
title 面经 categories:

- shengjunjie tags:
 -
-

2021.05.11 阿里JAVA研发工程师一面

面试流程（目前只记得这么多）

1.询问了项目的相关情况，主要包括介绍项目，以及自己目前开发做了哪些工作，你认为哪些比较困难，有什么收获。

2.讲一下静态**static**?静态方法能不能调用非静态的成员变量？为什么？

- 被**static**关键字修饰的方法或者变量不需要依赖于对象来进行访问，只要类被加载了，就可以通过类名去进行访问
- 在**static**方法内部不能调用非静态方法，反过来是可以的。而且可以在没有创建任何对象的前提下，仅仅通过类本身来调用**static**方法。
- 在静态方法中访问非静态方法/变量的话，那么如果在main方法中有下面一条语句：

```
MyObject.print2();
```

此时对象都没有，str2根本就不存在，所以就会产生矛盾了。同样对于方法也是一样，由于你无法预知在print1方法中是否访问了非静态成员变量，所以也禁止在静态成员方法中访问非静态成员方法。而对于非静态成员方法，它访问静态成员方法/变量显然是毫无限制的。

- 由于其被所有的对象所共享（当且仅当类初次加载时才会被初始化），其常常用于写工具类的工具方法以及单例模式。

```
//单例模式
public class Singleton{
    private static volatile Singleton instace=null;
    private Singleton(){

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

3.讲一下多态？

多态是同一个行为具有多个不同表现形式或形态的能力,就是同一个接口，使用不同的实例而执行不同操作;父类应用指向子类对象(左父右子)，可以使程序有良好的扩展，并可以对所有类的对象进行通用处理。常用于配合设计模式

```
//工厂模式
public interface Shape {
    void draw();
}

public class Rectangle implements Shape {

    @Override
    public void draw() {
        System.out.println("Inside Rectangle::draw() method.");
    }
}

public class Square implements Shape {

    @Override
    public void draw() {
        System.out.println("Inside Square::draw() method.");
    }
}

public class Circle implements Shape {

    @Override
    public void draw() {
        System.out.println("Inside Circle::draw() method.");
    }
}

public class ShapeFactory {

    //使用 getShape 方法获取形状类型的对象
    public Shape getShape(String shapeType){
        if(shapeType == null){
            return null;
        }
        if(shapeType.equalsIgnoreCase("CIRCLE")){
            return new Circle();
        } else if(shapeType.equalsIgnoreCase("RECTANGLE")){
            return new Rectangle();
        } else if(shapeType.equalsIgnoreCase("SQUARE")){
            return new Square();
        }
        return null;
    }
}
```

```
}

public class FactoryPatternDemo {

    public static void main(String[] args) {
        ShapeFactory shapeFactory = new ShapeFactory();

        //获取 Circle 的对象，并调用它的 draw 方法
        Shape shape1 = shapeFactory.getShape("CIRCLE");

        //调用 Circle 的 draw 方法
        shape1.draw();

        //获取 Rectangle 的对象，并调用它的 draw 方法
        Shape shape2 = shapeFactory.getShape("RECTANGLE");

        //调用 Rectangle 的 draw 方法
        shape2.draw();

        //获取 Square 的对象，并调用它的 draw 方法
        Shape shape3 = shapeFactory.getShape("SQUARE");

        //调用 Square 的 draw 方法
        shape3.draw();
    }
}
```

4.讲一下抽象类与接口的区别？

抽象类

```
abstract class Demo {

    abstract void method1();

    abstract void method2();

    ...

}
```

接口类

```
interface Demo {

    void method1();

    void method2();

}
```

```
...
}
```

- 在abstract class方式中，Demo可以有自己的数据成员，也可以有非abstract的成员方法，而在interface方式实现中，Demo只能有静态的不能被修改的数据成员（也就是必须是static final的，不过在interface中一般不定义数据成员），所有的成员方法都是抽象的。从某种意义上说，interface是一种特殊的abstract class。
- 首先，抽象类在Java中代表的是继承关系，一个类只能使用一次继承关系。但是，一个类却可以实现多个接口。其次，在抽象类的定义中，我们可以赋予方法的默认行为。但是在接口的定义中，方法却不能拥有默认行为。不过在jdk1.8中可以使用default关键字实现默认方法。

