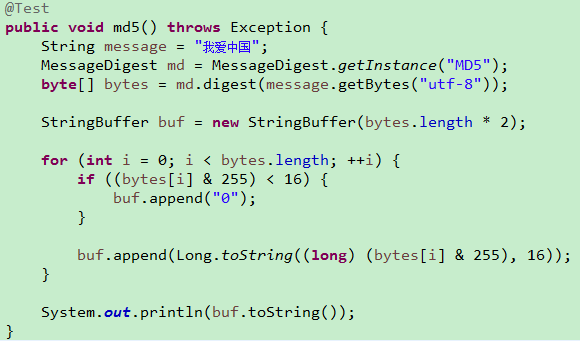
# 问题收集

## ****1.Java MessageDigest及MD5加密实现****

**例：字符串MD5加密**

* java.security.MessageDigestSpi
* java.security.MessageDigest



|  |
| --- |
| @Test  **public** **void** md5() **throws** Exception {  String message = "我爱中国";  MessageDigest md = MessageDigest.*getInstance*("MD5");  **byte**[] bytes = md.digest(message.getBytes("utf-8"));    StringBuffer buf = **new** StringBuffer(bytes.length \* 2);  **for** (**int** i = 0; i < bytes.length; ++i) {  **if** ((bytes[i] & 255) < 16) {  buf.append("0");  }  buf.append(Long.*toString*((**long**) (bytes[i] & 255), 16));  }  System.***out***.println(buf.toString());  } |

## **2.Java的位运算符详解实例——与（&）、非（~）、或（|）、异或（^）**

 位运算符主要针对二进制，它包括了：“与”、“非”、“或”、“异或”。从表面上看似乎有点像逻辑运算符，但逻辑运算符是针对两个关系运算符来进行逻辑运算，而位运算符主要针对两个二进制数的位进行逻辑运算。下面详细介绍每个位运算符。

 1．与运算符  
与运算符用符号“&”表示，其使用规律如下：  
两个操作数中位都为1，结果才为1，否则结果为0，例如下面的程序段。  
public class data13  
{  
public static void main(String[] args)  
{  
int a=129;  
int b=128;  
System.out.println("a 和b 与的结果是："+(a&b));  
}  
}  
运行结果  
a 和b 与的结果是：128  
下面分析这个程序：  
“a”的值是129，转换成二进制就是10000001，而“b”的值是128，转换成二进制就是10000000。根据与运算符的运算规律，只有两个位都是1，结果才是1，可以知道结果就是10000000，即128。  
2．或运算符  
或运算符用符号“|”表示，其运算规律如下：  
两个位只要有一个为1，那么结果就是1，否则就为0，下面看一个简单的例子。  
public class data14  
{  
public static void main(String[] args)  
{  
int a=129;  
int b=128;  
System.out.println("a 和b 或的结果是："+(a|b));  
}  
}  
运行结果  
a 和b 或的结果是：129  
下面分析这个程序段：  
a 的值是129，转换成二进制就是10000001，而b 的值是128，转换成二进制就是10000000，根据或运算符的运算规律，只有两个位有一个是1，结果才是1，可以知道结果就是10000001，即129。  
3．非运算符  
非运算符用符号“~”表示，其运算规律如下：

如果位为0，结果是1，如果位为1，结果是0，下面看一个简单例子。  
public class data15  
{  
public static void main(String[] args)  
{  
int a=2;  
System.out.println("a 非的结果是："+(~a));  
}  
}  
4．异或运算符  
异或运算符是用符号“^”表示的，其运算规律是：  
两个操作数的位中，相同则结果为0，不同则结果为1。下面看一个简单的例子。  
public class data16  
{  
public static void main(String[] args)  
{  
int a=15;  
int b=2;  
System.out.println("a 与 b 异或的结果是："+(a^b));  
}  
}  
运行结果  
a 与 b 异或的结果是：13  
分析上面的程序段：a 的值是15，转换成二进制为1111，而b 的值是2，转换成二进制为0010，根据异或的运算规律，可以得出其结果为1101 即13。

## **3.前置机及其作用**

答案

## **4.Nginx代理服务器**

答案

http://blog.csdn.net/u014749862/article/details/50522276

## **5.Nginx+Tomcat负载、集群部署**

**例：工会EBS系统部署**

在工会的前置机上部署Nginx反向代理服务器，后端集群部署多台Tomcat服务器。Nginx代理服务器分主从两台，通过keeplived监测代理服务器是否宕机，如果宕机则切换到从服务器上。最终可实现暴露给用户一个访问地址，并实现负载均衡，对高并发提供好的支持。

## **6.数据库锁机制**

Csdn博客：http://blog.csdn.net/samjustin1/article/details/52210125

**概述：**

1）概念：

数据库是一个多用户使用的共享资源。当多个用户并发地存取数据时，在数据库中就会产生多个事务同时存取同一数据的情况。若对并发操作不加控制就可能会读取和存储不正确的数据，破坏数据库的一致性。

