并行与分布式计算 - June 26'th, 2018

Transactions and Concurrency Control Concepts

https://github.com/rh01

事务简介

- 事务的定义
  - 事务是由用户定义的针对服务器对象的一组操作，它们组成一个不可分割的单元，由服务器执行。或者都做，或者都不做
- 事务的应用需求
  - 在一些应用中，客户要求给服务器的一组请求按下列意义是原子的
    * 它们不受其他并发客户的操作的干扰；
    * 所有操作或者全部成功完成，或者在面临服务器崩溃时，没有留下它们的任何影响
- 事务的目的
  - 所有由服务器管理的对象始终维持在一个一致的状态上
    * 在这些对象被多个事务访问时
    * 在服务器面临崩溃时
  - 事务的实现环境
    * 可恢复对象 (Recoverable object) 是能够在服务器崩溃后恢复的对象
    * 用于异步系统的设计
      · 假设消息可以被延迟

事务的故障模型/Lampson

- 磁盘故障、服务器故障以及通信故障
  - 对持久存储的写操作可能失败：文件写错误；向一个错误的块写
  - 服务器可以偶尔崩溃：服务器崩溃可能出现在任何时候，即使在服务器重启恢复时也可能出现
  - 消息传递可能有任意长的延迟；消息可能丢失、重复或者损坏
- 一些解决方法：
  - 读数据时可根据校验和来判断数据块是否损坏
  - 用原子写解决磁盘上的故障
  - 可以用新进程替换发生崩溃的进程
  - 用可靠的 RPC 机制屏蔽通信错误
  - 接收方通过校验和能够检测到受损消息。
  - 未发现的受损消息和伪造的消息会导致灾难性故障
- 这个故障模型的结论是：可以有算法在上述可预见故障下正确工作，而对不可预见的灾难性故障，则不能保证正常处理

事务的基本特性：ACID

- Atomic: To the outside world, the transaction happens indivisibly.

- Consistent: The transaction does not violate system invariants.
- Isolated: Concurrent transaction does not interfere with each other.
- Durable: Once a transaction commits, the changes are permanent.

事务的原子性

- 全有或全无 (All or nothing)
  - 一个事务或者成功完成，它的所有效果都记录到相关对象中；或者由于故障或有意放弃等原因而不留下任何的效果。这种全有或全无本身又包含两层含义：
    * 故障原子性：服务器崩溃时事务的效果是原子的
    * 持久性：事务成功完成以后，它的所有效果都被保存到持久存储中
- 隔离性
  - 每个事务的执行不受其他事务的干扰。换言之，事务在执行过程中的中间效果对其他事务是不可见的
  - 由并发控制确保隔离性

使用的例子

Account 接口的操作

- 每个账户用一个远程对象表示，这个远程对象的接口 Account 提供存取款和设置/获取余额的操作。

```
deposit(amount)
    deposit amount in the account
withdraw(amount)
    withdraw amount from the account
getBalance() → amount
    return the balance of the account
setBalance(amount)
    set the balance of the account to amount
```

Branch 接口的操作

- 银行的每个支行用一个远程对象表示，这个远程对象的接口 Branch 提供创建新账户、按名查找账户和计算分行总余额。它保存了账户名和对应的远程对象引用之间的关系。

```
create(name) → account
    create a new account with a given name
lookUp(name) → account
    return a reference to the account with the given name
branchTotal() → amount
    return the total of all the balances at the branch
```

平面事务 (Flat Transaction) 一个客户的银行事务

- Transaction T:
- a.withdraw(100);
- b.deposit(100);
- c.withdraw(200);
- b.deposit(200);

- 该事务指定涉及 A，B，C 账户的操作，在程序中用 a，b，c 表示 A，B，C 账户
- 头两个动作是从账户 A 转账 100 元至账户 B
- 后两个操作从账户 C 转账 200 元至账户 B
- 每个事务都由协调者 (Coordinator) 对象创建和管理

在 Coordinator 接口的操作

- 事务能力能加到有可恢复对象的服务器上
- 每个事务都由协调者对象 (Coordinator) 创建和管理，Coordinator 的接口如下：

`openTransaction() → trans;`
开始一个新事务，并返回该事务的唯一标识`TID trans`。
该标识将用于事务的其他操作中

`closeTransaction(trans) → (commit, abort);`
结束事务：如果返回值为`commit`表示该事务被成功提交；
否则返回 `abort`表示该事务被放弃

`abortTransaction(trans);`
放弃事务

事务活动历史

- 一个事务或者成功执行 (提交)，
  - 协调者看到所有的对象被保存在持久存储中
- 或者被客户放弃或者被服务器放弃
  - 所有临时的效果对其他事务不可见
  - 服务器放弃它的事务，客户将如何知道？

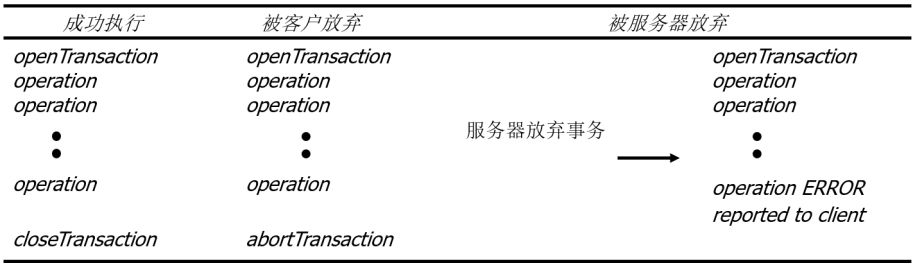

图 1：事务活动历史

嵌套事务

- 事务可以由其他事务构成
  - 从一个事务内可以发起几个事务

  - 分顶层事务和子事务，子事务还可以有它们自己的子事务
- 就事务的并发访问和故障处理而言，子事务对它的父事务是原子的
  - 并发访问：在同一个层次的子事务，例如 T1 和 T2，它们可以并发运行，但它们对公共对象的访问是串行化的
  - 故障处理：每一个子事务可能独立于父事务和其他子事务出现故障，如果某个子事务放弃了，由父事务决定怎么做，例如，启动另一个子事务来完成它的工作或干脆放弃
- 嵌套事务的优点
  - 在同一个层次的子事务可以并发运行
    * 提高了一个事务内的并发度
    * 如果这些子事务运行在不同的服务器上，那么它们能够并行执行
      · branchTotal 操作：在分行的每一个账户上调用 getBalance，实现 branchTotal 操作，如果分行具有独立的服务器，那么这些操作可以并行执行
  - 子事务可以独立提交和放弃
    * 若干嵌套的子事务可能更强壮
    * 父事务可以根据子事务是否放弃来决定不同的动作

