目录

[1 Javase 2](#Javase)

[1.1 Java基础知识 1](#_Toc14007808)

[1.1.1重载和重写的区别 1](#_Toc14007809)

[1.2 Java集合 2](#_Toc14007810)

[1.3 Java多线程 4](#_Toc14007811)

[1.4 IO 6](#_Toc14007812)

[1.5 JVM 8](#_Toc14007813)

[2 计算机网络 10](#_Toc14007814)

[3 Linux 12](#_Toc14007815)

[4 Mysql 14](#_Toc14007816)

[5 Java web编程 16](#_Toc14007817)

[6 JavaEE 18](#_Toc14007818)

[6.1 Spring 20](#_Toc14007819)

[6.2 Hibernate 22](#_Toc14007820)

[6.3 struts 24](#_Toc14007821)

[6.4 Mybaits 26](#_Toc14007822)

[6.5 MVC 28](#_Toc14007823)

[6.6 JPA 30](#_Toc14007824)

[7 操作系统 32](#_Toc14007825)

[8 算法与数据结构 34](#_Toc14007826)

[9 消息队列 36](#_Toc14007827)

[10 场景题 38](#_Toc14007828)

# Javase

## Java基础知识

### 1.1.1重载和重写的区别

答：a. 发生的范围不同 重载可以发生在父类和子类之间，而重写只能发生在同一类中。

b. 参数列表及返回值等要求不同。重写和重载都要求方法名完全一致，而重载时参数列表一定要和原方法不同，包括参数类型不同，参数顺序不同以及参数个数不同。重载时返回值可以和原方法不相同。重写时，方法名和参数列表要和原方法一模一样，包括之前说到的三点，类型，顺序，个数。重写的返回值类型必须小于等于父类的返回值类型。

c. 发生时期的不同，重载发生在编译期而重写发生在运行期。原因：编译时，编译器可以通过方法参数列表的不同来判断我们需要调用的是哪个重载方法。但是却不能判断具体是哪个对象要调用重写方法，只有在运行时才能得知是哪个对象调用的重写方法。

d. 抛出的异常范围要求不同。重写要求子类重写父类方法时抛出的异常类型小于或者等于父类抛出的异常类型。而重载则没有这个要求。重写要求访问修饰符的范围大于等于父类的访问修饰符。重载也没有这个要求。

public>protected>default>private

protected: 同一个类，同一个包以及子类

default:同一个类以及同一个包

### 1.1.2 String和StringBuffer、StringBuilder之间的区别是什么？String为什么是不可变的？

答：区别从三个方面来比较，一是线程安全性2是可变性3是性能

一String是不可变的，可以看作是常量，因此是线程安全的。StringBuffer对方法加了同步锁，也是线程安全的。StringBuilder线程不安全

二String不可变，而StringBuffer以及StringBuilder是可变的，这里可变与不可变的意思是当对String类型的变量进行操作时，不会改变原来的String对象，而是会重新创建一个新的String对象，原String类型引用会重新指向新生成的对象。比如String a = “abc”; a = a+”cd”此时会生成另一个String类型的对象”abccd”并让原引用a指向该对象。

三性能，在需要少量数据时，可以使用String类型变量

需要大量数据且单线程时，可以使用StringBuilder类型变量

需要大量数据且多线程时，使用StringBuffer类型变量

String不可变原因。String类中使用final关键字来保存数据，private final char value[]。因此String对象是不可变的。

### 1.1.3 自动装箱与拆箱

自动装箱：将基本类型自动转换为对应的引用类型比如int转换成Integer，调用的是Integer.valueOf()方法。场景：基本类型作为参数传递到相应的包装类型方法；基本类型分配到相应的包装类型变量。

自动拆箱：将引用类型自动转换成对应的基本类型Integer转换成int。调用Integer.intValue()方法。场景：包装类型作为参数传递到相应的基本类型方法；包装类型分配到相应的基本类型变量。

### 1.1.4 ==与equals

==判断是两个对象是否相等，即判断的是对象的地址是否一直，是否为同一对象。

equals如果没有被重写，等价于==判断是否为同一对象，如果被重写一般是判断两个对象的内容是否一致。

### 1.1.5 创建String类型对象

方式一 String a = “abc”，此时如果常量池中存在字符串”abc”，则让a直接指向常量池中的”abc”，如果没有，则创建”abc”并且放入常量池中。

方式二 String a = new String(“abc”) 此时先在堆内存中创建一个String类型的对象，其值为”abc”，再判断常量池中是否有”abc”如果有，则String对象的值为”abc”，如果没有，则在常量池中创建字符串”abc”；并将String对象的值指向”abc”。

### 1.1.6关键字final

final修饰变量，表示这个变量不可以被改变

final修饰方法，表示这个方法不能被重写。private修饰的方法也被隐式指定为final方法。

final修饰类，表示这个类不能被继承，final类中的所有成员方法都会被隐式地指定为final方法。

### 1.1.7Object类常见方法

//native方法，用于返回当前运行时对象的Class对象，使用了final关键字。

public final native Class<?> getClass()

public native int hashCode() //native方法，返回该对象的hash码

public boolean equals(Object obj) //用于比较两个对象的地址是否一致

protected native Object clone() throws CloneNotSupportedException//native方法，用于创建并返回该对象的一份拷贝。需要重写该方法

public String toString() //返回类名@实例的哈希码的16禁止字符串

public final native void notify() //native方法，不能重写，唤醒一个在此对象监视器上等待的线程。如果有多个线程在等待，只会唤醒其中一个。

public final native void notifyAll() //native方法，唤醒所有在此对象监视器上等待的线程

public final native void wait(long timeout) throws InterruptedException //native方法，不能重写，调用该方法的线程会等待，并释放锁。

public final void wait(long timeout,int nanos) throws InterruptedException//nanos参数表示额外的时间，超时的时间还要加上nanos毫秒。

public final void wait() throws InterruptedException //一直等待

protected void finalize() throws Throwable //实例被垃圾回收期回收时候触发的操作

### 1.1.8 java中的异常处理

分类：

受检查异常：编译时被强制检查的异常，方法声明中会被声明，一般为非程序引起的错误情况，比如FileNotFoundException找不到文件异常。

非受检查异常：方法中不会被声明的异常，运行时发生的异常都属于非检查异常，一般是程序引起的错误，比如空指针异常。

异常处理：使用throw、throws以及try-catch和finally关键字来进行异常的处理。在方法体内，将可能抛出异常的代码块放在try语句块里，try语句块后紧跟catch语句块用来捕获异常。并进行异常处理，比如输出异常日志等。finally里的代码不关try-cath里发生什么都会被执行。throw用来显示地抛出异常。throws用来在方法头中声明该方法可能会抛出的异常，如果方法可能会抛出地异常没有在方法里被捕获处理，就在方法头使用throws方法声明，告诉调用该方法的对象或方法需要处理的异常类型。

四种情况下finally里的代码块不会被执行：前面的代码执行了system.exit()退出程序，线程死亡，finally代码块里发生了异常，关闭CPU

### 1.1.9 获取键盘输入的两种方法

Scanner input = new Scanner(System.in);

String s = input.nextLine();

input.close();

BufferedReader input = new BufferedReader(new InputStreamReader(system.in));

String s = input.nextLine();

### 1.1.10 接口和抽象类的区别

一个类在实现接口时，必须实现接口的所有方法，而且可以实现多个接口，在继承抽象类时，可以不必实现抽象类的所有方法，只能继承一个抽象类。

抽象类中的方法可以是普通方法，也可以是抽象方法，抽象方法必须使用abstract关键字，接口中的方法都不能有方法体，接口中的变量不可以为private，且必须被初始化，方法可以为private，但此时必须有方法体，此时该方法只能在接口内部被调用方法不可以为protected。

## Java集合

### 1.2.1 ArrayList和LinkedList异同

线程安全：ArrayList和LinkedList都是线程不安全的

底层数据结构：ArrayList 底层实现是Object数组，LinkedList底层实现是双向链表。

插入删除效率：ArrrayList插入删除效率是O(n)，LinkedList插入删除效率是O(1)

是否支持随机访问：ArrayList支持，LinkedList不支持

### 1.2.2 ArrayList和Vector区别

Vector所有方法都是同步的，ArrayList方法不是同步的,线程不安全。多线程下使用Vector类，单线程时使用ArrayList。Vector会在同步操作上耗费较多时间。

