



# 第5章 事务和并发控制

---



## 第5章 事务和并发控制

---

- 事务
- 锁
- 乐观并发控制
- 时间戳排序
- 并发控制方法的比较
- 小结



# 事务

---

## ■ 事务

由客户定义的针对服务器对象的一组操作（操作序列），它们组成一个不可分割的单元，由服务器执行。

## ■ 事务的目标

在多个事务访问对象以及服务器面临故障的情况下，保证所有由服务器管理的对象始终保持一个一致的状态。



# 事务的故障模型

---

- 硬盘故障

- 对持久性存储的写操作可能发生故障（写操作无效或者写入错误值）。

- 服务器故障

- 服务器可能偶尔崩溃。

- 通信故障

- 消息传递可能有任意长的延迟。消息可能丢失、重复或者损害。

- 事务能够处理进程的崩溃故障和通信的遗漏故障，但是不能处理拜占庭行为



## 事务的特性

---

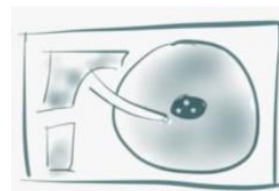
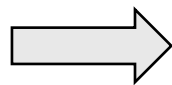
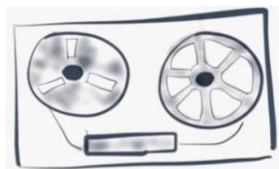
- ACID

- **A**tomicity (原子性)
- **C**onsistency (一致性)
- **I**solation (隔离性)
- **D**urability (持久性)

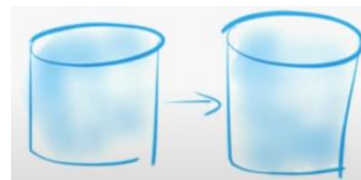
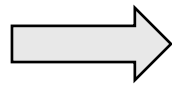
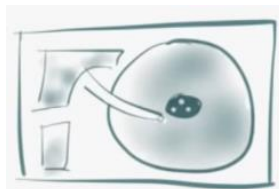
# Durability (持久性)

- 一旦事务完成，它的所有效果将被保存到持久存储中，文件中的数据不受服务器崩溃的影响

■ 磁带 → 硬盘



■ 硬盘 → 复制





# Consistency (一致性)

---

- 事务（应用）将系统从一个一致性状态转换到另一个一致性状态
  - 一致性状态：数据库满足一定的不变性即完整性约束
    - 例如，在会计系统中，所有贷款总和等于借款总和
  - 应用如何使用数据库，而非数据库本身的属性
- $\neq C$  in CAP theorem
  - Consistency: 每次读取都会收到最近的写或错误

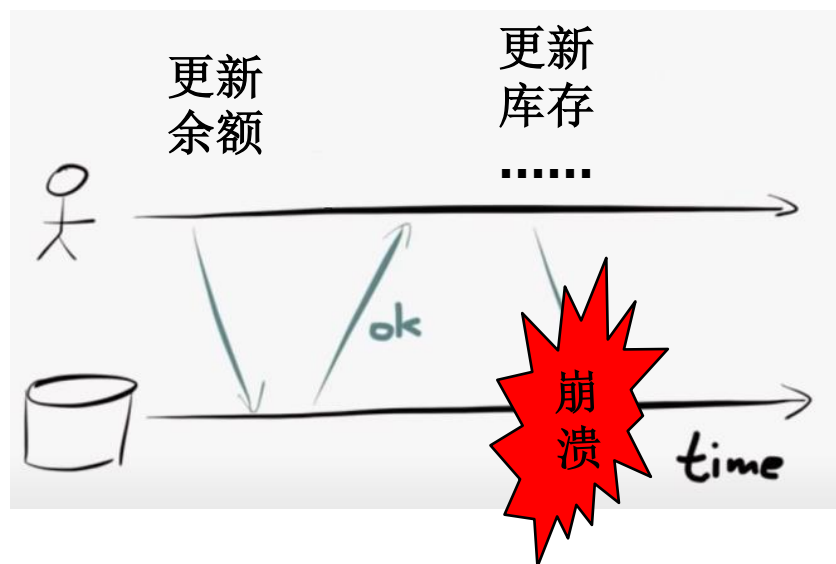
# Atomicity (原子性)

- 事务必须**全有**或者**全无**（多个对象的原子性）
  - 处理错误（死锁，网络异常，系统崩溃……）
  - 放弃（abort）时回滚（roll back）写入

**A**tomicity



**A**bortability







# Isolation (隔离性)

---

- 事务执行过程中的中间效果对其它事务不可见
  - 事务“拥有”整个数据库
  - 并发对事务不可见



# 事务的三种执行情况

成功执行

被~~客户~~放弃

被~~服务器~~放弃

*openTransaction*

操作

操作

●

●

操作

*closeTransaction*

*openTransaction*

操作

操作

●

●

操作

*abortTransaction*

服务器

放弃事务



*openTransaction*

操作

操作

●

●

向客户报告 *ERROR*

一旦事务被放弃，服务器必须保证清除所有效果，使该事务的影响对其他事务不可见



# 并发控制

---

- 并发事务中两个典型问题
  - 更新丢失
  - 不一致检索



## 并发控制

---

### ■ 更新丢失问题

初值：帐户A、B、C分别为\$100、\$200、\$300

操作：两次转帐，每次转帐金额为B当前帐户余额的10%

- $A \rightarrow B \quad 200 * 10\% = 20$
- $C \rightarrow B \quad 220 * 10\% = 22$

期望结果：B的终值应为\$242



# 银行示例函数

## ■ Account接口中的操作

<i>deposit(amount)</i>	//向帐户存amount数量的钱
<i>withdraw(amount)</i>	//从帐户中取amount数量的钱
<i>getBalance()</i> -> amount	//返回帐户中余额
<i>setBalance(amount)</i>	//将帐户余额设置成amount

## ■ Branch接口中的操作

<i>create(name)</i> -> account	//用给定的用户名创建一个新帐户
<i>lookUp(name)</i> -> account	//根据给定的用户名查找帐户，并返回该帐户的一个引用
<i>branchTotal()</i> -> amount	//返回支行中所有帐户余额的总和

# 更新丢失问题

两个事物都读取一个变量的  
**旧数据**并用它来计算**新数据**

初值：帐户A、B、C分别为\$100、\$200、\$300

事务T: 帐户A->帐户B

```
balance = b.getBalance();  
b.setBalance(balance*1.1);  
a.withdraw(balance/10)
```

*balance* = *b.getBalance()*;      \$200

*b.setBalance(balance\*1.1);*      \$**220**

*a.withdraw(balance/10)*      \$80

事务U: 帐户C->帐户B

```
balance = b.getBalance();  
b.setBalance(balance*1.1);  
c.withdraw(balance/10)
```

