者之间协调消费任务，每个消费者负责消费分区中的一部分消息。
3. **偏移量的管理：** 每个消费者都会记录自己消费的分区和偏移量，以便在重启或者重分配分区时能够继续消费之前的消息。消费者将消费的最新偏移量保存在 Kafka 中的内部主题（__consumer_offsets）中。
4. **消费者组协调器：** Kafka 集群中的一个特殊的 Broker 会负责充当消费者组的协调器（Consumer Group Coordinator），它负责分配分区给消费者组中的消费者，并确保每个分区只被一个消费者消费。

通过以上机制，Kafka 确保了消息不重复消费的原则：

- 消费者组内的每个消费者负责消费不同的分区，确保了分区内消息的顺序性。
- 消费者组中的消费者协调分配分区，避免了多个消费者同时消费同一个分区的情况。
- 消费者记录消费的偏移量，以便在重启或者重分配分区时能够继续消费之前的消息。

Kafka 通过消费者组、分区和偏移量的管理机制来保证消息不重复消费，从而确保了消息处理的准确性和一致性。

12. MySQL的三大日志

1. **二进制日志（Binary Log）：**
   二进制日志记录了数据库中所有的修改操作，以二进制格式保存。这些操作包括数据的插入、更新、删除以及数据结构的更改（如表结构的修改）。二进制日志对于数据库的备份、复制和恢复非常重要。
2. **重做日志（Redo Log）：**
   重做日志是InnoDB存储引擎特有的日志类型。它记录了数据库中每个事务所做的修改操作，但与二进制日志不同，重做日志以物理格式记录。重做日志用于数据库的恢复，以保证事务的持久性和一致性。
3. **撤销日志（Undo Log）：**
   撤销日志也是InnoDB存储引擎特有的日志类型。它记录了事务执行过程中所做的修改操作的逆操

作，用于回滚事务、处理并发事务和提供数据库的一致性视图。

13. ElasticSearch如何进行全文检索

Elasticsearch 是一个分布式、RESTful 风格的搜索和分析引擎，它提供了强大的全文检索功能。下面是 Elasticsearch 如何进行全文检索的简要流程：

1. **创建索引（Index）：** 在 Elasticsearch 中，数据被组织在索引中。索引类似于关系型数据库中的数据库，用于存储一类相关的数据。在进行全文检索之前，首先需要为数据创建一个索引。
2. **添加文档（Document）：** 在索引中添加文档，文档是 Elasticsearch 中存储的基本单位。文档通常是 JSON 格式的数据，可以包含一个或多个字段，每个字段包含一个或多个值。
3. **建立映射（Mapping）：** 映射定义了文档中每个字段的数据类型和分析方式。在创建索引时，可以自定义字段的映射，或者让 Elasticsearch 根据文档的结构自动推断映射。
4. **执行搜索请求：** 通过发送搜索请求来进行全文检索。搜索请求可以包含一个或多个查询条件，Elasticsearch 将根据这些条件来匹配文档，并返回符合条件的结果。
5. **分析查询条件：** Elasticsearch 会对查询条件进行分析，将查询条件解析为一个或多个查询子句。查询子句包括词项查询、短语查询、范围查询等，用于匹配文档中的字段值。
6. **执行查询：** Elasticsearch 执行查询操作，并使用倒排索引来快速定位匹配的文档。倒排索引是一种将文档中的每个词项映射到包含该词项的文档列表的数据结构，可以高效地进行全文检索。
7. **评分和排序：** Elasticsearch 根据匹配度对搜索结果进行评分，并按照评分对结果进行排序。评分是根据文档与查询条件的匹配程度来计算的，匹配度越高的文档排名越靠前。
8. **返回搜索结果：** Elasticsearch 返回符合查询条件的文档结果，通常包括文档的 ID、分数和其他字段值等信息。

Elasticsearch 的全文检索流程包括创建索引、添加文档、建立映射、执行搜索请求、分析查询条件、执行查询、评分和排序以及返回搜索结果等步骤。通过这些步骤，Elasticsearch 可以高效地进行全文检索，并返回符合查询条件的文档结果。

14. 除了IK分词器还用到哪些分词器

除了 IK 分词器外，Elasticsearch 还有其他常用的分词器，例如：

1. **Standard 分词器：** 标准分词器是 Elasticsearch 默认的分词器，它是一个基于 Unicode 标准的分词器，按照单词边界进行分词，支持多种语言。
2. **Whitespace 分词器：** 空格分词器根据空格字符将文本分割成单词，适合处理英文文本或者不需要分词的场景。
3. **Simple 分词器：** 简单分词器是一个非常简单的分词器，根据非字母字符将文本分割成单词，适用于处理非结构化文本数据。
4. **Lowercase 分词器：** 小写分词器将文本转换为小写，然后使用标准分词器进行分词，适用于处理不区分大小写的场景。
5. **Keyword 分词器：** 关键字分词器将整个文本作为一个单词，适用于不需要分词的场景，通常用于对字段进行精确匹配或者聚合操作。
6. **Pattern 分词器：** 模式分词器根据指定的正则表达式将文本分割成单词，适用于需要根据特定规则进行分词的场景。
7. **Stop 分词器：** 停止词分词器将指定的停止词（如常见的连接词、介词等）从文本中移除，适用于需要去除停止词的场景。
8. **Snowball 分词器：** 雪球分词器支持多种语言的词干提取，可以将单词转换为其词干形式，适用于需要进行词干提取的场景。

15. 分词器的底层实现

分词器（Tokenizer）在 Elasticsearch 中的底层实现通常是基于 Lucene 的分词器实现的。Lucene 是一个开源的全文搜索引擎库，Elasticsearch 是基于 Lucene 构建的分布式搜索和分析引擎，因此它继承了 Lucene 强大的分词器功能。

在 Lucene 中，分词器负责将文本按照一定规则进行分词，生成一个个词项（Term）。词项是搜索的基本单位，是搜索引擎索引中的最小单元。分词器根据具体的语言、词汇、文法等规则将文本分割成不同的词项，然后进行索引和搜索。

分词器的底层实现通常包括以下几个方面：

1. **字符解析：** 分词器首先将输入的文本字符流进行解析，识别出文本中的单词、标点符号、数字等字符。
2. **词项生成：** 根据一定的规则，分词器将解析后的字符流进行分割，生成一个个词项。词项的生成方式可以根据具体的分词器实现而不同，例如基于词典、规则、统计信息等。
3. **词项过滤：** 在生成词项的过程中，分词器可能会对词项进行过滤，去除一些不需要的词项，例如停止词（Stop Words）等。
4. **词干提取：** 有些分词器会对词项进行词干提取，将词项转换为其原始形式，以提高搜索的准确性和召回率。
5. **标记位置和偏移量：** 分词器还会为每个词项记录其在文本中的位置和偏移量，以便后续索引和搜索操作。
6. **分词器链：** 在 Elasticsearch 中，可以通过配置多个分词器来构建一个分词器链（Tokenizer Chain），每个分词器依次对输入的文本进行处理，生成词项序列。

分词器的底层实现是基于 Lucene 的分词器实现的，它通过对文本进行解析、分割、过滤、词干提取等操作，生成词项序列，为搜索引擎的索引和搜索提供了基础支持。在 Elasticsearch 中，分词器的实现可以根据具体的需求和场景进行自定义和配置。

16. 线性的数据结构（链表和数组），有什么区别

链表和数组是两种常见的线性数据结构，它们之间有一些区别，主要包括以下几个方面：

1. **内存分配方式：**
   - 数组在内存中是一段连续的内存空间，每个元素都占据固定大小的内存空间，可以通过索引直接访问任意位置的元素。
   - 链表中的元素在内存中不是连续存储的，每个元素包含一个指向下一个元素的指针，通过指针将各个元素连接起来，因此在链表中访问元素需要从头节点开始遍历。
2. **插入和删除操作：**
   - 数组的插入和删除操作比较低效，因为需要移动大量元素来维持连续的内存空间。在最坏情况下，时间复杂度为 O(n)。
   - 链表的插入和删除操作相对高效，只需要修改相邻元素的指针，时间复杂度为 O(1)。
3. **查找操作：**
   - 数组的查找操作比较高效，可以通过索引直接访问元素，时间复杂度为 O(1)。
   - 链表的查找操作比较低效，需要从头节点开始遍历链表，直到找到目标元素或者到达链表末尾，时间复杂度为 O(n)。
4. **空间复杂度：**
   - 数组需要一块连续的内存空间来存储所有元素，因此需要预先分配足够的空间，如果空间不足需要进行扩容操作。
   - 链表不需要预先分配连续的内存空间，可以动态分配内存，因此可以灵活地处理不同大小的数据集。

数组和链表各有优缺点，选择使用哪种数据结构取决于具体的应用场景和需求。如果需要频繁进行插入和删除操作，且数据量不确定，可以选择链表；如果需要频繁进行查找操作，且数据量相对稳定，可以选择数组。

17. 如何使用两个栈实现队列

使用两个栈实现队列的基本思路是，利用一个栈作为入队栈，用于存储入队的元素；另一个栈作为出队栈，用于存储出队的元素。具体实现如下：

1. **入队操作：**
    - 将元素压入入队栈中即可。
2. **出队操作：**
    - 如果出队栈为空，将入队栈中的所有元素依次弹出并压入出队栈中，然后从出队栈中弹出栈顶元素；
    - 如果出队栈不为空，直接从出队栈中弹出栈顶元素。

使用两个栈实现队列的关键在于在出队操作时将入队栈的元素倒序压入出队栈中，从而实现队列的先进先出（FIFO）的特性。

下面是使用 Python 语言实现的示例代码：

```
class MyQueue:
    def __init__(self):
        self.in_stack = []  # 入队栈
        self.out_stack = []  # 出队栈

    def push(self, x: int) -> None:
        self.in_stack.append(x)

    def pop(self) -> int:
        if not self.out_stack:
            while self.in_stack:
                self.out_stack.append(self.in_stack.pop())
        return self.out_stack.pop()

    def peek(self) -> int:
        if not self.out_stack:
            while self.in_stack:
                self.out_stack.append(self.in_stack.pop())
        return self.out_stack[-1]

    def empty(self) -> bool:
        return not self.in_stack and not self.out_stack
```

这样实现的队列的时间复杂度为：

- 入队操作的时间复杂度为 O(1)；
- 出队操作的平均时间复杂度为 O(1)；
- 空间复杂度为 O(n)。

18. Dns基于什么协议实现，为什么是udp

DNS（Domain Name System，域名系统）是基于 UDP（User Datagram Protocol，用户数据报协议）实现的。

UDP 是一种无连接的、不可靠的传输层协议，它不提供数据包的可靠性、流量控制和拥塞控制等功能，但它具有简单、轻量、低延迟的特点。而 DNS 的设计目标是快速响应用户查询，并且要求尽可能地减小网络通信的延迟，因此选择了 UDP 作为 DNS 协议的传输层协议。

DNS 之所以选择 UDP 而不是 TCP，主要是出于以下考虑：

1. **低延迟：** UDP 相对于 TCP 更加轻量，不需要建立连接、维护连接状态和进行错误重传等操作，因此可以减小网络通信的延迟，适合对延迟要求较高的场景。
2. **简单快速：** UDP 头部较小，没有 TCP 那样的连接管理和流量控制机制，因此可以更快速地发送和接收数据，减小数据包的大小，降低网络负载。
3. **高并发：** UDP 不需要维护连接状态，每个 UDP 数据包都是独立的，因此可以支持更高的并发连接数，适合于 DNS 服务器处理大量并发的查询请求。

尽管 UDP 存在丢包和数据包损坏的风险，但在 DNS 的设计中，这些风险是可以被容忍的。DNS 使用了一些机制来提高可靠性，例如查询超时重传、请求重试、缓存等，以确保数据传输的可靠性和正确性。

总的来说，DNS 选择 UDP 作为传输层协议，是为了追求更低的延迟、更高的性能和更好的并发能力，同时通过其他机制来提高数据传输的可靠性。

19. http的特点

HTTP（Hypertext Transfer Protocol，超文本传输协议）是一种用于传输超文本数据（如 HTML）的应用层协议，它是现代 Web 通信的基础。HTTP 的特点包括以下几个方面：

1. **无连接性（Connectionless）：** HTTP 是一种无连接的协议，每次请求-响应周期都是相互独立的，服务器在收到客户端请求并发送响应后立即断开连接。这种特点使得 HTTP 的连接开销较小，适用于短时交互的通信场景。
2. **无状态性（Stateless）：** HTTP 是一种无状态的协议，即服务器不会维护客户端的状态信息。每个请求都是独立的，服务器不能识别两个请求是否来自同一个客户端。这种特点简化了服务器的设计和实现，但也限制了 HTTP 在一些需要保持状态的场景下的应用。
3. **基于文本（Text-based）：** HTTP 使用文本格式的请求和响应消息进行通信，消息头部包含了一系列的键值对，用于描述消息的属性和特征，消息体则包含了具体的数据内容。这种基于文本的格式使得 HTTP 的消息易于阅读和理解，并且可以使用简单的文本编辑器进行调试和修改。
4. **支持多媒体类型（Media-independent）：** HTTP 不仅可以传输 HTML 文档，还可以传输各种不同类型的多媒体数据，如图片、音频、视频等。HTTP 使用 MIME（Multipurpose Internet Mail Extensions）标准来描述和传输多媒体数据。
5. **灵活性（Flexibility）：** HTTP 是一种灵活的协议，支持多种不同的请求方法（如 GET、POST、PUT、DELETE 等）、状态码（如 200 OK、404 Not Found、500 Internal Server Error 等）和头部字段（如 Content-Type、Content-Length、Cache-Control 等），可以根据具体的需求和场景进行定制和扩展。

HTTP 是一种简单、灵活、文本化的应用层协议，具有无连接性、无状态性、基于文本、支持多媒体类型和灵活性等特点，适用于传输超文本数据和多媒体数据的通信场景。

20. http无状态体现在哪

HTTP 的无状态性体现在服务器不会在请求之间保存客户端的状态信息。具体来说，无状态性表现在以下几个方面：

1. **每个请求独立性：** HTTP 是一种无连接的协议，每个请求-响应周期都是相互独立的，服务器在收到客户端请求并发送响应后立即断开连接。因此，服务器不能识别两个请求是否来自同一个客户端，也无法知道这两个请求之间是否存在关联性。
2. **不保存客户端状态：** 服务器不会保存客户端的状态信息，即服务器在处理每个请求时都是无状态的。客户端发送的每个请求都必须包含足够的信息来描述其自身的状态，服务器在处理请求时不能依赖之前的请求状态。
3. **状态信息在客户端：** 为了维护客户端的状态信息，客户端通常会在请求中包含一些标识符或者会话令牌（如 Cookie 或者 Session ID），服务器在接收到请求后会解析这些信息并根据需要进行处理。这种方式使得客户端的状态信息保存在客户端而不是服务器端，从而实现了无状态性。

4. **灵活性：** HTTP 的无状态性提供了灵活性，允许服务器在处理请求时不受之前请求的影响，可以根据当前请求的情况来进行处理。这种灵活性使得服务器可以更好地应对不同客户端的请求，并且使得 HTTP 协议更易于扩展和部署。

HTTP 的无状态性意味着服务器在处理每个请求时都是独立的、无状态的，不会保存客户端的状态信息，从而提高了服务器的灵活性和可伸缩性。

21. Cookie和session的区别

Cookie 和 Session 都是用来在客户端和服务器之间保存状态信息的机制，但它们有一些区别：

1. **存储位置：**
   - Cookie 是存储在客户端的一小段文本信息，以键值对的形式保存在客户端的浏览器中。
   - Session 是存储在服务器端的一种会话状态，通常存储在服务器内存、数据库或者其他持久化存储介质中。
2. **安全性：**
   - Cookie 存储在客户端的浏览器中，因此相对不安全，可能被篡改或者盗用。
   - Session 存储在服务器端，相对安全，客户端无法直接访问或修改。
3. **容量限制：**
   - Cookie 的大小通常有限制，一般不超过 4KB。
   - Session 的大小没有明确限制，受服务器资源和配置的影响，可以存储更多的信息。
4. **生命周期：**
   - Cookie 可以设置过期时间，在设置的过期时间之前一直保存在客户端，可以长期保存。
   - Session 的生命周期由服务器控制，通常在客户端关闭后自动失效，也可以手动设置过期时间。
5. **存储内容：**
   - Cookie 可以存储任意类型的数据，但是由于存储在客户端，不适合存储敏感信息。
   - Session 可以存储任意类型的数据，因为存储在服务器端，可以存储敏感信息。
6. **传输方式：**
   - Cookie 在 HTTP 头部中通过 Set-Cookie 和 Cookie 字段进行传输。
   - Session 通常通过在 Cookie 中保存 Session ID 来实现，客户端每次请求时将 Session ID 发送给服务器，服务器根据 Session ID 查找对应的会话状态。

Cookie 和 Session 是两种不同的状态管理机制，它们在存储位置、安全性、容量限制、生命周期、存储内容和传输方式等方面存在一些区别，开发人员需要根据具体的需求和场景选择合适的机制来管理状态信息。

8. 百度Java实习面试都这么难了吗？|0304

1. 联合索引底层存储结构(和其他种类的索引的存储结构有什么区别?)
2. 联合索引的叶子节点存的什么内容?
3. 事务会不会自动提交?
4. MySQL默认的隔离级别是什么?
5. Gc算法有哪些?
6. G1 垃圾回收器了解吗?
7. 什么时候会触发 GC?
8. 线程安全和线程不安全是什么意思?
9. 场景:有一个 key 对应的 value 是一个json,结构，json,当中有好几个子任务，这些子任务如果对 key 进行修改的话,会不会存在线程安全的问题?如何解决?如果是多个节点的情况，应该怎么加锁?
10. Setnx,知道吗? 用这个加锁有什么问题吗?怎么解决?
11. 你项目的难点有哪些?
12. 层次遍历一个 DAG 图，有向无环图。
13. leetcode 岛屿数量
14. 有一个主任务，它有四个子任务，分别存在下面几种状态:wait,running，success，fail。用if-else 的形式写出所有可能发生的情况。

1、联合索引底层存储结构(和其他种类的索引的存储结构有什么区别?)

联合索引是数据库中一种常见的索引类型，它允许在多个列上创建索引，以提高查询性能。与单列索引相比，联合索引的底层存储结构有一些区别，主要体现在如何组织和存储索引数据的方式上。

1. **存储数据的组织方式**：
   - 单列索引：单列索引只包含一个列的值和指向相应数据行的指针。通常，单列索引按照列值的大小顺序来组织存储数据。
   - 联合索引：联合索引则包含多个列的值，以及指向相应数据行的指针。联合索引可以按照多个列值的组合来组织存储数据，这意味着可以根据多个列的值来定位数据行。
2. **查询时的性能影响**：
   - 单列索引：单列索引适合用于只涉及单个列的查询。当查询条件涉及到索引列时，数据库可以更快地定位到匹配的数据行。
   - 联合索引：联合索引适合用于涉及多个列的查询。当查询涉及到联合索引中的多个列时，数据库可以利用索引中列值的组合来快速定位匹配的数据行。
3. **索引维护的复杂性**：
   - 单列索引：单列索引的维护相对简单，因为它只需要维护单个列的值和指针。
   - 联合索引：联合索引的维护相对复杂一些，因为它需要考虑多个列值的组合。当表中的数据发生变化时，数据库需要确保联合索引中的多个列值的组合保持有序，这可能需要更多的资源和时间。

联合索引在适当的情况下可以提供更好的查询性能，特别是对于涉及到联合索引中列值的组合的查询。然而，需要注意的是，联合索引的创建和维护可能会带来一些额外的开销，并且需要根据具体的查询需求和数据模式来合理选择索引策略。

2、联合索引的叶子节点存的什么内容?

联合索引的叶子节点存储的是索引列的值以及指向对应数据行的指针（聚集索引键值）。在联合索引中，叶子节点包含了多个列的值，以及指向对应数据行的指针（者聚集索引键值）的组合。

具体来说，叶子节点中存储了索引列的实际值，以及一个指向相应数据行的指针（者聚集索引键值）。这样，当数据库引擎根据联合索引执行查询时，它可以通过索引中的列值找到对应的叶子节点，然后使用叶子节点中的指针（键值）来定位到相应的数据行。

联合索引的叶子节点存储了索引列的值和指向数据行的指针（键值）的组合，这样可以在查询时快速定位到匹配的数据行。

3、事务会不会自动提交?

MySQL 默认开启事务自动提交模式，即除非显式的开启事务（BEGIN 或 START TRANSACTION），否则每条 SOL 语句都会被当做一个单独的事务自动执行。

要启用自动提交，在执行任何DML（Data Manipulation Language）语句（例如INSERT、UPDATE、DELETE）之前，可以使用以下语句开启自动提交：

```
SET autocommit = 1;
```

或者可以在连接到数据库时在连接字符串中指定`autocommit`参数为1。

```
mysql -u username -p -h hostname dbname --autocommit=1
```

这将使得MySQL会话处于自动提交模式，这意味着每个单独的DML语句都将自动成为一个事务，并在执行完成后立即提交。

如果想要禁用自动提交，可以使用以下语句：

```
SET autocommit = 0;
```

或者在连接字符串中指定`autocommit`参数为0。

4、MySQL默认的隔离级别是什么?

MySQL的默认隔离级别是可重复读（Repeatable Read）。在可重复读隔离级别下，事务可以读取其他事务已经提交的数据，但是不会看到其他事务未提交的数据。这意味着在同一个事务中多次读取相同的数据，将会得到相同的结果，即使其他事务对该数据进行了修改也是如此。

可重复读隔离级别保证了在同一个事务中多次读取数据时的一致性，但也可能导致一些并发问题，如幻读（Phantom Read）。MySQL提供了其他隔离级别来解决不同的并发问题，包括读未提交（Read Uncommitted）、读已提交（Read Committed）和串行化（Serializable）等。

需要注意的是，虽然MySQL的默认隔离级别是可重复读，但实际上可以在会话级别或全局级别进行更改，以满足特定的应用需求。

InnoDB 当前读下的幻读是通过间隙锁（gap_lock)来实现的。在事务A查询的时候，会锁住一个间隙，其它事务往这个间隙插入、删除等操作都是会被锁阻塞的。间隙锁和插入意向锁互斥，彻底解决了当前读下的幻读问题。

但是InnoDB 没有完全解决快照读下的幻读问题。

5、Gc算法有哪些?

Java的垃圾收集（Garbage Collection，GC）算法有多种，每种算法都有其独特的特点和适用场景。以下是几种常见的Java GC算法：

1. **标记-清除算法（Mark and Sweep）**：

    - 标记阶段：从根对象出发，递归地标记所有可以被访问到的对象。
    - 清除阶段：清除未被标记的对象，即不可达对象。
    - 缺点：产生内存碎片，可能导致内存分配效率降低。
2. **复制算法（Copying）**：
    - 将堆内存分为两个区域：年轻代和老年代。
    - 年轻代分为Eden区和两个Survivor区。
    - 对象首先被分配到Eden区，当Eden区满时，触发Minor GC，将存活的对象复制到Survivor区，然后清空Eden区和一个Survivor区，再将存活的对象从另一个Survivor区复制到空的Survivor区。
    - 优点：不会产生内存碎片，适用于频繁回收对象的场景。
3. **标记-整理算法（Mark and Compact）**：
    - 标记阶段：与标记-清除算法相同，标记所有可达对象。
    - 整理阶段：将所有存活的对象向一端移动，然后清理掉不可达对象，从而消除内存碎片。
    - 优点：不会产生内存碎片，可以提高内存分配效率。
4. **分代收集算法（Generational Collection）**：
    - 将堆内存分为年轻代和老年代两部分，使用不同的GC算法。
    - 年轻代通常使用复制算法，因为大多数对象在这里很快变得不可达，适合频繁进行垃圾收集。
    - 老年代通常使用标记-清除或标记-整理算法，因为老年代存活的对象较多，适合采用更加成熟的算法来进行垃圾收集。
5. **G1算法（Garbage-First）**：
    - 将堆内存分成多个大小相等的区域（Region），包括年轻代、老年代和Metaspace等。
    - 将整个堆内存划分为多个Region，通过标记-复制和标记-整理的方式来执行垃圾收集。
    - G1算法通过优先收集垃圾最多的Region来提高垃圾收集效率，因此称为“Garbage-First”。

6、G1 垃圾回收器了解吗?

G1（Garbage-First）垃圾回收器是Java虚拟机（JVM）中一种现代的垃圾回收器，引入自Java 7 Update 4版本。G1垃圾回收器旨在替代CMS（Concurrent Mark-Sweep）垃圾回收器，并且在大内存堆上表现更加稳定和高效。

G1垃圾回收器具有以下特点：

1. **区域化内存管理**：G1将堆内存划分为多个固定大小的区域（Region），每个区域可以是Eden区、Survivor区或Old区。这种区域化的内存管理有助于更好地控制垃圾回收过程，减少停顿时间。
2. **分代收集**：虽然G1并不是一个传统的分代收集器，但它仍然将堆内存划分为年轻代和老年代，并且使用不同的垃圾回收策略来处理这两个代。
3. **并发标记清除**：G1使用了并发标记（Concurrent Marking）来减少垃圾回收暂停时间。在标记阶段，G1通过并发标记线程来标记活动对象，而在应用程序运行的同时，也会继续标记操作。这样可以减少标记阶段对应用程序的影响。
4. **整理内存**：G1使用了复制算法来清理内存，不再使用传统的压缩算法。在垃圾收集过程中，G1会选择一些区域进行垃圾收集，并将存活对象复制到其他区域中，从而实现内存的整理和碎片整理。
5. **垃圾优先收集**：G1根据垃圾回收需求来选择优先回收的区域，以此来提高垃圾回收效率。它会优先选择包含垃圾最多的区域进行回收，从而最大程度地减少垃圾对象。

G1垃圾回收器在大内存堆上表现更加稳定和高效，尤其适用于需要低停顿时间和更加可控的垃圾回收的应用场景。

7、什么时候会触发 GC?

在Java虚拟机中，垃圾回收（GC）会在以下几种情况下触发：

1. **系统内存不足**：当Java虚拟机检测到系统内存不足时，会触发垃圾回收来释放内存空间，以确保应用程序的正常运行。这通常是通过监视堆内存的使用情况来检测的。
2. **调用System.gc()方法**：虽然调用System.gc()方法并不会立即触发垃圾回收，但它会向Java虚拟机发出建议性的垃圾回收请求。Java虚拟机可以选择是否立即响应这个请求。
3. **长时间停顿**：当应用程序执行时间较长，而且没有进行垃圾回收时，Java虚拟机可能会为了避免堆内存耗尽而触发垃圾回收。这种情况下，垃圾回收通常会引起一段较长的停顿时间，称为Full GC。
4. **Young Generation满**：在分代垃圾回收器中，当Young Generation区域满时，会触发一次Minor GC。这会导致Eden区和Survivor区的垃圾回收。
5. **Old Generation满**：如果Old Generation区域满了，会触发一次Major GC（也称为Full GC）。这种情况下，整个堆内存都会进行垃圾回收。
6. **永久代/元空间满**：对于HotSpot虚拟机，如果永久代（Java 7之前）或者元空间（Java 8及之后）满了，会触发一次垃圾回收。这种情况下，垃圾回收主要针对类的元数据和常量池。

需要注意的是，具体触发垃圾回收的时机和方式取决于Java虚拟机的实现，不同的虚拟机可能有不同的行为。此外，开发人员可以通过调整垃圾回收相关的参数来影响垃圾回收的行为，以优化应用程序的性能和资源利用率。

8、线程安全和线程不安全是什么意思?

"线程安全"和"线程不安全"是描述在多线程环境中并发操作的状态的术语。

1. **线程安全**：
   - 线程安全指的是在多线程环境下，对共享数据的访问操作能够保证在并发情况下不会导致数据的不一致性或损坏。一个线程安全的操作或数据结构能够在并发访问时维持其内部状态的一致性。
   - 线程安全的实现通常会采用同步机制（例如锁、信号量等）来保护共享资源的访问，以确保在任意时刻只有一个线程能够访问共享资源，从而避免竞态条件（Race Condition）和其他并发问题。
2. **线程不安全**：
   - 线程不安全指的是在多线程环境下，对共享数据的访问操作可能会导致数据的不一致性或损坏。线程不安全的操作或数据结构在并发访问时无法保证其内部状态的一致性，可能会导致意外的结果或程序错误。
   - 线程不安全的实现通常没有考虑到并发访问的情况，没有采取适当的同步措施来保护共享资源的访问，因此可能会出现竞态条件和其他并发问题。

举例来说，如果多个线程同时尝试向同一个数组中添加元素，而该数组的添加操作没有进行适当的同步控制，那么就可能导致线程不安全的情况，如数据覆盖、越界访问等。为了保证线程安全，需要在并发访问共享资源时使用适当的同步机制来确保数据的一致性。

9、场景:有一个 key 对应的 value 是一个json,结构，json,当中有好几个子任务，这些子任务如果对 key 进行修改的话,会不会存在线程安全的问题?如何解决?如果是多个节点的情况，应该怎么加锁?