```
interface InterfaceA {
    default void foo() {
        System.out.println("InterfaceA foo");
    }
}
```

总结:

- 抽象类和接口都不能直接实例化，如果要实例化，抽象类变量必须指向实现所有抽象方法的子类对象，接口变量必须指向实现所有接口方法的类对象。
- 抽象类要被子类继承，接口要被类实现。
- 接口里定义的变量只能是公共的静态的常量，抽象类中的变量是普通变量。
- 抽象类里可以没有抽象方法。
- 接口可以被类多实现（被其他接口多继承），抽象类只能被单继承。
- 接口中没有 this 指针，没有构造函数，不能拥有实例字段（实例变量）或实例方法。
- 抽象类不能在Java 8 的 lambda 表达式中使用。

5.讲一下线程的并发与调度？

- (并行指在同一时间点同时执行)并发是指在同一时间片段同时执行，多线程只能并发执行，实际还是顺执行，只是在同一时间片段，假似同时执行，cpu可以按时间切片执行，单核cpu同一个时刻只支持一个线程执行任务，多线程并发事实上就是多个线程排队申请调用cpu，cpu处理任务速度非常快，所以看上去多个线程任务说并发处理。
- 在运行池中，会有多个处于就绪状态的线程在等待CPU，JAVA虚拟机的一项任务就是负责线程的调度，线程调度是指按照特定机制为多个线程分配CPU的使用权。

1. 分时调度模式: 是指让所有的线程轮流获得cpu的使用权,并且平均分配每个线程占用的cpu的时间片。
2. 抢占式调度模式: JAVA虚拟机采用抢占式调度模式,是指优先让可运行池中优先级高的线程占用CPU,如果可运行池中的线程优先级相同,那就随机选择一个线程,使其占用CPU.处于运行状态的线程会一直运行,直至它不得不放弃CPU. 调整各个线程的优先级 让处于运行状态的线程进入block调用Thread.sleep()方法 让处于运行状态的线程进入Runnable调用Thread.yield()方法 让处于运行状态的线程调用另一个线程的join()方法等等 线程切换：不是所有的线程切换都需要进入内核模式 ##### (注意：sleep()方法是Thread类里面的，主要的意义就是让当前线程停止执行，让出cpu给其他的线程，但是不会释放对象锁资源以及监控的状态，当指定的时间到了之后又会自动恢复运行状态。wait()方法是Object类里面的，主要的意义就是

让线程放弃当前的对象的锁，进入等待此对象的等待锁定池，只有针对此对象调用notify方法后本线程才能够进入对象锁定池准备获取对象锁进入运行状态。）

6.讲一下乐观锁与悲观锁的区别？

- 乐观锁：是应用系统层面和数据的业务逻辑层次上的（实际上并没有加锁，只是一种锁思想），利用程序处理并发，它假定当某一个用户去读取某一个数据的时候，其他的用户不会来访问修改这个数据，但是在最后进行事务的提交的时候会进行数据的检查，以判断在该用户的操作过程中，没有其他用户修改了这个数据。乐观锁的实现大部分都是基于版本控制实现的，除此之外，还有CAS操作实现
- 悲观锁：每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会block直到它拿到锁。传统的关系型数据库里边就用到了很多这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁。

7.讲一下Spring中IOC与AOP的原理，spring的实现方式以及Spring的好处？

1. IOC的原理？好处？原理：“控制反转”：借助于“第三方”实现具有依赖关系的对象之间的解耦，通过容器，使两个对象之间失去直接的联系，假设当对象A运行到需要对象B的时候，容器会主动创建一个对象B注入到对象A需要的地方。好处：1.各个类之间只有与容器连接时才具有相关性，所以任何一方出问题都不会影响到另一方的运行，增加了维护性与单元测试性，便于调试程序。2.由于容器与类之间的无关性，在保证接口标准的情况下，可以将一个大中型项目分割为多个子项目，团队成员分工明确，提高产品的开发效率；3.具有可重用性
2. AOP的原理？好处？原理：通过动态代理的方式进行实现，主要包括JDK动态代理（代理对象必须是接口，核心•InvocationHandler和Proxy）和cglib动态代理（cglib是一个代码生成的类库，可以在运行时动态生成某个类的子类）。好处：降低模块之间的耦合，2.使系统更容易扩展。3.避免修改业务代码，避免引入重复的代码，有更好的重用性。使用场景：在日志处理以及事务处理
3. 自动装载机制的原理（https://blog.csdn.net/weixin_44588495/article/details/106310221）
4. 好处：1、非侵入式设计 Spring是一种非侵入式（non-invasive）框架，它可以使应用程序代码对框架的依赖最小化。2、方便解耦、简化开发 Spring就是一个大工厂，可以将所有对象的创建和依赖关系的维护工作都交给Spring容器的管理，大大的降低了组件之间的耦合性。3、支持AOP Spring提供了对AOP的支持，它允许将一些通用任务，如安全、事物、日志等进行集中式处理，从而提高了程序的复用性。4、支持声明式事务处理 只需要通过配置就可以完成对事物的管理，而无须手动编程。5、方便程序的测试 Spring提供了对Junit4的支持，可以通过注解方便的测试Spring程序。6、方便集成各种优秀框架 Spring不排斥各种优秀的开源框架，其内部提供了对各种优秀框架（如Struts、Hibernate、MyBatis、Quartz等）的直接支持。7、降低Java EE API的使用难度。Spring对Java EE开发中非常难用的一些API（如JDBC、JavaMail等），都提供了封装，使这些API应用难度大大降低。

8.讲一下GC的算法？

- 标记-清除算法：标记无用对象，然后进行清除回收。缺点：效率不高，无法清除垃圾碎片。
- 标记-整理算法：标记无用对象，让所有存活的对象都向一端移动，然后直接清除掉端边界以外的内存。
- 复制算法：按照容量划分二个大小相等的内存区域，当一块用完的时候将活着的对象复制到另一块上，然后再把已使用的内存空间一次清理掉。缺点：内存使用率不高，只有原来的一半。
- 分代算法：根据对象存活周期的不同将内存划分为几块，一般是新生代和老年代，新生代基本采用复制算法，老年代采用标记整理算法。

9.常见的运行时异常

1, java.lang.NullPointerException

这个异常的解释是 "程序遇上了空指针"，简单地说就是调用了未经初始化的对象或者是不存在的对象，这个错误经常出现在创建图片，调用数组这些操作中，比如图片未经初始化，或者图片创建时的路径错误等等。

2, java.lang.ClassNotFoundException

异常的解释是"指定的类不存在"，这里主要考虑一下类的名称和路径是否正确即可

3, java.lang.ArrayIndexOutOfBoundsException

这个异常的解释是"数组下标越界"，现在程序中大多都有对数组的操作，因此在调用数组的时候一定要认真检查，看自己调用的下标是不是超出了数组的范围，一般来说，显示（即直接用常数当下标）调用不太容易出这样的错，但隐式（即用变量表示下标）调用就经常出错了。

4, java.lang.NoSuchMethodError

方法不存在错误。当应用试图调用某类的某个方法，而该类的定义中没有该方法的定义时抛出该错误。

5, java.lang.IndexOutOfBoundsException

索引越界异常。当访问某个序列的索引值小于0或大于等于序列大小时，抛出该异常。

6, java.lang.NumberFormatException

数字格式异常。当试图将一个String转换为指定的数字类型，而该字符串确不满足数字类型要求的格式时，抛出该异常。

7, java.sql.SQLException

Sql语句执行异常 8, java.io.IOException

输入输出异常

9, java.lang.IllegalArgumentException

方法参数错误 10 java.lang.IllegalAccessException

无访问权限异常

10.runnable与callable的区别(具体见并发编程3.1)

Runnable执行方法是run(),Callable是call() 实现Runnable接口的任务线程无返回值；实现Callable接口的任务线程能返回执行结果 call方法可以抛出异常，run方法若有异常只能在内部消化 注意Callable接口支持返回执行结果，需要调用FutureTask.get()方法实现，此方法会阻塞主线程直到获取结果；当不调用此方法时，主线程不会阻塞！

11.Dao的设计模式

<https://www.cnblogs.com/ysyasd/p/11941423.html>

Dao设计模式封装了操作具体数据库的细节，对业务层提供操作数据库的接口，因此降低了业务层代码与具体数据库之间的耦合，有利于人员分工，增加了程序的可移植性。 Dao设计模式中主要包含这5个模块：

1、VO类：VO（Value Object）即值对象，每一个值对象对应一张数据库表，便于我们传递数据。

- 2、Dao接口：Dao接口定义了操作数据库的方法，业务层通过调用这些方法来操作数据库。
- 3、Dao实现类：操作数据库的方法的具体实现，封装了操作数据库的细节。
- 4、Dao工厂类：用于代替new操作，进一步降低业务层与数据层之间的耦合。
- 5、数据库连接类：封装了连接数据库、关闭数据库等常用的操作，减少重复编码。

12.“equal”与“==”

对于基本类型，== 判断两个值是否相等，基本类型没有 equals() 方法。对于引用类型，== 判断两个变量是否引用同一个对象，而 equals() 判断引用的对象是否等价。

```
Integer x = new Integer(1);
Integer y = new Integer(1);
System.out.println(x.equals(y)); // true
System.out.println(x == y);      // false
```

13.算法题 (动态规划类型的)

2021.05.12 美团 后端研发工程师 一面

面试流程 (目前只记得这么多)

1.讲解一下自己的项目，根据项目进行提问。

2.为什么java是跨平台的？

java会把文件编译成二进制字节码的class文件，由于java是运行在jvm上的，所以它的代码能在不同的平台的jvm上运行。

3.String的创建方式？

字符串常量池 (String Pool) 保存着所有字符串字面量 (literal strings)，这些字面量在编译时期就确定。不仅如此，还可以使用 String 的 intern() 方法在运行过程中将字符串添加到 String Pool 中。当一个字符串调用 intern() 方法时，如果 String Pool 中已经存在一个字符串和该字符串值相等 (使用 equals() 方法进行确定)，那么就会返回 String Pool 中字符串的引用；否则，就会在 String Pool 中添加一个新的字符串，并返回这个新字符串的引用。下面示例中，s1 和 s2 采用 new String() 的方式新建了两个不同字符串，而 s3 和 s4 是通过 s1.intern() 方法取得一个字符串引用。intern() 首先把 s1 引用的字符串放到 String Pool 中，然后返回这个字符串引用。因此 s3 和 s4 引用的是同一个字符串。