加锁是实现数据库并发控制的一个非常重要的技术。当事务在对某个数据对象进行操作前，先向系统发出请求，对其加锁。加锁后事务就对该数据对象有了一定的控制，在该事务释放锁之前，其他的事务不能对此数据对象进行更新操作。

（2）锁的分类：

共享（S)锁：多个事务可封锁一个共享页；任何事务都不能修改该页； 通常是该页被读取完毕，S锁立即被释放。

排它（X)锁：仅允许一个事务封锁此页；其他任何事务必须等到X锁被释放才能对该页进行访问；X锁一直到事务结束才能被释放。

更新（U)锁：更新锁在修改操作的初始化阶段用来锁定可能要被修改的资源，这样可以避免使用共享锁造成的死锁现象。因为使用共享锁时，修改数据的操作分为两步，首先获得一个共享锁，读取数据，然后将共享锁升级为排它锁，然后再执行修改操作。这样如果同时有两个或多个事务同时对一个事务申请了共享锁，在修改数据的时候，这些事务都要将共享锁升级为排它锁。这时，这些事务都不会释放共享锁而是一直等待对方释放，这样就造成了死锁。如果一个数据在修改前直接申请更新锁，在数据修改的时候再升级为排它锁，就可以避免死锁。

（3）锁的粒度：

在sql server2000中锁是具有粒度的，即可以对不同的资源加锁。锁定在较小的粒度的资源（例如行）上可以增加系统的并发量但需要较大的系统开销，从而也会影响系统的性能，因为锁定的粒度较小则操作可能产生的锁的数量会增加；锁定在较大的粒度（例如表）就并发而言是相当昂贵的，因为锁定整个表限制了其它事务对表中任意部分进行访问，但要求的开销较低，因为需要维护的锁较少，所以在这里是一种互相制约的关系。

Sql server2000中锁定的粒度包括 行、页、扩展盘区、表、库等资源。

**结合实例的分析：**

**1 前言**

数据库大并发操作要考虑死锁和锁的性能问题。看到网上大多语焉不详(尤其更新锁），所以这里做个简明解释，为下面描述方便，这里用T1代表一个数据库执行请求，T2代表另一个请求，也可以理解为T1为一个线程，T2 为另一个线程。T3,T4以此类推。下面以SQL Server(2005)为例。

