#### String和StringBuffer、StringBuilder的区别

可变性

String类中使用字符数组保存字符串，private

final char value[]，所以string对象是不可变的。StringBuilder与StringBuffer都继承自AbstractStringBuilder类，在AbstractStringBuilder中也是使用字符数组保存字符串，char[]

value，这两种对象都是可变的。

线程安全性

String中的对象是不可变的，也就可以理解为常量，线程安全。StringBuffer对方法加了同步锁或者对调用的方法加了同步锁，所以是线程安全的。StringBuilder并没有对方法进行加同步锁，所以是非线程安全的。

性能

StringBuffer相比StringBuilder加了锁，所以性能比StringBuilder差

#### 运行时数据区

根据 JVM 规范，JVM 内存共分为虚拟机栈、堆、方法区、程序计数器、本地方法栈五个部分。

运行时常量池在方法区中

虚拟机栈（JVM Stack）：一个线程的每个方法在执行的同时，都会创建一个栈帧（Statck Frame），栈帧中存储的有局部变量表、操作站、动态链接、方法出口等，当方法被调用时，栈帧在JVM栈中入栈，当方法执行完成时，栈帧出栈。局部变量表中存储着方法的相关局部变量，包括各种基本数据类型，对象的引用，返回地址等。在局部变量表中，只有long和double类型会占 用2个局部变量空间（Slot，对于32位机器，一个Slot就是32个bit），其它都是1个Slot。需要注意的是，局部变量表是在编译时就已经确定 好的，方法运行所需要分配的空间在栈帧中是完全确定的，在方法的生命周期内都不会改变。虚拟机栈中定义了两种异常，如果线程调用的栈深度大于虚拟机允许的最大深度，则抛出StatckOverFlowError（栈溢出）；不过多 数Java虚拟机都允许动态扩展虚拟机栈的大小(有少部分是固定长度的)，所以线程可以一直申请栈，知道内存不足，此时，会抛出 OutOfMemoryError（内存溢出）。每个线程对应着一个虚拟机栈，因此虚拟机栈也是线程私有的。

堆区（Heap）：堆区是理解Java GC机制最重要的区域，没有之一。在JVM所管理的内存中，堆区是最大的一块，堆区也是Java GC机制所管理的主要内存区域，堆区由所有线程共享，在虚拟机启动时创建。堆区的存在是为了存储对象实例，原则上讲，所有的对象都在堆区上分配内存（不过现代技术里，也不是这么绝对的，也有栈上直接分配的）。堆内存需要在逻辑上是连续的（在物理上不需要），堆内存不够时将会抛出OutOfMemoryError:Java heap space异常。

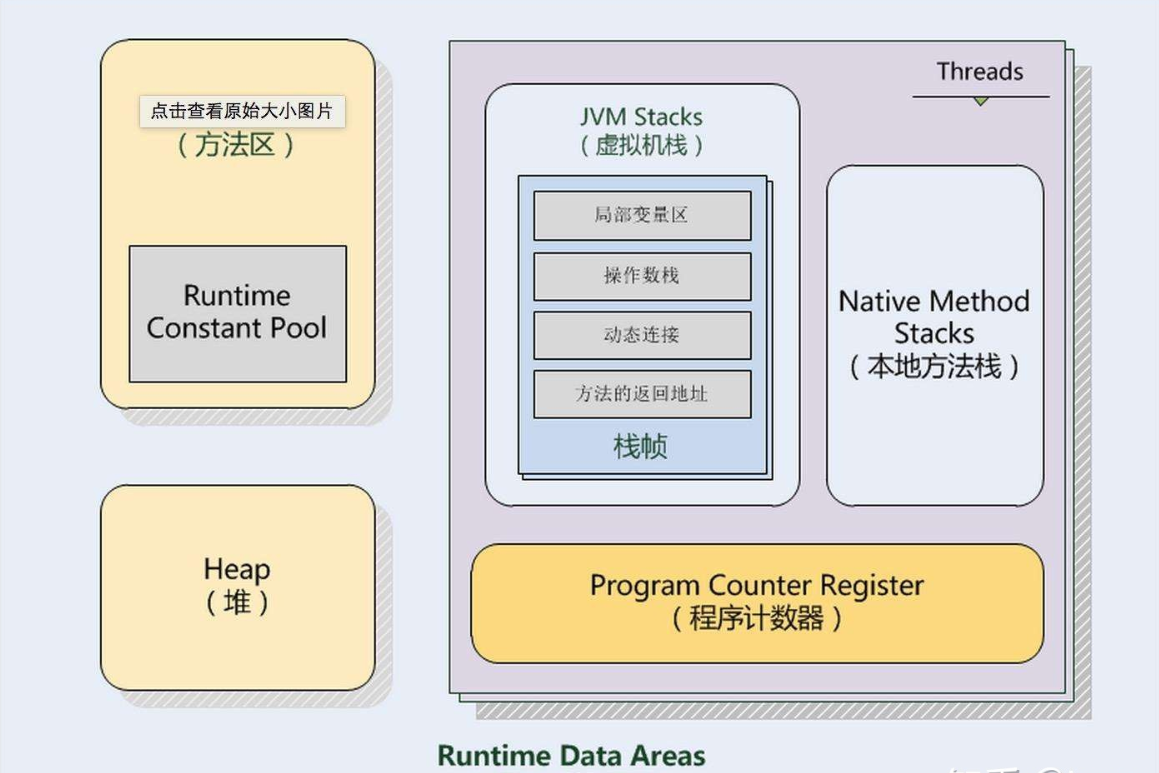
方法区（Method Area）：在Java虚拟机规范中，将方法区作为堆的一个逻辑部分来对待，但事实 上，方法区并不是堆（Non-Heap）；方法区是各个线程共享的区域，用于存储已经被虚拟机加载的类信息（即加载类时需要加载的信息，包括版本、field、方法、接口等信息）、final常量、静态变量、编译器即时编译的代码等。方法区在物理上也不需要是连续的，可以选择固定大小或可扩展大小，并且方法区比堆还多了一个限制：可以选择是否执行垃圾收集。一般的，方法区上 执行的垃圾收集是很少的，这也是方法区被称为永久代的原因之一（HotSpot），但这也不代表着在方法区上完全没有垃圾收集，其上的垃圾收集主要是针对 常量池的内存回收和对已加载类的卸载。在方法区上进行垃圾收集，条件苛刻而且相当困难，效果也不令人满意，所以一般不做太多考虑，可以留作以后进一步深入研究时使用。在方法区上定义了OutOfMemoryError:PermGen space异常，在内存不足时抛出。

运行时常量池（Runtime Constant Pool）是方法区的一部分，用于存储编译期就生成的字面常量、符号引用、翻译出来的直接引用（符号引用就是编码是用字符串表示某个变量、接口的位置，直接引用就是根据符号引用翻译出来的地址，将在类链接阶段完成翻译）；运行时常量池除了存储编译期常量外，也可以存储在运行时间产生的常量（比如String类的intern()方法，作用是String维护了一个常量池，如果调用的字符“abc”已经在常量池中，则返回池中的字符串地址，否则，新建一个常量加入池中，并返回地址）。

程序计数器（Program Counter Register）：程序计数器是一个比较小的内存区域，用于指示当前线程所执行的字节码执行到了第几行，可以理解为是当前线程的行号指示器。字节码解释器在工作时，会通过改变这个计数器的值来取下一条语句指令。每个程序计数器只用来记录一个线程的行号，所以它是线程私有（一个线程就有一个程序计数器）的。如果程序执行的是一个Java方法，则计数器记录的是正在执行的虚拟机字节码指令地址；如果正在执行的是一个本地（native，由C语言编写 完成）方法，则计数器的值为Undefined，由于程序计数器只是记录当前指令地址，所以不存在内存溢出的情况，因此，程序计数器也是所有JVM内存区 域中唯一一个没有定义OutOfMemoryError的区域。

本地方法栈（Native Method Statck）：本地方法栈在作用，运行机制，异常类型等方面都与虚拟机栈相同，唯一的区别是：虚拟机栈是执行Java方法的，而本地方法栈是用来执行native方法的，在很多虚拟机中（如Sun的JDK默认的HotSpot虚拟机），会将本地方法栈与虚拟机栈放在一起使用。本地方法栈也是线程私有的。

直接内存（Direct Memory）：直接内存并不是JVM管理的内存，可以这样理解，直接内存，就是 JVM以外的机器内存，比如，你有4G的内存，JVM占用了1G，则其余的3G就是直接内存，JDK中有一种基于通道（Channel）和缓冲区 （Buffer）的内存分配方式，将由C语言实现的native函数库分配在直接内存中，用存储在JVM堆中的DirectByteBuffer来引用。 由于直接内存收到本机器内存的限制，所以也可能出现OutOfMemoryError的异常。



#### HashMap和HashTable区别

1）.HashTable的方法前面都有synchronized来同步，是线程安全的；HashMap未经同步，是非线程安全的。

2）.HashTable不允许null值(key和value都不可以) ；HashMap允许null值(key和value都可以)。

3）.HashTable有一个contains(Object

value)功能和containsValue(Object

value)功能一样。

4）.HashTable使用Enumeration进行遍历；HashMap使用Iterator进行遍历。

5）.HashTable中hash数组默认大小是11，增加的方式是old\*2+1；HashMap中hash数组的默认大小是16，而且一定是2的指数。

6）.哈希值的使用不同，HashTable直接使用对象的hashCode； HashMap重新计算hash值，而且用与代替求模。

#### ArrayList和vector区别

ArrayList和Vector都实现了List接口，都是通过数组实现的。

Vector是线程安全的，而ArrayList是非线程安全的。

List第一次创建的时候，会有一个初始大小，随着不断向List中增加元素，当List 认为容量不够的时候就会进行扩容。Vector缺省情况下自动增长原来一倍的数组长度，ArrayList增长原来的50%。