```
String s1 = new String("aaa");
String s2 = new String("aaa");
System.out.println(s1 == s2);          // false
String s3 = s1.intern();
String s4 = s1.intern();
System.out.println(s3 == s4);          // true
```

如果是采用 "bbb" 这种字面量的形式创建字符串，会自动地将字符串放入 String Pool 中。

```
String s5 = "bbb";
String s6 = "bbb";
System.out.println(s5 == s6); // true
```

4.讲一下常用的集合？

(https://blog.csdn.net/qq_40574571/article/details/97612100讲解了HashMap) hashMap:在每个数组元素上都一个链表结构，当数据被Hash后，得到数组下标，把数据放在对应下标元素的链表上。系统将调用key的hashCode()方法得到其hashCode 值（该方法适用于每个Java对象），然后再通过Hash算法的后两步运算（高位运算和取模运算，下文有介绍）来定位该键值对的存储位置，有时两个key会定位到相同的位置，表示发生了Hash碰撞。当然Hash算法计算结果越分散均匀，Hash碰撞的概率就越小，map的存取效率就会越高。在JDK1.8版本中，对数据结构做了进一步的优化，引入了红黑树。而当链表长度太长（默认超过8）时，链表就转换为红黑树，利用红黑树快速增删改查的特点提高HashMap的性能，其中会用到红黑树的插入、删除、查找等算法。list、map、stack、queue、set

5.面试说了spring是不变的，我这里说没听过，然后给我讲了一下

这里面试官通过string的例子给我大致讲解了一下，其类似于finald的关键字

6.问了怎么部署的？单节点还是多节点？

说了docker，提了一下dockerFile这些，之后又问我怎么保证单节点部署的时候数据不丢失，回答了：通过挂载的方式，面试官建议多节点通过docker部署。

7.算法（力扣82. 删除排序链表中的重复元素 II）

题目：存在一个按升序排列的链表，给你这个链表的头节点 head，请你删除链表中所有存在数字重复情况的节点，只保留原始链表中 没有重复出现的数字。

返回同样按升序排列的结果链表。

```
public ListNode deleteDuplicates(ListNode head) {
    if(head==null||head.next==null) return head;

    ListNode dummy=new ListNode(-1,head);
    ListNode pre=dummy;

    while(head!=null&&head.next!=null){
        if(head.val==head.next.val){
            ListNode temp=head.next;
            while(temp!=null&&temp.val==head.val){
                temp=temp.next;
            }
            pre.next=temp;
            head=temp;
        }
        pre=pre.next;
        head=head.next;
    }
    return dummy.next;
}
```



```

        }else{
            head=head.next;
            pre=pre.next;
        }

    }

    return dummy.next;
}

```

字节跳动一面

1.为什么Linux系统中存在这么多的编码

主要是每个国家有自己的语言，就像GBK是国家标准GB2312的基础上扩容后兼容GB2312的标准。GBK的文字编码是用双字节来表示的，即不论中、英文字符均使用双字节来表示，为了区分中文，将其最高位都设定成1。GBK包含全部中文字符，是国家编码，通用性比UTF8差，不过UTF8占用的数据库比GBK大。

GBK、GB2312等与UTF8之间都必须通过Unicode编码才能相互转换：

GBK、GB2312<===>Unicode<===>UTF8

而UTF-8是用以解决国际上字符的一种多字节编码，它对英文使用8位（即一个字节），中文使用24为（三个字节）来编码。UTF-8包含全世界所有国家需要用到的字符，是国际编码，通用性强。

2.TCP、IP、HTTP各自在网络的那一层

- TCP属于传输层，IP属于网络层，HTTP属于应用层

3.使用联合索引需要注意什么

遵循最佳左前缀法则

4.了解过范式？

- 第一范式：属性不可分
- 第二范式：每个非主属性完全函数依赖键码
- 第三范式：非主属性不传递函数依赖于键码

5.索引的实现原理。

索引主要是基于B+树进行实现的。其中B+树的高度通常很小，在数据只有3-4左右，此外他的叶子节点是一个顺序列表结构，他的非叶子节点是用来存储键值的。

6.Select和poll以及epoll使用场景,以及epoll中的ET和LT

	select	poll	epoll
FD数量	1024	无限制	无限制

	select	poll	epoll
FD状态感知	轮询	轮询	事件通知
重置数据源	需要	不需要 (event/revent)	通知就绪的
运行模式	条件触发 (LT)	条件触发 (LT)	边缘触发 (ET) /条件触发(LT)

7.进程与线程的区别

I 拥有资源 进程是资源分配的基本单位，但是线程不拥有资源，线程可以访问隶属进程的资源。 II 调度 线程是独立调度的基本单位，在同一进程中，线程的切换不会引起进程切换，从一个进程中的线程切换到另一个进程中的线程时，会引起进程切换。 III 系统开销 由于创建或撤销进程时，系统都要为之分配或回收资源，如内存空间、I/O 设备等，所付出的开销远大于创建或撤销 线程时的开销。类似地，在进行进程切换时，涉及当前执行进程 CPU 环境的保存及新调度进程 CPU 环境的设置，而 线程切换时只需保存和设置少量寄存器内容，开销很小。 IV 通信方面 线程间可以通过直接读写同一进程中的数据进行通信，但是进程通信需要借助 IPC

8.死锁的原因

- 互斥
- 占有和等待
- 不可抢占
- 循环等待

9.死锁如何避免

1.安全状态 2.单个资源的银行家算法 3.多个资源的银行家算法

10.学生成绩表 **table1**, 学生、课程、成绩，(name,sbuject,score)，查询出所有课程都大于80分的学生的平均成绩。

```
select avg(score), name from table1 where name not in (select distinct name from table1 where score < 80)
group by name ;
```

11.给定1个二维字符数组**m**和1个单词**w**，搜索**w**是否在**m**中。搜索的定义是从**m**的任意位置开始，可以上下左右移动，依次和**w**每个字符匹配，如果**w**能匹配完，则存在，否则不存在。并且二维数组中的元素不能被重复使用。

```
public class Main {

    public int[][] dir={{1,0},{-1,0},{0,1},{0,-1}};

    public boolean exist(char[][] board, String word) {

        if (word==null) return false;
        boolean[][] dp=new boolean[board.length][board[0].length];
        for (int i = 0; i < board.length; i++) {
            for (int j = 0; j < board[i].length; j++) {
```