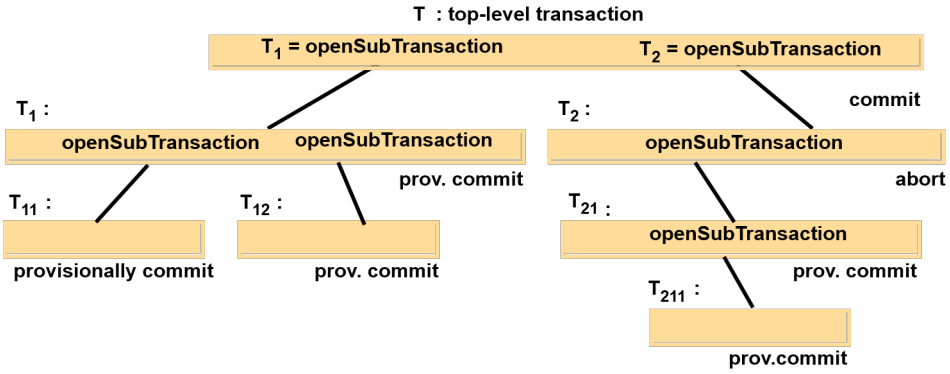

图 2：嵌套事务

嵌套事务的提交规则

- 事务在它的子事务完成以后，才能提交或放弃
- 当一个子事务执行完毕后，它可以独立决定是临时提交 (commit provisionally) 还是放弃。如果决定是放弃，那么这个决定是最终的
- 当父事务放弃时，所有的子事务都被放弃
- 如果某个子事务放弃了，那么父事务可以决定是放弃还是不放弃
- 如果顶层事务提交，那么所有临时提交的子事务将最终提交，这里假设它们的祖先没有一个被放弃

单个服务器上的事务

- 在面对并发事务时是原子的
  - 通过串行等价执行获得
- 在服务器崩溃时是原子的
  - 它们在持久存储中保存提交的状态
  - 考虑到放弃所可能产生的赃数据读取，它们可使用严格执行

  - 使用临时版本，用于提交/放弃 s
- 从子事务构造嵌套事务
  - 允许子事务的并发执行
  - 允许子事务的独立恢复

事务实现中的问题：并发控制

- 更新丢失和不一致检索问题
  - 更新丢失：两个事务都读一个变量的旧值，并用它计算新值
  - 不一致检索：一个检索事务观察的值正在进行一个更新事务
- 如何利用事务的串行等价执行来避免这些问题
  - 假设 deposit，withdraw，getBalance 和 setBalance 都是同步操作，即，它对记录账户余额的效果是原子的

更新丢失问题

- 这三个账户 A,B,C 的初始余额分别是 $100，$200 和 $300。
- 两次转账的金额都是当前 B 账户余额的 10%

事务 ***T***:		事务 ***U***:	
balance = b.getBalance(); *b.setBalance(balance*1.1);* *a.withdraw(balance/10)*		*balance = b.getBalance();* *b.setBalance(balance*1.1);* *c.withdraw(balance/10)*	
balance = b.getBalance();	$200		
		balance = b.getBalance();	$200
		*b.setBalance(balance*1.1);*	$220
*b.setBalance(balance*1.1);*	$220		
a.withdraw(balance/10)	$80		
		c.withdraw(balance/10)	$280

图 3：更新丢失问题. 两次转账的最终效果应该是增加账户 B 的余额 10%两次——200, 220, 242 但帐户 B 只得到 220。U 的修改丢失了.

不一致检索问题

- 在事务 W 调用 branchTotal 方法计算银行所有账户的总余额时 (应该是 $600)，事务 V 将资金 $100 由账户 A 转到账户 B
- 最终 W 计算得所有账户的总余额是 $500

事务 *V*:		事务 *W*:	
a.withdraw(100) *b.deposit(100)*		*aBranch.branchTotal()*	
a.withdraw(100);	$0		
		total = a.getBalance()	0
		total = total+b.getBalance()	$200
		total = total+c.getBalance()	
b.deposit(100)	$300	⋮	

图 4：因为在 W 计算 A 和 B 的余额时，V 仅做了取款部分，所以出现不一致检索

串行等价性

- 串行等价的交错执行：是指并发事务交错执行操作的综合效果与按某种次序一次执行一个事务的效果一样
- 同一效果是指：
  - 读操作返回相同的值
  - 事务结束时，所有对象的实例变量也具有相同的值

串行等价的交错执行 T 和 U(解决更新丢失问题)

- 如果两个事务 T 和 U 一前一后执行，就不会发生更新丢失
- 如果它们按串性等价顺序执行，也不会发生更新丢失

事务 *T*: *balance = b.getBalance()* *b.setBalance(balance*1.1)* *a.withdraw(balance/10)*		事务 *U*: *balance = b.getBalance()* *b.setBalance(balance*1.1)* *c.withdraw(balance/10)*	
balance = b.getBalance()	$200		
*b.setBalance(balance*1.1)*	$220		
		balance = b.getBalance()	$220
		*b.setBalance(balance*1.1)*	$242
a.withdraw(balance/10)	$80		
		c.withdraw(balance/10)	$278

图 5：对 B 的访问是串行的，其它部分可以重叠

串行等价的交错执行 V 和 W(解决不一致检索问题)

- 如果 W 在 V 之前或之后运行，那么不会出现不一致检索问题
- 如果以 V 和 W 的串行等价顺序执行，也不会出现不一致检索问题
- 图示是串行的，但它不需要是这样的

事务 *V*: *a.withdraw(100); b.deposit(100)*		事务 *W*: *aBranch.branchTotal()*	
a.withdraw(100);	$0		
b.deposit(100)	$300		
		total = a.getBalance()	$000
		total = total+b.getBalance()	$300
		total = total+c.getBalance()	
		...	