**2 锁的种类**

1. 共享锁(Shared lock)。
2. 例1：
3. ----------------------------------------
4. T1: select \* from table (请想象它需要执行1个小时之久，后面的sql语句请都这么想象）
5. T2: update table set column1='hello'
6. 过程：
7. T1运行 （加共享锁)
8. T2运行
9. If T1 还没执行完
10. T2等......
11. else
12. 锁被释放
13. T2执行
14. endif
15. T2之所以要等，是因为T2在执行update前，试图对table表加一个排他锁，
16. 而数据库规定同一资源上不能同时共存共享锁和排他锁。所以T2必须等T1
17. 执行完，释放了共享锁，才能加上排他锁，然后才能开始执行update语句。
18. 例2：
19. ----------------------------------------
20. T1: select \* from table
21. T2: select \* from table
22. 这里T2不用等待T1执行完，而是可以马上执行。
23. 分析：
24. T1运行，则table被加锁，比如叫lockA
25. T2运行，再对table加一个共享锁，比如叫lockB。
26. 两个锁是可以同时存在于同一资源上的（比如同一个表上）。这被称为共
27. 享锁与共享锁兼容。这意味着共享锁不阻止其它session同时读资源，但阻
28. 止其它session update
29. 例3：
30. ----------------------------------------
31. T1: select \* from table
32. T2: select \* from table
33. T3: update table set column1='hello'
34. 这次，T2不用等T1运行完就能运行，T3却要等T1和T2都运行完才能运行。
35. 因为T3必须等T1和T2的共享锁全部释放才能进行加排他锁然后执行update
36. 操作。
37. 例4：（死锁的发生）
38. ----------------------------------------
39. T1:
40. begin tran
41. select \* from table (holdlock) (holdlock意思是加共享锁，直到事物结束才释放)
42. update table set column1='hello'
43. T2:
44. begin tran
45. select \* from table(holdlock)
46. update table set column1='world'
47. 假设T1和T2同时达到select，T1对table加共享锁，T2也对加共享锁，当
48. T1的select执行完，准备执行update时，根据锁机制，T1的共享锁需要升
49. 级到排他锁才能执行接下来的update.在升级排他锁前，必须等table上的
50. 其它共享锁释放，但因为holdlock这样的共享锁只有等事务结束后才释放，
51. 所以因为T2的共享锁不释放而导致T1等(等T2释放共享锁，自己好升级成排
52. 他锁），同理，也因为T1的共享锁不释放而导致T2等。死锁产生了。
53. 例5：
54. ----------------------------------------
55. T1:
56. begin tran
57. update table set column1='hello' where id=10
58. T2:
59. begin tran
60. update table set column1='world' where id=20
61. 这种语句虽然最为常见，很多人觉得它有机会产生死锁，但实际上要看情
62. 况，如果id是主键上面有索引，那么T1会一下子找到该条记录(id=10的记
63. 录），然后对该条记录加排他锁，T2，同样，一下子通过索引定位到记录，
64. 然后对id=20的记录加排他锁，这样T1和T2各更新各的，互不影响。T2也不
65. 需要等。
66. 但如果id是普通的一列，没有索引。那么当T1对id=10这一行加排他锁后，
67. T2为了找到id=20，需要对全表扫描，那么就会预先对表加上共享锁或更新
68. 锁或排他锁(依赖于数据库执行策略和方式，比如第一次执行和第二次执行
69. 数据库执行策略就会不同）。但因为T1已经为一条记录加了排他锁，导致
70. T2的全表扫描进行不下去，就导致T2等待。
71. 死锁怎么解决呢？一种办法是，如下：
72. 例6：
73. ----------------------------------------
74. T1:
75. begin tran
76. select \* from table(xlock) (xlock意思是直接对表加排他锁)
77. update table set column1='hello'
78. T2:
79. begin tran
80. select \* from table(xlock)
81. update table set column1='world'
82. 这样，当T1的select 执行时，直接对表加上了排他锁，T2在执行select时，就需要等T1事物完全执行完才能执行。排除了死锁发生。
83. 但当第三个user过来想执行一个查询语句时，也因为排他锁的存在而不得不等待，第四个、第五个user也会因此而等待。在大并发
84. 情况下，让大家等待显得性能就太友好了，所以，这里引入了更新锁。
85. 更新锁(Update lock)
86. 为解决死锁，引入更新锁。
87. 例7：
88. ----------------------------------------
89. T1:
90. begin tran
91. select \* from table(updlock) (加更新锁)
92. update table set column1='hello'
93. T2:
94. begin tran
95. select \* from table(updlock)
96. update table set column1='world'
97. 更新锁的意思是：“我现在只想读，你们别人也可以读，但我将来可能会做更新操作，我已经获取了从共享锁（用来读）到排他锁
98. （用来更新）的资格”。一个事物只能有一个更新锁获此资格。
99. T1执行select，加更新锁。
100. T2运行，准备加更新锁，但发现已经有一个更新锁在那儿了，只好等。
101. 当后来有user3、user4...需要查询table表中的数据时，并不会因为T1的select在执行就被阻塞，照样能查询，相比起例6，这提高