*balance* = *b.getBalance()*;      \$200

*b.setBalance(balance\*1.1);*      \$220

*c.withdraw(balance/10)*      \$280



# 并发控制

---

- 不一致检索

初值：帐户 A、B 分别为 \$200、\$200

操作：帐户 A 转 100 到帐户 B

查询银行帐户 A 和 B 的总余额

期望结果：总余额为 \$400



# 不一致检索问题

**W**计算总和的时候  
**V**只完成了取款部分

初值：帐户A、B分别为\$200、\$200

事务V:

*a.withdraw(100)*

*b.deposit(100)*

事务W:

*aBranch.branchTotal()*

*a.withdraw(100);*                      \$100

*total = a.getBalance()*                      \$100

*total = total+b.getBalance()*                      \$300

*total = total+c.getBalance()*

*b.deposit(100)*                      \$300

•  
•





## 串行等价

- 串行等价：如果并发事务交错执行操作的效果等同于按某种次序一次执行一个事务的效果，那么这种交错执行是一种串行等价的交错执行。

T	U
write(i,2);	
	write(i,1);
	x=read(j);
write(j,3);	

- 使用串行等价作为并发执行的判断标准，可防止更新丢失和不一致检索问题。



## 串行等价避免更新丢失

事务T:

```
balance = b.getBalance()
b.setBalance(balance*1.1)
a.withdraw(balance/10)
```

*balance = b.getBalance()*      \$200

*b.setBalance(balance\*1.1)*      \$220

*a.withdraw(balance/10)*      \$80

事务U:

```
balance = b.getBalance()
b.setBalance(balance*1.1)
c.withdraw(balance/10)
```

*balance = b.getBalance()*      \$220

*b.setBalance(balance\*1.1)*      \$242

*c.withdraw(balance/10)*      \$278



## 串行等价避免不一致检索

事务V:

*a.withdraw(100);*  
*b.deposit(100)*

*a.withdraw(100);*            \$100  
*b.deposit(100)*            \$300

事务W:

*aBranch.branchTotal()*

*total = a.getBalance()*            \$100  
*total = total+b.getBalance()*    \$400  
*total = total+c.getBalance()*  
...



## 冲突操作

- 冲突操作：如果两个操作的执行效果和他们的**执行次序相关**，称这两个操作相互冲突（conflict）。
- Read和Write操作的冲突规则

不同事务的操作		是否冲突	原因
read	read	否	由于两个read操作的执行效果不依赖这两个操作的执行次序
read	write	是	由于一个read操作和一个write操作的执行效果依赖于它们的执行次序
write	write	是	由于两个write操作的执行效果依赖于这两个操作的执行次序



## 串行等价的充分必要条件

- 两个事务串行等价的**充要条件**：两个事务中所有的**冲突操作**都按**相同的次序**在它们访问的对象上执行。

T	U
	write(j,1);
write(i,2);	
x=read(j);	
	y=read(i);



## 非串行等价示例

事务T:  $x = \text{read}(i)$ ;  $\text{write}(i, 10)$ ;  $\text{write}(j, 20)$ ;  
事务U:  $y = \text{read}(j)$ ;  $\text{write}(j, 30)$ ;  $z = \text{read}(i)$ ;

事务T:	事务U:
$x = \text{read}(i)$ $\text{write}(i, 10)$  $\text{write}(j, 20)$	$y = \text{read}(j)$ $\text{write}(j, 30)$  $z = \text{read}(i)$



# 是否串行等价?

## 如何交错执行→串行等价?

---

T	U
x=read(i);	
y=read(j);	
	w=read(k);
write(i,1);	
	write(i,3);
	v=read(j);
write(j,2);	
	write(k,4);



# 并发控制

---

- 事务**放弃**相关的两个问题
  - 脏数据读取
  - 过早写入



一个事务读取了另一个**未提交**事务写入的数据

## 事务放弃时的恢复

**U提交后T被放弃**

**U被提交不可能被取消**

### 脏数据读取

某个事务读取了另一个未提交事务写入的数据

事务T:

*a.getBalance()*

*a.setBalance(balance + 10)*

*balance = a.getBalance()*      \$100

*a.setBalance(balance + 10)*      \$110

*abort transaction*

事务U:

*a.getBalance()*

*a.setBalance(balance + 20)*

*balance = a.getBalance()*      **\$110**

*a.setBalance(balance + 20)*      \$130

*commit transaction*

事务T放弃时的脏数据读取



# 事务放弃时的恢复

---

- 事务可恢复性

策略：推迟事务提交，直到它读取更新结果的其它事务都已提交。

- 连锁放弃

- 某个事务的放弃可能导致后续更多事务的放弃

- 防止方法：只允许事务读取**已提交**事务写入的对象

两个事务对同一事物进行write操作

## 事务放弃时的恢复

前镜像进行事务恢复的正确性

- **过早写入**：一些数据库在放弃事务时，将变量的值恢复到该事务所有write操作的“前映像”。
- 如果U提交T放弃，余额应是110，但是T的前映像是100，最终的结果是100的错误值；
- 如果T放弃，接着U放弃，余额应是100，由于U的前映像是105，最终的结果是105的错误值。

事务T:		事务U:	
<i>a.setBalance(105)</i>		<i>a.setBalance(110)</i>	
	\$100		
<i>a.setBalance(105)</i>	\$105	<i>a.setBalance(110)</i>	\$110



## 事务放弃时的恢复

---

- 为了保证使用前映像进行事务恢复时获得正确的结果，write操作必须等到前面修改同一对象的其它事务提交或放弃后才能进行。



# 事务放弃时的恢复

---

- 事务的严格执行

- 严格执行：read和write操作都推迟到写同一对象的其它事务提交或放弃后进行

- 临时版本

- 目的：事务放弃后，能够清除所有对象的更新
- 方法
  - 事务的所有操作更新将值存储在自己的临时版本中
  - 事务提交时，临时版本的数据才会用来更新对象



## 三种并发控制方法

---

### ■ 并发控制

并发控制协议都是基于串行等价的标准，源于用来解决操作冲突的规则。

- 锁
- 乐观并发控制
- 时间戳排序



## 第5章 事务和并发控制

---

- 事务
- 锁
- 乐观并发控制
- 时间戳排序
- 并发控制方法的比较
- 小结



# 锁

---

- 互斥锁是一种简单的事务串行化实现机制
  - 事务访问对象前请求加锁
  - 若对象已被其它事务锁住，则请求被挂起，直至对象被解锁
  - 使用示例





# 锁

## 事务T

```
balance = b.getBalance()  
b.setBalance(bal*1.1)  
a.withdraw(bal/10)
```

操作

锁

*openTransaction*

*bal = b.getBalance()* 锁住B

*b.setBalance(bal\*1.1)*

*a.withdraw(bal/10)* 锁住A

*closeTransaction* 对A,B解锁

## 事务U

```
balance = b.getBalance()  
b.setBalance(bal*1.1)  
c.withdraw(bal/10)
```

操作

锁

*openTransaction*

*bal = b.getBalance()* 等待事务T在B上的锁

... 锁住B

*b.setBalance(bal\*1.1)*

*c.withdraw(bal/10)* 锁住C

*closeTransaction* 对B,C解锁



## 两阶段加锁

---

- 为了保证两个事务的所有冲突操作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
- 每个事务都进行两阶段加锁(two-phase locking)：
  - 在第一个阶段，事务不断地获取新锁（增长阶段）；
  - 在第二个阶段，事务释放它的锁（收缩阶段）
- 目的是防止**不一致检索**和**更新丢失**问题。