图 6：我们可以重叠 W 的第 1 行和 V 的第 2 行

读写操作的冲突规则

- 如果两个操作的执行效果依赖于它们的执行次序，我们称这两个操作相互冲突
  - 例如：读和写

不同事务的操作		是否冲突	原因
read	*read*	No	由于两个读操作的执行效果不依赖这两个操作的执行次序
read	*write*	Yes	由于一个读操作和一个写操作的执行效果依赖于它们的执行次序
write	*write*	Yes	由于两个写操作的执行效果依赖于这两个操作的执行次序

图 7: 读写操作的冲突规则

用冲突操作定义的串行等价性

- 两个事务串行等价的充分必要条件是，两个事务中所有的有冲突的操作都按相同的次序执行对对象的访问考虑
  - T: x = read(i); write(i, 10); write(j, 20);
  - U: y = read(j); write(j, 30); z = read (i);
- 串行等价性要求下面两个条件之一
  - 事务 T 在事务 U 之前对 i 执行冲突访问，并且事务 T 在事务 U 之前对 j 执行冲突访问
  - 事务 U 在事务 T 之前对 i 执行冲突访问，并且事务 U 在事务 T 之前对 j 执行冲突访问
- 串行等价性可用做设计并发控制机制的一个准则

非串行等价地交错执行事务 **T** 和 **U** 的操作

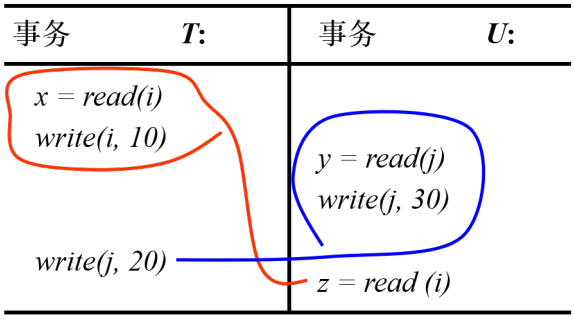

图 8: 读写操作的冲突规则

- 每个事务对对象 i 和 j 的访问，相对于另一个事务是串行的，但
- 事务 T 对变量 i 访问都在事务 U 对 i 访问之前，
- 事务 U 对变量 j 的访问都在事务 T 对 j 访问之前，
- 因此，这个交错执行顺序不是串行等价的

并发控制

- 普通的并发控制：对对象访问的并发控制
- 事务的并发控制
  - 要求事务的执行有串行等价的特性
  - 必须调度事务使它们对共享数据的执行效果是串行等价的
  - 串行等价性与数据复制中的顺序一致性
- 串行等价性要求：

  (a) 一个事务对一个对象的所有访问相对于其他事务进行的访问是串性化的。

  (b) 两个事务的所有的冲突操作对必须以相同的次序执行

事务实现中的问题：事务放弃时的恢复

- 如果事务取消，服务器必须保证其它并发事务看不到被取消事务的影响
- 脏数据读取 (dirty read)
  - 一个事务的读操作和另一个事务的写操作 (写操作先于读操作，该事务以后被放弃) 对同一个对象进行操作
  - 如果某个事务用脏数据 (被放弃事务的更新结果) 提交了，那么该事务是不可恢复的
- 过早写入 (premature writes)
  - 由不同事务对相同对象进行写操作，其中有一个写操作被放弃了

事务 T 放弃时的脏数据读取

事务 *T*:		事务 *U*:	
a.getBalance()		*a.getBalance()*	
a.setBalance(balance + 10)		*a.setBalance(balance + 20)*	
balance = a.getBalance()	$100		
a.setBalance(balance + 10)	$110		
		balance = a.getBalance()	$110
		a.setBalance(balance + 20)	$130
		commit transaction	
abort transaction			

图 9: 读写操作的冲突规则

- U 已经提交了，所以，它不能取消 (undo)

事务的可恢复性

- 如果某个事务 (例如 U) 在访问了以后被放弃事务的更新结果之后，提交了，那么，该事务是不可恢复的
- 为了可恢复性：
  - 提交要推迟，直到它读取更新结果的事务都已提交
    - 例如，U 应该等待 T 提交或放弃，如果 T 放弃，那么 U 也必须放弃

连锁放弃

- 假设事务 U 推迟提交直到事务 T 被放弃
  - 那么，此时事务 U 必须要放弃
  - 其他观察到 U 结果的事务同样也要放弃
  - 这些事务的放弃可能导致更多的事务被放弃
- 这种情况被称为连锁放弃
- 为了避免这种情况出现
  - 只允许事务读取由已提交事务修改的对象
  - 为了保证这一点，读某对象的操作必须推迟到写该对象数据的事务提交或放弃
- 防止连锁放弃是一个比保证事务可恢复性更强的条件