102. 了效率。
103. 例8:
104. ----------------------------------------
105. T1: select \* from table(updlock) (加更新锁）
106. T2: select \* from table(updlock) (等待，直到T1释放更新锁，因为同一时间不能在同一资源上有两个更新锁）
107. T3: select \* from table (加共享锁，但不用等updlock释放，就可以读）
108. 这个例子是说明：共享锁和更新锁可以同时在同一个资源上。这被称为共享锁和更新锁是兼容的。
109. 例9:
110. ----------------------------------------
111. T1:
112. begin
113. select \* from table(updlock) (加更新锁）
114. update table set column1='hello' (重点：这里T1做update时，不需要等T2释放什么，而是直接把更新锁升级为排他锁，然后执行update)
115. T2:
116. begin
117. select \* from table (T1加的更新锁不影响T2读取）
118. update table set column1='world' (T2的update需要等T1的update做完才能执行)
119. 我们以这个例子来加深更新锁的理解，
120. 第一种情况：T1先达，T2紧接到达；在这种情况中，T1先对表加更新锁，T2对表加共享锁，假设T2的select先执行完，准备执行update，
121. 发现已有更新锁存在，T2等。T1执行这时才执行完select，准备执行update，更新锁升级为排他锁，然后执行update，执行完成，事务
122. 结束，释放锁，T2才轮到执行update。
123. 第二种情况：T2先达，T1紧接达；在这种情况，T2先对表加共享锁，T1达后，T1对表加更新锁，假设T2 select先结束，准备
124. update，发现已有更新锁，则等待，后面步骤就跟第一种情况一样了。
125. 这个例子是说明：排他锁与更新锁是不兼容的，它们不能同时加在同一子资源上。
126. 排他锁（独占锁，Exclusive Locks)
127. 这个简单，即其它事务既不能读，又不能改排他锁锁定的资源。
128. 例10
129. T1: update table set column1='hello' where id<1000
130. T2: update table set column1='world' where id>1000
131. 假设T1先达，T2随后至，这个过程中T1会对id<1000的记录施加排他锁.但不会阻塞T2的update。
132. 例11 (假设id都是自增长且连续的）
133. T1: update table set column1='hello' where id<1000
134. T2: update table set column1='world' where id>900
135. 如同例10，T1先达，T2立刻也到，T1加的排他锁会阻塞T2的update.
136. 意向锁(Intent Locks)
137. 意向锁就是说在屋（比如代表一个表）门口设置一个标识，说明屋子里有人（比如代表某些记录）被锁住了。另一个人想知道屋子
138. 里是否有人被锁，不用进屋子里一个一个的去查，直接看门口标识就行了。
139. 当一个表中的某一行被加上排他锁后，该表就不能再被加表锁。数据库程序如何知道该表不能被加表锁？一种方式是逐条的判断该
140. 表的每一条记录是否已经有排他锁，另一种方式是直接在表这一层级检查表本身是否有意向锁，不需要逐条判断。显然后者效率高。
141. 例12：
142. ----------------------------------------
143. T1: begin tran
144. select \* from table (xlock) where id=10 --意思是对id=10这一行强加排他锁
145. T2: begin tran
146. select \* from table (tablock) --意思是要加表级锁
148. 假设T1先执行，T2后执行，T2执行时，欲加表锁，为判断是否可以加表锁，数据库系统要逐条判断table表每行记录是否已有排他锁，
149. 如果发现其中一行已经有排他锁了，就不允许再加表锁了。只是这样逐条判断效率太低了。
150. 实际上，数据库系统不是这样工作的。当T1的select执行时，系统对表table的id=10的这一行加了排他锁，还同时悄悄的对整个表
151. 加了意向排他锁(IX)，当T2执行表锁时，只需要看到这个表已经有意向排他锁存在，就直接等待，而不需要逐条检查资源了。
152. 例13：
153. ----------------------------------------
154. T1: begin tran
155. update table set column1='hello' where id=1
156. T2: begin tran
157. update table set column1='world' where id=1
158. 这个例子和上面的例子实际效果相同，T1执行，系统对table同时对行家排他锁、对页加意向排他锁、对表加意向排他锁。
159. 计划锁(Schema Locks)
160. 例14：
161. ----------------------------------------
162. alter table .... (加schema locks，称之为Schema modification (Sch-M) locks
163. DDL语句都会加Sch-M锁
164. 该锁不允许任何其它session连接该表。连都连不了这个表了，当然更不用说想对该表执行什么sql语句了。
165. 例15:
166. ----------------------------------------
167. 用jdbc向数据库发送了一条新的sql语句，数据库要先对之进行编译，在编译期间，也会加锁，称之为：Schema stability (Sch-S) locks
168. select \* from tableA
169. 编译这条语句过程中，其它session可以对表tableA做任何操作(update,delete，加排他锁等等），但不能做DDL(比如alter table)操作。
170. Bulk Update Locks 主要在批量导数据时用（比如用类似于oracle中的imp/exp的bcp命令）。不难理解，程序员往往也不需要关心，不赘述了。