## 严格的两阶段加锁

---

- 所有在事务执行过程中获取的新锁必须在事务提交或放弃后才能释放，称为严格的两阶段加锁。
- 为了保证可恢复性，锁必须在所有被更新的对象写入持久存储后才能释放。
- 目的是防止事务放弃导致的脏数据读取和过早写入问题。



## ■ 并发控制使用的粒度

- 如果并发控制同时应用到所有对象，服务器中对象的并发访问将会严重受限
- 如果将所有账户都锁住，任何时候只有一个柜台能够进行联机事务
- 访问必须被串行化的部分对象应尽量少



# 锁

---

- 读锁和写锁（多个读一个写的机制）
  - 支持多个并发事务同时读取某个对象（不同事务对同一对象的read操作不冲突）
  - 允许一个事务写对象（write操作前给对象加写锁）
- 事务的操作冲突规则
  - 如果事务T已经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。
  - 如果事务T已经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。



## 读锁和写锁的相容性

对某一对象		被请求的锁	
		read	write
已设置的锁	none	OK	OK
	read	OK	等待
	write	等待	等待

# 锁

在客户涉及交互程序的情况，死锁很常见，

因为交互程序运行时间较长，造成很多对象被锁住

## 死锁(一)

### ■ 死锁场景示例

两个事务都在等待并且只有对方释放锁后才能继续执行

事务T		事务U	
操作	锁	操作	锁
<i>a.deposit(100);</i>	给A加锁	<i>b.deposit(200)</i>	给B加写锁
<i>b.withdraw(100)</i>	等待事务U	<i>a.withdraw(200);</i>	等待事务T
...	在B上的锁	...	在A上的锁
...		...	
...		...	

# 锁

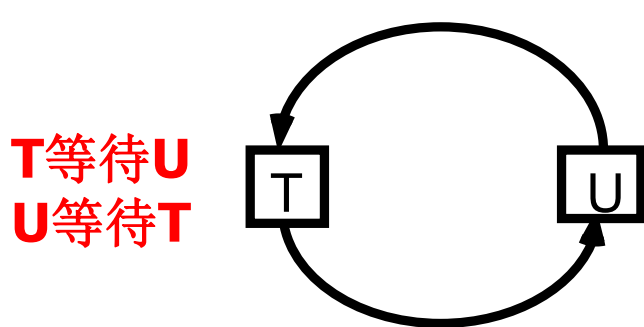
## 死锁(二)

### ■ 定义

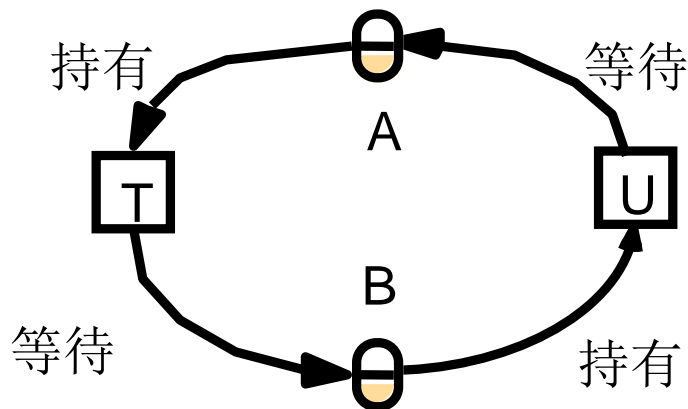
死锁是一种**状态**，在该状态下一组事务中的每一个事务都在等待其它事务释放某个锁。

### ■ 等待图

表示事务之间的等待关系。



T等待U  
U等待T



T持有A等待B  
U持有B等待A





# 锁

---

## ■ 预防死锁

- 每个事务在**开始运行时**锁住它要访问的**所有**对象

- 一个简单的**原子**操作，全有或者全无
- 不必要的资源访问限制
- 无法预计将要访问的对象

浏览型应用允许用户查找事先不知道的对象

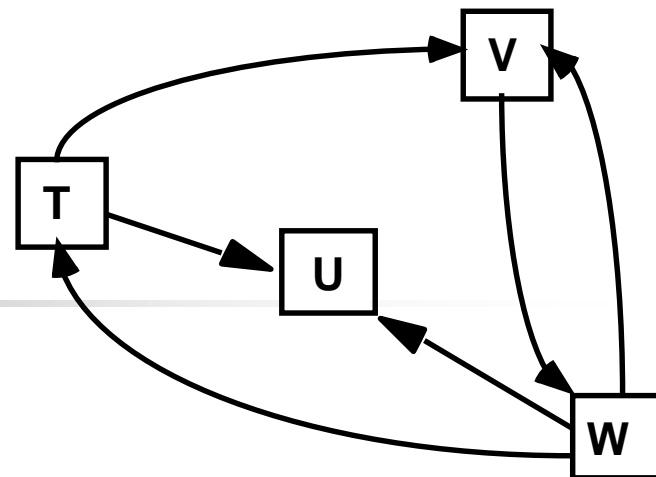
- 预定次序加锁

- 过早加锁
- 减少并发度

# 锁

## ■ 死锁检测

- 维护等待图
  - 检测等待图中是否存在环路
  - 若存在环路，则选择放弃一个事务
  - 414页图16-22
- 
- 如何确定检测的频率？
  - 如何选择放弃的事务？要考虑事务的运行时间和环路的数量





## ■ 锁超时：解除死锁最常用的方法之一

- 每个锁都有一个时间期限
  - 超过时间期限的锁成为可剥夺锁
  - 如果没有其它事务竞争被锁住的对象，那么具有可剥夺锁的对象会被继续锁住
  - 一旦有一个事务正在等待可剥夺锁保护的對象，这个锁将被等待事务剥夺（对象被解锁）。
- 
- 最坏的情况，没有死锁，某些锁变成可剥夺，正好有其它事务请求锁，这些事务被放弃
  - 需要长时间运行的事务经常被放弃



## 第5章 事务和并发控制

---

- 事务
- 锁
- 乐观并发控制
- 时间戳排序
- 并发控制方法的比较
- 小结



# 乐观并发控制

---

## 锁机制的缺点

- 锁的维护开销大
  - 只读事务（查询），不可能改变数据完整性，通常也需要锁保证数据不被其它事务修改，但锁只在最坏的情况下起作用
- 会引起死锁
  - 超时和检测对交互程序都不理想
- 并发度低
  - 为避免连锁放弃，事务结束才释放



## ■ 乐观策略

- 基于事实：在大多数应用中，两个客户事务访问同一个对象的可能性很低。
- 方法
  - 访问对象时不作检查操作
  - 事务提交时检测冲突
  - 若存在冲突，则放弃一些事务