在这个场景中，如果多个线程同时对同一个 key 对应的 JSON 结构中的子任务进行修改，就有可能出现线程安全的问题，因为多个线程同时对同一个数据结构进行修改可能导致数据不一致或损坏。

要解决这个问题，可以采用以下方法：

1. **使用线程安全的数据结构**：可以选择使用线程安全的数据结构来存储 JSON 数据，例如 `ConcurrentHashMap` 或 `CopyOnWriteArrayList`，它们内部提供了并发访问的安全保证，能够

在多线程环境下安全地进行操作。
2. **使用同步机制**：可以使用同步机制（例如锁）来保护对 JSON 数据的修改操作，确保在任意时刻只有一个线程能够修改数据。比如，在对 JSON 数据进行修改之前，先获取一个锁，并在操作完成后释放锁。
3. **粒度控制**：可以考虑将 JSON 数据的不同部分分别进行锁定，以减小锁的粒度，提高并发性能。比如，可以为不同的子任务或不同的节点设置不同的锁。

如果是多个节点的情况，需要在分布式环境下进行考虑。可以选择分布式锁机制来实现跨节点的数据同步和并发控制，比如使用 ZooKeeper、Redis 等分布式系统提供的分布式锁服务来确保在不同节点上对数据的并发修改操作是安全的。

10、Setnx,知道吗? 用这个加锁有什么问题吗?怎么解决?

`SETNX` 是 Redis 中的一个命令，用于设置键的值，但仅当键不存在时才设置成功。在分布式环境中，可以利用 `SETNX` 命令来实现分布式锁。具体步骤如下：

1. 客户端通过 `SETNX` 命令尝试将一个特定的键作为锁的标识，并设置一个唯一的值作为锁的持有者标识。
2. 如果 `SETNX` 命令成功执行（返回值为 1），表示当前客户端成功获取了锁，可以执行后续操作。
3. 如果 `SETNX` 命令执行失败（返回值为 0），表示当前锁已被其他客户端持有，当前客户端未获取到锁，需要等待一段时间后重新尝试获取锁。

虽然 `SETNX` 命令在某些情况下可以用来实现简单的分布式锁，但是它也存在一些问题：

1. **无法设置过期时间**：`SETNX` 命令本身不支持设置键的过期时间，因此当持有锁的客户端发生异常或程序出现问题时，可能导致锁无法被释放，造成死锁或锁泄露问题。
2. **非原子性操作**：尽管 `SETNX` 命令本身是原子性的，但是获取锁和释放锁通常需要多个命令的组合，例如获取锁时需要执行 `SETNX`，释放锁时需要执行 `DEL`。这种组合操作不是原子性的，可能会导致锁的不一致性问题。

为了解决这些问题，可以采用以下方法：

1. **配合 `EXPIRE` 命令设置过期时间**：在获取锁成功后，使用 `EXPIRE` 命令为锁设置一个合理的过期时间，确保即使持有锁的客户端发生异常，锁也能在一定时间后自动释放。
2. **使用 Lua 脚本确保原子性**：将获取锁和释放锁的操作封装在 Lua 脚本中执行，Lua 脚本可以在 Redis 中以原子性的方式执行多个命令，确保获取锁和释放锁的操作是原子性的，避免了竞态条件的发生。
3. **考虑使用 Redlock 算法等更复杂的分布式锁方案**：如果应用场景要求更高的分布式锁安全性和可靠性，可以考虑使用 Redlock 算法等更复杂的分布式锁方案，这些方案通常基于多个 Redis 实例，并结合超时机制和复制机制来保证分布式锁的安全性和可靠性。

11、层次遍历一个 DAG 图，有向无环图。

以下是用 Java 实现层次遍历（BFS）一个有向无环图（DAG）的代码示例：

```
import java.util.*;

public class TopologicalSort {
    public List<Integer> topologicalSort(int numCourses, int[][] prerequisites)
{
        List<Integer> result = new ArrayList<>();
        if (numCourses <= 0 || prerequisites == null || prerequisites.length ==
0) {
            return result;
        }
```

```
        // 创建入度数组和邻接表
        int[] indegree = new int[numCourses];
        List<List<Integer>> adjacencyList = new ArrayList<>();
        for (int i = 0; i < numCourses; i++) {
            adjacencyList.add(new ArrayList<>());
        }

        // 构建入度数组和邻接表
        for (int[] prerequisite : prerequisites) {
            int course = prerequisite[0];
            int prerequisiteCourse = prerequisite[1];
            indegree[course]++;
            adjacencyList.get(prerequisiteCourse).add(course);
        }

        // 使用队列进行拓扑排序
        Queue<Integer> queue = new LinkedList<>();
        for (int i = 0; i < numCourses; i++) {
            if (indegree[i] == 0) {
                queue.offer(i);
            }
        }

        while (!queue.isEmpty()) {
            int course = queue.poll();
            result.add(course);
            for (int neighbor : adjacencyList.get(course)) {
                indegree[neighbor]--;
                if (indegree[neighbor] == 0) {
                    queue.offer(neighbor);
                }
            }
        }

        // 如果有环，则无法完成拓扑排序
        if (result.size() != numCourses) {
            return new ArrayList<>();
        }

        return result;
    }

    public static void main(String[] args) {
        int numCourses = 4;
        int[][] prerequisites = {{1, 0}, {2, 0}, {3, 1}, {3, 2}};
        TopologicalSort solution = new TopologicalSort();
        List<Integer> result = solution.topologicalSort(numCourses,
prerequisites);
        System.out.println("Topological order: " + result); // Output: [0, 1, 2,
3]
    }
}
```

在这个示例中，定了一个 `TopologicalSort` 类，其中包含 `topologicalSort` 方法用于对有向无环图进行层次遍历（拓扑排序）。在 `topologicalSort` 方法中，我们首先构建了入度数组和邻接表，然后使用队列进行拓扑排序，并将排序结果保存在 `result` 中。最后，我们检查排序结果是否完整，如果存在环则返回空列表。

在 `main` 方法中，我们定义了一个示例的有向无环图，然后调用 `topologicalSort` 方法进行层次遍历，并输出排序结果。

12、leetcode 岛屿数量

以下是用 Java 实现 LeetCode 上「岛屿数量」问题的代码示例：

```
public class NumIslands {
    public int numIslands(char[][] grid) {
        if (grid == null || grid.length == 0 || grid[0].length == 0) {
            return 0;
        }

        int numIslands = 0;
        int rows = grid.length;
        int cols = grid[0].length;

        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                if (grid[i][j] == '1') {
                    numIslands++;
                    dfs(grid, i, j);
                }
            }
        }

        return numIslands;
    }

    private void dfs(char[][] grid, int i, int j) {
        int rows = grid.length;
        int cols = grid[0].length;

        if (i < 0 || i >= rows || j < 0 || j >= cols || grid[i][j] == '0') {
            return;
        }

        grid[i][j] = '0'; // Mark the current cell as visited

        // Explore the four neighboring cells
        dfs(grid, i - 1, j);
        dfs(grid, i + 1, j);
        dfs(grid, i, j - 1);
        dfs(grid, i, j + 1);
    }

    public static void main(String[] args) {
        char[][] grid = {
            {'1', '1', '0', '0', '0'},
            {'1', '1', '0', '0', '0'},
            {'0', '0', '1', '0', '0'},
```

```
            {'0', '0', '0', '1', '1'}
        };
        NumIslands solution = new NumIslands();
        int numIslands = solution.numIslands(grid);
        System.out.println("Number of islands: " + numIslands); // Output: 3
    }
}
```

在这个示例中，定义了一个 `NumIslands` 类，其中包含 `numIslands` 方法用于计算岛屿数量。在 `numIslands` 方法中，我们使用深度优先搜索（DFS）来遍历二维网格，并统计岛屿的数量。具体的深度优先搜索逻辑由 `dfs` 方法实现。在 `main` 方法中，我们定义了一个示例的二维网格 `grid`，并调用 `numIslands` 方法计算岛屿数量。

13、 有一个主任务，它有四个子任务，分别存在下面几种状态:wait,running，success，fail。用if-else 的形式写出所有可能发生的情况。

以下是使用 Java 实现主任务及其四个子任务可能的状态组合的代码示例：

```
public class MainTask {
    public static void main(String[] args) {
        String mainTaskStatus = "wait"; // 主任务状态
        String subTask1Status = "wait"; // 子任务1状态
        String subTask2Status = "running"; // 子任务2状态
        String subTask3Status = "success"; // 子任务3状态
        String subTask4Status = "fail"; // 子任务4状态

        if (mainTaskStatus.equals("wait")) {
            if (subTask1Status.equals("wait")) {
                System.out.println("主任务等待，子任务1等待");
            } else if (subTask1Status.equals("running")) {
                System.out.println("主任务等待，子任务1运行中");
            } else if (subTask1Status.equals("success")) {
                System.out.println("主任务等待，子任务1成功");
            } else if (subTask1Status.equals("fail")) {
                System.out.println("主任务等待，子任务1失败");
            }
        } else if (mainTaskStatus.equals("running")) {
            // 类似处理其他状态组合...
        } else if (mainTaskStatus.equals("success")) {
            // 类似处理其他状态组合...
        } else if (mainTaskStatus.equals("fail")) {
            // 类似处理其他状态组合...
        }
    }
}
```

这段 Java 代码中，定义了一个 `MainTask` 类，其中包含了 `main()` 方法作为程序入口。在 `main()` 方法中，定义了主任务和四个子任务的状态变量，并使用嵌套的 `if-else` 条件语句来处理不同的状态组合。根据实际需求，可以在每个条件分支中添加相应的逻辑处理。

分享者：@我真的很帅阿

9. 腾讯Java后端面经，范围极广|0306

1. 手撕算法（30min）

- 括号匹配
- 遍历二叉搜索树
- 给定一组非负整数，求能拼接成的最大整数。（没做对）

2. 八股

- Java语言和C语言有哪些区别?
- RPC了解吗?
- 介绍常用的Linux命令?
- Linux中如果出现了CPU满载，如何排查问题?
- 介绍一下 堆 这个数据结构?
- 动态链接库和静态链接库的区别?
- 介绍一下动态规划的思想？上楼梯问题了解吗？用了动态规划和暴力时间复杂度分别是多少?
- 什么是稳定排序?

3. 项目

- 图片分片上传的逻辑是什么?
- 介绍一下缓存是如何做的?
- 水平分表是怎么做的？分片键具体是怎么设置的?
- 如何用消息队列做削峰填谷的?

4.反问

- 腾讯WXG组上班作息？（上午10点到晚上21:00，双休）
- 你们真的都是双休吗？我很不可思议。（面试官：我们上一次周六上班还是2021年）
- 团建或者聚餐大概多久一次？（一个月一次）
- 成都租房贵吗？（不贵，也不重要，租房钱和工资相比就是洒洒水）
- 您在公司是做什么的？（面试官：我是C++工程师，平时写一些内部的通信组件）

1. 括号匹配

```
import java.util.Stack;

public class BracketMatching {
    public static boolean isBracketMatching(String str) {
        Stack<Character> stack = new Stack<>(); // 创建一个栈用于存储左括号

        for (int i = 0; i < str.length(); i++) {
            char ch = str.charAt(i);

            if (ch == '(' || ch == '[' || ch == '{') { // 如果是左括号，则入栈
                stack.push(ch);
            } else if (ch == ')' || ch == ']' || ch == '}') { // 如果是右括号
```

```
                    if (stack.isEmpty()) { // 如果栈为空，说明没有与之匹配的左括号，返回false
                        return false;
                    }

                    char top = stack.pop(); // 弹出栈顶元素

                    // 判断右括号与栈顶元素是否匹配
                    if ((ch == ')' && top != '(') || (ch == ']' && top != '[') || (ch == '}' && top != '{')) {
                        return false; // 不匹配，返回false
                    }
                }
            }

            return stack.isEmpty(); // 如果栈为空，说明所有括号都匹配成功，返回true；否则返回false
        }

        public static void main(String[] args) {
            String str1 = "(((())))";
            String str2 = "([{}])";
            String str3 = "({[}])";

            System.out.println(isBracketMatching(str1));  // true
            System.out.println(isBracketMatching(str2));  // true
            System.out.println(isBracketMatching(str3));  // false
        }
    }
```

2. 遍历二叉搜索树

```
    // 定义二叉树节点类
    class TreeNode {
        int val;
        TreeNode left;
        TreeNode right;

        public TreeNode(int val) {
            this.val = val;
        }
    }

    public class BSTTraversal {
        // 中序遍历二叉搜索树
        public static void inorderTraversal(TreeNode root) {
            if (root == null) {
                return;
            }

            inorderTraversal(root.left); // 递归遍历左子树
            System.out.print(root.val + " "); // 输出当前节点的值
            inorderTraversal(root.right); // 递归遍历右子树
        }
```

```
    // 前序遍历二叉搜索树
    public static void preorderTraversal(TreeNode root) {
        if (root == null) {
            return;
        }

        System.out.print(root.val + " "); // 输出当前节点的值
        preorderTraversal(root.left); // 递归遍历左子树
        preorderTraversal(root.right); // 递归遍历右子树
    }

    // 后序遍历二叉搜索树
    public static void postorderTraversal(TreeNode root) {
        if (root == null) {
            return;
        }

        postorderTraversal(root.left); // 递归遍历左子树
        postorderTraversal(root.right); // 递归遍历右子树
        System.out.print(root.val + " "); // 输出当前节点的值
    }

    public static void main(String[] args) {
        // 构建一个二叉搜索树
        TreeNode root = new TreeNode(4);
        root.left = new TreeNode(2);
        root.right = new TreeNode(6);
        root.left.left = new TreeNode(1);
        root.left.right = new TreeNode(3);
        root.right.left = new TreeNode(5);
        root.right.right = new TreeNode(7);

        System.out.print("Inorder traversal: ");
        inorderTraversal(root); // 中序遍历
        System.out.println();

        System.out.print("Preorder traversal: ");
        preorderTraversal(root); // 前序遍历
        System.out.println();

        System.out.print("Postorder traversal: ");
        postorderTraversal(root); // 后序遍历
        System.out.println();
    }
}
```

3. 给定一组非负整数，求能拼接成的最大整数。

```
import java.util.Arrays;
import java.util.Comparator;

public class MaxNumber {
```

```
    public static String largestNumber(int[] nums) {
        // 将整数数组转换为字符串数组
        String[] strNums = new String[nums.length];
        for (int i = 0; i < nums.length; i++) {
            strNums[i] = String.valueOf(nums[i]);
        }

        // 自定义比较器，用于比较两个字符串的拼接结果
        Comparator<String> comparator = new Comparator<String>() {
            @Override
            public int compare(String s1, String s2) {
                String order1 = s1 + s2;
                String order2 = s2 + s1;
                return order2.compareTo(order1); // 降序排列
            }
        };

        // 使用自定义比较器对字符串数组进行排序
        Arrays.sort(strNums, comparator);

        // 如果排序后的数组第一个元素是0，则直接返回"0"
        if (strNums[0].equals("0")) {
            return "0";
        }

        // 拼接排序后的字符串数组
        StringBuilder sb = new StringBuilder();
        for (String str : strNums) {
            sb.append(str);
        }

        return sb.toString();
    }

    public static void main(String[] args) {
        int[] nums = {10, 2, 5, 9, 23};
        String result = largestNumber(nums);
        System.out.println("最大整数: " + result);
    }
}
```

4. Java语言和C语言有哪些区别?

- 编程范式：C语言是一种过程式编程语言，而Java语言是一种面向对象编程语言。Java语言基于类和对象的概念，支持封装、继承和多态等面向对象的特性。
- 内存管理：C语言需要手动管理内存，包括分配和释放内存。而Java语言使用垃圾回收机制，自动管理内存，开发者不需要显式地进行内存管理。
- 平台依赖性：C语言是一种编译型语言，编写的程序需要针对特定的操作系统和硬件平台进行编译。而Java语言是一种跨平台的语言，通过Java虚拟机（JVM）实现了平台无关性，Java程序可以在不同的操作系统上运行。
- 异常处理：C语言使用错误码来处理异常情况，开发者需要手动检查错误码并进行相应的处理。Java语言引入了异常处理机制，通过try-catch-finally语句块来捕获和处理异常，使得代码更加清晰和可读。

- 标准库和生态系统：C语言的标准库相对较小，提供了基本的输入输出、字符串处理等功能。Java语言拥有丰富的标准库和第三方库，提供了大量的API和工具，涵盖了各种领域，方便开发者进行开发。
- 编译和执行方式：C语言通过编译器将源代码编译成机器码，然后直接执行。而Java语言通过编译器将源代码编译成字节码，然后在JVM上执行字节码。

5. RPC了解吗?

RPC（Remote Procedure Call，远程过程调用）。

RPC是一种通信协议，用于不同计算机之间的远程通信。它允许一个计算机程序调用另一个计算机上的函数或方法，就像调用本地函数一样，隐藏了底层网络通信的细节。

在RPC中，通信的两端分别是客户端和服务器。客户端发起一个远程调用请求，服务器接收请求并执行相应的操作，然后将结果返回给客户端。RPC框架负责处理底层的网络通信、序列化和反序列化等细节，使得远程调用过程对开发者透明。

常见的RPC框架包括Dubbo、gRPC、Thrift等。

RPC的优点包括：

- 简化分布式系统开发：RPC隐藏了底层通信细节，使得开发者可以像调用本地函数一样调用远程函数，简化了分布式系统的开发。
- 提高系统性能：RPC可以将计算任务分布到不同的服务器上，充分利用资源，提高系统的并发性和性能。
- 提高系统可扩展性：通过RPC，可以将系统拆分成多个服务，每个服务可以独立部署和扩展，提高系统的可扩展性。

6. 介绍常用的Linux命令?

1. `ls`：列出目录内容。
2. `cd`：切换当前工作目录。
3. `pwd`：显示当前工作目录的路径。
4. `mkdir`：创建新目录。
5. `rm`：删除文件或目录。
6. `cp`：复制文件或目录。
7. `mv`：移动文件或目录，也可用于重命名文件或目录。
8. `cat`：显示文件内容。
9. `grep`：在文件中搜索指定的模式。
10. `chmod`：修改文件或目录的权限。

7. Linux中如果出现了CPU满载，如何排查问题?

当Linux系统出现CPU满载的情况时，可以按照以下步骤进行问题排查：

1. 使用`top`命令查看系统当前的进程和CPU占用情况。按下`Shift + P`按CPU使用率排序进程，观察哪些进程占用了较高的CPU资源。
2. 使用`ps`命令结合`top`命令的结果，获取更详细的进程信息。例如，`ps auxf`可以显示所有进程的详细信息，包括进程ID、CPU使用率等。
3. 使用`htop`命令代替`top`命令，它提供了更友好的交互式界面，可以更方便地查看和管理进程。
4. 使用`pidstat`命令监测特定进程的CPU使用情况。例如，`pidstat -p <进程ID> -u 1`可以每秒钟显示指定进程的CPU使用情况。
5. 使用`iotop`命令查看磁盘I/O情况，因为高磁盘I/O可能导致CPU负载升高。
6. 使用`sar`命令查看系统的历史性能数据，例如，`sar -u`可以显示CPU使用率的历史数据。

7. 使用`strace`命令追踪进程的系统调用，以确定是否有异常的系统调用导致CPU满载。
8. 检查系统日志文件，如`/var/log/messages`、`/var/log/syslog`等，查找异常或错误信息。
9. 如果有可疑的进程或服务，可以尝试重启它们，或者使用`kill`命令终止它们。
10. 如果以上方法无法解决问题，可以考虑使用性能分析工具，如`perf`、`strace`、`gdb`等，对进程进行更深入的分析和调试。

8. 介绍一下 堆 这个数据结构?

堆（Heap）是一种特殊的树状数据结构，它满足以下两个主要性质：

1. 堆是一个完全二叉树（Complete Binary Tree）：除了最底层外，其他层的节点都是满的，且最底层的节点都靠左排列。
2. 堆中每个节点的值都必须满足堆的性质：对于最大堆（Max Heap），父节点的值大于或等于其子节点的值；对于最小堆（Min Heap），父节点的值小于或等于其子节点的值。

堆通常用于实现优先队列（Priority Queue）和堆排序（Heap Sort）等算法。

堆可以分为最大堆和最小堆两种类型：

- 最大堆：父节点的值大于或等于其子节点的值。根节点是堆中的最大值。
- 最小堆：父节点的值小于或等于其子节点的值。根节点是堆中的最小值。

堆的主要操作包括插入和删除操作：

- 插入操作：将一个新元素插入到堆中，保持堆的性质不变。
- 删除操作：删除堆中的根节点，并保持堆的性质不变。

堆的插入和删除操作的时间复杂度都是O(log n)，其中n是堆中元素的个数。

堆的应用场景包括：

- 优先队列：堆可以用于实现高效的优先队列，根据优先级获取最大或最小元素。
- 堆排序：堆排序是一种基于堆的排序算法，具有稳定的时间复杂度O(n log n)。
- 求Top K问题：通过维护一个大小为K的最小堆或最大堆，可以高效地求解Top K大或Top K小的元素。
- 图算法：堆可以用于实现Dijkstra算法、Prim算法等图算法中的优先级队列。

9. 动态链接库和静态链接库的区别?

动态链接库（Dynamic Link Library，DLL）和静态链接库（Static Link Library）是两种常见的库文件形式，它们在链接和加载方式上有以下区别：

1. 链接方式：
    - 静态链接库：在编译时将库的代码和应用程序的代码合并成一个可执行文件。链接器将库的代码复制到应用程序中，生成一个独立的可执行文件。应用程序与库的代码是静态链接的关系。
    - 动态链接库：在编译时只将库的引用信息记录在可执行文件中，而不将库的代码复制到应用程序中。在运行时，操作系统动态加载并链接库的代码，使得多个应用程序可以共享同一个库文件。
2. 文件大小：
    - 静态链接库：库的代码被完整地复制到每个应用程序中，因此每个应用程序都包含了库的完整代码，导致应用程序的文件大小较大。
    - 动态链接库：多个应用程序可以共享同一个库文件，因此每个应用程序只需要记录库的引用信息，导致应用程序的文件大小较小。
3. 内存占用：

- 静态链接库：每个应用程序都包含了库的完整代码，因此在内存中会有多份库的代码副本，导致内存占用较高。
- 动态链接库：多个应用程序共享同一个库文件，库的代码只需要加载一次，多个应用程序共享同一份库的代码，因此在内存中的占用较少。

4. 更新和维护：
   - 静态链接库：如果库的代码发生更新，需要重新编译和链接应用程序，将新的库代码合并到应用程序中。
   - 动态链接库：如果库的代码发生更新，只需要替换库文件即可，不需要重新编译和链接应用程序。

10. 介绍一下动态规划的思想？上楼梯问题了解吗？用了动态规划和暴力时间复杂度分别是多少？

动态规划（Dynamic Programming）是一种解决多阶段决策问题的优化方法，它将问题分解为多个子问题，并通过保存子问题的解来避免重复计算，从而提高算法的效率。

动态规划的基本思想是：将原问题分解为若干个子问题，先求解子问题的解，然后利用子问题的解构建原问题的解。通过保存子问题的解，避免重复计算，从而减少时间复杂度。

上楼梯问题是动态规划中的一个经典问题。假设有n个台阶，每次可以走1步或2步，问有多少种不同的方式可以爬到楼顶。

使用动态规划解决上楼梯问题的思路如下：

1. 定义状态：设dp[i]表示爬到第i个台阶的不同方式数。
2. 状态转移方程：由于每次可以走1步或2步，所以到达第i个台阶的方式数等于到达第i-1个台阶的方式数加上到达第i-2个台阶的方式数，即dp[i] = dp[i-1] + dp[i-2]。
3. 初始条件：dp[0] = 1，dp[1] = 1，表示到达第0个台阶和第1个台阶的方式数都为1。
4. 计算顺序：从小到大计算dp[i]，直到计算到dp[n]，即到达第n个台阶的方式数。

使用暴力方法解决上楼梯问题的思路如下：

1. 枚举所有可能的方式，对于每一种方式，计算到达楼顶的路径数。
2. 递归地计算每一种方式的路径数，直到到达楼顶。
3. 统计所有方式的路径数之和。

动态规划解决上楼梯问题的时间复杂度为O(n)，因为需要计算从第2个台阶到第n个台阶的方式数，每个台阶只需要计算一次。

暴力方法解决上楼梯问题的时间复杂度为O(2^n)，因为需要枚举所有可能的方式，每个台阶都有两种选择。

11. 什么是稳定排序?

稳定排序（Stable Sorting）是指在排序算法中，如果两个元素的比较结果相等，那么它们在排序后的结果中的相对位置保持不变。换句话说，如果在排序前，元素A在元素B的前面，且A与B的值相等，那么在排序后，A仍然在B的前面。

稳定排序的特点是能够保持相等元素的相对顺序，这在某些应用场景中非常重要。例如，当需要按照多个条件进行排序时，稳定排序可以确保先按照第一个条件排序，再按照第二个条件排序，而不会打乱第一个条件的顺序。

常见的稳定排序算法有冒泡排序、插入排序、归并排序和基数排序等。这些算法在排序过程中会考虑元素的相对位置，以保持排序的稳定性。

相对应的，非稳定排序（Unstable Sorting）是指在排序算法中，如果两个元素的比较结果相等，它们在排序后的结果中的相对位置可能会发生变化。非稳定排序算法在排序过程中可能会打乱相等元素的相对顺序。

在选择排序算法时，如果需要保持相等元素的相对顺序，就应选择稳定排序算法。否则，如果相等元素的相对顺序不重要，可以选择非稳定排序算法，它们通常具有更高的性能。

12. 图片分片上传的逻辑是什么?

图片分片上传是一种将大文件分割成多个小片段进行上传的策略，以提高上传效率和稳定性。其基本逻辑如下：

1. 客户端将待上传的图片文件进行分片切割，将文件分割成多个固定大小的片段（chunk）。
2. 客户端按照一定的顺序将这些分片依次上传到服务器端。可以使用HTTP协议的POST请求或其他上传协议进行分片上传。
3. 服务器端接收到每个分片后，将其暂存到临时存储区，通常是磁盘或内存。
4. 当所有分片都上传完成后，服务器端根据上传的顺序将这些分片进行合并，还原成完整的图片文件。
5. 完整的图片文件可以进行进一步的处理，如存储到数据库或文件系统中，或进行其他业务逻辑操作。

在图片分片上传的过程中，还需要考虑以下几个方面的逻辑：

- 分片大小：需要根据实际情况确定每个分片的大小，通常根据网络环境和服务器性能进行调整，以保证上传效率和稳定性。
- 分片顺序：客户端需要按照一定的顺序上传分片，通常是从第一个分片开始，依次上传到最后一个分片。
- 分片校验：客户端可以对每个分片进行校验，例如计算分片的哈希值，以确保分片的完整性和准确性。
- 断点续传：如果上传过程中出现网络中断或其他异常情况，客户端可以记录已上传的分片信息，下次继续上传时可以从断点处继续上传，以实现断点续传的功能。

通过图片分片上传的方式，可以有效地处理大文件的上传，提高上传效率和稳定性，并且可以灵活控制上传过程，适应不同的网络环境和服务器条件。

13. 如何用消息队列做削峰填谷的?