**3 何时加锁？**

如何加锁，何时加锁，加什么锁，你可以通过hint手工强行指定，但大多是数据库系统自动决定的。这就是为什么我们可以不懂锁也可

以高高兴兴的写SQL。

例15:

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

T1: begin tran

update table set column1='hello' where id=1

T2: SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED -- 事物隔离级别为允许脏读

go

select \* from table where id=1

这里，T2的select可以查出结果。如果事物隔离级别不设为脏读，则T2会等T1事物执行完才能读出结果。

数据库如何自动加锁的？

1) T1执行，数据库自动加排他锁

2) T2执行，数据库发现事物隔离级别允许脏读，便不加共享锁。不加共享锁，则不会与已有的排他锁冲突，所以可以脏读。

例16:

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

T1: begin tran

update table set column1='hello' where id=1

T2: select \* from table where id=1 --为指定隔离级别，则使用系统默认隔离级别，它不允许脏读

如果事物级别不设为脏读，则：

1) T1执行，数据库自动加排他锁

2) T2执行，数据库发现事物隔离级别不允许脏读，便准备为此次select过程加共享锁，但发现加不上，因为已经有排他锁了，所以就

等啊等。直到T1执行完，释放了排他锁，T2才加上了共享锁，然后开始读....

**4 锁的粒度**

锁的粒度就是指锁的生效范围，就是说是行锁，还是页锁，还是整表锁. 锁的粒度同样既可以由数据库自动管理，也可以通过手工指定hint来管理。

例17：

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

T1: select \* from table (paglock)

T2: update table set column1='hello' where id>10

T1执行时，会先对第一页加锁，读完第一页后，释放锁，再对第二页加锁，依此类推。假设前10行记录恰好是一页(当然，一般不可能

一页只有10行记录），那么T1执行到第一页查询时，并不会阻塞T2的更新。

例18：

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

T1: select \* from table (rowlock)

T2: update table set column1='hello' where id=10

T1执行时，对每行加共享锁，读取，然后释放，再对下一行加锁;T2执行时，会对id=10的那一行试图加锁，只要该行没有被T1加上行锁，

T2就可以顺利执行update操作。

例19：

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

T1: select \* from table (tablock)

T2: update table set column1='hello' where id = 10

T1执行，对整个表加共享锁. T1必须完全查询完，T2才可以允许加锁，并开始更新。

以上3例是手工指定锁的粒度，也可以通过设定事物隔离级别，让数据库自动设置锁的粒度。不同的事物隔离级别，数据库会有不同的

加锁策略（比如加什么类型的锁，加什么粒度的锁）。具体请查联机手册。

**5 锁与事物隔离级别的优先级**

手工指定的锁优先，

例20:

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

T1: GO

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE

GO

BEGIN TRANSACTION

SELECT \* FROM table (NOLOCK)

GO

T2: update table set column1='hello' where id=10

T1是事物隔离级别为最高级，串行锁，数据库系统本应对后面的select语句自动加表级锁，但因为手工指定了NOLOCK，所以该select

语句不会加任何锁，所以T2也就不会有任何阻塞。

**6 数据库的其它重要Hint以及它们的区别**

1) holdlock 对表加共享锁，且事物不完成，共享锁不释放。

2) tablock 对表加共享锁，只要statement不完成，共享锁不释放。

与holdlock区别，见下例：

例21

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

T1:

begin tran

select \* from table (tablock)

T2:

begin tran

update table set column1='hello' where id = 10

T1执行完select，就会释放共享锁，然后T2就可以执行update. 此之谓tablock. 下面我们看holdlock

例22

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

T1:

begin tran

select \* from table (holdlock)

T2:

begin tran

update table set column1='hello' where id = 10

T1执行完select，共享锁仍然不会释放，仍然会被hold(持有），T2也因此必须等待而不能update. 当T1最后执行了commit或

rollback说明这一个事物结束了，T2才取得执行权。

3) TABLOCKX 对表加排他锁

例23：

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

T1: select \* from table(tablockx) (强行加排他锁）

其它session就无法对这个表进行读和更新了，除非T1执行完了，就会自动释放排他锁。

例24：

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

T1: begin tran

select \* from table(tablockx)

这次，单单select执行完还不行，必须整个事物完成（执行了commit或rollback后）才会释放排他锁。

4) xlock 加排他锁

那它跟tablockx有何区别呢？

它可以这样用，

例25:

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

select \* from table(xlock paglock) 对page加排他锁

而TABLELOCX不能这么用。

xlock还可这么用：select \* from table(xlock tablock) 效果等同于select \* from table(tablockx)

**7 锁的超时等待**

例26

SET LOCK\_TIMEOUT 4000 用来设置锁等待时间，单位是毫秒，4000意味着等待

4秒可以用select @@LOCK\_TIMEOUT查看当前session的锁超时设置。-1 意味着

永远等待。

T1: begin tran

udpate table set column1='hello' where id = 10

T2: set lock\_timeout 4000

select \* from table wehre id = 10

T2执行时，会等待T1释放排他锁，等了4秒钟，如果T1还没有释放排他锁，T2就会抛出异常： Lock request time out period exceeded.

**8 附：各种锁的兼容关系表**

| Requested mode | IS | S | U | IX | SIX | X |

| Intent shared (IS) | Yes | Yes | Yes | Yes | Yes | No |

| Shared (S) | Yes | Yes | Yes | No | No | No |

| Update (U) | Yes | Yes | No | No | No | No |

| Intent exclusive (IX) | Yes | No | No | Yes | No | No |

| Shared with intent exclusive (SIX) | Yes | No | No | No | No | No |

| Exclusive (X) | No | No | No | No | No | No |

**9 如何提高并发效率**

1. 悲观锁：利用数据库本身的锁机制实现。通过上面对数据库锁的了解，可以根据具体业务情况综合使用事务隔离级别与合理的手工指定锁的方式比如降低锁的粒度等减少并发等待。
2. 乐观锁：利用程序处理并发。原理都比较好理解，基本一看即懂。方式大概有以下3种
   1. 对记录加版本号.
   2. 对记录加时间戳.
   3. 对将要更新的数据进行提前读取、事后对比。

不论是数据库系统本身的锁机制，还是乐观锁这种业务数据级别上的锁机制，本质上都是对状态位的读、写、判断。

**总结：**

数据库规定同一资源上不能同时共存共享锁和排他锁

事务与锁是不同的。事务具有ACID（

原子性、一致性、隔离性和持久性），锁是用于解决隔离性的一种机制。事务的隔离级别通过锁的机制来实现。另外锁有不同的粒度，同时事务也是有不同的隔离级别的（一般有四种：读未提交Read uncommitted，

