1.线程局部变量

局限于线程内部的变量,线程自身所有.不共享,threadlocal类来支持局部变量,实现线程安全,在管理环境下使用线程局部变量,工作线程的生命周期比任何变量的生命周期都要长,工作完成后,没有释放,很容易造成内存泄漏.

2.wait-notify解决生产者消费者的问题

wait ,notify,notifyall 多线程中的保留关键字,实现线程间的通信,

两个线程:生产者和消费者,生产者通知消费者消耗数据,消费者通知生产者补充数据.

wait暂停运行,生产者的缓存满时,消费者在缓存区为空时,暂停运行.notify和notifyall通知那些等待中的线程重新运行.notify只通知一个线程运行,不知道那个线程收到通知.notifyall会通知所有等待中的线程.

wait是Object的类,thread无法使用,在生产者和消费者的问题上,wait使用的对象应该是缓冲区队列.

使用synchronized是希望上锁的对象,被多个线程共享的对象,再生产者和消费者中synchronized的对象是缓冲区队列.

在while语句中调用wait,不在if中调用wait.(if语句存在问题,即使 条件没有满足,线程也会被错误的唤醒).

synchronzied(shareObject){

while (condition ){

shareObject.wait();

}

}

在while中使用wait的目的,是在线程在被唤醒前后,都持续检查条件是否被满足.如果条件并未改变,wait被调用之前,notify的通知来了,这样不能保证线程被唤醒,还可能导致死锁的问题.

linkedlist实现了queue接口,linkedList作为缓冲区队列



1).使用wait,notify实现线程间通信,

2).永远在synchronize中使用wait,notify,否则会有异常

IllegalMonitorStateException 等待对象的监视器,但是没有这个监视器

3.线程安全的单例模式

枚举作为单例是最简单的实现线程安全单例模式的方式

正常写个懒汉式和饿汉式，线程不安全，在获取实例的方法上添加synchronize关键字，保证线程安全，这样会在一定程度上，使程序的并发度降低，

使用双重检查锁，在保证线程安全的同时，做到好的并发度。

线程不安全只在初始化对象时，把同步的粒度降低，只在初始化对象时同步。多线程执行会出现对象没有被完整初始化的问题。

使用内部类来延迟加载对象，jls java虚拟机的机制保证线程安全



4.sleep和wait的区别

都是用来暂停当前的线程,sleep是短暂停顿,不会释放锁,,wait是条件等待,会释放锁,

5.不可变对象

不可变对象指的是,一旦创建,状态就不能改变.

涉及到不可变类,一个类为不可变类,这个类的对象为不可变对象.

获得类的实例引用,类的内容不可以改变.

包装类是不可变类的代表,

创建一个不可变类,用final修饰,不提供set方法

6.创建一个包含可变对象的不可变对象

可以创建,不要共享可变对象的引用.常见例子,对象中包含一个日期对象的引用.

7.表示价格的数据类型

不关心内存和性能, 使用BigDecimal,否则使用与精度型的都变了类型

8.byte转为String

使用Byte[]的构造器进行转换,但要注意编码格式

9.bytes转为long类型

1).通过移位运算转换

2).ByteBUffer

3).datainputstream

10.int强转byte

int是32位,byte是8位,强转后,int的高24 位会被丢弃

11.继承

12.那个类包含clone方法,是cloneable接口,还是Object

Cloneable是一个标识性接口,不包含任何接口,clone是在Object类中定义的,是一个本地方法.

13.++是线程安全的吗

++不是线程安全的操作.涉及到多个指令,如读取变量值,增加,存储会内存,会出现多个线程交叉.

14.a=a+b和a+=b的区别

+= 包含隐式的强制类型转换,然而a+b会出现错误.

15.3\*0.1不等于0.3 有的浮点数不能完整精确的表示出来

16. integer占用的内存多, integer是一个对象,需要存储对象的元数据

17.String是不可变的

设计者认为string的使用非常频繁,设为不可变可以允许多个客户端之间共享相同的字符串.

18.switch在java7以后可以使用string

19.构造器链,从一个构造器调用另一个构造器链,只在存在了类的构造器的时候才会出现.

20.jvm的int长度是固定的都是32位

21.Serial和Parallel GC的不同之处

两者在执行时,都会引起Stop-the-world停止所有线程.serial默认的是复制回收器算法,执行一个线程;而Parallel会使用多个线程.

22.java的引用

强引用:new创建的对象,强引用存在,该对象不会被回收,即使抛出内存溢出也不回收

软引用:可有可无,内存不够,就会回收,实现内存敏感的高速缓存.

弱引用:更短的生命周期 ,只要进行垃圾回收,就会被回收

虚引用:形同虚设,任何时候都可能被回收

23.integer的值缓存范围(-128 -127)

24.String,StringBuilder,StringBuffer

builder>buffer>

string不可变对象,操作时属于创建新对象回收旧对象

StringBuilder是线程不安全的,

25.反射就是把java类中的各个成分映射成一个个的java对象

获取class文件的三种方式a. .class b. getClass c. forName

26.注解:最开始是生成文档,跟踪代码依赖性实现地台配置文件功能

在编译期进行格式检查

@Retention(RetentionPolicy.RUNTIME)

public @interface MyTarget {

}

上面的注解是自定义注解必须的,为注解的元注解,括号中含有枚举类型的值,

java的元注解一共4个 Document,target,retention,inherited

27.常用的jdk包:

lang,util,io,net

28.序列化和反序列化

序列化:把对象转换成为序列字节的过程.

