计算机系统设计原理

An Introduction

第9章 原子性：全有或全无与先后关系

杰罗姆·H·萨尔泽

M. 弗兰斯·卡舒克

Massachusetts Institute of Technology

Version 5.0

建议、意见、更正及豁免许可限制的请求：请通过电子邮件发送至：

Saltzer@mit.edu
和 kaashoek@mit.edu

CHAPTER

原子性：全有或全无 与 前或后

9

章节内容

概述

本章探讨两种紧密相关的系统工程设计策略。第一种是*all-or-nothing atomicity*，一种用于掩盖程序解释过程中发生故障的设计策略。第二种是*before-or-after atomicity*，一种用于协调并发活动的设计策略。第8章[在线]介绍了故障掩盖，但未展示如何掩盖运行中程序的故障。第5章介绍了并发活动的协调，并针对几个具体问题提供了解决方案，但未阐明确保操作具备前后一致性的系统方法。本章将探索如何系统性地综合出一种设计，既能提供故障掩盖所需的全有或全无属性，又能满足协调所需的前后一致性属性。

许多实用的应用都能从原子性中受益。例如，假设你正尝试从一家网店购买一台烤面包机。你点击了标有“购买”的按钮，但在收到响应之前断电了。你希望得到某种保证，即尽管发生了断电，要么购买已正确完成，要么什么都没发生。你不希望事后发现信用卡被扣款，而网店却未收到发货烤面包机的通知。换句话说，你希望看到由“购买”按钮发起的动作，即便在可能出现故障的情况下，也能保持“全有或全无”的特性。如果商店库存中仅剩一台烤面包机，而两位顾客几乎同时点击了“购买”按钮，那么其中一位顾客应收到购买确认，另一位则应收到“抱歉，缺货”的通知。如果出现其他情况，就会有问题。

两位客户都收到了购买确认。换句话说，两位客户都希望看到，由他们自己点击“购买”按钮所发起的活动，要么完全在其他并发“购买”按钮点击之前完成，要么完全在其之后完成。

原子性的单一概念框架为思考全有或全无的故障屏蔽以及并发活动的前后排序提供了一种强有力的方式。*Atomicity*是执行一系列被称为*actions*的步骤，使其看起来像是完成了一个单一、不可分割的步骤，这在操作系统和架构文献中被称为*atomic action*，在数据库管理文献中则称为*transaction*。当故障在一个正确设计的原子动作中间导致失败时，对于原子动作的调用者来说，该动作要么成功完成，要么什么都没做——因此原子动作提供了全有或全无的原子性。类似地，当多个原子动作并发进行时，每个原子动作看起来要么完全在其他所有原子动作之前发生，要么完全在其之后——因此原子动作提供了前后原子性。全有或全无的原子性与前后原子性共同提供了一种特别强大的模块化形式：它们隐藏了原子动作实际上由多个步骤组成的事实。

结果是在系统可能状态的描述中出现了一个*sweeping simplification*。这一简化为从故障中恢复和并发活动的协调提供了系统化的方法基础，它简化了设计，便于后续维护者的理解，也简化了正确性的验证。这些需求尤为重要，因为由协调错误引发的故障通常依赖于外部事件及不同线程间的相对时序。当时序相关的错误发生时，发现和诊断它的难度可能比在纯顺序活动中找出错误高出数个数量级。原因在于，即便是少量的并发活动也可能产生极其庞大的实时序列组合。通常无法确定在众多可能的步骤序列中，究竟是哪一种导致了错误，因此实际上难以在更受控的环境中复现该错误。鉴于调试此类错误的难度极高，确保正确协调*a priori*的技术显得尤为宝贵。

值得注意的是，同样的系统性方法——原子性——不仅适用于故障恢复，也适用于并发活动的协调。事实上，由于必须在协调并发活动的同时处理故障，若试图对这两个问题采用不同策略，就必须确保这些策略相互兼容。能够对两者使用同一策略，则是另一个*sweeping simplification*优势。

原子操作是计算机系统设计中广泛应用的基本构建模块。在数据库管理系统、流水线处理器的寄存器管理、文件系统、用于程序开发的变更控制系统，以及诸如文字处理器和日历管理器等众多日常应用中，都能见到原子操作的身影。

边栏9.1：操作与事务　系统设计者在讨论原子性时使用的术语可能会令人困惑，因为这一概念是由数据库设计者和硬件架构师各自独立发现并发展的。

一个改变多个数据值的动作可以具备以下至少四种独立属性中的任意一种或全部：它可以是*all-or-nothing*（要么所有更改都发生，要么都不发生），可以是*before-or-after*（所有更改都在每个并发动作之前或之后完成），可以是*constraint-maintaining*（这些更改保持某些特定的不变性），还可以是*durable*（更改持续到不再需要为止）。

数据库管理系统的设计者通常只关注那些要么全有要么全无且具备前后一致性的操作，并将此类操作描述为*transactions*。此外，他们使用术语*atomic*主要指的是全有或全无的原子性。另一方面，硬件处理器架构师习惯上用术语*atomic*来描述展现前后一致性原子性的操作。

本书并不试图改变这些常见用法。相反，它使用了限定性术语“全有或全无原子性”和“之前或之后原子性”。不加限定的“原子”一词可能隐含全有或全无、之前或之后，或两者兼具，具体取决于上下文。文中使用“事务”一词来表示一个*both*全有或全无且之前或之后的行为。

全有或全无原子性与前后原子性是动作普遍定义的属性，而约束则是不同应用以不同方式定义的属性。持久性介于两者之间，因为不同应用对持久性有着不同的要求。与此同时，约束和持久性的实现通常以原子性为前提。由于原子性属性可与其他两者模块化分离，本章仅聚焦于原子性。第10章[在线]将探讨设计者如何利用事务来实现约束并增强持久性。

本章各节将定义原子性，考察一些原子操作的实例，并探讨实现原子性的系统化方法：*version histories*、*logging*和*locking protocols*。随后，第10章[在线]将探讨原子性的一些应用。两章末尾的案例研究提供了原子性作为构建实用系统工具的真实世界示例。

9.1 原子性

原子性是计算机系统设计中多个不同领域所需的一种属性。这些领域包括数据库管理、硬件架构开发、操作系统接口规范，以及更广泛的软件工程领域。下表列举了原子性适用的一些问题类型。在

本章我们将在这些不同的领域中遇到两种原子性的实例。

Area	All-or-nothing atomicity	Before-or-after atomicity
database management	updating more than one record	records shared between threads
hardware architecture	handling interrupts and exceptions	register renaming
operating systems	supervisor call interface	printer queue
software engineering	handling faults in layers	bounded buffer

9.1.1 数据库中的全有或全无原子性

作为第一个例子，考虑一个银行账户的数据库。我们定义一个名为TRANSFER的过程，该过程从一个账户扣款并存入第二个账户，这两个账户都存储在磁盘上，具体如下：

```
1       过程 转移 (debit_account, credit_account, amount) 2  获取
(dbdata, debit_account) - amount  放置 (dbdata, debit_account) 5  获取
3       dbdata ← dbdata + amount  放置 (crdata, credit_account)
6       crdata ← crdata
```

其中*debit_account*和*credit_account* 分别标识待借记和贷记的账户记录。

假设系统在执行第4行的PUT指令时崩溃。即便我们采用了8.5.4节所述的MORE_DURABLE_PUT方法，在极其不巧的时刻发生系统崩溃仍可能导致写入磁盘的数据混乱，并丢失*debit_account*的值。我们更希望数据要么完整写入磁盘，要么完全不写入。换言之，我们希望PUT指令具备全有或全无的原子性特性。9.2.1节将阐述实现这一目标的方法。

在TRANSFER过程中还存在一个全有或全无的原子性要求。假设第4行的PUT操作成功，但在执行第5行或第6行时电源中断，导致计算机突然停止。当电源恢复后，计算机重启，但包括运行TRANSFER过程的线程状态在内的易失性内存已经丢失。此时若有人查询*debit_account*和*credit_account*中的余额，数值将无法正确匹配，因为*debit_account* 已更新为新值而*credit_account*仍保留旧值。有人可能建议将第一个PUT操作推迟至第二个操作之前执行，但这只是缩小了风险窗口而非彻底消除——电源仍可能在两个PUT操作之间发生故障。要消除这个风险窗口，我们必须设法确保这两个PUT指令，甚至整个TRANSFER过程，以全有或全无的原子方式完成。

动作。在9.2.3节中，我们将设计一个具有全有或全无特性的TRANSFER 程序，而在9.3节中，我们将看到提供该特性的一些其他方法。

9.1.2 中断接口中的全有或全无原子性

全有或全无原子性的第二个应用体现在线程所见的处理器指令集接口中。回顾第2章和第5章的内容，线程通常按照当前程序的指令依次执行操作，但某些事件可能会引起线程解释器的注意，导致由解释器而非程序提供下一条指令。当此类事件发生时，运行在中断线程中的另一个程序将接管控制权。

如果事件是从解释器外部到达的信号，中断线程可以简单地调用一个线程管理原语，如ADVANCE，如第5.6.4节所述，以向其他线程通报该事件。例如，其他线程正在等待的I/O操作可能现已完成。随后，中断处理程序将控制权交还给被中断的线程。此例要求中断线程与被中断线程之间具备前或后原子性。若被中断线程当时正在调用线程管理器，则中断线程对ADVANCE的调用应当发生在该调用之前或之后。

另一种可能是解释器检测到被中断线程中出现了问题。此时，中断事件会调用一个异常处理器，该处理器在原始线程的环境中运行。（侧边栏9.2提供了一些示例。）异常处理器要么调整环境以消除某些问题（如缺页错误），使原始线程得以继续执行；要么判定原始线程失败并将其终止。无论哪种情况，异常处理器都需要检查原始线程在中断瞬间所执行操作的状态——该操作是已完成，还是处于部分完成状态?

理想情况下，异常处理程序希望看到关于状态的“全有或全无”报告：要么引发异常的指令已完成，要么它未执行任何操作。全有或全无报告意味着原始线程的状态完全由异常处理程序所在层的值来描述。此类值的一个例子是程序计数器，它标识线程接下来要执行的指令。而中间态报告则意味着状态描述涉及更低层的值，可能是操作系统或硬件处理器本身的值。在这种情况下，仅知道下一条指令是不够的；处理程序还需要了解当前指令的哪些部分已执行、哪些部分未执行。例如，一条指令可能先递增地址寄存器，再从新地址处获取数据，最后将该数据值与另一寄存器中的值相加。若获取数据时触发缺页异常，则当前状态的描述为地址寄存器已递增，但数据获取与加法操作尚未执行。这种中间态报告会带来问题，因为处理程序在获取缺失页面后，不能简单地让处理器跳转至失败的指令重新执行——这会导致地址寄存器再次递增，而程序的本意并非如此。

侧边栏 9.2：可能导致调用异常处理程序的事件

1. 发生硬件故障：
   - •处理器检测到内存奇偶校验错误。
   - •传感器报告电力已中断；电源中剩余的能量可能仅够执行一次正常关机。

2. 硬件或软件解释器在程序中遇到明显错误的内容：
   - •程序试图除以零。
   - •程序向平方根函数提供了一个负数参数。

3. 继续需要一些资源分配或延迟初始化：
   - •运行线程在虚拟内存系统中遇到了缺页异常。
   - •运行线程遇到了一个间接异常，表明它在当前程序中遇到了未解析的过程链接。

4. 更紧急的工作需要优先处理，因此用户希望终止线程：
   - •这个程序的运行时间远超预期。
   - •程序运行正常，但用户突然意识到该赶最后一班火车回家了。

5. 用户意识到出了问题，决定终止线程：
   - •计算π时，程序开始显示3.1415…
   - •用户要求程序复制了错误的文件集。

6. 死锁：
   - •线程A已获取扫描器，并正等待内存释放；线程B则占用了所有可用内存，同时等待扫描器被释放。系统要么会检测到这一组等待状态无法解除，要么更常见的是，某个本不应触发的计时器最终超时。系统或计时器会向一个或两个死锁线程发出异常信号。

正如预期的那样。直接跳转到下一条指令也不正确，因为这会遗漏加法步骤。全有或全无的报告更为可取，因为它避免了处理程序需要窥探下一层细节的情况。现代处理器设计者通常会谨慎避免设计不具备全有或全无特性的指令。很快我们将看到，高层解释器的设计者也必须同样谨慎。

9.1.3节和9.1.4节探讨了异常导致运行线程终止，从而引发故障的情况。9.1.5节则分析了被中断线程继续执行（但愿）未察觉中断的情形。

9.1.3 分层应用中的全有或全无原子性

第三个全有或全无原子性的例子体现在运行程序出现故障时的挑战：在故障发生的瞬间，程序通常正处于执行某项操作的过程中，而让操作半途而废通常是不可接受的。我们的目标是获得更优雅的响应方式，方法则是要求某些动作序列必须表现为具有全有或全无特性的原子动作。原子动作与分层组织所呈现的模块性密切相关。分层组件的特点是高层能够完全隐藏底层的存在。这种隐藏特性使分层结构在错误隔离和系统化应对故障方面表现出色。

要理解这一点，可以回顾第2章日历管理程序的分层结构，如图9.19.1所示（该图可能看起来很熟悉——它是图2.10的副本）。日历程序通过执行一系列Java语言语句来实现用户的每个请求。理想情况下，用户永远不会察觉到日历管理器所实现动作的复合性质。同样，Java语言的每条语句都由硬件层的多个动作实现。如果Java解释器被精心实现，那么基于机器语言的实现细节将完全对Java程序员隐藏。

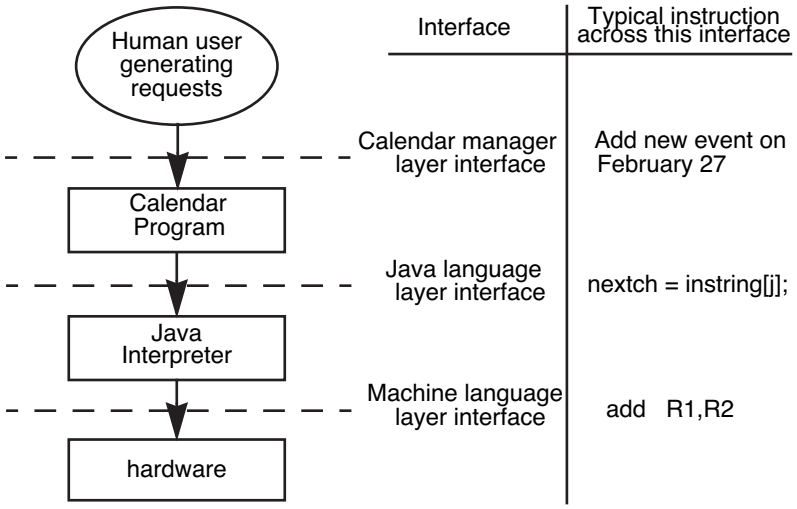


图9.1

一个具有三层解释的应用系统。用户请求的操作将会失败，但失败将在最底层被发现。优雅的响应需要在每个接口层面保持原子性{v*}。

现在考虑如果硬件处理器检测到一个应作为异常处理的情况会发生什么——例如，寄存器溢出。机器正处于机器语言层接口解释某个动作的过程中——即Java解释器程序中的某个ADD指令。这条ADD指令本身又处于解释Java语言接口动作的过程中——一个用于扫描数组的Java表达式。而这个Java表达式反过来又处于解释用户界面动作的过程中——用户向日历添加新事件的请求。报告“由位置41574处的ADD指令引起的溢出异常”对用户界面的用户来说是不可理解的；该描述仅在机器语言接口层面才有意义。遗憾的是，处于高层动作“中间”状态的隐含意义在于，对当前事态唯一准确的描述只能通过机器语言程序的执行进度来体现。

在我们这个例子中，一位全知观察者所理解的实际状况可能是这样的：寄存器溢出是由于在机器语言层面对一个存储着二进制补码负一的寄存器进行了加一操作。这条机器语言加法指令，在Java层面对应的是扫描字符数组的动作，而零值意味着扫描已到达数组末尾。Java层发起数组扫描，是为了响应用户要求在2月31日添加事件的请求。对于溢出异常的最高层级解读是"您试图在一个不存在的日期添加事件"。我们希望确保最终用户收到的是这份报告，而非关于寄存器溢出的技术细节。此外，我们还需要向用户保证：这个错误既不会在日历的其他位置生成空事件，也不会导致日历数据发生任何其他变更。由于系统无法执行请求的修改，它应当仅报告错误而不进行任何操作。无论是低层级的错误报告还是混乱的数据，都会向用户暴露出该动作的复合性。

基于分层应用的洞察，我们希望由下层检测到的故障能以特定方式被隔离，由此可以提出一个更形式化的全有或全无原子性定义：

全有或全无原子性

一系列步骤构成一个***all-or-nothing action***，如果从调用者的角度来看，该序列总是要么

- ***completes***,

或

- 以这样一种方式中止，使得看起来这个序列从未被首先执行过。也就是说，它***backs out.***

在分层应用程序中，设计理念是让每一层的每个动作都遵循“全有或全无”原则。也就是说，每当某一层的某个动作通过一系列操作执行时，

下一层级的动作要么完成其被要求执行的任务，要么完全回退，表现得仿佛从未被调用过。当控制权在底层检测到故障后返回到更高层级时，处于动作“中间状态”的问题便随之消失。

在我们的日历管理示例中，我们可能预期机器语言层会完成加法指令但触发溢出异常；Java解释器层在接收到溢出异常后，可能会判定其数组扫描已结束，并向日历管理层返回“扫描完成，未找到值”的报告；日历管理器会将此未找到报告视为应回退的信号，彻底撤销所有暂存更改，并告知用户由于该日期不存在，无法完成添加该日事件的请求。

因此，某些层级会运行至完成，而其他层级则会回退，表现得仿佛从未被调用过。但无论哪种情况，操作都是全有或全无的。在这个例子中，失败可能会一直传递回人类用户，由其决定下一步行动。而另一种失败（例如“日历中没有空间安排更多事件”）可能会被某个中间层拦截，该层知道如何掩盖这一失败（比如通过分配更多存储空间）。在这种情况下，全有或全无的要求在于：掩盖失败的层级必须发现，下层要么从未启动本应成为当前操作的动作，要么已经完成当前操作但尚未着手下一个动作。