#### ArrayList和LinkedList区别及使用场景

区别

ArrayList底层是用数组实现的，可以认为ArrayList是一个可改变大小的数组。随着越来越多的元素被添加到ArrayList中，其规模是动态增加的。

LinkedList底层是通过双向链表实现的， LinkedList和ArrayList相比，增删的速度较快。但是查询和修改值的速度较慢。同时，LinkedList还实现了Queue接口，所以他还提供了offer(),

peek(), poll()等方法。

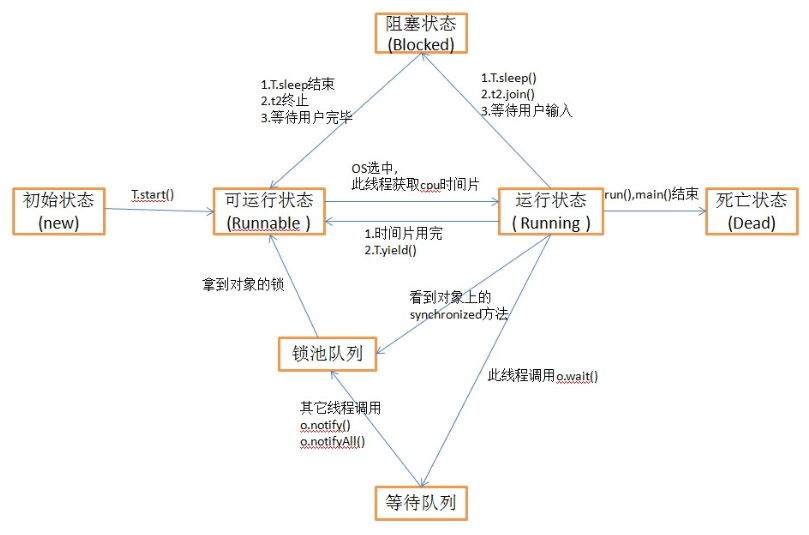
使用场景

LinkedList更适合从中间插入或者删除（链表的特性）。

ArrayList更适合检索和在末尾插入或删除（数组的特性）。

#### 线程的状态转换

线程一共有如图所示的5种状态



这里记录一下等待队列的状态切换

调用obj的wait(), notify()方法前，必须获得obj锁，也就是必须写在synchronized(obj) 代码段内。

与等待队列相关的步骤和图

线程1获取对象A的锁，正在使用对象A。

线程1调用对象A的wait()方法。

线程1释放对象A的锁，并马上进入等待队列。

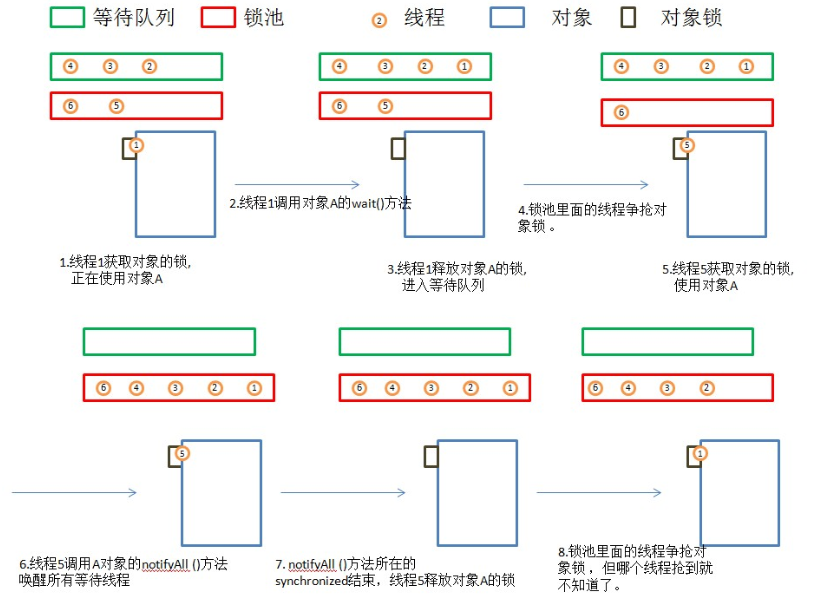
锁池里面的对象争抢对象A的锁。

线程5获得对象A的锁，这时候进入的是可运行状态，被系统分配了时间片，才进入运行状态，进入synchronized块，使用对象A。

线程5调用对象A的notifyAll()方法，唤醒所有线程，所有线程进入锁池。||||| 线程5调用对象A的notify()方法，唤醒一个线程，不知道会唤醒谁，被唤醒的那个线程进入锁池。

notifyAll()方法所在的synchronized结束，线程5释放对象A的锁。

锁池里面的线程争抢对象锁，但线程1什么时候能抢到就不知道了。||||| 原本锁池+第6步被唤醒的线程一起争抢对象锁。



#### sleep和wait的区别(考察的方向是是否会释放锁)

sleep()方法是Thread类中方法，而wait()方法是Object类中的方法。

sleep()方法导致了程序暂停执行指定的时间，让出cpu该其他线程，但是他的监控状态依然保持者，当指定的时间到了又会自动恢复运行状态，在调用sleep()方法的过程中，线程不会释放对象锁。而当调用wait()方法的时候，线程会放弃对象锁，进入等待此对象的等待锁定池，只有针对此对象调用notify()方法后本线程才进入对象锁定池准备。

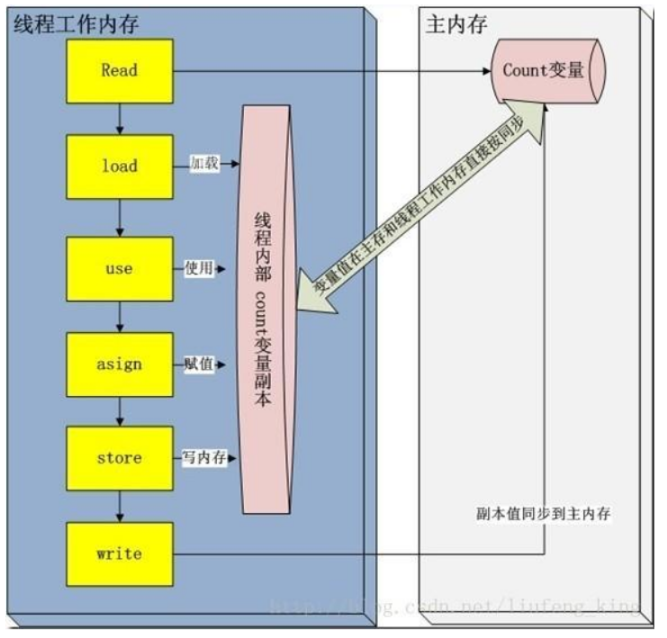
#### volatile关键字的理解

1. 阻止指令重排序

指令重排序可能导致异常，比如双检锁单例中如果没有保证指令顺序，有可能导致引用了空的对象，抛出空指针异常

1. volatile无法保证原子性

每一个线程运行时都有一个线程栈，线程栈保存了线程运行时候变量值信息。当线程访问某一个对象时候值的时候，首先通过对象的引用找到对应在堆内存的变量的值，然后把堆内存变量的具体值load到线程本地内存中，建立一个变量副本，之后线程就不再和对象在堆内存变量值有任何关系，而是直接修改副本变量的值，在修改完之后的某一个时刻（线程退出之前），自动把线程变量副本的值回写到对象在堆中变量。这样在堆中的对象的值就产生变化了。由于volatile不具有原子性，所以存在两个线程都将volatile对象存入线程副本变量，然后重复写到主内存，导致最终的结果并不正确。



read and load 从主存复制变量到当前工作内存

use and assign 执行代码，改变共享变量值

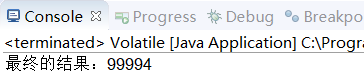
store and write 用工作内存数据刷新主存相关内容

其中use and

assign 可以多次出现，但是这一些操作并不是原子性，也就是在read load之后，如果主内存count变量发生修改之后，线程工作内存中的值由于已经加载，不会产生对应的变化，所以计算出来的结果会和预期不一样。

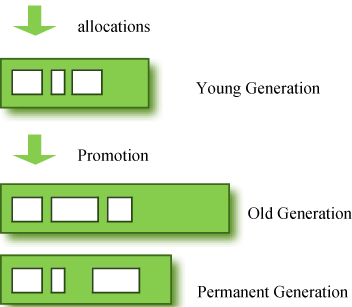
相对应的代码示例，由于无法保证原子性，最终结果和预期并不相同





#### Java内存分配机制

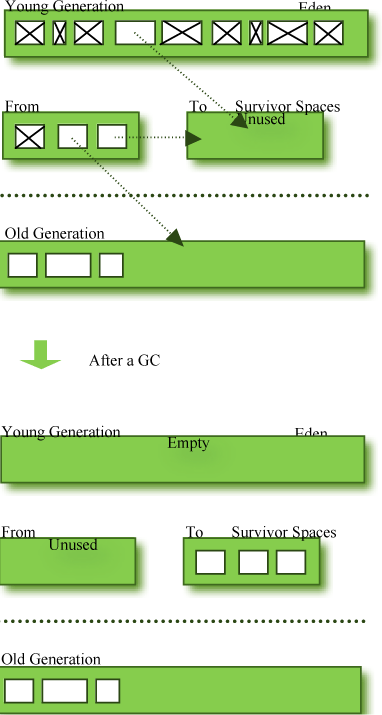
jvm内存分为：年轻代，年老代，永久代（Permanent Generation，也就是方法区）



年轻代（Young Generation）：一个Eden区，两个Survivor区。用-XX:SurvivorRatio参数来配置Eden区域Survivor区的容量比值，默认是8，代表Eden：Survivor1：Survivor2=8:1:1.

年轻代的gc叫Minor GC或叫Young GC。绝大多数刚创建的对象会被分配在Eden区，其中的大多数对象很快就会消亡。Eden区是连续的内存空间，因此在其上分配内存极快；当Eden区满的时候，执行Minor GC，将消亡的对象清理掉，并将剩余的对象复制到一个存活区Survivor0（此时，Survivor1是空白的，两个Survivor总有一个是空白的）；此后，每次Eden区满了，就执行一次Minor GC，并将剩余的对象都添加到Survivor0；当Survivor0也满的时候，将其中仍然活着的对象直接复制到Survivor1，以后Eden区执行Minor GC后，就将剩余的对象添加Survivor1（此时，Survivor0是空白的）。当两个存活区切换了几次（HotSpot虚拟机默认15次，用-XX:MaxTenuringThreshold控制，大于该值进入老年代）之后，仍然存活的对象（其实只有一小部分，比如，我们自己定义的对象），将被复制到老年代。年轻代使用的是“停止-复制（Stop-and-copy）”清理法（将Eden区和一个Survivor中仍然存活的对象拷贝到另一个Survivor中）

在Eden区，HotSpot虚拟机使用了两种技术来加快内存分配。分别是bump-the-pointer和TLAB（Thread- Local Allocation Buffers），这两种技术的做法分别是：由于Eden区是连续的，因此bump-the-pointer技术的核心就是跟踪最后创建的一个对象，在对象创建时，只需要检查最后一个对象后面是否有足够的内存即可，从而大大加快内存分配速度；而对于TLAB技术是对于多线程而言的，将Eden区分为若干 段，每个线程使用独立的一段，避免相互影响。TLAB结合bump-the-pointer技术，将保证每个线程都使用Eden区的一段，并快速的分配内 存。



年老代（Old Generation）：年老代分配两类对象，1.多次young gc还存活的对象2.大对象不分配在年轻代，直接分配在年老代。年老代的gc称为Major GC，也叫 Full GC。

可能存在年老代对象引用新生代对象的情况，如果需要执行Young GC，则可能需要查询整个老年代以确定是否可以清理回收，这显然是低效的。解决的方法是，年老代中维护一个512 byte的块——”card table“，所有老年代对象引用新生代对象的记录都记录在这里。Young GC时，只要查这里即可，不用再去查全部老年代，因此性能大大提高。

在发生Minor GC时，虚拟机会检查每次晋升进入老年代的大小是否大于老年代的剩余空间大小，如果大于，则直接触发一次Full GC，否则，就查看是否设 置了-XX:+HandlePromotionFailure（允许担保失败），如果允许，则只会进行MinorGC，此时可以容忍内存分配失败；如果不 允许，则仍然进行Full GC（这代表着如果设置-XX:+Handle PromotionFailure，则触发MinorGC就会同时触发Full GC，哪怕老年代还有很多内存，所以，最好不要这样做）。