```

        boolean ans=help(board,word,i,j,0,dp);
        if (ans) return true;
    }
}

return false;
}

private boolean help(char[][] board, String word, int i, int j, int index,
boolean[][] dp) {

    if (board[i][j]!=word.charAt(index)) return false;
    if (index==word.length()-1) return true;
    dp[i][j]=true;
    boolean result=false;
    for (int K = 0; K < dir.length; K++) {
        int newi=dir[K][0]+i;
        int newj=dir[K][1]+j;
        if (newi>=0&&newj>=0&&newi<board.length&&newj<board[0].length){
            if (!dp[newi][newj]){
                boolean flag=help(board,word,newi,newj,index+1,dp);
                if (flag){
                    result=true;
                    break;
                }
            }
        }
    }

    dp[i][j]=false;
    return result;
}
}

```

字节跳动二面

1.聊一下AQS

- AQS的核心思想是，如果被请求的共享资源空闲，则将当前请求资源的线程设置为有效的工作线程，并将共享资源设置为锁定状态，如果被请求的共享资源被占用，那么就需要一套线程阻塞等待以及被唤醒时锁分配的机制
- 实现了AQS的锁有：自旋锁、互斥锁、读锁写锁、条件产量、信号量、栅栏都是AQS的衍生物
- 1.Exclusive：独占，只有一个线程能执行，如ReentrantLock 2.Share：共享，多个线程可以同时执行，如Semaphore、CountDownLatch、ReadWriteLock、CyclicBarrier

2.聊一下异常？为什么异常分为运行时异常和非运行时异常



异常

- Error 一般是指java虚拟机相关的问题，如系统崩溃、虚拟机出错误、动态链接失败等，这种错误无法恢复或不可能捕获，将导致应用程序中断，通常应用程序无法处理这些错误，因此应用程序不应该捕获Error对象，也无须在其throws子句中声明该方法抛出任何Error或其子类。
- (1)运行时异常都是RuntimeException类及其子类异常，如NullPointerException、IndexOutOfBoundsException等，这些异常是不检查异常，程序中可以选择捕获处理，也可以不处理。这些异常一般是由程序逻辑错误引起的，程序应该从逻辑角度尽可能避免这类异常的发生。
- (2)非运行时异常是RuntimeException以外的异常，类型上都属于Exception类及其子类。如IOException、SQLException等以及用户自定义的Exception异常。对于这种异常，JAVA编译器强制要求我们必需对出现的这些异常进行catch并处理，否则程序就不能编译通过。所以，面对这种异常不管我们是否愿意，只能自己去写一大堆catch块去处理可能的异常。

3.keep-alive是怎么实现的

- 基于TCP的保活机制
- 如果两端的 TCP 连接一直没有数据交互，达到了触发 TCP 保活机制的条件，那么内核里的 TCP 协议栈就会发送探测报文。
1.如果对端程序是正常工作的。当 TCP 保活的探测报文发送给对端, 对端会正常响应，这样 TCP 保活时间会被重置，等待下一个 TCP 保活时间的到来。
2.如果对端主机崩溃，或对端由于其他原因导致报文不可达。当 TCP 保活的探测报文发送给对端后，石沉大海，没有响应，连续几次，达到保活探测次数后，TCP 会报告该 TCP 连接已经死亡。所以，TCP 保活机制可以在双方没有数据交互的情况，通过探测报文，来确定对方的 TCP 连接是否存活，这个工作是在内核完成的。

4.cas的ABA问题怎么解决的

- ABA问题：当有多个线程对一个原子类进行操作的时候，某个线程在短时间内将原子类的值A修改为B，又马上将其修改为A，此时其他线程不感知，还是会修改原子类的值A成功，而这和原本设计目的相悖。
- 解决：
1.使用数据库乐观锁解决ABA问题 实现思路：表中加一个VNO版本号字段，每次修改VNO = VNO + 1，则下一次修改则修改失败。
2.使用AtomicStampedReference解决ABA问题 实现思路：本质是有一个 int 值作为版本号，每次更改前先取到这个int值的版本号，等到修改的时候，比较当前版本号与当前线程持有的版本号是否一致，如果一致，则进行修改，并将版本号+1（当然加多少或减多少都是可以自己定义的），在zookeeper中保持数据的一致性也是用的这种方式。

5.数据库索引有没有什么办法A、C与A、B、C同时走索引

能不能把新建索引（Create index indexName on tableName(A,C,B)）？注意：数据库会进行自动优化 (1) select * from myTest where a=3 and b=5 and c=4; ---- abc顺序 abc三个索引都在where条件里面用到了，而且都发挥了作用

(2) select * from myTest where c=4 and b=6 and a=3; where里面的条件顺序在查询之前会被mysql自动优化，效果跟上一句一样

6.http2.0与http1.1的区别

- 头部压缩：压缩头（Header）如果你同时发出多个请求，他们的头是一样的或是相似的，那么，协议会帮你消除重复的分。
- 二进制格式：HTTP/2 不再像 HTTP/1.1 里的纯文本形式的报文，而是全面采用了二进制格式。
- 数据流：的数据包不是按顺序发送的，同一个连接里面连续的数据包，可能属于不同的回应。因此，必须要对数据包做标记，指出它属于哪个回应。每个请求或回应的所有数据包，称为一个数据流（Stream）

- 多路复用：HTTP/2 是可以在一个连接中并发多个请求或回应，而不用按照顺序一一对应。移除了 HTTP/1.1 中的串行请求，不需要排队等待，也就不会再出现「队头阻塞」问题，降低了延迟，大幅度提高了连接的利用率
- 服务器推送：还在一定程度上改善了传统的「请求 - 应答」工作模式，服务不再是被动地响应，也可以主动向客户端发送消息。举例来说，在浏览器刚请求 HTML 的时候，就提前把可能会用到的 JS、CSS 文件等静态资源主动发给客户端，减少延时的等待，也就是服务器推送（Server Push，也叫 Cache Push）。

7.聊了一下项目

8.buffer与cache的区别

- buffer 的存在原因是生产者和消费者对资源的生产/效率速率不一致;
- cache 的存在原因是资源调用的空间局部性

1.Buffer不是缓存，国内常用的翻译是缓冲区。2.其次，大部分场景中，Buffer是特指内存中临时存放的IO设备数据——包括读取和写入；而Cache的用处很多——很多IO设备（例如硬盘、RAID卡）上都有Cache，CPU内部也有Cache，浏览器也有Cache。3.Buffer并非用于提高性能，而Cache的目的则是提高性能。4.涉及到IO设备读写的场景中，Cache的一部分本身就是Buffer的一种。如果说某些场合Buffer可以提升IO设备的读写性能，只不过是因为Buffer本身是Cache系统的一部分，性能提升来自于Cache机制。5.Buffer占用的内存不能回收，如果被强行回收会出现IO错误。Cache占用的内存，除实现Buffer的部分外都可以回收，代价则是下一次读取需要从数据的原始位置（通常是性能更低的设备）读取。6.在IO读写过程中，任何数据的读写都必然会产生Buffer，但根据Cache算法，可能会有相当部分数据不会被Cache。

总结 第一，cache和buffer的根本区别在于它们解决的问题不同，cache解决的是性能问题，利用了访问局部性，buffer解决的是异步并发问题，第二，cache上下两端不是对等的，而buffer两端本质上是对等的agent，因此buffer类似生产者消费者队列，第三，具体实现是你中有我的，但一般是cache的实现需要内藏buffer

9.cookie与session的区别，然后它们之间怎么通讯

- Cookie 只能存储 ASCII 码字符串，而 Session 则可以存储任何类型的数据，因此在考虑数据复杂性时首选Session；
- Cookie 存储在浏览器中，容易被恶意查看。如果非要将一些隐私数据存在 Cookie 中，可以将 Cookie 值进行加密，然后在服务器进行解密；
- 对于大型网站，如果用户所有的信息都存储在 Session 中，那么开销是非常大的，因此不建议将所有的用户信息都存储到 Session 中
- 容量和个数限制：cookie 有容量限制，每个站点下的 cookie 也有个数限制。
- 存储的多样性：session 可以存储在 Redis 中、数据库中、应用程序中；而 cookie 只能存储在浏览器中。
- 存储位置不同：session 存储在服务器端；cookie 存储在浏览器端。
- 安全性不同：cookie 安全性一般，在浏览器存储，可以被伪造和修改。

通讯：1.服务器返回的响应报文的 Set-Cookie 首部字段包含了这个 Session ID，客户端收到响应报文之后将该 Cookie值存入浏览器中；2.客户端之后对同一个服务器进行请求时会包含该 Cookie 值，服务器收到之后提取出 Session ID，从 Redis 中取出用户信息，继续之前的业务操作。

10.防止数据库的注入

- 1、检查变量数据类型和格式
- 2、过滤特殊符号
- 3、绑定变量，使用预编译语句

11.单例模式