反序列化:把字节序列恢复为对象的过程.

需要序列化的情况:内存中的对象,保存到文件中或者数据库中时

想用套接字在网络上传送对象.

套接字就是ip:port作为tcp连接的端口，

通过RMI传输对象

过程:oos.writeObject()

ois.readObject()

transient 修饰的属性不会被序列化,静态static的属性不会序列化

29.wait ,notify放在object,因为同步的锁可以是任意对象

30.java8的新特性:

1).lambda表达式闭报和函数表达式,允许将函数作为参数传递给方法 ,或者把代码本身作为数据处理

java的匿名内部类,

最简单的lamdba表达式,由逗号隔开,的参数列表->和语句块组成

Arrays.asList( "a", "b", "d" ).forEach( e -> System.out.println( e ) );

2).默认方法和静态方法

默认方法向接口中添加方法,不强制实现了接口的类,同时实现新加的方法,

3).方法引用

4).重复注解允许在同一个方法多次使用同一个注解

5).更好的类型判断

6).拓宽注解的应用场景

7).注解可以使用在任何元素上,局部变量,接口类型,超类,接口实现类,

8).java编译器的新特性,参数名称

9).java官方库的新特性,

optional避免显示的null检查

10).stream将生成环境的函数式编程,引入到java库中,

11).vector 翻倍底层数组

31.arraylist50%

hashtable同步0.75

hashmap基于hash表 链表散列的数据结构 数组和链表的结合体 初始容量10 负载因子0.75

linkedlist

32.红黑树本质上就是二叉查找树,在其之上添加了一个标记,同时具有一定规则,这些规则保证了最坏时间复杂度为O(logn)

33. ConcurrentHashMap 相对于hashtable在线程安全的基础上,提供了更好的写并发能力,降低了对读一致性的要求,jdk版本不同,实现也不同

34. 多线程情况下HashMap死循环的问题

什么会出现:首先是链表结构,很容易形成闭合的回路,如果只对hashmap进行get操作,就会产生死循环,单线程不会产生闭合回路.

在多线程的情况下,扩容是,两个线程同时进行rehash就会产生闭合回路.

35. Hash出现Hash DOS攻击的问题

大部分网站接口未做hash碰撞攻击的防御,

大部分使用json数据,处理json数据时,首先要讲json串转为json对象,对象会默认存储在hashtable中,hashtable很容易被碰撞攻击,

只要将攻击数据放在json中,在转换对象时,cpu会like飙升至100%,16核的cpu,16个请求就能达到dos攻击的目的.

防御:在jsonDocode之前将非法用户拒绝,根据数据大小和参数,做白名单验证,重写hashtable

36. ConcurrentHashMap

大量利用了volatile,final,cas等lock-free技术来减少锁竞争的影响,

采用了分段锁的设计,只有在一个分段之间才存在竞争关系,不同的分段锁之间没有锁竞争,分段锁大大提高了高并发环境下的并发能力.并发度,程序运行能够更新cchm且不产生锁竞争的最大线程数,

37.并行,是同时做多件事情;并发:同时发生了多件事情,需要线程去执行,并发是可以同时执行多个任务,但这些任务不是同时执行.

38.创建线程的方式

继承thread类,实现 Runnable接口,实现callable接口

callable接口能返回执行结果,其中的call方法允许抛出异常,支持返回结果,需要调用FutureTask.get方法实现,会阻塞主线程直到获取将来结果,不调用此方法时,主线程不会阻塞.

Runnable接口只能在内部消化,不能抛出

39.进程间的5种通信方式

管道,消息队列,信号量,共享存储,socket,Streams

40.并发编程

CountDownLatch:是concurrent包下的,实现类似于计数器的功能,

任务a,要等待其他4个任务执行完毕后才能执行,

只有一个有参构造器,

方法:await();等待count为0后执行

await(time,timeUnit);,一段时间后,没有变也会继续执行

countDown(); 计数器减一

CyclicBarrier:字面意思回环栅栏,让一组线程等待到某个状态然后全部执行;回环是因为所有线程被释放后,cyclicBarrier可以被重用,

此时的状态为barrier,调用await方法后,线程就处于barrier状态.

两个有参构造方法,

参数1:可以控制多少个线程,等待至barrier状态,

参数2:当这些线程都达到barrier状态时执行的内容.

方法1:挂起当前线程,直至所有线程都达到barrier状态,再执行后续任 务 await()

方法2:让这些线程等待至一定时间,如果还有线程还没有达到barrier状态,就让达到状态的线程执行后续任务.await(a,b)

Semaphore:字面意思信号量可以控制同时访问的线程个数,通过acquire获得一个许可,如果没有就等待,而release择释放一个许可.

2个构造器:

参数1:表示可以允许多少线程访问,

参数2:表示是否公平,时间长的优先获得许可

方法:1.acquire()获取一个许可

2.acquire(a)获取a个许可

3.release()释放一个许可

4.release(a)释放a个许可

若四个方法都被阻塞,tryAcquire尝试获得许可.