#### 方法区（永久代）的内存回收

永久代的回收有两种：常量池中的常量，无用的类信息。

常量的回收很简单，没有引用了就可以被回收。对于无用的类进行回收，必须保证3点：

1.类的所有实例都已经被回收

2.加载类的ClassLoader已经被回收

3.类对象的Class对象没有被引用（即没有通过反射引用该类的地方）

永久代的回收并不是必须的，可以通过参数来设置是否对类进行回收。HotSpot提供-Xnoclassgc进行控制

使用-verbose，-XX:+TraceClassLoading、-XX:+TraceClassUnLoading可以查看类加载和卸载信息

-verbose、-XX:+TraceClassLoading可以在Product版HotSpot中使用；

-XX:+TraceClassUnLoading需要fastdebug版HotSpot支持

#### 乐观锁和悲观锁

悲观锁即认为数据之间存在激烈竞争，为了确保数据安全，会将数据锁住，只有当线程获得了唯一的锁的时候才会写入数据。当其他的线程或者数据连接修改数据的时候，则需要等待获取到锁

乐观锁默认认为线程少有竞争或者竞争不激烈，一般在更新数据的时候会带一个额外的字段来进行校验，比如数据库更新数据，可以加入一个版本号来进行校验，当更新特定数据之后，也要将版本号进行更新，当其他的线程进行更新数据的时候，发现版本号不匹配就会放弃更新数据，以此来保证数据安全可靠。1.该操作降低了性能损耗2.避免了锁带来的死锁问题3.可能出现aba问题。aba问题主要出现在数据改变的时候用来更新的关键关键数据存在重复的场景，aba问题时间点如下

|  |  |  |
| --- | --- | --- |
| 时间点 | 线程1 | 线程2 |
| t1 | 查询值是否为A |  |
| t2 |  | 查询值是否为A |
| t3 |  | 将值更新为B |
| t4 |  | 查询值是否为B |
| t5 |  | 将值更新为A |
| t6 | 将值更新为C |  |

t1和t2时间内，两个线程是干相同的活，但是操作的数据（在更新值之外，还操作了其他的数据）不一样。原本线程1的操作应该被舍弃，因为线程2获得了修改数据的权限。但是在t2-t5时间内，线程2又将用来判断的值改了回去，导致线程1也获得了修改数据的权利，将值修改了，更重要的是同时也修改了其他的数据。为了避免这种情况产生，需要保证用来校验能否做更新的数据的值是不会重复的，比如每次更新了后进行加1操作，而不是用容易重复的值导致出现异常。

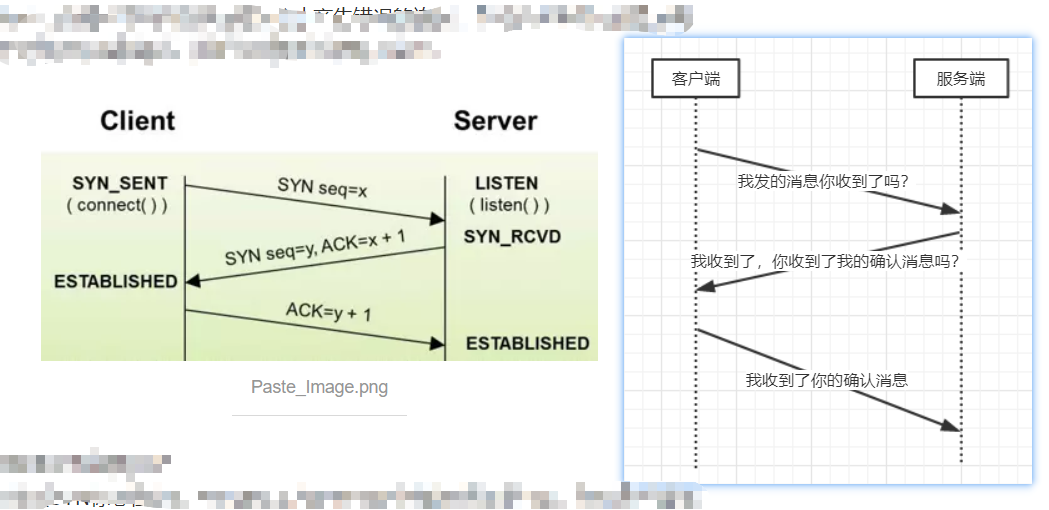
#### 三次握手和四次挥手

SYN：请求

ACK：应答

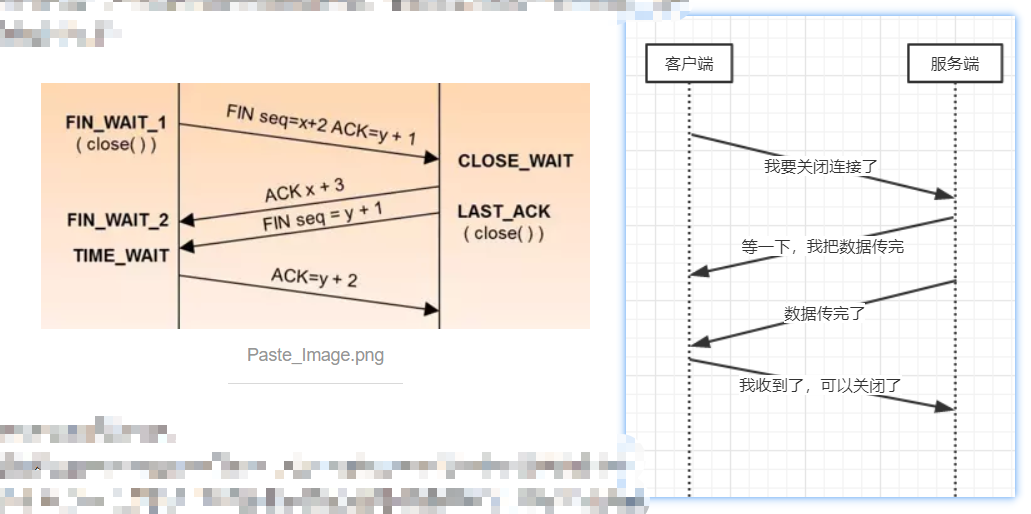
三次握手

当客户端和服务器之间都进行了一次请求和应答之后，连接就建立起来了。三次握手是连接建立的过程



四次挥手

客户端请求断开连接，服务器响应两次后，客户端再应答一次服务器。四次挥手是连接断开的过程。之所以服务端响应两次，因为第一次会需要一点时间处理断开，不一定是传输数据



#### servlet生命周期

初始化：init()只加载一次，启动服务器或客户机首次访问时调用

运行：service()方法，响应客户端请求

销毁：destroy()方法，仅执行一次，在服务器端停止且卸载Servlet时执行该方法。当Servlet对象退出生命周期时，负责释放占用的资源。一个Servlet在运行service()方法时可能会产生其他的线程，因此需要确认在调用destroy()方法时，这些线程已经终止或完成。

#### 面向对象的三大特性-封装、继承、多台

封装

可以设置自己的属性和方法的访问权限

继承

可以用来复用代码

多态

不同类的对象对同一消息作出不同的响应叫做多态。不同的子类调用相同类型的方法可以得到不同的结果

#### 数据库事务的几个特性：ACID

A：Atomicity，原子性

C： Consistency，一致性

I：Isolation，隔离性

D：Durability，持久化

原子性：要么成功，要么全部失败，不存在中间状态

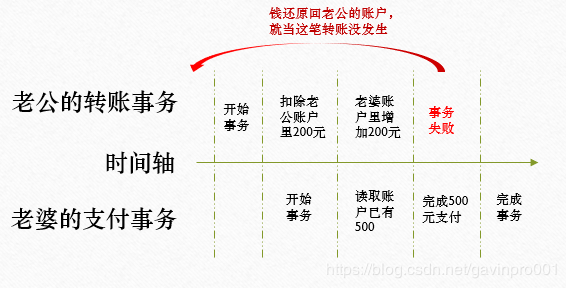
一致性：事务前后的数据完整性保持一致。比如转账，AB之间转账任意金额任意次数，AB之间的总金额需要保持不变

隔离性：数据库具有的解决多个事务同时对同一个数据库的同一条或多条数据进行操作出现问题的技术。有四个隔离级别

1）读未提交（Read uncommitted）： 一个事务可以读取另一个未提交事务的中间数据

这里说的“提交”是指事务完成的标志，一个事务成功提交就说明这个事务成功完成，不然事务就会一直处在执行中的状态，涉及修改的数据也不是最终值，如果数据库事务隔离级别设定为读未提交，那么该事务中间状态的数据就可能会被其它事务读到，用到别的事务应用中并导致错误的发生，比如以下示例：

老婆的银行账户里存有300元，打算使用自己的账户在网上购买一套500元的化妆品，但账户的钱不够支付该订单，然后就打电话给老公，让他转200元到她的账户，老公答应马上就转并同时告诉她过一会试试能不能支付，接下来老公通过手机银行APP启动一笔转200元到老婆账户的转账事务，但老婆并不知道钱什么时候到账，干脆就直接在网上启动支付事务试试，正常来说，需要等老公把这笔转账事务完成以后，老婆才能开始支付事务，但如果数据库隔离级别设定为读未提交，那么就会出现问题，见下图：



从上图，我门可以清晰的看到，老公发起的转账事务是先启动的，事务中有一项操作是给老婆账户里加200元，因为数据库设定的隔离级别是读未提交，那么老婆发起的支付事务是可以读到老公转账事务过程中的数据，当转账事务把老婆账户里的钱刚增加到500块，恰好接下来被老婆的支付事务读到并完成了500元支付，但之后老公的转账事务因系统问题未能成功提交，那么就按照失败回滚处理，就当没有给老婆转过钱，但结果老婆确能把这笔钱给花了，老公的钱一分钱也没少，亏的最后是银行。

由以上给出的示例，老婆的支付事务读取的那500元钱，其实并不是真实的，因为，老公的转账事务并没有真实的把那200元钱转过来，所以支付事务读出来的数据是不真实的，这在数据库事务概念中有一个专业的术语，叫“脏读”，即一个事务读取了另一个事务的未提交的数据。

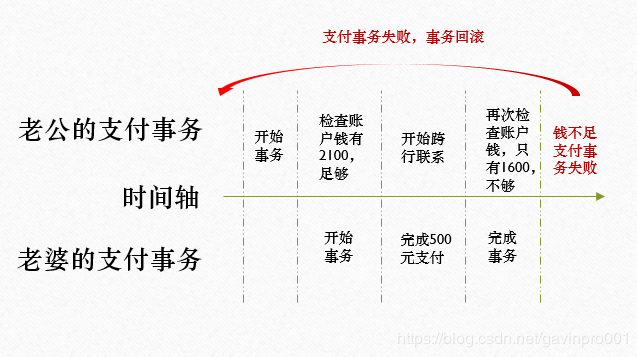
如果解决读未提交产生的脏读问题呢，那就需要将数据库事务的隔离级别提高到读提交的级别。