```
//单例模式
public class Singleton{
    private static volatile Singleton instace=null;
    private Singleton(){

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

12.学生成绩表 table1, 学生、课程、成绩，(name,sbuject,score)，查询出所有课程都大于80分的学生的平均成绩。

```
select avg(score), name from table1 where name not in (select distinct name from table1 where score < 80)
group by name ;
```

13.有一个 1GB 大小的文件，文件里每一行是一个词，每个词的大小不超过 16B，内存大小限制是 1MB，要求返回频数最高的 100 个词

14.平均延迟最大的调用链

字节跳动三面

1.数据库的索引以及两者的区别

聚簇索引和非聚簇索引，数据库的索引是用于加快数据的查询速度，索引分为聚簇索引与非聚簇索引。我们通常使用的InnoDB引擎，属于聚簇索引，他的数据查询主要依靠主键索引以及辅助索引，其底层的实现是B+树，叶子节点存储的是数据，通过主键索引可以直接查询到当前的数据，辅助索引的叶子节点存储了主键的key，如果想查询其他数据，他需要借助主键索引；而MyISAM的索引结构数据非聚簇索引，他的仍是B+树为底层，只不过其叶子节点存储的是数据的物理地址，其需要通过地址访问数据。

2.什么是幻读，数据库是怎么保证不出现幻读

- T1 读取某个范围的数据，T2 在这个范围内插入新的数据，T1 再次读取这个范围的数据，此时读取的结果和第一次读取的结果不同。
- MVCC中将readview中包含了多个查询，其数据结果要么同时提交，要么同时回滚。

3.数据库中锁的分类？间隙锁的实现原理

- 行级锁：是一种排他锁，防止其他事务修改此行；行级锁是Mysql中锁定粒度最细的一种锁，表示只针对当前操作的行进行加锁。行级锁能大大减少数据库操作的冲突。其加锁粒度最小，但加锁的开销也最大。行级锁分为共享锁和排他锁。特点：开销大，加锁慢；会出现死锁；锁定粒度最小，发生锁冲突的概率最低，并发度也最高。
- 页级锁：页级锁是MySQL中锁定粒度介于行级锁和表级锁中间的一种锁。表级锁速度快，但冲突多，行级冲突少，但速度慢。所以取了折衷的页级，一次锁定相邻的一组记录。BDB支持页级锁。特点：开销和加锁时间界于表锁和行锁之间；会出现死锁；锁定粒度界于表锁和行锁之间，并发度一般。
- 表级锁：锁定粒度大，发生锁冲突的概率最高,并发度最低。特点：开销小，加锁快；不会出现死锁(因为MyISAM会一次性获得SQL所需的全部锁)；

当我们用范围条件而不是相等条件检索数据，并请求共享或排他锁时，InnoDB 会给符合条件的已有数据记录的索引项加锁；对于键值在条件范围内但并不存在的记录，叫做“间隙 (GAP)”，InnoDB 也会对这个“间隙”加锁，这种锁机制就是所谓的间隙锁 (Next-Key 锁)。举例来说，假如 user 表中只有 101 条记录，其 userid 的值分别是 1,2,...,100,101，下面的 SQL：

```
Select * from user where userid > 100 for update;
```

是一个范围条件的检索，InnoDB 不仅会对符合条件的 userid 值为 101 的记录加锁，也会对userid 大于 101 (但是这些记录并不存在) 的“间隙”加锁，防止其它事务在表的末尾增加数据。

InnoDB 使用间隙锁的目的，为了防止幻读，以满足串行化隔离级别的要求，对于上面的例子，要是不使用间隙锁，如果其他事务插入了 userid 大于 100 的任何记录，那么本事务如果再次执行上述语句，就会发生幻读。

4.docker的内存与cpu的区别？以及两台主机分布式A.ip与B.ip怎么交互。

在这个过程中每个服务都在独立的容器里运行，每台机器上都运行着相互不关联的容器，所有容器共享宿主机的cpu、磁盘、网络、内存等，即实现了进程隔离（每个服务独立运行）、文件系统隔离（容器目录修改不影响主机目录）、资源隔离（CPU内存磁盘网络资源独立）。

Docker容器的实现原理就是通过Namespace命名空间实现进程隔离、

在底层的setNamespaces方法中传入进程、用户、网络等，创建新docker容器时把对应的隔离参数传递进去，从而实现了与宿主机、与各个容器的进程、用户、网络隔离。

- 到node1和node2宿主机都是可以正常通信的，由于我们都是设置的不同网段，所以可以在每个宿主机上单独添加一条静态路由，指定数据包的流向：

node1:

```
route add -net 10.10.0.0 /16 gw 192.168.56.12
```

node2:

```
route add -net 172.17.0.0 /16 gw 192.168.56.11
```

这样不同宿主机上的容器就可以互联了。但是维护成本太高了。


- 新建overlay network 使用集群部署

```
member bd93686a68a54c2d is healthy: got healthy result from
http://10.211.55.11:2379
member e5230093897f552c is healthy: got healthy result from
http://10.211.55.9:2379
cluster is healthy
```