全有或全无的原子性通常不是偶然实现的，而是通过精心的设计和规范达成的。设计者常常会犯错，而难以理解的错误信息正是设计出错的典型表现。为了深入理解其中的关键所在，让我们来看几个例子。

9.1.4 具有与不具有全有或全无属性的某些行为

在计算机架构中，尤其是高度流水线化的处理器上添加多级内存管理时，经常发现某些操作缺乏“全有或全无”特性。此时，处理器与操作系统之间的接口必须满足这一特性。若原始机器架构师在设计指令集时未充分考虑缺页异常，就可能出现指令执行“中途”触发缺页异常的情况——这可能在处理器覆写某些寄存器后发生，也可能在后续指令已进入流水线后发生。面对这种困境，试图添加多级内存功能的后继设计者将陷入两难：指令无法执行到底，因其所需操作数不在实存中；而从二级存储提取缺失页时，设计者希望允许操作系统将处理器转作他用（甚至可能运行获取缺失页的程序），但重用处理器需保存当前执行程序的状态以便后续重启。核心难题在于如何保存下一条指令指针。

如果每条指令都是一个“全有或全无”的操作，操作系统只需将遇到缺页异常的指令地址保存为下一条指令指针的值即可。这样保存的状态描述表明，程序正处于两条指令之间，其中一条已完全执行完毕，而下一条尚未开始。之后，当页面可用时，操作系统可以通过重新加载所有寄存器并将程序计数器设置为下一条指令指针所指示的位置来重启程序。处理器将从先前触发缺页异常的指令处继续执行，这次应该能够成功。　　另一方面，如果指令集中哪怕有一条指令不具备“全有或全无”特性，当执行该指令期间恰好发生中断时，操作系统如何保存处理器状态以供未来重启就完全不明朗了。设计者们提出了多种技术方案，在机器语言接口层面补全这一特性。第9.8节将描述一些曾存在此问题的机器架构实例，以及为它们添加虚拟内存所采用的技术。

第二个例子是监督程序调用（SVC）。5.3.4节指出，SVC指令会同时改变程序计数器和处理器模式位（在具有虚拟内存的系统中，还包括其他寄存器，如页映射地址寄存器），因此需要确保所有目标寄存器要么全部更新，要么完全不更新，即实现全有或全无的操作。此外，SVC会调用某个完整的内核过程。设计者希望将整个调用过程（即SVC指令与内核过程本身的组合）安排为一个全有或全无的动作。这种全有或全无的设计让应用程序员能够将内核过程视为硬件的延伸。然而，这一目标说来容易做来难，因为内核过程可能会检测到某些条件阻碍其执行预期操作。因此，必须精心设计内核过程。

考虑一个SVC（系统调用）到内核过程READ，该过程将下一个键入的击键传递给调用者。当应用程序调用READ时，用户可能尚未输入任何内容，因此READ的设计者必须安排等待用户输入。仅就这一点而言，情况并不特别成问题，但当还存在用户提供的异常处理程序时，情况就变得更为复杂。例如，假设在调用READ期间线程计时器可能到期，而用户提供的异常处理程序需要决定线程是否应继续运行一段时间。于是，场景是这样的：用户程序调用READ，需要等待，而在等待过程中，计时器到期，控制权转交给异常处理程序。不同系统为READ过程的设计选择了三种可能性之一，其中最后一种并非非此即彼的设计：

1. *An all-or-nothing design that implements the “nothing” option (blocking read)*：当发现没有可用输入时，内核过程首先调整返回指针（“将PC压回”），使得应用程序看起来像是在调用内核READ过程之前调用了AWAIT，随后将控制权转移至内核AWAIT 入口点。当用户最终键入内容导致AWAIT返回时，用户的线程会重新执行最初对READ的内核调用，此时便能找到已输入的内容。

输入。采用这种设计后，如果在等待期间发生定时器异常，当异常处理程序检查线程的当前状态时，它会发现答案是“应用程序正处于指令之间；其下一条指令是调用READ。”这一描述对用户提供的异常处理程序而言是可理解的，并且为该处理程序提供了几种选择。一种选择是继续执行线程，即继续执行对READ的调用。如果此时仍无输入，READ会再次将程序计数器（PC）回推并将控制权转移给AWAIT。另一种选择是让处理程序保存这一状态描述，计划在未来的某个时间点将某个线程恢复至此状态。

2. *An all-or-nothing design that implements the “all” option (non-blocking read)*：当内核发现没有可用输入时，会立即向应用程序返回一个零长度的结果，预期程序会检测并妥善处理这种情况。程序可能会检查结果的长度，若为零，则自行调用AWAIT，或转而执行其他操作。与之前的设计相同，此设计确保用户提供的定时器异常处理程序在任何时候都能看到线程当前状态的简单描述——即线程处于两条用户程序指令之间。然而，需注意避免对AWAIT的调用与下一个键入字符到达之间发生竞态条件。

3. *A blocking read design that is neither “all” nor “nothing” and therefore not atomic*：内核READ过程本身调用了AWAIT，阻塞线程直至用户输入一个字符。尽管这一设计在概念上看似简单，但从定时器异常处理器的视角来描述线程的状态却并不简单。它并非“处于两条用户指令之间”，而是“在用户调用内核过程READ的过程中等待某事件发生”。保存这一状态描述以供后续使用的选项已被排除。若要以该状态描述启动另一线程，异常处理器需能够请求“在调用内核READ 入口中的AWAIT之后立即启动此线程”。但允许此类请求会损害用户-内核接口的模块性。用户提供的异常处理器同样可以请求在内核的任何位置重启线程，从而绕过其门控机制，危及安全性。

第一和第二种设计直接对应于全有或全无操作定义中的两种选项，事实上某些操作系统确实同时提供了这两种选择。在第一种设计中，内核程序的运行方式使得调用看似从未发生过；而在第二种设计中，每次调用内核程序时都会确保其完整执行完毕。这两种设计都将内核过程转变为全有或全无操作，并且如果在等待期间发生异常，两者都会产生用户可理解的状态描述——程序正处于其两条指令之间的状态。

第4章介绍的客户端/服务器模型的一大吸引力在于，它倾向于迫使设计直面"全有或全无"的特性。由于服务器可能独立于客户端发生故障，客户端必须周密考虑恢复方案。

从服务器故障的角度来看，一个自然的模型是将服务器提供的每个动作都设为全有或全无的。

9.1.5 前后原子性：协调并发线程

在第5章中，我们学习了如何通过创建线程来表达并发机会，并发的目标是通过同时运行多个任务来提高性能。此外，上文第9.1.2节指出，中断同样能创造并发。并发线程在其执行路径交叉前不会引发任何特殊问题。路径交叉的方式总能用共享可写数据来描述：并发线程恰好在相近的时间点对同一块可写数据产生兴趣。这些并发线程甚至无需同时运行；若一个线程在操作过程中被挂起（可能由于中断），此时另一个运行中的线程就可能对挂起线程正在（或将来会再次）操作的数据产生兴趣。

从应用程序程序员的角度来看，第五章介绍了两种截然不同的并发协调需求：*sequence coordination* 和 *before-or-after atomicity*。序列协调是一种“动作*w*必须在动作*x*之前发生”的约束类型。为了确保正确性，第一个动作必须在第二个动作开始前完成。例如，从键盘读取输入的字符必须在运行将这些字符显示在屏幕上的程序之前完成。一般来说，编写程序时可以预见序列协调约束，且程序员知晓并发动作的身份。因此，序列协调通常通过特殊语言结构或共享变量（如第五章所述的事件计数器）进行显式编程。

相比之下，*before-or-after atomicity* 是一个更为普遍的约束条件，它要求多个同时操作同一数据的动作不应相互干扰。我们将前后原子性定义如下：

前或后原子性

并发操作具有***before-or-after***属性，如果从调用者的角度来看，其效果与这些操作以***completely before*** 或***completely after***的顺序发生相同。

在第5章中，我们了解了如何通过显式锁和实现ACQUIRE与RELEASE过程的线程管理器来创建先后动作。第5章展示了一些使用锁实现先后动作的实例，并强调编程实现正确的先后动作（例如协调多个生产者或多个消费者的有界缓冲区）可能是个棘手的命题。为确保正确性，必须构建一个令人信服的论证，证明每个涉及共享变量的操作都遵循了锁定协议。

前后原子性与顺序协调的一个不同之处在于，对于必须具有前后原子性属性的操作，其程序员未必知晓所有可能触及共享变量{v*}的其他操作的身份。这种认知缺失使得通过显式程序步骤来协调操作变得棘手。相反，程序员需要的是一种自动、隐式的机制，以确保每个共享变量{v*}都能得到妥善处理。本章将描述几种这样的机制。换言之，正确的协调要求并发线程在读写共享数据时遵循一定的规范。

计算机系统中前后原子性的应用比比皆是。在操作系统中，多个并发线程可能几乎同时决定使用共享打印机。若不同线程的打印行在输出中交错出现，将毫无意义。此外，哪个线程优先使用打印机其实并不重要；关键在于每次打印任务必须完整执行完毕后，下一个任务才能开始。因此，核心需求是为每个打印作业赋予前后原子性属性。

为了更详细的示例，让我们回到银行应用程序和TRANSFER过程。这次账户余额存储在共享内存变量中（记得声明关键字**reference**表示参数是按引用传递的，因此TRANSFER可以改变这些参数的值）：

```
过程 TRANSFER (引用 debit_account, 引用 credit_account, amount)
debit_account ← debit_account - amount  credit_account ← credit_account + amount
```

尽管表面上看似单一，但像“$x \leftarrow x + y$”这样的程序语句实际上是复合的：它涉及读取x和y的值，执行加法运算，然后将结果写回x。如果在此语句的读取和写入操作之间，有并发线程读取并更改了x的值，那么当此语句覆盖其更改时，其他线程可能会感到意外。

假设这一程序应用于账户A（初始金额为300美元）和B（初始金额为100美元），如

```
转账 (A, B, $10)
```

我们预计借方账户A最终会有290美元，而贷方账户B最终会有110美元。然而，假设有第二个并发线程正在执行该语句

```
转账 (B, C, $25)
```

其中账户C初始金额为175美元。当两个线程都完成转账后，我们预期B最终应有85美元，而C应有200美元。此外，无论两个转账操作中哪一个先发生，这一预期都应得到满足。但第一个线程中的变量*credit_account*与第二个线程中的变量*debit_account*绑定到了同一个对象（即账户B）。若两个转账操作几乎同时发生，就会产生正确性风险。要理解这一风险，请参考图9.2，该图展示了两个线程针对变量B的READ和WRITE步骤可能出现的几种时间序列。

图中每个时间序列展示了包含账户余额*B*的单元格数值历史。若步骤1-1和1-2均先于步骤2-1和2-2执行（或反之），两次转账将按预期运作，最终*B*余额为85美元。然而，若步骤2-1在步骤1-1之后、步骤1-2之前执行，则会出现错误：其中一笔转账本应作用于账户*B*却未能生效。前两种情况展示了共享变量*B*产生正确结果的历史记录；其余四种情况则呈现了导致*B*出现两种错误值的不同执行序列。

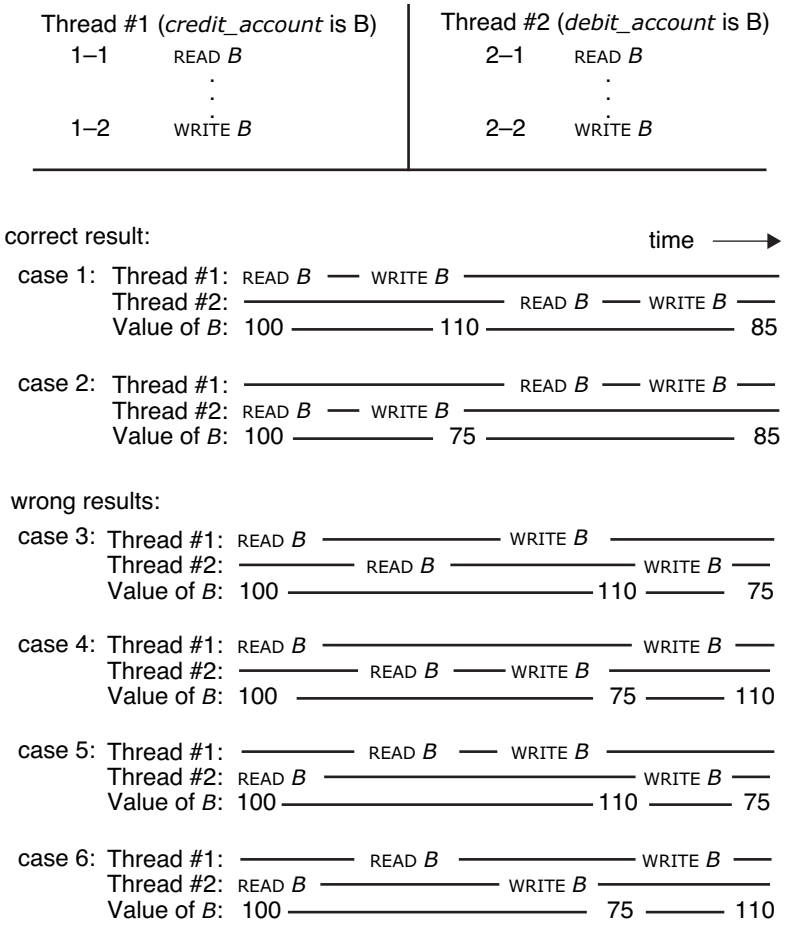


图9.2

变量B的六种可能历史，如果共享B的两个线程不协调它们的并发活动。

因此，我们的目标是确保前两个时间序列之一实际发生。实现这一目标的一种方法是，步骤1-1和1-2应当是原子性的，同样地，步骤2-1和2-2也应当是原子性的。在原程序中，这些步骤

debit_account ← *debit_account- amount*

和

credit_account ← *credit_account+ amount*

每个操作都应是原子性的。必须确保不存在这样的可能性：一个意图更改共享变量*debit_account*值的并发线程，在本语句的READ和WRITE步骤之间读取到该变量的值。

9.1.6 正确性与序列化

图9.2中前两个序列正确而其余四个错误的观点，是基于我们对银行应用的理解。若能建立一个不依赖于具体应用的、更普适的正确性概念则更为理想。应用独立性是模块化的目标：我们希望能在不涉及使用该机制的应用是否正确的前提下，单独论证提供"前或后"原子性的机制本身的正确性。

存在这样一种正确性概念：并发动作间的协调可被视为这些动作本身的正确性*if every result is guaranteed to be one that could have been obtained by some purely serial application*。

这一正确性概念背后的推理涉及几个步骤。考虑图9.3，它抽象地展示了这种影响

对系统施加某个动作（无论是原子性的还是非原子性的）时：该动作会改变系统的状态。现在，如果我们确信：

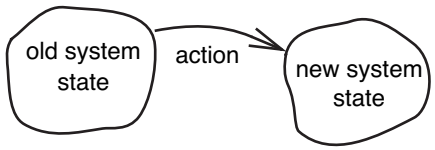


图9.3

单个动作将系统从一个状态转移到另一个状态。

1. 系统的旧状态是正确的
   从应用的角度来看，

2. 这一行动，完全由自身执行，
   正确地将任何正确的旧状态转换为正确的新状态，

那么我们可以推断新状态也必定是正确的。这一推理思路适用于任何依赖于应用的“正确”和“正确转换”定义，因此我们的推理方法与这些定义无关，从而与应用本身无关。

当多个动作并发执行时，如图9.4所示，其对应的要求是：最终的新状态应当是这些动作按某种顺序串行执行时可能产生的状态之一，如图9.5所示。这一正确性标准意味着，若并发动作的结果能确保等同于这些相同动作*some*纯粹串行应用所获得的结果，则说明并发动作的协调是正确的。

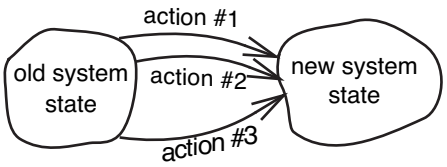


图9.4

当多个操作并发执行时，它们共同产生一个新状态。若这些操作满足前后顺序且旧状态正确，则新状态也将保持正确。

只要协调需求仅限于前后原子性，任何序列化{v*}方法都可行。

此外，我们甚至无需坚持系统必须实际遍历图9.5中任何特定路径上的中间状态——根据应用的定义，系统可以转而遵循虚线轨迹通过那些本身并不正确的中间状态。只要这些中间状态在实现层之上不可见，且系统最终必定会达到某个可接受的最终状态，我们就可以宣称该协调是正确的，因为存在一条通往该状态的轨迹，且该轨迹的每一步本都可以应用正确性论证。

由于我们对“前后原子性”的定义是，每一个前后动作都表现得像是在其他所有前后动作之前或之后完全执行，因此前后原子性直接引出了这一正确性概念。换言之，前后原子性的作用在于将动作序列化，由此可推知，前后原子性保证了协调的正确性。另一种表述方式是

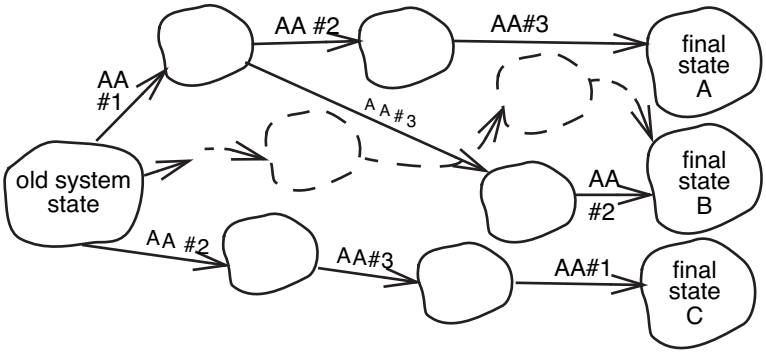


图9.5

我们坚持认为，最终状态必须能够通过某种原子操作的串行化序列达成，但具体是哪一种串行化序列则无关紧要。此外，我们无需强求中间状态必须真实存在。实际的状态轨迹可能如虚线所示，但前提是从外部无法观测到这些中间状态。

表达这一观点的一种方式是，当并发操作具有先后关系属性时，它们是 *serializable: there exists some serial order of those concurrent transactions that would, if followed, lead to the same ending state.**。因此，在图9.2中，情况1和情况2的序列可能源自串行化顺序，但情况3至6的操作则不可能。

在图9.2的示例中，仅存在两个并发动作，且每个并发动作仅包含两个步骤。随着并发动作数量及每个动作中步骤数的增加，各步骤可能发生的顺序组合将呈指数级增长，但其中仅有部分顺序能确保结果正确。由于并发的本质在于提升性能，我们期望能从所有正确顺序中筛选出性能最优的那个正确顺序。可以想见，这一选择过程通常极具挑战性。本章9.4与9.5节将介绍若干编程规范，这些规范确保从可能的顺序子集中进行选择——该子集所有成员均能保证正确性，但遗憾的是，可能不包含性能最优的正确顺序。

在某些应用中，采用比可串行化更强的正确性要求是合适的。例如，银行系统的设计者可能希望通过要求所谓的*external time consistency*来避免时代错位：若存在任何外部证据（如打印的收据）表明"之前或之后动作"T_1在"之前或之后动作"T_2开始前已结束，则系统内部T_1与T_2的串行化顺序应确保T_1先于T_2。另一个更强正确性要求的例子是，处理器架构师可能要求 *sequential consistency*：当处理器并发执行同一指令流中的多条指令时，其结果应如同这些指令按照程序员指定的原始顺序依次执行。

回到我们的例子，一个真实的资金转账应用通常有几个明确的前后原子性要求。考虑以下审计程序；其目的是验证所有账户余额之和为零（在复式记账法中，属于银行的账户，如金库中的现金金额，具有负余额）：

```
                程序 审计() sum ← 0 对每一个 W ← 在
                bank.accounts + W.balance 如果 (
sum ← sum       sum ≠ 0) 则要求调查
```

假设AUDIT在一个线程中运行的同时，另一个线程正在将资金从账户A转移到账户B。如果AUDIT在转账前检查账户A，在转账后检查账户B，就会将转账金额重复计算两次，

* The general question of whether or not a collection of existing transactions is serializable is an advanced topic that is addressed in database management. Problem set *36* explores one method of answering this question.