读已提交Read committed，

可重复读Repeatable read，

可串行化Serializable）。



## **7.** **数据库事务的四大特性以及事务的隔离级别**

**事务的四大特性（ACID）**

⑴ 原子性（Atomicity）

　　原子性是指事务包含的所有操作要么全部成功，要么全部失败回滚，这和前面两篇博客介绍事务的功能是一样的概念，因此事务的操作如果成功就必须要完全应用到数据库，如果操作失败则不能对数据库有任何影响。

⑵ 一致性（Consistency）

　　一致性是指事务必须使数据库从一个一致性状态变换到另一个一致性状态，也就是说一个事务执行之前和执行之后都必须处于一致性状态。

　　拿转账来说，假设用户A和用户B两者的钱加起来一共是5000，那么不管A和B之间如何转账，转几次账，事务结束后两个用户的钱相加起来应该还得是5000，这就是事务的一致性。

⑶ 隔离性（Isolation）

　　隔离性是当多个用户并发访问数据库时，比如操作同一张表时，数据库为每一个用户开启的事务，不能被其他事务的操作所干扰，多个并发事务之间要相互隔离。

　　即要达到这么一种效果：对于任意两个并发的事务T1和T2，在事务T1看来，T2要么在T1开始之前就已经结束，要么在T1结束之后才开始，这样每个事务都感觉不到有其他事务在并发地执行。

　　关于事务的隔离性数据库提供了多种隔离级别，稍后会介绍到。

⑷ 持久性（Durability）

持久性是指一个事务一旦被提交了，那么对数据库中的数据的改变就是永久性的，即便是在数据库系统遇到故障的情况下也不会丢失提交事务的操作。

**事务隔离级别**

**事务的隔离性导致并发操作可能存在以下五种异常情况：第一类更新丢失、脏读、不可重复读、第二类更新丢失、幻读。**

      数据库并发操作存在的异常情况：  
1. 更新丢失（Lost update）： 两个事务都同时更新一行数据但是第二个事务却中途失败退出导致对数据两个修改都失效了这是系统没有执 行任何锁操作因此并发事务并没有被隔离开来。

2. 脏读取（Dirty Reads）： 一个事务开始读取 了某行数据但是另外一个事务已经更新了此数据但没有能够及时提交。这是相当危险很可能所有操作都被回滚。

3. 不可重复读取（Non-repeatable Reads）： 一 个事务对同一行数据重复读取两次但是却得到了不同结果。例如在两次读取中途有另外一个事务对该行数据进行了修改并提交。

4. 两次更新问题（Second lost updates problem）： 无法重复读取特例，有两个并发事务同时读取同一行数据然后其中一个对它进行修改提交而另一个也进行了修改提交这就会造成 第一次写操作失效。

5. 幻读（Phantom Reads）： 也称为幻像（幻 影）。事务在操作过程中进行两次查询，第二次查询结果包含了第一次查询中未出现的数据（这里并不要求两次查询SQL语句相同）这是因为在两次查询过程中有 另外一个事务插入数据造成的。

      为了避免上面出现几种情况在标准SQL规范中定义了4个事务隔离级别，不同隔离级别对事务处理不同 。

1.未授权读取（Read Uncommitted）： 也称 未提交读。允许脏读取但不允许更新丢失，如果一个事务已经开始写数据则另外一个数据则不允许同时进行写操作但允许其他事务读此行数据。该隔离级别可以通过 “排他写锁”实现。事务隔离的最低级别，仅可保证不读取物理损坏的数据。与READ COMMITTED 隔离级相反，它允许读取已经被其它用户修改但尚未提交确定的数据。

2. 授权读取（Read Committed）： 也称提交 读。允许不可重复读取但不允许脏读取。这可以通过“瞬间共享读锁”和“排他写锁”实现，读取数据的事务允许其他事务继续访问该行数据，但是未提交写事务将 会禁止其他事务访问该行。SQL Server 默认的级别。在此隔离级下，SELECT 命令不会返回尚未提交（Committed） 的数据，也不能返回脏数据。

3. 可重复读取（Repeatable Read）： 禁止 不可重复读取和脏读取。但是有时可能出现幻影数据，这可以通过“共享读锁”和“排他写锁”实现，读取数据事务将会禁止写事务（但允许读事务），写事务则禁 止任何其他事务。在此隔离级下，用SELECT 命令读取的数据在整个命令执行过程中不会被更改。此选项会影响系统的效能，非必要情况最好不用此隔离级。

4. 串行（Serializable）： 也称可串行读。提 供严格的事务隔离，它要求事务序列化执行，事务只能一个接着一个地执行，但不能并发执行。如果仅仅通过“行级锁”是无法实现事务序列化的，必须通过其他机 制保证新插入的数据不会被刚执行查询操作事务访问到。事务隔离的最高级别，事务之间完全隔离。如果事务在可串行读隔离级别上运行，则可以保证任何并发重叠 事务均是串行的。