使用消息队列可以有效地实现削峰填谷的目标，具体的实现方式如下：

1. 创建消息队列：选择合适的消息队列系统，如RabbitMQ、Kafka等，并创建相应的消息队列。
2. 发送消息：将需要处理的任务或请求转化为消息，并发送到消息队列中。这些消息可以包含任务的相关信息，如任务类型、参数等。
3. 消费消息：编写消费者程序，从消息队列中获取消息，并进行相应的处理。消费者可以根据业务需求进行扩展，可以是单个消费者或多个消费者。
4. 控制消费速率：通过控制消费者的数量和处理速度，来控制任务的处理速度。可以动态调整消费者的数量，根据实际情况增加或减少消费者的数量。
5. 峰值处理：当任务量激增时，消息队列可以暂时存储大量的消息，而不会导致系统负载过高。消费者按照自身的处理能力逐步消费消息，避免了系统的峰值压力。
6. 谷值填充：当任务量减少时，消息队列中的消息可以被逐步消费，保证系统的资源得到充分利用，避免了资源的浪费。
7. 异步处理：通过消息队列的异步特性，可以将任务的处理与请求的接收解耦。请求可以快速响应，而任务的处理可以在后台进行，提高系统的响应速度和吞吐量。

通过使用消息队列进行削峰填谷，可以平滑处理系统的高峰期和低谷期，提高系统的稳定性和可伸缩性。同时，消息队列还可以提供消息持久化、消息重试等功能，增加系统的可靠性和容错性。

出处：Zyccccccc

10. 腾讯云后端一面，OMG | 0309

面试官上来夸了一下简历，然后开始做算法题：

给定非递减数组，先求每个数平方 返回平方后的数组，这个数组也要非递减
接下来问了项目里的一些点

1. session里存啥 session和cookie区别
2. redis和mysql双写一致性为什么删缓存而不是更新缓存
3. 布隆过滤器会不会比较耗内存，添加数据怎么办，删除数据怎么办
4. 解决缓存穿透其他方法
5. 缓存击穿解决方案，抛开分布式的话如何解决
6. 为什么要分布式事务
7. 给消息队列发消息失败了怎么办
8. 自研网关需要考虑哪些因素（高可用、轻量级、性能）
9. 限流是针对什么进行限流
10. 限流如何配置

然后突然开始问八股了（很多没答上来，问了一堆云原生、go相关的东西）

1. docker实现原理
2. docker的虚拟网络如何实现
3. xss攻击
4. sql注入
5. 如何防止sql注入
6. mysql的prepare 其实就是jdbc里那个prepare statement 一下子没反应过来
7. mysql唯一索引和主键索引区别 唯一索引可以空吗 主键索引可以空吗
8. redis的pipeline
9. 读接口并发量高怎么优化（限流，缓存，数据库查询优化）
10. 写接口并发量高怎么优化（异步+批量）

两点半到三点半 说好的40分钟结果拷打了一个小时 后二十分钟的八股太折磨了

1. session里存啥，session和cookie区别

解析：

计算机网络基础题，必备题。

参考答案：

> 在Web开发中，Session和Cookie是两种常用的机制，用于在服务器和客户端之间存储和传递数据。
>
> Session是服务器端存储用户信息的一种机制。当用户第一次访问服务器时，服务器会为该用户创建一个唯一的Session ID，并将该ID存储在Cookie中发送给客户端。客户端在后续的请求中会携带该Cookie，服务器通过Session ID可以找到对应的Session数据。服务器可以在Session中存储用户的登录状态、购物车信息等数据，以便在用户不同请求之间保持状态的一致性。Session数据存储在服务器端，相对来说更安全，但会占用服务器的内存资源。
>
> Cookie是存储在客户端的一小段文本信息。服务器在响应中通过Set-Cookie头部将Cookie信息发送给客户端，客户端会将Cookie保存起来，并在后续的请求中携带该Cookie发送给服务器。Cookie可以存储一些用户偏好设置、登录凭证等信息。由于Cookie存储在客户端，所以可以在不同的浏览器和设备之间共享，但也存在一定的安全风险，比如可能被恶意篡改或窃取。
>
> Session和Cookie的区别主要在于存储位置和安全性。Session数据存储在服务器端，相对安全但占用服务器资源；Cookie存储在客户端，方便共享但存在一定的安全风险。

学习指导： Cookie和Session的区别

> **面经专栏直通车**

> **面经专栏下载**

2. redis和mysql双写一致性为什么删缓存而不是更新缓存

解析：

考察Redis作为缓存相关知识，属于常考题，必备题，难度中等。

参考答案

> 主要有以下几个原因：
>
> 1. 数据一致性：当数据发生更新时，为了保证数据的一致性，需要先更新数据库，再更新缓存。如果先更新缓存，再更新数据库，可能会导致数据库更新失败或发生异常，导致数据库和缓存之间的数据不一致。而如果选择删除缓存，下次请求时会重新从数据库中读取最新数据并更新缓存，确保了数据的一致性。
> 2. 并发写入：在高并发的场景下，多个请求同时写入数据库可能会导致并发冲突和数据不一致的问题。如果先更新缓存，再更新数据库，可能会导致多个请求同时读取到旧的缓存数据并同时写入数据库，造成数据冲突。而删除缓存后，下次请求会重新从数据库读取最新数据并更新缓存，避免了并发写入的问题。
> 3. 缓存更新成本：更新缓存需要进行网络通信和缓存的写入操作，相对于删除缓存来说，更新缓存的成本更高。而删除缓存后，下次请求会重新从数据库读取最新数据并更新缓存，可以减少缓存更新的成本。
>
> 删除缓存后，下次请求会重新从数据库读取最新数据并更新缓存，可能会对系统的性能产生一定的影响，因为需要进行数据库查询和缓存写入的操作。因此，在实际应用中，需要根据具体的业务场景和性能需求来选择适合的缓存策略。

学习指导： Redis与MySQL双写一致性如何保证？

3. 布隆过滤器会不会比较耗内存，添加数据怎么办，删除数据怎么办

解析：

数据结构的基本应用，数据近今年新出的题，一般与Redis易通考察。、

参考答案

> 布隆过滤器是一种用于快速判断一个元素是否存在于集合中的数据结构，它的优点是查询效率高且占用内存较少。然而，布隆过滤器也存在一些限制和操作上的考虑。
>
> 1. 内存消耗：布隆过滤器的内存消耗主要取决于预期的误判率和要存储的元素数量。误判率越低，所需的内存空间就越大。一般情况下，布隆过滤器的内存消耗相对较小，但随着元素数量的增加，内存占用也会逐渐增加。
> 2. 添加数据：向布隆过滤器添加数据时，需要对元素进行哈希计算，并将对应的位标记为1。如果哈希冲突较多，可能会导致位的重复标记，进而影响误判率。在添加数据时，可以适当调整布隆过滤器的容量和哈希函数的数量，以平衡误判率和内存消耗。
> 3. 删除数据：布隆过滤器本身不支持直接删除已添加的元素，因为删除一个元素可能会影响其他元素的判断结果。如果需要删除元素，一种常见的做法是使用计数器或其他数据结构辅助记录元素的添加次数，然后在判断元素是否存在时，根据计数器的值进行判断。当计数器为0时，可以认为元素不存在。
>
> 需要注意的是，布隆过滤器在判断元素是否存在时，可能存在一定的误判率。因此，在使用布隆过滤器时，需要根据具体的应用场景和需求来选择合适的误判率和内存消耗。

学习指导：布隆过滤器优化内存占用过大问题

4. 解决缓存穿透其他方法

解析：

Redis 相关知识，属于必考题。

参考答案：

> 访问一个缓存和数据库都不存在的 key，此时会直接打到数据库上，并且查不到数据，没法写缓存，所以下一次同样会打到数据库上。此时，缓存起不到作用，请求每次都会走到数据库，流量大时数据库可能会被打挂。此时缓存就好像被“穿透”了一样，起不到任何作用。
>
> **解决方案：**
>
> 1、**接口校验。**在正常业务流程中可能会存在少量访问不存在 key 的情况，但是一般不会出现大量的情况，所以这种场景最大的可能性是遭受了非法攻击。可以在最外层先做一层校验：用户鉴权、数据合法性校验等，例如商品查询中，商品的ID是正整数，则可以直接对非正整数直接过滤等等。
>
> 2、**缓存空值。**当访问缓存和DB都没有查询到值时，可以将空值写进缓存，但是设置较短的过期时间，该时间需要根据产品业务特性来设置。
>
> 3、**布隆过滤器。**使用布隆过滤器存储所有可能访问的 key，不存在的 key 直接被过滤，存在的 key 则再进一步查询缓存和数据库。

学习指导：缓存穿透问题，线上解决方案

5. 缓存击穿解决方案，抛开分布式的话如何解决

解析：

Redis 相关知识，属于必考题。

> 缓存击穿：某一个热点 key，在缓存过期的一瞬间，同时有大量的请求打进来，由于此时缓存过期了，所以请求最终都会走到数据库，造成瞬时数据库请求量大、压力骤增，甚至可能打垮数据库。
>
> 1. 设置热点数据永不过期：对于一些热点数据，可以将其缓存设置为永不过期，确保即使缓存失效，也能够从缓存中获取到数据。这样可以避免热点数据失效后，大量请求直接访问数据库。
> 2. 加锁机制：在缓存失效的瞬间，可以使用互斥锁（如分布式锁）来保证只有一个请求能够访问数据库，其他请求等待结果。当第一个请求从数据库中获取到数据后，更新缓存并释放锁，其他请求再从缓存中获取数据。
> 3. 缓存预热：在系统启动或者定时任务中，将热点数据预先加载到缓存中。这样可以避免在热点数据失效时，大量请求直接访问数据库，而是直接从缓存中获取数据。
> 4. 异步更新缓存：在数据更新时，先更新数据库，然后异步更新缓存。这样可以保证数据的一致性，同时减少对数据库的访问延迟。
> 5. 限流和熔断：对请求进行限流和熔断处理，当请求量过大时，可以直接拒绝请求或者返回默认结果，避免对数据库造成过大的压力。

学习指导：缓存穿透、缓存击穿、缓存雪崩解决方案

6. 给消息队列发消息失败了怎么办

解析：

消息队列相关知识，难度中等，常考题

参考答案：

> 当给消息队列发送消息失败时，可以采取以下几种处理方式：
>
> 1. 重试发送：首先，可以尝试重新发送消息。消息队列通常提供了重试机制，可以设置重试次数和重试间隔。在发送失败后，可以根据重试策略进行自动或手动的重试操作，直到消息发送成功或达到最大重试次数。
> 2. 消息持久化：如果消息发送失败，可以将消息持久化到本地或者其他存储介质中，以便稍后进行重试。在持久化时，需要注意保证消息的唯一性和顺序性，避免重复发送或乱序发送。
> 3. 错误日志记录：记录发送失败的消息和相关错误信息到日志中，以便后续进行排查和处理。错误日志可以包含发送失败的消息内容、发送时间、错误码等信息，有助于定位问题和进行故障排除。
> 4. 监控和报警：建立监控系统，实时监控消息队列的发送状态和错误情况。当发现消息发送失败时，及时触发报警机制，通知相关人员进行处理和修复。
> 5. 容错处理：在设计系统时，可以考虑引入容错机制，例如使用备份队列或者消息重放机制。当主队列发送失败时，可以将消息发送到备份队列，或者通过消息重放机制重新发送消息，以确保消息的可靠性。
>
> 需要根据具体的业务场景和消息队列的特性选择合适的处理方式。同时，也要注意监控和及时处理发送失败的消息，以保证系统的稳定性和可靠性。

学习指导：RabbitMq消息丢失原因及其解决方案

7. 限流是针对什么进行限流

解析：

后端技术的常考提，难度简单。

参考答案：

> 限流是一种流量控制的手段，用于限制系统或服务的请求流量，以保护系统免受过载或恶意攻击的影响。限流可以针对以下几个方面进行限制：
>
> 1. 请求频率限流：限制单位时间内的请求次数或请求速率。例如，限制每秒钟的请求次数不超过某个阈值，或者限制每分钟的请求速率不超过某个阈值。这种限流方式可以防止系统被过多的请求压垮，保证系统的稳定性和可用性。
> 2. 并发连接限流：限制同时连接到系统的请求数量。通过设置最大并发连接数，可以防止系统因为过多的并发请求而资源耗尽或崩溃。这种限流方式适用于需要控制系统负载和资源消耗的场景。
> 3. API接口限流：对特定的API接口进行限流。可以根据接口的重要性、资源消耗情况、用户权限等因素，设置不同的限流策略。这种限流方式可以保护重要接口免受滥用或恶意攻击，确保接口的可用性和稳定性。
> 4. IP限流：对特定IP地址进行限流。可以根据IP地址的访问频率、访问行为等进行限制，防止某个IP地址对系统进行恶意攻击、爬虫行为或者过多的请求。这种限流方式可以保护系统免受恶意访问的影响。
>
> 限流的目的是为了保护系统的稳定性、可用性和安全性，防止系统被过载或恶意攻击。具体的限流策略和方式需要根据系统的实际情况和需求进行选择和配置。

学习指导：什么是限流？

8. docker实现原理

解析：

简历提到了docker，一般会问。

参考答案：

> Docker 是一种开源的容器化平台，它的实现原理主要包括以下几个方面：
>
> 1. 命名空间（Namespaces）：Docker 使用 Linux 的命名空间技术，包括 PID 命名空间、网络命名空间、挂载命名空间等，通过隔离进程、网络、文件系统等资源，使得每个容器拥有独立的运行环境。
> 2. 控制组（Cgroups）：Docker 使用 Linux 的控制组技术，通过限制和管理资源的使用，如 CPU、内存、磁盘、网络带宽等，实现对容器资源的控制和隔离。
> 3. 联合文件系统（UnionFS）：Docker 使用联合文件系统来构建容器的文件系统。联合文件系统允许将多个文件系统挂载到同一个目录下，形成一个统一的文件系统视图。Docker 默认使用的联合文件系统是 OverlayFS。
> 4. 容器镜像（Container Image）：Docker 使用容器镜像来打包和分发应用程序及其依赖。容器镜像是一个只读的模板，包含了运行应用程序所需的文件系统、库、环境变量等。Docker 镜像采用分层存储的方式，每一层都是一个只读的文件系统，多个镜像可以共享相同的基础层，节省存储空间。
> 5. 容器运行时（Container Runtime）：Docker 使用容器运行时来创建和管理容器。常用的容器运行时包括 Docker 自带的 Docker Engine、Containerd、CRI-O 等。容器运行时负责加载容器镜像、创建容器的命名空间和控制组、启动容器进程等。
> 6. Docker 守护进程（Docker Daemon）：Docker 守护进程是 Docker 的核心组件，负责接收和处理用户的命令，管理容器的生命周期，与容器运行时进行交互，监控容器的状态等。
>
> 通过以上的技术和组件，Docker 实现了轻量级、快速启动、隔离性好的容器化环境。用户可以使用 Docker 命令和 API 来创建、启动、停止、删除容器，以及构建、推送、拉取和管理容器镜像。

学习指导：一文弄懂Docker核心技术与实现原理

9. docker的虚拟网络如何实现

解析：

简历提到了docker，一般会问。

参考答案：

> Docker的虚拟网络实现主要依赖于Linux内核提供的Network Namespace功能，该功能可以创建多个隔离的网络空间，每个网络空间都有独自的网络栈信息。当Docker创建一个容器时，会执行一系列操作来设置和配置容器的虚拟网络。
>
> 以下是Docker创建一个容器时执行的主要操作：
>
> 1. 创建一对虚拟接口：Docker会在本地主机（Docker宿主机）和新创建的容器中各创建一个虚拟接口。
> 2. 连接虚拟接口到网桥：本地主机一端的虚拟接口会连接到默认的docker0网桥或指定的网桥上，并会获得一个以“veth”开头的唯一名字，如veth1234。
> 3. 放置并重命名容器端虚拟接口：容器一端的虚拟接口会被放置到新创建的容器中，并修改其名字，通常是“eth0”。这个接口只在容器的命名空间内可见。
> 4. 分配IP地址：Docker会从网桥可用地址段中获取一个空闲地址，例如172.17.0.2/16，并将其分配给容器的eth0接口。同时，还会配置默认路由网关为docker0网卡的内部接口IP地址。
>
> 完成上述操作后，容器就可以使用其eth0虚拟网卡来连接其他容器和访问外部网络。这种机制允许容器仿佛自己都在独立的网络中运行，而不同Network Namespace的资源相互不可见，彼此之间无法通信。
>
> 此外，Docker还提供了多种网络模式供用户选择，包括bridge模式（默认值，连接到默认的网桥）、host模式（容器使用本地主机的网络，拥有完全的本地主机接口访问权限）等。用户可以通过docker network命令来手动管理网络，定制适合自身需求的网络配置。

学习指导：Docker 网络模式详解及容器间网络通信

10. xss攻击

解析：

常见的安全相关的问题，如果问安全相关的题，属于常考题。

参考答案：

> XSS（Cross-Site Scripting）攻击是一种常见的网络安全漏洞，攻击者通过注入恶意脚本代码到受信任的网页中，使得用户在浏览器中执行这些恶意脚本，从而达到攻击的目的。XSS 攻击可以分为以下几种类型：
>
> 1. 存储型 XSS：攻击者将恶意脚本代码存储到目标网站的数据库中，当其他用户访问包含恶意代码的页面时，恶意代码会从服务器返回并在用户浏览器中执行。
> 2. 反射型 XSS：攻击者将恶意脚本代码作为参数附加在 URL 中，当用户点击包含恶意代码的链接时，服务器将恶意代码返回给用户浏览器并执行。
> 3. DOM 型 XSS：攻击者通过修改网页的 DOM 结构，使得恶意脚本代码被执行。这种攻击方式不涉及服务器的参与，而是直接在用户浏览器中执行。
>
> XSS 攻击可能导致以下危害：
>
> - 盗取用户敏感信息，如登录凭证、个人信息等。
> - 劫持用户会话，进行恶意操作。
> - 在受攻击网站上注入恶意广告或重定向到恶意网站。
> - 篡改网页内容，欺骗用户或破坏网站的可信度。
>
> 为了防止 XSS 攻击，可以采取以下措施：

- 输入验证和过滤：对用户输入的数据进行验证和过滤，确保只接受合法的数据。
- 输出编码：在将用户输入的数据输出到网页时，进行适当的编码，防止恶意代码被执行。
- 使用 HTTP 头部中的 Content Security Policy（CSP）来限制页面中可执行的脚本。
- 设置 HttpOnly 属性，防止恶意脚本通过 JavaScript 访问敏感的 Cookie 数据。
- 对于存储型 XSS，对用户输入的数据进行严格的过滤和转义，确保恶意代码无法存储到数据库中。

学习指导：web攻防之XSS攻击详解——XSS简介与类型

11. sql注入

解析：

常见的安全相关的问题，如果问安全相关的题，属于常考题。

> SQL注入是一种常见的网络安全漏洞，攻击者通过在应用程序的输入字段中注入恶意的SQL代码，从而绕过应用程序的输入验证，执行恶意的SQL查询或命令。这可能导致数据库被攻击者非法访问、数据泄露、数据篡改或系统瘫痪等问题。
>
> SQL注入攻击可以分为以下几种类型：
>
> 1. 基于错误的注入：攻击者通过构造恶意的SQL语句，利用应用程序返回的错误信息来获取数据库的信息。
> 2. 基于布尔的盲注入：攻击者通过构造恶意的SQL语句，利用应用程序的不同响应来判断SQL语句的执行结果，从而逐步获取数据库的信息。
> 3. 基于时间的盲注入：攻击者通过构造恶意的SQL语句，利用应用程序的延迟响应来判断SQL语句的执行结果，从而逐步获取数据库的信息。

学习指导：入坑解读 | 什么是SQL注入？

15. 如何防止sql注入

解析：

常见的安全相关的问题，如果问安全相关的题，属于常考题。

参考答案：

> 要防止SQL注入攻击，可以采取以下几个关键措施：
>
> 1. 输入验证和过滤：对用户输入的数据进行验证和过滤，确保只接受合法的数据。可以使用白名单或正则表达式来验证输入数据的格式和内容，并拒绝包含特殊字符或恶意代码的输入。
> 2. 使用参数化查询或预编译语句：使用参数化查询或预编译语句可以将用户输入的数据作为参数传递给SQL查询，而不是将其直接拼接到SQL语句中。这样可以防止攻击者通过注入恶意代码来改变SQL查询的逻辑。
> 3. 避免动态拼接SQL语句：尽量避免在应用程序中动态拼接SQL语句，特别是使用用户输入的数据拼接SQL语句。如果必须拼接SQL语句，确保对用户输入的数据进行适当的转义或编码，以防止恶意代码的注入。
> 4. 最小权限原则：数据库用户应该具有最小的权限，只能访问必要的数据和执行必要的操作。限制数据库用户的权限可以减少攻击者的影响范围，即使发生SQL注入攻击，也能最大程度地减少损失。
> 5. 使用安全的开发框架和库：使用经过安全审计和验证的开发框架和库，这些框架和库通常会提供内置的防御机制来防止SQL注入攻击。例如，ORM（对象关系映射）框架可以自动处理参数化查询，从而减少了手动编写SQL语句的风险。
> 6. 日志记录和监控：记录应用程序的日志，并监控异常的SQL查询，及时发现和阻止潜在的注入攻击。定期审查日志，以便及时发现异常行为并采取相应的应对措施。

7. 定期更新和维护：及时更新和维护应用程序和数据库的软件版本，以修复已知的安全漏洞。同时，定期进行安全审计和漏洞扫描，以发现和修复潜在的SQL注入漏洞。

通过综合应用上述措施，可以大大降低SQL注入攻击的风险，并保护应用程序和数据库的安全。

学习指导：sql注入攻击的原理以及防范措施

16. mysql的prepare

解析：

MySQL 比较冷门的题，适当了解即可。

参考答案：

MySQL的PREPARE语句是一种用于执行动态SQL的机制。它允许在执行之前预先准备SQL语句，并在需要时执行该语句。PREPARE语句的语法如下：

```
复制PREPARE statement_name FROM 'sql_statement';
```

其中，`statement_name`是自定义的准备语句的名称，`sql_statement`是要准备的SQL语句。

使用PREPARE语句的好处是可以将SQL语句与参数分离，从而提高查询的效率和安全性。通过将参数作为占位符（如`?`）放入SQL语句中，然后在执行时将具体的参数值传递给准备语句，可以避免SQL注入攻击，并且可以重复使用准备语句以提高性能。

下面是一个使用PREPARE语句的示例：

```
复制PREPARE stmt FROM 'SELECT * FROM users WHERE id = ?';
SET @id = 1;
EXECUTE stmt USING @id;
```

在上述示例中，首先使用PREPARE语句准备了一个查询语句，然后使用EXECUTE语句执行该准备语句，并通过USING子句将参数值传递给准备语句。

执行PREPARE语句后，可以使用DEALLOCATE PREPARE语句来释放准备语句的资源，例如：

```
复制DEALLOCATE PREPARE stmt;
```

通过PREPARE语句，可以动态构建和执行SQL语句，提高查询的效率和安全性。但需要注意的是，PREPARE语句在使用时需要谨慎处理参数，确保参数的合法性和正确性，以避免潜在的安全风险。

学习指导：mysql 存储过程中的 prepare语句

17. mysql唯一索引和主键索引区别 唯一索引可以空吗 主键索引可以空吗

解析：

MySQL 常考题，必会题。

参考答案：

唯一索引和主键索引在MySQL中有一些区别，包括是否允许为空和是否可以有重复值。

1. 唯一索引（Unique Index）：

    - 唯一索引允许为空值，即可以在索引列中存储NULL值。
    - 唯一索引可以用来保证数据表中某列的值是唯一的。
2. 主键索引（Primary Key Index）：
    - 主键索引不允许为空值，即主键列不能为空。
    - 主键索引要求每个索引值都是唯一的。
    - 主键索引用于唯一标识数据表中的每一行数据，每个数据表只能有一个主键。

学习指导：sql：主键和唯一索引区别

18. redis的pipeline

解析：

Redis 的中等难度题，常考题。

参考答案：

Redis的pipeline是一种在客户端与服务器之间进行批量命令传输和响应的技术。它允许客户端一次性发送多个命令给服务器，而不需要等待每个命令的响应。服务器收到这些命令后，会将它们按顺序执行，并将所有命令的响应一次性返回给客户端，从而减少了通信的往返次数，提高了性能和吞吐量。Pipeline对于需要批量处理的场景非常有用，例如批量写入、批量读取等。

学习指导：Redis Pipeline这一篇就够了

19. 读接口并发量高怎么优化（限流，缓存，数据库查询优化）

解析：

一般是和项目关联，结合项目回答即可。

参考答案：

当接口的并发量高时，确实需要通过多种策略来优化性能。以下是一些针对限流、缓存和数据库查询优化的建议：

限流

限流是控制接口请求速率的一种手段，目的是防止因过多的请求而导致的系统崩溃或资源耗尽。

1. **令牌桶算法或漏桶算法**：使用这两种算法来限制接口的请求速率。令牌桶算法允许突发流量，而漏桶算法则更加平滑。
2. **分布式限流**：对于微服务架构，可以使用Redis等分布式系统来实现全局限流。
3. **API网关限流**：在API网关层进行限流，可以保护后端服务不受过量请求的冲击。

缓存

缓存可以显著提高接口的响应速度，减少对数据库的访问次数。

1. **本地缓存**：使用如Guava Cache、Ehcache等本地缓存库，存储热点数据。
2. **分布式缓存**：对于需要共享缓存的场景，可以使用Redis、Memcached等分布式缓存系统。
3. **缓存击穿与雪崩**：注意处理缓存击穿（当缓存中没有数据，大量请求直接打到数据库）和缓存雪崩（大量缓存同时失效，导致请求全部打到数据库）的问题。

数据库查询优化

数据库查询优化是提升接口性能的关键环节。

1. **索引优化**：确保经常查询的字段都有合适的索引，同时避免全表扫描。

2. **SQL语句优化**：简化复杂的SQL语句，避免在SQL中使用子查询和复杂的连接操作。
3. **读写分离读写**：使用主从复制等技术，将读操作和写操作分离到不同的数据库服务器上，提高并发性能。
4. **批量操作**：尽量减少数据库的操作次数，可以将多次操作合并为一次批量操作。
5. **数据库连接池**：使用数据库连接池可以减少数据库连接的创建和销毁开销。

其他优化策略

除了上述提到的限流、缓存和数据库查询优化外，还可以考虑以下策略：

1. **异步处理**：对于非实时性要求较高的请求，可以使用异步处理的方式，减少主线程的阻塞时间。
2. **负载均衡**：通过负载均衡技术，将请求分发到多个服务器上，提高系统的处理能力。
3. **服务拆分**：对于复杂的业务逻辑，可以将其拆分为多个微服务，降低单个服务的复杂度。
4. **监控与告警**：建立完善的监控与告警系统，及时发现并解决性能瓶颈。

学习指导：高并发读写优化方案

20. 写接口并发量高怎么优化（异步+批量）

解析：

一般是和项目关联，结合项目回答即可。

参考答案：

写接口并发量高时，可以采取以下优化措施：

1. **异步处理：** 将写接口的处理逻辑进行异步化，使用消息队列等技术将请求放入队列中，然后异步处理队列中的请求。这样可以减少请求的排队等待时间，提高接口的并发处理能力。
2. **单线程变多线程**写、批量写操作
3. **批量处理,合并写请求：** 将多个写操作合并成批量操作，减少单个写操作的次数，从而减少对数据库的频繁访问。通过批量操作可以提高数据库的吞吐量，减少写入操作的响应时间。

学习指导：高并发读写优化方案

更多面经直通车

原贴连接

学习指导：mysql 存储过程中的 prepare语句

17. mysql唯一索引和主键索引区别 唯一索引可以空吗 主键索引可以空吗

解析：

MySQL 常考题，必会题。

参考答案：

唯一索引和主键索引在MySQL中有一些区别，包括是否允许为空和是否可以有重复值。

1. 唯一索引（Unique Index）：
   - 唯一索引允许为空值，即可以在索引列中存储NULL值。
   - 唯一索引可以用来保证数据表中某列的值是唯一的。
2. 主键索引（Primary Key Index）：
   - 主键索引不允许为空值，即主键列不能为空。

- 主键索引要求每个索引值都是唯一的。
- 主键索引用于唯一标识数据表中的每一行数据，每个数据表只能有一个主键。

学习指导：sql：主键和唯一索引区别

18. redis的pipeline

解析：

Redis 的中等难度题，常考题。

参考答案：

Redis的pipeline是一种在客户端与服务器之间进行批量命令传输和响应的技术。它允许客户端一次性发送多个命令给服务器，而不需要等待每个命令的响应。服务器收到这些命令后，会将它们按顺序执行，并将所有命令的响应一次性返回给客户端，从而减少了通信的往返次数，提高了性能和吞吐量。Pipeline对于需要批量处理的场景非常有用，例如批量写入、批量读取等。

学习指导：Redis Pipeline这一篇就够了

19. 读接口并发量高怎么优化（限流，缓存，数据库查询优化）

解析：

一般是和项目关联，结合项目回答即可。

参考答案：

当接口的并发量高时，确实需要通过多种策略来优化性能。以下是一些针对限流、缓存和数据库查询优化的建议：

限流

限流是控制接口请求速率的一种手段，目的是防止因过多的请求而导致的系统崩溃或资源耗尽。

1. **令牌桶算法或漏桶算法**：使用这两种算法来限制接口的请求速率。令牌桶算法允许突发流量，而漏桶算法则更加平滑。
2. **分布式限流**：对于微服务架构，可以使用Redis等分布式系统来实现全局限流。
3. **API网关限流**：在API网关层进行限流，可以保护后端服务不受过量请求的冲击。

缓存

缓存可以显著提高接口的响应速度，减少对数据库的访问次数。

1. **本地缓存**：使用如Guava Cache、Ehcache等本地缓存库，存储热点数据。
2. **分布式缓存**：对于需要共享缓存的场景，可以使用Redis、Memcached等分布式缓存系统。
3. **缓存击穿与雪崩**：注意处理缓存击穿（当缓存中没有数据，大量请求直接打到数据库）和缓存雪崩（大量缓存同时失效，导致请求全部打到数据库）的问题。

数据库查询优化

数据库查询优化是提升接口性能的关键环节。

1. **索引优化**：确保经常查询的字段都有合适的索引，同时避免全表扫描。
2. **SQL语句优化**：简化复杂的SQL语句，避免在SQL中使用子查询和复杂的连接操作。
3. **读写分离读写**：使用主从复制等技术，将读操作和写操作分离到不同的数据库服务器上，提高并发性能。
4. **批量操作**：尽量减少数据库的操作次数，可以将多次操作合并为一次批量操作。
5. **数据库连接池**：使用数据库连接池可以减少数据库连接的创建和销毁开销。

> **其他优化策略**
>
> 除了上述提到的限流、缓存和数据库查询优化外，还可以考虑以下策略：
>
> 1. **异步处理**：对于非实时性要求较高的请求，可以使用异步处理的方式，减少主线程的阻塞时间。
> 2. **负载均衡**：通过负载均衡技术，将请求分发到多个服务器上，提高系统的处理能力。
> 3. **服务拆分**：对于复杂的业务逻辑，可以将其拆分为多个微服务，降低单个服务的复杂度。
> 4. **监控与告警**：建立完善的监控与告警系统，及时发现并解决性能瓶颈。

学习指导：高并发读写优化方案

20. 写接口并发量高怎么优化（异步+批量）

解析：

一般是和项目关联，结合项目回答即可。

参考答案：

> 写接口并发量高时，可以采取以下优化措施：
>
> 1. **异步处理：** 将写接口的处理逻辑进行异步化，使用消息队列等技术将请求放入队列中，然后异步处理队列中的请求。这样可以减少请求的排队等待时间，提高接口的并发处理能力。
> 2. **单线程变多线程**写、批量写操作
> 3. **批量处理,合并写请求：** 将多个写操作合并成批量操作，减少单个写操作的次数，从而减少对数据库的频繁访问。通过批量操作可以提高数据库的吞吐量，减少写入操作的响应时间。