2）读提交（Read committed）： 一个事务要等另一个事务提交后才能读取数据

一个事务正在修改一条或多条数据，那么其它事务是不可以读这些数据，必须要等第一个事务完成后才能读，这样就不会像读未提交一样，其它事务读取了脏数据。

但读提交也有它的局限性，因为读提交只解决了不让其它事务读本事务正修改的数据问题，但控制不了其它事务修改其要读的数据，在有些场景依然会出现问题，如下面这个例子：

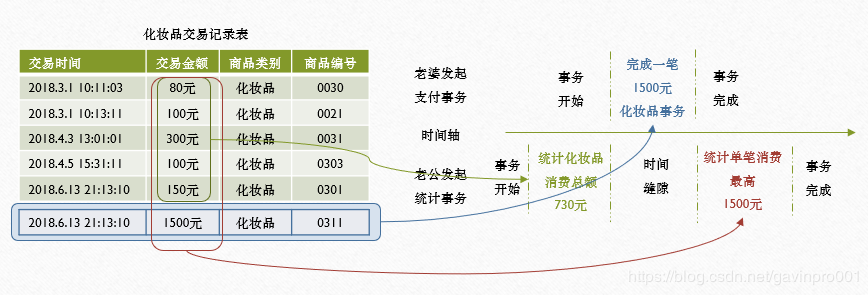
老公打算通过其银行账户在一个网上购物商城买一部2000元的小米手机，他知道自己账户里有2100元，然后点击支付按钮开始走支付事务，这个网商平台是实时结算，商城的账户在另一家银行，那么商城的第三方支付系统首先会先查老公的账户里钱够不够，然后联系另一家银行的系统准备划账，但就在这个时候，她老婆用他的卡在另一家商城购物，大概花费500元，也发起了一个支付事务，但这个事务完成的速度很快，把钱直接扣掉500元，剩下1600元，但接下来老公的这笔支付事务准备开始扣除2000元钱的时候，再次查询发现钱不够了，然后就通知老公剩下的钱不足，不能完成支付，老公在支付页面上看到的情况就是，您账户储蓄满足支付，开始跨行支付中->对不起，您账户的钱不足无法完成支付，并且是等了很久才给出的账户钱不足提示。这个问题就是 不可重复读 问题，一个事务中前后两次读同一条数据，结果值是不一样的，要解决不可重复读问题，就要将隔离级别提升到可重复读级别。



3）可重复读（Repeatable read）： 在一个事务开启时，其它事务不能修改其要读取的数据

当一个事务开始后，该事务所有涉及读的数据都不能被其它的事务修改，这样就能保证其事务执行过程中多次读取同样的数据不会发生变化，这种隔离级别在数据库事务中已经算比较高的级别，基本可以满足大多数事务应用的场景，但是可重复读虽然能够防止其它事务修改它的数据，但不能防止其它事务在它读取数据范围插入数据，举例：

老公通过使用银行的APP软件做一个统计，主要包括两项，第一项是找出化妆品总共消费额，另一项是化妆品单笔最高消费金额，老公启动了这次的统计事务，统计这两项都支出多少钱，事务开启以后，数据库系统先按化妆品的类别码进行扫描，将所有化妆品的订单记录都扫描出来，然后计算个汇总值，然后数据库系统进行第2次扫描，本次扫描是通过排序找出消费金额最高的单笔消费，但最后APP返回出来的结果就是，化妆品总共消费1000元，化妆品最高单笔消费1500元，这样老公很是纳闷，为什么总共消费1000元，其中的最高单笔消费确是1500元，不合理呀，其实问题就出在数据库系统统计事务第1项刚做完，正准备开始下一项统计的时候，老婆发起了另一笔支付事务完成了一笔1500元的化妆品消费，结果就在老公的统计事务的两项统计操作之间，出现了一笔新的订单。



由上边的示例可以了解到，可重复读不能解决数据库事务的幻读问题，幻读是什么呢？从字面上就能理解，就是在一个事务中重复读两次一个范围的数据，但发现第2次多读了数据出来，感觉就像是发生了幻觉，幻读同样也是数据库事务在多用户同时并发使用中出现的问题，但可重复读的事务隔离级别已经是很高的了，使用的也非常广泛，如果真的想杜绝所有的事务隔离问题，那只有最后一个隔离级别–序列化。

4）Serializable序列化：Serializable 是最高的事务隔离级别，在该级别下，事务串行化顺序执行，一个事务完成以后开始下一个事务，不会出现前三个隔离级别出现的脏读、不可重复读以及幻读的问题，但这种隔离级别效率最为低下，一般数据库事务不会使用这种隔离级别。

持久化：一个数据库事务一旦成功提交完成后，所有这个事务对数据库所有更改的内容全部被保留下来，在数据库系统正常运行的过程中，除非新的事务要去更改，否则内容不会改变，以及一些系统异常的情况下除非是存储磁盘损坏，不然数据也不会改变。

#### 分布式事务

CAP定理

CAP定理是由加州大学伯克利分校Eric Brewer教授提出来的，他指出WEB服务无法同时满足一下3个属性：

一致性(Consistency) ： 客户端知道一系列的操作都会同时发生(生效)

可用性(Availability) ： 每个操作都必须以可预期的响应结束

分区容错性(Partition tolerance) ： 即使出现单个组件无法可用,操作依然可以完成

具体地讲在分布式系统中，在任何数据库设计中，一个Web应用至多只能同时支持上面的两个属性。显然，任何横向扩展策略都要依赖于数据分区。因此，设计人员必须在一致性与可用性之间做出选择。

BASE理论

在分布式系统中，我们往往追求的是可用性，它的重要程序比一致性要高，那么如何实现高可用性呢？ 前人已经给我们提出来了另外一个理论，就是BASE理论，它是用来对CAP定理进行进一步扩充的。BASE理论指的是：

Basically Available（基本可用）

Soft state（软状态）

Eventually consistent（最终一致性）

BASE理论是对CAP中的一致性和可用性进行一个权衡的结果，理论的核心思想就是：我们无法做到强一致，但每个应用都可以根据自身的业务特点，采用适当的方式来使系统达到最终一致性（Eventual consistency）。

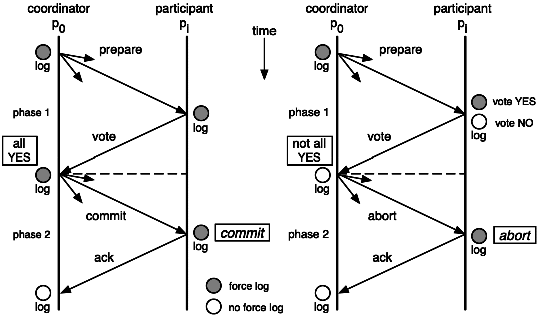
几种分布式事务的方式

两阶段提交（2PC）

XA 是一个两阶段提交协议，该协议分为以下两个阶段：

第一阶段：事务协调器要求每个涉及到事务的数据库预提交(precommit)此操作，并反映是否可以提交.

第二阶段：事务协调器要求每个数据库提交数据。



补偿事务（TCC）

TCC 其实就是采用的补偿机制，其核心思想是：针对每个操作，都要注册一个与其对应的确认和补偿（撤销）操作。它分为三个阶段：

Try 阶段主要是对业务系统做检测及资源预留

Confirm 阶段主要是对业务系统做确认提交，Try阶段执行成功并开始执行 Confirm阶段时，默认 Confirm阶段是不会出错的。即：只要Try成功，Confirm一定成功。

Cancel 阶段主要是在业务执行错误，需要回滚的状态下执行的业务取消，预留资源释放。

举个例子，假入 Bob 要向 Smith 转账，思路大概是：

我们有一个本地方法，里面依次调用

1、首先在 Try 阶段，要先调用远程接口把 Smith 和 Bob 的钱给冻结起来。

2、在 Confirm 阶段，执行远程调用的转账的操作，转账成功进行解冻。

3、如果第2步执行成功，那么转账成功，如果第二步执行失败，则调用远程冻结接口对应的解冻方法 (Cancel)。

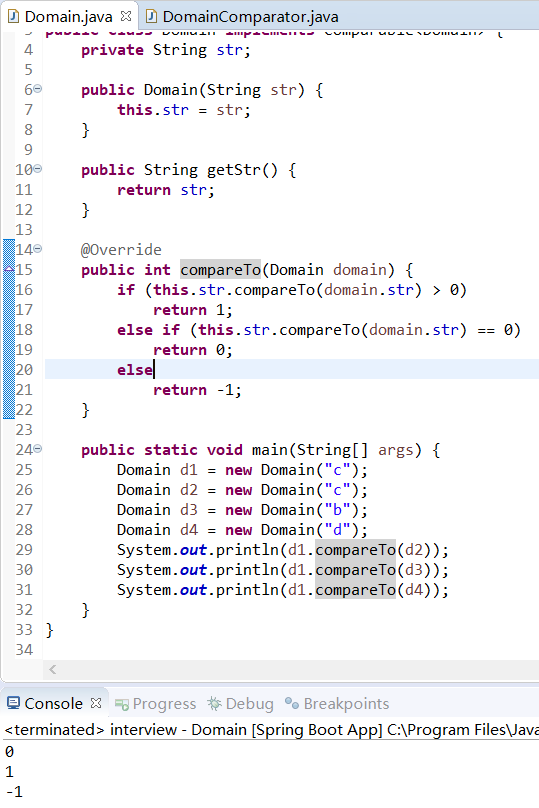
优点： 跟2PC比起来，实现以及流程相对简单了一些，但数据的一致性比2PC也要差一些

缺点： 缺点还是比较明显的，在2,3步中都有可能失败。TCC属于应用层的一种补偿方式，所以需要程序员在实现的时候多写很多补偿的代码，在一些场景中，一些业务流程可能用TCC不太好定义及处理。

#### Comparable和Comparator的区别

Comparable

1.实现了Comparable接口的对象可以通过obj1. compareTo(obj2)的方式将自己和另外一个对象进行比较。2.实现Comparable接口需要实现compareTo方法



Comparator

1. Comparator接口通过this.compare(obj1,obj2)的方式将没办法进行比较的两个对象进行比较（也可以是能够进行比较的对象，没有区别）

2. Comparator对象需要实现compare方法

3. Comparator指定了特定的泛型

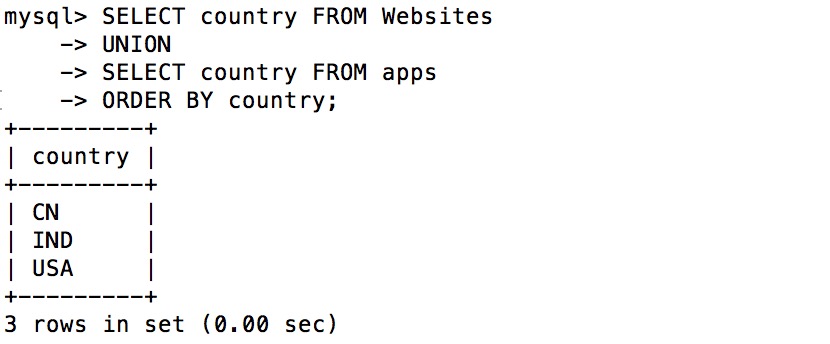


#### UNION操作符

UNION 操作符用于合并两个或多个 SELECT 语句的结果集。

UNION 操作符选取不同的值。如果允许重复的值，请使用 UNION ALL。

union的结果



union all的结果



#### 事务的传递

事务的配置

@Transactional

@Transactional(propagation = Propagation.***NEVER***)