### 1.2.3 HashMap的底层实现

JDK1.8之前使用数组和链表实现的散列表。HashMap通过key的hashcode来决定key-value存放在哪个地方。如果计算出的数组位置上已经存在元素，判断要存入的元素的Key值与已存在的元素的Key值是否相同，如果相同就直接覆盖值。不相同就通过拉链法解决冲突。

数组位置的计算：通过key的hashcode经过扰动函数后得到hash值，再通过除留余数法得到在数组位置。扰动函数可以减少碰撞。

JDK1.8之后的版本，当链表长度大于阈值(默认为8)时，将链表转换为红黑树，减少搜索时间。TreeMap和TreeSet底层都用到了红黑树。目的是为了解决二叉查找数的缺陷，因为二叉查找树在某些情况下会退化成一个线性结构。红黑树？定义，使用场景，遍历方法？

### 1.2.4 HashMap和Hashtable的区别

线程安全，效率，初始容量和扩容，底层数据结构

a. 线程安全：HashMap线程不安全，HashTable的方法都使用了sychronized关键字，线程安全

b. 效率 HashMap的效率高于HashTable。HashTable允许有一个为null的键，允许有多个键对应的值为null，但HashTable不允许键或者值为null。

c. 初始容量和扩容不一样，HashTable初始容量为11，每次扩容为原来的2n+1倍，HashMap默认初始容量为16，每次扩容为原来的2倍。若给定了初始容量，则HashTable的初始值即为给定值，而HashMap会扩展为2的n次幂。

d．底层数据结构，JDK1.8之后在解决冲突时有了较大改变，当链表长度大于8时，会将链表转换为红黑树，但是HashTable没有这样的机制。

1.2.5 HashMap的数组长度为什么是2的幂次方

HashMap的散列函数使用的方法是除留取余法，就是使用hashcode值除以某个p值，把余数当作数组的位置，p值一般取数组长度，而当数组长度为2的n次幂时，hashcode除以数组长度的余数和数组长度减一再和hashcode按位与的值一样，使用按位与相对%取模运算要高效，因此就将HashMap的数组长度设置为2的n次幂。

### 1.2.6 HashMap多线程操作导致死循环问题。

多线程下，HashMap的put操作可能会导致死循环的问题。原因在于HashMap的扩容resize()方法。扩容是新建一个数组，将原来的数组复制到新数组中。两个线程复制同一个数组时就会导致死循环的问题

### 1.2.7 HashSet和HashMap的区别

HashSet底层使用HashMap实现。HashSet中除了clone()，readObject(),writeObject()等方法是自己实现的，其余的都是直接调用HashMap的方法。

HashMap存储键值对，HashSet只存储对象。HashSet中值为同一个Object。

### 1.2.8 ConcurrentHashMap和HashTable的区别

同为线程安全，但是在实现上有较大差别

底层数据结构：ConcurrentHashMap采用分段的数组+链表实现，HashTable存储结构与冲突解决方式与HashMap一致

实现线程安全方式：ConcurrentHashMap(分段锁)对整个桶数组进行了分割分段，每一个锁只锁每一段中的数据，多线程访问时，如果访问的是不同段的数据，就不会存在锁竞争。到JDK1.8之后，使用和HashMap相同的Node数组+链表/红黑树的结构，并发控制使用synchronized和CAS来实现。从JDK1.7版本的ReentrantLock+segement+HashEntry变成JDK1.8的sychronized+CAS+HashEntry+红黑树，synchronized只锁定当前链表或红黑树的树节点。HashTable使用sychronized来保证线程安全，效率低下，当一个线程访问同步方法，其他线程可能会进入阻塞或轮询状态，如果使用put添加元素，另一个线程不能使用put也不能使用get，线程间竞争激励且效率低。

### 1.2.9集合框架底层数据结构总结

Collection

1. List

ArrayList：Object 数组

Vector：Object数组

LinkedList：双向链表

1. Set

HashSet(无序，唯一)基于HashMap实现，底层采用HashMap实现

LinkedHashSet:继承于HashSet，其内部是通过LinkedHashMap来是西安的

TreeSet(有序，唯一)：红黑树

Map

HashMap：数组+链表/红黑树

LinkedHashMap：继承自HashMap，底层基于拉链式散列结构。在该基础上，增加了一条双向链表，使得上面的结构可以保持键值对的插入顺序，同时通过对链表进行相应的操作，实现了访问顺序相关逻辑。

HashTable：数组+链表

TreeMap:红黑树

## Java多线程

### 1.3.1乐观锁与悲观锁

<https://www.cnblogs.com/kismetv/p/10787228.html>

基本概念：乐观锁与悲观锁是两种思想，用于解决并发场景下数据竞争问题

乐观锁：在对数据进行操作时，假设没有其他人对数据修改，只是在更新时检查数据是否已经被修改，如果已经被修改，就不进行更新，如果没有，就进行更新。

悲观锁：在对数据进行操作时，假设会有其他人对数据进行修改，将数据或代码块进行加锁操作，直到操作结束后才会释放锁，加锁期间其他人不能修改数据。

实现方式：

乐观锁：通过CAS和version来实现。CAS的操作逻辑是如果内存位置V的值等于给定值A，就将该位置的值更新为B。CAS含有比较和交换两个操作，CAS是由CPU支持的原子操作，其原子性是在硬件层面保证的。比如给需要进行更新的数据加入关键字AtomicInteger等。版本号机制，每次对数据进行修改时版本号加一，查找数据时，将数据和版本号进行一起查询，当需要对数据更新时检查版本号和之前查出来的版本是否一致，如果一致就进行更新操作。

悲观锁：通过给数据或者代码块加锁来实现。

优缺点和适用场景

功能限制：与悲观锁相比，乐观锁的适用场景收到了更多的限制，无论是CAS还是版本号机制。CAS只能保证单个变量操作的原子性，当涉及到多个变量时，CAS无能为力。但synchronized则可以对整个代码块加锁来处理。版本号机制：

竞争激烈程度：

竞争不激烈时，乐观锁更有优势，因为悲观锁会锁住代码块，其他线程无法访问，影响并发，而且加锁和释放锁都需要消耗额外的资源。

竞争激烈时，悲观锁更有优势，乐观锁在更新失败时频繁失败，需要不断重试，浪费CPU资源。

乐观锁加锁么？

乐观锁本身是不加锁的，只是在更新时检查一下数据是否被其他线程修改了。

CAS有哪些缺点；

ABA问题：线程一读取数据A，线程二先修改A为B，再将B改为A，此时线程A再执行CAS操作，会认为A没有被修改，但其实数据A已经被修改过了。在某些场景下ABA问题可能带来隐患，比如栈顶问题：一个栈的栈顶数据经过几次修改又恢复成原值，但此时栈可能已经被改变了。对于ABA问题，可以引入版本号来解决。进行CAS操作时，不仅比较原值，也比较版本号。

java中的AtomicStampedReference类便是使用版本号来解决ABA问题的。

### 1.3.2说一说自己对于synchrinized关键字的了解

同步和异步用来形容一次方法的调用。同步方法调用一旦开始，调用者必须等到方法调用完成才能进行后续的操作。异步方法通常会在另一个线程中执行。

synchronized关键字解决多个线程之间访问资源的同步性，synchronized关键字可以保证被它修饰的变量或者代码块在任意时刻只能被一个线程所执行。java早期版本中，synchronized属于重量锁，效率低下，监视器锁(monitor)依赖于底层操作系统的锁来实现的。Java的线程是映射到操作系统的原生线程上的。如果要挂起或者唤醒一个线程，都需要操作系统来帮忙，操作系统实现线程间的切换需要从用户态转换到内核态，这个状态之间的转换需要相对较长的时间。因此早期synchronized效率较低。但是在Java6之后，java官方对synchronized在jvm层面进行了较大优化。JDK1.6对锁的实现引入了大量的优化，如自旋锁，适应性自旋锁，锁消除，锁粗化，偏向锁，轻量级锁等操作来减少锁操作的开销。