总结 countdownlatch,cyclicBarrier,semaphore

前两者实现线程间的等待,countdownlatch值等待其他线程执行完,他才执行,cyclicBarrier是一组线程达到某个状态,这组线程在执行某个人物,countdownlatch是不能够重用的,cyclicBarrier是能够重用的.

semaphore和锁类似,一般用于控制某组资源的访问权限

41. Exchanger的工作原理

Exchanger交换者,是一个线程间协作的工作类,用于进行线程间的数据的交换,他提供一个同步点,在这个同步两个线程可以交换彼此的数据.一个线程先执行exchange方法,等待第二个线程执行exchange,两个线程达到了同步点,就会交换数据.该工具类类的线程对象时成对的.两个方法:一个是等待第二个线程的exchange方法,二是,等待一段时间后便停止等待.

42.ThreadLocal

线程本地变量,线程本地存储

问题:多线程连接数据库

一样的代码,会出现有的打开连接,有的关闭连接,

其实其他线程不会关心是否其他线程对connect进行修改.

每个线程进行判断连接数据库,关闭数据库,这样导致服务器压力巨大,严重影响程序执行性能,

出现ThreadLocal,对该变量会创建一个副本,在线程内部都可以使用,线程间互不影响,这样就不存在线程安全问题,也不会严重影响程序执行性能.

资源消耗大,内存占用多.

方法:get()获取线程中保存的变量副本,

set()设置当前线程中变量的副本,

remove()移除当前线程中变量的副本,

initAlValue(),会进行重写,延迟加载

应用场景:数据库连接,session管理

数据库连接的ThreadLocal



session管理的ThreadLocal:

public class SessionManager {

private static ThreadLocal threadSession =new ThreadLocal();

public static Session getSession(){

Session s=(Session) threadSession.get();

try {

if(s==null){

s=getSessionFactory().openSession();

threadSession.set(s);

}

} catch (Exception e) {

// TODO: handle exception

}

return s;

}

}

ThreadLocal不是用来解决对象共享访问问题的,主要是提供了保持对象的方法,和避免参数传递的方便额的对象访问方式。

1）.每个线程都有一个自己的ThreadLocalMap对象,可以将线程自己的对象保持到其中,各管各的,线程可以正确的访问到自己的对象.

2.将一个公用的ThreadLocal静态实例做为key,将不同对象的引用保持到不同线程的ThreadLocalMap中,然后在线程执行的各处通过这个静态ThreadLocal实例的get方法获取自己线程保存的对象,避免了将这个对象,作为参数传递的麻烦.

43.线程池的实现原理

1.5引入Executor线程池框架,通过他把任务的提交和执行进行解耦,只需要定义好任务,然后提交给线程池,而不用关心任务怎么样执行,被那个线程执行,何时执行.

工厂类:Executors类,

初始化四种类型的线程池

1).newFixedThreadPool()

初始化一个指定线程容量的线程池,其中corePoolSize==maxPoolSize,使用linkedBlockingQuene作为阻塞队列;

即使线程池没有可执行的任务,也不会释放线程.

2).newCachedThreadPool()

初始化一个可以缓存线程的线程池,默认缓存60s,线程池的线程数可以达到integer的最大值,内部使用synchronousqueue作为阻塞队列

没有线程执行时,空闲时间超过一定时间时,会自动释放线程资源,提交新任务时,如果没有空闲线程,则创建新线程执行任务,会导致一定的系统开销.注意控制现成的并发数,防止因创建大量线程若降低性能

3).newSingelThreadExecutor()

初始化只有一个线程的线程池,内部使用linkedBlockingQuene作为阻塞队列

如果该线程异常结束,会重新创建一个新的线程继续执行任务,唯一的线程可以保证锁提交任务的顺序执行.

4).newScheduledThreadPool()

初始化一个可以在指定时间内周期性的执行所提交的任务,在实际的业务应用中,使用该线程池定期的同步数据.

线程池内部是基于ThreadPoolExecutor实现的

类构造器

(corePoolsize,maxPoolSize,keepliveTime,timeUnit,workQuene,threadFactory, handle)

corepoolsize:线程核心数,

maxPoolsize:最大线程数

keepAliveTime:线程存活时间(造在大于核心小于最大时生效)

timeUnit:存活时间的时间单位

workQuene:阻塞对列(用来保存被执行的任务)

4中:arrayBlockingQuene

linkedBlockingQuene

synchronousQuene不存储元素,保持一出一进

priorityBlockingQuene:优先级的无界阻塞队列

threadFactory:线程工厂,用于创建线程.

handler:表示当拒绝处理任务的策略

当线程池的饱和策略,当阻塞队列满了,且没有空闲的工作线程,继续提交任务,丢弃任务抛出异常

丢弃任务不抛出异常

丢弃最前面任务,重新尝试执行任务

调用线程执行该任务.

线程池的5种状态:

running(接受任务,并处理阻塞任务),shutdown(不会接受任务,处理阻塞任务),stop(不处理,不接受,停止),tidying(线程池对线程进行优化),terminated(停止).

向线程池提交任务:

Excutor.execute(runnable)

ExecutorService.submit(Callable)

线程池关闭

shutdown:不立即终止,等待缓存队列的任务执行完毕,不会接受新任务,

shutdownnow:立即终止,打断正在执行的任务,清空缓存队列,返回尚未执行的任务.