因此将计算出错误的答案。因此，整个审计程序应在任何单次转账之前或之后进行：我们希望它是一个“前或后”的动作。

还有另一个前后原子性要求：如果AUDIT应在TRANSFER 中的语句之后运行

debit_account ← *debit_account- amount*

但在声明之前

credit_account ← *credit_account+ amount*

它将计算一个不包含*amount*的总和；因此我们得出结论，这两个余额更新应该完全在任何AUDIT 操作之前或之后发生；换句话说，TRANSFER应该是一个“之前或之后”的操作。

9.1.7 全有或全无与前后原子性

我们现在已经看到了原子性的两种形式示例：全有或全无（all-or-nothing）与前或后（before-or-after）。这两种形式有一个共同的根本目标：隐藏动作的内部结构。有了这一洞见，原子性显然是一个统一的概念：

原子性

An action is atomic if there is no way for a higher layer to discover the internal structure of its implementation.

这一描述实质上是对原子性的根本定义。由此，我们可以立即得出两个重要推论，分别对应全有或全无的原子性以及前后一致的原子性：

1. 从调用原子操作的过程来看，原子操作总是要么如期完成，要么什么都不做。这一特性使得原子操作在从故障中恢复时非常有用。 2. 从并发线程的角度看，原子操作的行为表现为它要么发生在所有其他并发原子操作之前*completely before*，要么之后*completely after*。这一特性使得原子操作在协调并发线程时非常有用。

这两点后果并无本质区别。它们只是两种视角：第一种来自调用该操作的线程内部其他模块，第二种则来自其他线程。两种观点都源于同一个核心理念——操作的内部结构对实现该操作的模块之外不可见。这种内部结构的隐藏正是模块化的精髓，但原子性是一种异常强大的模块化形式。原子性不仅隐藏了具体哪些{v*}

步骤构成了原子动作，但关键在于它本身具有结构。原子性与其它系统构建技术，如数据抽象和客户端/服务器组织，存在亲缘关系。数据抽象旨在隐藏数据的内部结构；客户端/服务器组织旨在隐藏主要子系统的内部结构。同样地，原子性旨在隐藏动作的内部结构。这三者都是实施工业级模块化的方法，从而确保复杂系统各组件间不会出现意料之外的交互。

我们多次使用了“从调用者的角度来看”这样的表述，暗示可能存在另一种视角，从中可以观察到内部结构*is*。这一视角由原子动作的实现者所持有，他们往往痛苦地意识到一个动作实际上是复合的，并且必须额外努力向更高层级及并发线程隐藏这一事实。因此，层级间的接口是定义原子动作不可或缺的部分，它们为动作的实现提供了以任何方式运作的可能性，只要最终能保证原子性。

隐藏原子动作内部结构还有另一个方面：原子动作可以具有良性的副作用。一个常见的例子是审计日志，其中遇到问题的原子动作会记录检测到的故障性质及恢复序列，以供后续分析。有人可能认为，当故障导致回滚时，审计日志也应一并回滚；但这样做将违背其初衷——审计日志的全部意义就在于记录故障的细节。关键在于，审计日志通常是实现原子动作的层的私有记录；在正常操作过程中，它对上层不可见，因此无需回滚。（另一项原子性要求是确保描述故障的日志条目完整无缺，不会在随后的恢复过程中丢失。）

另一个良性副作用的例子是性能优化。例如，在一个高性能数据管理系统中，当上层原子操作要求系统向文件插入新记录时，数据管理系统可能出于性能优化考虑，决定此时对文件进行物理重排以获得更优存储顺序。若原子操作失败并中止，只需确保新插入的记录被移除即可，无需将文件恢复至原先效率较低的存储布局。同样地，底层缓存若已载入被原子操作访问的变量，则无需清空缓存；堆存储的垃圾回收操作也无需回退。只要这些副作用对原子操作的高层客户端透明——除了可能影响后续操作的执行速度，或是通过专门报告性能指标或故障的接口暴露——它们就不会构成问题。

9.2 全有或全无原子性 I：概念

本章第9.1节定义了全有或全无原子性及前后原子性的目标，并提供了一个概念框架，至少在原则上允许设计者判断某个提议的算法是否正确协调了并发活动。然而，它并未提供实现任一目标的具体实例。本章的这一节连同下一节，将描述一些广泛适用的系统化实现*all-or-nothing*原子性的技术。本章后续章节将对前后原子性进行同样的探讨。

许多示例采用了第5章介绍的名为*boot-strapping*的技术，这种方法类似于归纳证明。回顾一下，自举意味着首先寻找一种系统化的方法，将一般性问题简化为同一问题的某个更为狭窄的特定版本。然后，利用某种专门的方法解决这个狭窄问题，该方法可能仅适用于该特定情况，因为它利用了具体情境的优势。通用解决方案因此由两部分组成：一个针对特殊情形的技术，加上一个将一般问题系统化归约为特殊情形的方法。 回顾5.2.3节通过实现一个名为ACQUIRE的过程，解决了从任意代码序列创建前后动作的通用问题。该过程本身要求在读取并设置锁值的两三行代码中具备前后原子性。随后，它借助一个特殊的硬件特性直接实现了该读写序列的前后动作，并展示了仅依赖硬件执行常规LOAD和STORE作为前后动作的软件实现（见边栏5.2）。 本章多次运用自举方法。第一个示例从特殊情形出发，随后引入将一般问题归约至该特殊情形的方法。这种被称为*version history*的归约方法在实践中使用频率不高，但一旦理解其原理，就能轻松领会第9.3节将介绍的更广泛使用的归约方法为何有效。

9.2.1 实现全有或全无原子性：ALL_OR_NOTHING_PUT

第一个例子是关于一种对单个磁盘扇区进行全有或全无更新的方案。要解决的问题是，如果系统在磁盘写入过程中崩溃（例如，操作系统遇到错误或电源故障），那么在故障瞬间正在写入的扇区可能会包含新旧数据的混乱混合，导致无法使用。目标是创建一个具有全有或全无特性的PUT，使得当GET后续读取该扇区时，总是返回旧数据或新数据，而绝不会是混乱的混合体。

为了使实现更加精确，我们开发了一个磁盘容错模型，该模型是对第8章[在线]中介绍的模型稍作变动的版本，并以个人电脑上的日历管理程序作为示例应用。用户期望的是，如果在向日历添加新事件时系统发生故障，当系统稍后重启时，日历仍能保持完好无损。无论新事件是否最终成功添加到日历中，

日历的重要性不及系统故障的不合时宜对日历造成的损害。该系统由人类用户、显示器、处理器、一些易失性内存、磁盘、操作系统以及日历管理程序组成。我们将该系统建模为几个部分：

Overall system fault tolerance model.

- 无差错操作：所有工作均按预期进行。用户发起诸如向日历添加事件等操作，系统通过向用户显示消息来确认这些操作。
- 容忍误差：发起操作的用户注意到系统在确认操作完成之前已发生故障，当系统再次运行时，会检查该操作是否实际执行。
- 不可容忍的错误：系统在用户未察觉的情况下发生故障，导致用户意识不到应该检查或重试某个可能未被系统完成的动作。

容错规范意味着，在可能的情况下，整个系统应实现快速失效：如果在更新过程中出现问题，系统会在接收更多请求之前停止运行，用户也会意识到系统已停止。通常，人们会设计这样的系统以尽量减少不可容忍错误的发生概率，例如通过要求人工用户的监督。这样，人工用户便能察觉（可能是由于缺乏响应）出问题了。系统重启后，用户知道去查询操作是否完成。这一设计策略应与我们第7章[在线]中对尽力而为网络的研究相呼应。底层（计算机系统）提供尽力而为的实现，而高层（人工用户）进行监督并在必要时重试。例如，假设人工用户向日历添加一个约会，就在点击“保存”时系统崩溃。用户不知道添加是否真的成功，因此系统再次启动后，第一件事就是打开日历查看发生了什么。

Processor, memory, and operating system fault tolerance model.

模型的这一部分只是更精确地规定了硬件和操作系统预期的快速失效特性：

- 无差错运行：处理器、内存和操作系统均遵循其规格要求。
- 检测到错误：硬件或操作系统出现故障。系统采用快速失效机制：硬件或操作系统检测到故障后，会从干净状态*before*重启，避免对磁盘发起任何进一步的PUT操作。
- 不可容忍的错误：硬件或操作系统中出现了故障。处理器在检测到故障前继续混乱运行，并将PUT损坏的数据写入磁盘。

该模型中处理器/内存/操作系统部分的主要目标是在任何损坏数据被写入磁盘存储系统之前检测故障并停止运行。关键在于在下一次磁盘写入前捕捉故障以实现错误隔离：若达成此目标，设计者可假定潜在错误值仅存在于处理器寄存器和易失性内存中，而磁盘数据应是安全的——除第8.5.4.2节所述例外情况：若系统崩溃时正在进行PUT写入磁盘的操作，故障系统可能已损毁易失性内存中的磁盘缓冲区，进而破坏了正在写入的磁盘扇区。

因此，恢复过程可以依赖于磁盘存储系统仅包含未损坏的信息，或至多一个损坏的磁盘扇区。实际上，重启后磁盘将包含*only*信息。“从零开始重启”意味着系统丢弃了易失性内存中保存的所有状态。这一步使系统达到与发生电源故障相同的状态，因此单一的恢复程序能够同时处理系统崩溃和电源故障。丢弃易失性内存还意味着所有当前活跃的线程消失，所以正在进行的一切都会突然停止，必须重新启动。

Disk storage system fault tolerance model.

实现全有或全无原子性涉及一些步骤，这些步骤类似于第8章[在线]中MORE_DURABLE_PUT/GET的衰变掩蔽——特别是算法会写入数据的多个副本。为了阐明全有或全无机制的工作原理，我们暂时回溯到CAREFUL_PUT/GET（见8.5.4.5节），它掩盖了软磁盘错误，但不包括硬盘错误或磁盘衰变。为了进一步简化，我们暂时假设磁盘永不衰变且不存在硬错误。（由于这种完美磁盘假设显然不切实际，我们将在第9.7节中予以推翻，该节描述了一种即使面对磁盘衰变和硬错误也能实现全有或全无原子性的算法。）

在完美磁盘假设下，只有一种情况可能出错：系统恰好在最糟糕的时刻崩溃。因此，这种简化谨慎磁盘系统的容错模型就变为：

- 无差错操作：CAREFUL_GET 返回最近一次在 *track* 上通过 *sector_number* 调用 CAREFUL_PUT 的结果，附带 *status* = OK。
- 可检测错误：操作系统在CAREFUL_PUT期间崩溃，损坏了易失性存储中的磁盘缓冲区，且CAREFUL_PUT将损坏的数据写入磁盘的一个扇区。

如果假设应用程序在调用CAREFUL_PUT 之前（即系统崩溃可能导致数据损坏之前）已经计算并包含了端到端的校验和，那么我们可以将此类错误归类为“可检测的”。

本次谨慎存储层修订的变动在于，当系统崩溃发生时，磁盘上的一个扇区可能损坏，但接口客户端确信：(1)该扇区是唯一可能受损的扇区；(2)若该扇区已损坏，任何后续读取该扇区的操作都将检测到问题。在处理器模型与存储系统模型之间，所有预期故障现在都会导致相同的情{v*}

```
1   procedure ALMOST_ALL_OR_NOTHING_PUT (data, all_or_nothing_sector)
2       CAREFUL_PUT (data, all_or_nothing_sector.S1)
3       CAREFUL_PUT (data, all_or_nothing_sector.S2)          // Commit point.
4       CAREFUL_PUT (data, all_or_nothing_sector.S3)

5   procedure ALL_OR_NOTHING_GET (reference data, all_or_nothing_sector)
6       CAREFUL_GET (data1, all_or_nothing_sector.S1)
7       CAREFUL_GET (data2, all_or_nothing_sector.S2)
8       CAREFUL_GET (data3, all_or_nothing_sector.S3)
9       if data1 = data2 then data ← data1                    // Return new value.
10      else data ← data3                                     // Return old value.
```

图9.6

ALMOST_ALL_OR_NOTHING_PUT 和 ALL_OR_NOTHING_GET 的算法。

系统检测到故障后，会重置所有处理器寄存器和易失性内存，清除所有活动线程并重新启动。最多只有一个磁盘扇区会受损。

我们的问题现在简化为提供“全有或全无”属性：目标是创建 *all-or-nothing disk storage*，它保证要么完全且正确地更改扇区上的数据，要么让未来的读者看起来似乎根本没有触碰过它。这里有一个简单但效率稍低的方案，利用虚拟化技术：为每个需要具备“全有或全无”属性的数据扇区分配三个物理磁盘扇区，分别标识为*S1*、*S2*和*S3.* 。这三个物理扇区共同构成一个虚拟的“全有或全无扇区”。在系统中原先使用该磁盘扇区的每个位置，用这个由三元组{*S1*, *S2*, *S3*}标识的“全有或全无扇区”替换它。我们从一个近乎正确的“全有或全无”实现ALMOST_ALL_OR_NOTHING_PUT开始，发现其中的一个错误，然后修复这个错误，最终创建出正确的ALL_OR_NOTHING_PUT。

当要求写入数据时，ALMOST_ALL_OR_NOTHING_PUT 会按顺序在*S1*、*S2*和*S3,*上各写入三次，每次等待前一次写入完成，这样即使系统崩溃，三个扇区中也只有一个会受到影响。读取数据时，ALL_OR_NOTHING_GET会读取所有三个扇区并比较它们的内容。如果*S1*和*S2*的内容一致，ALL_OR_NOTHING_GET会将该值作为全有或全无扇区的值返回。若*S1*与*S2*存在差异，ALL_OR_NOTHING_GET则返回*S3*的内容作为全有或全无扇区的值。图9.6展示了这段近乎正确的伪代码。

让我们探讨这一实现在系统崩溃时的表现。假设在之前的某个时刻，一条记录已被正确存储在一个全有或全无扇区中（换言之，所有三个副本完全相同），现在有人通过调用ALL_OR_NOTHING_PUT来更新它。目标是即使更新过程中发生故障，后续的读取者通过调用ALL_OR_NOTHING_GET也能始终确保获得某个完整、一致的记录版本。

假设ALMOST_ALL_OR_NOTHING_PUT在完成写入扇区*S2*之前的某个时刻被系统崩溃中断，从而损坏了*S1*或*S2*。在这种情况下，

```
1   procedure ALL_OR_NOTHING_PUT (data, all_or_nothing_sector)
2       CHECK_AND_REPAIR (all_or_nothing_sector)
3       ALMOST_ALL_OR_NOTHING_PUT (data, all_or_nothing_sector)

4   procedure CHECK_AND_REPAIR (all_or_nothing_sector) // Ensure copies match.
5       CAREFUL_GET (data1, all_or_nothing_sector.S1)
6       CAREFUL_GET (data2, all_or_nothing_sector.S2)
7       CAREFUL_GET (data3, all_or_nothing_sector.S3)
8       if (data1 = data2) and (data2 = data3) return     // State 1 or 7, no repair
9       if (data1 = data2)
10          CAREFUL_PUT (data1, all_or_nothing_sector.S3) return    // State 5 or 6.
11      if (data2 = data3)
12          CAREFUL_PUT (data2, all_or_nothing_sector.S1) return    // State 2 or 3.
13      CAREFUL_PUT (data1, all_or_nothing_sector.S2)       // State 4, go to state 5
14      CAREFUL_PUT (data1, all_or_nothing_sector.S3)       // State 5, go to state 7
```

图9.7

ALL_OR_NOTHING_PUT 和 CHECK_AND_REPAIR 的算法。

当ALL_OR_NOTHING_GET读取扇区*S1*和*S2*时，它们的值会有所不同，此时不清楚该信任哪一个。由于系统采用快速失败机制，扇区*S3*尚未被ALMOST_ALL_OR_NOTHING_PUT触及，因此它仍保留着之前的值。返回*S3*中找到的值，就能达到ALMOST_ALL_OR_NOTHING_PUT 未执行任何操作的预期效果。

现在，假设ALMOST_ALL_OR_NOTHING_PUT 在成功写入扇区*S2*后不久因系统崩溃而中断。在这种情况下，崩溃可能已损坏*S3*，但*S1*和*S2*均包含新更新的值。ALL_OR_NOTHING_GET 返回*S1*的值，从而实现了ALMOST_ALL_OR_NOTHING_PUT已完成其工作的预期效果。

