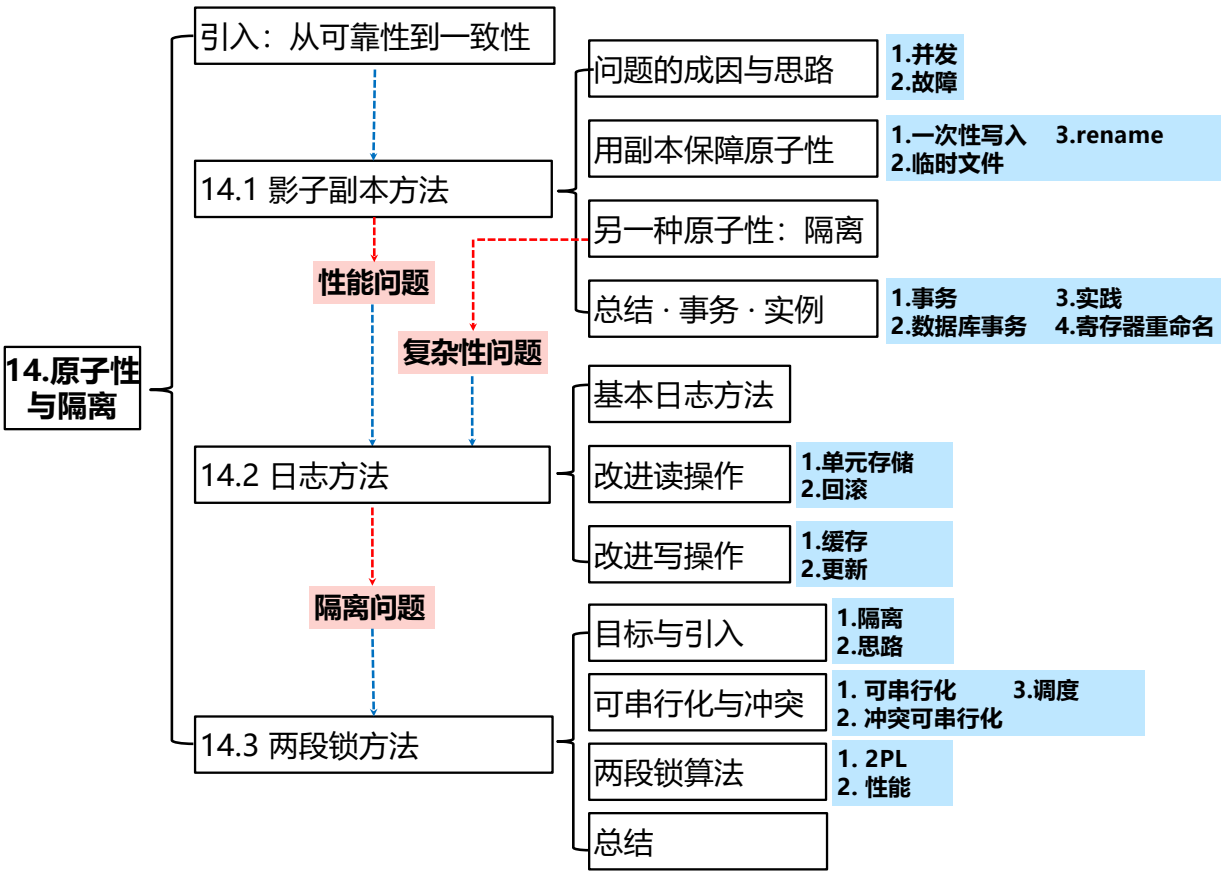
14. 原子性与隔离

1

本章相关的参考文献

- Gray J, McJones P, Blasgen M, et al. The recovery manager of the System R database manager[J]. ACM Computing Surveys (CSUR), 1981, 13(2): 223-242.
- C. Mohan et al. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. ACM Transactions on Database Systems 17, 1 (1992), 94–162.
- Lomet D. B. Process structuring, synchronization, and recovery using atomic actions. Proceedings of an ACM Conference on Language Design for Reliable Software (March 1977), 128–137. Published as ACM SIGPLAN Notices 12, 3 (March 1977); Operating Systems Review 11, 2 (April 1977); and Software Engineering Notes 2, 2 (March 1977).



多选题 1分 设置

交叠可以减少每个请求的时延

A 正确

B 错误

交叠可以减少批量请求的时延

C 正确

D 错误

只要资源的利用率不是100%，那么队列的长度就存在上限。

E 正确

F 错误

提交

填空题 2分 设置

为了提高飞机的可靠性，设计师决定使用3个引擎的设计方案。但由于引擎的重量很大，使得3引擎飞机必须有2个引擎同时工作方可正常飞行。

假设单个引擎的MTTF=6000小时。3个引擎的故障是相互独立的。

请慎重考虑后回答以下问题：

1.两个飞机的平均无故障运行时间MTTF，时间较短的是 [填空1]引擎飞机。

2.如果你要出行，会选择 [填空2] 引擎飞机 。

两个空请填写：单 或 3 。

作答

从可靠性到一致性

计算机系统工程的重要目标：

- 用不可靠的组件构建可靠、高效、正确的系统，即：
- 在高可用性的前提下：
  - 服务更多用户，存储更大数据量，提供更快速度 → **第12章 性能**

并发、流水线 ...

可靠性工程

- 识别错误、检测错误、处理错误，防止失效发生
- 逐步构造通用的抽象模块，逐层构造可靠性

第3章 命名
第4章 模块化
第5章 虚拟化

→ **第13章 可靠性**

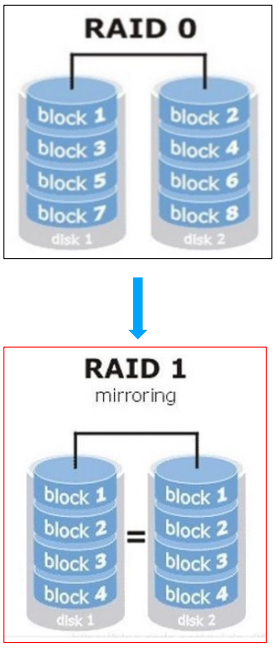
检错、纠错、冗余 ...

一致性目标

模块间行为是否有正确性的约束? → **第14章 原子性与隔离**

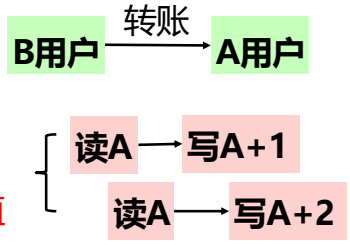
影子副本、日志、两段锁 ...

从可靠性到一致性：实例



问题：RAID能够保证存储过程可靠。但如果RAID上层的存储系统出现问题呢?

- 例：
  - **转账** A：0→1 && B：1→0
    - 操作只完成一半
  - **并发** A = A+1； A = A+2;
    - 语句2读到了旧值、覆盖了新值



为了保证一致性，提出了原子性和隔离：

- 保证存储系统多个连续操作的结果一致性。

两种应对思路

原子性： **All-or-Nothing** **属性**

- 屏蔽错误

隔离： **Before-or-After** **属性**

- 协调并发活动

在第5章、第13章已有涉及具体事例

本章系统性地讨论原子性、隔离两种属性

原子性与隔离：3种方案

计算机系统工程导论 2024

14.1 影子副本方法 (shadow copy)

问题分析

1. **并发问题引发隔离问题**

2. **故障问题引发原子性问题**

并发与隔离

- a转账给b（任务1），　　同时b转账给c（任务2）

```
1 transfer (bank, account_a, account_b, amount):
2     bank[account_a] = bank[account_a] - amount
3     bank[account_b] = bank[account_b] + amount
```

```
1 transfer (bank, account_b, account_c, amount):
2     bank[account_b] = bank[account_b] - amount
3     bank[account_c] = bank[account_c] + amount
```

CPU时间

```
1    tmp1 = bank[account_a] - amount
2    bank[account_a] = tmp1
3    tmp1 = bank[account_b] + amount
4    bank[account_b] = tmp1
```

```
1    tmp2 = bank[account_b] - amount
2    bank[account_b] = tmp2
3    tmp2 = bank[account_c] + amount
4    bank[account_c] = tmp2
```

故障

失效!

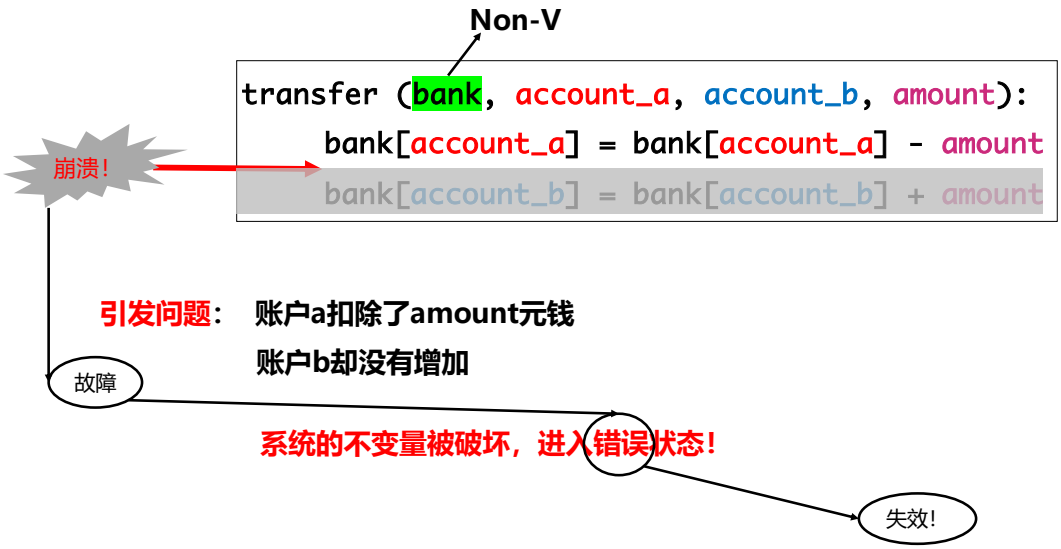
引发问题： **账户b入账操作被覆盖，导致总计被扣款amount**

账户a扣款、c入账，均正常

系统的不变量被破坏，进入错误状态!