事务的几种传递方式

REQUIRED（默认）：支持当前事务，如果没有则创建一个新的

SUPPORTS：支持当前事务，如果没有则不使用事务

MANDATORY：支持当前事务，如果没有事务则报错

REQUIRED\_NEW：新建一个事务，同时将当前事务挂起

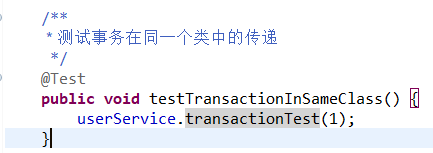
NOT\_SUPPORTED：以无事务的方式执行，如果当前有事务则将其挂起

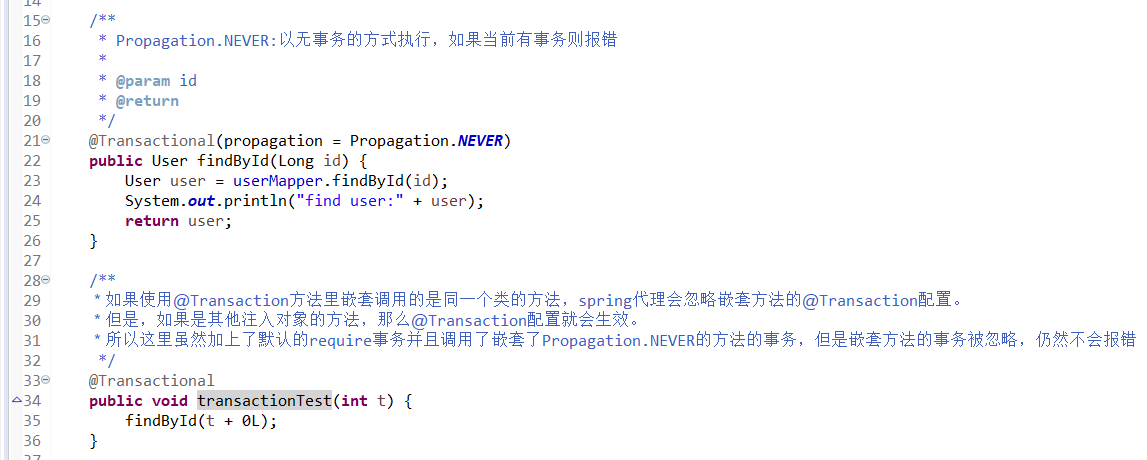
NEVER：以无事务的方式执行，如果当前有事务则报错

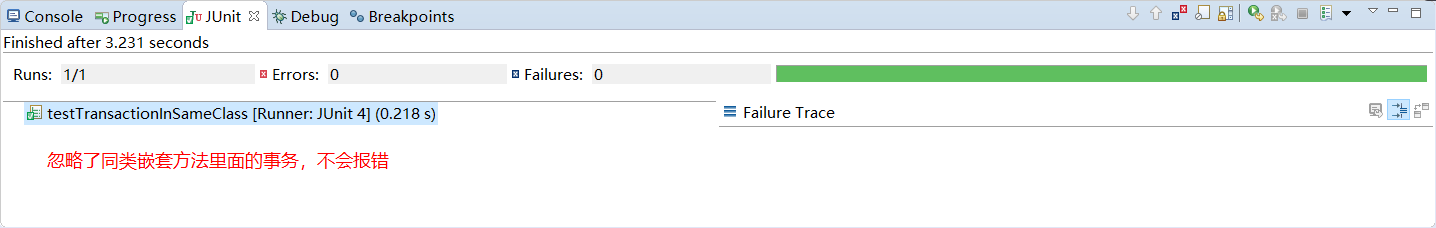
NESTED：如果当前有事务，则在当前事务内部嵌套一个事务，内部事务的回滚不影响当前事务。如果当前没有事务，就相当于REQUIRED

如果使用@Transaction方法里嵌套调用的是同一个类的方法，spring代理会忽略嵌套方法的@Transaction配置。

测试1：当前事务类型required的方法里面嵌套一个同类并且事务类型是never的方法，不会报错

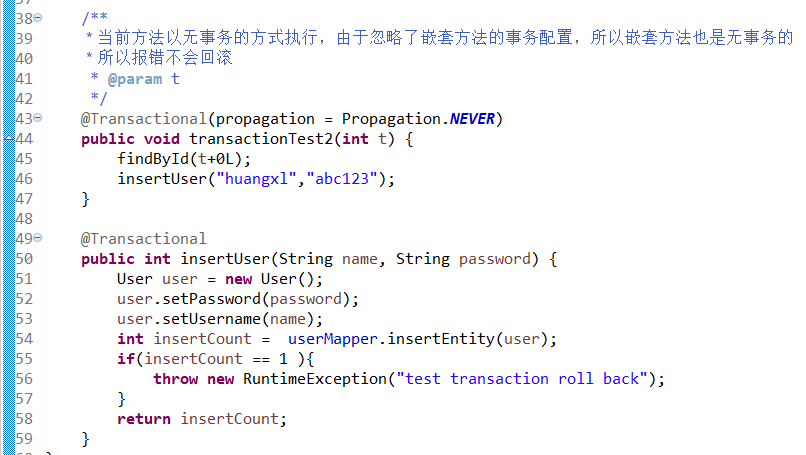


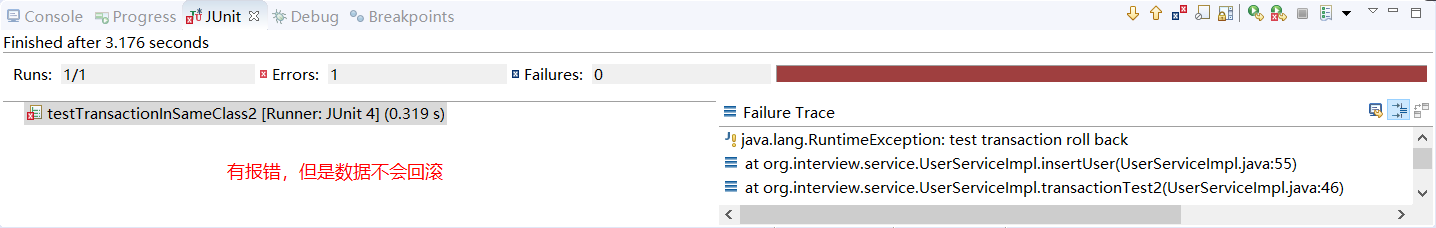


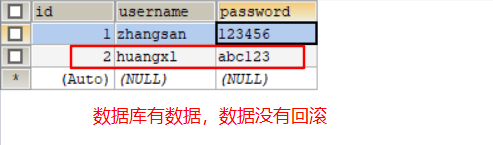


测试2：当前事务类型never的方法里面嵌套一个同类并且事务类型是required的方法，设置程序报错，数据有插入，以无事务的方式执行，不会回滚





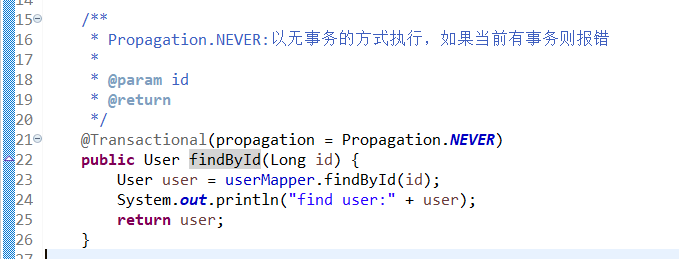


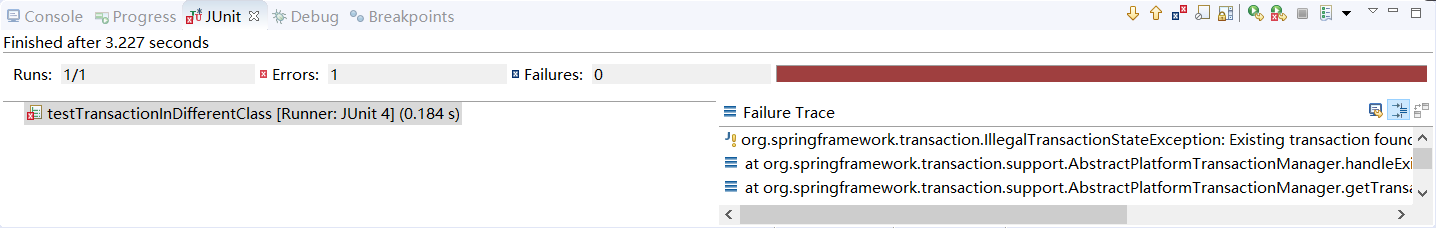


Propagation.NEVER的测试：当前事务类型为默认required的方法中嵌套一个不同的类的事务类型为never的方法，事务生效，报错





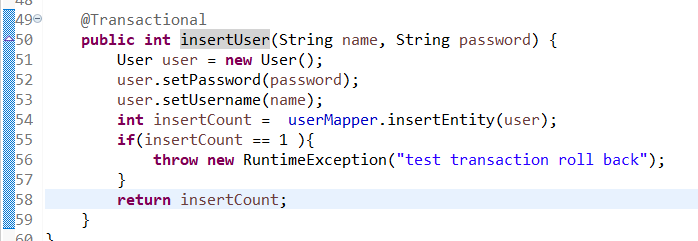


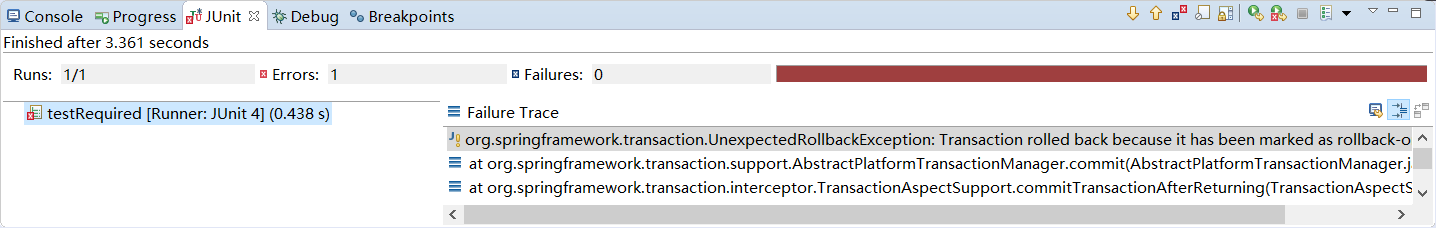


Propagation.REQUIRED的测试：在事务类型为required的方法里面嵌套一个不同类的事务类型为required的方法。如果嵌套的方法报错，则不能直接捕获，因为嵌套方法已经被设置成了rollback-only了，再次提交的话就会冲突，抛出异常：org.springframework.transaction.UnexpectedRollbackException: Transaction rolled back because it has been marked as rollback-only。