学习指导：高并发读写优化方案

> 更多面经直通车
>
> 原贴连接

11. 小米Java开发，太难了|0312

> 原贴连接

> 小米Java开发实习面经
>
> 1.JVM的架构，具体阐述一下各个部分的功能?
> 2.Zset的底层如何实现
> 3.Mysql隔离机制有哪些？怎么实现的？可串行化是怎么避免的三个事务问题?
> 4.Spring源码看过吗？Spring的三级缓存知道吗?
> 5.抛开Spring，讲讲反射和动态代理？那三种代理模式怎么实现的?
> 6.讲讲线程池？为什么用线程池?
> 7.集合里面的arraylist和linkedlist的区别是什么？有何优缺点
> 8.介绍一下计网里面的tcp和udp协议
> 9.介绍一下http和https的区别？为什么https安全?
> 10.Mysql有很大的数据量怎么办？怎么分表分库?
> 11.Redis的基本数据类型？Redis的持久化呢？有何优缺点?
> 12.B+树了解吗?底层呢？为什么这么用?
>
> 算法：链表对折
> 1-2-3-4-5-6-7对折之后为1-7-2-6-3-5-4 (需要自己定义链表结构，自己导入包和

1. JVM的架构，具体阐述一下各个部分的功能?

解析:

考察面试者对JVM有没有整体理解，一般在简历中写了相关技能，面试管会问

参考答案:

> JVM（Java Virtual Machine，Java虚拟机）是Java程序运行的环境，它负责将Java字节码转换成特定机器上的机器码并执行。JVM的架构主要由以下几个部分组成，每个部分都有其特定的功能：
>
> 1. 类加载子系统：负责加载类的信息到JVM中。当Java程序需要使用某个类时，类加载子系统负责找到对应的.class文件，并将其加载到JVM的方法区中。这个子系统确保了类的正确性和安全性，同时实现了类的动态加载。
> 2. 方法区：存放已加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。方法区是线程共享的内存区域，它在JVM启动的时候被创建，并且随着类的加载而动态扩展。方法区中包含运行时常量池，用于存放编译期生成的各种字面量和符号引用。
> 3. Java堆：Java堆是JVM所管理的最大一块内存区域，它是所有线程共享的内存区域。几乎所有的对象实例都在这里分配内存。Java堆是垃圾收集器管理的主要区域，因此也经常发生垃圾回收操作。
> 4. Java栈：每个Java虚拟机线程都有一个私有的Java栈，与线程同时创建。Java栈中保存着帧信息，每个方法在执行时都会创建一个栈帧，用于存储局部变量表、操作数栈、动态链接和方法出口等信息。
> 5. 程序计数器（PC寄存器）：这是一块较小的内存空间，它可以看作是当前线程所执行的字节码的行号指示器。字节码解释器工作时就是通过改变这个计数器的值来选取下一条需要执行的字节码指令。由于Java虚拟机的多线程是通过线程轮流切换并分配处理器执行时间的方式来实现的，在任何一个确定的时刻，一个处理器（对于多核处理器来说是一个内核）都只会执行一条线程中的指令。因此，为了线程切换后能恢复到正确的执行位置，每条线程都需要

有一个独立的程序计数器，各条线程之间的计数器互不影响，独立存储，我们称这类内存区域为“线程私有”的内存。

6. 本地方法栈：与虚拟机栈所发挥的作用非常相似，其区别不过是虚拟机栈为虚拟机执行Java方法（也就是字节码）服务，而本地方法栈则为虚拟机使用到的Native方法服务。

此外，JVM还包含直接内存，这是Java堆外的、直接向系统申请的内存空间。直接内存的访问速度通常优于Java堆，因此在一些读写频繁的场合可能会考虑使用直接内存。垃圾回收器可以对方法区、Java堆和直接内存进行回收。

JVM的执行引擎负责执行虚拟机的字节码，虚拟机会使用即时编译技术将方法编译成机器码后再执行。这样，JVM就能提供一个安全、稳定、高效的运行环境，使得Java程序能够跨平台运行。

学习指引：JVM组成结构以及各部分的功能详解

> **面经专栏直通车**

> **面经专栏下载**

2. Zset的底层如何实现

解析：

Redis 经典八股文之一，属于常考题。

参考答案：

Redis 的 Zset（有序集合）类型的底层实现会根据实际情况选择使用压缩列表（ziplist）或者跳跃表（skiplist）。Redis 会根据实际情况动态地在这两种底层结构之间切换，以在内存使用和性能之间找到一个平衡。

这主要取决于两个配置参数：`zset-max-ziplist-entries` 和 `zset-max-ziplist-value`。

1. **使用压缩列表**：当 Zset 存储的元素数量小于 `zset-max-ziplist-entries` 的值，且所有元素的最大长度小于 `zset-max-ziplist-value` 的值时，Redis 会选择使用压缩列表作为底层实现。压缩列表占用的内存较少，但是在需要修改数据时，可能需要对整个压缩列表进行重写，性能较低。

压缩列表是一种为节省内存而设计的特殊编码结构，它将所有的元素和分数紧凑地存储在一起。这种方式的优点是占用内存少，但是在需要修改数据时，可能需要对整个压缩列表进行重写，性能较低。当 Zset 存储的元素数量较少，且元素的字符串长度较短时，Redis 会选择使用压缩列表作为底层实现。

2. **使用跳跃表**：当 Zset 存储的元素数量超过 `zset-max-ziplist-entries` 的值，或者任何元素的长度超过 `zset-max-ziplist-value` 的值时，Redis 会将底层结构从压缩列表转换为跳跃表。跳跃表的查找和修改数据的性能较高，但是占用的内存也较多。

跳跃表是一种可以进行快速查找的有序数据结构，它通过维护多级索引来实现快速查找。这种方式的优点是查找和修改数据的性能较高，但是占用的内存也较多。当 Zset 存储的元素数量较多，或者元素的字符串长度较长时，Redis 会选择使用跳跃表作为底层实现。

跳跃表（skiplist）是一种可以进行快速查找的有序数据结构，它通过维护多级索引来实现快速查找。

这两个参数都可以在 Redis 的配置文件中进行设置。通过调整这两个参数，你可以根据自己的应用特性，选择更倾向于节省内存，还是更倾向于提高性能。

学习指引：Redis中zset的底层实现原理

3. Mysql隔离机制有哪些？怎么实现的？可串行化是怎么避免的三个事务问题？

解析：

MySQL 常见八股文，考察数据库隔离机制及实现原理。

参考答案：

> MySQL的隔离机制主要通过事务的隔离级别来实现。事务的隔离级别定义了事务之间如何相互隔离，以及如何影响其他事务。MySQL提供了四种事务隔离级别：READ UNCOMMITTED、READ COMMITTED、REPEATABLE READ和SERIALIZABLE。
>
> 1. **READ UNCOMMITTED（读未提交）**
>
> 这是最低的隔离级别。在这个级别下，一个事务可以读取另一个未提交事务的修改。这可能导致脏读、不可重复读和幻读。
> 实现：无需额外的机制，直接读取数据即可。
>
> 2. **READ COMMITTED（读已提交）**
>
> 大多数数据库系统的默认隔离级别（但不是MySQL的默认级别）。它只能读取已经提交的数据。这可以防止脏读，但仍然可能导致不可重复读和幻读。
> 实现：当事务进行时，它读取的数据行会被加上锁，其他事务不能修改这些数据行，直到当前事务提交或回滚。
>
> 3. **REPEATABLE READ（可重复读）**
>
> MySQL的默认隔离级别。它确保了在同一个事务中多次读取同样记录的结果是一致的。这可以防止脏读和不可重复读，但仍然可能导致幻读。
> 实现：MySQL使用多版本并发控制（MVCC）来实现这个隔离级别。每个事务都看到一个一致的数据快照，即使其他事务正在修改数据。
>
> 4. **SERIALIZABLE（可串行化）**
>
> 这是最高的隔离级别。它通过强制事务串行执行，从而避免脏读、不可重复读和幻读。但是，这可能会大大降低并发性能。
> 实现：在SERIALIZABLE级别下，每个读写事务都会获得一个唯一的锁，这确保了事务的串行执行。MySQL通过表锁或行锁来实现这一点，具体取决于存储引擎。
>
> **可串行化是如何避免三个事务问题的？**
>
> 三个事务问题通常指的是：脏读、不可重复读和幻读。
>
> 1. **脏读**：一个事务读取了另一个未提交事务的修改。在SERIALIZABLE隔离级别下，由于事务是串行执行的，未提交的事务的修改不会被其他事务读取，因此避免了脏读。
> 2. **不可重复读**：在同一个事务中，多次读取同一数据返回的结果有所不同。SERIALIZABLE级别通过强制事务串行执行，确保在事务期间数据的一致性，从而避免了不可重复读。
> 3. **幻读**：一个事务在执行两次相同的查询，但由于另一个并发事务的插入或删除操作，导致第二次查询返回了不同的结果集。在SERIALIZABLE级别下，由于每个读写事务都会获得一个唯一的锁，这确保了其他事务不能插入或删除数据，从而避免了幻读。
>
> 需要注意的是，虽然SERIALIZABLE级别可以完全避免这三个问题，但它也可能导致性能下降，因为它限制了并发性。在实际应用中，需要根据具体的应用需求和性能要求来选择合适的隔离级别。

学习指引：MySQL事务隔离级别和实现原理

4. Spring源码看过吗？Spring的三级缓存知道吗？

解析：

Spring 原理老八股问，常考。

参考答案：

> Spring框架中确实存在三级缓存机制，它主要用于解决循环依赖问题。具体来说，这三级缓存分别是：
>
> 1. **singletonObjects**：一级缓存，通常被理解为单例池，用于存放已经完全初始化的单例Bean实例。这些实例在缓存中是以键值对的形式存在的，键为Bean的名字，值为Bean实例。
> 2. **earlySingletonObjects**：二级缓存，用于存放已经创建但还未完成初始化的单例Bean实例。这些Bean实例通常是因为依赖其他Bean实例而无法完成初始化，处于不完整状态。
> 3. **singletonFactories**：三级缓存，用于存放Bean实例的工厂对象。这些工厂对象可以用来创建单例Bean实例，并在需要时提前暴露Bean，以便解决循环依赖问题。
>
> 需要注意的是，只有单例的Bean会通过三级缓存提前暴露来解决循环依赖的问题。非单例的Bean每次使用都会从容器中创建新对象，因此不会将其放到三级缓存中。此外，Spring之所以设计二三级缓存而不仅仅是二级缓存，主要是为了解决循环依赖对象需要提前被AOP代理的问题，以及在没有循环依赖的情况下，早期的Bean也不会真正暴露，避免不必要的代理过程。
>
> 总的来说，Spring的三级缓存机制是一个复杂但高效的设计，它允许Spring在创建和初始化Bean时处理各种复杂情况，包括循环依赖和AOP代理等。

学习指引：彻底搞懂Spring之三级缓存解决循环依赖问题

5. 抛开Spring，讲讲反射和动态代理？那三种代理模式怎么实现的？

解析：

Spring 原理八股，属于常考题

参考答案：

> 当谈到反射和动态代理时，通常是指在运行时动态地创建和操作类的实例，而不是在编译时静态地绑定类和方法。这些概念在Java编程中经常用于实现框架和库，以及在运行时实现各种功能的灵活性和可扩展性。
>
> **反射（Reflection）**：
>
> 反射允许在运行时检查、获取并操作类的属性、方法和构造函数，以及访问和修改类的成员变量。Java反射API提供了一组类和接口，例如`Class`、`Method`、`Field`等，可以用于实现这些功能。
>
> 反射的主要应用包括：
>
> 1. 在运行时创建类的实例。
> 2. 动态地调用类的方法。
> 3. 获取和修改类的成员变量。
> 4. 通过反射读取和操作注解信息。
>
> **动态代理（Dynamic Proxy）**：

> 动态代理是一种在运行时动态生成代理类的机制，代理类负责调用目标类的方法并在必要时执行额外的逻辑。Java中的动态代理通常通过`java.lang.reflect.Proxy`类实现，它要求目标类必须实现至少一个接口。动态代理的主要应用包括：
>
> 1. 实现AOP（面向切面编程）。
> 2. 实现远程方法调用（RMI）。
> 3. 实现虚拟代理、延迟加载等模式。
>
> **三种代理模式的实现方式：**
>
> 1. **静态代理（Static Proxy）**：
>
> 静态代理是通过手动编写代理类来实现的，在编译时就已经确定了代理类和目标类之间的关系。代理类通常实现与目标类相同的接口，并在方法中调用目标类的方法，并可以在方法前后添加额外的逻辑。
>
> 2. **JDK动态代理**：
>
> JDK动态代理是基于接口的代理，在运行时通过`java.lang.reflect.Proxy`类动态生成代理类。这种代理方式要求目标类必须实现至少一个接口，代理类在调用目标类方法时会通过`InvocationHandler`接口来处理方法的调用。
>
> 3. **CGLIB动态代理**：
>
> CGLIB动态代理是基于类的代理，它通过继承目标类并重写其方法来实现代理。与JDK动态代理不同，CGLIB动态代理可以代理没有实现接口的类，它使用`Enhancer`类生成代理类，并通过`MethodInterceptor`接口来处理方法的调用。

学习指引: 深入理解 Java 反射和动态代理

6. 讲讲线程池？为什么用线程池?

解析：

操作系统八股文，常考提，难度适中。

参考答案

> 线程池是一种多线程处理形式，处理过程中将任务添加到队列，然后在创建线程后自动启动这些任务。具体来说，线程池在系统中开辟一块区域，存放一些待命的线程，这些线程会等待任务的到来。一旦有任务需要执行，线程池会从这些待命的线程中选取一个来执行任务，任务执行完毕后，线程会返回线程池等待下一次的任务分配。
>
> 使用线程池的主要原因包括：
>
> 1. **资源重用**：通过重复利用已创建的线程，线程池显著降低了线程创建和销毁所带来的资源消耗，包括内存和CPU时间。这有助于减少系统的开销，并提高整体性能。
> 2. **提高响应速度**：由于线程已经预先存在，当任务到达时，它们可以立即开始执行，无需等待线程的创建过程。这大大提高了系统的响应速度，特别是在高并发场景下。
> 3. **控制并发数**：线程池可以有效地控制并发执行的任务数量，防止系统资源耗尽或响应速度下降。通过设置线程池的核心线程数、最大线程数以及队列大小等参数，可以根据系统的承载能力来限制并发级别，提高系统资源的利用率。
> 4. **提高线程的可管理性**：线程是稀缺资源，无限制的创建不仅会消耗系统资源，还可能降低系统的稳定性。线程池提供了统一的线程分配、调优和监控机制，使得线程的管理更加便捷和高效。
> 5. **提供灵活的线程调度策略**：线程池通常支持多种线程调度策略，如优先级队列、定时调度、阻塞队列等，这些策略可以根据任务的特性来合理分配线程资源，优化程序运行效率。

6. **异常处理与监控**：线程池能够更好地统一管理和处理线程的异常情况，避免因为单个线程的异常而导致整个应用程序崩溃。此外，线程池还提供了监控机制，可以实时查看线程的状态和性能数据，便于进行性能分析和调优。

线程池通过优化线程的使用和管理，提高了系统的性能和稳定性，特别是在处理大量并发任务时表现出色。因此，在需要频繁创建和销毁线程的场景中，使用线程池是一种非常有效的解决方案。

学习指引：【详解】为什么使用线程池？线程池的实现原理是什么？

7. 集合里面的arraylist和linkedlist的区别是什么？有何优缺点

解析：

JAVA 基础考察，难度简单

参考答案：

`ArrayList` 和 `LinkedList` 是 Java 集合框架中两种最常用的列表实现。它们的主要区别在于数据的内部存储和访问方式，因此它们在性能特性上有所不同。

1. **底层数据结构**：

`ArrayList`：基于数组实现。它在内存中维护一个连续的空间来存储元素。

`LinkedList`：基于链表实现。它使用双向链表存储元素，每个元素都包含指向其前一个和后一个元素的引用。

2. **访问元素**：

`ArrayList`：通过索引访问元素非常快，因为可以直接计算元素在内存中的位置。时间复杂度为 O(1)。

`LinkedList`：通过索引访问元素相对较慢，因为需要从头或尾开始遍历链表直到找到所需元素。时间复杂度为 O(n)。

3. **插入和删除元素**：
   `ArrayList`：在列表的开头或中间插入或删除元素时，可能需要移动大量元素以保持数组的连续性。因此，这些操作相对较慢，特别是在列表很大时。
   `LinkedList`：在列表的任何位置插入或删除元素都相对较快，因为只需要更新相邻元素的引用。
4. **内存消耗**：

`ArrayList`：由于数组需要预留连续的内存空间，所以可能存在一定的内存浪费。但是，由于它的内存布局更紧凑，所以在存储相同数量的元素时，`ArrayList` 通常比 `LinkedList` 占用更少的内存。 `LinkedList`：由于每个元素都需要存储指向其前一个和后一个元素的引用，所以链表实现通常会有更高的内存开销。

5. **用途**：

当你需要频繁地通过索引访问元素，且不需要经常插入或删除元素时，`ArrayList` 是一个好选择。

当你需要在列表的任何位置频繁地插入或删除元素时，`LinkedList` 更为合适。

总结优缺点：

ArrayList：

- **优点**：通过索引访问元素非常快，内存占用相对较小（相对于 LinkedList）。
- **缺点**：在列表开头或中间插入、删除元素时性能较差，因为可能需要移动大量元素。

LinkedList：

- **优点**：在列表的任何位置插入或删除元素都很快，不需要移动其他元素。
- **缺点**：通过索引访问元素较慢，因为需要从头或尾开始遍历链表。内存占用相对较大，因为每个元素都需要存储额外的引用。

学习指引：ArrayList和LinkedList的区别以及优缺点

8. 介绍一下计网里面的tcp和udp协议

解析：

考察计算机网络基础知识，属于基础题，需要注意的是要答全要点。

参考答案：

TCP（传输控制协议）和UDP（用户数据报协议）是计算机网络中用于处理传输数据包的两种重要协议，它们各自具有不同的特点和使用场景。

TCP协议是一种面向连接的、可靠的、基于字节流的传输层通信协议。在通信之前，TCP会在发送方和接收方之间建立连接，确保数据传输的可靠性。TCP通过确认机制、超时重传、流量控制以及拥塞控制等手段，确保数据能够无差错、不丢失、不重复、按序到达。TCP协议提供全双工通信，允许通信双方的应用进程在任何时候都可以发送数据。然而，由于TCP需要建立连接并进行一系列的检查与修改，其开销相对较大，实时性也不如UDP。

相比之下，UDP协议则是一种无连接、不可靠的传输层协议。UDP在发送数据前不建立连接，也不对数据报进行检查与修改，因此其传输效率较高，实时性较好。但是，由于UDP不保证数据的可靠性，可能会出现分组丢失、重复、乱序等问题，需要应用程序自行处理这些问题。UDP协议适用于对实时性要求较高，但对数据可靠性要求不高的场景，如视频流传输、实时语音通信等。

TCP和UDP协议各有优劣，应根据具体的应用场景和需求来选择合适的协议。对于需要保证数据可靠性的场景，如文件传输、电子邮件等，通常选择TCP协议；而对于实时性要求较高，但对数据可靠性要求不高的场景，则可以选择UDP协议。

学习指引：深入理解网络通讯和TCP、IP协议

9. 介绍一下http和https的区别？为什么https安全？

解析：

计算机网络基础知识，属于简单八股文，需要注意的是要答全要点。

参考答案：

HTTP和HTTPS的主要区别体现在安全性和数据传输方式上。

首先，HTTP是超文本传输协议，它采用明文方式传输数据，这意味着数据在传输过程中是不加密的，可能会被第三方截获或篡改。相比之下，HTTPS则是具有安全性的SSL加密传输协议，它通过在HTTP协议的基础上添加SSL/TLS加密层，实现了数据的加密传输。因此，HTTPS可以防止中间人攻击和数据窃听，使得数据在传输过程中更加安全。

其次，HTTPS需要到CA申请证书，一般情况下需要一定费用，而HTTP则无需此步骤。此外，HTTP和HTTPS使用的端口号也不同，HTTP的端口号是80，而HTTPS的端口号是443。

> HTTPS之所以安全，主要得益于其采用的加密技术和身份验证机制。SSL/TLS加密层对传输的数据进行加密处理，确保数据的完整性和机密性。同时，HTTPS还包含身份验证环节，通过验证服务器端的证书，确保客户端与正确的服务器进行通信，防止了中间人攻击和服务器劫持等安全问题。

学习指引：只看这一篇就能搞懂Http和Https协议区别

10. Mysql有很大的数据量怎么办？怎么分表分库?

解析：

MySQL 大数据处理，数据一定难度的题，也是常考题。

参考答案：

> 当MySQL数据库中的数据量变得非常大时，性能可能会受到影响，查询速度变慢，甚至可能导致系统崩溃。为了解决这个问题，我们可以采用分表分库的策略。分表分库是将一个大的数据库或表拆分成多个较小的、更易于管理的部分，以提高性能、可靠性和可维护性。
>
> 以下是一些关于如何分表分库的建议：
>
> **1. 垂直拆分**
>
> **垂直分库**：按照业务将不同的表分到不同的数据库上，每个数据库都包含各自独立的业务数据。
>
> **垂直分表**：将一个大表中的某些列拆分到另一个表中，通常是按照列的属性进行拆分。例如，将经常访问的列和不经常访问的列分开。
>
> **2. 水平拆分**
>
> **水平分库**（也叫分区）：按某个字段的某种规则，将同一个表中的数据拆分到多个数据库（服务器）上，每个库可以放在不同的服务器上。
>
> **水平分表**（也叫分片）：将同一个表中的记录拆分到多个结构相同的表，每个表只包含一部分数据。常见的分片键有用户ID、订单ID等。
>
> **3. 分片策略**
>
> *范围分片**：按某个范围来分片，比如按时间范围或ID范围。
>
> **哈希分片**：对某个字段进行哈希计算，然后按照哈希值的大小或范围进行分片。
>
> **目录分片**：创建路由表或目录，路由表中保存了数据记录的存放位置信息，系统执行查询操作时，根据路由表的地址信息去相应的分片中查询。
>
> **4. 注意事项**
>
> - **事务处理**：分表分库后，跨表或跨库的事务处理变得更加复杂，需要考虑分布式事务的解决方案。
> - **数据迁移与备份**：随着数据的增长，可能需要定期迁移数据或进行备份。确保有有效的数据迁移和备份策略。
> - **中间件**：使用数据库中间件（如MyCAT、Sharding-JDBC等）可以简化分表分库的操作，隐藏底层的复杂性。
> - **应用层调整**：分表分库后，应用层需要进行相应的调整，以支持新的数据结构和查询方式。
> - **测试与监控**：在实施分表分库之前和之后，都需要进行充分的测试和监控，以确保系统的稳定性和性能。
>
> 总之，分表分库是一个复杂的过程，需要根据具体的业务场景和需求来制定合适的策略。在实施之前，建议进行充分的调研和测试，以确保系统的稳定性和性能。

学习指引 MySQL数据库中，数据量越来越大，有什么具体的优化方案么？

11. Redis的基本数据类型？Redis的持久化呢？有何优缺点?

解析：

Redis 基础知识，简单的八股文，需要掌握好。

参考答案：

> Redis是一个开源的使用ANSI C语言编写的、支持网络、可基于内存亦可持久化的日志型、Key-Value数据库，并提供多种语言的API。以下是关于Redis的基本数据类型和持久化的详细解答：
>
> **一、Redis的基本数据类型**
>
> Redis支持五种基本数据类型：
>
> 1. **String（字符串）**：这是Redis最基本的数据类型，可以理解为与Memcached中的类型一致，即一个key对应一个value。String类型是二进制安全的，意味着它可以包含任何数据，如jpg图片或序列化的对象。在Redis中，字符串的value最大可以达到512M。
> 2. **Hash（哈希）**：Redis的Hash是一个键值对集合，实际上是一个String类型的field和value的映射表。Hash特别适合用于存储对象。
> 3. **List（列表）**：列表是用来存储多个有序的字符串，一个列表最多可以存储2^32-1个元素。
> 4. **Set（集合）**：Redis的Set是String类型的无序集合，它是通过HashTable实现的。
> 5. **Zset（有序集合）**：Redis的Zset和Set一样都是String类型元素的集合，且不允许重复的成员。但与Set不同的是，Zset的每个元素都会关联一个double类型的分数，Redis正是通过分数来为集合中的元素从小到大进行从小到大的排序。
>
> 此外，Redis还提供了其他三种特殊的数据结构类型，包括Geospatial（地理位置）、Hyperloglog（基数统计）和Bitmap（位图）。
>
> **二、Redis的持久化**
>
> Redis提供了两种主要的持久化方法：RDB（Redis DataBase）和AOF（Append Only File）。
>
> 1. **RDB**：
>
> - 原理：通过创建子进程，将数据快照写入一个临时文件，然后替换之前的文件达到持久化。主进程在持久化过程中不进行IO操作，保证了持久化过程的高性能。
> - 优点：节省磁盘空间，恢复速度快。
> - 缺点：如果突然宕机，可能会丢失最后一次备份后的数据。此外，由于是利用拷贝技术，如果数据量较大，持久化过程可能会消耗较多性能。
>
> 2. **AOF**：
>
> - 原理：通过记录每次写操作命令，并在启动时重新执行这些命令来恢复数据。
> - 优点：提供了更稳健的数据恢复机制，丢失数据少，生成的日志可读，便于处理误操作。
> - 缺点：相比RDB，AOF占用了更多的内存空间，恢复备份的速度较慢。如果每次读写都同步的话，会有一定的性能压力。
>
> 在选择持久化方式时，需要根据实际的应用场景和需求进行权衡。例如，如果更关心数据的完整性和可靠性，可以选择AOF；如果更看重性能和磁盘空间的利用率，可以选择RDB。当然，也可以结合使用两种持久化方式，以达到最佳的效果。

学习指引： redis的五种数据类型、redis持久化机制

12. B+树了解吗?底层呢？为什么这么用?

解析：

数据库底层原理，常考题，要掌握。

参考答案：

> B+树是一种多路搜索树，其定义基本与B-树相同，但在某些方面进行了优化。B+树的主要特点包括：
>
> 1. 所有的叶子结点中包含了全部关键字的信息，以及指向包含这些关键字记录的指针，且叶子结点本身依关键字的大小自小而大顺序链接。
> 2. 所有的非终端结点可以看成索引部分，结点中仅含有其子树（根结点）中最大（或最小）关键字。
> 3. 非叶子节点的关键字数目等于它的分支数量。
>
> 在B+树中，数据存放的更加紧密，具有更好的空间局部性，因此访问叶子节点上关联的数据具有更好的缓存命中率。同时，由于叶子结点都是相链的，对整棵树的遍历只需要一次线性遍历叶子结点即可，且由于数据顺序排列并且相连，所以便于区间查找和搜索。
>
> B+树的底层实现主要是基于多路平衡查找树。在查询过程中，从根节点出发，查找到叶子节点方可获得所查键值，然后根据查询判断是否需要回表查询数据。这种结构使得B+树在访问数据时具有更高的效率。
>
> B+树在数据库和许多其他数据结构中得到广泛应用的原因主要有以下几点：
>
> 1. I/O次数更少：由于B+树的层数比其他树（如二叉搜索树）小，因此读取节点并进行I/O操作的次数也会减少。同时，B+树中间节点不存储数据，只存储索引，使得相同大小的磁盘页可以容纳更多的节点元素，进一步减少了I/O次数。
> 2. 查询更加稳定：B+树每次查询都必须访问到叶子节点，而B-树可能在中间节点或叶子节点找到匹配元素，因此B+树的查询性能更为稳定。
> 3. 更利于查询范围：由于B+树的叶子节点首尾相连，因此在进行范围查询时，只需要在链表上进行遍历，效率非常高。

学习指引：面试官：MySQL索引为什么要用B+树实现？

12. 【腾讯云】CSIG一面，凉凉凉|0313

> 仓促准备下的腾讯云一面，40+min，分享一下，可能有一两个问题记漏了
>
> 上来先是道算法题，最小不重复子串，把题理解错了没写出来
>
> 挑了一个项目问，为什么选择标注+模型，遇到的问题是什么，有没有想过怎么提升模型表现，感觉面试官对这一块也不是特别熟悉，没有深究
>
> 由于简历和岗位不太匹配，之后基本都是八股：
>
> 1. 介绍下TCP UDP的区别
> 2. TCP怎么保证他的连接可靠
> 3. 介绍一下四次挥手
> 4. TIME_WAIT出现在TCP连接的什么时候
> 5. 介绍一下HTTP连接的流程
> 6. HTTP有长连接吗，怎么实现的
> 7. TCP长短连接流程，怎么实现
> 8. 介绍一下线程
> 9. 线程的调度怎么完成
> 10. 线程在什么时候会阻塞
> 11. 介绍一下c++的内存分配策略
> 12. c++从编译到运行发生了什么
> 13. c++的动态链接和静态链接

14. 大小端字节序
15. 网络字节序是大端还是小端，本地字节序是大端还是小端
16. MySQL什么时候会用索引
17. Redis会吗（俩数据库的都不会，直接尬住）

最后问面试官有哪些地方可以改进，面试官尬笑一下说算法题有点可惜
原贴连接

1. 介绍下TCP UDP的区别

解析：

计算机网络基础知识，简单题，必考题。

参考答案：

TCP（传输控制协议）和UDP（用户数据报协议）是计算机网络中传输层的两个主要协议，它们各自具有不同的特点和适用场景。以下是TCP和UDP之间的主要区别：

1. 连接特性：

- TCP是面向连接的协议。在传输数据之前，需要通过三次握手来建立连接，并在数据传输完成后通过四次挥手来释放连接。这种连接机制确保了数据传输的可靠性和顺序性。
- UDP则是无连接的协议。发送数据前不需要建立连接，直接发送数据包，这使得UDP在传输数据时更加灵活和高效。