# 乐观并发控制

---

## ■ 事务的三个阶段

### ■ 工作阶段

- 每个事务拥有所修改对象的临时版本
  - 放弃时没有副作用
  - 临时值（写操作）对其他事务不可见
  - 读操作立即执行，如果临时版本存在，则读临时版本，否则访问对象提交的最新值
- 每个事务维护访问对象的两个集合
  - 读集合和写集合



## ■ 验证阶段

- 在收到closeTransaction请求，判断是否与其它事务存在冲突

  - 成功允许提交

  - 失败放弃当前事务或者放弃冲突的事务

## ■ 更新阶段

- 只读事务通过验证立即提交

- 写事务在对象的临时版本记录到持久存储器后提交



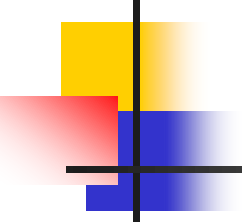


# 乐观并发控制

---

## ■ 事务的验证

- 通过读-写冲突规则确保某个事务的执行对其他重叠事务而言是串行等价的
- 重叠事务是指该事务启动时还没有提交的任何事务

- 
- 每个事务在**进入验证阶段前**被赋予一个事务号
    - 事务号是整数，并按升序分配，定义了事务所处的时间位置
    - 事务按事务号顺序**进入验证阶段**
    - 事务按事务号提交
  - 对事务 $T_v$ 的验证测试是基于 $T_i$ 和 $T_v$ 之间的操作冲突
    - 事务 $T_v$ 对事务 $T_i$ 而言是可串行化的，符合以下规则



## 乐观并发控制

$T_v$	$T_i$	规则
write	read	1. $T_i$ 不能读取 $T_v$ 写的对象
read	write	2. $T_v$ 不能读取 $T_i$ 写的对象
write	write	3. $T_i$ 不能写 $T_v$ 写的对象, 并且 $T_v$ 不能写 $T_i$ 写的对象

每次仅允许一个事务处于验证和更新阶段， $T_v$ 和 $T_i$ 在更新阶段无重叠，规则3自动满足，只需保证 $T_v$ 和 $T_i$ 遵循规则1和2



## ■ 向后验证

- 向后：较早的重叠事务

- 规则1：  $T_i$ 不能读取 $T_v$ 写的对象
  - $T_i$ 读时 $T_v$ 还没有写，自动满足

- 规则2：  $T_v$ 不能读取 $T_i$ 写的对象
  - 需要验证



# 乐观并发控制

检查 $T_v$ 的读集和其它较早重叠事务的写集是否重叠

## ■ 算法

- startTn:  $T_v$ 进入工作阶段时已分配的最大事务号
- finishTn:  $T_v$ 进入验证阶段时已分配的最大事务号

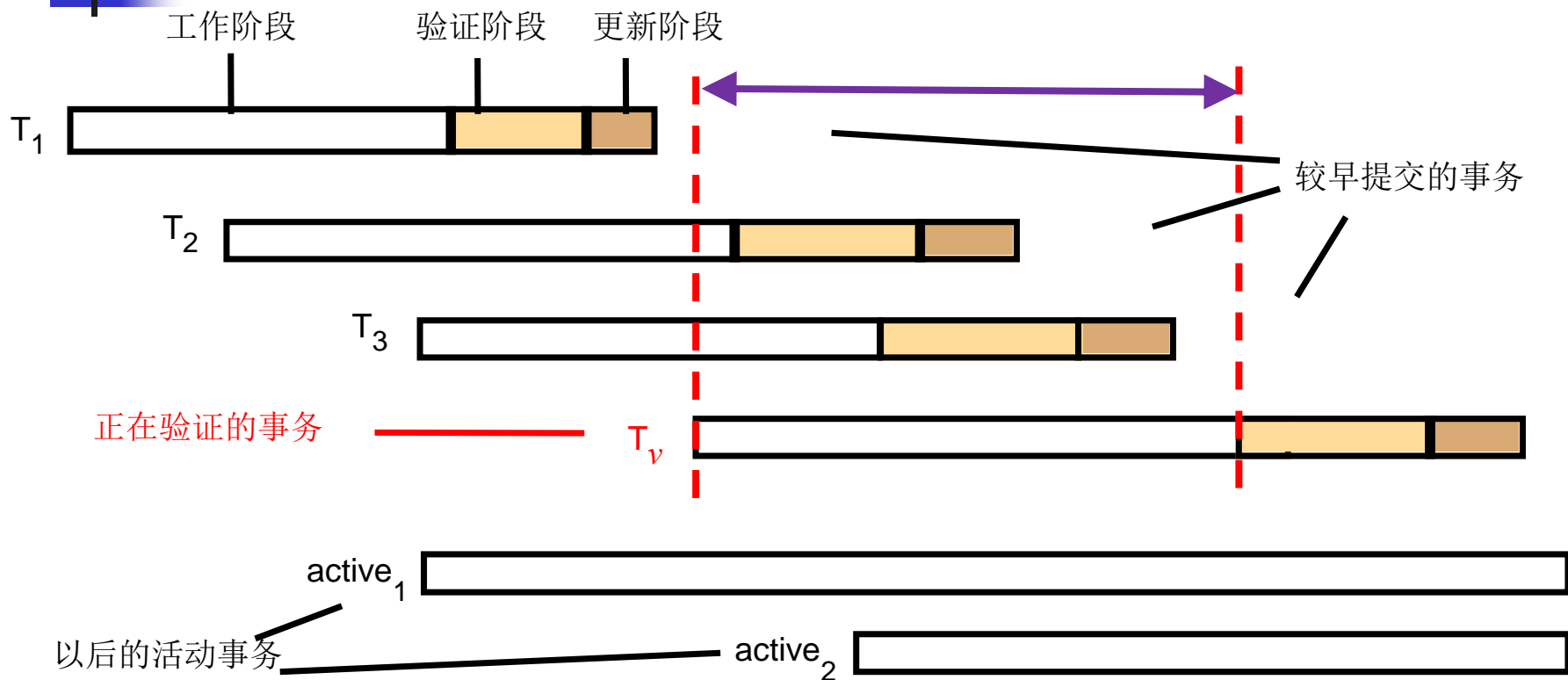
Boolean valid = true

```
For ( int  $T_i = startT_n + 1$ ;  $T_i \leq finishT_n$ ;  $T_i++$ ) {  
    if (read set of  $T_v$  intersects write set of  $T_i$ )  
        valid = false  
}
```