那么这个设计有什么问题呢？ ALMOST_ALL_OR_NOTHING_PUT假设开始时所有三个副本都是相同的。但之前的故障可能会违反这一假设。假设ALMOST_ALL_OR_NOTHING_PUT在写入*S3*时被中断。下一个调用ALL_OR_NOTHING_GET的线程会发现*data1=data2*，因此它会按预期使用*data1*。接着，新线程调用ALMOST_ALL_OR_NOTHING_PUT，但在写入*S2*时被中断。现在，*S1*不等于*S2*，所以下一次调用ALMOST_ALL_OR_NOTHING_PUT会返回损坏的*S3*。

针对此错误的修复方案是让ALL_OR_NOTHING_PUT 确保在更新前三个扇区完全相同。它可以通过调用名为CHECK_AND_REPAIR的过程来提供这一保证，如图9.7所示。CHECK_AND_REPAIR 只需比较三个副本，如果它们不一致，就会强制使其一致。T要理解其工作原理，假设有人在三个副本当前都包含相同值（我们称之为“旧”值）时调用ALL_OR_NOTHING_PUT 。由于ALL_OR_NOTHING_PUT会写入“新”值

将值依次且按顺序填入*S1*、*S2*和*S3*，即使发生崩溃，在下次调用ALL_OR_NOTHING_PUT时，CHECK_AND_REPAIR只需考虑七种可能的数据状态：

data state:	1	2	3	4	5	6	7
sector *S1*	old	bad	new	new	new	new	new
sector *S2*	old	old	old	bad	new	new	new
sector *S3*	old	old	old	old	old	bad	new

阅读此表的方法如下：如果*S1*、*S2*和*S3*三个扇区均包含“旧”值，则数据处于状态1。此时，若CHECK_AND_REPAIR发现三个副本完全相同（图9.7第8行），则数据处于状态1或状态7，因此CHECK_AND_REPAIR直接返回。若该测试未通过，而*S1*和*S2*扇区的副本相同（第9行），则数据必然处于状态5或状态6，于是CHECK_AND_REPAIR强制*S3*扇区与之匹配并返回（第10行）。若*S2*和*S3*扇区的副本相同，则数据必处于状态2或状态3（第11行），因此CHECK_AND_REPAIR强制*S1*扇区与之匹配并返回（第12行）。唯一剩下的可能是数据处于状态4，此时*S2*扇区必定损坏，但*S1*扇区包含新值而*S3*扇区包含旧值。选择使用哪个值是任意的；如所示流程，将*S1*扇区的新值复制到*S2*和*S3*两个扇区。

如果在运行CHECK_AND_REPAIR时发生故障怎么办？该程序系统性地推动状态从状态4向前推进到状态7，或从状态3向后回退到状态1。如果CHECK_AND_REPAIR本身被另一次系统崩溃中断，重新运行它将从上一次尝试中断的地方继续。

我们可以对ALL_OR_NOTHING_GET和ALL_OR_NOTHING_PUT实现的算法做出几点观察：

1. 这一全有或全无原子性算法假设同一时间仅有一个线程尝试执行ALL_OR_NOTHING_GET或ALL_OR_NOTHING_PUT。该算法实现了全有或全无原子性，但未达成先后原子性。

2. CHECK_AND_REPAIR是*idempotent*。这意味着一个线程可以启动该过程，执行任意数量的步骤，被崩溃中断，然后无论多少次返回到起点，只要后续调用ALL_OR_NOTHING_GET，最终结果都是相同的。

3. 在ALMOST_ALL_OR_NOTHING_PUT的第*3*行完成标记为“提交点”的CAREFUL_PUT操作后，新数据将对未来的ALL_OR_NOTHING_GET操作可见。在该步骤开始执行之前，调用ALL_OR_NOTHING_GET将看到旧数据。而在第*3*行完成后，调用ALL_OR_NOTHING_GET则会看到新数据。

4. 尽管该算法会写入数据的三个副本，但副本的主要目的并非如第9.5节所述提供持久性保障。实际上，之所以依次按特定顺序写入三个副本，是为了在任何时刻及所有故障场景下都能确保遵守*golden rule of atomicity*原则，这一原则将是下一节讨论的主题。

实现全有或全无磁盘扇区有几种方法。在第8章[在线]接近尾声处，我们介绍了一种针对衰变事件的容错模型，该模型不掩盖系统崩溃，并应用了称为RAID的技术来屏蔽衰变，以实现持久存储。这里，我们从一个略为不同的容错模型出发，该模型忽略了衰变，转而设计了掩盖系统崩溃并实现全有或全无存储的技术。实际上，我们应当从同时考虑系统崩溃和衰变的容错模型入手，设计出既全有或全无又持久的存储系统。由Xerox公司的研究人员Butler Lampson和Howard Sturgis提出的这样一个模型，连同其所需的更复杂恢复算法，构成了第9.7节的主题。该模型还有一个额外特点，即每个全有或全无扇区仅需两个物理扇区。

9.2.2 系统性原子性：提交与黄金法则

ALL_OR_NOTHING_PUT 和ALL_OR_NOTHING_GET 的例子展示了全有或全无原子性（all-or-nothing atomicity）的一个有趣特例，但它并未提供如何系统化构建更通用全有或全无操作的指导。通过该示例，我们的日历程序现在拥有了一种工具，能够以全有或全无属性写入单个扇区，但这与安全地向日历添加事件并不相同——因为添加事件可能需要重组数据结构，进而涉及写入多个磁盘扇区。我们可以对多个扇区执行一系列ALL_OR_NOTHING_PUT操作，确保每个扇区自身以全有或全无方式写入，但若在写入一个扇区后、下一个扇区前发生崩溃，仍会导致整个日历添加操作处于部分完成状态。要实现整个日历添加操作的全有或全无特性，我们需要一种通用化的解决方案。

理想情况下，人们可能希望能够在程序中选取任意指令序列，像图9.8所示那样用某种**begin**和**end** 语句将其包围，并期待语言编译器和操作系统施展某种魔法，使被包围的序列成为全有或全无的操作。遗憾的是，目前无人知晓如何实现这一点。但如果程序员愿意为满足全有或全无的原子性要求做出适度让步，我们就能接近这一目标。这种让步体现在对全有或全无操作的组成步骤施加一套规范约束。

该学科首先将序列中的某个单一步骤识别为*commit point*。因此，如图9.9所示，全有或全无操作被划分为两个阶段：*pre-commit phase*和*post-commit phase*。在预提交阶段，设计的约束规则是无论发生什么情况，都必须能够以不留痕迹的方式回退这一全有或全无操作。而在后提交阶段，设计的约束规则则是无论发生什么情况，操作都必须成功执行到底。因此，全有或全无操作只有两种可能结果。若操作启动后未达提交点即回退，则称其*aborts*；若操作通过了提交点，则称其*commits*。

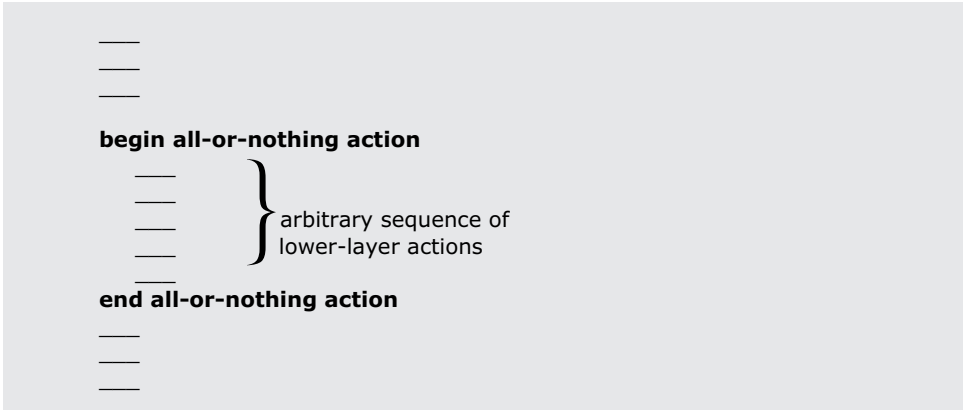


图9.8

我全有或全无行为无痛编程的虚拟语义　　离子。

我们可以对预提交阶段的限制做出几点观察。预提交阶段必须识别完成全有或全无操作所需的所有资源，并确认它们的可用性。数据名称应被绑定，权限需得到检查，待读取或写入的页面应在内存中，可移动介质需挂载，栈空间必须分配等。换言之，预提交阶段应完成所有必要步骤，以预见并满足后提交阶段“一鼓作气执行到底”的严苛要求。此外，预提交阶段必须保持随时中止的能力。若此全有或全无操作中止，预提交阶段对系统状态的任何更改都必须是可撤销的。通常，这一要求意味着共享的

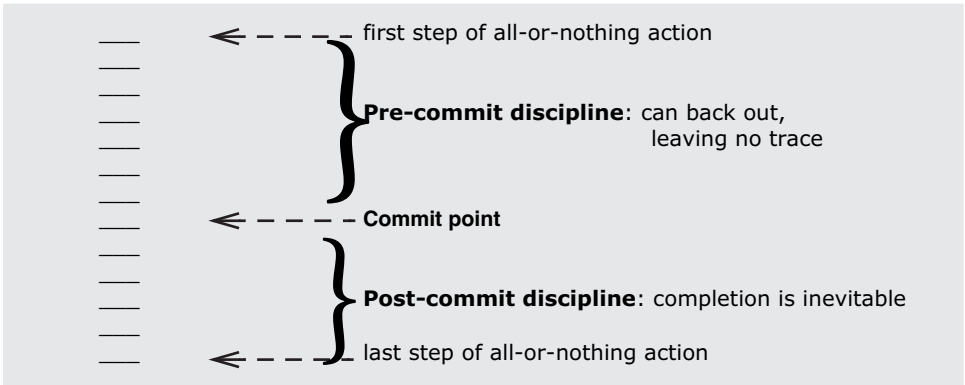


图9.9

全有或全无行动的提交点。

资源一旦被预留，在通过提交点之前无法释放。原因在于，如果一个全有或全无操作释放了共享资源，其他并发线程可能会占用该资源。若撤销全有或全无操作的某些效果需要该资源，释放资源就等于放弃了中止操作的能力。最后，可逆性要求意味着全有或全无操作在提交点之前不应执行任何外部可见的操作，例如打印支票或发射导弹。（虽然限制可以稍宽松些，但这会使情况更复杂。侧边栏9.3探讨了这种可能性。）

相比之下，提交后阶段可以公开结果，释放不再需要的预留资源，并执行外部可见的操作，如打印支票、打开现金抽屉或钻孔。但它不能尝试获取额外资源，因为获取尝试可能会失败，而提交后阶段不允许有失败的余地。提交后阶段必须仅限于完成那些在预提交阶段就已规划好的活动。

乍看之下，若系统在提交后阶段完成前发生故障，似乎所有努力都将付诸东流，因此确保全有或全无原子性的唯一方法就是始终将提交步骤作为整个动作的最后一步。虽然这通常是保证原子性最简单的方式，但实际要求并非如此严苛。提交后阶段的关键特性在于它被封装在实现全有或全无动作的层级内部，因此只要这种延迟对调用层透明，允许提交后阶段在系统故障后*after*完成的方案也是可行的。某些全有或全无原子性方案会通过以下机制实现：确保每次系统故障后都会调用清理程序，或在下次使用数据前预先执行清理——这一切都发生在更高层级有机会察觉异常之前。这个思路应当似曾相识：图9.7中ALL_OR_NOTHING_PUT的实现就采用了该策略，它始终通过名为CHECK_AND_REPAIR的清理程序完成数据更新前的预处理工作。

一种实现全有或全无原子性的流行技术被称为*shadow copy*。它被文本编辑器、编译器、日历管理程序以及其他修改现有文件的程序所采用，以确保在系统故障后，用户不会得到损坏的数据或仅包含部分预期更改的数据：

- 预提交：创建待修改文件的完整工作副本。然后，对该工作副本进行所有更改。

边栏9.3：级联中止*(Temporary) sweeping simplification*。在关于提交点的初步讨论中，我们有意避开了一种更复杂且设计难度更大的可能性。某些系统允许其他并发活动查看待定的结果，甚至可能在提交前允许执行外部可见的操作。因此，这些系统必须准备好追踪并中止这些并发活动（这种追踪称为*cascaded abort*），或执行*compensating*外部操作（例如，发送信件要求退回支票或为导弹发射道歉）。第10章[在线]中关于层次结构和多站点的讨论引入了级联中止的一个简单版本。

•提交点：谨慎地将工作副本与原始版本进行交换。通常，这一步是通过文件系统的低层RENAME入口点引导完成的，该入口点提供类似原子性的保证，例如第2.5.8节中针对UNIX版RENAME 所描述的那些保证。

•提交后：释放原占用的空间。

图9.7中的ALL_OR_NOTHING_PUT算法可视为影子拷贝策略的一个具体实例，而该策略本身又是通用预提交/后提交规范的一个特例。提交点发生在*S2*新值成功写入磁盘的瞬间。在预提交阶段，当ALL_OR_NOTHING_PUT检查三个扇区并写入影子拷贝*S1*时，系统崩溃不会留下该活动的任何痕迹（即后续调用ALL_OR_NOTHING_GET时无法发现的痕迹）。ALL_OR_NOTHING_PUT的后提交阶段则包含写入*S3*的操作。

从这些例子中我们可以提炼出一个重要的设计原则：

原子性的黄金法则

Never modify the only copy!

为了使一个复合动作具有“全有或全无”的特性，必须存在某种方式能够逆转其预提交阶段各组成动作的效果，这样若动作未能提交，就有办法回退。随着我们继续探索“全有或全无”原子性的实现方式，会发现正确的实现最终总是归结为创建影子副本。原因在于这种结构确保了实现遵循黄金法则。

9.2.3 系统性全有或全无原子性：版本历史

本节提出了一种方案，旨在为修改任意数据结构的通用程序提供全有或全无的原子性保障。该方案的正确性显而易见，但其机制可能影响性能。本章第9.3节将介绍该方案的变体，其正确性需要更多思考来理解，但允许更高性能的实现。如前所述，我们目前专注于全有或全无的原子性。虽然前后原子性的某些方面也会显现，但我们把对该主题的系统性讨论留到本章第9.4和9.5节。因此，本节需牢记的模型是仅有一个线程在运行。若系统崩溃，重启后原线程将消失——回忆第8章[在线]所述*sweeping simplification*，线程属于易失状态，崩溃时丢失，仅持久状态得以保留。崩溃后，一个新的不同线程会尝试查看数据。目标是确保新线程总能发现崩溃时正在进行全有或全无操作要么从未开始，要么已成功完成。

在探讨一般情况时，一个根本性难题浮现出来：随机存取存储器和磁盘对程序员而言，通常表现为一组被命名、共享且可重写的存储单元，称为*cell storage*。存储单元具有这样的语义特性——实际上极难实现全有或全无的操作，因为存储行为会破坏旧数据，从而可能违反原子性的黄金法则。若后续全有或全无操作中止，旧值将不可挽回地丢失；至多只能通过其他位置保留的信息进行重建。此外，数据一旦存储就会暴露给后续线程查看，无论存储该值的全有或全无操作是否已达到提交点。若全有或全无操作恰好只产生一个输出值，那么将该值写入存储单元可作为提交机制，此时不存在问题。但如果结果应由多个输出值构成，且这些值需同时对外可见，则较难构建全有或全无操作。一旦首个输出值被存储，剩余输出的计算就必须成功；此时已无退路。若系统发生故障而我们未加防范，后续线程可能会看到部分旧值与部分新值混杂的情况。

这些细胞存储的局限并未困扰帕多瓦的商人们，他们在14世纪发明了复式记账法。他们的存储介质是装订成册的纸张，用羽毛笔记录新条目。他们从不擦除甚至划掉错误的记录；一旦出错，就另作一笔反向冲销的条目，从而在账簿中完整保留操作、错误及更正的痕迹。直到20世纪50年代，当程序员开始将记账系统自动化时，覆写数据的概念才出现。在此之前，若记账员在记录时猝死，其他人总能无缝接管账簿。这种纸质系统稳健性的观察启示我们，或许存在一种原子性黄金法则的形式，让人能够系统化操作：永不抹去任何记录。

研究文本编辑器所使用的影子副本技术，为我们提供了第二个实用的思路。该机制的核心在于：文本编辑器能够对文件进行多次修改，却能在完成前不显露任何改动，其奥秘在于——其他潜在文件读取者只能通过文件名访问文件。在提交之前，编辑器操作的文件副本要么尚未命名，要么拥有线程外不可知的唯一名称，因此修改后的副本实际上处于不可见状态。通过重命名新版本这一步骤，所有更新内容便能同时呈现给后续的读取者。

这两点观察表明，全有或全无操作更适合采用一种与单元存储行为相异的存储模型：不同于旧数据被存储操作直接覆盖的模型，我们转而创建数据的新暂定版本，使得该版本在此全有或全无操作提交前，对任何外部读取者均不可见。即便从传统单元内存出发，我们仍能通过介入一个位于单元存储与读写数据的程序之间的中间层，来提供此类语义。该层实现了所谓的*journal storage*。日志存储的基本理念直截了当：我们为每个命名变量关联的并非单一单元，而是非易失性存储中的单元列表；列表中的值记录了变量的历史。图9.10对此进行了说明。每当任一操作

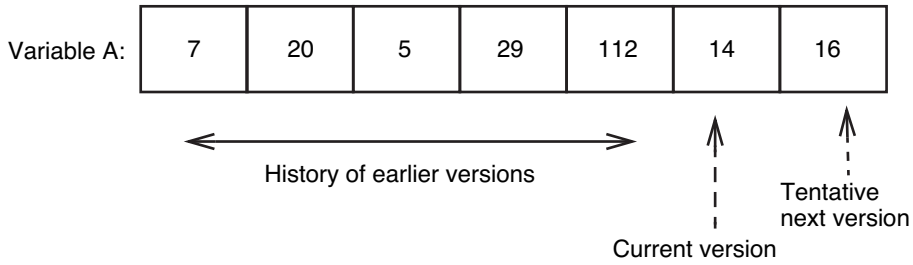


图9.10

变量在日志存储中的版本历史。

提议向变量写入新值时，日志存储管理器会将预期的新值追加到列表末尾。显然，这种保留历史记录的方法有望发挥作用，因为如果一个全有或全无操作中止，可以设想一种系统化的方法来定位并丢弃它所写入的所有新版本。此外，我们可以指示日志存储管理器准备接收暂定值，但只有在创建它们的全有或全无操作提交后，才予以采纳。实现这种预期的基础机制相当简单：日志存储管理器应在每个新版本旁边记录创建它的全有或全无操作的标识。之后，它随时可以通过查询该全有或全无操作是否已提交，来发现暂定版本的状态。