过早写入：重写 (overwriting) 未提交的值

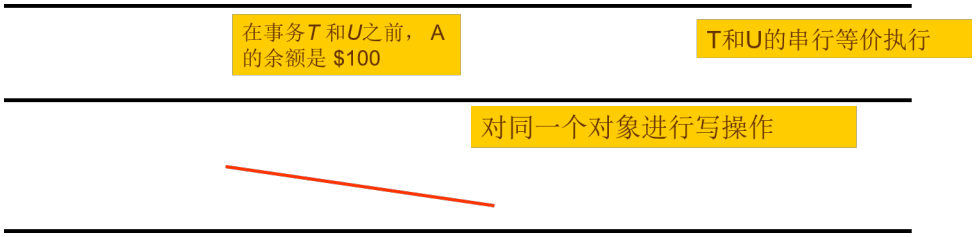

图 10: 过早写入

- 一些数据库系统保存“前映像”(before images)，在事务放弃之后，用前映像中的值来恢复它们
  - e.g. $100 是 T 写操作的前映像, $105 是 U 写操作的前映像

  - 如果 U 放弃了，我们获得正确的余额 $105,
  - 如果 U 提交了，接着 T 放弃了，我们从前映像获得 a 的值是 $100 而不是 $110
- 为了保证使用前映像进行事务恢复能获得正确的结果，写操作必须等到前面修改同一对象的其他事务提交或放弃后才能进行

事务的严格执行

- 事务的严格执行
  - 为了避免脏数据读取和过早写入，延迟读和写操作
  - 如果对一个对象的读操作和写操作都推迟到先前写同一对象的其他事务提交或放弃后才进行，那么这种事务的执行被称为是严格的
  - 事务的严格执行可以真正保证事务的隔离特性
- 在事务的进行过程中可使用临时版本 (tentative versions)

并发控制之锁

- 服务器可以 (用锁) 通过串行化对象访问来达到事务的串行等价，实现 (a)(一个事务对一个对象的所有访问相对于其他事务进行的访问是串性化的)
- 为了确保 (b)(两个事务的所有的冲突操作对必须以相同的次序执行)，事务在释放任何一个锁之后，都不允许再申请新的锁，即采用两阶段锁
  - 两阶段锁：有一个“增长”阶段和一个“收缩”阶段
- 严格两阶段锁：所有在事务执行过程中获取的锁必须在事务提交或放弃后才能释放，可防止事务放弃时的脏数据读取和过早写入问题
- 锁的粒度——将锁加到小东西上

事务 T 和 U 使用互斥锁

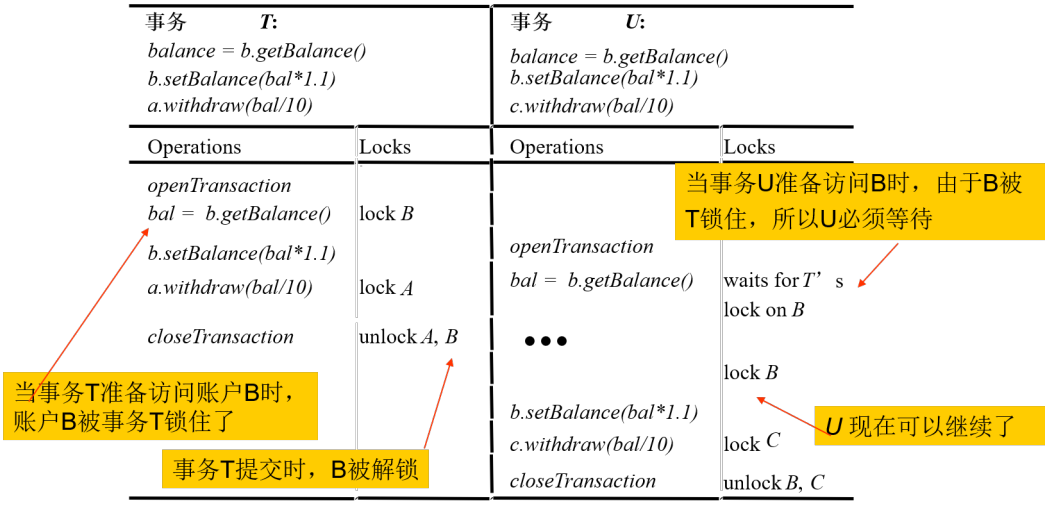

图 11: 事务 T 和 U 使用互斥锁

- 开始时，A、B 和 C 的余额均未被加锁

读写冲突规则

- 并发控制协议用于解决不同事务中的操作访问同一个对象时的冲突

- 按什么决定一对操作是冲突的？如果操作的效果与操作执行的顺序有关，就称操作是冲突的
- 不同事务对同一个对象的读操作是不冲突的
- 对读和写操作都使用简单的互斥锁会过多地降低并发度。因此，互斥锁会过多降低并发性
- “多个读/一个写”机制能够支持多个并发事务同时读取某个对象，或者允许一个事务写对象，但它不允许这两者同时存在
- 使用两种锁：读锁和写锁 (读锁也被称为共享锁)

锁兼容性

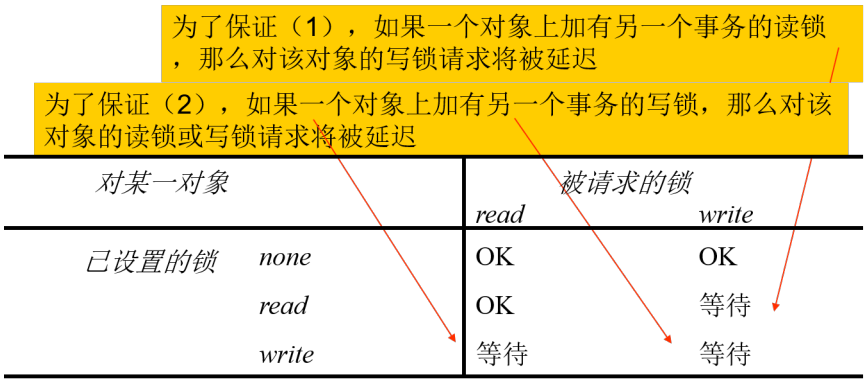

图 12: 锁兼容性

- 读写操作冲突规则告诉我们：
  1. 如果事务 T 已经针对某个对象进行了读操作，那么并发事务 U 在事务 T 提交或放弃前不能写该对象
  2. 如果事务 T 已经针对某个对象进行了写操作，那么并发事务 U 在事务 T 提交或放弃前不能写或读该对象