5.三次握手与四次挥手

- 当A向B发送请求，此时会将同步位SYN为1，并选择序号seq=x,代表传输的数据第一个数据字节的序号为x；当B接受到报文请求后，会将SYN=1、确认值 ACK=1置为1，确认号ack=x+1,选择序号seq=y;A收到此报文后会向B发送请求，此时ACK=1,ack=y+1;seq=x+1,当确认结束后将SYN=0;
- 1.两次握手无法判断当前连接是否是历史连接（序列号过期或者超时）。如果是历史连接（序列号过期或超时），则第三次握手发送的报文是 RST 报文，以此中止历史连接；如果不是历史连接，则第三次发送的报文是 ACK 报文，通信双方就会成功建立连接；2.(无法同步双方初始序列号) 只保证了一方的初始序列号能被对方成功接收，没办法保证双方的初始序列号都能被确认接收。3.由于没有第三次握手，服务器不清楚客户端是否收到了自己发送的建立连接的 ACK 确认信号，所以每收到一个 SYN 就只能先主动建立一个连接，即两次握手会造成消息滞留情况下，服务器重复接受无用的连接请求 SYN 报文，而造成重复分配资源。
- 四次握手其实也能够可靠的同步双方的初始化序号，但由于第二步和第三步可以优化成一步，所以就成了「三次握手」。

四次挥手

四次挥手 A向B发出请求，此时会将FIN=1,seq=u;此时B收到请求，将会向A发送报文，其中ACK=1,ack=U+1, seq=v;这时TCP服务器进程通知高层应用进程，从A到B这个放向的连接就释放了，Tcp的连接处于半关闭状态。B若发送数据，A仍要接收。若B已经没有向A发送的数据，其应用进程就通知TCP释放连接。此时B向A发送释放连接的请求，ACK=1,FIN=1,seq=w,ack=u+1;当A收到这段报文后，必须发出确认。ACK=1,ack=w+1;seq=u+1;同时要注意此时A要等待2MSL（60s）的时间，确保A发送的最后一个请求能够到达A端，还能防止“已失效的连接请求报文段”出现在下一次请求中。

6.怎么抵挡syn攻击

- syn攻击：我们都知道 TCP 连接建立是需要三次握手，假设攻击者短时间伪造不同 IP 地址的 SYN 报文，服务端每接收到一个 SYN 报文，就进入SYN_RCVD 状态，但服务端发送出去的 ACK + SYN 报文，无法得到未知 IP 主机的 ACK 应答，久而久之就会占满服务端的 SYN 接收队列（未连接队列），使得服务器不能为正常用户服务。
- 通过修改 Linux 内核参数，控制队列大小和当队列满时应做什么处理。

1.当网卡接收数据包的速度大于内核处理的速度时，会有一个队列保存这些数据包。控制该队列的最大值如下参数：

```
net.core.netdev_max_backlog
```

2.SYN_RCVD 状态连接的最大个数：

```
net.ipv4.tcp_max_syn_backlog
```

3.超出处理能时，对新的 SYN 直接回 RST，丢弃连接：

```
net.ipv4.tcp_abort_on_overflow
```

- 1.当「SYN 队列」满之后，后续服务器收到 SYN 包，不进入「SYN 队列」；2.计算出一个 cookie 值，再以 SYN + ACK 中的「序列号」返回客户端，3.服务端接收到客户端的应答报文时，服务器会检查这个 ACK 包的合法性。如果合法，直接放入到「Accept 队列」。4.最后应用通过调用 `accept()` socket 接口，从「Accept 队列」取出的连接。

7. LeetCode 31题

```
public void nextPermutation(int[] nums) {
    if(nums==null||nums.length==0) return;
    int firstIndex=-1;
    for (int i = nums.length-2; i>=0; i--) {
        if (nums[i]<nums[i+1]){
            firstIndex=i;
            break;
        }
    }
    if (firstIndex== -1) {
        Arrays.sort(nums);
        return;
    }
    int second=-1;
    for (int i = nums.length-1; i >= 0; i--) {
        if (nums[firstIndex]<nums[i]){
            second=i;
            break;
        }
    }
    swap(nums,firstIndex,second);
    reverse(nums,firstIndex+1,nums.length-1);
    return;
}
```

```
public void reverse(int[] nums,int i,int j){
    while (i<j){
        swap(nums,i++,j--);
    }
}

public void swap(int[] nums,int i,int j){
    int temp=nums[i];
    nums[i]=nums[j];
    nums[j]=temp;
}
```

8.从数据库表查询语文成绩及格但是平均分不及格的学生

```
select student.id,student.score,a.avg from student
inner JOIN (
    select id,avg(score) as avg from student
    group by id having avg(score) < 60
) a
on student.id = a.id where student.name='Chinese' and score >= 60
```

大华二面

1.线程池的拒绝策略

- CallerRunsPolicy (调用者运行策略) : 如果添加到线程池失败, 那么主线程自己会去执行该任务; 如果执行程序已关闭 (主线程运行结束), 则会丢弃该任务, 但是由于是调用者线程自己执行的, 当多次提交任务时, 就会阻塞后续任务执行, 性能和效率自然就慢了。
- AbortPolicy (中止策略) : 默认, 队列满了丢任务抛出异常。中止策略的意思也就是打断当前执行流程
- DiscardPolicy (丢弃策略) : 队列满了丢任务不抛出异常。直接静悄悄的丢弃这个任务, 不触发任何动作。所以这个策略基本不用了。
- DiscardOldestPolicy (弃老策略) : 将最早进入队列的任务删除 (弹出队列头部的元素), 之后尝试加入队列。

2.出现异常时, jvm怎么操作的

- 1.JVM会在当前出现异常的方法中, 查找异常表, 是否有合适的处理者来处理
- 2.如果当前方法异常表不为空, 并且异常符合处理者的from和to节点, 并且type也匹配, 则JVM调用位于target的调用者来处理。
- 3.如果上一条未找到合理的处理者, 则继续查找异常表中的剩余条目
- 4.如果当前方法的异常表无法处理, 则向上查找 (弹栈处理) 刚刚调用该方法的调用处, 并重复上面的操作。
- 5.如果所有的栈帧被弹出, 仍然没有处理, 则抛给当前的Thread, Thread则会终止。

6.如果当前Thread为最后一个非守护线程，且未处理异常，则会导致JVM终止运行。（finally代码块是如何去实现的？在编译阶段对finally代码块进行处理 当前版本Java编译器的做法，是复制finally代码块的内容，分别放到所有正常执行路径，以及异常执行路径的出口中。）