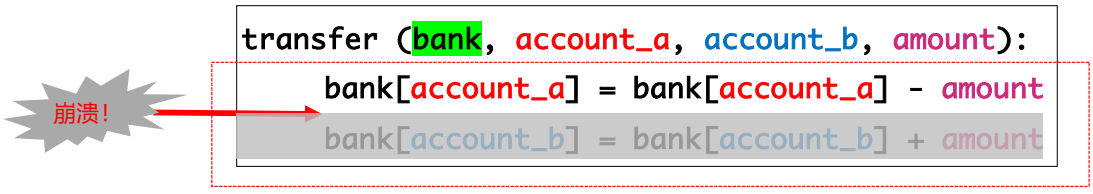
故障与原子性

- a转账给b



原子性问题的思路

- a转账给b

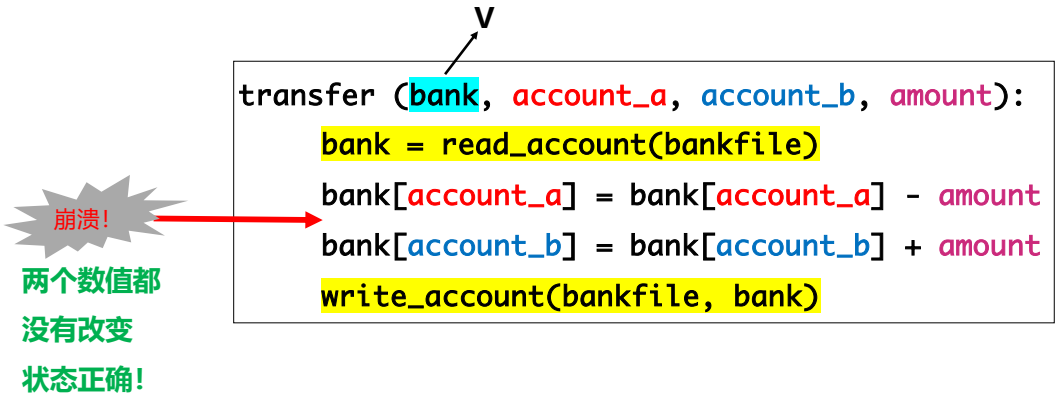


引发问题: 账户a扣除了amount元钱
账户b却没有增加

系统的不变量被破坏，进入错误状态!

解决方案: 如果系统使两个操作或全做或全不做，那么这个组件就可防止此错误。
如何修改transfer的代码?

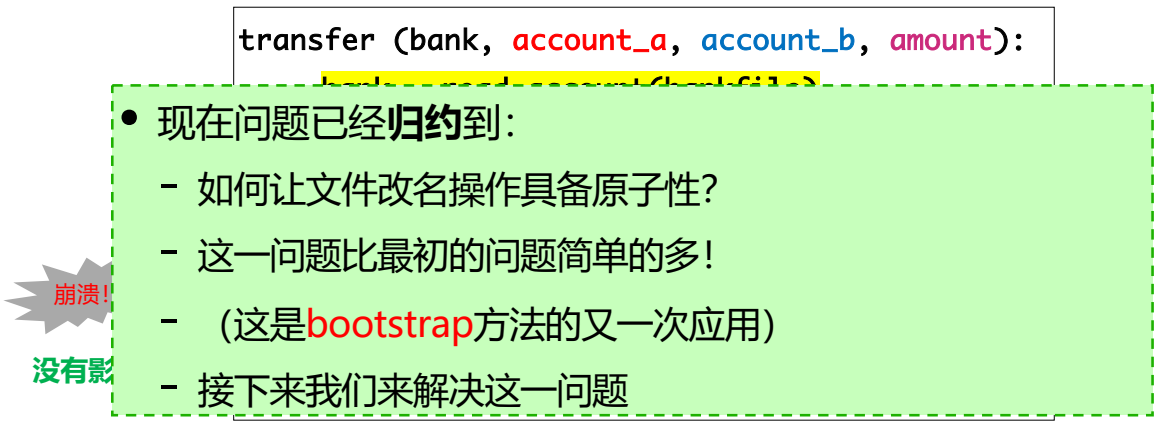
代码改进1：一次性写入



思路： 将结果写入一个文件。如果中途崩溃，两个数值都不会改变。

新问题：写入bankfile文件时候如果崩溃，bankfile将处于不确定的状态!

代码改进2：使用临时文件



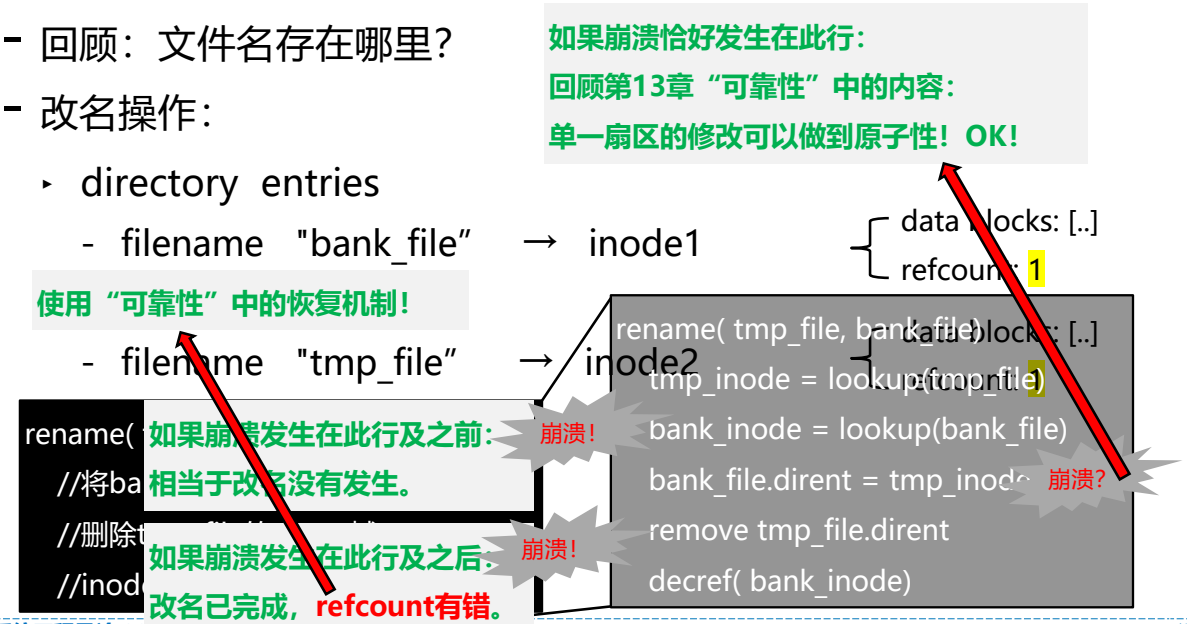
新问题：写入bankfile文件时候如果崩溃，bankfile将处于不确定的状态!

新思路：写文件变成“写临时文件”+“改名”操作，改名操作保证原子性即可。

又有新问题：改名操作的原子性问题!

回顾rename

- 第3章“命名”部分讲过的UNIX文件结构
  - 回顾：文件名存在哪里？
  - 改名操作：
    - directory entries
      - filename "bank_file" → inode1
      - filename "tmp_file" → inode2



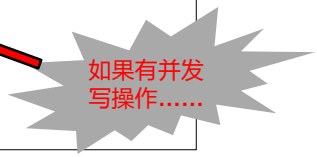
恢复机制：存储垃圾回收

```
recover(disk):
    for inode in disk.inodes:
        inode.refcount = find_all_refs(dis.root_dir,inode)
    if exists(tmp_file)
        unlink(tmp_file)
```

通过恢复机制，当崩溃发生时，虽然状态存在错误，
但可以清除这些错误！

另一个问题：隔离

```
transfer (bank, account_a, account_b, amount):
    bank = read_account(bankfile)
    bank[account_a] = bank[account_a] - amount
    bank[account_b] = bank[account_b] + amount
    write_account(tmp_file,bank)
    rename(tmp_file,bankfile)
```



解决隔离问题的目标是：

使两个并发操作串行，即使它们并发出现。

锁与隔离

用锁机制保护共享内存的隔离性

- acquire / release （lock）

```
transfer (bank, account_a, account_b, amount):
    acquire(lock)
    bank = read_account(bankfile)
    bank[account_a] = bank[account_a] - amount
    bank[account_b] = bank[account_b] + amount
    write_account(tmp_file,bank)
    rename(tmp_file,bankfile)
    release(lock)
```

锁机制可以解决这个并发问题。　　但在此场景下会大大降低性能！

锁的使用，需要在全局进行考虑，在适当的场景使用。

总结

能够保证两种原子性的抽象，称为：

- 事务处理（transaction processing） （通用概念）

事务处理的抽象

- 更好地寻求多模块组合的可靠性

方案总结与评价

- 原子性： 已经有了一种效率不高的方案
- 隔离： 已经有了一种不完善的方案

数据库事务

1970年，在事务处理基础上，Jim Gray在数据库领域定义了一个可靠的事务系统的属性，缩写为ACID：

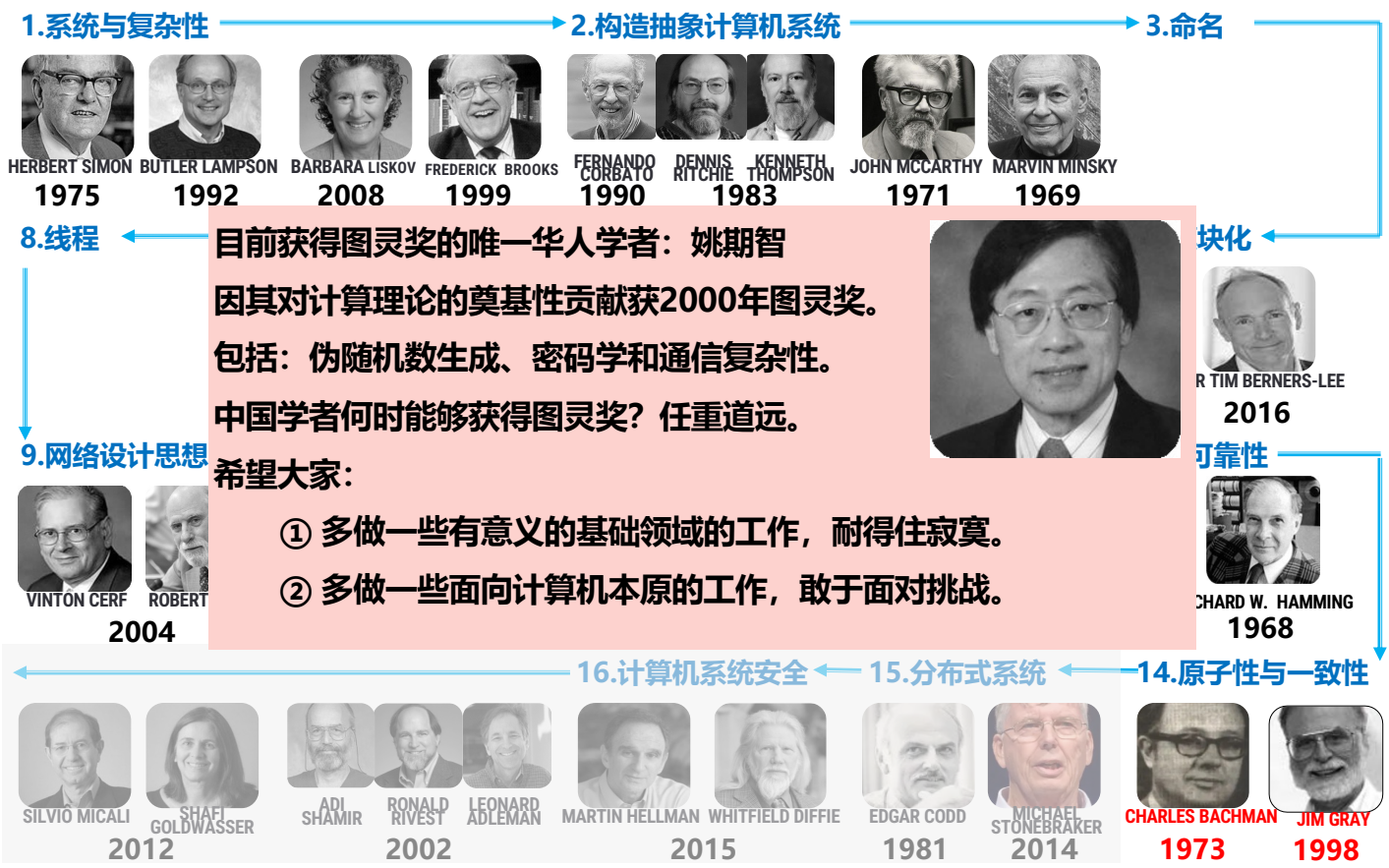
Atomicity（原子性）

Consistency（一致性）

Isolation（隔离性）

Durability（持续性）

Jim Gray（1998年图灵奖获得者）



原子性与隔离的实例

领域	原子性	隔离
数据库管理	更新1条以上记录	多线程操作同一条记录
硬件体系结构	处理中断或异常	寄存器重命名
操作系统	系统调用接口	打印队列
软件工程	处理下层错误	有界缓冲区的操作