## **8.** **Spring事务传播机制**

**链接：https://www.cnblogs.com/softidea/p/5962612.html**

**Spring在TransactionDefinition接口中规定了7种类型的事务传播行为，它们规定了事务方法和事务方法发生嵌套调用时事务如何进行传播,  
即协调已经有事务标识的方法之间的发生调用时的事务上下文的规则（是否要有独立的事务隔离级别和锁）**

**概述**

当我们调用一个基于Spring的Service接口方法（如UserService#addUser()）时，它将运行于Spring管理的事务 环境中，Service接口方法可能会在内部调用其它的Service接口方法以共同完成一个完整的业务操作，因此就会产生服务接口方法嵌套调用的情况， Spring通过事务传播行为控制当前的事务如何传播到被嵌套调用的目标服务接口方法中。

事务传播是Spring进行事务管理的重要概念，其重要性怎么强调都不为过。但是事务传播行为也是被误解最多的地方，在本文里，我们将详细分析不同事务传播行为的表现形式，掌握它们之间的区别。

**事务传播行为种类**

Spring在TransactionDefinition接口中规定了7种类型的事务传播行为，它们规定了**事务方法**和**事务方法发生嵌套调用时**事务如何进行传播：

表1事务传播行为类型

|  |  |
| --- | --- |
| 事务传播行为类型 | 说明 |
| PROPAGATION\_REQUIRED | 如果当前没有事务，就新建一个事务，如果已经存在一个事务中，加入到这个事务中。这是最常见的选择。 |
| PROPAGATION\_SUPPORTS | 支持当前事务，如果当前没有事务，就以非事务方式执行。 |
| PROPAGATION\_MANDATORY | 使用当前的事务，如果当前没有事务，就抛出异常。 |
| PROPAGATION\_REQUIRES\_NEW | 新建事务，如果当前存在事务，把当前事务挂起。 |
| PROPAGATION\_NOT\_SUPPORTED | 以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。 |
| PROPAGATION\_NEVER | 以非事务方式执行，如果当前存在事务，则抛出异常。 |
| PROPAGATION\_NESTED | 如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则执行与PROPAGATION\_REQUIRED类似的操作。 |

当使用PROPAGATION\_NESTED时，底层的数据源必须基于JDBC 3.0，并且实现者需要支持保存点事务机制。

**几种容易引起误解的组合事务传播行为**

当服务接口方法分别使用表1中不同的事务传播行为，且这些接口方法又发生相互调用的情况下，大部分组合都是一目了然，容易理解的。但是，也存在一些容易引起误解的组合事务传播方式。

下面，我们通过两个具体的服务接口的组合调用行为来破解这一难点。这两个服务接口分别是UserService和ForumService， UserSerice有一个addCredits()方法，ForumSerivce#addTopic()方法调用了 UserSerice#addCredits()方法，发生关联性服务方法的调用：

public class ForumService {

private UserService userService;

public void **addTopic()**{**①调用其它服务接口的方法**

//add Topic…

userService.**addCredits()**;**②被关联调用的业务方法**

}

}

嵌套调用的事务方法

对Spring事务传播行为最常见的一个误解是：当服务接口方法发生嵌套调用时，被调用的服务方法只能声明为 PROPAGATION\_NESTED。这种观点犯了望文生义的错误，误认为PROPAGATION\_NESTED是专为方法嵌套准备的。这种误解遗害不 浅，执有这种误解的开发者错误地认为：应尽量不让Service类的业务方法发生相互的调用，Service类只能调用DAO层的DAO类，以避免产生嵌 套事务。

其实，这种顾虑是完全没有必要的，PROPAGATION\_REQUIRED已经清楚地告诉我们：事务的方法会足够“聪明”地判断上下文是否已经存在一个事务中，如果已经存在，就加入到这个事务中，否则创建一个新的事务。

依照上面的例子，假设我们将ForumService#addTopic()和UserSerice#addCredits()方法的事务传播行为都设置为PROPAGATION\_REQUIRED，这两个方法将运行于同一个事务中。

为了清楚地说明这点，可以将Log4J的日志设置为DEBUG级别，以观察Spring事务管理器内部的运行情况。下面将两个业务方法都设置为PROPAGATION\_REQUIRED，Spring所输出的日志信息如下：

Using transaction object

[org.springframework.jdbc.datasource.DataSourceTransactionManager$DataSourceTransactionObject@e3849c]

**①为ForumService#addTopic()新建一个事务**

**Creating new transaction with name [com.baobaotao.service.ForumService.addTopic]:** PROPAGATION\_REQUIRED,ISOLATION\_DEFAULT

Acquired Connection [org.apache.commons.dbcp.PoolableConnection@dc41c5] for JDBC transaction

Switching JDBC Connection [org.apache.commons.dbcp.PoolableConnection@dc41c5] to manual commit