### 1.3.3如何使用synchronized关键字，项目中使用到了么？

synchronized关键字三种主要使用方式：

修饰实例方法：相当于对当前实例对象加锁，进入同步代码块之前要获得当前实例对象的锁。

修饰静态方法：相当于对当前类对象加锁，进入同步代码块之前要获得当前类对象的锁。

修饰代码块，指定加锁对象，进入同步代码块之前要获得指定加锁对象的锁。

项目中使用到了synchronized关键字，通过指定加锁对象修饰的代码块。第一处是有一个方法，返回所有学校列表信息，如果是第一次执行该方法，会将所有学校的所有网站的ipv6支持情况进行重新检测更新，并将更新结果存入缓存中，之后再调用该方法时，直接从缓存中读取数据，减少响应时间。使用synchronized锁定了SchoolSeviceImp这个类对象。如果有两个线程同时进行调用该方法时，只有一个线程能进入该代码块，进行所有学校网站信息的更新。另一个线程获得该类的对象时，此时是否为第一次访问标志位已经更改，该线程便直接从缓存中读取数据。保证了代码块的同步性。添加学校的方法里也使用synchronized关键字。锁的也是SchoolSeviceImp这个类对象，可能会同时存在两个线程执行添加操作，可能会导致数据的不一致，因此添加了同步代码块。添加学校的方法里主要是创建了一个学校的对象school，并将传进来的schoolName设置为学校名称，将ipv6等支持率设置为初值，其他一些参数也设置统一的初值，但是设置用户名称时，会先计算数据库中表的长度，再进行加一，user\_123这样，如果不进行同步操作，可能会导致两个学校出现相同的用户名。

单例模式中使用synchronized 单例模式：一个类只有一个实例，自行实例化，并向整个系统提供。

public class SingleTon{

//私有实例，需要使用volatile关键字，禁止指令重排

private volatile static singleTon uniqueInstance;

//私有构造函数

private SingleTon(){

}

public static SingleTon getUniqueInstance(){

if(uniqueInstance == null){

synchronized(SingleTon.class){

if(uniqueInstance == null){

uniqueInstance = new SingleTon();

}

}

}

return uniqueInstance;

}

}

### 1.3.4 synchronized关键字底层实现原理

(1) synchronized实现同步语句块时：

使用的是monitorenter和monitorexit命令。当执行monitorenter命令时，线程会尝试获取指定对象的锁，当指定对象的锁计数器为0，便可以获得该锁，并将指定对象的锁计数器加一。执行monitorexit命令时，锁计数器会被置为0，释放该锁，其他线程此时可以获取该对象的锁了。

(2)synchronized实现同步方法时：

使用标识ACC\_SYNCHRONIZED指明该方法为一同步方法，JVM通过ACC\_SYNCHRONIZED访问标识来辨别一个方法是否声明为同步方法。在调用时会先检查一个方法的ACC\_SYNCHRONIZED标识，如果该标识被设置，执行线程会先尝试去获取monitor，如果获取成功，则执行同步方法，否则，阻塞。在同步方法调用结束后，会自动释放monitor。

### 1.3.5jdk1.6之后synchronized关键字底层的优化

jdk1.6之后对锁的实现引入了大量的优化，比如偏向锁，轻量级锁，自旋锁，适应性自旋锁，锁消除，锁粗化等技术来减少锁操作的开销。锁主要存在四种状态：无锁状态，偏向锁状态，轻量级锁状态以及重量级锁状态。锁的状态会随着竞争的激烈而逐渐升级。锁可以升级而不能降级，这种策略是为了提高获得和释放锁的效率。

(1)偏向锁：偏向锁，会“偏心”第一个获得他的线程，如果没有其他线程来竞争该锁时，会直接消除掉同步。即如果该锁没有被其他线程获取，则持有偏向锁的线程将永远都不需要再进行同步。当有另外的线程尝试去获取这个锁时，偏向模式就结束。偏向撤销后，如果对象锁已经释放，则转为无锁状态，如果对象锁没有被释放，则转为轻量级锁的状态。场景：如果程序中大多数锁总是被不同的线程访问，则禁用偏向锁优化会提高性能。

(2)轻量级锁：轻量是相对于使用操作系统互斥量的传统锁而言。可以减少性能消耗。

轻量级锁的加锁和解锁否输采用了CAS操作。如果锁竞争激烈，轻量级锁会膨胀为重量级锁。场景：对于绝大部分锁，在整个同步期都是不存在竞争的。如果没有竞争，则使用CAS操作避免使用互斥操作的开销。如果竞争，除了互斥量的开销外还存在CAS的开销，会比传统的重量级锁更慢。(原理：虚拟机会在当前线帧里创建一个Mark word的拷贝lock record，虚拟机使用CAS操作尝试将Mark word更新为lock recod的指针，如果更新成功，则该线程获取了该对象的锁，如果更新失败，检查mark word是否指向当前线帧，如果执行，执行同步操作，如果不指向，说明该锁已经被其他线程抢占。 )

(3)自旋锁和自适应自旋

轻量级锁失败后，虚拟机为了避免线程真实的在操作系统层面挂起，还会进行一项称为自旋锁的优化手段。

一般线程持有锁的时间都不是很长，如果一个线程没有获取到锁，此时不会挂起，而是等待一段时间，让线程执行一个忙循环，看持有锁的线程是否释放了该锁。如果超过了自旋次数，还没有获取到锁，该线程就挂起。默认自旋次数是10。自适应自旋锁，自旋时间不固定，而是由前一次在同一个锁上的自旋时间以及锁的拥有状态来决定。如果在同一个锁对象上，自旋等待刚刚成功获得过锁，并且持有锁的线程正在运行，虚拟机认为此次自旋很有可能会成功，进而允许自旋等待较长时间。如果对于某个锁，自旋等待很少成功，就在获取这个锁的过程中省略掉自旋过程。

(4)锁消除

虚拟机即时编译器在运行时，对一些代码上要求了同步，但是被检测到不可能存在共享数据竞争的锁进行消除。

(5)锁粗化

如果虚拟机探测到有一串零碎的操作都对同一个对象加锁，会把加锁同步的范围扩展(粗化)到整个操作序列的外部，这样只需要加锁一次就可以了。减少频繁加锁解锁的性能消耗。

### 1.3.6 谈谈synchronized和Reentrantlock的区别

两者都是可重入锁。

synchronized是虚拟机层面的锁，而reentrantlock是API层面的锁，reentrantlock可以显式地进行加锁与解锁，支持中断，lock.lockInterruptibly()。在等待锁的线程可以放弃等待，转而处理中断事务。可以使用公平锁和非公平锁，可以实现选择性通知，通过condition来实现，唤醒时只唤醒在condition中注册了的线程。从而达到选择性通知的目的。而synchronized唤醒时只能选择唤醒一个线程或者唤醒所有线程，。而synchronized则不行。

### 1.3.7 java内存模型

jdk1,2之前，java内存模型实现总是存主存(共享内存)中读取变量，此时不需要注意同步，但是在当前内存模型中，线程可以把变量存储在本地内存中，此时可能有别的线程对主存中的变量进行修改，而当前线程还是读取的本地内存中的变量旧值，此时便存在数据不一致的问题。要解决这个问题，使用volatile关键字。当一个变量被volatile关键字修饰后，便要求JVM每次读取时都从主存中去进行读取该值，保证该变量的可见性。volatile的另一个作用是防止指令重排。

### 1.3.8 volatile和synchronized区别

volatile只能保证数据的可见性，不能保证数据的同步性，synchronized两者都能保证。volatile只能修饰变量，synchronized可以修饰方法以及代码块。volatile不会引起阻塞，而synchronized会引起阻塞。

### 1.3.9 为什么要使用线程池

降低资源消耗：通过重复利用已经创建的线程，降低创建线程销毁线程带来的资源消耗。

提高响应速度：不用重新创建线程，可以使用已经存在的线程，提高响应速度。

提高线程可管理性：可以对线程进行统一的调配管理使用。

### 1.3.10实现Runnable和Calllable接口的区别

实现了这个两个接口的实现类都可以被ThreadPoolExecutor或者ScheduledThreadPoolExecutor执行，两者的区别在于Runnable不会返回结果，但是Callable会返回结果。

### 1.3.11执行execute()方法和submit()方法的区别

执行execute()方法用于提交没有返回值的任务，submit()方法用于提交有返回值的任务。线程池会返回一个future类型的对象，通过这个future对象可以判断任务是否执行成功。通过futuren的get()方法来获取返回值。get()方法会阻塞当前线程直到任务完成。