- 验证失败后，冲突解决方法  
 放弃当前进行验证的事务

# 乐观并发控制

$T_v$ 较早的重叠事务 $T_2$ 和 $T_3$



向后验证中， $T_v$ 和谁比较？



# 乐观并发控制

---

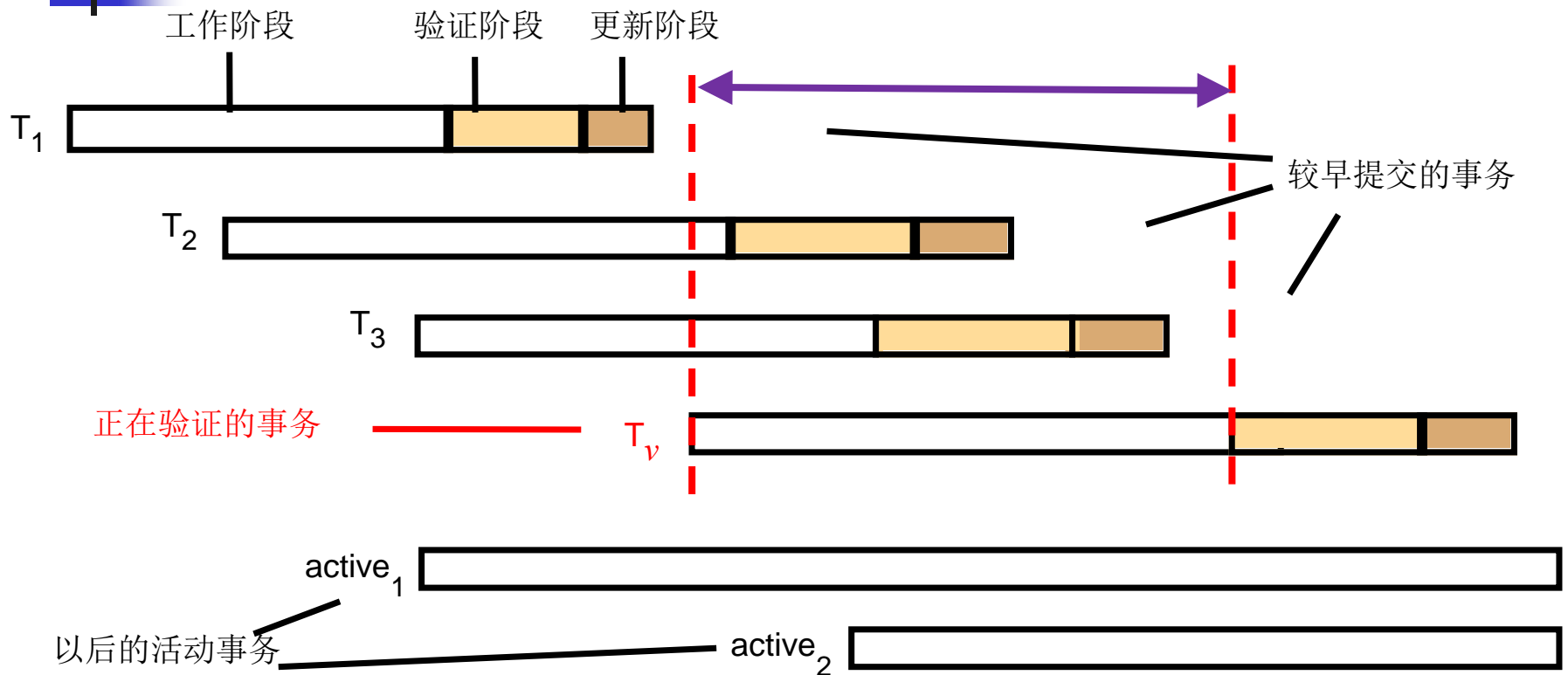
## 向后验证(续)

### ■ 事务的验证过程

- $T_1$ 、 $T_2$ 、 $T_3$ 是较早开始的事务
- $T_1$ 在 $T_v$ 开始之前提交
- $T_2$ 、 $T_3$ 是 $T_v$ 的较早重叠事务
- $\text{start}T_{n+1}=T_2$ ,  $\text{finish}T_n=T_3$

# 乐观并发控制

比较 $T_v$ 的读集和 $T_2$ 、 $T_3$ 的写集



向后验证中，比较的内容？





---

## ■ 向前验证

- 向前：重叠的活动事务

- 规则1：  $T_i$  不能读取  $T_v$  写的对象
  - 需要验证

- 规则2：  $T_v$  不能读取  $T_i$  写的对象
  - 活动事务不会在  $T_v$  完成前写，自动满足



# 乐观并发控制

比较 $T_v$ 的写集合和所有重叠的活动事务的读集合

## ■ 算法

- 设活动事务具有连续的事务标示符 $active_1 \sim active_N$

Boolean valid = true

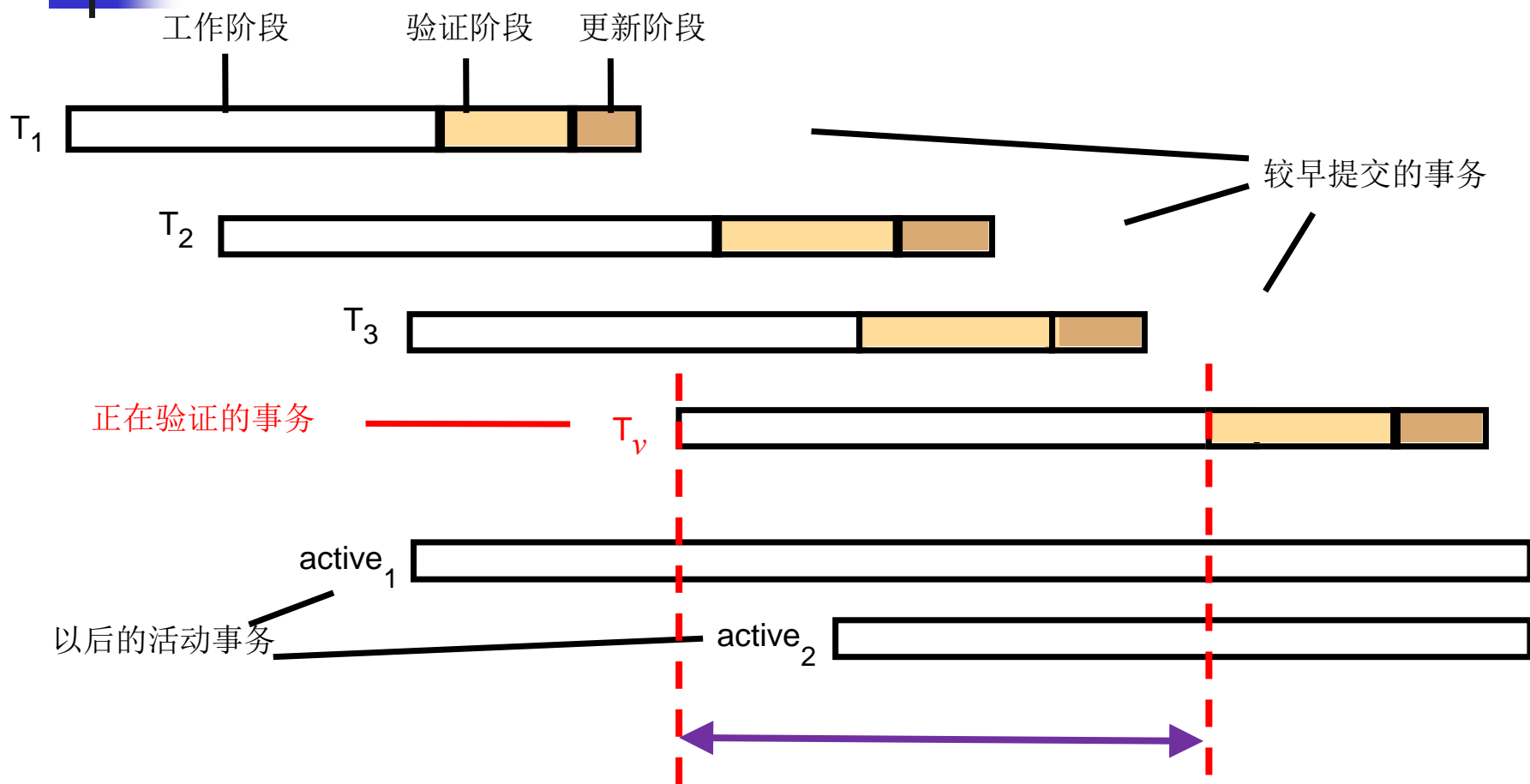
```
for ( int  $T_{id} = active_1$  ;  $T_{id} \leq active_n$  ;  $T_{id}++$  ) {  
    if (write set of  $T_v$  intersects read set of  $T_{id}$ )  
        valid = false  
}
```