Bound value [org.springframework.jdbc.datasource.ConnectionHolder@ee1ede] for key [org.apache.commons.dbcp.BasicDataSource@4204] to thread [main]

Initializing transaction synchronization

Getting transaction for [com.baobaotao.service.ForumService.addTopic]

Retrieved value [org.springframework.jdbc.datasource.ConnectionHolder@ee1ede] for key [org.apache.commons.dbcp.BasicDataSource@4204] bound to thread [main]

Using transaction object [org.springframework.jdbc.datasource.DataSourceTransactionManager$DataSourceTransactionObject@8b8a47]

**②UserService#addCredits()简单地加入到已存在的事务中（即①处创建的事务）**

**Participating in existing transaction**

Getting transaction for [com.baobaotao.service.UserService.addCredits]

Completing transaction for [com.baobaotao.service.UserService.addCredits]

Completing transaction for [com.baobaotao.service.ForumService.addTopic]

Triggering beforeCommit synchronization

Triggering beforeCompletion synchronization

Initiating transaction commit

**③调用底层Connection#commit()方法提交事务**

Committing JDBC transaction on Connection [org.apache.commons.dbcp.PoolableConnection@dc41c5]

Triggering afterCommit synchronization

Triggering afterCompletion synchronization

Clearing transaction synchronization

嵌套事务

将ForumService#addTopic()设置为PROPAGATION\_REQUIRED时， UserSerice#addCredits()设置为PROPAGATION\_REQUIRED、PROPAGATION\_SUPPORTS、 PROPAGATION\_MANDATORY时，运行的效果都是一致的（当然，如果单独调用addCredits()就另当别论了）。

当addTopic()运行在一个事务下（如设置为PROPAGATION\_REQUIRED），而addCredits()设置为 PROPAGATION\_NESTED时，如果底层数据源支持保存点，Spring将为内部的addCredits()方法产生的一个内嵌的事务。如果 addCredits()对应的内嵌事务执行失败，事务将回滚到addCredits()方法执行前的点，并不会将整个事务回滚。内嵌事务是内层事务的一 部分，所以只有外层事务提交时，嵌套事务才能一并提交。

**嵌套事务**不能够提交，它必须**通过外层事务**来完成提交的动作，外层事务的回滚也会造成内部事务的回滚。

嵌套事务和新事务

PROPAGATION\_REQUIRES\_NEW 和 PROPAGATION\_NESTED也是容易混淆的两个传播行为。  
PROPAGATION\_REQUIRES\_NEW 启动一个新的、和外层事务无关的“内部”事务。该事务拥有自己的**独立隔离级别**和**锁**，不依赖于外部事务，独立地提交和回滚。当内部事务开始执行时，外部事务 将被挂起，内务事务结束时，外部事务才继续执行。

由此可见， PROPAGATION\_REQUIRES\_NEW 和 PROPAGATION\_NESTED 的最大区别在于：  
PROPAGATION\_REQUIRES\_NEW 将创建一个全新的事务，它和外层事务没有任何关系，  
而  
PROPAGATION\_NESTED 将创建一个**依赖于外层事**务的子事务，当外层事务提交或回滚时，子事务也会连带提交和回滚。

其它需要注意问题

以下几个问题值得注意：

1) 当业务方法被设置为**PROPAGATION\_MANDATORY**时，它就不能被非事务的业务方法调用。  
如将ForumService#addTopic ()设置为PROPAGATION\_MANDATORY，如果展现层的Action直接调用addTopic()方法，将引发一个异常。  
正确的情况是： addTopic()方法必须被另一个带事务的业务方法调用（如ForumService#otherMethod()）。  
所以 PROPAGATION\_MANDATORY的方法一般都是被其它业务方法**间接调用**的。

2) 当业务方法被设置为**PROPAGATION\_NEVER**时，它将不能被拥有事务的其它业务方法调用。假设UserService#addCredits ()设置为PROPAGATION\_NEVER，当ForumService# addTopic()拥有一个事务时，addCredits()方法将抛出异常。所以PROPAGATION\_NEVER方法一般是被**直接调用**的。

3)当方法被设置为PROPAGATION\_NOT\_SUPPORTED时，外层业务方法的事务**会被挂起**，当内部方法运行完成后，外层方法的事务重新运行。如果外层方法没有事务，直接运行，不需要做任何其它的事。

**小结**

在Spring声明式事务管理的配置中，事务传播行为是最容易被误解的配置项，原因在于事务传播行为名称（如 PROPAGATION\_NESTED：嵌套式事务）和代码结构的类似性上（业务类方法嵌套调用另一个业务类方法）。这种误解在很多Spring开发者中 广泛存在，本文深入讲解了Spring事务传播行为对业务方法嵌套调用的真实影响，希望能帮助读者化解对事务传播行为的困惑。