线程池容量的动态调整:

setCorePoolSize()

setMaxinumPoolSize()

总结:

线程池中的核心线程数,当提交一个任务时,线程池会创建一个新线程执行任务,直到当前线程数等于corePoolSize,如果继续执行,继续提交的任务会保存到阻塞对列,等待被执行,当阻塞对列满了,创建新的线程执行当前任务,直到线程数达到maxPoolSize,再有任务来,只能执行reject处理该任务;

44.线程的生命周期及状态转移

创建(start)--就绪

阻塞--运行--终止

new只是创建个对象分配内存

调用start进入就绪状态,位于可运行池中,具备了运行的条件

run方法,进入运行状态,

进入阻塞状态,获取同步锁,调用阻塞式的io,wait,sleep

45.重入锁,防止死锁

重进入,,任意线程在获取到锁后,再次获取该锁,不会被该锁阻塞,.关联一个线程持有者+计数器,重入意味着锁操作的颗粒度为线程.

需要解决的两个问题:

线程再次获取锁:锁需要识别获取锁的线程是否为当前占据所得线程,如果是,则再次获取成功

锁的最终释放:线程重复n次获取锁,随后在第n次释放锁后,其他线程能够获取该锁,获取+1,退出-1;0时释放该锁.

重入锁实现重入性,每个锁关联一个线程持有者和计数器.+1,-1

重入锁防止死锁,非公平锁的原因.

正常公平锁,一般是等待时间长的线程先获得锁,

非公平锁,有一个计数器,数值大的有限获得锁.

46.产生死锁的条件

进程中的每个线程都在等待其他进程执行,才能继续往下执行.若无外力,此时就处于死锁.

死锁的原因:因竞争资源产生死锁,系统中多个进程共享的资源数目不能满足全部进程的需求.

资源包括可剥夺资源,不可剥夺资源,临时资源;

可剥夺资源,该资源被某进程获取后,仍可以被其他进程或系统剥夺.资源分配后,不能强制收回,只能在进程使用完后自动释放.

竞争不可剥夺资源,系统中的不可剥夺资源不足以满足所有进程运行的需求,争夺资源陷入死锁.

进程推进顺序不当时,发生死锁.

产生死锁的四个必要条件:

1).互斥条件,进程对所分配到的资源,不允许其他进程访问,只能等待占用该资源的进程释放资源

2).请求和保持条件:进程获得一定资源后,又需要其他资源,但其他资源被别的进程占有,此事阻塞,但对自己的资源部释放

3).不可剥夺条件

4).环路等待条件:进程发生死锁后,必然存在一个进程资源之间的环形链.

处理死锁:

1).预防,通过限制条件,破坏产生死锁的必要条件,

2).避免,在资源分配过程中,避免进入不安全的状态,从而避免发生死锁

3).检测死锁,允许死锁的发生,但检测到后,采取措施,将死锁清除掉

4).解除死锁,与检测死锁配合使用

47. 如何检查死锁（通过jConsole检查死锁）

发生死锁后,jdk自带两个工具jstack,jconsole,可以用来检测分析死锁的发生原因.

jstack工具用来生成虚拟机当前时刻的线程快照,就是当前虚拟机每一条线程正在执行的方法堆栈的集合,生成的快照可以定位线程死锁,死循环.

jConsole是一个可视化监视管理工具,用于连接正在执行的jvm进程,以监控java应用程序性能,跟踪代码.

jstack的使用:tasklist|findstr"java"

jstack 端口 >目标文件位置

48. volatile 实现原理（禁止指令重排、刷新内存）

活跃的,不稳定的

cpu-主存-高速缓存

缓存一致性问题,一个线程对应一个高速缓存

解决缓存不一致问题,在总线加lock#锁,通过缓存一致性协议.

总线加锁会导致cpu无法访问,效率低下,

缓存一致性协议,保持每个缓存中使用的共享变量的副本是一致的,操作数据是,发现时共享变量,会通知其他cpu,设置该变量的缓存无效.其他cpu读取时,发现无效,就会从内存中重新读取.

并发编程的三个概念:

原子性,可见性(一个线程修改了变量,其他线程能看到),有序性(会发生指令重拍)

指令重排序:为了提高程序运行效率,可能对输入的代码进行优化,他不保证程序中的各个语句的执行顺序与代码顺序一致,但能保证,最终执行结果是一样的.重排序时会考虑指令之间的数据依赖性.

指令重排序不会影响单个线程的运行,但会影响线程并发执行的正确性.

java内存模型也会存在,缓存一致性和重排序的问题.

java内存模型,,,,,,,主存,,,,工作内存

只有简单的读取,赋值(只能是将数字赋值给变量)才是原子操作。

最新jdk中，保证对64位数据的读取和赋值也是原子性操作

要是想实现更大范围的原子性操作，要通过锁来实现，

java内存模型通过volatile来保证可见性，被volatile修饰的变量，一发生改变，就会更新到主存中。其他线程读取时直接从主存中读取新的值。锁也能保持可见性。

volatile能保持有序性。

一个共享变量被volatile修饰后，，不同线程的可见性，禁止进行指令重排序。volatile不能保证操作的原子性。

volatile修饰的会产生一个lock前缀指令。

49. synchronized 实现原理

synchronized是一个重量级锁，保证方法或者代码块在运行时，同一时刻只有一个方法可以进入到临界区没，同时保存共享变量的内存可见性。

java的每个对象做可以作为锁，是synchronized实现同步的基础.

普通同步方法,锁是当前实例对象;静态同步方法,锁是当前累的class对象;同步方法块,锁是括号里的对象.

50.锁类型

可重入锁:执行对象中的所有同步方法,不用再次获得锁

可中断锁:在等待获取锁过程中可中断

公平锁:按等待时间获取锁,时间长的优先获得锁

读写锁:读可以多线程读,写必须同步写.