2. 可靠性：

- TCP提供可靠的数据传输服务。它使用校验和、重传控制、序号标识、滑动窗口和确认应答等机制来确保数据的无差错、不丢失、不重复且按序到达。
- UDP则提供尽最大努力交付的服务，不保证数据的可靠传输。在数据传输过程中，如果发生丢包或乱序，UDP不会进行重传或顺序控制。

3. 效率与实时性：

- UDP具有较好的实时性和较高的工作效率，因为它不需要建立连接和进行复杂的控制操作。这使得UDP适用于对高速传输和实时性有较高要求的通信或广播通信，如语音、视频、直播等。
- TCP虽然提供了可靠的数据传输，但由于其复杂的控制机制，相对UDP而言效率较低。

4. 通信模式：

- TCP连接只能是点对点的，即一条TCP连接只能连接两个端点。
- UDP则支持一对一、一对多、多对一和多对多的交互通信模式，这使得UDP在广播和多播等场景中更具优势。

5. 首部开销：

- TCP的首部较大，通常为20字节，这增加了每个数据包的开销。
- UDP的首部较小，只有8字节，减少了数据包的开销，提高了传输效率。

TCP和UDP在连接特性、可靠性、效率与实时性、通信模式以及首部开销等方面存在显著差异。在选择使用TCP还是UDP时，需要根据具体的应用场景和需求进行权衡。例如，对于需要可靠数据传输的场景（如文件传输、远程登录等），TCP是更好的选择；而对于实时性要求较高或需要广播和多播的场景（如语音、视频等），UDP则更具优势。

学习指引：TCP和Udp的区别是什么？

面经专栏直通车

> 面经专栏下载

2. TCP怎么保证他的连接可靠

解析：

计算机网络常见题，必考题。

参考答案：

> TCP（传输控制协议）通过一系列机制保证连接的可靠性。以下是TCP确保连接可靠性的主要方式：
>
> 1. **序列号与确认应答**：TCP给每一个发送的数据包都赋予一个序列号，同时要求接收方在收到数据后回复一个确认应答（ACK），告知发送方已成功接收。如果发送方在一定时间内没有收到ACK，它会认为数据包丢失并重传。
> 2. **校验和**：TCP在发送数据前会计算数据的校验和，并在接收端对数据进行校验和比对。如果校验和不匹配，接收端会丢弃该数据包，并要求发送方重传。
> 3. **超时重传**：TCP在发送数据包后会启动一个定时器。如果定时器超时前未收到ACK，发送方会重传该数据包。
> 4. **连接管理**：TCP通过三次握手来建立连接，确保双方都已准备好进行数据传输。在数据传输完成后，通过四次挥手来关闭连接，确保数据已完全传输并释放资源。
> 5. **流量控制**：TCP使用滑动窗口机制来实现流量控制，确保发送方不会发送过多数据导致接收方缓冲区溢出。
> 6. **拥塞控制**：TCP通过慢开始、拥塞避免、快重传和快恢复等算法来避免网络拥塞，确保数据传输的稳定性和可靠性。

学习指引：TCP协议如何保证可靠传输

3. 介绍一下四次挥手

解析：

计算机网络常见题，必考题。

参考答案：

> 四次挥手是TCP（传输控制协议）中用于释放已建立的连接的一个过程。这个过程确保客户端和服务器双方都能正常、有序地关闭连接，并释放相关资源。以下是四次挥手的详细步骤：
>
> 1. **第一次挥手**：客户端（假设为主动关闭的一方）发送一个FIN报文给服务器，用来关闭客户端到服务器的数据传送。此时，客户端进入FIN_WAIT_1状态，等待服务器的确认。
> 2. **第二次挥手**：服务器收到FIN报文后，发送一个ACK报文给客户端，确认收到客户端的关闭请求。此时，服务器进入CLOSE_WAIT状态，客户端收到这个ACK后，进入FIN_WAIT_2状态。
> 3. **第三次挥手**：服务器在完成所有数据传输后，发送一个FIN报文给客户端，表示服务器也准备关闭连接。此时，服务器进入LAST_ACK状态，等待客户端的确认。
> 4. **第四次挥手**：客户端收到服务器的FIN报文后，发送一个ACK报文给服务器，确认收到服务器的关闭请求。此时，服务器进入CLOSE状态，完成连接的关闭。客户端在发送完ACK后，会等待一段时间（通常是2MSL，即两倍的最长报文段寿命），确保服务器已经收到ACK并关闭了连接。然后，客户端进入CLOSE状态，完成四次挥手过程。
>
> 通过四次挥手，TCP协议确保了双方都能正常、有序地关闭连接，避免了资源泄露和网络问题。这也是TCP协议保证连接可靠性的一个重要方面。

学习指引：一文彻底搞懂 TCP三次握手、四次挥手过程及原理

4. TIME_WAIT出现在TCP连接的什么时候

解析：

计算机网络基础知识，简单题，必考题。

参考答案：

> TIME_WAIT状态出现在TCP连接的关闭过程中，特别是在四次挥手的阶段。具体来说，当主动关闭连接的一方（通常是客户端）发送完最后一个ACK报文后，它会进入TIME_WAIT状态。
>
> 在四次挥手的过程中，主动关闭连接的一方首先发送一个FIN报文给对端，表示希望关闭连接。然后，它等待并接收到对端的ACK报文，进入FIN_WAIT_2状态。接着，当对端也准备好关闭连接并发送FIN报文时，主动关闭连接的一方会发送一个ACK报文进行确认，并进入TIME_WAIT状态。
>
> 在TIME_WAIT状态中，主动关闭连接的一方会等待一段时间（通常是2MSL，即两倍的最长报文段寿命），以确保对端收到了ACK报文并关闭了连接。这样做的目的是为了防止已经关闭的连接中的报文段还在网络中滞留，导致出现"迷途"的数据包，影响下一次在相同端口上建立的新连接。

学习指引：TCP中time_wait解释及解决方法

5. 介绍一下HTTP连接的流程

解析：

在计算机网络中，HTTP 属于常考题，必考题。

参考答案：

> 1. 浏览器进行DNS域名解析，得到对应的IP地址
> 2. 根据这个IP，找到对应的服务器建立连接（三次握手）
> 3. 建立TCP连接后发起HTTP请求（一个完整的http请求报文）
> 4. 服务器响应HTTP请求，浏览器得到html代码（服务器如何响应）
> 5. 浏览器解析html代码，并请求html代码中的资源（如js、css、图片等）
> 6. 浏览器对页面进行渲染呈现给用户
> 7. 服务器关闭TCP连接（四次挥手）

学习指引：一次完整的HTTP请求过程是怎么样的呢？

6. HTTP有长连接吗，怎么实现的

解析：

在计算机网络中，HTTP 进阶题，常考题。

参考答案：

> **HTTP有长连接。**HTTP的长连接通常是通过在HTTP请求头中设置`Connection: keep-alive`字段来实现的。这个字段告诉服务器，客户端希望保持与服务器的TCP连接，直到超时或显式地断开连接。
>
> HTTP长连接的实现基于TCP协议。在HTTP/1.1中，长连接是默认启用的，除非显式地在请求头中设置`Connection: close`。在使用长连接的情况下，客户端和服务端建立一次连接之后，可以在这条连接上进行多次请求/响应操作。当客户端发送完一个HTTP请求后，连接并不会立即关闭，而是等待服务器发送响应。服务器在发送完响应后，也不会立即关闭连接，而是保持连接开放，以便客户端可以继续发送新的请求。这样，客户端和服务器之间可以重用已经建立的TCP连接，避免了频繁地建立和关闭连接所带来的开销，提高了网络性能。

> 需要注意的是，长连接并不意味着连接会一直保持打开状态。在实际应用中，服务器可能会根据一些策略（如超时时间、并发连接数等）来主动关闭连接，或者客户端在发送完所有请求后也可以主动关闭连接。此外，HTTP/2协议中采用了更高效的连接管理方式，称为多路复用，它默认使用长连接，并通过更复杂的机制来管理多个请求在同一连接上的并发传输。
>
> HTTP长连接的实现依赖于TCP协议的支持和HTTP协议中相关字段的设置。通过合理使用长连接，可以提高Web应用的性能和用户体验。

学习指引：HTTP长连接实现原理

7. TCP长短连接流程，怎么实现

解析：

参考答案：

> TCP长短连接的流程如下：
>
> **短连接**：
>
> 1. **建立连接**：客户端与服务器之间建立TCP连接。
> 2. **发送消息**：客户端向服务器发送消息或请求。
> 3. **响应回传**：服务器响应回传消息或处理结果给客户端。
> 4. **关闭连接**：完成一次发送与接收（读、写）服务后，客户端和服务器都会关闭连接。
>
> **长连接**：
>
> 1. **建立连接**：客户端与服务器之间建立TCP连接。
> 2. **发送消息**：客户端向服务器发送消息或请求。
> 3. **响应回传**：服务器响应回传消息或处理结果给客户端。
> 4. **多次通信**：一次请求响应结束后，当前连接不关闭，通过当前连接实现多次通信。
> 5. **关闭连接**：客户端或服务器在不再需要连接时，会主动关闭连接。通常，客户端会发起关闭连接的请求，而服务器则依据连接的超时时间（timeout）来关闭连接。
>
> **实现方式**：
>
> TCP长短连接的实现主要依赖于TCP协议本身以及应用程序的设计。在TCP层面，连接的建立、数据的传输和连接的关闭都是通过TCP协议的相关机制来完成的。而在应用层面，如HTTP协议，可以通过在请求头中设置相应的字段（如`Connection: keep-alive`或`Connection: close`）来指示使用长连接还是短连接。
>
> **特点**：
>
> - **短连接**：管理简单，每次请求都需要建立连接和释放连接，因此每次连接只用于一次读写操作。它适合于访问不频繁的操作，如简单的web页面访问、用户注册、数据查询或文件下载等。
> - **长连接**：可以省去较多的TCP建立和关闭的操作，减少浪费，节约时间。它适用于需要频繁通信的场景，如实时数据传输、聊天应用等。
>
> TCP长短连接的实现主要依赖于TCP协议和应用层协议（如HTTP）的设定。根据应用的需求和场景，可以选择使用短连接或长连接来优化性能和资源利用。

学习指引：TCP 长连接 短连接

8. 介绍一下线程

解析：

操作系统必考题，简单难度。

参考答案：

> 线程，被称为轻量级进程（Lightweight Process, LWP），是程序执行流的最小单元，也是CPU调度和分派的基本单位。线程间共享进程的所有资源，每个线程有自己的堆栈和局部变量。使用线程的最大作用是提高程序并行执行的能力，充分利用CPU的利用率。然而，线程并非越多越好，因为过多的线程切换会消耗资源，反而可能导致程序的运行时间增加。
>
> 线程与进程类似，但线程是一个比进程更小的执行单位。一个进程在其执行的过程中可以产生多个线程。与进程不同的是，同类的多个线程共享同一块内存空间和一组系统资源，所以系统在产生一个线程，或是各个线程之间做切换工作时，负担要比进程小得多。
>
> 实现多线程主要有三种方法：继承Thread类、实现Runnable接口和实现Callable接口。线程与进程之间的主要区别在于，进程是资源分配的最小单位，而线程是CPU资源调度的最小单位。
>
> 总的来说，线程是操作系统中进行运算调度的最小单位，它被包含在进程之中，是进程中的实际运作单位。线程自己不拥有系统资源，只拥有一点在运行中必不可少的资源（如程序计数器，一组寄存器和栈），但它可与同属一个进程的其它的线程共享进程所拥有的全部资源。

学习指引：进程？线程？有什么区别？

9. 线程的调度怎么完成

解析：

操作系统必考题，难度中等。

参考答案：

> 线程调度的完成通常涉及以下几个步骤：
>
> 1. 选择调度算法：操作系统可以使用不同的调度算法来决定线程的执行顺序。常见的调度算法包括先来先服务（FCFS）、最短作业优先（SJF）、轮转调度（Round Robin）等。选择适合系统需求的调度算法是很重要的。
> 2. 确定优先级：每个线程通常都有一个优先级，优先级决定了线程被调度的顺序。高优先级的线程会被优先执行，而低优先级的线程则可能被延迟执行。操作系统可以根据线程的类型、重要性等因素来确定线程的优先级。
> 3. 创建和管理线程队列：操作系统会维护一个线程队列，用于存储等待执行的线程。当一个线程需要执行时，它会被添加到队列中。调度器会从队列中选择一个线程，并将其分配给可用的处理器执行。
> 4. 上下文切换：当调度器决定切换到另一个线程时，它会进行上下文切换操作。上下文切换包括保存当前线程的上下文信息（如寄存器状态、程序计数器等），并加载下一个线程的上下文信息，以便继续执行。
> 5. 执行线程：一旦调度器选择了一个线程，并完成了上下文切换，该线程就会开始执行。线程执行的时间片通常是有限的，当时间片用完后，调度器会再次选择下一个线程执行。
>
> 不同的操作系统可能会有不同的实现方式和策略。线程调度的目标是合理分配系统资源，提高系统的性能和响应能力。

学习指引：【操作系统】进程和线程调度

10. 线程在什么时候会阻塞

解析：

操作系统常考题，难度中等。

参考答案：

> 线程在多种情况下可能会进入阻塞状态。以下是一些常见的情况：
>
> 1. **线程休眠**：当线程执行了`Thread.sleep(int n)`方法时，它会放弃CPU的使用权，进入休眠状态，持续n毫秒后恢复运行。在休眠期间，线程处于阻塞状态。
> 2. **等待获取同步锁**：当线程需要执行一段同步代码块，但无法获得相关的同步锁时，它会进入阻塞状态。只有当获得了所需的同步锁后，线程才能继续执行。
> 3. **执行`wait()`方法**：线程在执行一个对象的`wait()`方法时，会进入阻塞状态。此时，线程会释放它持有的对象锁，并等待其他线程执行该对象的`notify()`或`notifyAll()`方法，才能被唤醒并继续执行。
> 4. **等待相关资源**：线程在执行I/O操作（如文件读写、网络通信等）时，可能会因为等待相关资源而进入阻塞状态。例如，在远程通信中，客户端线程可能在等待服务器端响应时阻塞。
> 5. **读写数据时的阻塞**：
>   - 线程在向Socket的输出流写一批数据时，可能会进入阻塞状态，直到所有数据都输出，或者发生异常。
>   - 线程从Socket的输入流读取数据时，如果没有足够的数据，就会进入阻塞状态，直到读取到足够的数据，或者到达输入流的末尾，或者出现异常。
> 6. **其他线程操作**：当一个线程调用了`yield()`方法时，它会将执行权礼让给同等级或更高优先级的线程，此时也可能进入阻塞状态。
> 7. **线程设置延迟关闭**：如果线程在Socket通信中设置了延迟关闭（通过`Socket.setSoLinger()`方法），那么在调用`Socket.close()`方法时，线程会等待底层Socket发送完所有剩余数据，或者超过设定的延迟时间，才返回。这期间，线程也会处于阻塞状态。
>
> 此外，线程的阻塞还可能由其他因素导致，如用户输入、文件上传、加载等操作。这些因素都会导致线程需要等待某个事件或条件成立后才能继续执行，从而进入阻塞状态。

学习指引：[什么是线程阻塞?为什么会出现线程阻塞？](https://www.cnblogs.com/JonaLin/p/11570858.html)

11. 介绍一下c++的内存分配策略

解析：

C++ 原理基础题，必会题。

参考答案：

> C++的内存分配策略主要包括静态内存分配、栈内存分配和堆内存分配。
>
> 1. 静态内存分配：静态内存分配是在程序编译阶段完成的，用于存储全局变量和静态变量。这些变量的内存空间在程序启动时就被分配，并在整个程序的生命周期内保持不变。
> 2. 栈内存分配：栈内存分配是由编译器自动管理的，用于存储局部变量和函数调用的上下文信息。栈内存的分配和释放是自动进行的，遵循"先进后出"的原则。当一个函数被调用时，它的局部变量会被分配到栈上，当函数执行完毕时，这些变量会被自动释放。
> 3. 堆内存分配：堆内存分配是由程序员手动管理的，用于存储动态分配的对象。在堆上分配内存需要使用特定的操作符（如new、malloc等），并在不需要时手动释放内存（使用delete、free等）。堆内存的分配和释放不受函数调用的限制，可以在程序的任意位置进行。
>
> 静态内存分配适用于全局变量和静态变量，栈内存分配适用于局部变量和函数调用，堆内存分配适用于动态分配的对象。

学习指引：C++内存分配的几种策略

12. c++从编译到运行发生了什么

解析：

C++ 原理题，常考题。

参考答案：

> C++程序从编译到运行经历了以下几个步骤：
>
> 1. 预处理（Preprocessing）：在编译之前，预处理器会对源代码进行处理。它会处理以"#"开头的预处理指令，例如#include、#define等，并展开宏定义，删除注释等。预处理后的代码成为编译单元。
> 2. 编译（Compilation）：编译器将预处理后的代码翻译成汇编语言。它会进行词法分析、语法分析、语义分析等过程，生成汇编代码。
> 3. 汇编（Assembly）：汇编器将汇编代码转换成机器码，生成目标文件（通常是二进制文件）。目标文件包含了可执行代码、数据和符号表等信息。
> 4. 链接（Linking）：链接器将目标文件与其他必要的库文件进行链接，生成可执行文件。它会解析符号引用，将目标文件中的符号与其他目标文件或库文件中的符号进行关联，生成最终的可执行文件。
> 5. 加载（Loading）：操作系统将可执行文件加载到内存中，并为程序分配运行所需的资源。加载过程包括分配内存空间、建立程序入口点、加载依赖的动态链接库等。
> 6. 运行（Execution）：程序开始执行，按照指令序列进行操作。它会分配栈空间、执行全局初始化、调用main函数等。程序执行过程中可能会进行输入输出、调用其他函数、分配和释放内存等操作。

学习指引：一个程序从编译到运行的全过程

13. c++的动态链接和静态链接

解析：

C++ 原理题，难度中等。

参考答案：

> C++中的链接可以分为动态链接和静态链接两种方式。
>
> 1. 静态链接（Static Linking）：
>   静态链接是将所有的目标文件和库文件在编译时链接到可执行文件中。在静态链接的情况下，目标文件中使用的所有函数和库函数的代码都会被复制到最终的可执行文件中。这意味着可执行文件独立于系统中的动态链接库，可以在没有其他依赖的情况下运行。静态链接的优点是执行速度快，不受系统环境的影响。但缺点是可执行文件的体积较大，占用磁盘空间较多。
> 2. 动态链接（Dynamic Linking）：
>   动态链接是在程序运行时将目标文件和库文件链接到可执行文件中。在动态链接的情况下，可执行文件只包含对库函数的引用，而不包含实际的库函数代码。当程序运行时，操作系统会在内存中加载所需的动态链接库，并将其与可执行文件进行链接。动态链接的优点是可执行文件的体积较小，节省了磁盘空间。同时，多个程序可以共享同一个动态链接库，减少了内存的占用。然而，动态链接的缺点是在程序运行时需要查找和加载动态链接库，可能会稍微降低程序的执行速度。
>
> 在C++中，可以使用编译器提供的选项来指定链接方式。对于静态链接，可以使用编译器的静态库选项（如"-static"），将库文件静态链接到可执行文件中。对于动态链接，可以使用编译器的动态库选项（如"-shared"）生成动态链接库文件，并在编译时指定动态链接库的路径。

> 选择动态链接还是静态链接取决于具体的需求。静态链接适用于独立的、可移植的可执行文件，而动态链接适用于共享库、模块化的程序设计和资源共享的场景。

学习指引：深入浅出静态链接和动态链接

14. 大小端字节序

解析：

存储相关的常考题，难度中等。

参考答案：

> 大小端字节序（Endianness）是指在多字节数据类型（如整数、浮点数）在内存中存储时，字节的顺序是从高位到低位还是从低位到高位。
>
> 1. 大端字节序（Big Endian）：在大端字节序中，高位字节存储在低地址，低位字节存储在高地址。例如，十六进制数0x12345678在大端字节序中存储为：0x12 0x34 0x56 0x78。
> 2. 小端字节序（Little Endian）：在小端字节序中，低位字节存储在低地址，高位字节存储在高地址。例如，十六进制数0x12345678在小端字节序中存储为：0x78 0x56 0x34 0x12。
>
> 选择使用大端字节序还是小端字节序是由硬件架构和操作系统决定的。x86架构的计算机通常采用小端字节序，而一些网络协议（如TCP/IP）通常使用大端字节序。
>
> 在C++中，可以使用类型转换和位操作来处理大小端字节序的问题。例如，可以使用memcpy函数将字节序进行转换，或者使用位操作来逐个字节读取和写入数据。
>
> 在跨平台开发时，应该考虑字节序的问题，以确保数据在不同平台上的正确解析和传输。

学习指引：字节序（大小端）理解

15. 网络字节序是大端还是小端，本地字节序是大端还是小端

解析：

考的不多，多读几遍有印象即可。

参考答案：

> 网络字节序是大端字节序，即高位字节存储在内存的低地址端，低位字节存储在内存的高地址端。这种字节序的采用主要是为了不同平台之间的兼容性问题，确保数据在网络传输中的一致性和准确性。
>
> 而本地字节序则取决于具体的硬件和操作系统。例如，Intel和AMD的处理器采用的是小端字节序，即低位字节存储在内存的低地址端，高位字节存储在内存的高地址端。然而，也有一些处理器采用大端字节序。因此，在进行网络通信或跨平台编程时，需要注意字节序的转换问题，以确保数据的正确解析和处理。

学习指引：网络字节序为什么使用大端字节序呢？

16. MySQL什么时候会用索引

解析：

MySQL 重点题，常考题，必会题。

参考答案：

> MySQL在以下情况下会使用索引：
>
> 1. **查询条件中的关键字**：当查询条件中使用了某个字段的关键字，且该字段上建立了索引时，MySQL会尝试使用索引来加速查询。例如，使用`WHERE`子句中的`=`、`>`、`<`、`>=`、`<=`等操作符进行条件筛选时，如果相关字段有索引，MySQL通常会利用这些索引。
> 2. **联接操作**：在进行表联接（如`JOIN`操作）时，如果联接条件中的字段有索引，MySQL会尝试使用这些索引来加速联接过程。
> 3. **排序和分组操作**：当使用`ORDER BY`或`GROUP BY`子句时，如果排序或分组的字段有索引，MySQL可能会利用这些索引来优化排序和分组操作。
> 4. **覆盖索引**：如果一个查询只需要访问索引中的信息，而无需回表查询原始数据，那么这个索引被称为覆盖索引。当MySQL检测到可以使用覆盖索引时，它会优先使用覆盖索引，因为这样可以避免回表操作，从而提高查询效率。
> 5. **唯一性约束和主键约束**：在MySQL中，为表的主键和具有唯一性约束的字段自动创建索引。这些索引不仅用于加速查询，还用于确保数据的唯一性。
>
> 虽然索引可以提高查询性能，但它们也会占用额外的磁盘空间，并可能降低写操作的性能（如INSERT、UPDATE和DELETE操作）
>
> 。因此，在创建索引时，需要权衡查询性能与存储和写操作的开销。通常建议只对经常用于查询条件、排序、分组或联接操作的字段创建索引。同时，可以通过定期分析查询性能和数据访问模式来确定哪些索引是最有益的。

学习指引：MySQL索引的分类、何时使用、何时不使用、何时失效?

13. 【腾讯】WXG视频号面经，撑到了二面|0315

> 投的后台开发，给我捞客户端去了。但是是微信视频号团队，咬咬牙面了。我客户端小白，全程没问客户端的知识。
>
> **3.11一面 1h10min**
>
> 1. 问字节实习经历，做的需求，问细节，问数据供给和处理的链路。
> 2. 问开源项目，看代码，看代码风格，问设计细节。
> 3. 问os项目，问内存分配的设计，MMU，malloc是怎么实现的，缺页机制的实现。
> 4. 又回去问开源项目，扣一些细节。
> 5. 终于开始问八股了。select. poll. epoll讲一下底层原理，多路复用epoll相比select和poll的主要改进点是什么?
> 6. docker用过吗？什么场景怎么用的？你认为docker里的空间和虚拟机有什么区别?
> 7. TCP和UDP的区别?
> 8. TCP三次握手和四次挥手的过程？三次握手为什么不能是两次？四次挥手为什么不能是三次?
> 9. HTTP3.0了解过吗？QUIC协议的底层原理是什么?
> 10. os场景题：一个32位系统，dump结果有1G，但是用户申请512M却触发OOM了，有几种原因?
> 11. 两道算法题：简单题反转链表. 困难题LFU空间复杂度O1，需要自己写结构体自己写main测
>
> **3.14二面 1h10min**
>
> 二面没八股也没算法题，纯问实习经历和项目，问了一个多小时。
>
> 扣得更细节，喜欢钻那种性能问题和意外问题的检测和处理，问压测怎么做之类的。
>
> 项目更喜欢问开源项目和os，问os比较多，比如说用户态切换到内核态是怎么实现的，如何确保用户操作不会越界。

> 会问项目/实习开发过程中遇到什么问题怎么解决，如果这个功能要你实现的更好应该怎么去做。

1. 操作系统的内存分配设计原理

解析：

OS 常见题，难度中等，考察频率适中。

参考答案：

> 操作系统的内存分配设计原理主要基于虚拟内存、内存分页、内存分段以及多种内存分配算法。
>
> 1. 虚拟内存：操作系统为每个进程分配一个虚拟地址空间，这个空间远大于实际的物理内存。进程在运行时，其所需的代码、数据等被映射到虚拟地址空间中。当进程需要访问某个虚拟地址时，如果该地址对应的物理内存页尚未加载，则发生缺页中断，操作系统负责将该页从磁盘或其他存储介质加载到物理内存中。这种方式有效地扩展了可用内存，使得程序可以像使用大内存一样运行。
> 2. 内存分页：操作系统将物理内存划分为固定大小的页，每页的大小通常是固定的，如4KB。同样，虚拟地址空间也被划分为页。当进程需要访问某个虚拟页时，操作系统通过页表查找该页对应的物理页。如果物理页不存在，则触发缺页中断。
> 3. 内存分段：与分页不同，分段是根据程序的逻辑结构来划分内存的。每个段都有自己的名称和长度，并且可以有不同的访问权限。这种方式可以更好地管理程序的内存使用，但实现起来相对复杂。
> 4. 内存分配算法：操作系统使用多种算法来管理内存分配，包括首次适应算法、最佳适应算法、最坏适应算法等。这些算法在分配内存时考虑不同的因素，如分配速度、内存碎片等。例如，首次适应算法从空闲链表的起始位置开始查找，找到第一个满足需求的空闲分区进行分配；而最佳适应算法则查找最小的满足需求的空闲分区进行分配，以减少内存碎片。
>
> 操作系统的内存分配设计原理是为了高效、公平、安全地管理计算机系统的内存资源。通过虚拟内存、内存分页、内存分段以及多种内存分配算法，操作系统能够在有限的物理内存条件下满足多个进程的运行需求，提高系统的整体性能。

学习参考：[十年大佬讲述，操作系统内存管理(图文详解)]

> **面经专栏直通车**

> **面经专栏下载**

2. . 操作系统的MMU

解析：

OS 概念题，难度中等，考察频率中等。

参考答案：

> **操作系统的MMU（Memory Management Unit，内存管理单元）**是中央处理器（CPU）中用来管理虚拟存储器、物理存储器的控制线路。MMU主要负责虚拟地址到物理地址的映射，以及提供硬件机制的内存访问授权，这在多用户多进程操作系统中尤为重要。
>
> MMU的工作原理主要可以概括为地址转换和访问权限控制两个方面。当CPU发出一个访存请求时，MMU会将逻辑地址转换为对应的物理地址，确保能够准确地访问到内存中的数据。同时，MMU还负责控制内存的访问权限，确保程序只能访问到其拥有权限的内存区域，这有助于提高系统的安全性和稳定性。

> 在实际的计算机系统中，MMU通常通过页表来实现地址转换和访问权限控制。页表是一个存储在内存中的数据结构，记录了逻辑地址和物理地址之间的映射关系，以及每个页面的访问权限信息。当CPU发出访存请求时，MMU会根据页表中的映射关系将逻辑地址转换为物理地址，并检查访问权限是否合法。
>
> 除了页表，MMU还可以通过地址变换缓冲器（TLB）来提高地址转换的效率。当CPU发出访存请求时，MMU会首先在TLB中查找对应的页表项，如果找到则直接进行地址转换，否则再去访问页表。

学习参考： MMU原理

3. malloc 是怎么实现的

解析：

c 语言和 os 综合题，难度中等，考察频率高。

参考答案：

> `malloc`是C语言中用于动态分配内存的函数。它的实现方式可以因编译器和操作系统而异，下面是一种常见的实现方式：
>
> 1. 首先，`malloc`函数会接收一个参数，即需要分配的内存大小。
> 2. `malloc`函数会检查当前的空闲内存块链表，查找是否有足够大小的空闲内存块。
> 3. 如果找到了足够大小的空闲内存块，则将其从空闲内存块链表中移除，并返回该内存块的起始地址给调用者。
> 4. 如果没有找到足够大小的空闲内存块，则会进行内存分配。这通常涉及到向操作系统请求更多的内存空间。
> 5. 操作系统会分配一块连续的内存空间，并将其标记为已使用。
> 6. `malloc`函数将分配到的内存块的起始地址返回给调用者，并将其添加到已分配内存块链表中，以便后续的内存管理。
> 7. 调用者可以使用返回的内存块地址来存储数据。
> 8. 当不再需要使用这块内存时，调用者可以使用`free`函数将其释放，将其重新添加到空闲内存块链表中，以便其他程序可以再次使用。
>
> 需要注意的是，`malloc`函数的实现可能会有一些额外的处理，例如内存对齐、内存池等，以提高内存分配的效率和性能。具体的实现细节可能因编译器和操作系统而异。

学习参考： malloc函数实现原理！

4. 缺页机制的实现

解析：

os 基础题，必会题，面试频率较高。

参考答案：

> 当程序试图访问一个尚未映射到物理内存中的虚拟页面时，就会发生缺页中断，从而触发缺页机制。以下是缺页机制的基本实现步骤：
>
> 1. **中断触发**：当CPU尝试访问一个不在物理内存中的页面时，硬件会生成一个缺页中断。这个中断会暂停当前程序的执行，并将控制权交给操作系统。
> 2. **中断处理**：操作系统捕获到缺页中断后，会开始缺页处理流程。首先，它会检查发生缺页中断的虚拟页面的信息。这些信息通常保存在一个特殊的硬件寄存器中，或者通过检查程序计数器和分析当前指令来获取。

3. **页面分配**：一旦确定了需要加载的页面，操作系统会在物理内存中查找一个空闲的页框（Frame）。如果找不到空闲页框，它可能会选择一个当前正在使用的页框进行置换，这个过程通常涉及到页面置换算法（如LRU、FIFO等）来决定哪个页面应该被替换。
4. **页面调入**：当找到空闲页框后，操作系统会从磁盘等辅助存储器中加载相应的页面内容到该页框中。这通常涉及到磁盘I/O操作，因此可能是缺页处理中耗时较长的一个步骤。
5. **更新页表**：页面加载完成后，操作系统会更新页表，将虚拟页面的映射关系更新为新的物理页框。这样，当程序再次访问该页面时，就可以直接从物理内存中获取数据了。
6. **恢复执行**：最后，操作系统会恢复缺页中断发生前的程序执行状态，将程序指令器重新指向引起缺页中断的指令，并继续执行程序。

在整个缺页处理过程中，操作系统需要确保数据的一致性和完整性，同时尽量减少缺页中断对程序性能的影响。这通常涉及到一些复杂的策略和算法，如预取策略、缓存机制等，以优化内存访问的性能。

学习参考：深入理解【缺页中断】

5. select. poll. epoll讲一下底层原理

解析：

操作系统常见题，必会题。

参考答案：

`select`、`poll`和`epoll`都是用于实现I/O多路复用的机制。

1. `select`：`select`是最古老的I/O多路复用机制之一。它通过一个位图来管理文件描述符集合，通过轮询的方式检查每个文件描述符的状态变化。当有文件描述符就绪时，`select`会返回，然后通过遍历文件描述符集合来确定哪些文件描述符可读、可写或出错。
2. `poll`：`poll`是对`select`的改进，它使用一个结构体数组来管理文件描述符集合，每个结构体中保存了文件描述符和关注的事件。与`select`不同的是，`poll`不需要遍历整个文件描述符集合，而是通过系统调用将文件描述符集合传递给内核，内核会将就绪的文件描述符填充到结构体数组中。
3. `epoll`：`epoll`是Linux特有的I/O多路复用机制，相对于`select`和`poll`，它具有更高的性能和扩展性。`epoll`使用了事件驱动的方式，通过将文件描述符添加到内核事件表中，当有事件发生时，内核会通知应用程序。`epoll`提供了三个系统调用：`epoll_create`用于创建一个`epoll`实例，`epoll_ctl`用于控制事件的注册和删除，`epoll_wait`用于等待事件的发生。

在底层原理上，`epoll`利用了Linux内核的事件驱动机制，通过使用红黑树和双链表来管理文件描述符集合，以及利用回调机制来处理就绪的事件。相比于`select`和`poll`的轮询方式，`epoll`能够更高效地处理大量的文件描述符，并且支持边缘触发和水平触发两种工作模式。

`select`、`poll`和`epoll`都是用于实现I/O多路复用的机制，它们在底层原理上有所不同，而`epoll`相对于`select`和`poll`具有更高的性能和扩展性。

学习参考：Linux下的IO多路复用技术：poll、epoll与select的比较

6. 多路复用epoll相比select和poll的主要改进点是什么?