锁的提升

- 更新丢失——两个事务读一个对象，然后用它计算一个新值
- 通过让后一个事务延迟读直到前一个事务完成，可以避免更新丢失问题
- 每个事务在读对象时都设置一个读锁，然后在写同一个对象时将读锁提升为写锁
- 当后继事务要求一个读锁/写锁时，该请求将被延迟直到当前事务完成工作为止
- 提升锁：将某个锁转化为更强的锁——即更具有互斥性的锁
  - 锁降级（让锁变得更弱）是不允许的

在严格的两阶段锁中使用锁

1. 在某个事务中有一个操作访问某个对象时：
   (a) 如果该对象未被加锁，那么它被加上锁并且操作继续执行
   (b) 如果该对象已被其他事务设置了一个冲突锁，那么该事务必须等待直到对象被解锁
   (c) 如果该对象被其他事务设置了一个不冲突的锁，那么这个锁被共享并且操作继续执行
   (d) 如果该对象已被同一事务锁住，那么必要时提升该锁，并且继续执行操作 (当一个冲突的锁阻止了锁的提升，那么使用规则 (b))
2. 当事务被提交或被放弃时，服务器将释放该事务在对象上所加的所有锁

写锁造成的死锁

事务 *T*		事务 *U*	
Operations	**Locks**	**Operations**	**Locks**
a.deposit(100);	write lock *A*		
		b.deposit(200)	write lock *B*
b.withdraw(100)			
•••	waits for *U*'s	*a.withdraw(200);*	waits for *T*'s
•••	lock on *B*	•••	lock on *A*
•••		•••	
		•••	

图 13: 写锁造成的死锁

- 在使用锁的情况下，两个事务 T 和 U 分别获取了一个账户的写锁，但在访问另一个账户时被阻塞，这就是死锁。
- 锁管理器必须设计成能处理死锁

等待图

- 死锁定义
  - 死锁是一种系统状态，在该状态下一组事务中的每一个事务都在等待其他事务释放某个锁。
  - 等待图可用来表示当前事务之间的等待关系。如果事务 T 在等待事务 U 释放某个锁，那么在等待图中有一条从结点 T 指向结点 U 的边

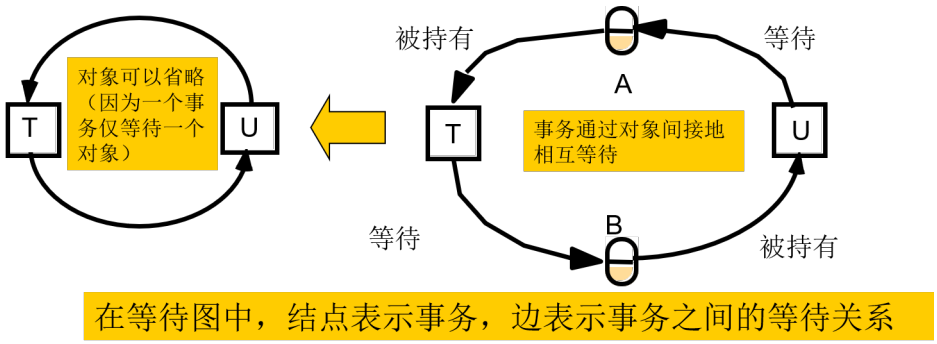

图 14: 等待图

等待图中的环路

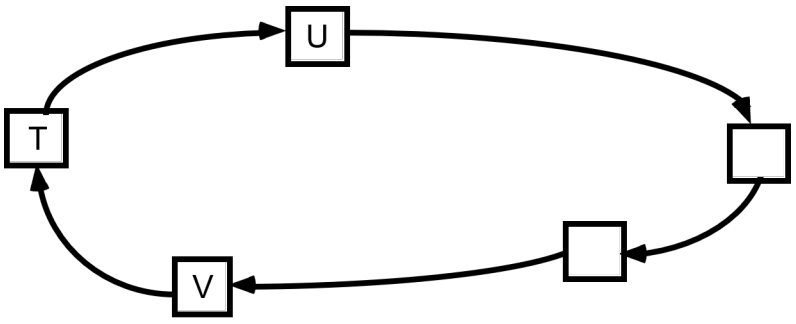

图 15: 等待图中的环路

- 假设等待图中包含环路 T→U→⋯→V→T

- 环路中的每一个事务都在等待下一个事务
- 所有的事务都阻塞着等待锁
- 没有一个锁会释放，那么，这些事务发生死锁了
- 如果环路中的某一个事务被放弃，那么它的锁就被释放，环路就被打破了

另一个等待图

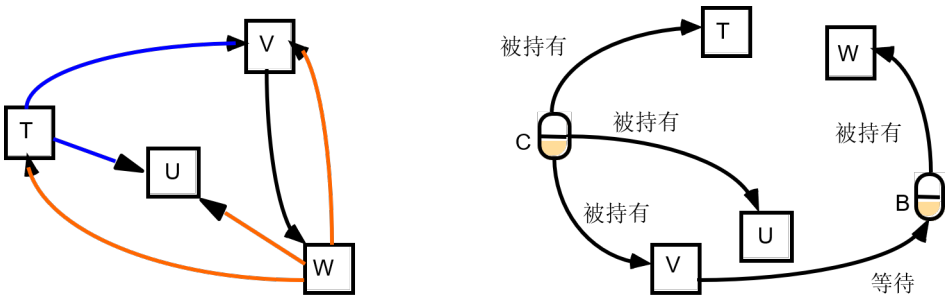

图 16: 另一个等待图

- 事务 T、U 和 V 共享对象 C 上的读锁
- 事务 W 拥有对象 B 上的写锁，而事务 V 正在等待对象 B 的锁。
- 接着，事务 T 和 W 请求对象 C 上的写锁，那么系统将进入死锁状态：V 在两个回路中，见左图