51.synchronized与lock的区别

syn关键字 在虚拟机层面;lock是一个类

执行完或者发生异常,syn释放锁;lock必须在finally中释放封锁

无法获取锁,syn会一直等待,lock不用等待.

syn无法判断锁的状态,lock可以判断锁的状态

syn的锁类型,可重入不可中断,非公平, lock可重入可判断,公平(非公平)

syn 少量同步,lock大量同步

52.AQS同步队列

全拼:AbstractQueneSynchronize 抽象的同步队列.

基于volatile in state这样的volatile变量,配合Unsafe工具对其原子性的操作,来实现对当前锁状态的修改.同步器内部依赖一个FIFO的双向队列来完成资源获取线程的排队工作.

同步器的主要使用方式是通过继承,子类通过继承同步器并实现它的同步方法,来管理同步状态,对同步状态的修改或者访问,主要通过同步器的三个方法:

getState()获取当前的同步状态,

setState(新的状态)设置当前同步状态,

compareAndSetState(a,b)通过cas设置当前状态,giant方法能够保证状态设置的原子性.

同步器支持独占式的获取同步状态,也可以支持共享式的获取同步状态,这样可以实现不同类型的同步组件.

同步器也是实现锁的关键,在锁的实现中聚合同步器,利用同步器实现锁的语义.

AQS同步队列:

同步器AQS内部的实现是依赖同步队列(一个FIFO的同步队列,其实就是数据结构双向链表)来完成同步状态的管理

当前线程获取同步状态失败时,同步器AQS会将当前线程和等待状态等信息构造成一个节点加入到同步队列,同时会阻塞当前线程.

当同步状态释放时,会把首节点中的线程唤醒,使首节点的线程再次尝试获取同步状态,

AQS是独占锁和共享锁的实现的父类.

独占锁中含有公平锁与非公平锁.

53. CAS无锁的概念、乐观锁和悲观锁

悲观锁:传统的关系型数据库,行锁,表锁读锁,写锁, synchronzied关键字的实现也是悲观锁.

乐观锁:使用的是版本号机制,write\_condition机制,concurrent是乐观锁;

cas是乐观锁的实现方式.

悲观锁的问题:加锁,释放锁,影响性能问题.会导致其他需要此锁的线程挂起，优先级高的等待优先级低的锁，导致优先级倒置。

乐观锁的具体实现：冲突检测，数据更新。其实就是cas

乐观锁会一直尝试，会给cpu带来很大的执行开销。原子一致性问题。只能保证一个共享变量的原子操作。

54.常见的原子操作类

util.concurrent.atomic一共提供了13个类,属于4中类型中的原子更新方式:原子更新基本数据类型,原子更新数组,原子更新引用,原子更新属性.

55.ABA问题

A线程先获取数据,B线程改变数据又改回来,A线城读取得到的数据没有改变.

java避免ABA问题,AtomicStampedReference和AtomicMarkedReference来解决ABA问题.

AtmoicStampReference可以原子更新两个值:引用和版本号,通过版本号来区别节点的循环使用.AtomicMarkedReference可以更新一个布尔类型的标记位,和引用类型.

56.乐观锁的业务场景以及实现方式

----------------------

57. Java 8并法包下常见的并发类

java.util.concurrent.atomic

58. 偏向锁、轻量级锁、重量级锁、自旋锁的概念

偏向锁:Biased Locking java6引入的多线程优化,他会偏向于第一个访问锁的线程.同步锁只有一个线程访问,不存在多线程,不需要触发同步,就会给线程加一个偏向锁.,如果遇到其他线程访问,持有偏向锁的线程会被挂起,jvm消除他身上的偏向锁,将锁恢复到标准的轻量级.

轻量级锁:相对于偏向锁而来的,抢夺锁时,会升级为轻量级锁.

重量级锁:synchronized会导致争用不到锁的线程进入阻塞状态,是java中的重量级的同步操纵,称为重量级锁.

轻量锁和偏向锁都属于乐观锁.

自旋锁:持有锁的线程很快就会释放锁,等待锁的线程就不会做内核状态和用户之间的切换进入阻塞挂起状态,他们只需自旋,就会在释放锁后获得锁,避免了用户线程和内核的切换到消耗.

59.jvm的内存区域划分.

java虚拟机栈,方法区,堆,本地方法栈,程序计数器

60. 内存溢出OOM和堆栈溢出SOE的示例及原因、如何排查与解决

oom :outofMemory内存溢出,

soe:stackoverflowerror堆栈溢出,

针对于内存区

java虚拟机栈:线程请求的栈深度大于虚拟机所允许的最大深度,抛出 堆栈溢出异常;虚拟机可以动态扩展,扩展无法申请到足够内存,抛 出内存溢出异常

本地方法栈:

程序计数器:

堆:

方法区:

=============================================

61. 如何判断对象是否可以回收或存活

使用了根搜索算法,他把内存中的每一个对象都看作一个节点,并将这些对象作为根节点,如果这个对象中有另一个对象的引用,那么就认为此对象有一个指向第二个对象的边,jvm会起一个线程开始遍历,如果有一些对象不可达,那么这些对象已经没用了,需要被回收.

在回收时,会执行一个finalize方法,给对象一个重生的方法,如果与别的有关联,就不会被回收.