### 1.3.12如何创建线程池

a.通过构造方法ThreadPoolExecutor。项目中使用的就是直接通过构造函数创建线程池。创建了两个线程池，WEBSITE\_ADD\_POOL和WEBSITE\_UPADTE\_POOL。网站批量更新时，需耗费较长时间，因此使用线程池来异步地进行更新。重写了参数ThreadFactory的newThread方法，包括设置线程的名称，是否为守护线程。

b.通过Executor框架的工具类Executors。这些方法的内部实际上是调用了ThreadPoolExcutor的构造方法。

可以创建三种类型的ThreadPoolExecutors

FixedThreadPool：该方法返回一个固定大小的线程池。当有任务过来时，如果线程池中可用的线程，则直接执行任务，否则将任务暂存在任务队列中等待有线程空闲。Executors.newFixedThreadPool()

SingleThreadPool：返回只有一个线程的线程池

CachedThreadPool：返回一个可以根据时机情况调整线程数量的线程池。当线程池中没有可用线程时，会创建新的线程。

### 1.3.13介绍一下Atomic原子类

原子类就是具有原子操作特征的类。原子性：一旦操作开始就不可被中断。

原子类在包java.util.concurrent.atomic下。包括常用的AtomicInteger，AtomicBoolean,AtomicIntegerArray,AtomicReference等。

### 1.3.14JUC包中的原子类是哪些

基本类型

AtomicInteger,AtomicBoolean,AtomicLong

数组类型

AtomicIntegerArray,AtomicLongArray,AtomicReferenceArray

引用类型

AtomicReference,AtomicStampedReference(指该引用类型里的字段会被进行原子更新)AtomicMarkableReference(指带有标记为的引用类型会被原子更新)

对象的属性修改类型

AutomicIntegerFieldUpdater(原子更新整形字段的更新器)

AtomicStampedReference:原子更新带有版本号的引用类型。该类将整数值与引用关联起来，可用于解决原子的更新数据和数据的版本号。

### 1.3.15 AtomicInteger的使用

主要是一些AtomicInteger的方法

getAndIncrement()获取当前值并且自增

CompareAndSet()如果输入的值等于预期值，则以原子方式将该值设置为输入值

项目里在几个地方用到了AtomicInteger一是在创建线程池，实现 ThreadFactory接口时，定义了一个AtomicInteger类型的count，用于线程的名称的一部分：count.IncrementAndGet()

### 1.3.16 AtomicInteger类的原理

AtomicInteger利用CAS+volatile和native方法来保证原子性，从而避免了synchronized的高开销，执行效率大大提升。

### 1.3.17 AQS介绍

AQS AbstractQueueSynchronizer 抽象队列同步器，在java.util.concurrent.locks包里。**AQS是一个用来构建锁和同步器的框架，**使用AQS能够简单高效地构造出应用广泛的大量同步器。比如ReentrantLock，Semaphore。也可以根据自己需求构建出属于自己的同步器。

### 1.3.16AQS原理分析

AQS的核心思想是，如果共享资源空闲，线程可以直接操作该共享资源，否则就需要一套**线程阻塞等待以及被唤醒**时**锁分配的机制**。这个机制AQS是使用CLH队列实现的，将暂时获取不到锁的线程加入到队列中。

CLH是一个虚拟的(虚拟双向队列即不存在队列实例，仅存在节点之间的关联关系)双向队列，CLH将每一个请求资源的线程封装成一个CLH锁队列的一个结点(node)来实现锁的分配。AQS通过使用一个int成员变量state来表示同步状态。

### 1.3.17 AQS对资源的共享方式

两种 独占：只有一个线程能执行，又可以分为公平锁和非公平锁。比如reentrantlock。

共享：多个线程可以同时执行，Semaphore。

### 1.3.18 进程和线程的区别

线程定义：进程的一个执行单元，是比进程更小的独立运行的基本单位，是轻量级的进程。

进程定义：并发执行的程序在执行过程中分配和管理资源的基本单位。CPU调度资源的基本单位。

关系：一个程序至少有一个进程，一个进程至少有一个线程。

区别：

1. 线程之间共享内存，线程自己几乎不拥有系统资源，都是共享的，进程有属于进程自己的系统资源，进程之间不共享内存。
2. 进程崩溃后，其他进程不受影响，而一个线程崩溃后，整个进程都会崩溃。
3. 进程进行切换时，资源消耗大，效率低，需要频繁切换时，线程优于进程。
4. 进程有一个运行的入口，顺序执行序列和程序的入口，但线程必须依存于应用程序中。

优缺点：线程执行开销小，不利于资源管理和保护。而进程则相反。

### 1.3.19 创建线程的几种方式

1.直接继承Thread类

在Thread类里重写方法run

Thread thread = new MyThread();

thread.start();

2.实现Runnable接口,重写run方法，创建Thread，将实现Runnable接口的类myTask作为参数。启动thread。

class myTask implements Runnable{

@Override

public void run(){

//do something

System.out.println(“doing”)

}

}

Thread thread = new Thread(myTask);

thread.start();

1. 实现Callable接口，重写方法call，注意返回值，需要和FutureTask的返回值一致，使用实现了Callable的类MyTask作为FutureTask的构造方法的参数，创建future，再将future作为Thread的参数，创建Thread。启动

class MyTask implments Callable{

@Override

public String call() throws Exception{

//do something

System.out.println(“doing”)

}

}

MyTask myTask = new MyTask();

FutureTask<String> future = new FutureTask<Strring>(myTask);

Thread myThread = new Thread(future);

myThread.start();

### 1.3.20 Condition

用法：和重入锁搭配使用，实现wait和notify的功能。

Java高并发程序设计P82

重入锁和Condition为全局变量。写在实现Runnble的类里，run方法里首先获取锁lock.lock，再使用condition.await()。在finally语句里释放锁lock.unlock();

创建方法：使用重入锁来创建

lock = new ReentrantLock();

condition = lock.newCondition();

唤醒时使用condition.signal()唤醒

### 1.3.21 Semaphore 信号量

用法：可以指定一个或多个线程，同时访问某一个资源

构造函数:

public Semaphore(int permits)

public Semaphore(int permits,boolean fair)

参数permit为能够获取准入许可的线程数

方法acquire()线程尝试获取一个准入的许可。无法获得许可时，线程会等待，直到有线程释放一个许可或者当前线程被中断。

方法release()线程再访问资源结束后释放一个许可

在runnable方法里使用semp.acquire()来获取一个许可。

## IO

### 1.4.1FileChannel(文件通道)

包:java.nio，属于NIO的一种，FileChannel是阻塞的

FileChannel fileChannel = new RandomAccessFile(new File(“db.data”), “rw”).getChannel();

//写入

byte[] data = new byte[4096];

long position = 1024L;

//通过ByteBuffer将原有数据打包，并写入到文件通道的指定position

fileChannel.write(ByteBuffer.wrap(data),position);

//读入

ByteBuffer buffer = ByteBuffer.allocate(4096);

long position = 1024L;

//从指定的fileChannel位置读取出数据到buffer中

fileChannel.read(buffer,position);

//从当前文件指针的位置读取4kb的数据

fileChannel.read(buffer);

内部通过private final Object positionLock = new Object()；锁来控制并发

FileChannel把ByteBuffer中的数据写入PageCache,这是内存和磁盘之间的一层缓存。FileChannel提供了一个force()方法，用于通知操作系统进行及时的刷盘。

1.4.2 Mmap读写

MappedByteBuffer mappedByteBuffer = fileChannel.map(FileChannel.MapMode.READ\_WRITE,0,filechannel.size());