死锁预防是不现实的

- 让每个事务在开始运行时就锁住它要访问的所有对象
  - 带来了不必要的资源访问限制
  - 很多情况下在事务开始时无法预计事务将访问哪些对象
- 死锁还可以通过以固定次序加锁来预防
  - 但是这会造成过早加锁和减少并发度

死锁检测

- 通常，锁管理器负责死锁检测
- 锁管理器必须拥有一个等待图的表示，以便不时地进行死锁检测
- 锁管理器的 setLock 和 unLock 操作用于增加或删除等待图中的边
- 一旦检测出死锁，必须选择出要放弃的事务，然后删除等待图中与该事务有关的所有边
- 检测出死锁之后，必须选择出要放弃的事务，打破环路，选择一个要放弃的事务不是个简单的问题。要考虑的因素有事务的运行时间、它处于多少环路中等等

锁超时

- 锁超时可以被用于解除死锁
  - 每个锁都被给定一个时间期限，一旦超过期限，锁将成为可剥夺的
  - 如果没有其他事务竞争被锁住的对象，那么具有可剥夺锁的这个对象可以继续被锁住
  - 一旦有一个事务正在等待由可剥夺锁保护的对象时，这个锁将被等待事务剥夺，即，对象被解锁，等待事务将继续执行，被剥夺锁的事务通常被放弃
- 锁超时的问题

– 系统中本没有死锁，但是这些事务被放弃了，原因是这些事务的锁变成可剥夺的，而其他事务正在等待这些事务的锁
– 如果系统过载，那么超时事务的数量将增加，长时间运行的事务会被经常放弃
– 适当的超时时间长度很难确定

利用锁超时解决死锁

Transaction T		Transaction U	
Operations	**Locks**	**Operations**	**Locks**
a.deposit(100);	write lock *A*		
		b.deposit(200)	write lock *B*
b.withdraw(100)			
•••	waits for *U*'s lock on *B*	*a.withdraw(200);*	waits for T's lock on *A*
	(timeout elapses)	•••	
	T's lock on *A* becomes vulnerable, unlock *A*, abort T	•••	
		a.withdraw(200);	write locks *A* unlock *A*, *B*

图 17: 利用锁超时解决死锁

在加锁机制中增加并发度

- 双版本锁 (two-version locking)
  – 允许一个事务针对对象的临时版本进行写操作，而其他的事务针对同一对象提交后的版本进行读操作

For one object		***Lock to be set***		
		read	***write***	***commit***
Lock already set	***none***	OK	OK	OK
	read	OK	OK	wait
	write	OK	wait	
	commit	wait	wait	

图 18: 双版本锁

- 使用混合粒度的锁——层次锁 (hierarchic locks)
  – 例如，branchTotal 操作用一个锁锁住所有账户，而其他操作锁住单个账户，从而减少锁的数量
  – 层次锁具有减少锁数量的优势。但是它的相容性表和锁提升规则更加复杂

For one object		*Lock to be set*			
		read	*write*	*I-read*	*I-write*
Lock already set	*none*	OK	OK	OK	OK
	read	OK	wait	OK	wait
	write	wait	wait	wait	wait
	I-read	OK	wait	OK	OK
	I-write	wait	wait	OK	OK

图 19: 层次锁

锁的缺点

- 锁有开销，即使是只读事务
  - 锁只在最坏的情况下，就是有其他事务来修改这个数据的时候才起作用
- 使用锁会引起死锁
- 为了避免连锁放弃，锁必须保留到事务结束才能释放。这会大大降低潜在并发度

乐观并发控制

- 事务可以不受限制地执行，直到 closeTransaction
- 然后，检测是否与其他事务有冲突。
- 如果确实存在冲突，那么，就要放弃一个事务
  - 这种方案被称为乐观的，因为两个客户事务访问同一个对象的可能性是很低的
- 每个事务有三个阶段：
  - 工作阶段
  - 验证阶段
  - 更新阶段

三个阶段

- 工作阶段
  - 每个事务使用它访问的对象的临时版本
  - 协调者记录每个事务的读集合和写集合
- 验证阶段
  - 在接收到 closeTransaction 请求时，协调者将验证事务
  - 如果验证成功，那么该事务就允许提交
  - 否则，或者放弃当前事务，或者放弃其他与当前事务冲突的事务
- 更新阶段
  - 当事务通过验证以后，记录在所有临时版本中的更新将持久化
  - 只读事务可在通过验证后立即提交
  - 写事务在对象的临时版本记录到持久存储后即可提交

事务的验证

- 每个事务在进入验证阶段之前被赋予一个事务号
- 为了确保某个事务的执行对其他重叠事务是串行等价的，使用下列读-写规则：
- 重叠事务是指在该事务启动时还没有提交的事务
  - T_v：被验证的事务
  - T_i：重叠事务

T_v	T_i	规则	
write	read	1. T_i 不能读取被事务T_v写的对象	向前
read	write	2. T_v 不能读取被事务T_i写的对象	向后
write	write	3. T_v 不能写被事务T_i写的对象，并且T_i 不能写被事务T_v写的对象	