寄存器重命名

流水线导致相邻指令可能并发

寄存器重命名（隔离）示例：

寄存器名：r1, r2, r3　　寄存器单元：l1, l2, l3, l4, l5, l6, l7

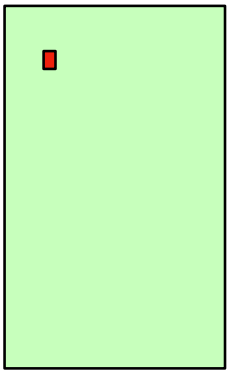
原始指令	r1	r2	r3	重命名后
	l1	l2	l3	
add r1, r2, r3	l4	l2	l3	add l4, l2, l3
sub r3, r2, r1	l4	l2	l5	sub l5, l2, l4
mul r1, r2, r3	l6	l2	l5	mul l6, l2, l5
div r2, r1, r3	l6	l7	l5	div l7, l6, l5

为什么要重命名?

交叠（pipeline）

14.2 日志方法（logging）

影子副本的性能问题



```
transfer (bank, account_a, account_b, amount):
      bank = read_account(bankfile)
      bank[account_a] = bank[account_a] - amount
      bank[account_b] = bank[account_b] + amount
      write_account(tmp_file,bank)
      rename(tmp_file,bankfile)
```

无论多小的修改，都需要复制整个bankfile，修改文件后，再改名文件！

假设文件大小为1G，而修改的记录为1K?

锁的复杂性问题

```
transfer
  acquire(lock)
    ……

  //错误使用lock


  release(lock)
```

```
transfer (bank, account_a, account_b, amount):
      acquire(lock)
      bank = read_account(bankfile)
      bank[account_a] = bank[account_a] - amount
      bank[account_b] = bank[account_b] + amount
      write_account(tmp_file,bank)
      rename(tmp_file,bankfile)
      release(lock)
```

```
transfer
  acquire(lock)
    ……

  //死锁、活锁
 // a、b锁分开?

  release(lock)
```

锁需要做全局的分析，了解每一处访问，并设置不同的锁！

假设文件有1000种数据，每处访问是2-10种数据的组合?

日志方法

目标：原子性

收益：性能

成本：复杂度

日志方法的思路：记录下这些操作，通过某种方式来确保这些操作组成完成的“事务”！

```
begin          //T1
write( A, 100)
write( B, 50)
commit         //A=100, B=50

begin          //T2
write( A, read(A)-20)
write( B, read(B)+20)
commit         //A=80, B=70

begin          //T3
write( A, read(A)+30)
```

没有comit，需要确保A的值必须是80！而非110

日志方法

```
begin          //T1
write( A, 100)
write( B, 50)
commit         //A=100, B=50

begin          //T2
write( A, read(A)-20)
write( B, read(B)+20)
commit         //A=80, B=70

begin          //T3
write( A, read(A)+30)
```

真的可以提供原子性吗?

日志方法
与影子副本不同
只写入改变的值

唯一

TID	T1	T1	T1	T2	T2	T2	T3
Type	UPDATE	UPDATE	COMMIT	UPDATE	UPDATE	COMMIT	UPDATE
Var	A	B		A	B		A
Old	A=0	B=0		A=100	B=50		A=80
New	A=100	B=50		A=80	B=70		A=110

日志方法：读取

```
# 从日志中获得A的值
read( log, var):
  commits = [ ]
  # 向后扫描
  for record r in log[len(log)-1] .. log[0]:
  # 处理commits
  if r.type == COMMIT:
  commits.add(r.tid)
  # 找到var的最后一次提交的值
  elif r.type == UPDATE and \
    r.tid in commits and r.var == var:
      return r.new_value  # 此处退出
```

```
begin          //T1
write( A, 100)
write( B, 50)
commit         //A=100, B=50

begin          //T2
write( A, read(A)-20)
write( B, read(B)+20)
commit         //A=80, B=70

begin          //T3
write( A, read(A)+30)
```

commits=[ ]

commits=[T1]

TID	T1	T1	T1	T2	T2	T2	T3
	UPDATE	UPDATE	COMMIT	UPDATE	UPDATE	COMMIT	UPDATE
Old	A=0	B=0		A=100	B=50		A=80
New	A=100	B=50		A=80	B=70		A=110

日志方法：事务内读取

```
# 从日志中获得A的值
read( log, var):
  commits = [ ]
  # 向后扫描
  for record r in log[len(log)-1] .. log[0]:
  # 处理commits
  if r.type == COMMIT:
  commits.add(r.tid)
  # 找到var的最后一次提交的值
  elif r.type == UPDATE and \
    r.tid in commits \
    and r.var == var:
      return r.new_value  # 此处退出
```

```
begin          //T1
write( A, 100)
write( B, 50)
commit         //A=100, B=50

begin          //T2
write( A, read(A)-20)
write( A, read(A)-30)
commit         //A=50, B=50

begin          //T3
write( A, read(A)+30)
```

TID	T1	T1	T1	T2	T2	T2	T3
	UPDATE	UPDATE	COMMIT	UPDATE	UPDATE	COMMIT	UPDATE
Old	A=0	B=0		A=100	A=80		A=50
New	A=100	B=50		A=80	A=50		A=80

日志方法：事务内读取

```
# 从日志中获得A的值
read( log, var):
  commits = [ ]
  # 向后扫描
  for record r in log[len(log)-1] .. log[0]:
  # 处理commits
  if r.type == COMMIT:
  commits.add(r.tid)
  # 找到var的最后一次提交的值
  elif r.type == UPDATE and \
   (r.tid in commits or r.tid ==current_tid )\
    and r.var == var:
      return r.new_value  # 此处退出
```

```
begin          //T1
write( A, 100)
write( B, 50)
commit         //A=100, B=50

begin          //T2
write( A, read(A)-20)
write( A, read(A)-30)
commit         //A=50, B=50

begin          //T3
write( A, read(A)+30)
```

TID	T1	T1	T1	T2	T2	T2	T3
	UPDATE	UPDATE	COMMIT	UPDATE	UPDATE	COMMIT	UPDATE
Old	A=0	B=0		A=100	A=80		A=50
New	A=100	B=50		A=80	A=50		A=80

日志方法

```
# 从日志中获得A的值
read( log, var):
  commits = [ ]
  # 向后扫描
  for record r in log[len(log)-1] .. log[0]:
  # 处理commits
  if r.type == COMMIT:
  commits.add(r.tid)
  # 找到var的最后一次提交的值
  elif r.type == UPDATE and \
    (r.tid in commits or r.tid ==current_tid )\
    and r.var == var:
      return r.new_value  # 此处退出
```

```
begin          //T1
write( A, 100)
write( B, 50)
commit         //A=100, B=50

begin          //T2
write( A, read(A)-20)
write( B, read(B)+20)
commit         //A=80, B=70

begin          //T3
write( A, read(A)+30)
```

TID	T1	T1	T1	T2	T2	T2	T3
	UPDATE	UPDATE	COMMIT	UPDATE	UPDATE	COMMIT	UPDATE
Old	A=0	B=0		A=100	B=50		A=80
New	A=100	B=50		A=80	B=70		A=110

日志方法

```
# 从日志中获得A的值
read( log, var):
  commits = [ ]
  # 向后扫描
  for record r in log[len(log)-1] .. log[0]:
  # 处理commits
  if r.type == COMMIT:
  commits.add(r.tid)
  # 找到var的最后一次提交的值
  elif r.type == UPDATE and \
    (r.tid in commits or r.tid ==current_tid )\
    and r.var == var:
      return r.new_value  # 此处退出
```

崩溃后，log保持正确！

崩溃！

```
begin          //T1
write( A, 100)
write( B, 50)
commit         //A=100, B=50

begin          //T2
write( A, read(A)-20)
write( B, read(B)+20)
commit         //A=80, B=70

begin          //T3
write( A, read(A)+30)
```

TID	T1	T1	T1	T2	T2	T2	T3
	UPDATE	UPDATE	COMMIT	UPDATE	UPDATE	COMMIT	UPDATE
Old	A=0	B=0		A=100	B=50		A=80
New	A=100	B=50		A=80	B=70		A=110