62.Gc回收算法含义

判断是否垃圾

引用计数法,根搜索法

复制:适用于新生代,垃圾对象多余存活对象,高效:解决了内存碎片问题,但内存折半

标记整理:标记出需要回收的对象,将存活对象压缩到内存的另一端,清理边界的所有对象.

标记清除:标记出需要回收的对象,然后进行回收:容易产生大量内存碎片

分代回收算法:根据不同区域采用不同回收算法

63.常见的垃圾回收器

Serial收集器是最基本历史最悠久的垃圾回收器,,他是一个单线程的收集器,进行垃圾回收时,必须暂停其他所有的工作线程,直到他收集技术,高效简单.

Parallel scavenger是一个新生代收集器,是使用复制算法的收集器,并行的多线程的收集器,关注的是吞吐量.

CMS收集器基于标记清除算法,初始标记-并发标记-重新标记-并发清除.最短停顿时间

G1收集器:基于标记-整理算法,不会产生内存碎片,能够精确的控制停顿.

64.类加载的过程:加载\验证\准备\解析\初始化

加载: 通过类的全限定名来获取此类的二进制字节流.

通过字节流锁代表的静态存储结构转化为方法区的运行时结构.

在java堆中生成一个class对象,最为方法取得数据访问入口.

验证:确保字节流中包含的信息符合要求.并且不会危害虚拟机的自身 安全,主要验证文件格式验证运输局验证,字节码验证,符号应用验证

准备:正式为类变量分配内存,设置累的初始化值,内存在方法区中进 行分配

解析:将常量池中放入符号引用替换为直接引用

初始化:真正执行定义的java代码.

64.常见的设计模式

单例模式:此类只有一个实例,提供一个访问他的全局访问点

懒汉式:延迟加载适合单线程操作,

缺点:线程不安全,,多线程中很容易穿线不同步的情况.

双重线程检查模式:优点,延迟加载,线程安全

写法复杂,不简洁

内部类的实现:延迟加载线程安全,减少了内存消耗,推荐使用内部类方式.

工厂模式:

为创建对象提供过度接口,以便将创建对象的具体过程屏蔽起来,达到提高灵活性的目的.简单工厂模式,工厂方法模式,抽象工厂模式,

简单工厂模式:是由一个具体的雷区创建其他类的实例,父类是相同的,具体的.

工厂方法模式:抽象的父类定义公共接口,,,子类负责生成具体的对象,目的是将类的实例化延长到子类中区

抽象工厂模式:提供一个创建一系列相关或互相依赖对象的接口,无需指定他们具体的类.

建造模式:

是一种对象构建的设计模式,可以将复杂对象的建造过程抽象出来,抽象过程的实现方法构造出不同表现的对象.

观察者模式,适配器模式,装饰者模式,代理模式

65. 设计模式的的六大原则及其含义

开闭原则:对扩展开放,对修改关闭

里氏代换原则:基类可以被子类替代

依赖倒转原则:要依赖于抽象,不要依赖于具体

接口隔离原则:使用多个隔离的接口,比使用单个接口好

迪米特原则:一个软件实体尽可能少的与其他实体发生相互作用

合成复用原则:尽量使用合成聚合,而不是使用集成

66.spring的设计模式

单例(bean) 前端控制器 (spring的DispatcherServlet)工厂模式(beanfactory创建对象)

适配器模式讲一个类的接口转换成为客户希望的另一个接口spring 对aop的处理

包装模式,动态的给对象添加额外的职责wrapper,decorator

代理jdk代理和cglib代理

观察者:listenner的实现

策略:实例化对象

模板方法:jdbctemplate

67.mybatis的设计模式

装饰者模式:cache的实现

建造者模式:basebuilder,xmlapperBuilder

工厂方法:sqlsessionFactory

适配者模式:Log,logFactory

模板方法:BaseExecutor.SimpleExecutor

动态代理:

责任链模式:interceptor,InterceptorChain

68.

1.7、数据结构

树（二叉查找树、平衡二叉树、红黑树、B树、B+树）

深度有限算法、广度优先算法

克鲁斯卡尔算法、普林母算法、迪克拉斯算法

什么是一致性Hash及其原理、Hash环问题

常见的排序算法和查找算法：快排、折半查找、堆排序等

1.8、网络/IO基础

BIO、NIO、AIO的概念

什么是长连接和短连接

Http1.0和2.0相比有什么区别，可参考《Http 2.0》

Https的基本概念

三次握手和四次挥手、为什么挥手需要四次

从游览器中输入URL到页面加载的发生了什么

69.MySQL索引使用的注意事项

索引的优点:大大加快数据的查询速度

使用分组和排序进行数据查询,可以显著减少查询是分组和排序的时间

创建唯一索引,能够保证每行数据的唯一性,

在实现数据的参考完整方面,可以加速表和表之间的连接

缺点:创建索引和维护索引需要时间

索引需要占据磁盘

对数据库进行增删改查时,索引需要动态维护,降低了维

护速度

原则:频繁更新的列不应该设置索引

数据量小不要

重复数据多的字段不要(男女)

优先在where和order by 涉及的列建立索引

优化查询:

在where不用使用=函数,索引不会生效

不要使用<>,!=,not in索引不会生效

避免字段对null进行判断,索引不会生效

使用like时,%%会使索引失效,只是用后半%不会

避免使用or,可以用union替代

使用exist代替in

数据类型隐形转换,索引不会生效

联合索引必须按顺序

尽量避免使用游标

70. DDL、DML、DCL分别指什么

DDL:数据库定义语言create alert drop ,一般用于定义改变表的结构,数据类型,表之间的连接以及初始化,大多在建表时使用

DML:数据库操作语言select update insert delete 对数据库的数据 进行操作的语言

DCL:数据库控制语言设置或更改数据库用户和角色权限的语句

71. explain命令

explain命令是查看优化器如何决定执行查询的主要方法,

具体使用:explain +sql 语句

72. left join，right join，inner join

73. 数据库事物ACID

原子性(不可分割),一致性(事务开始前后,数据哭的完整性没有被破坏),隔离性(不同的事务操作相同的数据,每个事务都有各自的完整数据空间),持久性(事务对数据库的操作持久的保存在数据库中).

74. 事物的隔离级别（读未提交、读以提交、可重复读、可序列化读）

读未提交:(脏读) 可以读取到另一个事务未提交的数据 读提交可以解决脏读

读提交:读取需要另一个事务提交后,不可重复读

重复读: 对应的是修改update,而不是insert操作

造成幻读

序列化:解决幻读事务串行化顺序执行

oracle的默认隔离级别是重复读

mysql的默认隔离级别是不可重复读

75. 数据库的几大范式

一共8种,常用的三种

第一范式:强调的是列的原子性,列不能在分为其他几列

第二范式:一个表必须有一个主键,没有包含在主键中的列必须完全依赖主键,而不能依赖主键的一部分.（联合主键）

第三范式:非主键列必须直接依赖主键,不能存在传递依赖。

76. 说说分库与分表设计

对于大型的互联网应用，数据库的行数数以亿计，面临着极高的并发访问。采用master-slave复制模式的mysql架构，只能对数据读的读进行扩展，写操作还是集中在master上，单个master挂载的slave不可能无限制多，slave收到master能力和负载的限制，分表就是减少单表的记录条数，以便减少数据查询需要的时间，提高数据的吞吐量。选择适当的分表策略。

分表能够解决单表数据量过大，带来的产讯效率下降的问题。无法为数据库的并发能力，带来的变化，

不管如何扩展slave，无效，对数据库进行拆分，关键字取模的方式，对数据访问进行路由

分库分表

中间变量id%（库数\*表数）

库=中间变量/表数量 取整

表=中间变量%每个库的表数量

77.sql优化

**http://www.cnblogs.com/yunfeifei/p/3850440.html**

78. mysql遇到死锁问题,如何排查

多个用户并发的存取数据时,在数据库中就会存在多个事务同时存取统一数据的情况,若对并发操作不加控制,就会读取和存储不正确的数据,,破坏数据库的一致性.两组事务需要一组有冲突的锁,而不能将事务继续下去,就会产生死锁.

在数据库中有两种基本的锁类型,排它锁x锁,共享锁s锁,当数据对象加上排它锁时,其他事务不能对他进行查找和修改,添加了共享锁的数据对象可以被其他事务读取,但不能被修改,

a访问a表锁住a表,b用户访问需要a释放a表,才能继续,这样相互的,就会产生死锁. 产生是由于程序的bug,需要按照相同的顺序来锁定资源.

a查询记录,修改记录,用户b修改该记录,.a的事务的锁由查询的共享锁上升为独占锁,b的独占锁由于a的共享锁必须等a释放共享锁,a由于b的独占锁而无法上升为独占锁不能释放共享锁,于是就出现了死锁,产生情况:快速点击同一个按钮.解决按钮防多安的设置

采用乐观锁(可能产生脏数据)

采用悲观锁(开率高并发的操作)

执行不满足条件的update语句,执行全局扫描行级锁上升为表级锁,多次执行,产生阻塞或者死锁.

79. 存储引擎的 InnoDB与MyISAM区别，优缺点，使用场景

innoDB事务安全,支持夯机所,不支持全文索引,

MyISAM保存成文件的形式

应用场景:select查询使用myISAM,

innoDb用于事务,具有ACID,需要insert和update,提高多用户并发操作的性能.

80. 索引类别及原理

B+树,是一个平衡的多叉树,从根节点到叶子节点的高度差不超过1,同层级事务节点有指针相互连接,

常规检索,从根节点到叶子节点的搜索效率基本相当,不会出现大幅波动,而且基于索引的顺序扫描时,利用双向指针快速左右移动,效率非常高.

hash索引:就是采用一定的hash算法,把键值换算成新的hash值,,不需要逐级查找只需一次hash算法,就能定位

明显区别:

等值查询,hash有优势但要求键值是唯一的.

范围查询,需要B+树查询.

hash不能完成排序,和模糊查询(ccc%)

hash不支持多列联合索引的最左匹配规则

有大量重复键值.就会产生hash碰撞问题.

81.自适应hash索引

hash是一种非常快的查找方法,在一般情况下,时间复杂度为o(1),即一般需要一次查找就能找到数据.

B+树的查找次数取决于B+树的高度,生产环境中,高度一般为3-4层,需要3-4次的查询.

InnoDb存储引擎会监控对表上各索引页的查询,如果观察到建立hash索引可以带来速度提升,则建立hash索引,称之为自适应hash索引.AHI是通过缓冲池的B+树叶构造而来的,因此建立速度很快.而且不需要对整张表构建hash索引.InnoDB存储引擎会自动根据访问的频率和模式,来自动的为某些热点建立hash索引.