图9.11展示了此类日志存储系统的整体结构，该系统作为隐藏单元存储系统的抽象层实现。（为简化图示，该日志存储系统省略了创建新变量与删除旧变量的调用接口。）在此特定模型中，我们将提供全有或全无编程工具的主要职责赋予日志存储管理器。因此，全有或全无动作的实现者应通过调用日志存储管理器入口NEW_ACTION来启动该动作，随后通过调用COMMIT或ABORT来完成动作。若动作执行过程中所有数据读写操作均通过调用日志存储管理器的READ_CURRENT_VALUE和WRITE_NEW_VALUE入口完成，我们期望该动作将自动具备全有或全无特性，无需实现者额外关注。

这种自动的全有或全无原子性如何实现？第一步是，日志存储管理器在NEW_ACTION被调用时，应为预期的全有或全无操作分配一个唯一标识符，并在非易失性存储单元中创建一条记录，记载这一新标识符及新全有或全无操作的状态。该记录被称为*outcome record*；它初始存在于PENDING状态；根据操作结果，它最终应迁移至COMMITTED或ABORTED状态之一，如图9.12所示。除最终废弃结果记录外，不允许其他状态转换。

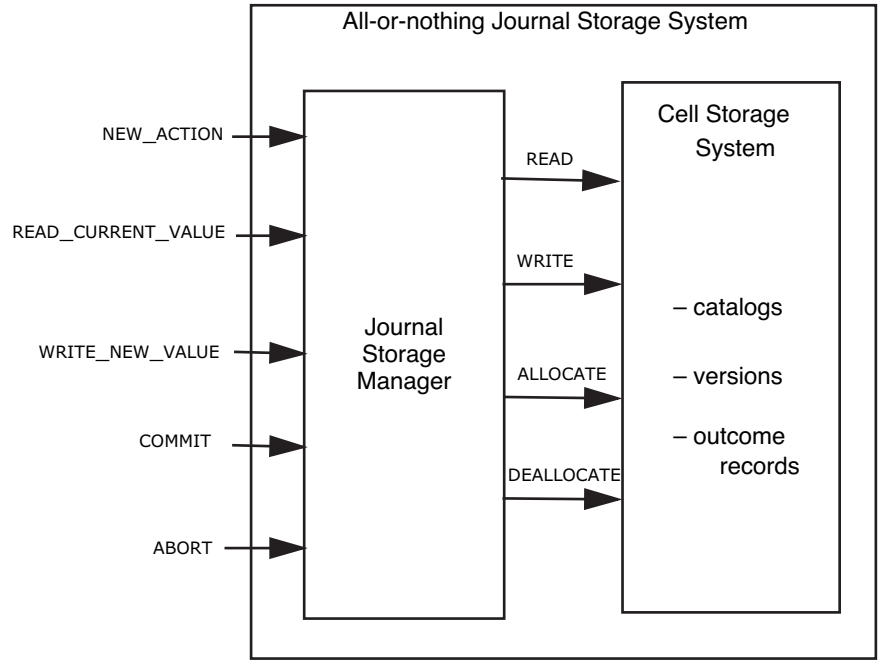


图9.11

基于版本历史和日志存储的全有或全无存储系统的接口与内部组织结构。

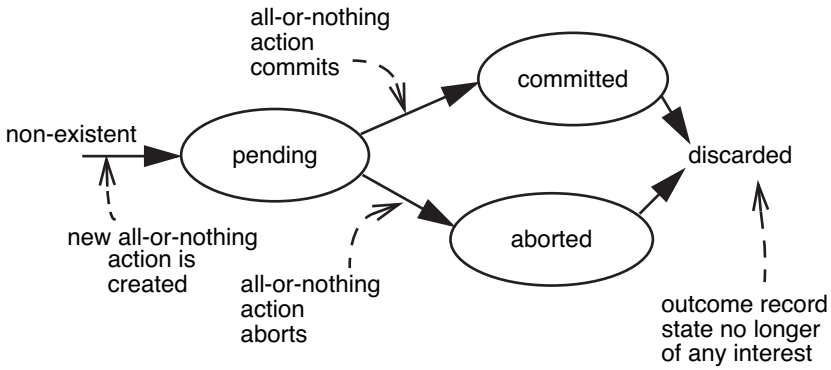


图9.12

The allowed state transitions of an outcome record.

```
1  procedure NEW_ACTION ()
2      id ← NEW_OUTCOME_RECORD ()
3      id.outcome_record.state ← PENDING
4      return id

5  procedure COMMIT (reference id)
6      id.outcome_record.state ← COMMITTED

7  procedure ABORT (reference id)
8      id.outcome_record.state ← ABORTED
```

图9.13

程序 NEW_ACTION、 COMMIT 和 ABORT。

对其状态已无进一步兴趣。图9.13展示了三个程序NEW_ACTION、 COMMIT和ABORT的实现。

当一个全有或全无操作调用日志存储管理器写入某个数据对象的新版本时，该操作需提供数据对象的标识符、新版本的暂定值以及全有或全无操作的标识符。日志存储管理器会调用底层存储管理系统，在非易失性单元存储中分配足够空间以容纳新版本；随后将新数据值及全有或全无操作的标识符存入新分配的存储单元。如此，日志存储管理器便创建了如图9.14所示的版本历史。此时，

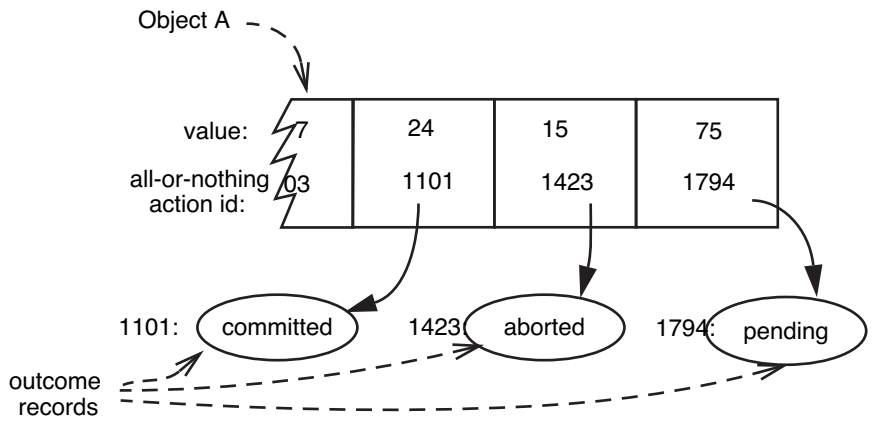


图9.14

版本历史的一部分，附有结果记录。某个线程最近调用了WRITE_NEW_VALUE，指定了 *data_id = A*、*new_value = 75*和*client_id = 1794*。调用READ_CURRENT_VALUE的调用者将读取到A的值为24。

```
1  procedure READ_CURRENT_VALUE (data_id, caller_id)
2      starting at end of data_id repeat until beginning
3          v ← previous version of data_id        // Get next older version
4          a ← v.action_id  // Identify the action a that created it
5          s ← a.outcome_record.state             // Check action a's outcome record
6          if s = COMMITTED then
7              return v.value
8          else skip v                            // Continue backward search
9      signal ("Tried to read an uninitialized variable!")

10 procedure WRITE_NEW_VALUE (reference data_id, new_value, caller_id)
11     if caller_id.outcome_record.state = PENDING
12         append new version v to data_id
13         v.value ← new_value
14         v.action_id ← caller_id
                   else signal ("Tried to write outside of an all-or-nothing action!")
```

图9.15

遵循READ_CURRENT_VALUE和WRITE_NEW_VALUE的算法。参数*caller_id*是由NEW_ACTION返回的动作标识符。在此版本中，仅WRITE_NEW_VALUE使用了*caller_id*。后续，READ_CURRENT_VALUE也将采用它。

当有人提议通过调用READ_CURRENT_VALUE,来读取数据值时，日志存储管理器可以审查版本历史，从最新版本开始，并返回最近提交版本中的值。通过检查结果记录，日志存储管理器可以忽略那些由全有或全无动作写入的版本，这些动作要么已中止，要么从未提交。

过程READ_CURRENT_VALUE和WRITE_NEW_VALUE因此遵循图9.15中的算法。这对算法的重要特性在于：若当前全有或全无操作在抵达COMMIT调用前被意外中断，其所创建的新版本对READ_CURRENT_VALUE调用者不可见。（这些版本对写入它们的全有或全无操作本身也不可见。由于全有或全无操作有时需要读取其暂存写入的内容，可通过另一个名为READ_MY_PENDING_VALUE的过程实现——该过程除第6行,的判定条件不同外与READ_CURRENT_VALUE 完全相同。）此外，例如当全有或全无操作99在修改十九个数据对象的过程中崩溃时，所有十九项修改对后续READ_CURRENT_VALUE调用者均不可见。若全有或全无操作99成功执行至COMMIT调用，该调用将在将结果记录从PENDING 改为COMMITTED的瞬间，以原子方式同时提交全部变更集。待定版本对任何使用READ_CURRENT_VALUE读取数据的并发操作同样不可见，这一特性将在引入并发线程讨论前后原子性时发挥重要作用，但目前我们仅需...

```
1   procedure TRANSFER (reference debit_account, reference credit_account,
                                                                  amount)
2       my_id ← NEW_ACTION ()
3       xvalue ← READ_CURRENT_VALUE (debit_account, my_id)
4       xvalue ← xvalue - amount
5       WRITE_NEW_VALUE (debit_account, xvalue, my_id)
6       yvalue ← READ_CURRENT_VALUE (credit_account, my_id)
7       yvalue ← yvalue + amount
8       WRITE_NEW_VALUE (credit_account, yvalue, my_id)
9       if xvalue > 0 then
10          COMMIT (my_id)
11      else
12          ABORT (my_id)
13          signal("Negative transfers are not allowed.")
```

图9.16

一种基于日志存储的全有或全无TRANSFER过程。（此程序假设它是唯一运行的线程。将转账过程设为原子操作，因为其他线程可能同时更新同一账户，这需要本章后续讨论的额外机制。）

担忧在于系统崩溃可能会阻止当前线程提交或中止操作，而我们希望确保后续线程不会遇到部分结果。如同第9.2.1节中的日历管理器案例，我们假设当崩溃发生时，任何正在进行中的全有或全无操作都由某个外部代理监管，该代理意识到崩溃已发生，会利用{v*}来查明情况，并在必要时启动替代的全有或全无操作。

图9.16展示了第9.1.5节中的TRANSFER流程通过版本历史机制被重新编程为一个全有或全无（但暂时不考虑先后顺序）的操作。这一TRANSFER的实现比之前的版本更为精细——它会检查待扣款账户是否有足够资金完成转账，若不足则中止操作。转账流程中的步骤顺序除计算正确结果外，几乎不受任何其他因素约束。例如，读取*credit_account*的操作可以随意调整到NEW_ACTION与重新计算*yvalue*之间的任意位置。我们由此得出结论：日志存储系统使得预提交规则的执行负担比预期轻得多。

还有一个未解决的问题：对版本历史的更新和结果记录的变更必须是全有或全无的。也就是说，如果系统在线程处于WRITE_NEW_VALUE阶段（调整结构以追加新版本）或COMMIT 阶段（更新结果记录）时发生故障，正在写入的单元格状态必须保持清晰；它要么保持崩溃前的原状，要么完全变更为预期的新值。解决方案在于设计日志存储内部结构的所有修改操作，确保它们能够

可以通过覆盖单个单元格来完成。例如，假设一个拥有版本历史的变量名指向一个包含最新版本地址的单元格，且各版本通过地址引用从最新版本向后链接。添加新版本包括为新版本分配空间、读取前一个版本的当前地址、将该地址写入新版本的后向链接字段，然后更新描述符为新版本的地址。最后这一更新操作只需覆盖单个单元格即可实现。同理，将结果记录从PENDING更新为COMMITTED 也只需覆盖单个单元格。

作为第一个引导步骤，我们已经将创建全有或全无动作的普遍问题简化为对单个单元格. 进行全有或全无覆盖的具体问题。在剩余的引导步骤中，回顾已知两种实现单单元格全有或全无覆盖的方法：应用图9.7中的ALL_OR_NOTHING_PUT流程（若存在并发操作，版本历史内部结构的更新同样需满足前或后原子性。第9.4节将探讨实现这一保障的方法。）

9.2.4 版本历史的使用方式

细心的读者可能会注意到，关于刚刚描述的版本历史方案，有两处可能令人困惑的地方。当我们讨论本章第9.4节中的并发性和前后原子性时，这两点将变得不那么令人费解：

1. 由于READ_CURRENT_VALUE 会跳过属于任何其他全有或全无动作的版本（这些动作的OUTCOME记录未处于COMMITTED状态），因此当全有或全无动作中止时，实际上并不需要更改OUTCOME记录；该记录可以无限期地保持在PENDING状态。然而，当我们引入并发性时，会发现一个待处理的动作可能会阻止其他线程读取该动作创建了新版本的变量，因此区分已中止的动作与真正仍处于待处理状态的动作将变得至关重要。

2. 正如我们定义了READ_CURRENT_VALUE，比最近提交版本更早的版本将无法访问，它们完全可以被丢弃。丢弃操作可以作为日志存储管理器的一个额外步骤来实现，或者作为独立垃圾回收活动的一部分。另一种情况是，这些旧版本可能作为历史记录（称为*archive*）而具有价值，只需在提交记录上添加时间戳，并配备能够定位并返回过去特定时间创建的旧值的程序即可。因此，版本历史系统有时被称为*temporal database*，或被认为提供了*time domain addressing*。银行业中存在大量利用历史信息的需求，例如报告所有银行账户余额的一致总和、在每月15日根据当月1日的余额支付利息，或计算上月的平均余额。当我们讨论并发性时，还会出现另一个不立即丢弃旧版本的原因。

前后原子性：为了正确性，并发线程可能需要在创建并提交新版本后仍读取旧版本。

直接实现版本历史会引发对性能的担忧：与简单地读取一个命名的存储单元不同，必须至少通过一个间接引用的描述符来定位包含当前版本的存储单元。如果存储设备是磁盘，这种额外的引用可能成为性能瓶颈，尽管可以通过缓存来缓解。更难以缓解的瓶颈出现在更新操作上。每当应用程序写入新值时，日志存储层必须在未使用的存储空间中分配位置，写入新版本，并更新版本历史描述符，以便后续读取者能找到新版本。这一过程可能需要多次磁盘写入。这些额外的磁盘写入可以被隐藏在日志存储层内部，通过巧妙设计延迟到提交时批量处理，但它们依然存在开销。当存储访问延迟成为性能瓶颈时，额外的访问操作会拖慢整体速度。

因此，版本历史主要应用于低性能场景。一个常见例子是用于协调程序开发团队的修订管理系统。程序员“签出”一组文件进行修改后“签入”结果。签出与签入操作具有原子性——签入操作会将每个被修改文件确立为该文件完整历史中的最新版本，以便后续发现问题时追溯（签入操作还会验证文件在签出期间未被他人修改，此举能捕获部分而非全部协作错误）。第二个例子体现在某些交互式应用中，如文字处理或图像编辑系统提供的“深度撤销”功能，允许用户发现近期编辑不当后回退至早期满意状态。第三个例子是某些文件系统会在应用程序以写入模式打开现有文件时自动创建新版本：当应用程序关闭文件时，系统会为旧版本文件名添加数字后缀，并将原名称赋予新版本。这些界面采用版本历史机制，因其易于用户理解，且能在系统故障和用户错误时提供原子性保障。多数此类应用还提供存档功能，既便于参考，也能回滚至已知稳定版本。

需要高性能的应用则是另一回事。它们同样要求全有或全无的原子性，但通常通过采用一种称为*log*的专门技术来实现。日志是我们接下来要讨论的主题。

9.3 全有或全无原子性II：实用考量

诸如航空订票系统或银行系统之类的数据库管理应用通常需要高性能以及全有或全无的原子性，因此其设计者采用简化的原子性技术。这些技术中最重要的一点是将数据的读写操作与故障恢复机制严格分离。

其核心思想是最小化最常见操作（应用程序读取和更新）所需的存储访问次数。代价则是那些极少执行的操作（故障恢复，人们希望这类操作即便存在也应极少发生）可能无法实现最小化存储访问。该技术被称为*logging*。日志记录不仅用于保证原子性，还被用于其他目的，其中几个用途在侧边栏9.4中有所阐述。

9.3.1 原子性日志

原子性日志记录的基本思想是将日志存储的全有或全无原子性与单元存储的速度相结合，通过让应用程序对数据的每次变更进行两次记录来实现。应用程序首先在日志存储中*logs*变更，随后在单元存储中*installs* 变更*。有人可能会认为，将数据写入两次比仅写入版本历史一次成本更高，但这种分离允许专门的优化，从而可能使整个系统运行得更快。

首次记录至日志存储时，通过为所有变量创建单一交错版本历史（称为*log*，优化了快速写入性能。描述每次数据更新的信息形成一条记录，应用程序将其追加到日志末尾。由于仅存在单一日志，仅需一个指向日志末端的指针即可定位系统中任意变量变更记录的追加位置。若日志介质为磁盘，且该磁盘专用于日志记录，同时磁盘存储管理系统连续分配扇区，则磁盘寻道臂仅在柱面写满时才需移动，从而消除大多数寻道延迟。后续将看到，恢复过程确实涉及扫描日志（这一操作成本较高），但恢复应属罕见事件。因此，采用日志正是遵循*optimize for the common case*提示的范例。

第二次记录，即对单元存储的写入，经过优化以实现快速读取：应用程序只需覆盖该变量之前的单元存储记录即可完成安装。保存在单元存储中的记录可视为一种缓存，读取时无需费力在日志中定位最新版本。此外，由于不从日志读取，日志磁盘的寻道臂可保持原位，随时准备下一次更新。这两个步骤，LOG和INSTALL，成为图9.11中WRITE_NEW_VALUE接口的另一种实现方式。图9.17展示了这种两步实现方案。

其核心理念在于，日志是记录操作结果的权威依据。单元存储仅作为参考副本；若发生丢失，可从日志中重建。在单元存储中安装副本的目的，是为了同时提升日志记录和读取的速度。通过双重记录数据，我们得以在写入时获得高性能，在读取时同样高效，同时确保操作的原子性——要么全部完成，要么全部不执行。

有三种常见的日志配置，如图9.18所示。在这三种配置中，日志均驻留在非易失性存储介质中。对于*in-memory*

* A hardware architect would say "…it *graduates* the change to cell storage". This text, somewhat arbitrarily, chooses to use the database management term "install" .