•通过省略规则3，简化验证

图 20: 事务的验证

事务的向后验证

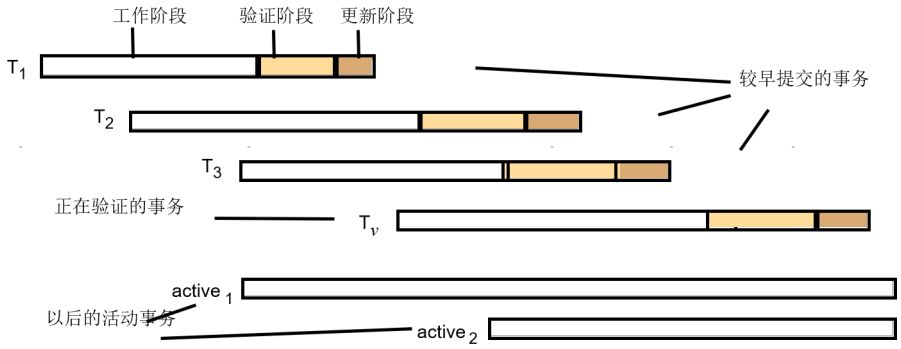

图 21: 事务的向后验证

- 由于较早的重叠事务的读操作在 T_v 验证 (和可能的更新) 之前进行，所以它们不会受当前事务写操作的影响，所以满足规则 1(T_v 的写对 T_i 的读)
- 检查 T_v 与前面重叠的事务：检查 T_v 的读集合是否和早先的 T_i 的写集合重叠
  - 在 T_v 完成它的工作阶段之前，T2 和 T3 提交了，所以，T_v 的读集合必须与 T2 和 T3 的写集合比较
- 规则 3：(写对写) 假设验证和更新没有重叠，所以自动满足规则 3

事务的向前验证

- 规则 1 ：T_v 的写集合与所有重叠的活动事务的读集合进行比较——活动事务是那些处在工作阶段中的事务
  - 在图中，T_v 的写集合必须与 active1、active2 的读集合进行比较
- 规则 2 ：(T_v 读对 T_i 写) 自动满足，因为活动事务在 T_v 完成之前不会进行写操作

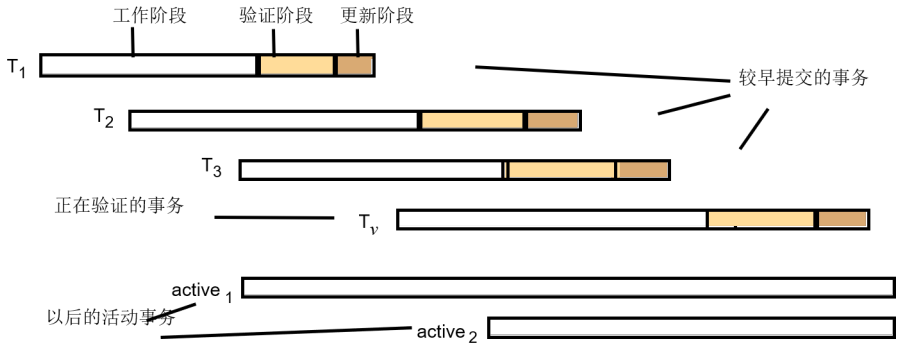

图 22: 事务的向前验证

向前验证 vs. 向后验证的

- 发生冲突时，选择放弃事务的方法
  - 向前验证在处理冲突时有较强的灵活性，而向后验证只有一种选择，即，放弃被验证的事务
- 通常读集合大于写集合
  - 向后验证
    * 将较大的读集合和较早事务的写集合进行比较
    * 存储旧的写集合的开销
  - 向前验证
    * 将较小的写集合和其他活动事务的读集合比较
    * 需要允许在验证时启动新的事务

饥饿还是死锁？

- 锁机制造成死锁：通过放弃事务解除死锁
- 乐观并发控制造成饥饿
  - 在一个事务被放弃后，它通常由客户程序重新启动，但是不能保证事务最终能够通过验证检查
- 在这两种情况中，被放弃的事务不被保证以后会成功完成
- 哪一个更可能？——饥饿或死锁
  - 发生死锁的可能性比饥饿少，因为并发控制机制用于避免事务发生冲突。一般情况下，锁让事务等待
  - 分布式死锁检测非常难实现

时间戳排序并发控制

- 事务中的每一个操作在执行之前首先进行验证
  - 如果该操作不能通过验证，那么该事务将被立即放弃
  - 每个事务在启动时被赋予一个唯一的时间戳，时间戳定义了该事务在事务时间顺序中的次序，事务的请求可以根据它们的时间戳进行全排序
- 基本的时间戳排序规则
  - 只有在对象最后一次读访问或写访问是由一个较早的事务执行的情况下，事务的对该对象的写请求是有效的
  - 只有在对象的最后一次写访问是由一个较早的事务执行的情况下，事务的对该对象的读请求是有效的
  - 以上规则假设系统中的每个对象只有一个版本
  - 如果对象保持多个版本 (一个提交版本多个临时版本)，通过细化时间戳排序规则，可利用临时版本保证每个事务访问的对象版本是一致的

细化时间戳排序规则

规则	T_c	T_i	
1.	write	read	如果T_i>T_c，那么T_c不能写被T_i读过的对象，这要求T_c≥该对象的最大读时间戳。
2.	write	write	如果T_i>T_c，那么T_c不能写被T_i写过的对象，这要求T_c>已提交对象的写时间戳。
3.	read	write	如果T_i>T_c，那么T_c不能读被T_i写过的对象，这要求T_c >已提交对象的写时间戳。

T_i>T_c 意味着T_i 晚于 T_c

图 23: 事务的向前验证

- 写操作针对临时版本，每当一个事务对某个对象进行写操作时，服务器就创建该对象的一个新的临时版本，并将该临时版本的写时间戳设置为这个事务的时间戳
- 当进行一个事务的读操作时，它作用于时间戳为小于等于该事务时间戳的最大写时间戳的对象版本上
- 当事务被提交时，临时版本的值变成了对象提交版本的值，临时版本的写时间戳变成了该提交对象的写时间戳