解析：

操作系统常见题，必会题。

参考答案：

多路复用epoll相比select和poll的主要改进点体现在以下几个方面：

1. **效率与性能**：
   - **select**：select在每次调用时都需要遍历整个文件描述符集合，即使只有少数几个描述符就绪，也会进行无用的遍历，导致效率低下。此外，select能同时处理的文件描述符数量是有上限的，这限制了它在高并发场景下的应用。
   - **poll**：虽然poll解决了select的文件描述符数量限制问题，并且通过将输入输出参数进行分离来减少每次设定的开销，但它仍然需要遍历整个文件描述符数组来查找就绪的描述符，因此在文件描述符数量很大时，性能依然有限。
   - **epoll**：epoll通过红黑树和就绪链表来管理文件描述符，并只关注那些状态已经发生变化的文件描述符。这种机制避免了不必要的遍历，从而大大提高了效率。当有大量文件描述符需要监视时，epoll的性能优势尤为明显。
2. **工作模式**：
   - **select和poll**：它们只能工作在低效的LT（水平触发）模式下，即当有事件发生时，需要多次循环处理直到没有事件为止。
   - **epoll**：支持高效的ET（边沿触发）模式。在这种模式下，当有事件响应时，应用程序必须立即处理。这种机制减少了不必要的轮询和等待，进一步提高了性能。
3. **事件处理**：
   - **select和poll**：采用轮询的方式处理事件，即每次调用时都需要检查所有文件描述符的状态。
   - **epoll**：采用回调的方式处理事件。当有就绪的文件描述符时，会触发回调函数，将对应的事件插入到内核就绪事件队列中。内核在适当的时机将该队列中的内容拷贝到用户空间，避免了不必要的遍历和检查。
4. **扩展性**：`epoll`支持边缘触发（Edge Triggered）和水平触发（Level Triggered）两种工作模式。边缘触发模式只在状态变化时通知应用程序，而水平触发模式则在状态保持的情况下持续通知应用程序。这使得应用程序可以更灵活地处理事件。
5. **内存管理**：`epoll`使用红黑树和双链表来管理文件描述符集合，这种数据结构使得添加和删除文件描述符的操作更高效。而`select`和`poll`则使用位图或数组来管理文件描述符集合，效率较低。
6. **大规模并发**：由于`epoll`的高效性和扩展性，它更适合处理大规模并发的场景。它能够处理成千上万个文件描述符，而`select`和`poll`在文件描述符数量较大时性能会下降。

epoll相比select和poll在效率、性能和工作模式等方面都有显著的改进，使其在高并发、大文件描述符数量的场景下具有更好的表现。因此，在实际应用中，epoll通常被作为首选的多路复用机制。

学习参考：Select、Poll、Epoll详解

7. docker用过吗？什么场景怎么用的？你认为docker里的空间和虚拟机有什么区别？

解析：

容器相关基础知识，常考题。

参考答案：

Docker是一种开源的容器化平台，它可以将应用程序及其依赖项打包成一个独立的容器，提供了一种轻量级、可移植和可扩展的软件交付解决方案。

以下是一些常见的使用场景和区别：

1. 应用程序部署：Docker可以将应用程序及其依赖项打包成一个容器，使得应用程序在不同的环境中能够一致地运行，简化了部署过程。

> 2. 微服务架构：Docker容器可以独立运行，每个容器承载一个小型的、独立的服务，使得微服务架构更加灵活和可扩展。
> 3. 持续集成和持续部署：Docker容器可以与持续集成和持续部署工具集成，实现自动化的构建、测试和部署流程。
>
> Docker空间和虚拟机的区别：
>
> 1. 资源占用：虚拟机需要独立的操作系统和硬件资源，因此占用的资源较多。而Docker容器共享主机的操作系统内核，只需要额外的容器运行时和应用程序依赖的资源，因此占用的资源较少。
> 2. 启动速度：虚拟机需要启动整个操作系统，因此启动速度较慢。而Docker容器只需要启动容器运行时和应用程序，因此启动速度较快。
> 3. 隔离性：虚拟机提供了较高的隔离性，每个虚拟机都有独立的操作系统和内核。而Docker容器共享主机的操作系统内核，隔离性相对较弱，但通过Linux内核的命名空间和控制组等技术，仍然能够提供一定程度的隔离。
> 4. 可移植性：虚拟机可以在不同的物理主机或虚拟化平台上运行。而Docker容器可以在任何支持Docker的主机上运行，提供了更高的可移植性。
>
> Docker相对于虚拟机具有更轻量级、更快速的启动和更高的资源利用率，适用于快速部署和扩展应用程序的场景。虚拟机提供了更高的隔离性和可移植性，适用于需要完全隔离的环境或跨平台部署的场景。

学习参考：Docker入门，这一篇就够了，建议收藏

8. TCP和UDP的区别?

解析：

TCP 相关内容属于计算机网络必考部分，要熟练掌握。

参考答案：

> - TCP是面向连接的，UDP是无连接的
> - TCP是可靠的，UDP是不可靠的
> - TCP是面向字节流的，UDP是面向数据报文的
> - TCP只支持点对点通信，UDP支持一对一，一对多，多对多
> - TCP报文首部20个字节，UDP首部8个字节
> - TCP有拥塞控制机制，UDP没有
> - TCP协议下双方发送接受缓冲区都有，UDP并无实际意义上的发送缓冲区，但是存在接受缓冲区

学习参考：TCP与UDP的区别（超详细）

9. TCP三次握手和四次挥手的过程？三次握手为什么不能是两次？四次挥手为什么不能是三次?

解析：

TCP 相关内容属于计算机网络必考部分，要熟练掌握。

参考答案：

> HTTP本身并不涉及三次握手和四次挥手的过程，这些过程实际上是TCP（传输控制协议）建立连接和断开连接时使用的机制。HTTP是建立在TCP之上的应用层协议，因此了解TCP的这些机制有助于更好地理解HTTP的工作方式。
>
> **三次握手的过程**：

1. **SYN（同步）阶段**：客户端向服务器发送一个SYN包，并等待服务器的确认。这个SYN包中包含了客户端的初始序列号。
2. **SYN-ACK（同步-应答）阶段**：服务器收到SYN包后，向客户端发送一个SYN-ACK包作为应答。这个SYN-ACK包中包含了服务器的初始序列号以及对客户端初始序列号的确认。
3. **ACK（应答）阶段**：客户端收到SYN-ACK包后，向服务器发送一个ACK包作为最终的确认。这个ACK包中包含了对服务器初始序列号的确认。

为什么不能是两次：

- **防止已失效的连接请求报文段突然又传送到了服务端**：客户端发出的连接请求报文段可能因为网络拥堵而在某个网络节点长时间滞留，以致延误到连接释放以后的某个时间才到达服务端。这时，服务端会误以为这是一个新的连接请求，于是又向客户端发出确认报文段，同意建立连接。如果不采用“三次握手”，那么只要服务端发出确认，新的连接就建立了。由于现在客户端并没有发出建立连接的请求，因此不会理睬服务端的确认，也不会向服务端发送数据。但服务端却认为新的运输连接已经建立，并一直等待客户端发来数据。这样，服务端就白白浪费了许多资源。采用“三次握手”的办法可以防止上述现象发生。例如上面这种情况发生时，客户端不会向服务端的确认发出确认。服务端由于收不到确认，就知道客户端并没有要求建立连接。
- **防止已失效的连接请求报文段被服务端接收**：客户端发出的第一个连接请求报文段并没有丢失，而是在某个网络结点长时间的滞留后，最终到达了服务端。这本是一个早已失效的报文段。但服务端收到此失效的连接请求报文段后，就误认为是客户端再次发出的一个新的连接请求。于是就向客户端发出确认报文段，同意建立连接。假设不采用“三次握手”，那么只要服务端发出确认，连接就建立了。但由于客户端并没有发出建立连接的请求，因此不会理睬服务端的确认，也不会向服务端发送数据。但服务端却认为新的运输连接已经建立，并一直等待客户端发来数据。这样，服务端就又白白浪费了许多资源。

四次挥手的过程：

1. **FIN（结束）阶段**：客户端向服务器发送一个FIN包，表示客户端想要关闭连接。
2. **ACK（应答）阶段**：服务器收到FIN包后，向客户端发送一个ACK包作为应答，表示已经收到客户端的关闭请求。此时，服务器仍然可以发送数据给客户端。
3. **FIN阶段**：当服务器完成所有数据发送后，它也向客户端发送一个FIN包，表示服务器也想要关闭连接。
4. **ACK阶段**：客户端收到服务器的FIN包后，向服务器发送一个ACK包作为最终的确认，表示客户端已经收到服务器的关闭请求。至此，连接完全关闭。

为什么不能是三次：

- **确保数据完全传输**：在四次挥手中，第一次和第二次挥手确保了客户端发起关闭连接的请求，并得到服务器的确认。而第三次和第四次挥手则确保了服务器在发送完所有数据后也发起关闭连接的请求，并得到客户端的确认。这样的设计可以确保数据在双方之间完全传输，避免了数据丢失或未完全发送的情况。
- **处理半关闭状态**：TCP连接允许半关闭状态，即一方可以关闭它的发送通道，但继续接收来自另一方的数据。四次挥手能够处理这种半关闭状态，确保在双方都同意关闭连接之前，连接不会被意外中断。

学习参考：一文彻底搞懂 TCP三次握手、四次挥手过程及原理

9. HTTP3.0了解过吗？QUIC协议的底层原理是什么？

解析：

HTTP3.0 相关内容属于计算机网络常考部分，要熟练掌握。

参考答案：

> HTTP/3.0是下一代HTTP协议，其最显著的特点是采用了QUIC协议作为传输层协议，取代了之前的TCP+TLS+HTTP/2.0的组合。QUIC协议的设计初衷是为了解决TCP协议在建立连接时的延迟问题，同时提供更高效、更安全的数据传输。
>
> QUIC协议的底层原理主要包括以下几个方面：
>
> 1. **基于UDP的传输**：QUIC协议是基于UDP（用户数据报协议）的，这使得QUIC能够避免TCP协议在建立连接时的三次握手过程，从而降低了连接建立的延迟。此外，UDP的无连接特性也使得QUIC在数据传输时更加灵活和高效。
> 2. **多路复用和流量控制**：QUIC协议实现了类似TCP的流量控制功能，能够根据网络状况动态调整数据的发送速率，以避免网络拥塞。同时，QUIC也支持多路复用，这意味着在同一个物理连接上，可以有多个独立的逻辑数据流同时进行数据传输，从而提高了数据的传输效率。
> 3. **加密和安全性**：QUIC协议集成了TLS（传输层安全性协议）加密功能，使得数据传输过程中可以确保数据的机密性和完整性。这种加密功能不仅提高了数据传输的安全性，也避免了TCP+TLS组合时可能产生的握手延迟。
> 4. **快速握手**：由于QUIC协议是基于UDP的，因此它支持0-RTT（零往返时间）和1-RTT（一次往返时间）的快速握手功能。这意味着在客户端和服务器之间已经建立过连接的情况下，客户端可以无需再次进行完整的握手过程就能发送数据，从而进一步降低了连接的延迟。
>
> HTTP/3.0和QUIC协议的推广和应用也面临一些挑战。例如，目前许多服务器和浏览器端对HTTP/3.0的支持还不够完善，这在一定程度上限制了HTTP/3.0的普及。此外，由于UDP协议在现有网络中的优化程度还低于TCP，因此在部署HTTP/3.0和QUIC协议时可能需要考虑网络环境的优化问题。

学习参考：[QUIC/HTTP3 协议原理]

10. 一个32位系统，dump结果有1G，但是用户申请512M却触发OOM了，有几种原因?

解析：

操作系统中，难度较高，属于提高题。

参考答案：

> 在一个32位系统中，尽管系统整体的dump结果（可能是内存转储或系统日志）有1G，但用户申请512M内存时触发OOM（Out of Memory，内存溢出）可能有以下几种原因：
>
> 1. **内存碎片**：系统中的可用内存可能已经被分割成许多小块，无法组合成足够大的连续内存块来满足512M的请求。这种情况通常发生在长时间运行且频繁进行内存分配和释放的系统中。
> 2. **内核或驱动占用大量内存**：内核本身或某些驱动程序可能占用了大量内存，导致用户空间可用的内存减少。即使系统整体的内存使用没有达到上限，用户空间也可能因为内核或驱动的占用而无法申请到足够的内存。
> 3. **缓存和缓冲区占用**：系统的缓存（如文件缓存、页面缓存等）和缓冲区可能占用了大量内存。这些内存虽然可以被回收，但在某些情况下（如内存压力较大时），回收过程可能不够及时，导致用户空间无法申请到足够的内存。
> 4. **内存泄漏**：系统中可能存在内存泄漏的问题，即某些进程或模块在申请内存后没有正确释放，导致内存逐渐耗尽。即使总的dump结果有1G，但可用的连续内存可能已经很少。
> 5. **内存超配**：某些虚拟化环境或容器技术可能会采用内存超配的策略，即允许虚拟机或容器使用的内存超过其物理内存的限制。在这种情况下，即使物理内存足够，但由于超配策略的限制，也可能触发OOM。
> 6. **内存限制**：系统管理员可能为用户或进程设置了内存使用限制（如通过cgroups或ulimit等工具）。即使系统整体有足够的内存，但如果用户或进程达到了其内存使用限制，也会触发OOM。

> 7. **内核参数配置不当**：某些内核参数（如overcommit策略）的配置可能影响到系统对内存申请的处理方式。如果配置不当，可能导致系统即使有足够的内存也拒绝用户的内存申请。

学习参考：[什么是OOM，为什么会OOM及一些解决方法]

> [更多面经直通车]
>
> [原贴连接]

14.【京东】JDS 后端面经，面试体验很棒|0316

> 面试管真的很好，有什么地方不懂都会耐心解答。
>
> 1. 你的这几个项目有 SQL 调优的经验吗?
> 2. 来说说 MySQL的索引?
> 3. MySQL索引为什么不用红黑树，AVL树，为什么要用 B+树?
> 4. 说说 MySQL语句的执行过程?
> 5. MySQL有缓存吗?
> 6. MySQL的事务说一下?
> 7. Spring 的事务和我们 MySQL的事务是一回事吗?
> 8. Spring的事务什么时候会失效?
> 9. 非聚簇索引的回表机制?
> 10. 哪些情况下索引会失效?
> 11. MQ 如何保证消息一定被消费?
> 12. MQ 如何保证消息消费的顺序性?
> 13. 什么原因会导致 MQ 消息积压?
> 14. MQ 的死信队列了解吗，有没有在项目中用过?
> 15. ES的深度分页怎么解决?
> 16. ES 的写入性能调优了解吗?
> 17. MyBatisPlus 遇到慢 SQL怎么排查?
> 18. TomCat 服务器 CPU 负载特别高，但是内存不高可能是什么问题导致的?
> 19. 我们的 Redis 缓存怎么保证和数据库中数据的一致性?
> 20. Redis 什么情况会导致读写性能突然变慢?
> 21. 单个 Redis 中有热 Key，压力特别大，怎么解决?
> 22. Redis 主从，出现宕机一个节点后，怎么恢复数据，怎么产生新的丰?
> 23. Redis 的内存淘汰策略了解吗?

1. 来说说 MySQL的索引?

解析：

MySQL 索引属于必考题，难度中等。

参考回答：

> **一、MySQL索引原理**
>
> MySQL索引采用了B+树的数据结构，能够大大提高查询效率。它类似于书籍的目录，通过索引，数据库系统可以迅速定位到表中的特定数据，无需扫描整个表。
>
> **二、MySQL索引的优点**
>
> 1. **提高查询速度**：通过索引，数据库可以迅速找到所需数据，避免了全表扫描的耗时操作。

2. **保证数据的唯一性**：通过唯一索引，可以确保数据库表中每一行数据的某列或多列组合是唯一的。
3. **加速表与表之间的连接**：在执行连接操作时，如果连接的字段已经被索引，那么连接的速度会更快。

三、MySQL索引的缺点

1. **占用磁盘空间**：索引本身需要占用一定的磁盘空间。
2. **降低写操作的性能**：每次对表中的数据进行增、删、改操作时，索引也需要进行相应的调整，这可能会降低写操作的性能。

四、MySQL索引的使用场景

1. **经常需要搜索的列**：对于经常出现在WHERE子句中的列，应该考虑建立索引。
2. **作为连接键的列**：如果某列经常出现在连接（JOIN）操作中，也应该考虑建立索引。
3. **经常需要排序的列**：如果某列经常需要按照其值进行排序，也可以考虑建立索引。

五、注意事项

1. **避免过度索引**：不是每个列都需要建立索引，过多的索引会占用更多的磁盘空间，并可能降低写操作的性能。因此，需要根据实际情况进行选择。
2. **定期维护索引**：随着时间的推移，数据库中的数据会发生变化，索引的性能也可能会受到影响。因此，需要定期检查和优化索引。

总的来说，MySQL索引是提高数据库查询性能的重要工具，但也需要根据实际情况进行合理使用和维护。

学习指引：MySQL索引详解（一文搞懂）

2. MySQL索引为什么不用红黑树，AVL树，为什么要用B+树?

解析：

MySQL 索引属于必考题，难度中等。

参考回答：

MySQL索引选择使用B+树而不是红黑树或AVL树，主要是基于B+树在数据库环境中的特定优势和特性。

1. **多路搜索特性**：B+树是一个多路平衡搜索树，这意味着每个节点可以有多于两个的子节点。这种特性使得B+树相对于二叉树（如红黑树和AVL树）在树的高度上具有优势。在数据库中，树的高度直接关系到查询性能，因为每次查询都需要从根节点遍历到叶子节点。多路搜索特性有助于降低树的高度，从而减少I/O操作和查询时间。
2. **范围查询和排序性能**：B+树特别适用于范围查询和排序操作。由于其叶子节点之间通过指针相连，可以很方便地遍历叶子节点以获取范围内的数据。相比之下，红黑树和AVL树在进行范围查询时可能需要中序遍历，效率较低。此外，B+树在叶子节点中存储了所有的数据，使得排序操作更加高效。
3. **磁盘I/O优化**：在数据库中，数据通常存储在磁盘上，而磁盘I/O操作是数据库性能的关键瓶颈之一。B+树的非叶子节点只存储键值信息，而实际的数据存储在叶子节点中。这种设计使

> 得非叶子节点可以容纳更多的键值信息，从而减少了树的高度和磁盘I/O次数。此外，由于B+树的叶子节点之间通过指针相连，数据库系统可以更加高效地读取和缓存数据。
> 4. **平衡性**：B+树在插入和删除节点时能够保持较好的平衡性，避免了树的高度过度增长。这种平衡性保证了查询性能的稳定性，不会因为数据的插入和删除操作而产生显著的波动。
>
> 红黑树和AVL树虽然也是平衡二叉树，但在数据库环境中可能并不适用。它们的树高度相对较高，导致查询性能下降；同时，它们在处理范围查询和排序操作时效率较低。此外，红黑树在插入和删除节点时需要调整树的结构以保持平衡，这也会增加额外的开销。

学习指引：为什么mysql索引底层使用的是B+树存储，而不是红黑树吗？

3. 说说 MySQL语句的执行过程?

解析：

MySQL 的基础题，常考题。

参考回答：

> MySQL语句的执行过程涉及多个关键阶段，以下是其大致的执行流程：
>
> 1. **客户端发送SQL语句**：
>   当用户在客户端（如MySQL命令行工具、图形化界面工具等）输入SQL语句并执行时，这条语句首先被发送到MySQL服务器。
> 2. **词法解析和语法解析**：
>   MySQL服务器接收到SQL语句后，首先会进行词法解析，将SQL语句拆分成一个个的词汇单元（如关键字、表名、列名等）。接着进行语法解析，根据MySQL的语法规则判断这条SQL语句是否合法，并生成一个“解析树”。
> 3. **预处理**：
>   在语法解析之后，MySQL会进行预处理阶段。这个阶段会检查SQL语句中涉及的表、列等是否存在，并解析权限。如果涉及存储过程、函数或触发器，也会在这一阶段进行预处理。
> 4. **优化器优化**：
>   MySQL查询优化器会对解析树进行优化，选择最高效的执行计划。它会考虑多种因素，如表的大小、索引、统计信息等，以决定如何最快地获取查询结果。
> 5. **生成执行计划**：
>   优化器根据优化结果生成一个详细的执行计划，这个计划描述了如何获取数据、如何连接表、如何排序等。
> 6. **执行引擎执行**：
>   执行引擎按照执行计划开始执行。它可能会与存储引擎交互，从磁盘上读取数据、写入数据或更新数据。如果涉及多个表，执行引擎还会负责表的连接操作。
> 7. **返回结果**：
>   执行引擎将查询结果返回给客户端。这个结果可能是查询到的数据行，也可能是受影响的行数（如INSERT、UPDATE、DELETE操作）。
> 8. **清理**：
>   查询执行完毕后，MySQL会进行清理工作，如释放内存、关闭临时表等。
>
> 在整个过程中，MySQL还会涉及一些日志的写入，如二进制日志（用于复制和恢复操作）、慢查询日志（记录执行时间较长的查询）等。此外，如果开启了事务，还会涉及事务的管理和锁定机制。

学习指引：【SQL】Mysql中一条sql语句的执行过程

4. MySQL有缓存吗?

解析：

考察 MySQL 的基础，属于中等难度题。

参考回答：

> **MySQL有缓存机制。**MySQL数据库会缓存已经执行过的SQL语句和语句执行结果。如果下次提交同一个SQL语句，MySQL就会直接从缓存中读取执行结果，而不是重新分析、执行SQL，这样可以减少SQL语句的执行时间，提高查询效率。
>
> 但是，如果表中的数据发生变化，所有与之相关的缓存都会被释放刷新，以避免出现数据脏读问题。此外，MySQL也提供了手动选择是否使用缓存查询的功能，可以在SQL查询语句的字段前增加SQL_NO_CACHE或SQL_CACHE关键字来控制是否使用缓存。
>
> 总的来说，MySQL的缓存机制是其性能优化的一部分，有助于提高查询速度和效率。然而，需要注意的是，缓存并不总是有利的，特别是在数据频繁变化的情况下，因此需要根据实际情况合理使用和管理缓存。

学习指引：[玩转MySQL之四]MySQL缓存机制

5. MySQL的事务说一下?

解析：

考察数据库基础知识，必考题。

参考回答：

> MySQL的事务（Transaction）是数据库管理系统执行过程中的一个逻辑单位，它由一个或多个SQL语句组成，这些语句要么全部执行，要么全部不执行。事务的主要目的是确保数据的完整性和一致性，在并发操作中保持数据的正确状态。
>
> 事务具有以下四个关键特性，通常被称为ACID特性：
>
> 1. **原子性（Atomicity）**：事务被视为一个不可分割的工作单位，事务中包括的诸操作要么都做，要么都不做。
> 2. **一致性（Consistency）**：事务必须使数据库从一个一致性状态变换到另一个一致性状态。这意味着一个事务在执行前后，数据库都必须处于一致性状态。
> 3. **隔离性（Isolation）**：在事务执行过程中，其他事务不能访问该事务的数据，直到该事务完成。这确保了并发执行的事务不会相互干扰。
> 4. **持久性（Durability）**：一旦事务提交，则其结果就是永久性的，即使系统崩溃也不会丢失。
>
> 在MySQL中，特别是InnoDB存储引擎，事务得到了全面的支持。当你在InnoDB中执行一个事务时，可以包含多个SQL语句，这些语句要么全部成功，要么在发生错误时全部回滚（撤销）。这通过维护一个撤销日志（undo log）来实现，当事务需要回滚时，可以利用这个日志将数据恢复到事务开始之前的状态。
>
> 此外，MySQL还提供了事务控制语句，如`COMMIT`和`ROLLBACK`，来显式地提交或回滚事务。`COMMIT`用于提交事务，即将事务中的修改永久保存到数据库中；而`ROLLBACK`则用于撤销事务中的修改，将数据库恢复到事务开始之前的状态。

学习指引：MySQL事务【详解-最新的总结】

6. Spring 的事务和我们 MySQL的事务是一回事吗?

解析：

考察Spring和MySQL的对比，需要了解。

参考回答：

> Spring 的声明式事务和数据库的事务并不是一回事，尽管它们在某些方面存在关联。
>
> 声明式事务是 Spring 框架提供的一种机制，用于在应用程序中管理事务的执行。它允许你通过配置方式定义事务的行为，而无需显式编写事务管理代码。声明式事务可以应用于多个业务方法，而不仅仅是数据库操作。
>
> 数据库的事务是数据库引擎提供的一种机制，用于确保数据库操作的原子性、一致性、隔离性和持久性（ACID 特性）。数据库事务通常用于在数据库操作期间维护数据的完整性和一致性。它可以包含多个数据库操作，并且在事务执行期间，这些操作要么全部提交成功，要么全部回滚到事务开始前的状态。
>
> 在 Spring 中，声明式事务可以应用于包含数据库操作的方法，以确保这些操作在事务的上下文中执行，并根据需要进行提交或回滚。
>
> 声明式事务通过将事务管理逻辑与业务逻辑分离，提供了更高层次的抽象和灵活性。它可以用于管理其他类型的事务，如消息队列、远程服务调用等。
>
> 总结来说，声明式事务是 Spring 框架提供的一种事务管理机制，可以应用于多种业务操作，包括数据库操作。而数据库的事务是数据库引擎提供的一种机制，用于确保数据库操作的一致性和完整性。Spring 的声明式事务可以管理数据库事务，但它不仅限于数据库操作。

学习指引：spring声明式事务和数据库的事务是一回事吗

7. Spring的事务什么时候会失效?

解析：

考察Spring的熟悉程度，常考题。

参考回答：

> Spring的事务在多种情况下可能会失效，以下是一些常见的原因：
>
> 1. **自调用**：当类中的方法调用本类中的另一个方法时，如果调用是通过this进行的（通常省略），那么此时this并不是代理对象，而是实际的类实例。因此，事务不会生效。解决这个问题的方法是，从Spring的IoC容器中获取该类的代理对象，并通过代理对象来调用方法。
> 2. **方法访问权限问题**：Spring要求被代理的方法必须是public的。如果方法不是public的，事务将不会生效。此外，如果方法被final修饰，Spring的动态代理无法代理final方法，因此事务也会失效。
> 3. **数据库不支持事务**：某些数据库引擎（如MySQL的MyISAM引擎）不支持事务，因此即使Spring配置了事务，这些操作也不会在事务中执行。
> 4. **方法没有被Spring管理**：如果类没有被Spring管理（即没有添加@Controller、@Service、@Repository等注解），那么它的方法不会被Spring的事务管理器控制，因此事务不会生效。
> 5. **异常处理不当**：如果在事务方法中发生异常，并且该异常没有被Spring的事务管理器捕获，那么事务不会回滚。此外，对于非RuntimeException（即checked异常），Spring默认不会回滚事务，除非在@Transactional注解中明确指定了rollbackFor属性。
> 6. **多线程调用**：由于Spring的事务管理是基于ThreadLocal的，不同线程间的事务是隔离的。因此，如果在一个线程中开启事务，然后在另一个线程中执行数据库操作，那么这些操作不会参与之前线程的事务。
> 7. **错误的传播属性**：@Transactional注解有一个propagation属性，用于指定事务的传播行为。如果使用了错误的传播属性，可能导致事务的行为不符合预期。
> 8. **自定义了回滚异常与事务回滚异常不一致**：如果在@Transactional注解中自定义了回滚的异常类型，但实际抛出的异常与该类型不匹配，那么事务不会回滚。

学习指引：Spring事务失效场景

8. 非聚簇索引的回表机制?