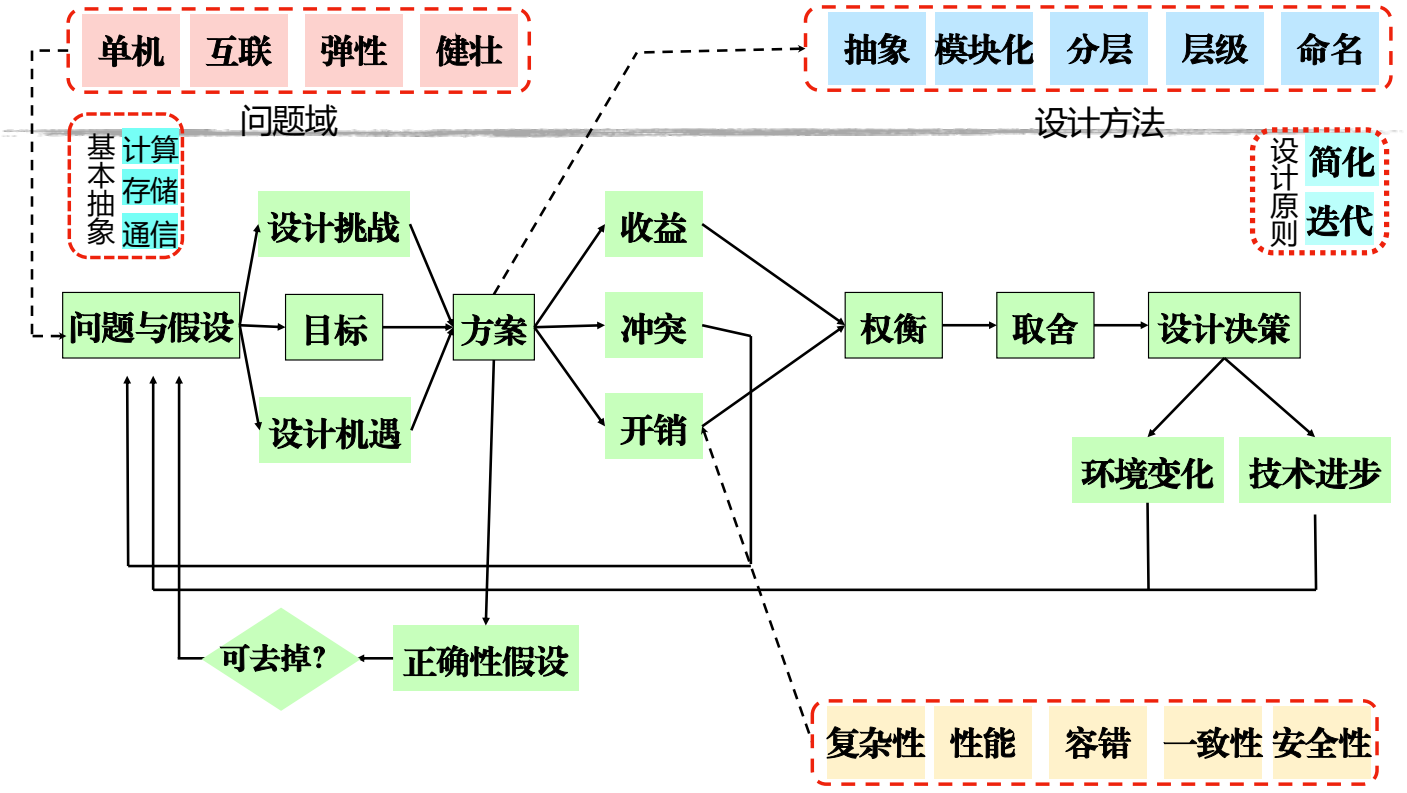
改进读操作

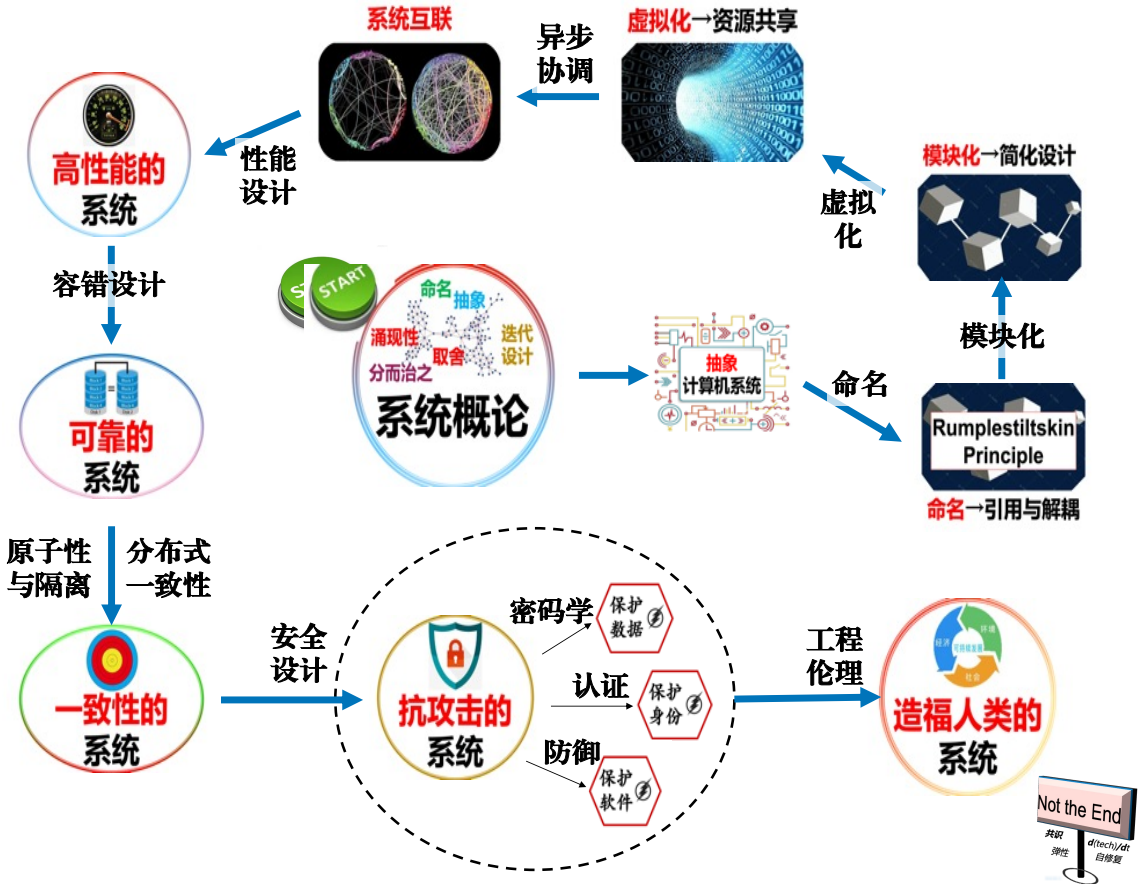
新问题：读操作很慢

```
begin          //T1
write( A, 100)
write( B, 50)
commit         //A=100, B=50

begin          //T2
write( A, read(A)-20)
write( B, read(B)+20)
commit         //A=80, B=70

begin          //T3
write( A, read(A)+30)
```

TID	T1	T1	T1	T2	T2	T2	T3
	UPDATE	UPDATE	COMMIT	UPDATE	UPDATE	COMMIT	UPDATE
Old	A=0	B=0		A=100	B=50		A=80
New	A=100	B=50		A=80	B=70		A=110


第14章（上）结束


单机
互联
弹性
健壮
问题域
抽象
模块化
分层
层级
命名
设计方法
基本抽象
计算
存储
通信
设计原则
简化
迭代
设计挑战
收益
问题与假设
目标
方案
冲突
权衡
取舍
设计决策
设计机遇
开销
环境变化
技术进步
可去掉?
正确性假设
复杂性
性能
容错
一致性
安全性
系统属性

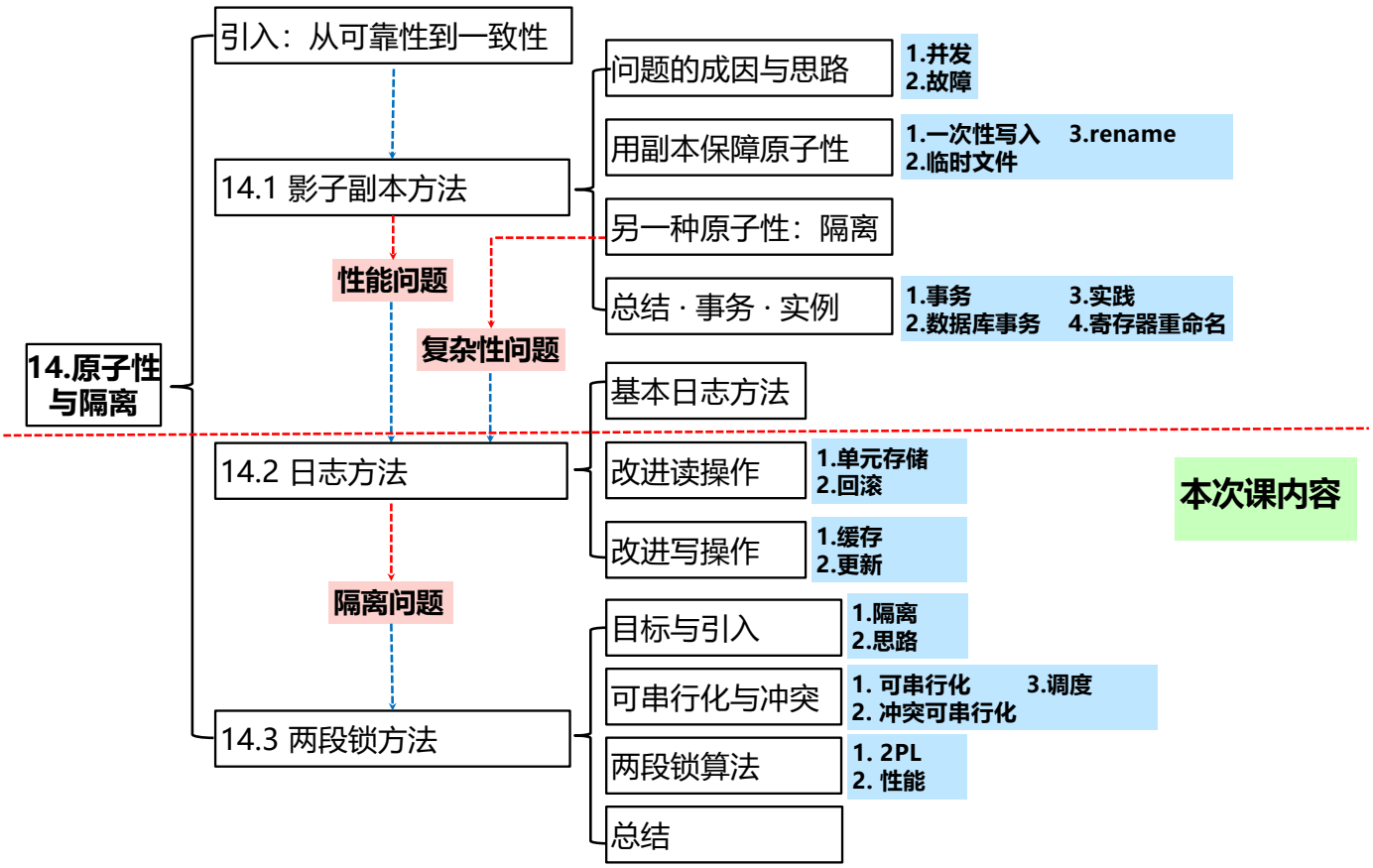


计算机系统相关新闻

Hospital System Ascension Hit by Cyberattack, Takes Systems Offline

EMS has reportedly been directed to bypass Ascension-owned hospitals, while doctors have reverted to paper records.

"On Wednesday, May 8, we detected unusual activity on select technology network systems, which we now believe is due to a cyber security event," Ascension says. An investigation is ongoing, but "access to some systems have been interrupted as this process continues." Ascension also reports a "disruption to clinical operations."



回顾

1. **提出了能够保证两种原子性的抽象：**
   - 事务处理（transaction processing）
2. **提出了2种方案**
   - 原子性：
     - 已经有了一种效率不高的方案：影子副本
     - 已经有了改善读写量的方案：日志方法，但读操作效率仍有问题
   - 隔离：
     - 已经有了一种不完善的方案：互斥锁

单选题 1分

设置

思考

按照之前我们在虚拟化、性能、可靠性等章节学习的算法和工程方法，你认为后续我们可能会用哪种方法来改进读的性能?