侧边栏 9.4：日志的多种用途　日志是一种以追加新记录为主要使用方式的对象。日志实现通常提供按从旧到新或相反顺序读取条目的过程，但通常不提供修改先前条目的任何过程。日志被用于几种截然不同的目的，而这些用途的范围有时会在实际设计和实现中造成混淆。以下是日志最常见的一些用途：

1. *Atomicity log*。如果记录下一个全有或全无操作的组件动作，并附带足够的操作前后信息，那么崩溃恢复过程就能够撤销（从而回滚）那些未能完成的全有或全无操作的效果，或者完成那些已提交但未能记录其全部效果的全有或全无操作。

2. *Archive log*。如果日志被无限期保留，它便成为一个可以存储数据旧值以及系统或其应用程序所采取操作序列以供审查的地方。存档信息有多种用途：监测故障模式、审查系统在安全漏洞发生前后及期间的行为、从应用层错误中恢复（例如，职员误删账户）、历史研究、欺诈控制，以及满足记录保存的合规要求。

3. *Performance log*。大多数机械存储介质在顺序访问时的性能远高于随机访问。由于日志是按顺序写入的，它们非常适合这类存储介质。通过将待写入的数据结构化为日志形式，可以充分利用介质物理特性的这一匹配优势。当与一个能消除大多数磁盘读取操作的缓存结合使用时，性能日志可以带来显著的速度提升。正如附文所述，原子性日志通常同时也是性能日志。

4. *Durability log*。如果日志存储在非易失性介质上——例如磁带——其故障方式和时间与单元存储介质（可能是磁盘）的故障相互独立，那么日志中的数据副本可作为备份，在单元存储中的数据副本受损时使用。这类日志有助于实现持久化存储。任何使用非易失性介质的日志，无论其设计初衷是为了原子性、归档还是性能，通常也都有助于支持持久性。

在审视或设计日志实现时，务必清晰区分这些不同用途——全有或全无的原子性、归档、性能以及持久化存储——因为它们会导致设计权衡中的优先级差异。当目标是归档时，存储介质的低成本通常比快速访问更为重要，因为归档日志体积庞大但在实践中很少被读取。若以实现持久化存储为目标，则可能需要采用具有不同物理特性的存储介质，以确保故障模式尽可能独立。当追求全有或全无原子性或性能时，最大限度减少存储设备的机械运动就成为高度优先事项。鉴于各类日志存在相互冲突的目标，通常明智的做法是为不同功能实现独立专用的日志。

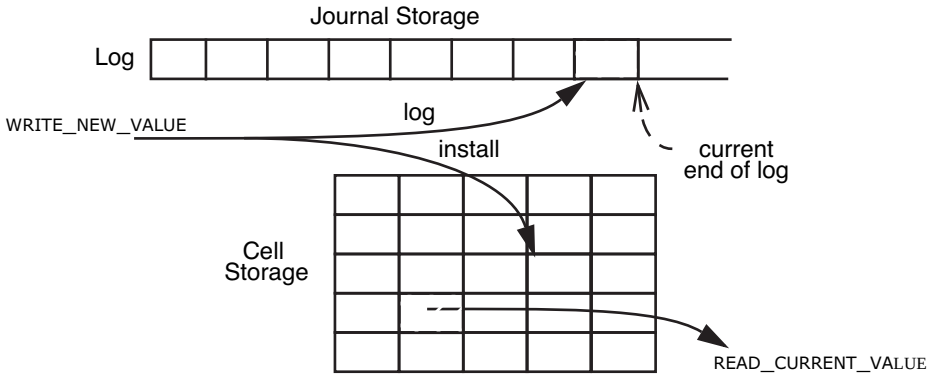


图9.17

记录以实现全有或全无的原子性。应用程序通过首先将新值的记录追加到日志存储中的日志，然后通过覆盖方式在单元存储中安装新值，来执行WRITE_NEW_VALUE。应用程序通过仅从单元存储读取，来执行READ_CURRENT_VALUE。

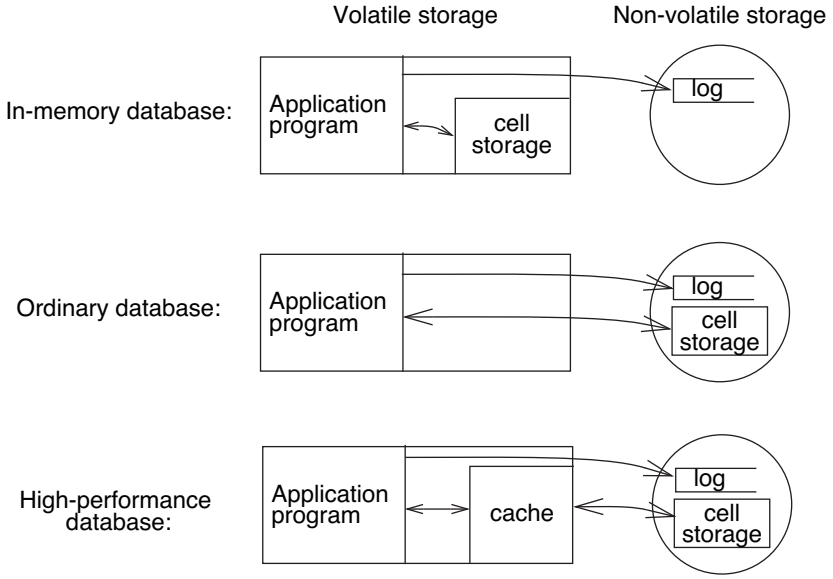


图9.18

三种常见的日志配置。箭头展示了应用程序读取、记录和安装数据时的数据流向。

*database*在第一种常见配置中，单元存储完全位于某种易失性存储介质中。第二种常见配置下，单元存储与日志一同驻留在非易失性存储器内。最后，高性能数据库管理系统通常融合上述两种配置：通过在易失性介质中实现单元存储缓存，并由一个可能独立的多级内存管理算法在缓存与非易失性单元存储之间迁移数据。

记录所有内容两次为全有或全无原子性增添了一个重大复杂性，因为系统可能在更改被记录和安装之间崩溃。为了维持全有或全无原子性，日志系统遵循一个包含两项基本要求的协议。第一项要求是对记录和安装顺序的约束。第二项要求是在每次崩溃后运行一个明确的*recovery*过程。（我们在图9.7中预览了使用恢复过程的策略，该图使用了一个名为CHECK_AND_REPAIR的恢复过程。）

9.3.2 日志协议

存在多种原子性日志，它们在操作顺序和记录信息的细节上各有不同。然而，所有这些日志都涉及图9.17中箭头编号所隐含的顺序约束。该约束是*golden rule of atomicity*（永不修改唯一副本）的一个变体，被称为*write-ahead-log*（预写日志，~~WAL）协议~~：

预写式日志协议

记录更新 ***before*** 的安装过程。

原因在于日志记录是追加操作，而安装则是覆盖操作。如果某个应用违反此协议，在记录更新前就安装更新，随后又因故必须中止，或系统崩溃，就没有系统化的方法来发现已安装的更新，并在必要时撤销它。预写式日志协议确保一旦发生崩溃，恢复程序能通过查阅日志，系统性地找出所有已完成及预期的存储单元变更，并根据实际情况将这些记录恢复为旧值或更新为新值。

原子性日志的基本元素是*log record*。在执行一个全有或全无操作之前，当它要安装一个数据值时，会在日志末尾追加一条类型为CHANGE的新记录。在一般情况下，该记录包含三部分信息（后续我们将看到允许省略第2项或第3项的特殊情况）：

1. 执行更新的全有或全无操作的身份。
2. 一个组件操作，若执行，则会在单元格存储中安装预期值。此组件操作相当于一种保险措施，以防系统崩溃。如果全有或全无操作已提交，但系统在有机会执行安装前崩溃，恢复程序可以执行该安装。

代表该动作。有些系统称这一动作为*do*动作，其他系统则称之为*redo*动作。为了与第3项的助记兼容性，本文将其称为重做动作。

3. 第二个组件动作，如果执行，将逆转计划安装对单元存储的影响。该组件动作被称为*undo*动作，因为如果在安装后，全有或全无动作中止或系统崩溃，恢复过程可能需要逆转(*undo*)安装的影响。

应用程序通过调用下层过程LOG追加日志记录，该过程本身必须是原子性的。LOG过程是另一个引导示例：从本章前文所述的ALL_OR_NOTHING_PUT出发，日志设计者创建了一个通用的LOG 过程，应用程序开发者随后可利用LOG 过程，为任何设计合理的复合动作实现全有或全无的原子性。

如图9.17所示，LOG和INSTALL实现了图9.11接口中WRITE_NEW_VALUE部分的日志记录功能，而READ_CURRENT_VALUE则直接来自单元存储的READ。我们还需要为图9.11接口的其余部分实现日志记录功能。实现NEW_ACTION的方法是记录一个仅包含新全有或全无操作标识的BEGIN记录。当全有或全无操作进入预提交阶段时，它会记录CHANGE记录。要实现COMMIT或ABORT，全有或全无操作会记录一个OUTCOME记录，该记录将成为操作结果的权威指示。全有或全无操作记录OUTCOME记录的瞬间即为提交点。例如，图9.19展示了我们现已熟悉的采用日志机制实现的TRANSFER操作。

由于日志是操作行为的权威记录，全有或全无操作可以在任何符合预写日志协议（write-ahead-log protocol）的适当时机，向单元存储执行安装操作，这一过程既可在记录OUTCOME之前，也可在其后进行。操作的最终步骤是记录一个END 条目，该条目同样仅包含操作标识，用以表明该操作已完成所有安装任务。（记录全部四种活动类型——BEGIN, CHANGE、OUTCOME, 和END———虽比实际需求更为通用。如后文所述，某些日志系统能够合并记录，例如将OUTCOME与END合并，或将BEGIN与首个CHANGE合并。）图9.20展示了三条日志记录的示例，其中两条是典型全有或全无TRANSFER操作的CHANGE记录，它们与另一可能完全无关的全有或全无操作的OUTCOME记录交错排列。

将结果安装到单元存储的一个后果是，对于全有或全无操作而言，若要中止它，可能需要进行一些清理工作。此外，如果系统非自愿地终止了一个正处于全有或全无操作中的线程（例如，因为该线程陷入了死锁或无限循环），那么必须由该不幸线程之外的某个实体来负责清理工作。如果省略了这一清理步骤，全有或全无操作可能会无限期地保持挂起状态。系统不能简单地无限期忽略这些挂起的操作，因为由其他线程发起的全有或全无操作很可能需要使用被终止操作所更改的数据。（这实际上是一个{v*}

```
1   procedure TRANSFER (debit_account, credit_account, amount)
2       my_id ← LOG (BEGIN_TRANSACTION)
3       dbvalue.old ← GET (debit_account)
4       dbvalue.new ← dbvalue.old - amount
5       crvalue.old ← GET (credit_account, my_id)
6       crvalue.new ← crvalue.old + amount
7       LOG (CHANGE, my_id,
8           "PUT (debit_account, dbvalue.new)",              //redo action
9           "PUT (debit_account, dbvalue.old)" )             //undo action
10      LOG ( CHANGE, my_id,
11          "PUT (credit_account, crvalue.new)"              //redo action
12          "PUT (credit_account, crvalue.old)")             //undo action
13      PUT (debit_account, dbvalue.new)                     // install
14      PUT (credit_account, crvalue.new)                    // install
15      if dbvalue.new > 0 then
16          LOG ( OUTCOME, COMMIT, my_id)
17      else
18          LOG (OUTCOME, ABORT, my_id)
19          signal("Action not allowed. Would make debit account negative.")
20      LOG (END_TRANSACTION, my_id)
```

图9.19

一种全有或全无的TRANSFER程序，通过日志记录实现。

前后原子性关注点（这是全有或全无原子性与前后原子性相交的地方之一。）

如果被中止的操作执行了任何安装，这些安装仍保留在单元存储中，因此仅仅在日志中追加一条OUTCOME记录，声明操作已中止，并不足以让后续观察者认为这个全有或全无的操作什么都没做。解决这一问题的办法是执行一个通用的ABORT过程。该ABORT过程

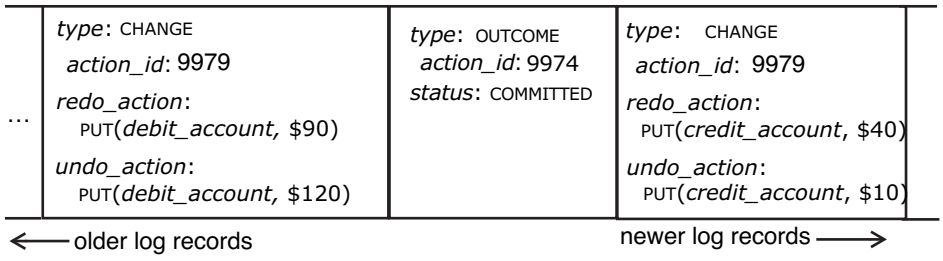


图9.20

原子性日志的一个片段示例，展示了针对动作ID为9979的TRANSFER操作的两条CHANGE记录，以及另一不同全有或全无动作的OUTCOME记录。

dure将所有由全有或全无操作设置的单元格存储变量恢复至其旧值。ABORT过程简单地逆向扫描日志，寻找由该全有或全无操作创建的日志条目；对于发现的每一个CHANGE记录，它执行记录的*undo_action*,操作，从而恢复单元格存储中的旧值。当ABORT过程找到该全有或全无操作的BEGIN记录时，逆向搜索终止。图9.21对此进行了说明。

撤销单元格存储安装所需的额外工作，当全有或全无操作中止时，是*optimizing for the common case*的另一个例证：人们预期大多数全有或全无操作将会提交，而中止操作应相对罕见。偶尔回滚单元格存储值的额外努力（但愿如此）将通过更频繁地在更新、读取和提交操作上获得的性能提升得到超额补偿。

9.3.3 恢复流程

预写式日志协议是日志系统中两个必需协议要素中的第一个。第二个必需协议要素是，在每次系统崩溃后，系统必须运行恢复程序，才能允许普通应用程序使用数据。恢复程序的具体细节取决于日志与单元存储针对易失性和非易失性内存的特定配置。

首先考虑图9.18中内存数据库的恢复。由于系统崩溃可能损坏易失性内存中的任何内容，包括单元存储状态和当前运行线程的状态，重启崩溃系统通常从重置所有易失性内存开始。这种重置的效果是放弃所有单元

```
1  procedure ABORT (action_id)
2      starting at end of log repeat until beginning
3          log_record ← previous record of log
4          if log_record.id = action_id then
5              if (log_record.type = OUTCOME)
6                  then signal ("Can't abort an already completed action.")
7              if (log_record.type = CHANGE)
8                  then perform undo_action of log_record
9              if (log_record.type = BEGIN)
10                 then break repeat
11     LOG (action_id, OUTCOME, ABORTED)                    // Block future undos.
12     LOG (action_id, END)
```

图9.21

Generic ABORT procedure for a logging system. The argument *action_id* identifies the action to be aborted. An atomic action calls this procedure if it decides to abort. In addition, the operating system may call this procedure if it decides to terminate the action, for example to break a deadlock or because the action is running too long. The LOG procedure must itself be atomic.

```
1  procedure RECOVER () // Recovery procedure for a volatile, in-memory database.
2      winners ← NULL
3      starting at end of log repeat until beginning
4          log_record ← previous record of log
5          if (log_record.type = OUTCOME)
6              then winners ← winners + log_record                    // Set addition.

7      starting at beginning of log repeat until end
8          log_record ← next record of log
9          if (log_record.type= CHANGE)
10             and (outcome_record ← find (log_record.action_id) in winners)
11             and (outcome_record.status = COMMITTED) then
12             perform log_record.redo_action
```

图9.22

一种针对内存数据库的幂等仅重做恢复过程。由于RECOVER 仅写入易失性存储，如果在运行过程中发生崩溃，再次运行它是安全的。

数据库的存储版本以及崩溃时正在进行的所有全有或全无操作。另一方面，由于日志驻留在非易失性日志存储上，它不受崩溃影响，应仍保持完整。

最简单的恢复过程会对日志进行两次扫描。第一次扫描时，它从最后一条记录开始逆向检查日志*backward*，因此对于每个全有或全无操作，恢复过程首先遇到的证据将是该操作最后记录的日志条目。这种逆向日志扫描有时被称为LIFO（后进先出）式日志审查。在逆向扫描过程中，恢复程序会收集所有在崩溃前记录过OUTCOME条目的全有或全无操作的身份标识和完成状态，存入一个集合中。这些操作，无论已提交还是已中止，都被称为*winners*。

当后向扫描完成时，获胜者集合也随之确定，恢复过程随即开始对日志进行前向扫描。之所以需要前向扫描，是因为崩溃后的重启完全重置了单元存储。在前向扫描过程中，恢复程序按照日志中的顺序，执行每个获胜者的REDO 动作——只要其OUTCOME记录表明该动作COMMITTED。这些REDO操作会将所有已提交的值重新载入单元存储，因此扫描结束时，恢复程序已将单元存储恢复到理想状态。这一状态如同所有在崩溃前提交的全有或全无动作均已完成，而所有中止或在崩溃时仍处于待定状态的全有或全无动作从未存在过。此时数据库系统便可重新开放进行正常操作。图9.22对此进行了说明。