解析：

考察MySQL索引，必考题。

参考回答：

> 非聚簇索引的回表机制是数据库查询中的一个重要概念，尤其在MySQL的InnoDB存储引擎中。以下是对非聚簇索引回表机制的详细解释：
>
> 首先，我们需要理解聚簇索引和非聚簇索引的基本区别。聚簇索引的叶子节点存储的是行记录，也就是说，数据与索引是存储在一起的。而非聚簇索引的叶子节点存储的则是主键值，也就是说，数据和索引是分开的。
>
> 当我们使用非聚簇索引进行查询时，查询过程大致如下：
>
> 1. 数据库首先在非聚簇索引中找到满足查询条件的数据行的主键值。
> 2. 然后，根据这些主键值，数据库需要再次回到聚簇索引（通常是主键索引）中查找相应的数据行。这个过程就像是根据一个地址（主键值）去找到实际的房屋（数据行）。
>
> 这个从非聚簇索引回到聚簇索引查找数据行的过程，就是所谓的“回表”。回表操作会增加额外的IO操作和时间开销，因为需要再次访问数据表。在大量数据或者频繁进行回表查询的场景下，这会对查询性能产生显著影响。
>
> 为了避免频繁的回表查询，一种优化策略是使用覆盖索引。覆盖索引是指查询只需要通过索引就可以返回所需要的数据，而无需回表去访问实际的数据行。这可以显著提高查询效率。

学习指引：MySQL - 聚簇索引和非聚簇索引，回表查询，索引覆盖，索引下推，最左匹配原则

9. 哪些情况下索引会失效?

解析：

考察MySQL索引失效，必考题。

参考回答：

> 索引失效是指在某些情况下，数据库查询无法有效利用索引来加速查询过程，从而导致查询性能下降。以下是一些常见的导致索引失效的情况：
>
> 2. **联合索引列顺序不正确**：联合索引的列顺序对查询效率有重要影响。如果查询条件中经常使用的列没有放在联合索引的前面，那么索引可能无法被充分利用。
> 3. **索引列上使用了函数或表达式**：当在索引列上应用函数或表达式时，数据库通常无法直接利用索引进行查询，从而导致索引失效。
> 4. **数据库表设计不合理**：表设计的不合理，如包含大量不必要的字段或冗余数据，或字段类型选择不当，都可能影响索引的有效性。
> 5. **列类型不匹配**：当查询条件中的数据类型与索引列的数据类型不匹配时，索引可能无法被使用，从而导致索引失效。
> 6. **使用了IS NULL或IS NOT NULL**：如果索引列中包含NULL值，并且在查询条件中使用了IS NULL或IS NOT NULL，那么索引可能无法被有效利用。
> 7. **隐式类型转换**：当查询条件中涉及到隐式类型转换时，如将字符串类型与数值类型进行比较，索引可能会失效。
>
> 为了避免索引失效，需要合理设计数据库表结构、选择适当的索引类型、定期更新数据库统计信息，并在编写查询语句时注意避免上述可能导致索引失效的操作。同时，定期监控和分析查询性能，根据需要进行索引优化和调整，也是保持索引有效性的重要措施。

学习指引：索引失效的情况及解决(超详细)

10. MQ 如何保证消息一定被消费?

解析:

MQ常考题，必会题。

参考回答:

> MQ（消息队列）是一种用于在分布式系统中进行异步通信的机制。为了保证消息一定被消费，MQ 通常会采用一系列机制和技术。以下是一些常见的方法：
>
> 1. **确认机制**：当消息被发送到MQ后，MQ会等待消费者的确认。消费者处理完消息后，会向MQ发送一个确认消息，表示该消息已经被成功消费。如果MQ在一定时间内没有收到确认消息，它会认为该消息没有被成功消费，然后会重新发送消息给消费者。这种机制确保了消息至少被消费一次。
> 2. **持久化存储**：MQ 通常会将消息持久化存储到磁盘或数据库中，以防止消息丢失。即使MQ服务器宕机或重启，也能从持久化存储中恢复消息，确保消息不会被遗漏。
> 3. **重试机制**：如果消费者在处理消息时失败，MQ会尝试重新发送消息给消费者。重试的次数和间隔可以根据需要进行配置。通过重试机制，可以确保在消费者暂时不可用或处理失败的情况下，消息仍然能够被成功消费。
> 4. **死信队列**：对于多次尝试消费都失败的消息，MQ 可以将其发送到死信队列中。这样可以避免消息一直阻塞在正常队列中，同时也为开发者提供了处理这些消息的机会。开发者可以定期检查死信队列，对其中的消息进行特殊处理。
> 5. **消息幂等性**：对于某些业务场景，要求即使重复消费相同的消息也不会产生副作用。这时，需要保证消息的幂等性。在消费者处理消息时，可以通过一些技术手段（如唯一ID、分布式锁等）来确保消息只被处理一次。
> 6. **监控和告警**：MQ 通常提供监控和告警功能，可以实时监控消息的消费情况。当消息消费出现异常时，MQ会及时发出告警通知，以便开发者能够及时处理问题。

学习指引: RabbitMQ 消息可靠性的保证

11. MQ 如何保证消息消费的顺序性?

解析:

MQ常考题，必会题。

参考回答:

> 1. 使用单个消费者：让单个消费者处理队列中的所有消息，这样可以确保消息按照它们进入队列的顺序被处理。但是，这种方法在处理大量消息时可能会成为性能瓶颈。
> 2. 使用锁机制：使用锁机制可以防止多个消费者同时访问同一消息。例如，使用数据库锁或分布式锁来确保在任何时候只有一个消费者可以处理特定消息。然而，这种方法可能会增加系统的复杂性和开销。
> 3. 使用事务：将消息处理与事务结合使用可以确保消息的一致性和顺序性。在事务期间，消费者会锁定消息，进行一些处理，然后提交事务。如果事务失败，消息将被回滚到队列中，以便其他消费者可以重新处理。
> 4. 使用消息序列号：为每条消息分配一个唯一的序列号，消费者根据这个序列号来处理消息。当消费者处理完一条消息后，它可以将该序列号提交到下一个待处理的消息。这种方法可以避免消息丢失和重复处理的问题。
> 5. 使用可靠的消息队列：选择一个可靠的消息队列系统，如RabbitMQ或Apache Kafka，这些系统提供了消息持久化和确认机制，以确保消息不会丢失或被重复处理。
> 6. 限制并发处理：通过限制消费者的并发级别，可以控制同时处理多少消息。这样可以避免因同时处理大量消息而导致的问题，如死锁和竞态条件。

7. 考虑使用单向消息传递：采用单向消息传递模型，其中消息只能从生产者流向消费者，而不能返回生产者。这样可以减少因消息返回而引起的复杂性和性能问题。

学习指引：如何保证MQ中消息的顺序性？

12. 什么原因会导致 MQ 消息积压?

解析：

MQ常考题，必会题。

参考回答：

MQ消息积压是指生产者发送的消息在Broker端大量堆积，无法被消费者及时消费，导致业务功能无法正常使用。以下是一些导致MQ消息积压的常见原因：

1. **流量变大而服务器配置偏低**：当消息的产生速度大于消费速度时，如果RabbitMQ服务器配置较低，就可能导致消息积压。
2. **消费者故障**：如果消费者出现宕机或网络问题，导致无法及时消费消息，消息会持续堆积。
3. **程序逻辑设计问题**：如果生产者持续生产消息，但消费者由于某种原因（如处理逻辑耗时过长）消费能力不足，也会造成消息积压。
4. **新上线的消费者功能存在BUG**：新上线的消费者功能如果有缺陷，可能导致消息无法被正常消费，从而引发消息堆积。
5. **配置不合理**：消息队列的容量设置过小或消费者的线程数设置过少，都可能导致消息积压。
6. **生产者推送大量消息**：在特定场景下，如大促活动，生产者可能短时间内推送大量消息至Broker，如果消费者的消费能力不足以应对这种突发流量，也会导致消息堆积。

为了解决MQ消息积压问题，可以采取以下策略：

1. **扩容**：纵向扩容，增加服务器资源，如内存和CPU；横向扩容，将单机改为集群模式，增加集群节点，并增加消费者数量。
2. **优化程序逻辑**：确保生产者和消费者的逻辑设计合理，避免生产者过快生产消息或消费者处理消息过慢。
3. **监控和报警**：建立有效的监控和报警机制，及时发现并解决消息积压问题。

学习指引：消息积压解决方案

13. MQ 的死信队列了解吗，有没有在项目中用过?

解析：

MQ常考题，必会题。

参考回答：

MQ的死信队列（Dead-Letter-Exchange，简称DLX）是一个在RabbitMQ中用于处理无法被正常消费的消息的机制。当消息在队列中因为某些原因（如被拒绝、过期或队列达到最大长度）而无法被消费时，它们会被发送到死信交换机，进而路由到死信队列中等待进一步处理。

以下是关于RabbitMQ死信队列的一些关键点：

1. **消息被拒绝并设置requeue为false**：当消费者使用basic.reject或basic.nack方法拒绝消息，并且设置requeue参数为false时，消息不会重新入队，而是会被发送到死信交换机。
2. **消息过期**：可以为队列或消息设置TTL（Time-To-Live）值，当消息在队列中的存活时间超过这个值时，消息会变为死信。
3. **队列达到最大长度**：当队列中的消息数量达到最大限制，并且无法再接受新消息时，如果队列的设置是丢弃最旧的消息或者将消息转为死信，那么被丢弃或转为死信的消息会发送到死信交换机。

4. **配置死信交换机和路由**：为了确保死信能够被正确处理，需要为每个业务队列配置一个死信交换机，并为死信交换机配置一个或多个路由键和队列。这样，当消息变为死信时，它们会根据配置的路由键被路由到相应的死信队列。
5. **死信队列的处理**：死信队列中的消息需要被特别关注和处理，因为它们代表了系统中存在的问题或异常情况。开发者可以编写特定的消费者来监听死信队列，以便及时发现并解决这些问题。

通过合理配置和使用死信队列，可以帮助我们更好地管理和监控MQ中的消息，确保消息能够被正确处理，避免消息丢失或积压，提高系统的稳定性和可靠性。

学习指引：深入解析RabbitMQ死信队列

14. ES的深度分页怎么解决?

解析：

ES深度分页属于常考内容，难度中等。

参考回答：

ES（Elasticsearch）的深度分页问题通常是由于查询大量数据时，性能会受到影响。

1. **设置max_result_window参数**：这是分页返回的最大数据量的设置。虽然这可以暂时解决问题，但随着数据量的增大，OOM（内存溢出）问题可能会更加严重。
2. **设置数据限制**：参考一些大型互联网公司（如淘宝、百度、谷歌等）的做法，对于越往后的数据，其对用户的影响通常越小，因此可以限制返回的数据量。
3. **滚动查询（Scroll Search）**：滚动查询通过保存快照，并在查询时通过快照获取数据。然而，这种查询方式对Client端并不友好，因为数据的更新、删除和新增都不会影响快照。
4. **search_after方式**：这种方式是根据上一页的最后一条数据来确定下一页的位置。由于这种特殊的查询方式不支持跳页查询，只能依赖上一页的数据。

学习指引：ElasticSearch深度分页解决方案

15. ES 的写入性能调优了解吗?

解析：

ES性能调优属于常考内容，难度中等。

参考回答：

1. 批量写入：使用批量写入API可以显著提高写入性能。将多个文档一次性提交到ES，而不是逐个提交，可以减少网络开销和请求处理时间。
2. 刷新间隔：ES默认每秒自动刷新一次索引，可以通过增加刷新间隔来提高写入性能。刷新间隔越长，写入性能越高，但是数据的可见性会有所延迟。
3. 副本数：副本是ES中用于提高数据冗余和可用性的机制。增加副本数可以提高读取性能，但会降低写入性能。可以根据需求权衡副本数和写入性能。
4. 索引分片：ES将索引分成多个分片，每个分片可以独立地进行读写操作。增加分片数可以提高写入性能，但也会增加集群的负载和资源消耗。需要根据集群规模和硬件配置来确定合适的分片数。
5. 硬件优化：使用高性能的硬件可以提升ES的写入性能。例如，使用SSD硬盘可以加快磁盘写入速度，增加内存可以提高缓存效果。
6. 禁用不必要的功能：ES提供了许多功能和插件，但并不是所有功能都对写入性能有利。禁用不必要的功能和插件可以减少系统开销，提高写入性能。
7. 异步写入：使用异步写入机制可以将写入操作放入后台线程进行处理，提高写入性能。但需要注意异步写入可能会导致数据丢失的风险，需要根据业务需求进行权衡。

学习指引：ES写入性能优化方案

16. MyBatisPlus 遇到慢 SQL怎么排查?

解析：

MyBatisPlus常考题，必会题。

参考回答：

> 1. 确认是否是慢SQL：首先，确认是否真的是SQL语句导致了性能问题。可以通过日志或者性能监控工具来查看具体的SQL执行时间和性能指标。
> 2. 分析SQL语句：仔细分析慢SQL语句，检查是否存在不必要的查询、多表关联、大数据量操作等问题。可以使用数据库的查询分析工具（如EXPLAIN）来查看SQL的执行计划，判断是否存在索引缺失、全表扫描等性能问题。
> 3. 检查索引：确保数据库表中的相关字段都有适当的索引。索引可以加快查询速度，减少数据库的IO操作。可以通过数据库的索引优化工具或者命令来检查索引的使用情况。
> 4. 优化SQL语句：根据分析结果，对慢SQL进行优化。可以考虑使用合适的索引、优化查询条件、减少不必要的字段查询等方式来提高SQL的执行效率。
> 5. 使用缓存：如果某些查询结果是经常被使用的，可以考虑使用缓存来提高查询性能。MyBatis Plus提供了缓存的支持，可以配置二级缓存或者使用其他缓存框架（如Redis）来缓存查询结果。
> 6. 调整数据库连接池：检查数据库连接池的配置参数，确保连接池大小、最大连接数等参数设置合理。过小的连接池可能导致连接等待，从而影响SQL的执行性能。
> 7. 监控和日志：使用监控工具和日志记录来跟踪SQL的执行情况和性能指标。可以使用数据库的性能监控工具、MyBatis Plus的日志配置等方式来获取更详细的信息。

学习指引：Mybatis-Plus执行超长SQL性能问题排查

17. TomCat 服务器 CPU 负载特别高，但是内存不高可能是什么问题导致的?

解析：

考察TomCat问题解决能力，需要一定知识储备。

参考回答：

> 当Tomcat服务器的CPU负载特别高，但内存占用并不高时，可能是以下几个问题导致的：
>
> 1. 高并发请求：如果Tomcat服务器面临大量的并发请求，CPU负载会增加。即使内存占用不高，但CPU需要处理大量的请求和线程调度，导致负载升高。
> 2. 长时间运行的线程：如果Tomcat中存在长时间运行的线程，例如长时间的数据库查询或者耗时的业务逻辑处理，这些线程会占用CPU资源，导致CPU负载升高。
> 3. 错误的配置：Tomcat的配置参数可能不合理，导致CPU负载升高。例如，线程池配置过小，无法处理大量的并发请求，导致CPU负载升高。
> 4. 死循环或者无限循环：代码中存在死循环或者无限循环的情况，导致CPU一直在执行循环，造成CPU负载升高。
> 5. 第三方库或应用的问题：某些第三方库或应用可能存在性能问题，导致CPU负载升高。可以检查是否有更新版本的库或应用可用，或者尝试禁用某些库或应用来排除问题。
>
> 针对以上问题，可以采取以下措施进行排查和解决：
>
> - 检查Tomcat的访问日志和线程堆栈，查看是否有异常请求或者长时间运行的线程。
> - 检查Tomcat的配置参数，确保线程池、连接池等参数设置合理。
> - 检查应用代码，查找是否存在死循环或者无限循环的情况。

- 使用性能监控工具，分析CPU占用高的线程和方法，定位性能瓶颈。
- 更新或替换可能存在性能问题的第三方库或应用。

学习指引：[排查tomcat服务器CPU使用率过高

18. 我们的 Redis 缓存怎么保证和数据库中数据的一致性?

解析：

参考回答：

1. **先更新数据库，再更新缓存**：
   - 当需要更新数据时，首先更新数据库。
   - 然后，删除或更新对应的Redis缓存项。
   - 这种方法的优点在于操作直观且简单。然而，如果第二步更新缓存失败，可能会导致数据不一致。
2. **先删除缓存，再更新数据库**：
   - 当需要更新数据时，首先删除Redis缓存中的对应项。
   - 然后，更新数据库。
   - 这种方法的缺点是，在删除缓存和更新数据库之间，如果有其他线程读取数据，可能会读取到旧的、已经从缓存中删除的数据，从而导致短暂的数据不一致。
3. **延时双删策略**：
   - 在更新数据库之前，先删除Redis缓存。
   - 更新数据库后，等待一段时间（这个时间通常是读操作可能的最长耗时，包括Redis主从同步、网络耗时等），然后再次删除Redis缓存。
   - 这种方法可以有效解决在更新过程中其他线程读取旧数据的问题。
4. **设置缓存过期时间**：
   - 为Redis缓存项设置一个合理的过期时间。这样，即使出现不一致的情况，缓存中的数据也会在一段时间后自动失效，从而确保最终一致性。
5. **使用消息队列保证顺序**：
   - 将数据库更新和缓存更新的操作放入消息队列中，确保它们按照正确的顺序执行。
   - 这种方法可以确保操作的原子性，但可能会增加系统的复杂性和延迟。
6. **分布式锁**：
   - 在更新数据库和缓存的过程中，使用分布式锁来确保操作的原子性。
   - 但需要注意的是，过度使用分布式锁可能会导致性能问题。
7. **读写分离**：
   - 在某些场景中，可以将读操作和写操作分离到不同的服务或数据库中。例如，写操作更新主数据库，而读操作从Redis缓存或只读副本数据库中获取数据。
8. **应用层补偿**：
   - 在应用层实现补偿机制，例如监听数据库的更新事件，并在必要时主动更新或删除Redis缓存。

学习指引：如何保障数据库和redis缓存的一致性

19. Redis 什么情况会导致读写性能突然变慢?

解析：

考察Redis基础，属于常考题，必会题。

参考回答：

> Redis的读写性能突然变慢可能由多种因素导致。以下是一些常见的原因：
>
> 1. **内存不足**：当Redis使用的内存达到其上限时，操作系统可能会开始使用交换分区（swap），这会导致Redis的读写操作变慢。此外，如果Redis实例运行的机器内存不足，也可能导致性能下降。
> 2. **网络延迟**：网络问题，如网络IO压力大或客户端使用短连接与Redis相连，都可能导致读写性能下降。短连接需要频繁地建立和关闭连接，增加了额外的开销。
> 3. **复杂命令或查询**：使用复杂度高的命令或一次性查询全量数据会增加Redis的处理时间，导致性能下降。
> 4. **大键（bigkey）操作**：操作包含大量元素或占用大量内存空间的键（bigkey）会导致性能问题。例如，删除、修改或查询bigkey时，Redis需要消耗更多的CPU和内存资源。
> 5. **大量键集中过期**：当大量键在相近的时间点集中过期时，Redis需要处理大量的过期事件，这可能导致性能突然下降。
> 6. **数据持久化**：当Redis数据量较大时，无论是生成RDB快照还是进行AOF重写，都会导致fork耗时严重，从而影响读写性能。此外，如果AOF的写回策略设置为always，那么每个操作都需要同步刷回磁盘，这也会增加写操作的延迟。
> 7. **CPU绑定不合理**：如果Redis实例的进程绑定到不合适的CPU核上，可能会导致性能下降。同样，如果Redis实例运行机器上开启了透明内存大页机制，也可能影响性能。
> 8. **硬件问题**：硬件故障或性能瓶颈，如磁盘I/O速度较慢、CPU性能不足等，也可能导致Redis读写性能变慢。

学习指引：Redis变慢的五大原因以及排查方法

20. 单个 Redis 中有热 Key，压力特别大，怎么解决?

解析：
考察Redis基础，属于常考题，必会题。
参考回答：

> 1. **分离热 Key**：
>   - 将热 Key 分离出来，存储在一个独立的 Redis 实例或集群中，以减轻主 Redis 实例的负载。
>   - 可以使用 Redis 的哈希标签（hash tag）功能，确保与热 Key 相关的数据也存储在同一实例或分片上，以保持数据一致性。
> 2. **使用更高效的数据结构**：
>   - 如果热 Key 对应的是复杂的数据结构（如哈希表、列表等），考虑是否可以使用更高效的数据结构或编码方式。
>   - 例如，对于频繁更新的列表，可以考虑使用 Redis 的有序集合（sorted set）或跳表（skip list）数据结构。
> 3. **缓存热点数据**：
>   - 在应用层引入缓存机制，将热 Key 的数据缓存到本地缓存（如 Memcached）中，减少对 Redis 的访问。
>   - 当本地缓存中的数据过期或不存在时，再从 Redis 中获取数据。
> 4. **分布式缓存**：
>   - 如果热 Key 的数据量大到单个 Redis 实例无法承载，可以考虑使用分布式缓存方案。
>   - 将热 Key 的数据分散到多个 Redis 实例或分片中，通过分片键（sharding key）进行路由，实现负载均衡。
> 5. **使用读写分离**：
>   - 对于读操作特别频繁的热 Key，可以考虑使用读写分离架构，将读请求和写请求分散到不同的 Redis 实例上。

    - 写请求仍然发送到主 Redis 实例，而读请求可以发送到从 Redis 实例或从 Redis 集群中的多个节点。
6. **使用 Redis 集群**：
    - 如果单个 Redis 实例已经无法满足需求，可以考虑使用 Redis 集群进行水平扩展。
    - Redis 集群可以自动将数据分散到多个节点上，实现负载均衡和高可用性。

学习指引：Redis中什么是热Key问题？如何解决热Key问题？

21. Redis 主节点宕机后，怎么恢复数据，怎么产生新的主?

解析：

考察Redis集群的稳定性，属于常考题，必会题。

参考回答：

在Redis主从架构中，当主节点宕机后，可以通过一系列机制来恢复数据并产生新的主节点，以保证数据的可用性和一致性。具体过程如下：

1. **故障检测**：从节点会定期向主节点发送心跳检测包（通常是PING命令），以确认主节点的状态。如果在设定的时间内没有收到主节点的响应（PONG回复），从节点会认为主节点已经宕机。
2. **选举新的主节点**：在从节点中，会按照优先级、复制偏移量等因素选举出一个新的主节点。优先级高的节点更容易被选为新的主节点，如果优先级相同，则比较复制偏移量，偏移量大的节点（即数据更完整的节点）更有可能成为新的主节点。
3. **数据同步**：新的主节点一旦选举出来，其他的从节点会开始向新的主节点发送SYNC或PSYNC命令，进行数据的同步。如果是第一次同步，会使用SYNC命令进行全量同步；如果是后续同步，由于已经部分同步过数据，会使用PSYNC命令进行增量同步，只同步缺失的部分数据。
4. **客户端重定向**：在故障转移过程中，客户端可能会收到一些错误响应，因为原来的主节点已经不可用。此时，客户端需要重新连接到新的主节点，或者通过代理层（如Redis Sentinel）来自动处理这种重定向。

在Redis Sentinel模式下，这一过程是自动的。Sentinel会监控主从节点的状态，当检测到主节点故障时，会自动执行故障转移，选举新的主节点，并通知客户端更新连接信息。

学习指引：Redis主节点宕机，如何处理？

22. Redis 的内存淘汰策略了解吗?

解析：

考察Redis内存淘汰策略，属于常考题，必会题。

参考回答：

1. **noeviction**：这是默认的淘汰策略。当 Redis 内存使用达到上限时，它不会淘汰任何数据，而是直接拒绝新的写请求（除了 DEL 和一些特定的命令）。读请求仍然会正常处理。
2. **volatile-ttl**：这个策略会优先淘汰那些设置了过期时间且剩余存活时间（TTL）较短的键。
3. **volatile-random**：这个策略会随机淘汰那些设置了过期时间的键。
4. **volatile-lru**：这个策略会淘汰那些设置了过期时间且最久未使用的键（使用 LRU 算法，即最近最少使用）。
5. **volatile-lfu**：这个策略会淘汰那些设置了过期时间且最少使用的键（使用 LFU 算法，即最近最不常用）。
6. **allkeys-lru**：这个策略会淘汰整个数据集中最久未使用的键（使用 LRU 算法）。

7. **allkeys-random**：这个策略会随机淘汰数据集中的任意键。
8. **allkeys-lfu**：这个策略会淘汰整个数据集中最少使用的键（使用 LFU 算法）。

学习指引：Redis 内存淘汰策略 （史上最全）

更多面经直通车

原贴连接

15.【美团】美团一面，八股盛宴|0329

1. cookie和session的区别?
2. 如何实现session共享，你会怎么做?
3. 反向代理是起什么作用的?
4. 数据库主从复制
5. 深拷贝和浅拷贝的区别?
6. 用过哪些深拷贝的方法?
7. GC分几种
8. minGC和FullGC有什么区别?
9. 消息队列起什么作用?
10. MySQL的隔离级别有哪些?
11. 默认是什么隔离级别?
12. 可重复读是什么概念？读未提交？串行化?
13. MySQL中的索引树是怎么维护的?
14. 什么叫B+树?
15. 为什么要采用这种数据结构?
16. Linux的命令平时用到哪些?
17. Linux中查看某个进程占用率较高的话，用哪个命令?
18. 我们想查找一个日志文件，空指针异常，查这个关键字用哪个命令查找？（不清楚）
19. 设计模式接触过哪些?

总结：自己Java基础不行，还需要继续巩固，多思考为什么要这么做，继续学下去

分享者thehou

面经专栏直通车，欢迎订阅

面经专栏下载， 点击跳转

1. cookie和session的区别?

参考回答：

Cookie和Session在多个方面存在显著的区别，主要体现在以下几个方面：

1. **存储位置**：

- Cookie的数据存储在客户端的浏览器中，用户可以通过浏览器工具查看和修改cookie信息。
- Session的数据则存放在服务器上，用户无法直接访问服务器上的session数据。

2. **存储容量与类型**：

- Cookie的存储容量相对较小，通常限制在4KB以内，并且只能存储字符串类型的数据。
- Session的存储容量没有明确的限制（但考虑到服务器性能，通常不建议存放过多数据），并且它可以存储任意类型的数据。

3. **有效期与生命周期**：

- Cookie的有效期可以在设置时指定，只要不超过设置的过期时间，它可以长期保存在客户端。
- Session的生命周期则通常较短，它会在一定的操作时间（如30分钟）后失效，并且在用户关闭浏览器或会话结束时，Session数据会被清除。

4. **安全性**：

- 由于Cookie存储在客户端，存在被第三方截获或篡改的风险，因此安全性相对较低。攻击者可以通过分析或伪造Cookie来欺骗系统。
- Session数据存储在服务器上，相对更加安全，不容易被攻击者直接获取或修改。

5. **跨域支持**：

- Cookie支持跨域名访问，即可以在不同的域名之间共享。
- Session则通常与特定的客户端和服务器端关联，不支持跨域访问。

6. **对服务器压力**：

- Cookie不占用服务器资源，因为数据存储在客户端。
- Session则需要在服务器上存储数据，因此对服务器的资源占用和性能有一定影响。

Cookie和Session都是在Web开发中用来跟踪用户会话状态的机制，区别如下：

1. **存储位置**：
    - Cookie: 存储在用户浏览器中，以文本文件的形式保存在用户计算机上。
    - Session: 存储在服务器端，通常以一种特殊的数据结构（如哈希表）保存在服务器的内存中或者持久化存储（如数据库）中。
2. **安全性**：
    - Cookie: 可以在用户浏览器上被查看和修改，因此相对不太安全。但是可以通过设置Cookie的属性，如HTTPOnly和Secure，来提高安全性。
    - Session: 存储在服务器端，相对于Cookie更安全，因为用户无法直接查看或修改会话数据。
3. **容量**：
    - Cookie: 由于存储在用户浏览器中，单个Cookie的容量通常有限制（一般约为4KB），因此它适合存储少量的数据。
    - Session: 存储在服务器端，通常没有明确的容量限制，但会受到服务器资源的限制。
4. **生命周期**：
    - Cookie: 可以设置过期时间，也可以是会话Cookie（浏览器关闭后自动删除）。
    - Session: 通常在用户关闭浏览器或者长时间不活动后过期，但也可以设置特定的过期时间。
5. **跨页面访问**：
    - Cookie: 可以跨页面访问，即使用户跳转到其他页面，Cookie仍然会被发送到服务器。
    - Session: 也可以跨页面访问，但需要在页面之间传递会话标识符（如Session ID）来实现。

学习指引：[Cookie和Session的区别（面试必备）]

2. 如何实现session共享?