## ■ 验证失败后，冲突解决方法

- 放弃当前进行验证事务
- 推迟验证
- 放弃所有冲突的活动事务，提交已验证事务。

# 乐观并发控制

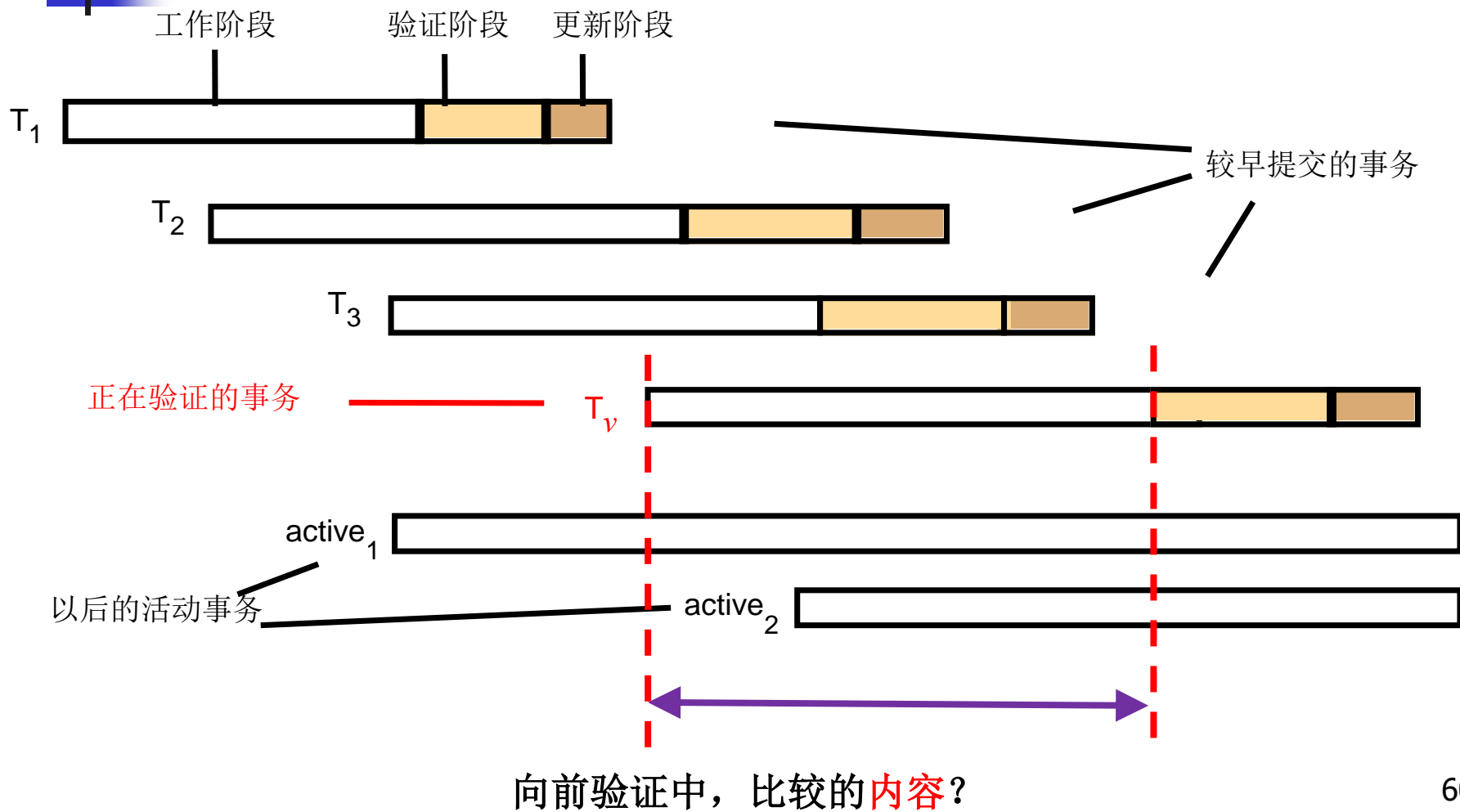
$T_v$ 重叠的活动事务  $active_1$  和  $active_2$