A **通过并发来提高吞吐率**

B **通过虚拟化来保持接口**

C **通过交叠（流水线）来减少总体时延**

D **寻找设计中的机会，读性能因为是朴素设计所以尚未达到设计最优**

提交

用单元存储改进读操作

新问题：读操作很慢

单元存储 cell storage

A 110	B 70

```
begin          //T1
write( A, 100)
write( B, 50)
commit         //A=100, B=50

begin          //T2
write( A, read(A)-20)
write( B, read(B)+20)
commit         //A=80, B=70

begin          //T3
write( A, read(A)+30)
```

TID	T1	T1	T1	T2	T2	T2	T3
Old New	UPDATE A=0 A=100	UPDATE B=0 B=50	COMMIT	UPDATE A=100 A=80	UPDATE B=50 B=70	COMMIT	UPDATE A=80 A=110

读操作

```
read( var):
  return cell_read( var)
```

单元存储 | A 110 | B 70 |

要求：写入值时，必须将单元存储更新到最新

```
begin          //T1
write( A, 100)
write( B, 50)
commit         //A=100, B=50

begin          //T2
write( A, read(A)-20)
write( B, read(B)+20)
commit         //A=80, B=70

begin          //T3
write( A, read(A)+30)
```

TID	T1	T1	T1	T2	T2	T2	T3
	UPDATE	UPDATE	COMMIT	UPDATE	UPDATE	COMMIT	UPDATE
Old	A=0	B=0		A=100	B=50		A=80
New	A=100	B=50		A=80	B=70		A=110

写操作

```
read( var):
  return cell_read( var)

write( var, value):
  log.append( \
    current_tid, “UPDATE”, \
    var, read(var), value)
  cell_write(var, value)
```

单元存储 | A 110 | B 70 |

```
begin          //T1
write( A, 100)
write( B, 50)
commit         //A=100, B=50

begin          //T2
write( A, read(A)-20)
write( B, read(B)+20)
commit         //A=80, B=70

begin          //T3
write( A, read(A)+30)
```

崩溃！

要求：具备恢复能力

TID	T1	T1	T1	T2	T2	T2	T3
	UPDATE	UPDATE	COMMIT	UPDATE	UPDATE	COMMIT	UPDATE
Old	A=0	B=0		A=100	B=50		A=80
New	A=100	B=50		A=80	B=70		A=110

崩溃恢复操作

```
read( var):
  return cell_read( var)
write( var, value):
  log.append( current_tid, “UPDATE”, var, read(var), value)
  cell_write(var, value)
recover( log):
  commits = [ ]
  for record r in log[len(log)-1] .. log[0]:
    if r.type == COMMIT:
      commits.add( r.tid)
    if r.type == UPDATE and r.tid not in commits:
      cell_write( r.var, r.old_val)  #undo
```


示例

```
read( var):
  return cell_read( var)
write( var, value):
  log.append( current_tid, “UPDATE”, var, read(var), value)
  cell_write(var, value)
recover( log):
  commits = [ ]
  for record r in log[len(log)-1] .. log[0]:
    if r.type == COMMIT:
      commits.add( r.tid)
    if r.type == UPDATE and r.tid not in commits:
      cell_write( r.var, r.old_val)  #undo
```

commits=[ ]

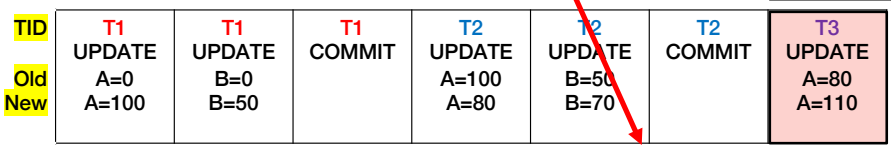
TID	T1	T1	T1	T2	T2	T2	T3
	UPDATE	UPDATE	COMMIT	UPDATE	UPDATE	COMMIT	UPDATE
Old	A=0	B=0		A=100	B=50		A=80
New	A=100	B=50		A=80	B=70		A=110

cell storage A 110 B 70

回滚（rollback）

```
read( var):
  return cell_read( var)
write( var, value):
  log.append( current_tid, “UPDATE”, var, read(var), value)
  cell_write(var, value)
recover( log):
  commits = [ ]
  for record r in log[len(log)-1] .. log[0]:
    if r.type == COMMIT:
      commits.add( r.tid)
    if r.type == UPDATE and r.tid not in commits:
      cell_write( r.var, r.old_val)  #undo
```

commits=[ ]



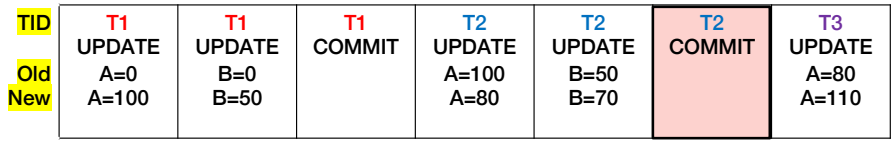
cell storage A 80 B 70

防止进一步回滚

```
read( var):
  return cell_read( var)
write( var, value):
  log.append( current_tid, “UPDATE”, var, read(var), value)
  cell_write(var, value)
recover( log):
  commits = [ ]
  for record r in log[len(log)-1] .. log[0]:
    if r.type == COMMIT:
      commits.add( r.tid)
    if r.type == UPDATE and r.tid not in commits:
      cell_write( r.var, r.old_val)  #undo
```

commits=[T2]

commits=[ ]



cell storage A 80 B 70

防止进一步回滚

```
read( var):
  return cell_read( var)
write( var, value):
  log.append( current_tid, “UPDATE”, var, read(var), value)
  cell_write(var, value)
recover( log):
  commits = [ ]
  for record r in log[len(log)-1] .. log[0]:
    if r.type == COMMIT:
      commits.add( r.tid)
    if r.type == UPDATE and r.tid not in commits:
      cell_write( r.var, r.old_val)  #undo
```

思考：如果在恢复过程中再次崩溃，算法是否正确？

commits=[T2,T1]

commits=[T2]

commits=[ ]

TID	T1	T1	T1	T2	T2	T2	T3
	UPDATE	UPDATE	COMMIT	UPDATE	UPDATE	COMMIT	UPDATE
Old	A=0	B=0		A=100	B=50		A=80
New	A=100	B=50		A=80	B=70		A=110

cell storage A 80 B 70

为什么可以多次恢复?

```
read( var):
  return cell_read( var)
write( var, value):
  log.append( current_tid, “UPDATE”, var, read(var), value)
  cell_write(var, value)
recover( log):
  commits = [ ]
  for record r in log[len(log)-1] .. log[0]:
    if r.type == COMMIT:
      commits.add( r.tid)
    if r.type == UPDATE and r.tid not in commits:
      cell_write( r.var, r.old_val)  #undo
```

TID	T1	T1	T1	T2	T2	T2	T3
	UPDATE	UPDATE	COMMIT	UPDATE	UPDATE	COMMIT	UPDATE
Old	A=0	B=0		A=100	B=50		A=80
New	A=100	B=50		A=80	B=70		A=110

cell storage A 80 B 70

写性能问题

```
read( var):
  return cell_read( var)
write( var, value):
  log.append( current_tid, “UPDATE”, var, read(var), value)
  cell_write(var, value)
recover( log):
```

读操作的性能已改善

写操作的性能怎么改进?

```
    commits.add( r.tid)
  if r.type == UPDATE and r.tid not in commits:
    cell_write( r.var, r.old_val)  #undo
```

TID	T1	T1	T1	T2	T2	T2	T3
	UPDATE	UPDATE	COMMIT	UPDATE	UPDATE	COMMIT	UPDATE
Old	A=0	B=0		A=100	B=50		A=80
New	A=100	B=50		A=80	B=70		A=110

cell storage | A 80 | B 70

改进写操作：缓存

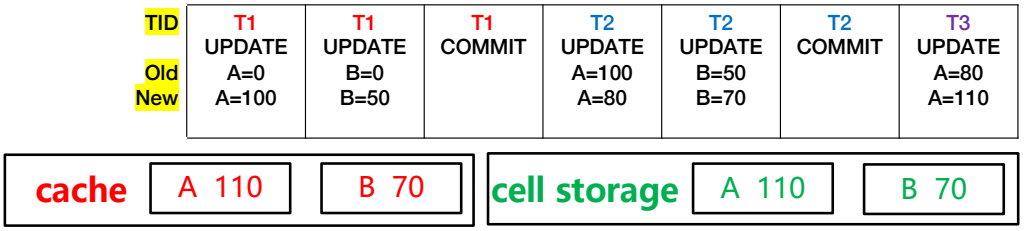
TID	T1	T1	T1	T2	T2	T2	T3
	UPDATE	UPDATE	COMMIT	UPDATE	UPDATE	COMMIT	UPDATE
Old	A=0	B=0		A=100	B=50		A=80
New	A=100	B=50		A=80	B=70		A=110

cache | A 110 | B 70

cell storage | A 110 | B 70

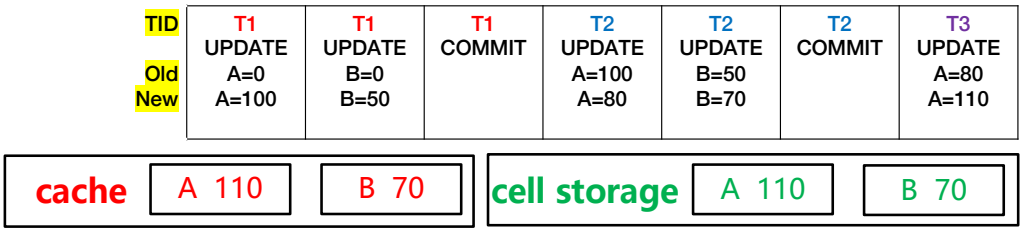
改进写操作：读

```
read( var):
  if var in cache:
    return cache[var]
  else:
    # 可能需要载入并驱逐其他数据到cell存储
    cache[var] = cell_read(var)
    return cache[var]
```



改进写操作：写

```
read( var):
  if var in cache:
    return cache[var]
  else:
    # 可能需要载入并驱逐其他数据到cell存储
    cache[var] = cell_read(var)
    return cache[var]

write( var, value):
  log.append( current_tid, update, var, read(var), value)
  cache[var] = value
```



改进写操作：刷新

```
read( var):
  if var in cache:
    return cache[var]
  else:
    # 可能需要载入并驱逐其他数据到cell存储
    cache[var] = cell_read(var)
    return cache[var]

write( var, value):
  log.append( current_tid, update, var, read(var), value)
  cache[var] = value

flush( ):  #偶尔调用
  cell_write( var, cache[var]) for each var
```

崩溃怎么办?