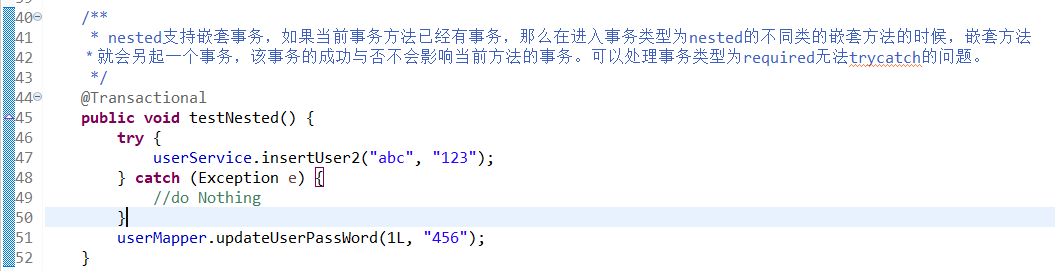
required继承了当前的事务，两个事务合并在一起了，同时成功或者同时失败，不存在第三种情况





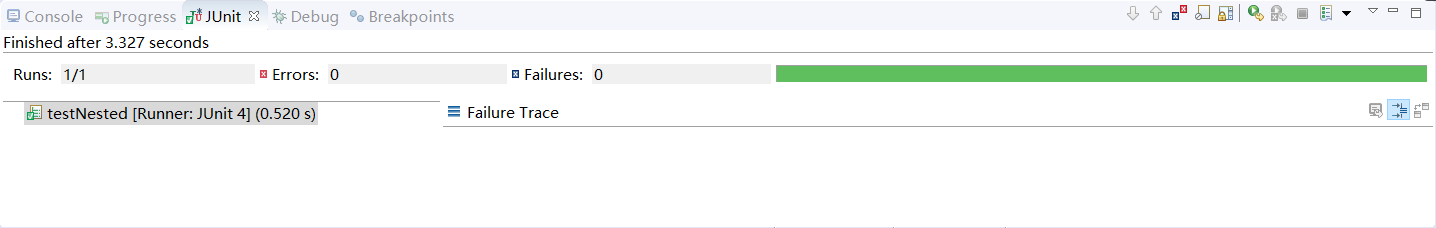


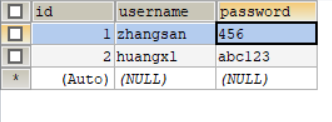
Propagation.NESTED的测试：如果当前有事务，则在当前事务内部嵌套一个事务，内部事务的回滚不影响当前事务。如果当前没有事务，就相当于REQUIRED。







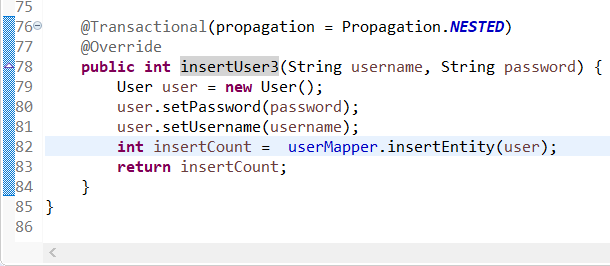


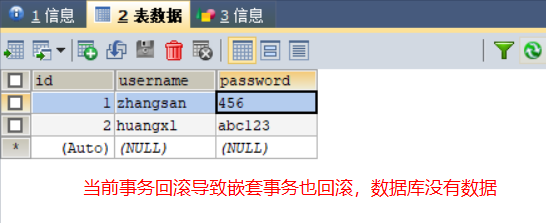


NESTED子事务回滚不会影响当前事务的提交(catch回滚异常的情况下)，但是当前事务回滚会回滚子事务。也就是说只有当前事务提交成功了，子事务才会提交成功。



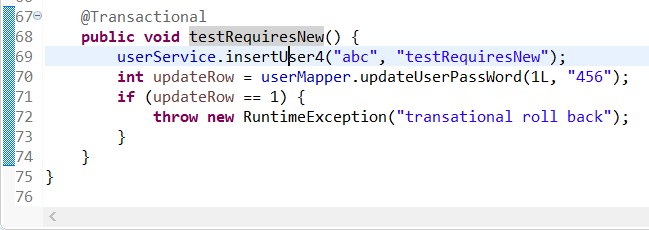






Propagation. REQUIRES\_NEW的测试：REQUIRED\_NEW：新建一个事务，同时将当前事务挂起





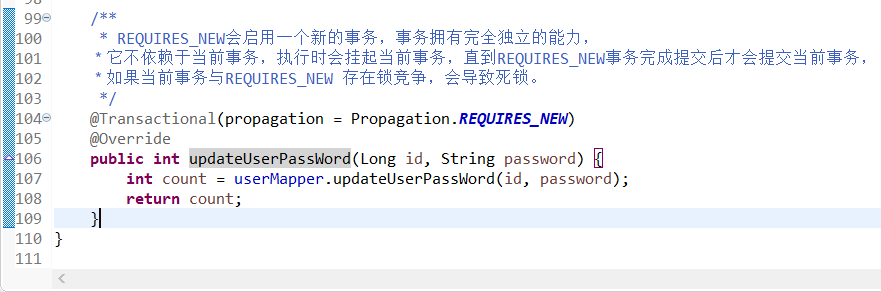


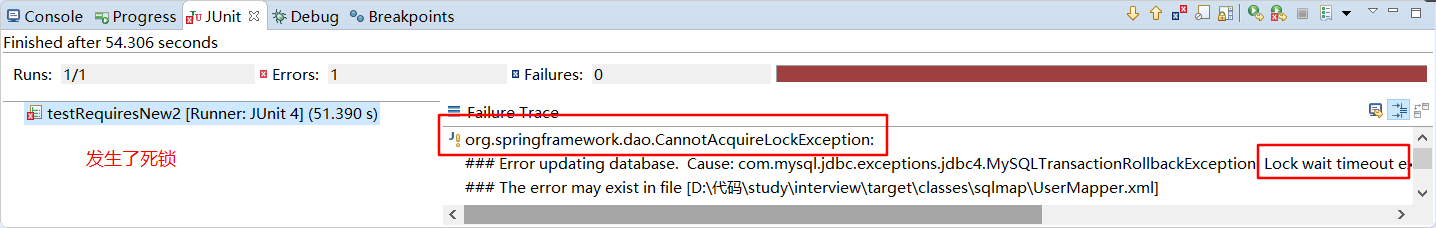


REQUIRES\_NEW会启用一个新的事务，事务拥有完全独立的能力，它不依赖于当前事务，执行时会挂起当前事务，直到REQUIRES\_NEW事务完成提交后才会提交当前事务，如果当前事务与REQUIRES\_NEW 存在锁竞争，会导致死锁。





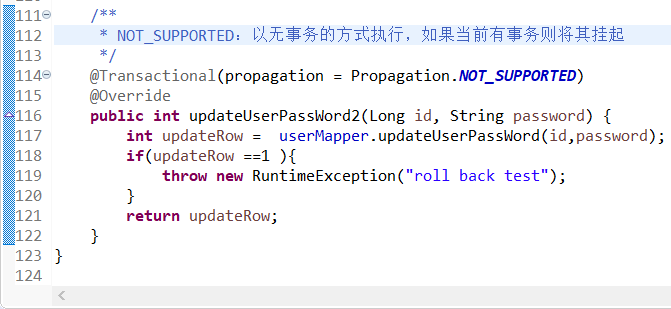


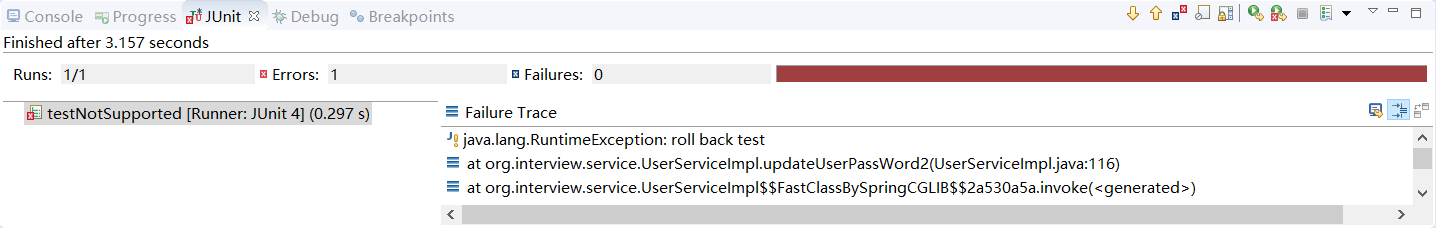


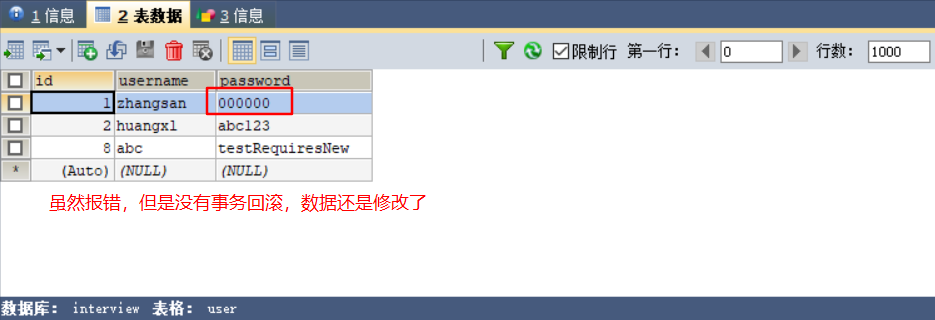
NOT\_SUPPORTED：以无事务的方式执行，如果当前有事务则将其挂起





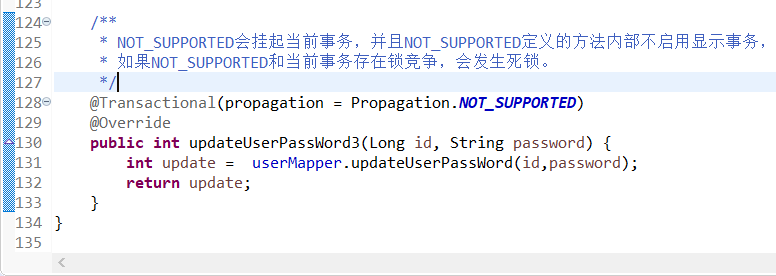




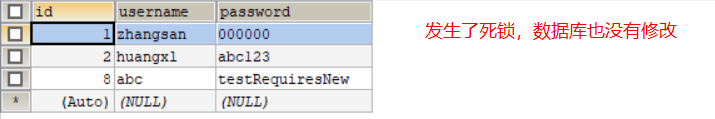


NOT\_SUPPORTED会挂起当前事务，并且NOT\_SUPPORTED定义的方法内部不启用显示事务，如果NOT\_SUPPORTED和当前事务存在锁竞争，会发生死锁。