这一恢复流程强调了一个观点：日志可被视为整个数据库的权威版本，足以完整重构单元存储中的参考副本{v*}。

在某些情况下，当数据的持久性要求极低时，这种恢复流程可能显得过于繁琐。例如，全有或全无的操作可能

对易失性存储中的软状态进行了一系列更改。如果崩溃导致软状态完全丢失，也无需重新执行安装操作，因为软状态的定义就是应用程序能够在崩溃后重建新的软状态。换句话说，在“全有”或“全无”的选择中，当数据全是软状态时，崩溃后“全无”始终是一个合适的结果。

恢复过程的一个关键设计特性是，即便在恢复期间系统再次崩溃，也必须能够继续恢复。此外，必须确保无论发生多少次崩溃-重启循环，都不会影响最终结果的正确性。其方法是将恢复过程设计为*idempotent*。也就是说，设计时要确保即使恢复过程被中断并从开头重新启动，其产生的结果与完整运行至结束时的结果完全相同。对于内存数据库配置而言，这一目标很容易实现：只需确保恢复过程仅修改易失性存储。这样，若恢复过程中发生崩溃，易失性存储的丢失会自动将系统状态恢复到恢复开始时的状态，从而可以安全地从头重新运行恢复过程。只要恢复过程最终完成，无论中间经历了多少次中断和重启，数据库的单元存储副本状态都将保持正确。

ABORT 过程同样需要是幂等的，因为如果一个全有或全无操作决定中止，并且在执行ABORT过程中某些计时器到期，系统可能会决定终止并为同一个全有或全无操作调用ABORT。如果各个撤销操作本身是幂等的，那么图9.21中的中止版本将满足这一要求。

9.3.4 其他日志配置：非易失性单元存储

将单元格存储置于易失性内存是一种*sweeping simplification*，对于中小型数据库运行良好，但某些数据库规模过大，这种做法便不切实际。因此，设计者发现有必要将单元格存储置于更廉价、非易失性的存储介质上，例如图9.18第二种配置中的磁盘。然而，非易失性存储介质使得安装操作能在系统崩溃后保留，这样一来，内存数据库所采用的简单恢复程序便暴露出两大缺陷：

1. 如果在系统崩溃时，存在一些已安装更改的“全有或全无”待处理操作，这些更改将在系统崩溃后保留。恢复过程必须撤销这些更改的影响，就像这些操作已中止一样。 2. 该恢复过程会重新安装整个数据库，尽管在此情况下，大部分数据可能在非易失性存储中完好无损。如果数据库规模庞大，需要使用非易失性存储来容纳，那么每次恢复时都毫无必要地重新完整安装整个数据库，其成本很可能是无法接受的。

此外，对非易失性单元存储器的读写操作往往较慢，因此设计者几乎总是会在易失性存储器中安装一个缓存，并配备一个

多级内存管理器，从而转向图9.18的第三种配置。但这一新增又引入了另一个缺点：

3. 在多级存储系统中，数据从易失性层级写入非易失性层级的顺序通常由多级存储管理器控制，该管理器可能运行诸如最近最少使用算法等策略。因此，在系统崩溃的瞬间，一些被认为已安装的数据可能尚未迁移至非易失性存储器中。

为了暂缓考虑这一缺陷，让我们暂且假设多级内存管理器实现了写透缓存（write-through cache）。（下文第9.3.6节将重新讨论非写透缓存的情况。）采用写透缓存时，我们可以确信应用程序已安装的所有内容都已写入非易失性存储。这一假设暂时将第三个缺陷从我们关注的问题列表中移除，此时的情况就如同我们使用图9.18中无缓存的“普通数据库”配置一样。但我们仍需解决前两个缺陷，同时必须确保修改后的恢复过程仍保持幂等性。

为了解决第一个缺陷，即数据库中可能包含需要撤销操作的安装记录，我们需要修改图9.22中的恢复流程。在恢复过程执行初始反向扫描时，不再寻找成功操作，而是将所有在崩溃时仍在进行中的全有或全无操作的身份信息收集到一个集合中。这个集合中的操作被称为*losers*，它们可能包括已提交和未提交的操作。失败操作很容易识别，因为在反向扫描中遇到的第一个包含其身份信息的日志记录将不是END记录。为了识别失败操作，伪代码在一个名为*completeds*的辅助列表中跟踪哪些操作记录了END记录。当RECOVER遇到一个不属于*completed*的操作的日志记录时，它会将该操作添加到名为*losers*的集合中。此外，在反向扫描过程中，每当恢复过程遇到属于失败操作的CHANGE记录时，它会执行记录中列出的UNDO 操作。通过这种后进先出的日志审查，所有由失败操作执行的安装都将被回滚，单元格存储的状态将如同这些失败的全有或全无操作从未开始过一样。接下来，RECOVER执行日志的正向扫描，对已提交的全有或全无操作执行重做操作，如图9.23所示。最后，恢复过程为失败操作列表中的每一个全有或全无操作记录一个END记录。这个END记录将失败操作转化为已完成操作，从而确保未来的恢复过程会忽略它，不会再次执行其撤销操作。让未来的恢复过程忽略已中止的失败操作不仅是一种性能优化，更是为了避免错误地撤销未来全有或全无操作对相同变量所做的更新，这一点至关重要。

与之前一样，恢复过程必须是幂等的，这样如果在恢复过程中发生崩溃，系统可以再次运行恢复程序。除了之前使用的将恢复过程的临时变量置于易失性存储的技术外，每个单独的撤销操作也必须是幂等的。因此，无论是重做

```
1  procedure RECOVER ()// Recovery procedure for non-volatile cell memory
2      completeds ← NULL
3      losers ← NULL
4      starting at end of log repeat until beginning
5          log_record ← previous record of log
6          if (log_record.type = END)
7              then completeds ← completeds + log_record          // Set addition.
8          if (log_record.action_id is not in completeds) then
9              losers ← losers + log_record                 // Add if not already in set.
10             if (log_record.type = CHANGE) then
11                 perform log_record.undo_action

12     starting at beginning of log repeat until end
13         log_record ← next record of log
14         if (log_record.type = CHANGE)
15             and (log_record.action_id.status = COMMITTED) then
16             perform log_record.redo_action

17     for each log_record in losers do
18         log (log_record.action_id, END)                    // Show action completed.
```

图9.23

一种针对执行非易失性单元内存安装系统的幂等撤销/重做恢复流程。在此恢复流程中，*losers* 代表系统崩溃时正在进行的所有或全无操作。

撤销操作通常表示为*blind writes*。盲写是指简单地覆盖数据值，而不参考其先前值。由于盲写本身具有幂等性，无论重复执行多少次，结果始终相同。因此，如果在记录失败者的END记录过程中发生崩溃，立即重新运行恢复程序仍能保持数据库的正确性。那些现在拥有END记录的失败者在重新运行时将被视为已完成，但这没有问题，因为恢复程序的上一次尝试已经撤销了它们的安装。

至于第二个缺点，即恢复过程不必要地重做每一个安装操作，甚至包括那些不属于失败者的安装，我们可以通过分析为何需要重做任何安装来大幅简化（并加速）恢复过程。原因在于，尽管WAL协议要求在安装前记录变更，但提交与安装之间并无必然的顺序要求。在某个已提交的操作记录其END 记录之前，无法确保该操作的任何特定安装已经实际发生。另一方面，任何已记录END记录的已提交操作，其安装过程已经完成。由此得出的结论是，恢复过程无需

```
1  procedure RECOVER ()          // Recovery procedure for rollback recovery.
2      completeds ← NULL
3      losers ← NULL
4      starting at end of log repeat until beginning          // Perform undo scan.
5          log_record ← previous record of log
6          if (log_record.type = OUTCOME)
7              then completeds ← completeds + log_record      // Set addition.
8          if (log_record.action_id is not in completeds) then
9              losers ← losers + log_record                   // New loser.
10             if (log_record.type = CHANGE) then
11                 perform log_record.undo_action

12     for each log_record in losers do
13         log (log_record.action_id, OUTCOME, ABORT)         // Block future undos.
```

图9.24

一种幂等的仅撤销回滚日志恢复过程。

对于任何已记录其END记录且已提交的操作，redo会重新执行安装。一个有益的练习是修改图9.23中的流程，以利用这一观察结果。

如果恢复过程永远不需要重做*any*安装，那就更好了。我们可以通过在应用程序上添加另一个要求来实现这一点：它必须在记录其OUTCOME记录之前完成所有*before*安装。这一要求，加上直写缓存，确保了每个完成的“全有或全无”操作的安装都已安全地存储在非易失性单元存储器中，因此永远不需要执行*any*重做操作。（这也意味着无需记录END记录。）结果是恢复过程只需撤销失败者的安装，并且可以跳过整个前向扫描，从而得到图9.24中更简单的恢复过程。这种方案，因为它只需要撤销操作，有时被称为*undo logging*或*rollback recovery*。回滚恢复的一个特性是，对于已完成的操作，单元存储与日志具有同等的权威性。因此，可以垃圾回收日志，丢弃已完成操作的日志记录。现在小得多的日志可能能够放入更快的存储介质中，其持久性要求仅需超过待处理操作的寿命即可。

有一种替代的、对称性约束被某些日志系统所采用。不同于要求所有安装操作必须记录OUTCOME记录时完成*before*，可以转而要求所有安装操作在记录OUTCOME记录时完成*after*。在这种约束下，日志中属于该全有或全无动作的CHANGE记录集合便成为其意图的描述。如果在记录OUTCOME记录前发生崩溃，我们知道尚未执行任何安装操作，因此恢复过程无需执行任何撤销操作。另一方面，它可能需要为已提交的全有或全无动作执行安装操作。此方案被称为*redo logging*或*roll-forward recovery*。此外，由于我们无法确定哪些安装操作实际已发生，恢复过程必须

对所有未记录END的全有或全无操作执行*all*的日志安装。任何记录了END的全有或全无操作必须已完成其所有安装，因此恢复过程无需再次执行。因此，恢复过程简化为仅对那些在记录OUTCOME和END之间被中断的全有或全无操作进行安装。尽管重做日志恢复需要前后扫描整个日志，但其过程可以相当迅速。

我们可以将原子性日志记录的程序总结如下:

- 在安装到单元存储之前记录到日志存储（WAL协议）
- 如果全有或全无操作在执行*all*安装到非易失性存储之前记录其OUTCOME记录，那么恢复时只需撤销未完成且未提交操作的安装。（回滚/撤销恢复）
- 如果全有或全无操作在记录其OUTCOME记录之前将*no*安装到非易失性存储中，那么恢复时只需重做未完成已提交操作的安装。（前滚/重做恢复）
- 如果全有或全无操作在何时安装到非易失性存储上缺乏纪律性，那么恢复过程不仅需要重做未完成已提交操作的安装*and*，还需撤销未完成未提交操作的安装。

除了读取和更新内存外，一个全有或全无操作可能还需要发送消息，例如向外界报告其成功。发送消息的行为就如同全有或全无操作中的任何其他组成部分一样。为了提供全有或全无的原子性，消息发送可以采用与内存更新类似的方式处理。即，记录一个CHANGE日志条目，其中包含一个用于发送消息的重做操作。如果在全有或全无操作提交后发生崩溃，恢复过程将执行此重做操作以及其他执行安装的重做操作。理论上，也可以记录一个*undo_action*日志条目来发送补偿消息（“请忽略我之前的信息！”）。然而，全有或全无操作通常会谨慎地在操作提交之前不实际发送任何消息，因此适用前滚恢复。基于此原因，设计者通常不会为消息或任何其他具有外界可见性的操作（如打印收据、打开现金抽屉、钻孔或发射导弹）指定撤销操作。

顺便提一下，尽管关于数据库原子性与恢复的大量专业文献采用“胜者”和“败者”这两个术语来描述恢复过程，但不同的恢复系统会依据具体的日志方案对这两组定义存在细微差异，因此仔细审阅这些定义是个明智的做法。

9.3.5 检查点

限制安装顺序必须在记录OUTCOME 之前或之后全部完成，并非我们加速恢复的唯一手段。另一种能缩短日志扫描的技术是偶尔向非易失性存储写入一些额外信息，即所谓的 *checkpoint*。尽管原理始终如一，但具体的

放置在检查点中的信息因系统而异。一个检查点可以包含写入单元存储的信息，或写入日志的信息（此时称为*checkpoint record*），或两者兼有。

例如，假设日志系统在易失性内存中维护一个列表，记录所有已启动但尚未记录END 记录的全有或全无操作的标识符及其挂起/已提交/已中止状态，并通过观察日志调用来保持该列表的最新状态。日志系统随后会偶尔将此列表作为CHECKPOINT 记录进行日志记录。当稍后发生崩溃时，恢复过程会像往常一样开始LIFO（后进先出）日志扫描，收集已完成操作和失败操作的集合。当遇到CHECKPOINT记录时，它可以通过添加那些在检查点中列出但后来未记录END记录的全有或全无操作，立即填充失败操作集合。此列表可能包含一些在CHECKPOINT记录中标记为COMMITTED的全有或全无操作，但在崩溃时尚未记录END记录。这些操作的安装仍需执行，因此需要将它们添加到失败操作集合中。LIFO扫描继续进行，但仅直到找到每个失败操作的BEGIN记录为止。

随着CHECKPOINT记录的加入，恢复过程变得更加复杂，但在时间和精力上可能更短：

1. 对日志进行LIFO（后进先出）扫描，回溯至最近的CHECKPOINT记录，收集失败者的标识并撤销其记录的所有操作。 2. 根据检查点中的信息完善失败者列表。 3. 继续LIFO扫描，撤销失败者的操作，直至找到每个失败者相关的所有BEGIN记录。 4. 从该点开始向前扫描至日志末尾，执行失败者列表中所有全有或全无动作已提交的操作，这些操作需记录有状态为COMMITTED的OUTCOME记录。

在那些长时间运行的“全有或全无”操作并不常见的系统中，第三步通常会非常简短，甚至为空，从而大大缩短恢复时间。一个很好的练习是修改图9.23中的恢复程序，使其能够适应检查点机制。

检查点同样被用于内存数据库，以提供持久性，而无需在每次系统崩溃后重新处理整个日志。对于内存数据库，一种实用的检查点流程是：创建完整数据库的快照，将其写入两个交替（用于全有或全无原子性）专用非易失性存储区域之一，随后记录一个包含最新快照地址的CHECKPOINT记录。恢复时则需扫描日志至最近的CHECKPOINT记录，收集已提交的全有或全无操作列表，还原该记录描述的快照，然后从CHECKPOINT 记录到日志末尾执行这些已提交操作的重做操作。此场景中的主要挑战在于处理与快照写入并发的更新活动。应对这一挑战可通过在快照期间阻止所有更新，或采用更复杂的前后原子性技术（如本章后续章节所述）来实现。

9.3.6 如果缓存不是直写式会怎样？（高级主题）

在日志与直写缓存之间，上述日志配置方案要求每次数据更新时，必须执行两次对非易失性存储的同步写入操作，并伴随等待写入完成的延迟。由于引入日志的初衷是提升性能，这两次同步写入延迟往往成为系统性能瓶颈。追求性能最大化的设计者更倾向于使用非直写式缓存，从而将写入操作推迟至合适时机批量执行。然而，这会导致应用程序丧失对数据实际写入非易失性存储顺序的控制权。写入顺序失控对我们的"全有或全无"原子性算法影响重大——因为这些算法的正确性依赖于对写入顺序的严格约束，以及确认哪些写入已完成的确切信息。

首要关注的是日志本身，因为预写日志协议要求将CHANGE记录追加到日志的操作必须先于对应数据在单元存储中的安装。强制执行WAL协议的一个简单方法是仅使日志写入采用直写模式，而允许单元存储的写入在缓存管理器认为方便时进行。然而，这种宽松处理意味着若系统崩溃，无法保证任何特定安装已实际迁移至非易失性存储。在最坏情况下，恢复过程无法利用检查点，必须从日志起始处重新执行安装操作。为避免这种情况，常规设计对策是在记录每个检查点时刷新缓存作为日志记录的一部分。遗憾的是，为避免与并发更新产生冲突，缓存刷新与检查点记录必须作为原子操作完成，这又带来了新的设计挑战。该挑战虽可克服，但系统复杂度正持续攀升。

有些系统对性能的追求更进一步。一种常见的技术是将日志写入易失性缓冲区，仅当执行全有或全无的提交动作时，才将整个缓冲区*force*到非易失性存储中。这一策略允许将多条CHANGE记录与下一条OUTCOME记录批量处理，通过一次同步写入完成。尽管这一步骤看似违背了预写日志协议，但通过稍微细化用于单元存储的缓存设计即可恢复协议合规性——其管理算法必须避免回写任何日志记录仍驻留于易失性缓冲区的安装操作。关键在于按顺序*number*每条日志记录，并在单元存储缓存中为每条记录标记其对应的日志序列号。每当系统强制刷写日志时，会向缓存管理器通报已写入的最后日志序列号，而缓存管理器则确保绝不回写任何标记着更高日志序列号的缓存记录。

在本节中，我们看到了*law of diminishing returns*发挥作用的一些优秀范例：那些提升性能的方案有时会显著增加复杂性。在实施任何此类方案之前，必须仔细评估能够获得多少额外的性能提升。

9.4 前后原子性 I：概念

本章前几节所开发的机制确保了在故障情况下的原子性，使得故障发生及后续恢复后执行的其他原子操作能够观察到，一个被中断的原子操作要么看似执行了其所有步骤，要么看似未执行任何步骤。本节及下一节将探讨如何进一步实现并发操作的原子性，即*before-or-after atomicity*。在这一进展中，我们将提供*both*全有或全无的原子性*and*前后一致的原子性，因此我们现在能够将最终实现的原子操作称为*transactions*。

并发原子性需要额外的机制，因为当一个原子操作将数据存入单元存储时，该数据会立即对所有并发操作可见。尽管版本历史机制能够向并发原子操作隐藏待定更改，但这些操作仍可能读取第一个原子操作计划更改的其他变量。因此，一个多步骤原子操作的组合性质仍可能被恰好在其执行过程中查看变量值的并发原子操作所察觉。因此，要使一个组合操作相对于并发线程具有原子性——即将其转化为*before-or-after action*——还需要进一步的努力。