TID	T1	T1	T1	T2	T2	T2	T3
	UPDATE	UPDATE	COMMIT	UPDATE	UPDATE	COMMIT	UPDATE
Old	A=0	B=0		A=100	B=50		A=80
New	A=100	B=50		A=80	B=70		A=110

cache	A 110	B 70

cell storage	A 110	B 70

改进写操作：恢复

```
read( var):
  if var in cache:
    return cache[var]
  else:
    # 可能需要载入并驱逐其他数据到cell存储
    cache[var] = cell_read(var)
    return cache[var]

write( var, value):
  log.append( current_tid, update, var, read(var), value)
  cache[var] = value

flush( ):  #偶尔调用
  cell_write( var, cache[var]) for each var
```

```
recover( log):
  commits = [ ]
  for record r in log[len(log)-1] .. log[0]:
    if r.type == COMMIT:
      commits.add( r.tid)
    if r.type == UPDATE and r.tid not in commits:
      cell_write( r.var, r.old_var) #undo
```

TID	T1	T1	T1	T2	T2	T2	T3
	UPDATE	UPDATE	COMMIT	UPDATE	UPDATE	COMMIT	UPDATE
Old	A=0	B=0		A=100	B=50		A=80
New	A=100	B=50		A=80	B=70		A=110

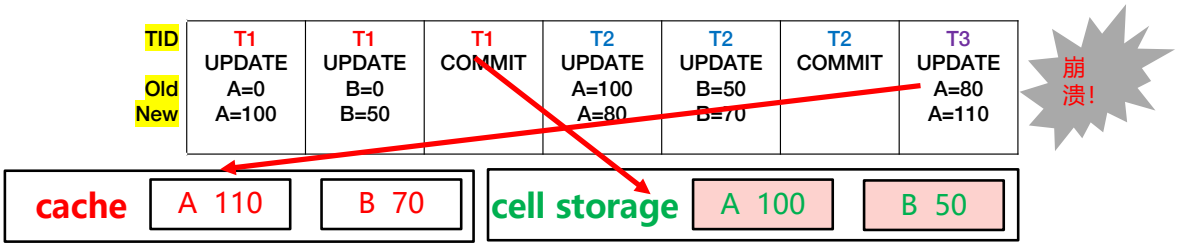
cache	A 110	B 70

cell storage	A 110	B 70

示例

```
read( var):
  if var in cache:
    return cache[var]
  else:
    # 可能需要载入并驱逐其他数据到cell存储
    cache[var] = cell_read(var)
    return cache[var]

write( var, value):
  log.append( current_tid, update, var, read(var), value)
  cache[var] = value

flush( ):  #偶尔调用
  cell_write( var, cache[var]) for each var
```

```
recover( log):
  commits = [ ]
  for record r in log[len(log)-1] .. log[0]:
    if r.type == COMMIT:
      commits.add( r.tid)
    if r.type == UPDATE and r.tid not in commits:
      cell_write( r.var, r.old_var) #undo
```

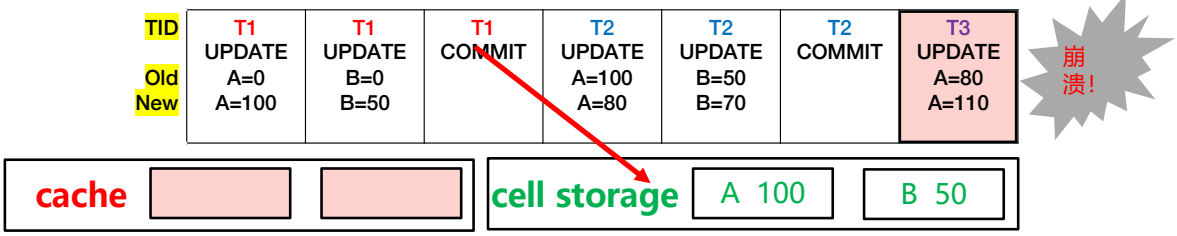


示例：扫描日志

```
read( var):
  if var in cache:
    return cache[var]
  else:
    # 可能需要载入并驱逐其他数据到cell存储
    cache[var] = cell_read(var)
    return cache[var]

write( var, value):
  log.append( current_tid, update, var, read(var), value)
  cache[var] = value

flush( ):  #偶尔调用
  cell_write( var, cache[var]) for each var
```

```
recover( log):
  commits = [ ]
  for record r in log[len(log)-1] .. log[0]:
    if r.type == COMMIT:
      commits.add( r.tid)
    if r.type == UPDATE and r.tid not in commits:
      cell_write( r.var, r.old_var) #undo
```

commits=[ ]



撤销（undo）

```
read( var):
  if var in cache:
    return cache[var]
  else:
    # 可能需要载入并驱逐其他数据到cell存储
    cache[var] = cell_read(var)
    return cache[var]

write( var, value):
  log.append( current_tid, update, var, read(var), value)
  cache[var] = value

flush( ):  #偶尔调用
  cell_write( var, cache[var]) for each var
```

```
recover( log):
  commits = [ ]
  for record r in log[len(log)-1] .. log[0]:
    if r.type == COMMIT:
      commits.add( r.tid)
    if r.type == UPDATE and r.tid not in commits:
      cell_write( r.var, r.old_var) #undo
```

commits=[ ]

TID	T1	T1	T1	T2	T2	T2	T3
	UPDATE	UPDATE	COMMIT	UPDATE	UPDATE	COMMIT	UPDATE
Old	A=0	B=0		A=100	B=50		A=80
New	A=100	B=50		A=80	B=70		A=110

崩溃!

cache | cell storage: A 80 | B 50

数据不一致问题

```
read( var):
  if var in cache:
    return cache[var]
  else:
    # 可能需要载入并驱逐其他数据到cell存储
    cache[var] = cell_read(var)
    return cache[var]

write( var, value):
  log.append( current_tid, update, var, read(var), value)
  cache[var] = value

flush( ):  #偶尔调用
  cell_write( var, cache[var]) for each var
```

```
recover( log):
  commits = [ ]
  for record r in log[len(log)-1] .. log[0]:
    if r.type == COMMIT:
      commits.add( r.tid)
    if r.type == UPDATE and r.tid not in commits:
      cell_write( r.var, r.old_var) #undo
```

B没有被写入！如何恢复?

commits=[ ]

TID	T1	T1	T1	T2	T2	T2	T3
	UPDATE	UPDATE	COMMIT	UPDATE	UPDATE	COMMIT	UPDATE
Old	A=0	B=0		A=100	B=50		A=80
New	A=100	B=50		A=80	B=70		A=110

崩溃!

cache | cell storage: A 80 | B 50

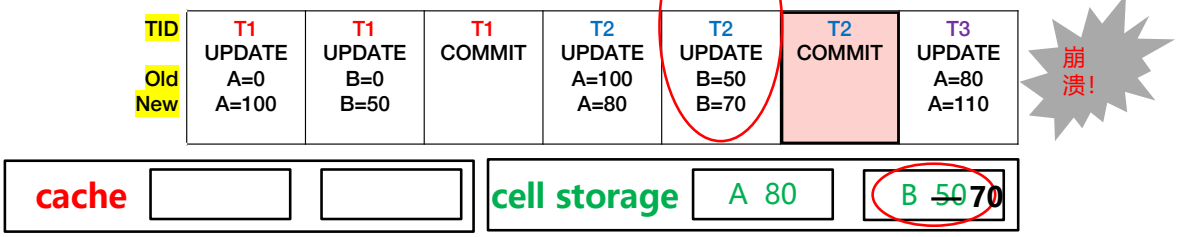
重做（redo）

```
read( var):
  if var in cache:
    return cache[var]
  else:
    # 可能需要载入并驱逐其他数据到cell存储
    cache[var] = cell.read(var)
```

```
recover( log):
  commits = [ ]
  for record r in log[len(log)-1] .. log[0]:
    if r.type == COMMIT:
      commits.add( r.tid)
    if r.type == UPDATE and r.tid not in commits:
```

- 现在读、写操作的性能问题都已经得到改善。
- 但是恢复操作很耗时。
- 如何改进？？？

写入检查点，丢弃一部分日志（未来学习内容）



多选题 1分 设置

使用影子副本方法，为减少大文件读写，会采用：

- A 压缩存储
- B 临时文件
- C 并发

使用日志方法，为加快写操作，会采用：

- D cache
- E 流水线
- F 并发

提交

总结

能够保证两种原子性的抽象 →

- 事务处理（transaction processing）
- 在事务处理抽象之上，可以更好地实施可靠性

方案评价

- 原子性：　使用日志得到了更好的方案
- 隔离：　只有不完善的方案

14.3 两段锁方法（2PL）

回顾

能够保证两种原子性的抽象 →

- 事务处理（transaction processing）（通用概念）

在事务处理抽象之上，我们可以更好地实施可靠性

方案评价

- 原子性： 使用日志得到了更好的方案
- 隔离： 不完善的方案

- 将介绍两段锁（two-phase lock）方法

目标：隔离

并发运行多个事务T1...TN，令执行结果与串行相同

与串行相同的含义?

T1	T2
begin	begin
T1.1 read(x)	T2.1 write(x,20)
T1.2 tmp = read(y)	T2.2 write(y,30)
T1.3 write(y,tmp+10)	commit
commit	

问题描述：

事务内部的顺序是可保证的。

例：T1.1在T1.2前。

但两个进程可能有交迭。

例：T2发生在T1.1与T1.2之间。

直接、有效、也是低效的方法：使用锁（甚至全局锁），使操作实际上串行化！

目标：隔离　　举例——结果对比

T1	T2
begin	begin
T1.1 read(x)	T2.1 write(x,20)
T1.2 tmp = read(y)	T2.2 write(y,30)
T1.3 write(y,tmp+10)	commit
commit	

串行1

```
T1.1 read(x)
T1.2 tmp = read(y)
T1.3 write(y,tmp+10)
T2.1 write(x,20)
T2.2 write(y,30)

结果: x=20; y=30
```

串行2

```
T2.1 write(x,20)
T2.2 write(y,30)
T1.1 read(x)
T1.2 tmp = read(y)
T1.3 write(y,tmp+10)

结果: x=20; y=40
```

T1与T2的一些不同的并发执行顺序（尚未穷举）：

并发1

```
T2.1 write(x,20)
T1.1 read(x)
T2.2 write(y,30)
T1.2 tmp = read(y)
T1.3 write(y,tmp+10)

结果: x=20; y=40
```

并发2

```
T1.1 read(x)
T2.1 write(x,20)
T1.2 tmp = read(y)
T2.2 write(y,30)
T1.3 write(y,tmp+10)

结果: x=20; y=10
```

并发3

```
T1.1 read(x)
T2.1 write(x,20)
T2.2 write(y,30)
T1.2 tmp = read(y)
T1.3 write(y,tmp+10)

结果: x=20; y=40
```

第二个执行顺序的结果：x=20; y=10 与两个串行方式的结果都不同！

目标：隔离　　举例——结果对比

T1	T2
begin	begin
T1.1 read(x)	T2.1 write(x,20)
T1.2 tmp = read(y)	T2.2 write(y,30)
T1.3 write(y,tmp+10)	commit
commit	

串行1

```
T1.1 read(x)
T1.2 tmp = read(y)
T1.3 write(y,tmp+10)
T2.1 write(x,20)
T2.2 write(y,30)

结果: x=20; y=30
```

T1读到的x为0，y为0

串行2

```
T2.1 write(x,20)
T2.2 write(y,30)
T1.1 read(x)
T1.2 tmp = read(y)
T1.3 write(y,tmp+10)

结果: x=20; y=40
```

T1读到的x为20，y为30

T1与T2的一些不同的并发执行顺序（尚未穷举）：

并发1

```
T2.1 write(x,20)
T1.1 read(x)
T2.2 write(y,30)
T1.2 tmp = read(y)
T1.3 write(y,tmp+10)

结果: x=20; y=40
```

并发2

```
T1.1 read(x)
T2.1 write(x,20)
T1.2 tmp = read(y)
T2.2 write(y,30)
T1.3 write(y,tmp+10)

结果: x=20; y=10
```

并发3

```
T1.1 read(x)
T2.1 write(x,20)
T2.2 write(y,30)
T1.2 tmp = read(y)
T1.3 write(y,tmp+10)

结果: x=20; y=40
```

T1读到的
x为0
y为30

✍ 不同的可串行化要求

可串行化不只有一种定义

使用不同的定义取决于应用的需求

✍ 可串行化 一定 正确吗?