## JVM

### 1.5.1 JVM分区

Java的内存一般被分为5个区域：程序计数器、堆区、方法区、本地方法栈、虚拟机栈区。

程序计数器：线程私有，可以表示程序进行到了哪一行

堆区：共享，存放对象实例

方法区：共享，存放类信息，常量，静态变量以及即时编译器编译后的代码等数据。

虚拟机栈：线程私有，存放表示方法执行内存模型的栈帧，栈帧里包括：局部变量表，操作数栈，动态连接，方法出口。

本地方法栈：和虚拟机栈类似，但是是为本地方法服务。

### 1.5.2 JVM回收算法和回收器，CMS采用哪种回收算法，怎么解决内存碎片问题？

回收算法：标记-清除算法：先将可以进行回收的对象进行标记，再清除这些对象。缺点：效率不高，清楚后可能有很多内存碎片，造成内存不连续。

复制算法：基础的复制算法：将内存区域分成两块，当第一块中有对象不能被回收时，将这些对象复制到第二块内存中，将第一块内存中所有的对象全部回收，解决了标记-清除算法带来的内存不连续问题。缺点是可以使用的内存少了一半。且在对象存活较多的情况下，需要有较多的复制操作，效率会变低。

标记-整理算法：将所有存活的对象进行标记，移到内存的一端，再清理掉边界之外的内存中的对象。分代：老年代：标记-整理 新生代：复制算法。

回收器：Serial ParNew Parallel Scavenge (新生代)

CMS serial old parallel old 老年代

G1

CMS基于标记-清除算法，CMS在进行一定次数的FullGC(标记-清除)时候，进行一次标记-整理算法。UseCMSCompactAtFullCollection

### 1.5.3 类对象的的生命周期以及何时被回收了

类对象的生命周期：加载，验证，准备，解析，初始化，使用，卸载

验证，准备，解析可以统称为连接。

**加载**：将类的class文件加载到内存中，并创建一个类对象

加载是通过虚拟机的加载器进行的，开发者也可以通过ClassLoader创建自己的类加载器。类加载可以通过本地文件，jar包，网络进行加载。

类加载时虚拟机做的事情：通过一个类的全限定名来获取定义此类的二进制字节流。将这个二进制字节流代表的静态存储结构转换为方法区的运行时数据结构。

在内存中生成一个代表该类的java.lang.Class对象，作为方法区这个类的各种数据的访问入口。

**验证：**检验被加载的类是否具有正确的内部结构，并和其他类协调一致。

文件格式验证：验证字节流的格式是否符合Class文件格式规范，包括主次版本号是否符合虚拟机处理要求。常量池中是否有不被支持的常量类型，指向常量的索引值是否存在不存在的常量或者不符合类型的常量。

元数据验证：对字节码描述的语义信息进行分析，检查是否符合java语法规则。

字节码验证：最重要的验证环节，分析数据流和控制，检查语义是否合法，是否符合逻辑。对元数据验证之后，方法体的验证，以保证类方法在运行时不会有危害出现。

符号引用验证：针对符号引用转为直接引用，对类自身之外的信息(常量池中的符号引用)进行匹配校验。

**准备：**类准备阶段负责为类的静态变量分配内存，并设置默认初始值

**解析：**将类的二进制数据中的符号引用转换为直接引用(指向目标的指针，偏移量，直接定位的句柄)。

**初始化：**给类的静态变量赋实际初始值

**回收的类必须满足三个条件：**

该类的所有实例都已经被回收，Java堆中不存在该类的任何实例

加载该类的ClassLoader已经被回收

该类对应的java.lang.Class对象没有在任何地方被引用，无法在任何地方通过反射访问该类。

1.5.4

# 计算机网络

## 2.1 TCP和UDP的区别

TCP是面向连接的，可靠的传输协议。在传输数据前必须先建立连接，数据传输结束后要释放连接。数据传输时，有确认，窗口，重传，拥塞控制等机制，传输效率较低，所需资源较多。一般用于邮件或者文件传输，远程的登录等场景。

UDP是无连接的，不可靠传输协议。在传输数据前不需要建立连接。没有TCP那些确认控制机制。传输效率高，所需资源少。一般用于实时性要求较高而可靠性要求不那么高的场景，比如语音，视频等。

## 2.2 浏览器输入url地址->显示主页的过程

输入地址后，浏览器先会查找域名的IP地址，这时候会用到DNS协议用以获取对应的IP；找到对应IP地址后，浏览器会向web服务器发送一个HTTP请求，发送请求前，先会建立TCP连接，建立TCP协议时，需要发送数据，发送数据在网络层使用IP协议，网络层的数据传输需要用到一些路由协议比如OPSF，来进行路由选择，在进行路由选择和实际发送数据之前，需要用到ARP协议，ARP协议将IP地址映射到MAC地址，根据对应的MAC去寻找对应的路由器和主机。在TCP建立连接完成后，使用HTTP协议访问网页。服务器处理请求，发回一个HTML响应，浏览器开始显示HTML。

OPSF：链路状态协议：通过 路由器之间交换链路状态信息，建立一个链路状态数据库，再通过Dijkstra最短路径算法计算自己到个目的网络的最优路径，只存下一条。

RIP：路径向量协议：通过询问邻居的路由器得到到达每个节点的最短路径，维护一个向量表，但是最多只能存15跳，超过15跳则认为不可达，而且存在坏消息传得慢等缺点。适用于小型的网络。

## 2.3 DNS过程

1.先去浏览器缓存以及系统缓存里去寻找有没有对应的IP地址，有就直接用。

2.如果没有，就发给本地域名服务器，如果找到就结束，没有找到就向根域名服务器，根域名服务器告诉本地域名服务器应该去哪一个顶级域名服务器中找。

3.顶级域名服务器告诉本地域名服务器对应的IP或者告诉本地域名服务器应该去哪里找这个IP地址。这样一级一级找下去，最终找到对应的IP，返回给本地域名服务器，本地域名服务器再告诉主机这个域名对应的IP地址是多少。

4.本机知道IP之后，会在浏览器和系统缓存中增加这个域名对应的IP地址，至此DNS过程结束。

## 2.4 TCP三次握手

三次握手：

1. 客户端向服务端发送TCP连接请求，SYN位置1表示这是一个连接请求，该TCP报文的序号seq=x,ACK=0；
2. 服务端响应客户端TCP请求，发送一个确认报文，SYN位置1，该TCP报文序号seq=y，确认序号为x+1，表示之前客户端发送给他的x序号及之前的报文他都已经收到了。确认位ACK为1。
3. 客户端响应服务端的TCP请求，发送一个确认报文，该TCP报文序号为seq=x+1，确认序号为y+1，表示服务器给他的y序号以及之前的报文他都收到了，确认位ACK为1。

## 2.5 三次握手能不能是两次或者四次？

不能为两次，如果不要最后一步，服务端不知道客户端是不是接收正常的，也不知道自己是不是发送正常的。

不能为四次，三次就够了，四次浪费。

## 2.6 四次分手

a.客户端发送终止发送数据报文，终止位FIN=1，序号seq=x。

b.服务端响应，发送确认报文，报文序号为seq = y,确认位ACK为1，确认序号x+1，表示客户端发送的序号x及之前的的数据报文他都已经收到了。此时客户端到服务器端的连接已经释放了。

c．如果服务端也不需要向客户端发送数据，也会向客户端发送一个终止发送数据的报文FIN为1，报文序号为k,确认位ACK为1，确认序号为x+1。

d．客户端响应服务端，发送一个确认报文，报文序号为x+1，确认位ACK为1,确认序号为k+1。

还要等的2MSL,才能释放连接。

为什么是2MSL.因为要确认客户端发送给服务器的ACK报文已经传给了服务端，而且服务端也已经接收到了ACK报文。如果ACK没有收到，服务端会发送一个超时重传报文。最少是超时时间+FIN的传输时间，保证可靠，使用2MSL

MSL：报文段最大生存时间。

2.7 流量和拥塞控制

滑动窗口来控制流量，

## 2.8 http和https的区别

https协议需要到ca申请证书，需要一定的经济成本

http是明文传输，https是加密的安全传输

连接端口不一样，http是80,https是443

http连接简单，没有状态，https是ssl加密的传输，身份认证的网络协议，相对http传输比较安全。

## 2.9 http状态码

200：客户端发过来的请求在服务端被正确地处理

302：临时性重定向，表示资源临时被分配了新的URL。浏览器在拿到服务器返回的这个状态码之后会自动跳转到一个新的URL地址，这个地址可以从响应的Location首部中获取(用户看到的效果就是他输入的地址A瞬间变成了另一个地址B)

403：对请求资源的访问被服务器拒绝。表示没有权限访问此站。服务器理解了该请求但是拒绝执行该任务。

404：表示在服务器上没有找到请求的资源。

500：服务器在执行请求时发生了错误。

# Linux

3.1 使用过哪些linux命令？

3.2 如何登录远程服务器？

3.3 如何在一个文件中查找指定字符串？

# Mysql

4.1一条Sql语句

# Java web编程

## 5.1 Cookie和Session的区别

1.Cookie存放在客户端。保存的数据有类型和大小限制，保存的数据不能超过4k，只能存放字符串。Google浏览器的存放位置C:\Users\Administrator\AppData\Local\Google\Chrome\User Data\Default\Cache，Session存放在服务端，可以存放在文件，数据库里等。没有具体的内存大小限制。

2.