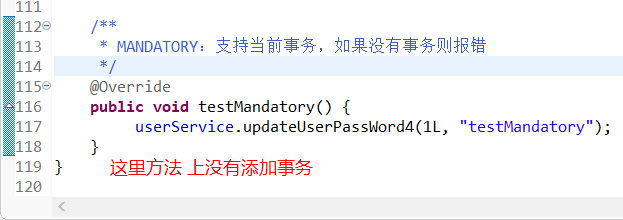






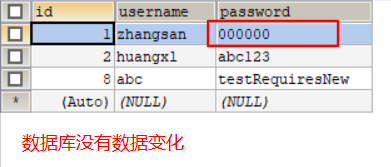
MANDATORY：支持当前事务，如果没有事务则报错





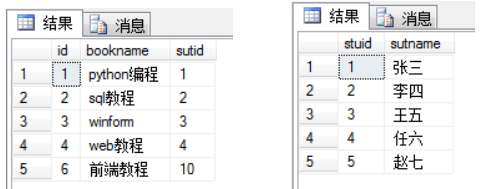






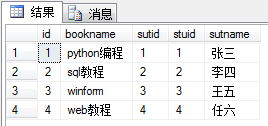
#### 内连接、左联接、右连接、全连接、交叉连接

数据库数据



内连接

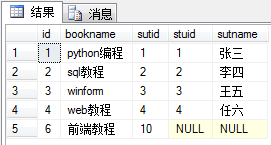
select \* from book as a inner join stu as b on a.sutid = b.stuid



外连接：左联接、右连接、全连接

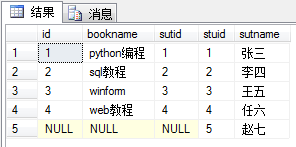
左联接：是以左表为基准，将a.stuid = b.stuid的数据进行连接，然后将左表没有的对应项显示，右表的列为NULL

select \* from book as a left join stu as b on a.sutid = b.stuid



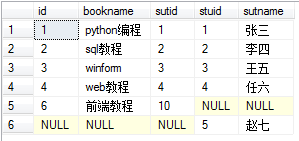
右连接：是以右表为基准，将a.stuid = b.stuid的数据进行连接，然以将右表没有的对应项显示，左表的列为NULL

select \* from book as a right join stu as b on a.sutid = b.stuid



全连接：完整外部联接返回左表和右表中的所有行。当某行在另一个表中没有匹配行时，则另一个表的选择列表列包含空值。如果表之间有匹配行，则整个结果集行包含基表的数据值。

select \* from book as a full outer join stu as b on a.sutid = b.stuid



交叉连接

交叉连接：交叉联接返回左表中的所有行，左表中的每一行与右表中的所有行组合。交叉联接也称作笛卡尔积。

select \* from book as a cross join stu as b order by a.id



#### 依赖注入和控制反转的理解

Ioc—Inversion of Control，即“控制反转”，不是什么技术，而是一种设计思想。在Java开发中，Ioc意味着将你设计好的对象交给容器控制，而不是传统的在你的对象内部直接控制。

DI—Dependency Injection，即“依赖注入”：由容器动态的将某个依赖关系注入到组件之中。autowired注入对象

#### mysql索引

索引的优点和缺点：

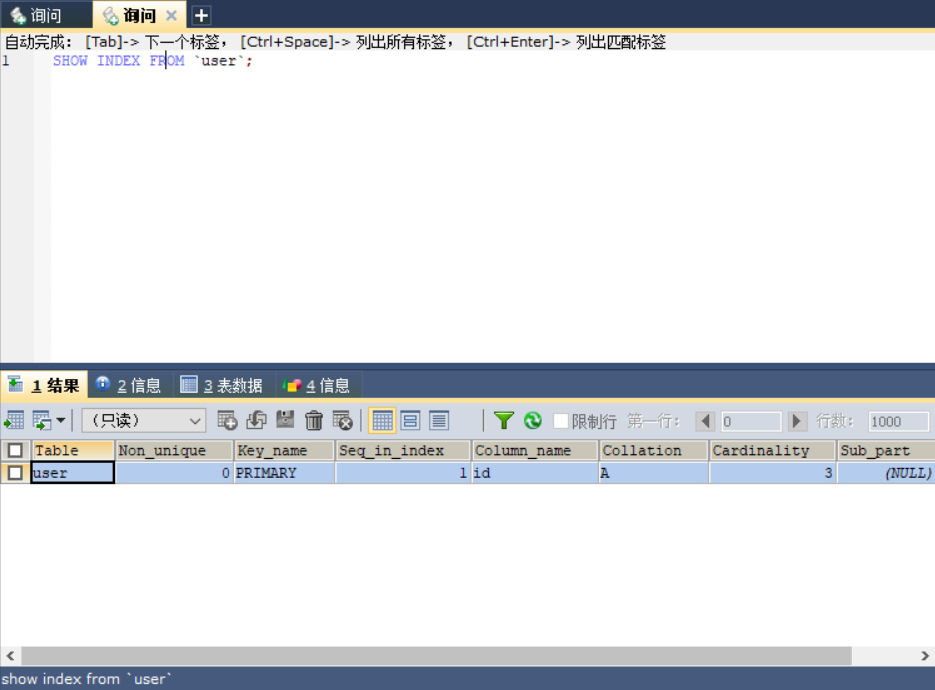
优点：大大提高了查询速度

缺点：1.降低更新表的速度2.建立索引会占用磁盘空间的索引文件。

查看一个表中是否已经创建索引

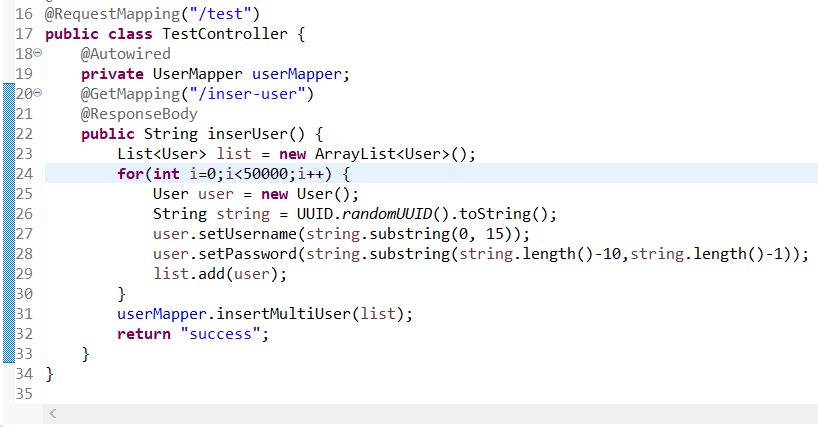
show index from 表名;

SHOW INDEX FROM `user`;