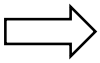
更强的正确性要求

1. 外部时间一致性
   - 以银行业为例：外部事件的时序，必须反映到操作上

2. 顺序一致性
   - 以CPU为例：并发执行某指令流时，应与程序按原始顺序执行结果相同

冲突：如两个操作访问同一对象，且至少一个操作是写操作，那么称这两个操作存在冲突（conflict）。

T1	T2
begin	begin
T1.1 read(x)	T2.1 write(x,20)
T1.2 tmp = read(y)	T2.2 write(y,30)
T1.3 write(y,tmp+10)	commit
commit	



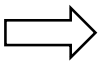
冲突
T1.1 read(x) and T2.1 write(x,20)
T1.2 tmp = read(y) and T2.2 write(y,30)
T1.3 write(y,tmp+10) and T2.2 write(y,30)

思考：为什么需要这样的定义？（回忆第6章）

在调度中，冲突的一对操作的先后顺序（A早于B或B早于A），称之为该调度中该冲突的序。

调度：

T1.1 read(x)
T1.2 tmp = read(y)
T1.3 write(y,tmp+10)
T2.1 write(x,20)
T2.2 write(y,30)

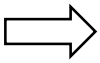


该调度中冲突的序：

冲突的序
T1.1 read(x) → T2.1 write(x,20)
T1.2 tmp = read(y) → T2.2 write(y,30)
T1.3 write(y,tmp+10) → T2.2 write(y,30)

观察串行执行冲突的序

T1	T2
begin	begin
T1.1 read(x)	T2.1 write(x,20)
T1.2 tmp = read(y)	T2.2 write(y,30)
T1.3 write(y,tmp+10)	commit
commit	



冲突
T1.1 read(x) and T2.1 write(x,20)
T1.2 tmp = read(y) and T2.2 write(y,30)
T1.3 write(y,tmp+10) and T2.2 write(y,30)

观察串行执行时冲突的序：或T1的操作总在前，或T2的操作总在前。

T1.1 read(x)
T1.2 tmp = read(y)
T1.3 write(y,tmp+10)
T2.1 write(x,20)
T2.2 write(y,30)

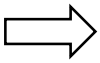
冲突的序
T1.1 → T2.1
T1.2 → T2.2
T1.3 → T2.2

T2.1 write(x,20)
T2.2 write(y,30)
T1.1 read(x)
T1.2 tmp = read(y)
T1.3 write(y,tmp+10)

冲突的序
T2.1 → T1.1
T2.2 → T1.2
T2.2 → T1.3

观察并发执行冲突的序

T1	T2
begin	begin
T1.1 read(x)	T2.1 write(x,20)
T1.2 tmp = read(y)	T2.2 write(y,30)
T1.3 write(y,tmp+10)	commit
commit	



冲突

T1.1 read(x) and T2.1 write(x,20)

T1.2 tmp = read(y) and T2.2 write(y,30)

T1.3 write(y,tmp+10) and T2.2 write(y,30)

1

T2.1 write(x,20)
T1.1 read(x)
T2.2 write(y,30)
T1.2 tmp = read(y)
T1.3 write(y,tmp+10)

冲突的序

T2.1 → T1.1

T2.2 → T1.2

T2.2 → T1.3

此执行与T2→T1序相同

2

T1.1 read(x)
T2.1 write(x,20)
T2.2 write(y,30)
T1.2 tmp = read(y)
T1.3 write(y,tmp+10)

冲突的序

T1.1 → T2.1

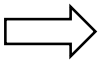
T2.2 → T1.2

T2.2 → T1.3

此执行与两个序都不同

冲突图

T1	T2
begin	begin
T1.1 read(x)	T2.1 write(x,20)
T1.2 tmp = read(y)	T2.2 write(y,30)
T1.3 write(y,tmp+10)	commit
commit	



冲突

T1.1 read(x) and T2.1 write(x,20)

T1.2 tmp = read(y) and T2.2 write(y,30)

T1.3 write(y,tmp+10) and T2.2 write(y,30)

将冲突的序用冲突图进行表示：Ti到Tj有一条边 **当且仅当** Ti与Tj存在冲突，且Ti先执行。

T2.1 write(x,20)
T1.1 read(x)
T2.2 write(y,30)
T1.2 tmp = read(y)
T1.3 write(y,tmp+10)

冲突的序

T2.1 → T1.1

T2.2 → T1.2

T2.2 → T1.3

此调度是冲突可串行化的

T1 ← T2

T1.1 read(x)
T2.1 write(x,20)
T2.2 write(y,30)
T1.2 tmp = read(y)
T1.3 write(y,tmp+10)

冲突的序

T1.1 → T2.1

T2.2 → T1.2

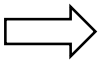
T2.2 → T1.3

此调度不是冲突可串行化的

T1 ⇄ T2

判定法则

T1	T2
begin	begin
T1.1 read(x)	T2.1 write(x,20)
T1.2 tmp = read(y)	T2.2 write(y,30)
T1.3 write(y,tmp+10)	commit
commit	



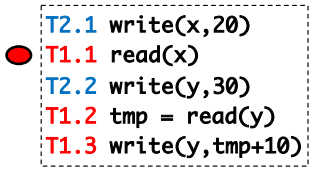
冲突

T1.1 read(x) and T2.1 write(x,20)

T1.2 tmp = read(y) and T2.2 write(y,30)

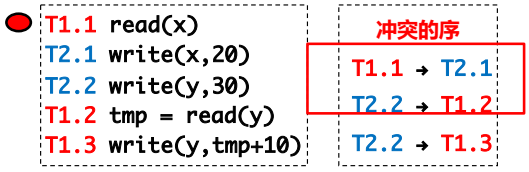
T1.3 write(y,tmp+10) and T2.2 write(y,30)

法则：冲突可串行化 当且仅当 其冲突图中没有环。



此调度是冲突可串行化的

T1 ← T2

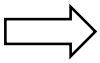


此调度不是冲突可串行化的

T1 ⇄ T2

判定法则的应用

T1	T2
begin	begin
T1.1 read(x)	T2.1 write(x,20)
T1.2 tmp = read(y)	T2.2 write(y,30)
T1.3 write(y,tmp+10)	commit
commit	



冲突

T1.1 read(x) and T2.1 write(x,20)

T1.2 tmp = read(y) and T2.2 write(y,30)

T1.3 write(y,tmp+10) and T2.2 write(y,30)

法则：冲突可串行化 当且仅当 其冲突图中没有环。

目标：并发运行事务 但 生成可串行化的调度。

如何实现?

遍历所有可能的调度，检查冲突图，选择可能的调度?

时间成本很高!

能否基于法则设计可以动态调度的算法?

2PL（two-phase locking）

1. 共享变量有自己的锁；
2. 任何操作前，事务必须获得该变量的锁；
3. 事务释放锁之后，不能获得其他锁

一般在commit后释放锁，这是技术上严格的2PL

死锁、性能在最后讨论

2PL提供了放置锁的位置

T1	T2
begin	begin
acquire(x.lock)	acquire(x.lock)
T1.1 read(x)	T2.1 write(x,20)
acquire(y.lock)	acquire(y.lock)
T1.2 tmp = read(y)	T2.2 write(y,30)
T1.3 write(y,tmp+10)	commit
commit	release(x.lock)
release(x.lock)	release(y.lock)
release(y.lock)	

注意：2PL强制两个事务串行运行！

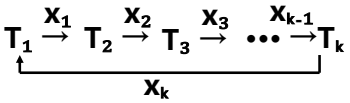
正确性

1. 共享变量有自己的锁；
2. 任何操作前，事务必须获得该变量的锁；
3. 事务释放锁之后，不能获得其他锁

为什么2PL是正确的算法?

命题：2PL生成的调度是冲突可串行化的。

证明：假设命题错误，2PL生成的调度不是冲突可串行化的，即2PL生成的冲突图中存在环，如下：



每对有边的事务<T,T*>有共享变量x，按照2PL规则，图中的边表示T*执行时T释放了x的锁，也就是T已执行完。

但图中有循环。形成循环依赖。

出现矛盾。

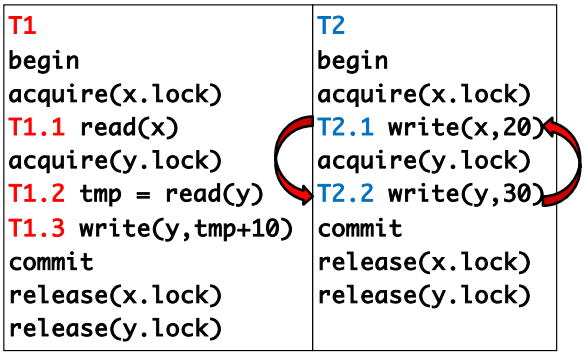
结论：2PL生成的调度是冲突可串行化的。

死锁的产生

1. 共享变量有自己的锁；
2. 任何操作前，事务必须获得该变量的锁；
3. 事务释放锁之后，不能获得其他锁

新问题：2PL会产生死锁

该调度不会死锁：



但如果交换T2.1与T2.2……

死锁的产生

1. 共享变量有自己的锁；
2. 任何操作前，事务必须获得该变量的锁；
3. 事务释放锁之后，不能获得其他锁

新问题：2PL生成的调度

不能避免死锁的发生

交换T2.1与T2.2后……

T1	T2
acquire(x.lock)	acquire(y.lock)
acquire(y.lock)	acquire(x.lock)

（回顾“虚拟链路”的内容）

产生了死锁的可能！

死锁问题的解决

1. 共享变量有自己的锁；
2. 任何操作前，事务必须获得该变量的锁；
3. 事务释放锁之后，不能获得其他锁

新问题：2PL生成的调度不能避免死锁的发生

如何消除死锁?