# JavaEE

## 6.1 Spring

6.1.1项目中使用过哪些注解？

1.Bean注入相关的注解

@Service

写在Service实现类的上面。作用在类上，其作用域默认为单例模式。被@Service注解标注的类会被Spring扫描并注册为Bean。给定名字注册的bean名称就为指定名称，否则默认为当前类名。

@Entity和@Id

该注解是在实体类上(School)

@Id注解标注在实体类中的字段，主键字段标注为@Id

@Component

作用在类上。默认作用域为Singleton。使用该注解的类会被Spring扫描并注册为Bean

1. Controller相关

@Controller

被@Controller标注的类为一个控制器，用于处理http请求。分发处理器会扫描使用该注解的类的方法，并检测该方法使用使用了@RequestMapping注解

@RestController

@ResponseBody和@Controller的结合。@ResponseBody告诉控制器该类的方法返回对象都会被自动序列化为JSON。并且传回HttpResponse这个对象。

@RequestMapping

用来配置url隐射

1. 取值

@RequestParam获取请求参数的值

1. 日志相关

@Slf4j

直接写log.info()等

## 6.2 Hibernate

## struts

## 6.4 Mybaits

## MVC

### 6.4.1一个请求到了SpringMVC流程是什么样的？

1.前端控制器DispatcherServlet接收请求。

2.HandlerMapping解析请求对应的Handler，交给HandlerAdapter处理

3.HandlerAdapter会按照特定的规则去执行Handler，此时会处理业务逻辑，与Model层有交互

4.处理器返回一个模型视图ModelAndView

5.ViewResolver会根据逻辑View查找实际的view

6.DsipatcherServlet把返回的Model传给View

7.把View返回给请求者

## 6.6 JPA

# 操作系统

### 7.1典型的调度算法

先来先服务：就是选择就绪队列中在队头的进行进行先处理。不能作为分时系统和实时系统的主要调度策略。如果一个进程的处理时间太长，可能造成后面的进程等待时间太长。

短进程调度算法：选择等待队列中估计运行最短的进行，将处理机分配给改进程，让它开始执行。该算法对长进程不利，完全未考虑进程的紧迫程度。SIJ调度算法的平均等待时间，平均周转时间最少。

优先级调度算法：每次先选择优先级最高的进行，分配处理机，使之执行。

高相应比优先调度算法：相对于短进程调度算法，考虑到了等待时间，在相同的等待时间里，优先考虑执行时间最短的进程。

时间片轮调度算法：适用于分时系统，与先来先服务算法一样，每次先运行在队头的进程，但是只分配给该进程一段时间片，比如100ms,时间结束，不管进程有没有执行完毕，都必须释放出处理机给下一个就绪进程，并到队列队尾进行排队。

多级反馈队列调度算法：是时间片轮调度算法和优先级算法的综合，通过动态地调整进程优先级和时间片大小来兼顾多方面的系统目标。首先分为多级队列，每个队列的优先级不一样，队列的分片时间随着优先级的的降低而增加，一个进程进来之后，首先将其放入第一级队列的末尾，按先到先服务的原则等待调度，如果在一个时间片里该进程执行完毕，便准备撤离系统，如果它在一个时间片结束后还没有完成，就将该进程转入到第二级队列末尾，依次进行，如果到了第n级队列，还没有完成，就采用时间片轮转的方式。

# 设计模式

设计模式：一套被反复使用的代码设计经验总结，针对情景中的一个问题经过正式的一个解决方案。使用设计模式为了可重用代码，让代码更容易被他人理解，保证代码的可靠性。

分三类：创建型，指对类的实例化过程的抽象化，结构型：描述如何将类或对象结合在一起形成更大的结构，行为型：对在不同的对象之间划分责任和算法的抽象化。

## 8.1单例模式

单例模式：一个类只有一个对象，自行实例化，向整个系统提供

**双重锁判断机制：**JVM底层模型原因，偶尔会出问题。

public class SingleTon(){

private volatile static SingleTon uniqueInstance;

private SingleTon(){

}

public static SingleTon getInstance(){

if(uniqueInstance == null){

synchronized(SingleTon.class){

if(uniqueInstance == null){

uniqueInstance = new SingleTon();

}

}

}

return uniqueInstance。

}

}

**懒汉式：**直接给静态get方法加锁。

**饿汉式：**直接创建私有对象，线程安全，调用效率高，但是不能延时加载。可能会造成不必要的资源浪费。

**静态内部类：**线程安全，调用效率高，可以延时加载。

public class SingleTon{

private SingleTon(){

}

private static class singleTonClass{

private volatile static SingleTon uniqueInstance = new SingleTon();

}

public static SingleTon getInstance(){

return singleTonClass.uniqueInstance;

}

}

## 8.2 代理模式

代理主要功能：管理和控制对当前对象的访问。解决问题：不能直接访问该对象，比如说远程访问。

模式结构：代理类和被代理类实现同一个接口，用户访问的时候先访问代理对象，然后再让代理对象去访问被代理对象。框架里使用Spring里的AOP

代理类：会在代理类里创建被代理的对象。

代理分为：远程代理：控制访问远程对象，虚拟代理：控制访问创建开销大的资源和保护代理：基于权限控制对资源的访问。

JDK里使用:java.lang.reflect.proxy

**动态代理：**在使用时才被动态创建的代理，使用了反射机制。

过程：

1. 定义一个委托类和公共接口，委托类和代理都实现该公共接口。
2. 定义一个调用处理器类，该类实现了InvocationHandler接口，代理中的具体方法的执行都是在该调用处理器类中完成的。代理类调用任何方法都会经过这个调用处理器类。(method.invoke())
3. 生成代理对象，需要为其指定委托对象，该对象是西安的一系列接口，调用处理器类的实例。

主要涉及的类：java.lang.reflect.Proxy生成代理类的主类，java.lang.reflect.InvocationHandler

## 8.3观察者模式

定义了一系列对象之间一对多的关系，当一个对象状态改变时，其他依赖者都会收到通知。当主题里的数据发生改变时，所有订阅了主题的观察者都会收到通知。

过程：

可观察者继承obervable类，该类中定义数组，数组里存放在该类中注册了的观察者。该类中有注册，删除观察者，通知观察者等方法，当发生变化时，调用通知观察者方法，通知所有观察者自己的数据已经改变(循环调用观察者的update方法)。

观察者实现Observer接口，该接口里有update方法，观察者的构造函数中传入Observable类的继承类，在更新之前先调用Observable的注册方法，将当前Observer注册到Observable中。update方法中，检查是否被通知修改参数，如果通知了，就将参数修改为通知的参数。

## 8.4 (抽象)工厂模式

抽象工厂和工厂方法模式的区别

工厂模式的思想

## 8.5 MVC设计模式

MVC模式由三个子系统组成，分别是Model、View、Controller。

Model：用来存储数据要素，扮演数据库的角色，并不依赖于View和Controller而存在。

View：程序的界面显示部分，用于和用户进行交互

Controller：负责处理用户的动作，必要时更新Model和View。

为什么要使用MVC模式

多个view能共享一个Model:同一个web应用程序会提供多种用户界面，比如手机端，电脑端等。在MVC设计模式中，Model响应用户请求并返回响应数据，View负责把他们呈现给用户，业务逻辑和表示层分离。同一个Model可以被不同的View重用，大大提高了代码的可重用性。