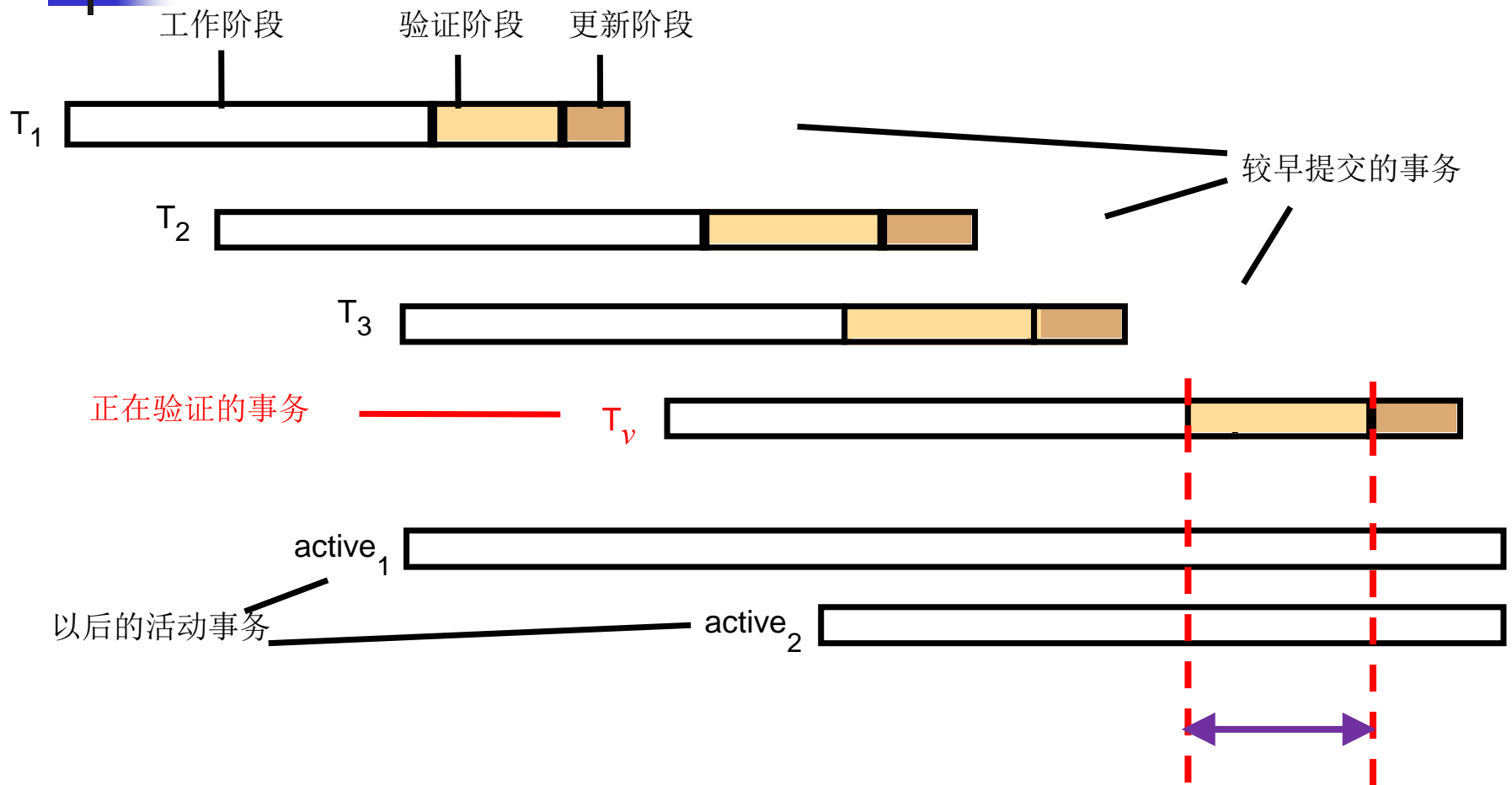
向前验证中， $T_v$ 和谁比较？

# 乐观并发控制

比较 $T_v$ 的写集和 $active_1$ 、 $active_2$ 的读集



# 乐观并发控制



向前验证中，验证阶段是否允许启动新事务？为什么？



# 乐观并发控制

## ■ 向前验证和向后验证的比较

- 向前验证处理冲突时更灵活，向后验证只能选择放弃被验证的事务
- 向后验证将**较大的读集合**和较早事务的写集合进行比较
- 向前验证将**较小的写集合**和活动事务的读集合进行比较
- 向后验证需要存储已提交事务的写集合
- 向前验证不允许在验证过程中开始新的事务

## ■ 饥饿

- 一个事务放弃后会被重启，但不能保证事务最终能通过验证检查
- 利用信号量，实现资源的互斥访问，避免事务饥饿



## 第5章 事务和并发控制

---

- 事务
- 锁
- 乐观并发控制
- 时间戳排序
- 并发控制方法的比较
- 小结



# 时间戳排序

## 基本思想

- 事务中的每个操作在**执行前**先进行验证
- 时间戳
  - 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序列中的位置
- 冲突规则
  - **写请求有效**：对象的最后一次读访问或写访问由一个**较早**（时间戳早）的事务执行。
  - **读请求有效**：对象的最后一次写访问由一个**较早**的事务执行。





# 时间戳排序

## 基于时间戳的并发控制

- 临时版本

- 写操作记录在对象的临时版本中
- 临时版本中的写操作对其它事务不可见

- 对象的写时间戳和读时间戳

- 已提交对象的写时间戳比所有临时版本都要早
- 对象的所有读时间戳可用其中的最大值来代表

规则1和2与 $T_c$ 写操作有关

规则3与 $T_c$ 读操作有关

## 时间戳排序

### 基于时间戳的并发控制(续)

#### ■ 操作冲突

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 $T_c \geq$ 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



# 时间戳排序

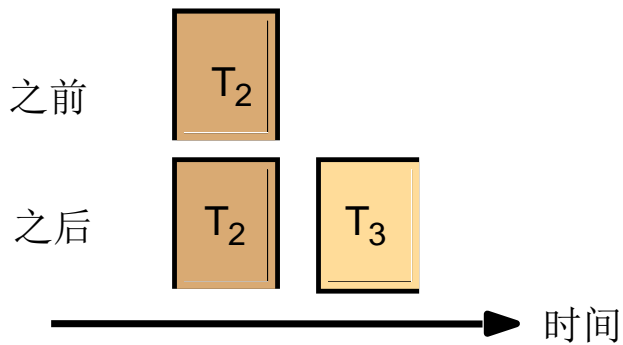
## 时间戳排序的写规则

- 结合规则1和2，决定是否接受事务 $T_c$ 对对象D执行的写操作

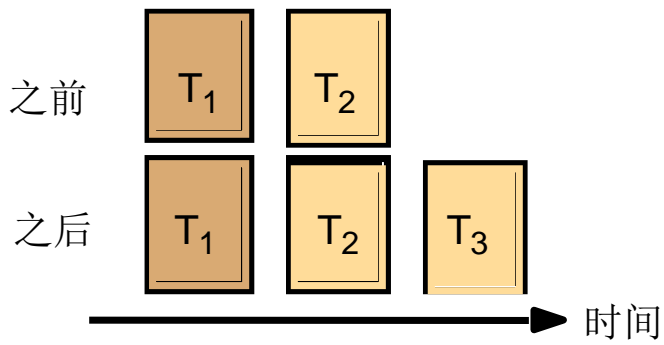
if ( $T_c \geq D$ 的**最大读时间戳** &&  $T_c > D$ 的**提交版本上**的写时间戳)  
    在D的**临时版本上**执行写操作，写时间戳置为 $T_c$   
else /\* 写操作太晚了 \*/  
    放弃事务 $T_c$