3.年轻代回收以及full GC，持久代能回收？

- 年轻代回收：大多数情况下，对象在新生代 Eden 上分配，当 Eden 空间不够时，发起 Minor GC。
- full GC回收
 1. 调用 System.gc() 只是建议虚拟机执行 Full GC，但是虚拟机不一定真正去执行。不建议使用这种方式，而是让虚拟机管理内存。
 2. 老年代空间不足 老年代空间不足的常见场景为前文所讲的大对象直接进入老年代、长期存活的对象进入老年代等。为了避免以上原因引起的 Full GC，应当尽量不要创建过大的对象以及数组。除此之外，可以通过 -Xmn 虚拟机参数 调大新生代的大小，让对象尽量在新生代被回收掉，不进入老年代。还可以通过 -XX:MaxTenuringThreshold 调大 对象进入老年代的年龄，让对象在新生代多存活一段时间。
 3. 空间分配担保失败 使用复制算法的 Minor GC 需要老年代的内存空间作担保，如果担保失败会执行一次 Full GC。。
 4. JDK 1.7 及以前的永久代空间不足 在 JDK 1.7 及以前，HotSpot 虚拟机中的方法区是用永久代实现的，永久代中存放的为一些 Class 的信息、常量、静态变量等数据。当系统中要加载的类、反射的类和调用的方法较多时，永久代可能会被占满，在未配置为采用 CMS GC 的情况下也会执行 Full GC。如果经过 Full GC 仍然回收不了，那么虚拟机会抛出 java.lang.OutOfMemoryError。为避免以上原因引起的 Full GC，可采用的方法为增大永久代空间或转为使用 CMS GC。
 5. Concurrent Mode Failure 执行 CMS GC 的过程中同时对对象要放入老年代，而此时老年代空间不足（可能是 GC 过程中浮动垃圾过多导致暂时性的空间不足），便会报 Concurrent Mode Failure 错误，并触发 Full G
- 永生代也是可以回收的，条件是 1.该类的实例都被回收。 2.加载该类的classLoader已经被回收 3.该类不能通过反射访问到其方法，而且该类的java.lang.class没有被引用 当满足这3个条件时，是可以回收，但回不回收还得看jvm。（会触发full gc）

4.redis的持久化存储

- AOF 日志：文件的内容是操作命令
- RDB 快照：文件的内容是二进制数据（RDB 快照就是记录某一个瞬间的内存数据，记录的是实际数据，而 AOF 文件记录的是命令操作的日志，而不是实际的数据。）

因此在 Redis 恢复数据时，RDB 恢复数据的效率会比 AOF 高些，因为直接将 RDB 文件读入内存就可以，不需要像 AOF 那样还需要额外执行操作命令的步骤才能恢复数据。

5.消息队列的交换机

直连交换机：Direct exchange 扇形交换机：Fanout exchange 主题交换机：Topic exchange 首部交换机：Headers exchange

6.检查性异常与非检查性异常

1.非检查性异常：Error 和 RuntimeException 以及他们的子类。Java语言在编译时，不会提示和发现这样的异常，不要求在程序中处理这些异常。所以我们可以程序中编写代码来处理（使用try...catch...finally）这样的异常

常，也可以不做任何处理。对于这些错误或异常，我们应该修正代码，而不是去通过异常处理器处理。这样的异常发生的原因多半是由于我们的代码逻辑出现了问题。

2.Java语言强制要求程序员为这样的异常做预备处理工作（使用try...catch...finally或者throws）。在方法中要么用try-catch语句捕获它并处理，要么用throws子句声明抛出它，否则编译不会通过。这样的异常一般是由程序的运行环境导致的。因为程序可能被运行在各种未知的环境下，而程序员无法干预用户如何使用他编写的程序，于是程序员就应该为这样的异常时刻准备着。如SQLException, IOException, ClassNotFoundException等。

7.springboot如果一个接口有多个子类，会出现问题？

可以 1、@Autowired 是通过 byType 的方式去注入的，使用该注解，要求接口只能有一个实现类。 2、@Resource 可以通过 byName 和 byType的方式注入，默认先按 byName的方式进行匹配，如果匹配不到，再按 byType的方式进行匹配。 3、@Qualifier 注解可以按名称注入（bean的名称）

阿里CTO部门一面

1.缓存线程池的了解

它是一种用来处理大量短时间工作任务的线程池，具有几个鲜明特点：它会试图缓存线程并重用，当无缓存线程可用时，就会创建新的工作线程；如果线程闲置的时间超过 60 秒，则 被终止并移出缓存；长时间闲置时，这种线程池，不会消耗什么资源。其内部使用SynchronousQueue 作为工作队列；

2.Aop的应用场景

场景一：记录日志

场景二：监控方法运行时间（监控性能）

场景三：权限控制

场景四：缓存优化（第一次调用查询数据库，将查询结果放入内存对象，第二次调用，直接从内存对象返回，不需要查询数据库）

场景五：事务管理（调用方法前开启事务，调用方法后提交关闭事务）

3.TCp连接的状态以及当客户端发生奔溃时，状态的变化

服务器会等待一段时间，如果超过定时器的时间就会单方面关闭连接，一般为5个定时器的时间，服务器端 ESTABLISHED->close（ACK：该位为 1 时，「确认应答」的字段变为有效，TCP 规定除了最初建立连接时的 SYN 包之外该位必须设置为 1。

RST：该位为 1 时，表示 TCP 连接中出现异常必须强制断开连接。

SYC：该位为 1 时，表示希望建立连，并在其「序列号」的字段进行序列号初始值的设定。

FIN：该位为 1 时，表示今后不会再有数据发送，希望断开连接。当通信结束希望断开连接时，通信双方的主机之间就可以相互交换 FIN 位置为 1 的 TCP 段。)

4.快排的时间复杂度与空间复杂度

时间复杂度： $n\log n$, 空间复杂度 $n\log n$

主要是递归造成的栈空间的使用，最好情况，递归树的深度为 $\log_2 n$ 空间复杂度也就为 $O(\log n)$

最坏情况，

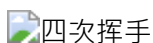
需要进行 $n-1$ 递归调用，其空间复杂度为 $O(n)$,

平均情况，

空间复杂度也为 $O(\log n)$ 。

腾讯一面总结

1. 四次回收的 `close_wait`



- 客户端打算关闭连接，此时会发送一个 TCP 首部 FIN 标志位被置为 1 的报文，也即 FIN 报文，之后客户端进入 `FIN_WAIT_1` 状态。
- 服务端收到该报文后，就向客户端发送 ACK 应答报文，接着服务端进入 `CLOSED_WAIT` 状态。
- 客户端收到服务端的 ACK 应答报文后，之后进入 `FIN_WAIT_2` 状态。
- 等待服务端处理完数据后，也向客户端发送 FIN 报文，之后服务端进入 `LAST_ACK` 状态。
- 客户端收到服务端的 FIN 报文后，回一个 ACK 应答报文，之后进入 `TIME_WAIT` 状态。
- 服务器收到了 ACK 应答报文后，就进入了 `CLOSE` 状态，至此服务端已经完成连接的关闭。
- 客户端在经过 `2MSL` 一段时间后，自动进入 `CLOSE` 状态，至此客户端也完成连接的关闭。(这里一点需要注意的是：主动关闭连接的，才有 `TIME_WAIT` 状态。)

2. 僵尸进程

一个进程的进程描述符在子进程退出时不会释放，只有当父进程通过 `wait()` 或 `waitpid()` 获取了子进程信息后才会释放。如果子进程退出，而父进程并没有调用 `wait()` 或 `waitpid()`，那么子进程的进程描述符仍然保存在系统中，这种进程称之为僵尸进程。僵尸进程通过 `ps` 命令显示出来的状态为 `Z (zombie)`。系统所能使用的进程号是有限的，如果产生大量僵尸进程，将因为没有可用的进程号而导致系统不能产生新的进程。要消灭系统中大量的僵尸进程，只需要将其父进程杀死，此时僵尸进程就会变成孤儿进程，从而被 `init` 进程所收养，这样 `init` 进程就会释放所有的僵尸进程所占有的资源，从而结束僵尸进程。