方便地改变应用程序数据层和业务规则：如果需要移植数据库，只需要修改Model即可。不需要修改其他两个模块

Controller提高了应用程序的灵活性和可配置性

# 算法与数据结构

# 消息队列

# 场景题

# 公司面试题准备

1.自我介绍：面试管你好，我叫陈羽茜，今年24岁，本科毕业于武汉理工大学，现在是华中科技大学计算机技术研一在读。研究生期间，我参加了一个学校网络中心的项目，项目主要负责监测湖北省高校的IPv6支持情况，我主要负责后台代码的编写，在实现基础的添加删除更新网站及学校信息之后，针对项目中出现的一些问题，使用缓存机制以及线程池，提高了系统响应时间，使用AOP实现了日志和缓存的统一管理，使用JWT和拦截器保证了登录的安全。我主要使用的计算机语言是Java，熟悉java的基础知识，了解java虚拟机，熟悉计算机网络的知识。希望自己能满足公司的要求。

1. TCP三次握手

第一步：客户端向服务端发送一个连接请求，该报文头部同步位SYN为1，确认位ACK为0。同时客户端会随机选择一个报文的起始序列号seq=x。

第二步：服务端响应客户端，如果同意建立连接，就向客户端发回确认，并为该TCP连接分配TCP缓存和变量。报文头同步位SYN为1，确认位ACK为1，确认号字段为x+1。并且服务器也会随机选取一个起始序号：seq=y。

这一步可以确认服务端可以响应客户端的消息。

第三步：客户端收到确认报文后，还要向服务器给出确认，也要分配TCP缓存和变量。这个报文的ACK为1，序号为x+1，确认号字段为y+1。

这一步表示客户端可以成功响应服务端的消息。

确认号字段：期望收到对方的下一个报文段的数据的第一个字节的序号。若确认号为x+1，表明之前到序号x的报文都已经成功收到。

不能改成两次，两次只能确认服务端可以响应客户端，但是客户端是否可以响应客户端呢？还不能确定。

不能改成四次，因为三次就可以确认对方是否能响应自己了，改成四次造成了浪费。

1. TCP的四次挥手是什么？能不能合成3次？

四次挥手是指客户端和服务器关闭TCP连接的过程。

第一步：客户端打算关闭连接时，发送一个终止传送数据的报文，报文头中终止位FIN被置为1，报文序号为u，u是这个报文发送之前已经成功传送的最后一个报文的序号+1。

第二步：服务器收到连接释放报文后，发出确认报文，确认号是u+1，该报文的序号为v。v等于这个报文发送之前服务端成功发送的最后一个报文的序号加一。此时客户端向服务器的连接已经关闭，客户端已经不能向服务端发送数据报文。

第三步：如果服务端不需要给客户端发送数据，服务端也会向客户端发送一个终止位FIN为1的报文。

第四步：客户端给出确认报文，确认号为w+1。序号为u+1。此时TCP连接还没有释放掉，必须经过2MSL之后连接才释放。

如果客户端要求和服务端断开连接时，服务端此时没有数据需要传输，则可以直接向客户端发送关闭连接请求与响应。相当于第二步和第三步合并。

因此可以合并成三步。

1. 求整数数组的最大子数组和
2. B+树和B树的区别？什么情况下可以用B树？

相同点：都是平衡多路查找树

不同点：m阶(所有节点的孩子节点数的最大值称为该树的阶)的B树和B+树，每个节点中关键字的个数不相同：B树节点的关键字最多为m-1，而B树中节点的关键字最多为m。

关键字的含义也不同：B树中一个节点的子节点中所有关键字大于指向该子节点的指针的前一个关键字，小于后一个关键字。而B树中一个节点的子节点树小于等于关键字。

叶节点不同：B树的叶节点里不包含任何信息，但是B+树的叶节点里包含所有的关键字的具体存储位置。查找时，B树可能终止于非叶节点，但是B+树只能终止与叶节点。

比如数据库，他的索引结构就是用B树。

1. 给两个无序链表排序？
2. 说一下数据库的四种隔离级别？

第一级别：允许读取还没有提交的数据，可能会造成脏读

第二级别：允许读取并发事务已经提交的数据，可以阻止脏读，但是不可重复读和幻读仍有可能会发生

第三级别：可重复读，对同一字段的多次读取结果是一致的，可以阻止脏读和不可重复读，但幻读仍有可能会发生。

第四级别：最高的隔离级别，完全服从ACID的隔离级别。所有的事务依次逐个执行，事务之间完全不能产生干扰，该级别可以防止脏读、不可重复读以及幻读。

1. 甲先抛硬币，乙再抛硬币，谁先抛到正面，谁赢，求甲赢的概率？

考虑在所有次数下，甲均赢的概率，将这些相加，得到甲赢的概率

1. 介绍一下项目，在项目中做了写什么？

这个项目主要是显示湖北省高校网站的IPv6 IPv4的支持情况。我在这个项目中主要负责写后台代码。这个项目底层是通过脚本测试网址得出结果，再将这些结果显示出来。主要使用了两张表，一张表是网站表，还有一张是学校表。其中一个学校有一个或多个网站。最开始把基本的增删改查功能做完之后，发现如果重置一个学校的所有网站信息，或者添加网站后，等脚本跑完，响应会很慢，就思考能不能异步来做这个网站检测。当所有网站的信息都更新完成之后，再提示刷新。后来就使用线程池实现了这个异步的功能。还使用了本地缓存减少响应时间，登录时使用JWT和拦截器来保证登录的安全。

1. 说说equals和==的区别
2. 说说synchronized和lock的区别

synchronized是jvm层面实现的锁，而lock是一个java类，是API层面的。

synchronized无法中断，只能实现非公平锁，而lock锁可以中断，可以实现公平锁。

synchronized获得锁和释放锁都是隐式的，但是lock获取锁和释放锁都是显式实现的。

1. 类对象的生命周期，什么时候回收？
2. 数据库：

场景：学生属性：姓名，学号，班级，性别，成绩

1. 查找学号为\*\*\*\*的学生的成绩

select grade from students s where s.id = 12243;

1. 查询学号为\*\*\*、性别为\*\*的学生的成绩

select grade from students s where s.id=111 and s.sex=’女’;

要建立什么样的索引？

13算法题：买股票最佳时机：一次买进卖出，求最大利益(leetcode121)

14 java7 java8的区别

lambda表达式

15 java内存模型

jdk1.2之前读取变量数据都是从主存里进行读取的，主存是共享的，jdk1.2之后，线程可以把变量保存在本地内存中，有可能其他线程已经修改了主存中该变量，但是本线程还是读取的是本地内存中的旧数据，就可能会造成数据不一致。

16.CAP是什么？

指在一个分布式系统中，一致性，可用性，分区容忍性这三个要素最多只能实现两点，不能三者兼顾。

17.CAP能同时到达么？能达到什么？为什么？

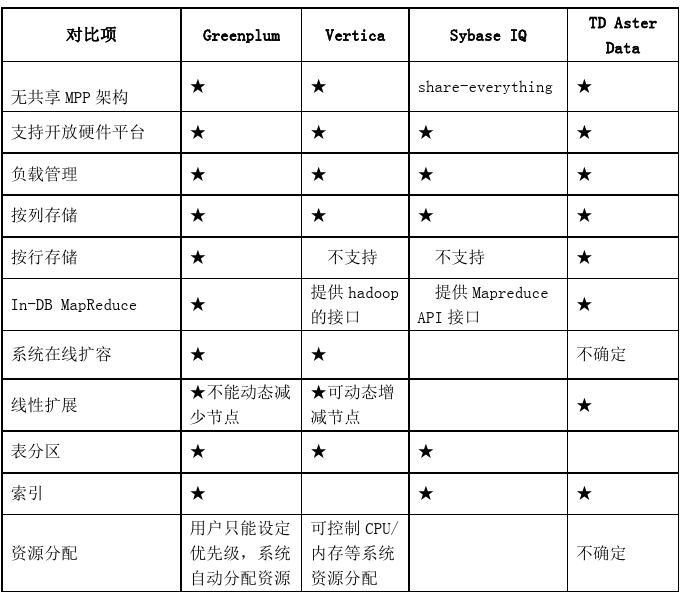
不能同时达到。C是指一致性，表示分布式系统中所有数据备份，在同一时刻是否是同样的值。A是指可用性，一个集群部分节点发生故障之后，集群整体是否还能响应客户端的读写请求。P指分区容忍性，系统如果不能在时限里达成数据一致性，就意味着发生了分区的情况。此时必须在C和A中做出选择。CAP原则的精髓就是要么AP，要么CP，要么AC，但是不存在CAP。如果在某个分布式系统中数据无副本， 那么系统必然满足强一致性条件， 因为只有独一数据，不会出现数据不一致的情况，此时C和P两要素具备，但是如果系统发生了网络分区状况或者宕机，必然导致某些数据不可以访问，此时可用性条件就不能被满足，即在此情况下获得了CP系统，但是CAP不可同时满足。必然导致某些数据不可以访问，此时可用性条件就不能被满足，即在此情况下获得了CP系统，但是CAP不可同时满足[1] 。