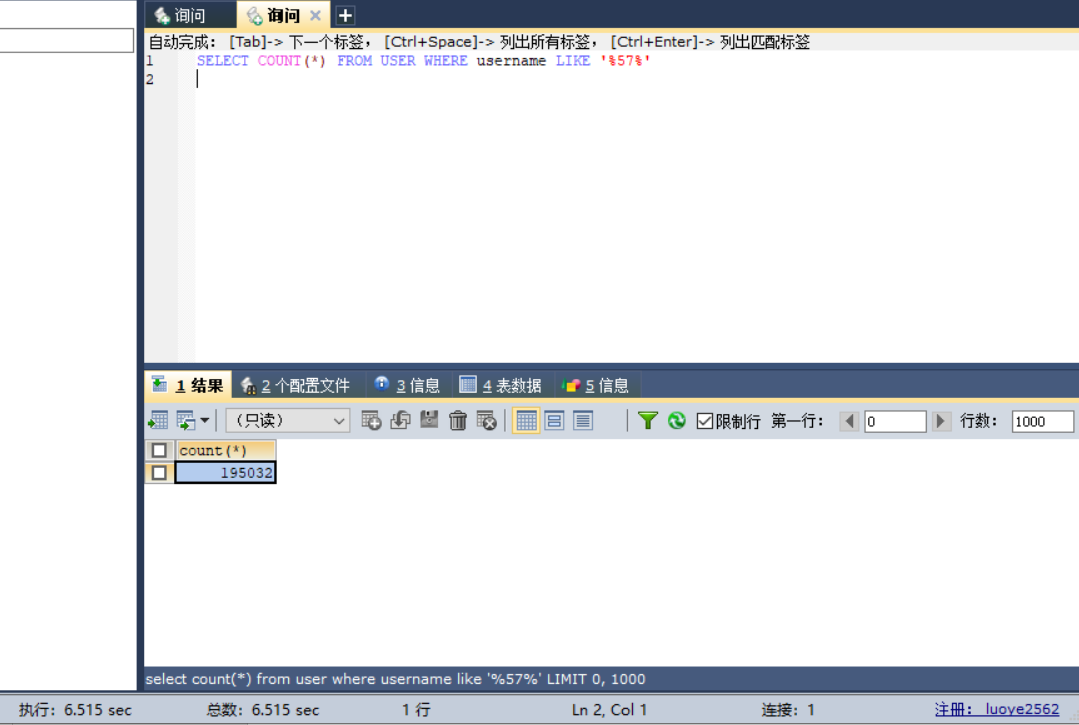


创建索引（如果指定字段是字符串，需要指定长度，建议长度与定义字段时的长度一致，字段类型如果不是字符串，可以不填写长度部分）

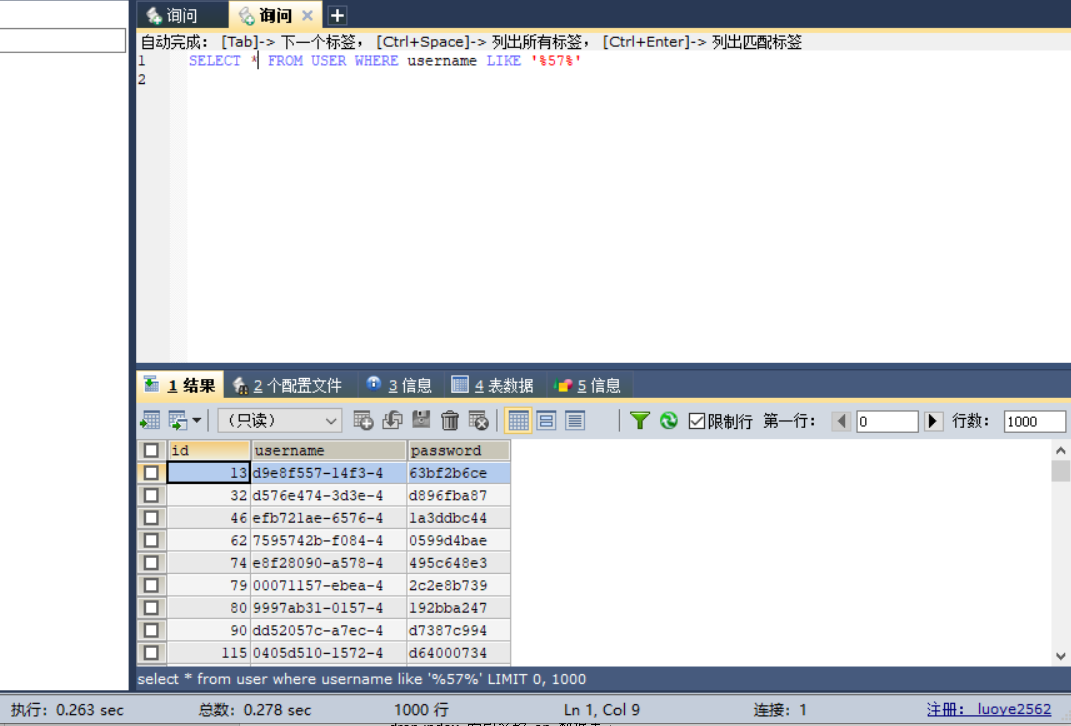
插入多组数据



select count(\*) from user where username like '%57%'，耗时6.515秒

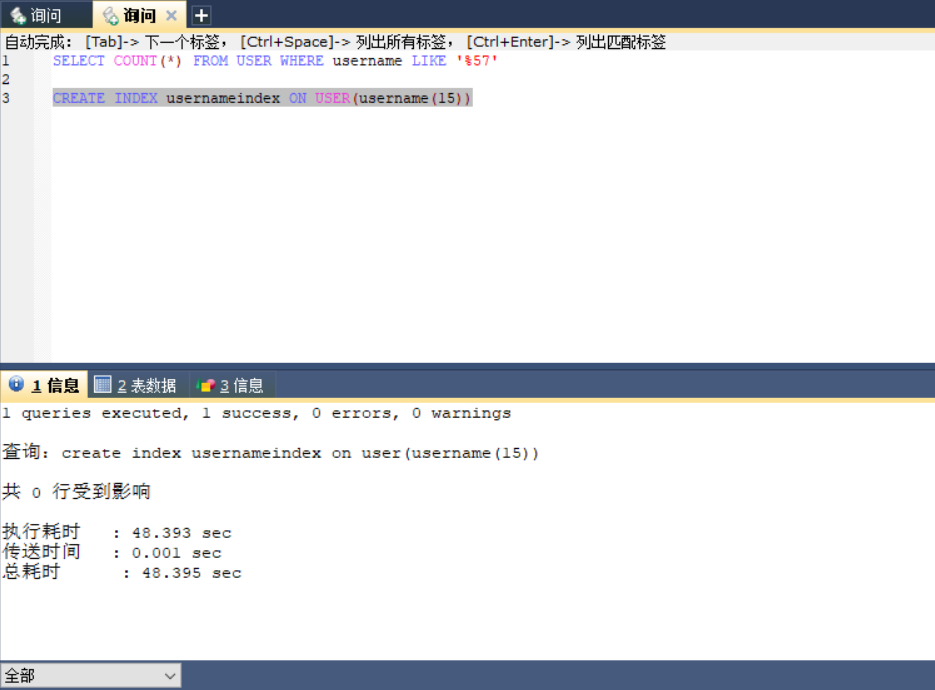


SELECT \* FROM USER WHERE username LIKE '%57%'，耗时0.278秒



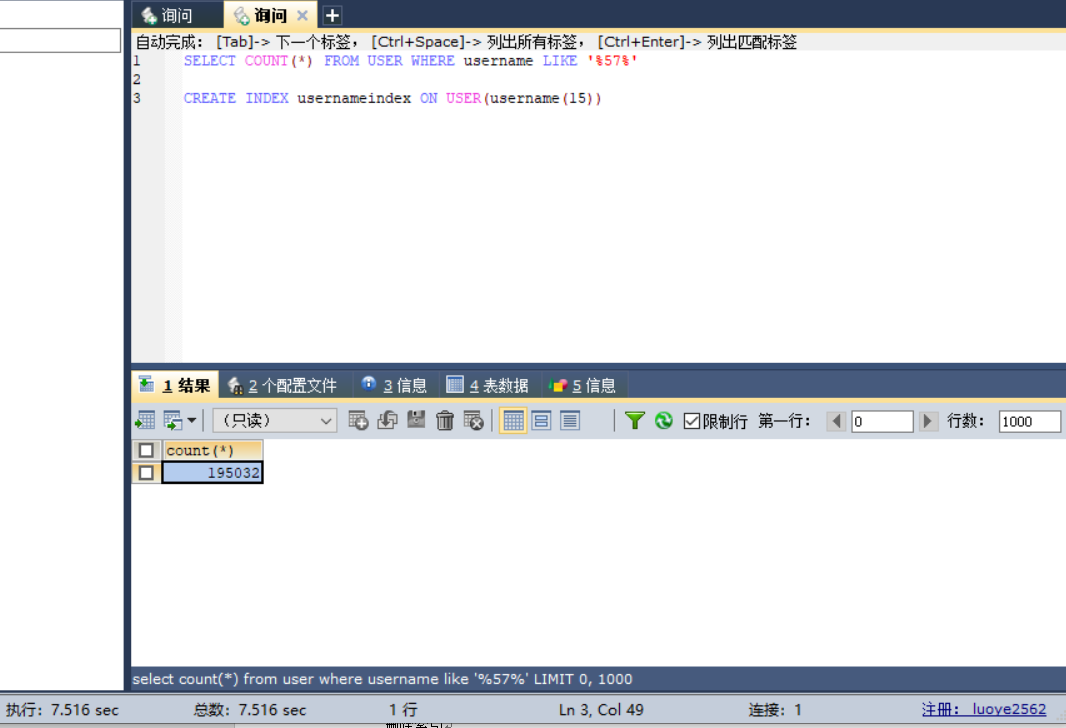
create index 索引名 on 数据表（字段名称长度）

create index usernameindex on user(username(15))



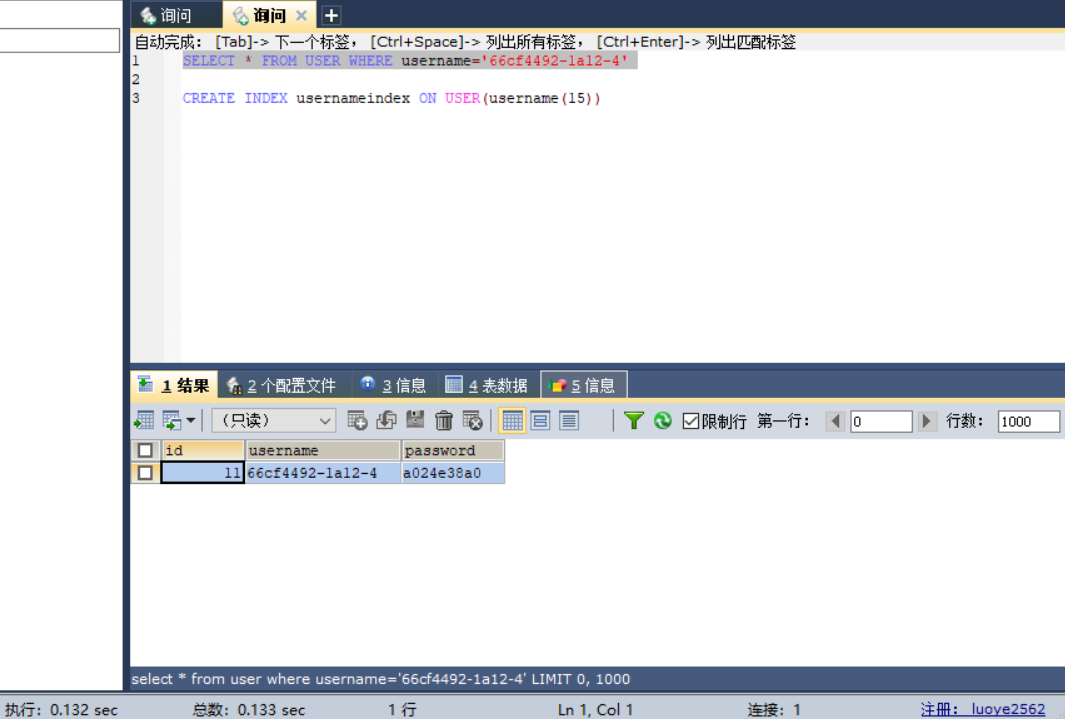
SELECT COUNT(\*) FROM USER WHERE username LIKE '%57%'

创建索引后进行模糊查询，发现性能还慢了一点



直接查询全部，性能挺快的

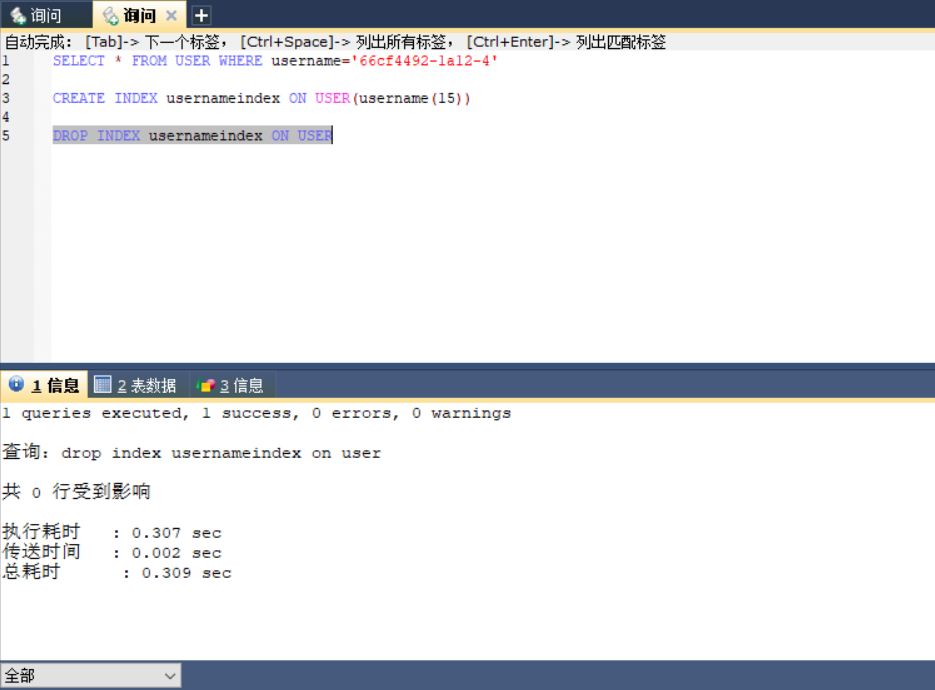
SELECT \* FROM USER WHERE username='66cf4492-1a12-4'



删除索引

drop index 索引名称 on 数据表

DROP INDEX usernameindex ON USER



删掉索引后重新查询刚刚那串

select \* from user where username='66cf4492-1a12-4'

发现耗时居然达到了7.345秒。所以索引确实很能提高效率，但是模糊查询的话没有用



#### 冒泡排序

传统的冒泡排序的特点是对n个数进行n-1次排序，每次排序都将最大或者最小的数据放到最后一位，下一轮的时候忽略最后一位，将前面的数据继续进行排序。选出最大或者最小的数据的方法是将全部的数据从第一个开始到第n-1个都和后面的数据进行比较，比如大的就放到后面，遇到大的就会放到后面，经过完全的遍历，最大的肯定是最后面的，但是这一次并不能将全部的数据都排序了，因为从前往后排的话只知道待比较的两个数据哪个最大，不知道小的那个数据和前面的比较后的大小关系。比如3,5,1。这里的1并不能知道3的存在，也没有进行比较。

通过筛选出最大或者最小的数据放到末尾，下一次从剩下的数据里面筛选出最大的数据放到倒数第二，再下一次以此类推，逐步将数据进行排序。

这里可以做一个优化点，当某一轮排序中没有发现需要变换位置的数据，则说明全程都是没有排序的，可以不用继续排序了