参考回答：

实现session共享有多种方法，主要取决于你的应用架构和所使用的技术栈。以下是一些常见的方法来实现session共享：

1. **数据库存储**：
    - 将session数据存储在数据库中，而不是仅仅存储在内存中。这样，无论请求发送到哪个服务器，都可以通过查询数据库来获取session数据。
    - 这种方法需要设计好数据库表结构，并编写相应的存储和检索逻辑。
2. **缓存系统**：
    - 使用像Redis或Memcached这样的缓存系统来存储session数据。每个服务器都可以访问这个共享的缓存系统来获取和更新session数据。
    - 这种方法通常比数据库访问更快，并且能更好地处理大量并发请求。
3. **粘性session**：
    - 通过负载均衡器实现粘性session（或称为持久连接）。负载均衡器根据某种算法（如基于IP哈希）将用户的请求始终路由到同一台服务器。
    - 这种方法不需要跨服务器共享session数据，但可能导致服务器负载不均衡。
4. **Session复制**：
    - 当一个服务器上的session数据发生变化时，将这些变化复制到其他服务器上。这通常通过消息队列（如RabbitMQ）或分布式事件系统（如Apache Kafka）实现。
    - 这种方法的缺点是可能存在数据同步延迟，且在大规模部署中可能不太实用。
5. **中央认证服务**：
    - 使用如OAuth、OpenID Connect或SAML等中央认证服务来管理用户的认证和授权信息。
    - 用户登录后，认证服务会颁发一个令牌（token），客户端可以使用这个令牌来访问资源，而不需要依赖session数据。
6. **无状态应用**：
    - 设计应用为无状态，即不依赖服务器端的session数据来识别用户。
    - 可以通过JWT（JSON Web Tokens）或其他令牌机制来实现用户身份验证和授权。
7. **使用框架和库**：
    - 某些Web框架和库提供了内置的session共享机制。例如，Spring Session可以与Redis或数据库集成来实现session共享。

学习指引：[如何实现session跨服务器共享]

3. 反向代理是起什么作用的?

参考回答：

反向代理是一种位于客户端和后端服务器之间的代理服务器模型，它接收来自客户端的请求，并将其转发到后端服务器，然后将响应返回给客户端。反向代理在多个方面发挥着重要的作用：

1. **负载均衡**：反向代理通过将请求分发到多个后端服务器，平衡负载，从而减少单个服务器的负担，提高系统的可伸缩性。对于流量较高的网站，这种分发机制可以确保服务器容量得到充分利用，以处理大量请求。如果某台服务器过载并出现故障，反向代理还可以将流量重定向至其他在线服务器，确保服务的连续性和稳定性。
2. **安全性和访问控制**：反向代理通过实现身份验证、授权和防火墙功能，增强了系统的安全性。它可以拦截所有传入请求，为后端服务器提供更高层级的保护。通过阻止来自特定IP地址的可疑流量，反向代理有助于防止恶意访问者滥用网页服务器。此外，反向代理还可以隐藏后端服务器的真实地址，进一步提高系统的安全性。
3. **缓存数据**：反向代理可以缓存经常请求的数据，减少对后端服务器的访问次数，提高系统的性能。对于需要存储大量信息流数据的大型用户，缓存机制可以有效降低网站服务器的负载，提高网站的响应速度和用户体验。
4. **服务治理**：通过反向代理，管理员可以监控和管理后端服务器的状态，包括健康检查、服务降级等，确保系统的稳定性和可靠性。

学习指引：什么是反向代理？

4. MySQL数据库主从复制原理?

参考回答：

> MySQL数据库的主从复制原理涉及主服务器（Master）和从服务器（Slave）之间的数据同步过程。以下是这个过程的详细解释：
>
> **主服务器（Master）操作**：
>
> 1. **二进制日志（Binary Log）记录**：当主服务器上的数据发生更改时（如INSERT、UPDATE或DELETE操作），这些更改会被记录到主服务器的二进制日志（通常称为binlog）中。二进制日志是MySQL用来记录数据库更改的日志文件，它包含了描述数据库更改的"事件"。
> 2. **发送二进制日志**：从服务器会定期向主服务器请求最新的二进制日志事件。一旦从服务器请求，主服务器会发送从上次同步点之后的二进制日志事件到从服务器。
>
> **从服务器（Slave）操作**：
>
> 1. **接收和写入中继日志（Relay Log）**：从服务器接收到主服务器发送的二进制日志事件后，首先将这些事件写入到本地的中继日志中。中继日志是从服务器用来临时存储从主服务器接收到的二进制日志事件的日志文件。
> 2. **执行SQL语句**：从服务器的一个SQL线程会读取中继日志中的事件，并将这些事件转换成对应的SQL语句，然后在从服务器上执行这些SQL语句。这样，从服务器上的数据就被更新为与主服务器一致。
>
> **复制延迟与同步**：
>
> 在实际运行中，由于网络延迟、从服务器负载等原因，从服务器的数据可能会稍微落后于主服务器。这种延迟是主从复制的一个常见问题，需要通过监控和管理来确保数据的及时同步。
>
> 此外，MySQL还提供了多种复制模式，如异步复制、半同步复制等，这些模式在数据同步的实时性和一致性方面有所不同，用户可以根据实际需求选择合适的复制模式。

学习指引：超详细！MySQL如何实现主从复制和读写分离

5. Java 的深拷贝和浅拷贝的区别?

参考回答：

> 在 Java 中，深拷贝和浅拷贝主要涉及到对象复制时处理对象属性和引用关系的方式。这两种拷贝方式的主要区别如下：
>
> **浅拷贝（Shallow Copy）**
>
> 浅拷贝是创建一个新对象，并复制原对象中的非引用类型数据（如 int、double、float 等基本数据类型），但对于引用类型数据（如对象、数组等），只复制其引用而不复制引用的对象。换句话说，浅拷贝对于非引用类型字段进行值传递，对引用类型字段进行引用传递。因此，如果原对象的引用类型字段被修改，浅拷贝后的对象也会受到影响。
>
> **深拷贝（Deep Copy）**
>
> 深拷贝则不同，它不仅创建一个新对象，并复制原对象中的非引用类型数据，还会递归地复制原对象中的引用类型数据。也就是说，对于原对象的引用类型字段，深拷贝会创建一个新的对象，并复制原引用对象的内容到新对象中。因此，如果原对象的引用类型字段被修改，深拷贝后的对象不会受到影响。

学习指引：Java深入理解深拷贝和浅拷贝区别

6. Java 有哪些深拷贝的方法?

参考回答：

> 在 Java 中，实现深拷贝的方法有多种：
>
> 1. **重写 `clone()` 方法**：Java 中的 `Object` 类提供了一个 `clone()` 方法，用于创建对象的副本。但是，默认情况下，`clone()` 方法执行的是浅拷贝。为了实现深拷贝，需要重写 `clone()` 方法，并在其中对对象的属性进行递归拷贝。
> 2. **使用序列化和反序列化**：Java 中的序列化和反序列化可以实现对象的深拷贝。通过将对象写入字节流，然后再从字节流中读取出来，就可以创建一个新的对象，而不是简单地复制引用。这种方式要求被复制的类及其所有被引用的类都必须实现 `Serializable` 接口。
> 3. **使用第三方库**：有些第三方库提供了深拷贝的功能，比如 Apache Commons Lang 和 Gson。这些库通常提供了更灵活和强大的深拷贝实现，可以处理更复杂的对象结构和关系。

学习指引：Java如何对一个对象进行深拷贝?

7. Java 的 GC分几种?

参考回答：

> Java的垃圾回收（GC）主要可以分为以下几种：
>
> 1. **按线程数分**：
>   - **串行垃圾回收器**：在同一时间段内只允许有一个CPU用于执行垃圾回收操作，此时工作线程被暂停，直至垃圾收集工作结束。
>   - **并行垃圾回收器**：使用多个线程同时执行垃圾回收工作，以提高垃圾回收的效率。
> 2. **按工作模式分**：
>   - **并发式垃圾回收器**：垃圾回收线程与应用程序线程同时运行，可以在垃圾回收的同时进行应用程序的执行。
>   - **独占式垃圾回收器（Stop the World）**：一旦运行，就停止应用程序中的所有用户线程，直到垃圾回收过程完全结束。
> 3. **按碎片处理方式分**：
>   - **压缩式垃圾回收器**：在垃圾回收后，会对内存空间进行整理，消除内存碎片。
>   - **非压缩式垃圾回收器**：在垃圾回收后，不整理内存空间，可能产生内存碎片。
> 4. **按工作的内存区间分**：
>   - **年轻代垃圾回收器**：主要负责回收新生代中的垃圾对象。
>   - **老年代垃圾回收器**：主要负责回收老年代中的垃圾对象。
> 5. **按垃圾回收种类分**：
>   - **Partial GC**：部分收集模式，可以是Young GC（只收集年轻代的GC）或Old GC（只收集老年代的GC）。
>   - **Mixed GC**：收集整个年轻代以及部分老年代的GC，例如G1收集器就有这种模式。
>   - **Full GC**：收集整个堆和方法区，堆是垃圾回收的主要区域，方法区很少会被回收。
>
> 在Java虚拟机（JVM）中，程序计数器、虚拟机栈、本地方法栈都是随线程而生随线程而灭，栈帧随着方法的进入和退出做入栈和出栈操作，实现了自动的内存清理。而Java堆内存被划分为新生代和年老代两部分，新生代主要使用复制和标记-清除垃圾回收算法，年老代主要使用标记-整理垃圾回收算法。

学习指引：Java常见的GC有哪些?

8. Java 的 minGC和FullGC有什么区别?

参考回答：

> MinGC（通常指的是Minor GC或Young GC）和Full GC（也称为Major GC或Old Generation GC）是Java虚拟机（JVM）中两种不同的垃圾回收（GC）操作，它们主要针对的是堆内存中的不同部分，并且在执行时有着不同的特点。
>
> **Minor GC（Young GC）**：
>
> - 主要针对新生代（Young Generation）进行垃圾回收。新生代是堆内存中的一个区域，主要用于存放新创建的对象。
> - 由于新生代中的对象生命周期通常较短，因此Minor GC的执行频率通常较高，但每次回收的耗时相对较短。
> - Minor GC采用复制算法，将存活对象从一个内存区域复制到另一个内存区域，同时清理掉不再使用的对象。
>
> **Full GC（Major GC或Old Generation GC）**：
>
> - 涉及整个Java堆（Heap）的回收，包括年轻代、老年代（Old Generation），以及方法区（也称为永久代在JDK 7或之前的版本，或者元空间在JDK 8以后的版本）。
> - Full GC的执行时间通常较长，因为它需要扫描整个堆内存，识别不再存活的对象并进行清理。
> - Full GC可能导致系统停顿，因为在垃圾回收过程中，所有的应用线程都会被暂停，直到GC完成。
> - Full GC的触发条件通常与老年代的空间不足或需要整理碎片化的内存有关。
>
> Minor GC和Full GC的主要区别在于它们针对的堆内存区域不同，执行频率和耗时也不同。Minor GC主要关注新生代的垃圾回收，而Full GC则涉及整个堆内存的回收。在设计Java应用时，需要尽量减少Full GC的次数，以保证系统的性能。同时，可以通过优化代码、减少内存泄漏和不必要的对象引用等方式来降低垃圾回收的频率和耗时。

学习指引：JVM: GC过程总结(minor GC 和 Full GC)

9. 消息队列用途有哪些?

参考回答：

> 消息队列是一种重要的中间件技术，用于在分布式系统中实现异步通信，它的用途非常广泛。以下是消息队列的一些常见用途：
>
> 1. **解耦**：消息队列可以将应用程序解耦，发送者和接收者之间不直接通信，而是通过消息队列进行通信。这样可以降低系统间的耦合度，提高系统的灵活性和可维护性。
> 2. **异步通信**：发送者发送消息到消息队列后即可立即返回，不需要等待接收者处理完消息。这种方式可以提高系统的响应速度和吞吐量。
> 3. **削峰填谷**：消息队列可以作为流量控制的工具，当系统负载较高时，可以将请求放入消息队列中，等待系统负载下降后再处理。这样可以平滑处理系统的高峰流量，保证系统的稳定性。
> 4. **异步任务处理**：将需要异步执行的任务发送到消息队列中，由后台工作线程消费消息并执行任务。这样可以提高系统的并发能力和处理效率。
> 5. **日志收集**：将系统产生的日志消息发送到消息队列中，由日志消费者进行处理、分析、存储或者展示。这样可以集中管理日志，并且降低日志处理对系统性能的影响。
> 6. **事件驱动架构**：消息队列可以用于构建事件驱动的架构，不同组件之间通过发送和接收消息来进行通信。这种架构可以实现系统的松耦合和高内聚，提高系统的可扩展性和可维护性。
> 7. **分布式系统集成**：在分布式系统中，消息队列可以用于不同系统之间的数据传输和集成，实现系统之间的解耦和通信。
> 8. **顺序性保证**：某些消息队列提供了消息的顺序性保证，可以确保消息按照发送的顺序进行处理，适用于一些有序性要求的场景。

学习指引：新手也能看懂，消息队列其实很简单

10. MySQL的隔离级别有哪些?

参考回答：

> MySQL提供了四种事务隔离级别，分别是读未提交（Read Uncommitted）、读提交（Read Committed）、可重复读（Repeatable Read）和串行化（Serializable）。不同的隔离级别可以解决并发访问数据库时的各种问题，提供不同程度的隔离和数据一致性。
>
> 1. 读未提交（Read Uncommitted）：
>   - 最低的隔离级别，事务中的修改操作对其他事务是可见的，即未提交的修改可以被其他事务读取。
>   - 存在脏读（Dirty Read）问题，即一个事务读取到了另一个未提交事务的修改结果。
> 2. 读提交（Read Committed）：
>
> - - 事务中的修改操作在提交之前对其他事务是不可见的，只有已提交的修改才能被其他事务读取。
>   - 解决了脏读问题，但可能存在不可重复读（Non-repeatable Read）问题，即在同一事务内多次读取同一数据可能会得到不同的结果。
>
> 3. 可重复读（Repeatable Read）：
>
> - - 保证在同一事务内多次读取同一数据时，得到的结果始终一致。
>   - 解决了不可重复读问题，但可能存在幻读（Phantom Read）问题，即在同一事务内多次执行查询，得到的结果集不一致。
>
> 4. 串行化（Serializable）：
>
> - - 最高的隔离级别，完全隔离事务，确保事务串行执行，避免并发问题。
>   - 解决了幻读问题，但在高并发情况下会导致性能下降，因为事务需要串行执行。

学习指引：MySQL的4种事务隔离级别你还不清楚吗?

11. MySQL默认是什么隔离级别?

参考回答：

> MySQL数据库的默认隔离级别是**可重复读（Repeatable Read）**。
>
> 在这个隔离级别下，同一个事务中多次读取同样记录的结果是一致的。这主要是因为MySQL使用了多版本并发控制（MVCC）机制来实现并发控制，避免了脏读和不可重复读的问题。但需要注意的是，可重复读隔离级别并不能完全避免幻读问题，即在事务执行过程中，其他事务可能插入或删除某些行，导致最终读取到的数据行数发生了变化。
>
> MySQL也支持其他三种隔离级别：读未提交、读已提交和串行化。
>
> 这些隔离级别在数据一致性和并发性能之间提供了不同的权衡。在实际应用中，可以根据具体的业务需求和性能要求来选择合适的隔离级别。如果需要更高的并发性和性能，可以考虑使用较低的隔离级别，如读已提交；而如果对数据一致性有更高的要求，可以选择更高的隔离级别，如串行化。但请注意，高隔离级别可能会带来更大的性能开销。

学习指引：mysql默认事务隔离级别是什么?

12. 可重复读是什么概念？读未提交？串行化?

参考回答：

> 在MySQL中，不同的隔离级别决定了事务如何与其他事务并发执行以及数据的一致性和可见性。以下是关于“可重复读”、“读未提交”和“串行化”这三种隔离级别的概念解释：
>
> **可重复读（Repeatable Read）**
>
> **可重复读**是MySQL的默认隔离级别。在这个级别下，一旦事务开始，该事务内部对同一数据的多次读取结果都是一致的，即使其他事务在此期间修改了该数据并提交。这是通过多版本并发控制（MVCC）实现的，每个事务都可以看到一个数据的一致快照，就像是在事务开始时拍摄的照片一样。这确保了同一事务中的多次读取操作不会受到其他事务的影响。
>
> 然而，需要注意的是，虽然同一事务内的多次读取是一致的，但不同事务之间仍然可以并发修改数据，这可能导致“幻读”现象，即一个事务在读取某个范围内的记录时，另一个并发事务插入或删除了这个范围内的记录，导致前一个事务在后续的读取中看到了不同的行数。
>
> **读未提交（Read Uncommitted）**
>
> **读未提交**是最低的隔离级别。在这个级别下，一个事务可以读取到其他事务尚未提交的数据。这可能导致脏读（Dirty Read）问题，即读取到了未经验证的数据。如果其他事务回滚了这些未提交的数据，那么当前事务读取到的数据就是无效的或“脏”的。因此，读未提交隔离级别下，数据的一致性和可靠性无法得到保证。
>
> **串行化（Serializable）**
>
> **串行化**是最高的隔离级别。在这个级别下，事务是串行执行的，即一个事务必须等待前一个事务完成后才能开始执行。这确保了每个事务都能看到一个一致的数据视图，从而避免了脏读、不可重复读和幻读的问题。然而，由于事务是串行执行的，这大大降低了系统的并发性能，特别是在高并发场景下，可能导致性能瓶颈。
>
> 在选择隔离级别时，需要根据具体的应用场景和需求进行权衡。如果业务对数据一致性要求非常高，且可以牺牲一定的并发性能，那么可以选择串行化隔离级别。如果业务对并发性能有较高要求，且可以容忍一定程度的数据不一致性，那么可以选择较低的隔离级别，如可重复读或读已提交。

学习指引：数据库隔离级别------串行化，可重复读，读提交，读未提交

13. MySQL中的索引树是怎么维护的?

参考回答：

> MySQL中的索引树（主要是B+树）的维护涉及多个方面，包括插入、删除和更新操作。这些操作都需要确保索引树的平衡和高效性。以下是关于MySQL中索引树维护的一些关键点：
>
> 1. **插入操作**：
>   - 当向表中插入新数据时，MySQL需要在索引树中找到合适的位置插入新的索引条目。
>   - 如果插入的位置在叶子节点且有空余空间，直接插入。
>   - 如果叶子节点已满，需要进行分裂操作。分裂后，可能需要向上层节点插入新的分裂键。
>   - 为了保持树的平衡，可能需要进行一系列的分裂和重新平衡操作。
> 2. **删除操作**：
>   - 当从表中删除数据时，相应的索引条目也需要从索引树中删除。
>   - 如果删除的索引条目位于叶子节点，并且删除后该节点还有足够的条目，直接删除即可。
>   - 如果删除后叶子节点条目过少（例如，少于某个阈值），可能需要与相邻的兄弟节点合并，或者从父节点“借用”条目。
>   - 如果合并或借用操作导致父节点也需要调整，这个过程可能会向上层传播。
> 3. **更新操作**：

    - 更新操作在索引维护方面通常可以视为先删除旧条目，再插入新条目。
    - 如果更新的键没有改变，可能只需要更新叶子节点中的值部分。
    - 如果更新的键发生了改变，则需要在索引树中找到旧键的位置进行删除，并在新位置插入新键。
4. **索引树的优化和重建**：
    - 随着数据的插入、删除和更新，索引树可能会变得不那么紧凑或平衡，这可能导致性能下降。
    - 为了优化索引性能，MySQL提供了工具（如`OPTIMIZE TABLE`）来重建索引树，使其更加紧凑和平衡。
    - 重建索引通常涉及创建一个新的索引树，并将旧树中的数据逐步迁移到新树中。完成后，新树会替换旧树。
5. **并发控制**：
    - 在高并发环境下，多个事务可能同时尝试修改索引树。
    - MySQL使用锁机制（如行锁、表锁或元数据锁）来确保索引树的并发修改不会导致数据不一致。
    - 锁的使用和粒度取决于存储引擎（如InnoDB或MyISAM）和具体的配置。
6. **日志和恢复**：
    - 为了确保在系统崩溃或其他故障情况下能够恢复数据，MySQL使用重做日志（如InnoDB的redo log）来记录对索引树的修改。
    - 在系统启动时，MySQL可以读取这些日志并重新应用修改，以恢复索引树到一致的状态。

学习指引：MySQL：插入数据时，是如何维护好不同索引的B+树的

14. 什么叫B+树?

参考回答：

B+ 树是一种平衡树数据结构，通常用于实现有序的索引结构。它是 B 树的变种，在 B 树的基础上做了一些改进和优化。

B+ 树的特点包括：

1. **有序性**：B+ 树的所有叶子节点被链接成一个有序链表，这使得范围查询和区间扫描变得更加高效。
2. **平衡性**：B+ 树保持平衡，即所有叶子节点到根节点的距离相同，这确保了检索效率的稳定性。
3. **多路性**：B+ 树的每个节点可以拥有多个子节点，通常被称为“多路平衡”，这降低了树的高度，减少了磁盘 I/O 次数，提高了检索效率。
4. **非叶子节点仅存储索引信息**：与 B 树不同，B+ 树的非叶子节点只存储索引信息，而不存储实际的数据，这使得 B+ 树的节点更小，能够容纳更多的索引条目。
5. **叶子节点存储数据记录**：B+ 树的叶子节点存储实际的数据记录，这样可以避免在非叶子节点存储大量的数据，减少了索引的维护成本。

B+ 树通常被用于数据库系统中实现索引结构，特别是在关系型数据库管理系统（RDBMS）中，例如 MySQL、PostgreSQL 等。B+ 树的高效性和稳定性使得它成为了一种理想的索引结构，能够满足大多数数据库系统的需求。

学习指引：什么是B+树?

15. 为什么要采用B+树最为索引?

参考回答：

> B+ 树通常被用作数据库系统中的索引结构，而不是其他类型的树，主要基于以下几个原因：
>
> 1. **高效的检索和范围查询**：B+ 树的有序性和平衡性使得在进行查找、范围查询或区间扫描时非常高效。由于所有叶子节点被链接成一个有序链表，因此范围查询的性能非常好，可以快速地找到满足条件的数据记录。
> 2. **稳定的检索性能**：B+ 树的平衡性确保了树的高度相对较小，因此检索操作的性能是稳定的，不会因为数据量的增加而显著下降。这使得 B+ 树适用于大型数据库系统，能够处理大量数据并保持高效的检索性能。
> 3. **适用于磁盘存储**：B+ 树的设计考虑了磁盘 I/O 操作的特点，通过多路平衡的节点结构和叶子节点存储数据记录的方式，减少了磁盘 I/O 次数，提高了数据访问的效率。这使得 B+ 树非常适合在磁盘上存储大型数据集，并且能够有效地利用磁盘空间。
> 4. **适用于范围查询**：B+ 树的有序叶子节点链表和多路平衡的节点结构使得范围查询的性能非常好。这对于数据库系统中经常需要执行的范围查询操作来说是至关重要的，例如按时间范围或者按字母顺序查询数据。
> 5. **易于扩展和维护**：B+ 树的平衡性和多路性使得它们易于扩展和维护。当数据库中的数据量增加时，可以很容易地添加新的索引条目，并且不会显著影响整个树的结构。此外，B+ 树的节点结构简单，易于实现和维护。
>
> B+ 树具有高效的检索性能、稳定的性能表现、适用于磁盘存储、适用于范围查询以及易于扩展和维护等优点，因此被广泛应用于数据库系统中作为索引结构。

学习指引：面试官：为什么MySQL的索引要使用B+树，而不是其它树？比如B树？

16. Linux的命令平时用到哪些?

参考回答：

> 在日常使用 Linux 系统时，经常会使用以下一些常用的命令：
>
> 1. **文件和目录操作**：
>   - `ls`：列出当前目录下的文件和子目录。
>   - `cd`：切换工作目录。
>   - `pwd`：显示当前工作目录的路径。
>   - `mkdir`：创建新目录。
>   - `rm`：删除文件或目录。
>   - `cp`：复制文件或目录。
>   - `mv`：移动文件或目录。
>   - `touch`：创建空文件或更新文件的时间戳。
> 2. **文件查看和编辑**：
>   - `cat`：显示文件内容。
>   - `more` 或 `less`：分页查看文件内容。
>   - `head` 和 `tail`：分别显示文件的开头和结尾部分。
>   - `vi` 或 `vim`：文本编辑器，用于编辑文件。
> 3. **文件权限管理**：
>   - `chmod`：修改文件或目录的权限。
>   - `chown`：修改文件或目录的所有者。
>   - `chgrp`：修改文件或目录的所属组。
> 4. **系统信息查看**：
>   - `uname`：显示系统信息。
>   - `df`：显示磁盘空间使用情况。
>   - `free`：显示系统内存使用情况。
>   - `top`：实时显示系统资源占用情况。

- `ps`：显示当前进程信息。

5. **网络操作**：
   - `ping`：测试网络连接。
   - `ifconfig` 或 `ip`：显示和配置网络接口信息。
   - `ssh`：远程登录到其他主机。
   - `scp`：在本地和远程主机之间传输文件。
6. **压缩和解压缩**：
   - `tar`：打包和解压缩文件。
   - `gzip` 和 `gunzip`：压缩和解压缩文件。
7. **系统管理**：
   - `shutdown`：关闭系统。
   - `reboot`：重新启动系统。
   - `service` 或 `systemctl`：管理系统服务。
   - `cron`：定时执行任务。

学习指引：常用Linux命令及其作用

17. Linux中查看某个进程占用率较高的话，用哪个命令?

参考回答：

要查看 Linux 中某个进程的 CPU 和内存占用率较高的话，可以使用 `top` 命令。 `top` 命令是一个动态实时显示系统运行进程信息的命令行工具，可以列出当前系统中所有正在运行的进程，并按照 CPU 和内存占用率进行排序。

要查看某个进程的 CPU 和内存占用率，你可以按以下步骤操作：

1. 打开终端窗口。
2. 运行 `top` 命令。
3. 在 `top` 的交互界面中，你可以按下 `Shift + P` 来按 CPU 占用率排序，或者按下 `Shift + M` 来按内存占用率排序。这样就会把占用率较高的进程排到前面。
4. 找到的进程，查看其 CPU 和内存占用率以及其他相关信息。

此外，也可以使用 `htop` 命令，它是 `top` 命令的增强版，提供了更加友好和交互式的界面，并且支持鼠标操作。

18. 我们想查找一个日志文件，空指针异常，查这个关键字用哪个命令查找?

参考回答：

如果想在Linux系统中查找包含“空指针异常”关键字的日志文件，你可以使用`grep`命令。`grep`是一个强大的文本搜索工具，它可以搜索含有特定模式的行。

以下是如何使用`grep`命令来查找包含“空指针异常”关键字的日志文件：

```
grep -r "空指针异常" /path/to/logs/
```

这里的参数解释如下：

- `-r` 或 `-R`：递归搜索，即在指定目录及其子目录中搜索。
- `"空指针异常"`：你要搜索的关键字或模式。
- `/path/to/logs/`：你要搜索的目录路径。你需要替换为实际的日志文件所在目录。

如果你想忽略大小写，可以使用`-i`选项：

```
grep -ri "空指针异常" /path/to/logs/
```

如果你还想看到匹配行的行号，可以添加`-n`选项：

```
grep -rin "空指针异常" /path/to/logs/
```

如果你知道日志文件的具体名称或模式，你还可以结合使用`find`命令和`xargs`或`-exec`选项来更精确地搜索。例如，如果你知道所有的日志文件都是以`.log`结尾的，你可以这样做：

```
find /path/to/logs/ -name "*.log" -print0 | xargs -0 grep -i "空指针异常"
```

或者：

```
find /path/to/logs/ -name "*.log" -exec grep -i "空指针异常" {} +
```

这些命令会搜索`/path/to/logs/`目录及其子目录中所有以`.log`结尾的文件，并查找包含“空指针异常”的行。记得将`/path/to/logs/`替换为你的实际日志文件路径。

面经专栏直通车，欢迎订阅

面经专栏下载，点击跳转