因此在进行分布式架构设计时，必须做出取舍。当前一般是通过分布式缓存中各节点的最终一致性来提高系统的性能，通过使用多节点之间的数据异步复制技术来实现集群化的数据一致性。通常使用类似 memcached 之类的 NOSQL 作为实现手段。虽然 memcached 也可以是分布式集群环境的，但是对于一份数据来说，它总是存储在某一台 memcached 服务器上。如果发生网络故障或是服务器死机，则存储在这台服务器上的所有数据都将不可访问。由于数据是存储在内存中的，重启[服务器](https://baike.baidu.com/item/%E6%9C%8D%E5%8A%A1%E5%99%A8/100571)，将导致数据全部丢失。当然也可以自己实现一套机制，用来在分布式 memcached 之间进行数据的同步和持久化，但是实现难度是非常大的[2]

18.讲一下你知道的分布式数据库

hbase、redis、MongoDB、cassandra，solr

Greenplum，Vertica,SybaseIQ TD Aster Data



19.你们的项目满足CAP么？怎么做到的？

20.逻辑始终了解么？有什么用？

21.讲一下数据库ACID什么意思？

数据库事务正确执行的四个基本要素。

A表示原子性：表示事务中的操作，要么全部完成，要么就一个操作都不做，不能停在中间某个环节。如果事务在执行过程中发生错误，会被回滚到事务开始前的状态。

C指一致性：事务执行前后，数据都具有一致性。两个或多个线程在事务发生前后读取这个数据，得到的结果都是一致的。

I隔离性：指多个用户并发访问数据库时，数据库为每一个用户开启的事务，不能被其他事务的操作数据干扰，多个并发事务之间要相互隔离。

D持久性：只一旦一个事务提交，它对数据库的影响应该是永久的。

27. 索引的底层实现：B+树

MyISAM索引实现：使用B+树作为索引结构，叶节点data域存放的是数据记录的地址。

28. B+树和红黑树的区别

29. 数据库引擎有哪些，什么特点？innoDB 和myISAM

不同的存储引擎提供不同的存储机制，索引技巧，锁定水平等。

存储引擎：InnoDB,MyISAM

30.状态码401，403

状态码403表示服务器理解了这个请求，但是由于权限，拒绝处理用户的这个请求。

401：表示请求里需要有通过Http认证的认证信息

31.快排算法

32.数据库查询语句，多表从查询。

33.数据库mysql索引最左匹配原则？

22.dns解析的过程

浏览器先检查自身缓存以及操作系统缓存中有没有域名对应的IP,如果有，则直接使用，如果没有，就请求本地域名服务器来解析这个域名，本地域名服务器如果没有命中，就跳到根域名服务器，它会返回给本地域名服务器一个所查询域的主域名服务器(国际顶尖域名服务器)，此时本地域名服务器会再发送请求给上一步返回的主域名服务器，接收请求的主域名服务器会查找并返回这个域名对应的Name Server的地址，这个Name Server就是网站注册的域名服务器。Name Server根据映射关系表找到目标IP，返回给本地域名服务器。本地域名服务器将其加入到缓存中。本地域名服务器把解析的结果返回给用户，用户根据TTL值缓存到本地系统缓存中。

23.url->页面的过程

浏览器通过url去寻找url对应的服务器地址，此时需要使用DNS域名解析协议，解析到服务器的IP地址之后，浏览器已一个随机端口向服务器的80端口发送一个TCP连接，建立TCP连接之后，发送一个HTTP请求。服务器处理请求并返回HTTP报文，浏览器解析渲染页面，连接结束。

24. tcp和udp对比

tcp是面向连接，可靠的数据传输协议

udp是无连接的，不可靠的数据传输协议

tcp需要连接，且在数据传输时的确认机制，拥塞控制等等会导致使用tcp传输数据的效率较低，消耗资源大，对于一些实时性要求高，可靠性要求低的情况，比如视频等，使用的是UDP连接，但是对于一些可靠性要求高的应用比如邮件传输，文件传输，访问网站等使用的是TCP连接

6.1.0 5个人去一个海岛寻宝，最后一共找到了100枚金币。他们约定了一个分配方案。

6.1.1 给你一个有序整数数组，数组中的数可以是正数、负数、零，请实现一个函数，这个函数返回一个整数：返回这个数组所有数的平方值中有多少种不同的取值。

6.1.2 一个环有10个节点，编号0-9。从0点出发，走N步又能回到0点，共有多少种走法？

6.1.3 一个乱序数组，求第K大的数。排序方式使用字典序。

6.1.4 一棵二叉树，求最大通路长度。（即最大左右子树高度之和）

6.1.5 进程和线程的区别，使用线程真的能节省时间？

6.1.6 go协程的调度方式，使用协程真的能节省时间？

6.1.7 水平触发边沿触发的区别？在边沿触发下，一个socket有500的数据，已读取200然后不再处理，是不是剩下的300就永远无法读取？

6.1.8 有函数如下，输入1，返回什么？

6.1.9 设计http协议，A端发送 AAAA，至少让B端知道AAAA已发送完成。

6.2.0 流量总入口为api\_gateway，api\_gateway挂了会导致全部挂挂，用什么机制增大可用性？

6.2.1 mysql为什么要用b+树，不用平衡二叉树做索引结构？

6.2.2 创建数据库索引应该怎么考虑？

6.2.3 使用int 做primary key和使用string 有什么优劣？

6.2.4 数据库分表的方法？

6.2.5 表结构，订单纪录如下，写一个语句，求卖的最好的 top 10 product\_id。

6.2.6 微服务，A服务请求B服务B1接口，B1接口又请求A服务A2接口。会不会有问题？

6.2.7 不使用高级工具，只使用Linux自带的工具，你会如何debug?

6.2.8 如何预估一个mysql语句的性能？

6.2.9 go函数中，返回值未命名，发生了panic，但是在函数内recover了。函数返回什么值？

6.3.0 socket中，在tcp协议层面，数据分为10个报文发放。1-7次很顺利，第8次丢失。这次通信一定失败吗？如果第8次数据会重发，那在接收端是不是：先读取到1-7次的数据，然后读取到8-10次的数据?还是9-10次的数据会先到达？

6.3.1 free -h，buffers 和cached有什么不同

6.3.2 后台进程有什么特点，如果要你设计一个进程是后台进程，你会考虑什么

6.3.3 僵尸进程是什么，如果产生一个僵尸进程，如何查找僵尸进程

6.3.4 孤儿进程是什么

6.3.5 一个进程有20个线程，在某个线程中调用fork，新的进程会有20个线程吗？

6.3.6 tcp/ip 流量控制和拥塞控制

6.3.7 301/302有什么区别？应用上有什么异同。

6.3.8 50X相关错误码的内涵是什么？

6.3.9 close wait和time wait是什么？如何排查？有什么意义？

6.4.0 http req和resp的中数据有哪些

6.4.1 什么是连接的半打开，半关闭状态

6.4.2 假如一个业务依赖单点redis，此redis故障将导致业务不可用，如何改进

6.4.3 redis sharding有哪些做法

6.4.4 当大量数据要求用redis保存，单机单点难以满足需要，设计（换寻找）一个负载均衡的方案

6.4.5 当redis 采用hash做sharding，现在有8个节点，负载方案是 pos = hash(key) % 8，然后保存在pos节点上。这样做有什么好处坏处？当8个节点要扩充到10个节点，应该怎么办？有什么更方便扩充的方案吗？（一致性hash, presharding）

6.4.6 如何保证redis和数据库数据的一致性。比如用户名既保存在数据库，又保存在redis做缓存。有如下操作 update\_db(username); update\_redis(username)。但是执行update\_db后故障，update\_redis没有执行。有什么简单办法解决这个问题。