T1	T2
begin	begin
acquire(x.lock)	acquire(y.lock)
T1.1 read(x)	T2.1 write(y,30)
acquire(y.lock)	acquire(x.lock)
T1.2 tmp = read(y)	T2.2 write(x,20)
T1.3 write(y,tmp+10)	commit
commit	release(x.lock)
release(x.lock)	release(y.lock)
release(y.lock)	

（继续回顾“虚拟链路”的内容）

全局锁排序是一种（低效）选择。

（回顾影子副本、日志方法）

利用原子性方法，撤销死锁操作。

性能优化

带读写锁的2PL

1. 共享变量有自己的2个锁：读锁和写锁；
2. 任何读/写操作前，事务必须获得该变量的读/写锁；
3. 变量的读锁可以同时被多个事务获得，变量的写锁只能同时被1个事务获得；
4. 事务释放锁之后，不能获得其他锁。

分别获得 读/写 锁

T1	T2
begin	begin
acquire(x.reader_lock)	acquire(y.writer_lock)
T1.1 read(x)	T2.1 write(y,30)
acquire(y.reader_lock)	acquire(x.writer_lock)
T1.2 tmp = read(y)	T2.2 write(x,20)
acquire(y.writer_lock)	commit
T1.3 write(y,tmp+10)	release(x.writer_lock)
commit	release(y.writer_lock)
release(x.reader_lock)	
release(y.reader_lock)	
release(y.writer_lock)	

继续优化：

读锁能否在commit前释放?

提前释放

带读写锁的2PL

1. 共享变量有自己的2个锁：读锁和写锁；
2. 任何读/写操作前，事务必须获得该变量的读/写锁；
3. 变量的读锁可以同时被多个事务获得，变量的写锁只能同时被1个事务获得；
4. 事务释放锁之后，不能获得其他锁。

读锁在commit前释放

T1	T2
begin	begin
acquire(x.reader_lock)	acquire(y.writer_lock)
T1.1 read(x)	T2.1 write(y,30)
acquire(y.reader_lock)	acquire(x.writer_lock)
T1.2 tmp = read(y)	T2.2 write(x,20)
acquire(y.writer_lock)	commit
release(x.reader_lock)	release(x.writer_lock)
release(y.reader_lock)	release(y.writer_lock)
T1.3 write(y,tmp+10)	
commit	
release(y.writer_lock)	

不同类型的可串行化

这个调度是冲突可串行化的吗?

1. **T1 read(x)**
2. **T2 write(x)**
3. **T1 write(x=x+10)**
4. **T3 write(x)**

T1 **T2**

T3

T1 – T2 – T3 **视图可串行化** **NP-Hard** **性能 vs. 隔离**

但是……

2PL

- 可串行化的不同类型，可以精细区分不同的业务需求，进而指导不同的并发调度方式。
- 冲突可串行化是相对严格的一种调度方式。
- 2PL可以产生冲突可串行化的调度，通过将读锁和写锁区分，可以提升其性能。
  - 2PL → 冲突可串行化，但2PL不产生所有的冲突可串行化调度

总结

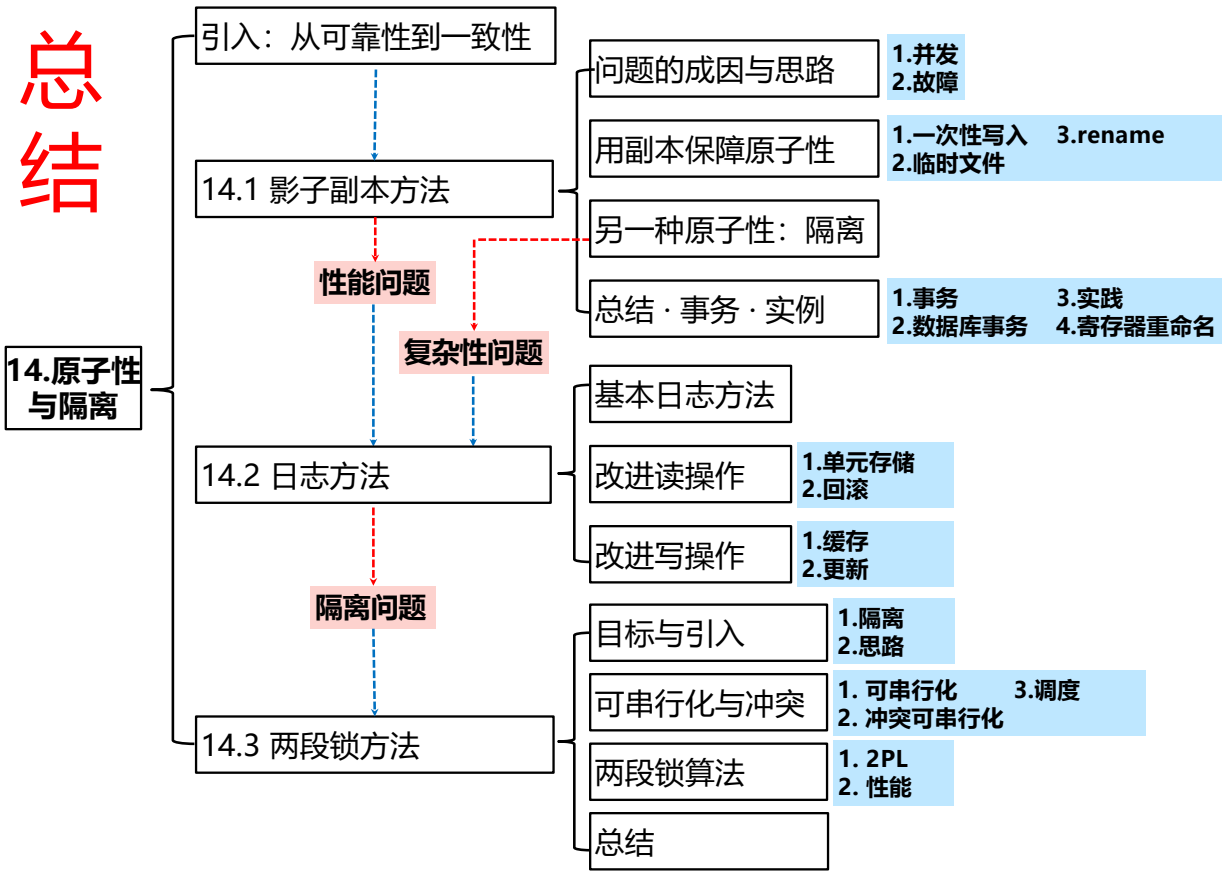
能够保证两种原子性的抽象 →

- 事务处理（transaction processing）

在事务处理抽象之上，我们可以更好地实施可靠性

方案评价

- 原子性：　使用日志得到了更好的方案
- 隔离：　使用2PL进行保证



填空题 3分

设置

实践中的原子性与一致性

我们已经讨论过，在x86指令中，每个指令都是具有原子性，即1个指令所包含的操作，要么全执行完、要么全未执行，不会发生执行了一半的情况。而且与别的指令在并发执行时，要么在前、要么在后，不会发生时序交叠的情况。

现在我们思考，在Java字节码（Java）汇编中，也有read、load、store、write等面向基本类型的操作。这些操作，字节码的解释器（JVM）来自动确保它的原子性吗？（ [填空1] ）。

我们接着思考。如果并发操作先后调用read、write，JVM能够保证它们的时序吗？ [填空2]

所以说：原子性 [填空3]（能/不能）确保一致性

作答

课后习题 6：原子性与隔离

占课程成绩比例：0.8%	作业完成时长：1 周

作业要求：认真阅读题目完成作答，并进行简要分析。作业答案要求手写。

Keys-R-Us 已与你签约，要求你实现一个内存中的键-值事务存储，命名为 KeyDB。KeyDB 提供了一个哈希表接口，用于存储键-值对，并检索之前与键关联的值。

你决定使用锁来提供隔离。锁 Lk 是针对键 k 的锁，对应于 KeyDB[k]。单个事务可以读取或写入多个 KeyDB。你的目标是为所有使用 KeyDB 的事务实现正确的隔离。事务可能会中止。在对 KeyDB[k]进行第一次读取或写入之前，会获取锁 Lk，在对 KeyDB[k]的最后一次访问之后，会释放锁 Lk。

问题 1：对于以下每个锁的相关规则，这个规则对始终保证任何一组并发事务之间正确的隔离，是必要的、充分的，还是既不必要也不充分？

A. 在事务开始后且在第一个读取或写入 KeyDB[k]之前必须获取锁 Lk，在最后一个读取或写入 KeyDB[k]之后但在事务结束之前必须释放锁 Lk。

B. 每个所需锁的获取操作必须在事务开始后且在任何其他操作之前发生，如果线程修改了相应的数据项，则在事务完成或中止之前不能释放锁。

C. 每个所需锁的获取操作必须在事务开始后且在第一个释放操作之前发生，如果线程修改了相应的数据项，则在事务完成或中止之前不能释放锁。

D. 所有获取多个锁的线程必须按照相同的顺序获取锁，不能在事务完成或中止之前之前释放锁。

每个所需锁的获取操作必须在事务开始后且在第一个释放操作之前发生，在事务完成或中止之前可以在最后一个读取或写入相应数据之后的任何时间释放锁。

问题 2：确定以下每个锁的相关规则是否可以避免或可能（随着时间无限接近 1 的可能性）消除任何一组并发事务之间的永久死锁。

A. 每个所需锁的获取操作必须在事务开始后且在任何其他操作之前发生，且在事务完成或中止之前不能释放锁。

B. 每个所需锁的获取操作必须在事务开始后且在第一个释放操作之前发生，且在事务完成或中止之前不能释放锁。

C. 所有获取多个锁的线程必须按相同的顺序获取锁。

D. 当事务开始时，设置一个定时器，其时间长于预计的事务执行时间。如果定时器超时，则中止当前事务，并通过随机指数退避算法选择一个新的时间值，然后再次尝试执行事务。

本章相关的参考文献

- Gray J, McJones P, Blasgen M, et al. The recovery manager of the System R database manager[J]. ACM Computing Surveys (CSUR), 1981, 13(2): 223-242.
- C. Mohan et al. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. ACM Transactions on Database Systems 17, 1 (1992), 94–162.
- Lomet D. B. Process structuring, synchronization, and recovery using atomic actions. Proceedings of an ACM Conference on Language Design for Reliable Software (March 1977), 128–137. Published as ACM SIGPLAN Notices 12, 3 (March 1977); Operating Systems Review 11, 2 (April 1977); and Software Engineering Notes 2, 2 (March 1977).

思考

- x86曾经有一类指令，在操作的过程中既有读内存操作，也有写内存操作。但在虚拟内存出现后，这一类的指令就被去掉了，为什么呢?
  - 思考该问题涉及的知识
    - 虚拟内存（第6章）
    - 多级内存管理与处理器流水线（第12章）
    - 中断（第7章）
  - 提示：缺页发生在指令中间，如何处理?

内存一致性与性能

- 多线程的内存一致性并不是对上层透明的
  - 例：java的volatile
- 为什么不透明
  - 一致性会损失性能，只在必要时使用
- 为什么编译器/解释器不自动判定
  - 不知道是否互斥