3. 孤儿进程

一个父进程退出，而它的一个或多个子进程还在运行，那么这些子进程将成为孤儿进程。孤儿进程将被 `init` 进程（进程号为 1）所收养，并由 `init` 进程对它们完成状态收集工作。由于孤儿进程会被 `init` 进程收养，所以孤儿进程不会对系统造成危害。

4. `accept` 在什么时候触发

发生在三次握手结束之后

5.socket通讯

 socket通讯 服务端和客户端初始化 socket，得到文件描述符；

服务端调用 bind，将绑定在 IP 地址和端口；

服务端调用 listen，进行监听；

服务端调用 accept，等待客户端连接；

客户端调用 connect，向服务器端的地址和端口发起连接请求；

服务端 accept 返回用于传输的 socket 的文件描述符；

客户端调用 write 写入数据；服务端调用 read 读取数据；

客户端断开连接时，会调用 close，那么服务端 read 读取数据的时候，就会读取到了 EOF，待处理完数据后，服务端调用 close，表示连接关闭。

6.数据库中的回表

先通过普通索引的值定位聚簇索引值，再通过聚簇索引的值定位行记录数据，需要扫描两次索引B+树，它的性能较扫一遍索引树更低。（解决方法索引覆盖）

7.http3.0

QUIC 是一个在 UDP 之上的伪 TCP + TLS + HTTP/2 的多路复用的协议。

- 解决了对头阻塞问题：基于UDP，在一条链接上可以有多个流，流与流之间是互不影响的，当一个流出现丢包影响范围非常小，从而解决队头阻塞问题。
- 0RTT 建链：QUIC则第一个数据包就可以发业务数据，从而在连接延时有很大优势，可以节约数百毫秒的时间。（衡量网络建链的常用指标是RTT Round-Trip Time，也就是数据包一来一回的时间消耗。RTT 包括三部分：往返传播时延、网络设备内排队时延、应用程序数据处理时延）

QUIC的0RTT也是需要条件的，对于第一次交互的客户端和服务端0RTT也是做不到的，毕竟双方完全陌生。

因此，QUIC协议可以分为首次连接和非首次连接，两种情况进行讨论。

使用QUIC协议的客户端和服务端要使用1RTT进行密钥交换，使用的交换算法是DH(Diffie-Hellman)迪菲-赫尔曼算法。

1 首次连接 简单来说一下，首次连接时客户端和服务端的密钥协商和数据传输过程，其中涉及了DH算法的基本过程：

客户端对于首次连接的服务端先发送client hello请求。

服务端生成一个素数 p 和一个整数 g ，同时生成一个随机数(笔误-此处应该是 K_{s_pri})为私钥，然后计算出公钥 $= g \mod p$ ，服务端将 p, g 三个元素打包称为config，后续发送给客户端。

客户端随机生成一个自己的私钥，再从config中读取 g 和 p ，计算客户端公钥 $= g^{\text{私钥}} \mod p$ 。

客户端使用自己的私钥和服务端发来的config中读取的服务端公钥，生成后续数据加密用的密钥 $K = \text{服务端公钥}^{\text{客户端私钥}} \mod p$ 。

客户端使用密钥K加密业务数据，并追加自己的公钥，都传递给服务端。

服务端根据自己的私钥和客户端公钥生成客户端加密用的密钥 $K = \text{mod } p$ 。

为了保证数据安全，上述生成的密钥K只会生成使用1次，后续服务端会按照相同的规则生成一套全新的公钥和私钥，并使用这组公私钥生成新的密钥M。

服务端将新公钥和新密钥M加密的数据发给客户端，客户端根据新的服务端公钥和自己原来的私钥计算出本次的密钥M，进行解密。

之后的客户端和服务端数据交互都使用密钥M来完成，密钥K只使用1次。

2. 非首次连接 前面提到客户端和服务端首次连接时服务端传递了config包，里面包含了服务端公钥和两个随机数，客户端会将config存储下来，后续再连接时可以直接使用，从而跳过这个1RTT，实现0RTT的业务数据交互。

客户端保存config是有时间期限的，在config失效之后仍然需要进行首次连接时的密钥交换。

8.docker的隔离、限制、文件联合，以及他是这么搭建的，docker上传镜像的命令。

容器技术的核心功能就是通过约束和修改进程的动态表现，从而为其创造出一个边界，Cgroup技术是用来制造约束的主要手段，而namespace是用来修改进程视图的主要方法

1.Namespace

我们知道运行的服务即一个进程，进程提供了服务运行需要的软硬件环境，在一台宿主机上同时启动多个服务时，可能会出现资源的争夺、进程互相影响等，因此通过namespace就可以将宿主机上同时运行的多个服务划分成每个独立的服务，自己单独进程运行

2.以PID Namespace为例

虽然容器内的第1号进程在“障眼法”的干扰下只能看到容器里的情况，但是宿主机上，它作为第100号进程与其他所有进程之间依然是平等的竞争关系。这就意味着，虽然第100号进程表面上被隔离了起来，但是它所能够使用到的资源（比如CPU、内存），却可随时被宿主机上其他进程（或容器）占用的。当然，这个100号进程自己也可能把所有资源吃光。这些情况，显然都不是一个“沙盒”应该表现出来的合理行为。

Linux Cgroups就是Linux内核中用来为进程设置资源限制的一个重要功能。Google的工程师在2006年发起这项特性的时候，曾将它命名为“进程容器”（process container）。实际上，在Google内部，“容器”这个术语长期以来都被用于形容被Cgroups限制过的进程组。后来Google的工程师们说，他们的KVM虚拟机也运行在Borg所管理的“容器”里，其实也是运行在Cgroups“容器”当中。这和我们今天说的Docker容器差别很大。

Linux Cgroups的全称是Linux Control Group。它最主要的作用，就是限制一个进程组能够使用的资源上限，包括CPU、内存、磁盘、网络带宽等等。此外，Cgroups还能够对进程进行优先级设置、审计，以及将进程挂起和恢复等操作。只探讨它与容器关系最紧密的“限制”能力，并通过一组实践来认识一下Cgroups。

在Linux中，Cgroups给用户暴露出来的操作接口是文件系统，即它以文件和目录的方式组织在操作系统的/sys/fs/cgroup路径下

3.联合文件系统（UnionFS）是一种分层、轻量级并且高性能的文件系统，它支持对文件系统的修改作为一次提交来一层层的叠加，同时可以将不同目录挂载到同一个虚拟文件系统下（unite several directories into a single virtual filesystem）。它就是把多个目录联合放在同一个目录下，而这些目录的物理位置是分开的。在docker的镜像设计中，用户制作镜像的每一步操作就是多增加一个目录（docker中称之为层layer），这个java程序1和

java程序2所在的容器就引用相同的操作系统层、java环境层，再结合应用程序层，启动docker容器时通过UnionFS把相关的层全放在一个目录里，作为容器的根文件系统，而容器的启动就是可写层，来对docker镜像进行操作。

docker-compose push/docker push