交替查询不会创建AHI,

启用AHI,读写速度提升2倍,辅助索引的连接操作性能可以提升5倍.AHI的设计思想是数据库自优化,不需要dba对数据库进行手动调整.

show engine innodb status\G

查看AHI的信息.设置此属性的开启状态,默认是开启的.

82. 为什么要用 B+tree作为MySQL索引的数据结构

1).文件很大,不可能全部存储在内存中,故要存储在磁盘上,

2).索引的结构组织要尽量减少查找过程中磁盘io的存取此时.

3).局部性原理与磁盘预读,预读的长度一般为页的整数倍,页的大小4k.

4).数据库系统巧妙地利用了磁盘预读原理.将一个节点的大小设为一个页,每个节点只需要一个io就能载入。

83. 聚集索引与非聚集索引

聚集索引和非聚集索引引用的是B+树索引。

聚集索引：数据行的物理顺序与列值（一般是主键的那一列）的逻辑顺序相同，一个表只能拥有一个聚集索引。

系统会自动创建一个隐含列作为表的聚集索引。

mysql中的主键就是聚集索引。

非聚集索引：相反的，一个表可以拥有多个非聚集索引。

使用聚集索引的查询效率比非聚集索引的效率高，如果需要频繁修改聚集索引的值，写入性能不高，需要移动对应数据的物理位置。

非聚集索引在查询的时候可以避免二次查询，提高性能。

不是所有的表适合建立索引，数据量大的才适合建立索引。

84.索引失效

隐式转换

where条件进行运算

使用内部函数

使用<> ,not in ,not exist,!=,like %号在前面,null,单独引用符合索引非第一位置的索引列,time和date

使用双引号.

将索引的所在的表空间和数据所在的表空间分别植物不同的磁盘上,有助于提高索引查询的效率.

Oracle使用的基于代价的sql优化器非常依赖于统计信息.一旦统计信息不正常,查询时就不使用索引或使用错误的索引.

oracle在查询时只会用一个索引

优先尽可能使用分区索引.

85. limit 20000 加载很慢怎么解决

当数据表过于庞大,需要增加orderby ,orderby的字段需要建立索引.

实用子查询,子查询必须是连续的没有where条件,where条件会过滤数据的连续性.

limit分页优化方法:

子查询优化法:先找出第一条数据,然后大于这条数据的就是要获取的数据

缺点,数据必须是连续的,不能有where条件.

倒排表优化法:

倒排表类似于建立索引,用一张表维护页数,然后通过高效的连接得到数据

只适合数据固定的情况,数据不能删除,维护页表困难.

反向查找优化法:

当偏移超过一半时,先用排序,这样偏移就反转了,

缺点:order by 优化比较麻烦,要增加索引,索引影响数据的修改效率,并且要知道总记录数偏移偏移大于数据的一般.

limit限制优化法:

把limit偏移量下那会低于某个数,超过这个数等于没数据.

86.如何选择合适的分布式主键

分布式系统中,经常需要对大量的数据,消息,http请求进行唯一标识.在分布式系统之间http请求需要一个唯一标识,调用链路分析时,需要使用这个唯一id.需要生成一个全局唯一id

特点:全局唯一,不能出现重复id;高可用:id生成系统是基础系统,被许多关键系统使用,一旦宕机,造成严重影响.

经典方案:

UUID计算机算法随机生成,本低生成,不需要进行远程调用,时延低,性能高.缺点过长36 的长度,很多场景不使用,没有排序无法保证趋势递增.

Flicker方案:主要采用了mysql自增长id的机制.充分借助了数据库的自增id机制,可靠性高,生成有序的id.缺点:id生成依赖单台数据库的读写性能.依赖于数据库,当数据库异常时,整个系统不可用.

类snowflake方案:

生成64byte的数字,分别为时间戳,机器编码,序号

优点:时间戳在高位,自增序列在高位,整个id趋势是递增的,按照时间有序.性能高,每秒生成几百万id.可以根据自身业务灵活调整byte位,满足不同需求.

缺点:依赖机器时钟,如果机器时钟回拨,造成重复id生成.

在单机上是递增的,涉及到分布环境,每台机器上的时钟不可能完全同步,有时候会出现不是全局递增的情况.

TDDL序列生成方式:

TDDL是阿里的分库分表中间件,他里面包含了全局数据库id的生成方式,主要思路是:

使用数据库同步id信息.每次批量取一定数量的可用ID置于内存中,使用完后,在请求数据库重新获取下一批可用id,每次获取的可用id数量由步长控制,实际业务中可以根据使用速度进行配置.

每个业务可以给自己的序列起一个唯一的名字,隔离各个业务系统的id.

优点,相对于flicker降低了数据库写压力,数据库不再是性能瓶颈,生成id性能大幅度提高,因为获取可用号段分配,比每次读取性能提高,,不同业务可以区分.

缺点:强烈依赖数据库,当数据库异常时整个系统不可用.

87. 选择合适的数据存储方案

关系型数据库MYSQL

内存数据库Redis基于内存存储数据,可以提高查询性能.

文档数据库MongoDB

列族数据库HBase

全文搜索引擎ElasticSearch

88.MySql数据库优化方案

选用合适的字段属性,尽量设置为Null;

使用Join代替子查询;

使用UNION代替手动创建临时表

事务

索引

外键 :保证数据的关联性

锁定表

优化sql