- 示例

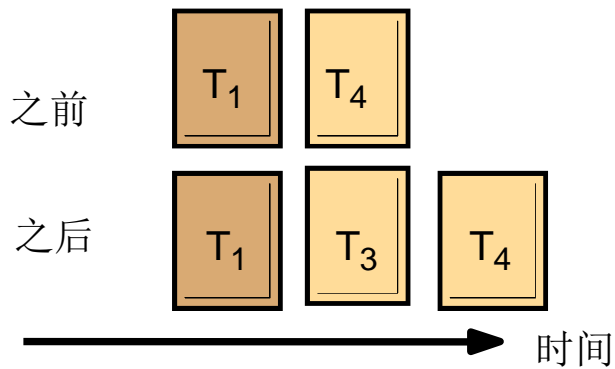
# 时间戳排序



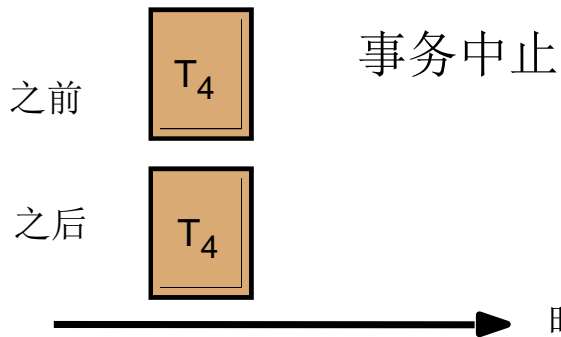
a)  $T_3$  写操作



b)  $T_3$  写操作

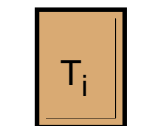


c)  $T_3$  写操作

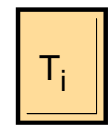


d)  $T_3$  写操作

图例



提交版本



临时版本

时事务 $T_i$ 产生的对象  
(写时间戳为 $T_i$ )

$T_1 < T_2 < T_3 < T_4$



# 时间戳排序

## 时间戳排序的读规则

- 应用规则3，决定是否接受事务 $T_c$ 对对象D执行的读操作

if ( $T_c > D$ 提交版本的写时间戳)

  设 $D_{selected}$ 是D的具有最大写时间戳的版本 $\leq T_c$ ;

  if ( $D_{selected}$ 已提交)

    在 $D_{selected}$ 版本上完成读操作

  else

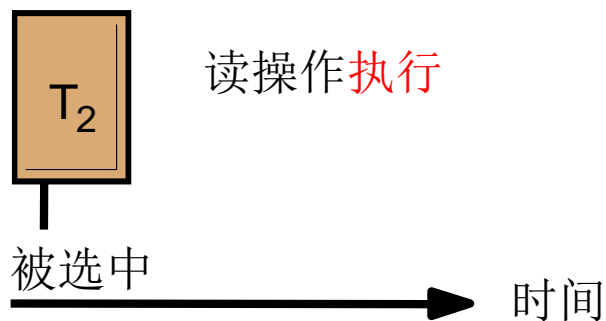
    等待直到形成 $D_{selected}$ 版本的事务提交或放弃，然后重新应用读规则;

} else

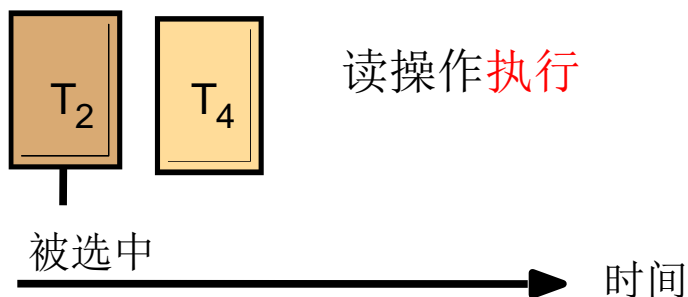
  放弃事务 $T_c$

- 示例

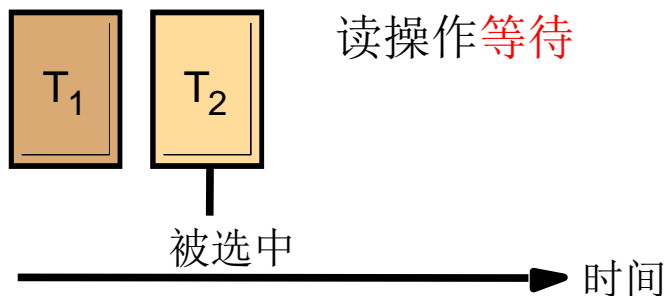
# 时间戳排序



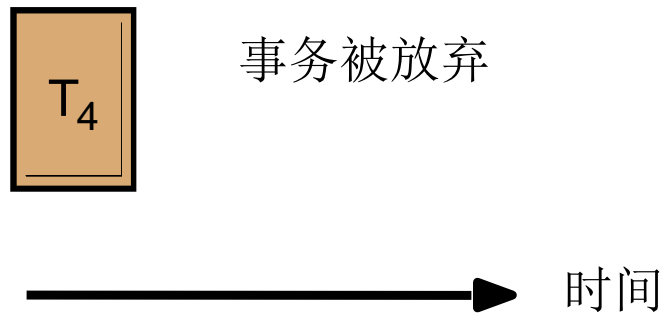
(a)  $T_3$  读操作



(b)  $T_3$  读操作

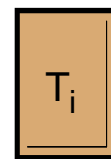


(c)  $T_3$  读操作

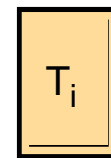


(d)  $T_3$  读操作

图例:



提交版本



临时版本

时事务 $T_i$ 产生的对象  
(写时间戳为 $T_i$ )

$T_1 < T_2 < T_3 < T_4$

# 利用时间戳排序控制并发事务示例

T	U	a		b		c	
		RTS	WTS	RTS	WTS	RTS	WTS
		{}	S	{}	S	{}	S
openTransaction							
bal=b.getBalance()				{T}			
	openTransaction						
b.setBalance(bal*1.1)					S,T		
	bal=b.getBalance()		选中T (最大写时间戳版本)				
	wait for T		T未提交, 需要等待				
a.withdraw(bal/10)	...		S,T				
commit	...		T		T		
	bal=b.getBalance()			{U}			
	b.setBalance(bal*1.1)				T,U		
	c.withdraw(bal/10)						S,U



## 第5章 事务和并发控制

---

- 事务
- 锁
- 乐观并发控制
- 时间戳排序
- 并发控制方法的比较
- 小结





# 并发控制方法的比较

---

- 两阶段加锁
  - 事务执行中
- 乐观并发控制
  - 事务执行后
- 时间戳排序
  - 事务执行前



# 并发控制方法的比较

---

- 悲观方法

- 两阶段加锁
- 时间戳排序

- 乐观方法

- 乐观并发控制



# 并发控制方法的比较

---

## ■ 两阶段加锁

- 写较多事务优于时间戳排序
- 只在死锁时放弃
- 容易死锁

## ■ 时间戳排序

- 读较多事务优于两阶段加锁
- 到来较晚的事务必须被放弃

## ■ 乐观并发控制

- 执行过程中不进行检测，提交前必须经过验证
- 不能通过验证的事务不断放弃，可能引发饥饿

# 小结

## ■ 事务

## ■ 并发控制

- 准则：串行等价性及其充要条件
- 读已提交：更新丢失和不一致检索
- 读未提交（放弃）：脏数据读取、过早写入

## ■ 并发控制方法

- 两阶段加锁
- 乐观并发控制
- 时间戳排序