时间戳排序的写规则

- 通过组合规则 1(写/读) 和规则 2(写/写)，我们可以得到下列规则，用于决定是否可接受由事务 Tc 对对象 D 执行写操作

写操作和时间戳

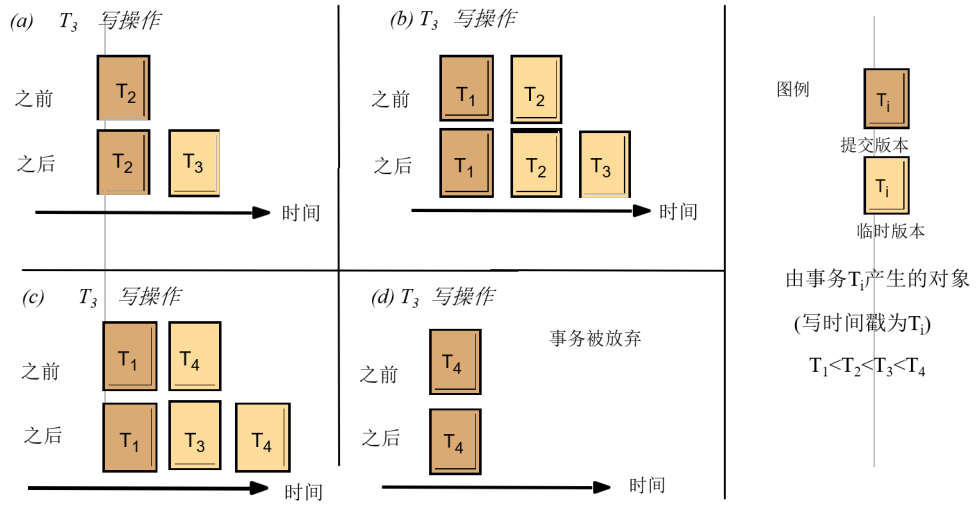

图 24: 写操作和时间戳

- 在情况 (a), (b) 和 (c) 中，T3> 提交版本的写时间戳，因此服务器创建一个时间戳为 T3 的对象临时版本，并插入到按事务时间戳排序的临时版本列表中。
- 为了允许写，T3≥ 对象的最大的读时间戳

时间戳排序的读规则

- 应用规则 3，我们可以得到下面的规则，用于决定事务 Tc 对对象 D 执行读操作，或者是马上接受，或者等待，或者拒绝

读操作和时间戳

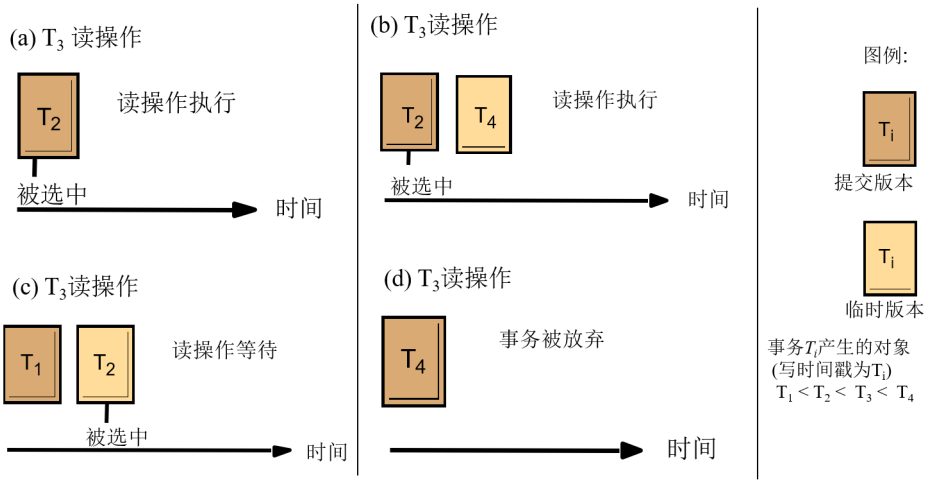

图 25: 读操作和时间戳

- 情况（a）和（b）中，读操作针对提交版本，在（a）中该提交版本是对象的唯一版本，而（b）中有一个临时版本属于另一个较晚的事务。
- 用 T3 读说明时间戳排序读规则，这里，写时间戳的版本 $\leq$T3

用时间戳排序的事务提交

- 当一个协调者收到事务提交请求后，由于事务的所有操作在执行之前都与早先的事务进行了一致性检查，因此它总能提交
  - 每个对象的提交版本必须按照时间戳排序创建
  - 服务器有时需要等待
  - 为了保证在服务器崩溃后事务是可恢复的，在确认客户的提交事务的请求之前，必须将对象的临时版本和提交信息记录到持久存储中
- 时间戳排序算法是严格的
  - 时间戳排序的读规则要求事务对对象的读操作等待，直到所有写该对象的较早事务提交或者放弃
  - 对象的提交版本也按时间戳序排列，保证了事务对对象的写操作必须等待，直到所有写该对象的较早事务提交或者放弃

并发控制方法的比较

- 悲观方法，即在访问每个对象时都检测事务之间是否会产生冲突
  - 时间戳排序：静态地决定事务之间的串行顺序
  - 加锁：动态地决定事务之间的串行顺序
  - 对读多于写的事务，时间戳排序较好
  - 对写多于读的事务，锁较好
  - 放弃的策略：
    * 时间戳排序-马上放弃
    * 锁-等待但可能得到死锁
- 乐观方法
  - 所有的事务都被允许执行，但可能在结束时被放弃
  - 如果并发事务之间的冲突较小时，乐观并发控制具有较好的性能，但当事务被放弃时，乐观并发控制需要重复可观的工作

- 上述并发控制机制还不够用，例如：
  - 在协同工作领域，用户要求在其他用户更新数据时马上得到通知
  - 诸如协同 CAD 等领域需要用户参与冲突的解决

并发控制方法的小结

- 操作冲突形成了各种并发控制协议的基础
- 在调度事务的某个操作时有三种策略
  - 1. 立即执行
    2. 推迟执行
    3. 放弃事务
  - 严格的两阶段锁使用了前两种策略，只有在死锁时才求助于放弃事务
    * 根据事务访问公共对象的时间对事务进行排序
    * 主要缺点是会造成死锁
  - 时间戳排序使用上述三种策略，没有死锁
    * 根据事务开始时的时间来排列事务对对象的访问顺序
  - 乐观并发控制在事务的执行过程中不进行任何形式的检测，直到事务完成
    * 要执行验证，会发生饥饿