回想一下，9.1.5节定义了并发操作的正确性为这些相同操作的*if every result is guaranteed to be one that could have been obtained by some purely serial application*。因此，我们寻求的技术既要保证产生与串行应用并发操作相同的结果，又要通过允许并发最大化可实现的性能。

在第9.4节中，我们将探讨三种逐步优化的前后原子性方案，这里的“优化”指的是方案能支持更高的并发性。为了阐明这些概念，我们重新审视版本历史——它为每种方案提供了直观且有力的正确性论证。由于版本历史在实际中较少应用，接下来的9.5节我们将研究一种略有不同的方法：锁机制。锁因其能提供更高性能而被广泛采用，但其正确性论证则更为复杂。

9.4.1 实现前后原子性：简单序列化

版本历史为每个原子操作分配唯一标识符，以便将变量的暂定版本链接到该操作的结果记录。假设我们要求这些唯一标识符必须是连续的整数（我们将其解释为序列号），并通过添加以下*sim-ple serialization*规则来修改BEGIN_TRANSACTION流程：每个新创建的事务n在读取或写入任何数据之前，必须等待前一个事务$n-1$提交或中止。（为确保始终存在事务$n-1$，假定系统初始化时创建了编号为零的事务，其OUTCOME记录处于已提交状态。）图9.25展示了BEGIN_TRANSACTION的这一版本。该方案强制所有事务按照线程调用BEGIN_TRANSACTION的串行顺序执行。由于这一机制

```
1  procedure BEGIN_TRANSACTION ()
2      id ← NEW_OUTCOME_RECORD (PENDING)              // Create, initialize, assign id.
3      previous_id ← id - 1                           return id
4      wait until previous_id.outcome_record.state ≠ PENDING
```

图9.25

使用简单的串行化规则来实现BEGIN_TRANSACTION的前后原子性。为了确保每个*id*值都对应一个*id* – 1，系统启动时必须创建一个虚拟事务，其中*id* = 0且*id.outcome_record.state*设置为COMMITTED。图9.30展示了过程NEW_OUTCOME_RECORD的伪代码实现。

顺序是各种事务可能的一种串行序列，根据定义，简单串行化将产生被串行化的事务，从而成为正确的前后操作。简单串行化自然而然地提供了前后原子性，且事务仍保持全有或全无的特性，因此无论在故障情况下还是在并发存在时，事务现在都具有原子性。

简单串行化通过过于保守的方式提供前后原子性：它阻止了事务间的所有并发，即使这些事务实际上不会相互干扰。尽管如此，这种方法确实具有一定的实用价值——在某些应用中，基于简洁性的考量，它可能恰恰是最合适的选择。并发线程可以在大部分时间里并行工作，因为简单串行化仅在线程执行事务时发挥作用，而通常这些时刻仅限于它们操作共享变量的时候。如果这类时刻并不频繁，或者需要前后原子性的操作都修改同一小组共享变量，那么简单串行化的效果很可能与其他任何方案不相上下。此外，通过仔细研究其运作原理，我们可以发现更为宽松的方法，这些方法允许更高的并发度，同时仍能有力论证其正确性得以保持。换言之，对前后原子性技术后续研究的本质，无非是发明并分析那些日益高效——也日益复杂——的性能优化措施。

版本历史为这一分析提供了有用的表现形式。图9.26在一张图中展示了由四个账户（分别命名为*A*、*B*、*C*和*D*）组成的银行系统，在执行六个序列号为1至6的交易过程中的版本历史。首个交易将所有对象初始化为0值，随后的交易则在成对账户之间来回转移不同金额。

该图直观地解释了为何简单的串行化能正确运作。以事务3为例，它必须按顺序读取和写入对象*B*与*C*以完成资金转账。若要使事务3产生如同在事务2之后执行的效果，其所有输入对象的值必须包含事务2的全部影响——即当事务2提交时，它所修改的任何对象

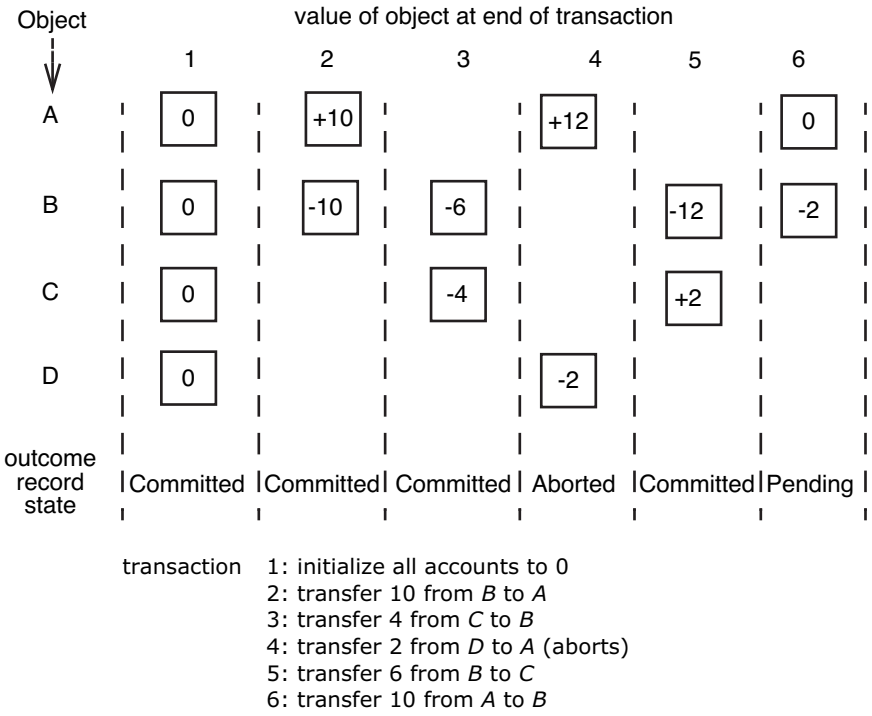


图9.26

银行系统的版本历史。

发生了变化，且3个使用应该具有新的值；如果事务2中止，则它暂态修改的任何对象及3个使用应包含事务2开始时的原有值。在此例中，由于事务3读取了*B*而事务2创建了*B*的新版本，显然事务3要产生正确结果，必须等待事务2提交或中止。简单的序列化要求这一等待，从而确保了正确性。

图9.26也为如何提高并发性提供了一些线索。观察事务4（示例显示事务4最终会因某些原因中止，但假设我们刚开始事务4且尚未知晓该结果），显然简单的串行化过于严格。事务4仅从*A*和*D*读取数值，而事务3对这两个对象均无涉及。因此无论事务3是否提交，*A*和*D*的数值都将保持不变，强制要求事务4等待事务3完成的规则会导致事务4被不必要地延迟。另一方面，事务4确实使用了事务2修改的对象，因此事务4必须等待事务2完成。当然，简单串行化能确保这一点——因为事务4需待事务3完成后才能开始，而事务3又必须等待事务2完成后才能启动。

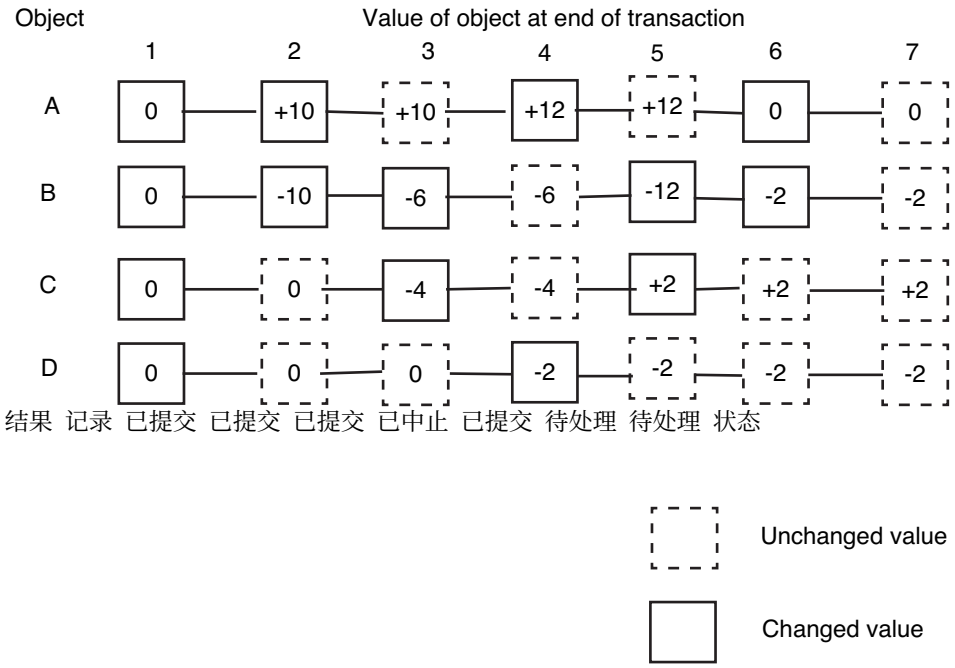


图9.27

系统状态历史，未更改的值已显示。

这些观察表明，可能存在其他更为宽松的规则，仍能确保结果的正确性。它们还暗示，任何此类规则很可能都需要详细检查每个事务具体读取和写入哪些对象。

图9.26以序列化顺序呈现了整个系统的状态历史，而图9.27稍有不同的表现形式使该状态历史更为显化。在图9.27中，看似每笔事务都反常地为每个对象创建了新版本{v*}，其中未实际更改的对象以虚线框标示原值不变。这种表示方式强调，例如事务3的垂直槽位实质上是为系统中每个对象预留的状态历史记录位；事务3有权根据自身需求为任意对象提议新值。

系统状态历史之所以对讨论有帮助，是因为只要最终得到的状态历史中，各框内的值与所示一致，那么各个对象值实际被放入这些框中的实时顺序并不重要。例如，在图9.27中，事务3可以先于事务2创建对象*C*的新版本，而非*B*的新版本。我们并不关心事件发生的具体时机，只要最终填充历史的结果与严格遵循这一串行顺序所得到的值集合相同即可。这使得实际时间序列变得无关紧要——

重要之处正是我们的目标，因为这使我们能够让并发线程处理各种事务。当然，时间顺序上存在约束，但通过检查状态历史，这些约束会变得显而易见。

图9.27让我们能够看清系统状态历史记录这一特定交易序列时必须遵守的时间约束。为了使其交易结果符合序列中的位置要求，每笔交易都应使用其所有输入项的最新版本{v*}作为输入值。若图9.27可用，交易4可沿其输入项*A*和*D*的历史记录回溯至最近的实心方框（分别由交易2和1创建），从而正确判定：只要交易2和1已提交，即使交易3尚未完成对*B*和*C*的赋值且未决定是否提交，交易4仍可继续执行。

这一观察表明，任何事务只要能够发现其左侧版本历史框的虚线或实线状态，就拥有足够信息来确保相对于其他事务的先后原子性。该观察还引出了一个具体的先后原子性规则，该规则将确保正确性。我们称这一规则为*mark-point*。

9.4.2 标记点规则

并发线程调用图9.15中实现的READ_CURRENT_VALUE时，无法看到任何变量的待定版本。这一观察在设计前后原子性规则时非常有用，因为它允许事务通过简单地将其OUTCOME记录的值更改为COMMITTED来一次性展示所有结果。但除此之外，我们还需要一种方法让后续需要读取待定版本的事务能够等待其变为已提交状态。实现这一目标的方法是修改READ_CURRENT_VALUE，使其等待而非跳过由顺序排序中较早事务（即具有较小*caller_id*的事务）创建的待定版本，如图9.28第4-9行所示。由于在并发情况下，排序较后的事务可能在该事务读取同一变量之前创建其新版本，READ_CURRENT_VALUE仍会跳过任何由具有较大*caller_id*的事务创建的版本。此外，与之前一样，提供一个READ_MY_VALUE过程（未展示）可能较为便利，该过程可返回当前事务先前写入的待定值。

在READ_CURRENT_VALUE中添加等待待定版本的能力是第一步；为确保正确的前后原子性，我们还需安排事务所需的所有输入变量——那些早先未提交事务计划修改的变量——都拥有待定版本。为此，我们要求应用程序员（例如编写TRANSFER事务的程序员）额外做一些工作：每个事务应为其计划修改的每个变量创建新的待定版本，并在完成时予以声明。创建待定版本的效果是标记这些变量暂不供后续事务读取，因此我们将事务完成创建所有待定版本的时间点称为该事务的*mark point*。

事务通过调用名为MARK_POINT_ANNOUNCE的过程来宣布其已通过标记点，该过程仅在该事务的结果记录中设置一个标志.

标记点规则即是：在前一事务达到其标记点或不再处于挂起状态之前，任何事务都不能开始读取其输入数据。这一规则要求每个事务必须明确标识它将更新的数据。如果事务在发现其他需要更新的数据对象身份之前必须先修改某些数据对象，它可以选择延迟设置其标记点，直到确实知晓所有待写入对象为止（这当然也会延迟所有后续事务），或者采用下一节所述的更为复杂的规则。

例如，在图9.27中，新到达事务7下方的方框均为虚线标注；事务7应首先标记其计划转为实线的部分。为便于标记操作，我们将图9.15中的WRITE_NEW_VALUE流程拆分为两部分，分别命名为NEW_VERSION和WRITE_VALUE，如图9.29所示。标记过程简化为一系列对NEW_VERSION的调用。完成标记后，事务调用MARK_POINT_ANNOUNCE。随后它便可执行其业务逻辑，根据目标需求进行数值的读写操作。

最后，我们通过在图9.30所示的BEGIN_TRANSACTION处设置一个测试并根据其结果决定是否等待，来强制执行标记点规则。这样，任何事务在前一事务报告其已达到标记点或不再处于PENDING状态之前，都不能开始执行。图9.30还展示了MARK_POINT_ANNOUNCE的一种实现方式。如图9.13所示，过程ABORT和COMMIT无需任何修改，因此此处不再重复。

由于任何事务都必须在前一事务达到其标记点后才能开始，因此序列顺序中较早的所有事务也必然已经通过了它们的标记点，这意味着序列顺序中较早的每个事务都已经创建了它将生成的所有版本。既然READ_CURRENT_VALUE现在等待较早的待定值变为

```
1  procedure READ_CURRENT_VALUE (data_id, this_transaction_id)
2      starting at end of data_id repeat until beginning
3          v ← previous version of data_id
4          last_modifier ← v.action_id
5          if last_modifier ≥ this_transaction_id then skip v        // Keep searching
6          wait until (last_modifier.outcome_record.state ≠ PENDING)
7          if (last_modifier.outcome_record.state = COMMITTED)
8              then return v.state
9              else skip v                                           // Resume search
10     signal ("Tried to read an uninitialized variable")
```

图9.28

READ_CURRENT_VALUE for the mark-point discipline. This form of the procedure skips all versions created by transactions later than the calling transaction, and it waits for a pending version created by an earlier transaction until that earlier transaction commits or aborts.

```
1  procedure NEW_VERSION (reference data_id, this_transaction_id)
2      if this_transaction_id.outcome_record.mark_state = MARKED then
3          signal ("Tried to create new version after announcing mark point!")
4      append new version v to data_id
5      v.value ← NULL
6      v.action_id ← transaction_id

7  procedure WRITE_VALUE (reference data_id, new_value, this_transaction_id)
8      starting at end of data_id repeat until beginning
9          v ← previous version of data_id
10         if v.action_id = this_transaction_id
11             v.value ← new_value
12             return
13     signal ("Tried to write without creating new version!"))
```

图9.29

NEW_VERSION 和 WRITE_VALUE 的标记点规范版本。

```
1  procedure BEGIN_TRANSACTION ()
2      id ← NEW_OUTCOME_RECORD (PENDING)
3      previous_id ← id - 1
4      wait until (previous_id.outcome_record.mark_state = MARKED)
5          or (previous_id.outcome_record.state ≠ PENDING)
6      return id

7  procedure NEW_OUTCOME_RECORD (starting_state)
8      ACQUIRE (outcome_record_lock)        // Make this a before-or-after action.
9      id ← TICKET (outcome_record_sequencer)
10     allocate id.outcome_record
11     id.outcome_record.state ← starting_state
12     id.outcome_record.mark_state ← NULL
13     RELEASE (outcome_record_lock)
14     return id

15 procedure MARK_POINT_ANNOUNCE (reference this_transaction_id)
16     this_transaction_id.outcome_record.mark_state ← MARKED
```

图9.30

The procedures BEGIN_TRANSACTION, NEW_OUTCOME_RECORD, and MARK_POINT_ANNOUNCE for the mark-point discipline. BEGIN_TRANSACTION presumes that there is always a preceding transaction. so the system should be initialized by calling NEW_OUTCOME_RECORD to create an empty initial transaction in the *starting_state* COMMITTED and immediately calling MARK_POINT_ANNOUNCE for the empty transaction.

无论事务是提交还是中止，它总会向客户端返回一个代表所有先前事务最终结果的值。因此，事务的所有输入值都包含了序列顺序中早于它的所有事务已提交的结果，就如同遵循了简单的串行化规则一样。这样一来，无论各事务实际以何种实时顺序将数据值写入其版本槽，结果都保证与串行顺序产生的结果完全相同。正如简单串行化规则的情况一样，由此规则产生的特定串行顺序，就是事务被NEW_OUTCOME_RECORD分配序列号的顺序。

全有或全无原子性与前后原子性之间存在一种潜在的交互作用。如果待定版本在系统崩溃后得以保留，那么在重启时，系统必须追踪所有PENDING事务记录并将其标记为ABORTED，以确保READ_CURRENT_VALUE的未来调用者不会等待那些已永久消失的事务完成。

标记点规则通过从一个更原始的前后原子性机制中引导，提供了前后原子性。如同引导过程中的常见做法，其核心思想是将一个普遍问题——此处即为为任意应用程序提供前后原子性——简化为一个适合特殊解决方案的特殊案例。这里的特殊案例是新建并初始化一个结果记录。图9.30中的过程NEW_OUTCOME_RECORD本身必须是一个前后原子操作，因为它可能被多个不同线程并发调用，且必须确保为每个线程分配不同的序列号。同时，它还需创建完全初始化的结果记录，其中*value*和*mark_state*分别设置为PENDING 和NULL，因为并发线程可能立即需要访问这些字段之一。为实现前后原子性，NEW_OUTCOME_RECORD从第5.6.3节的TICKET过程引导获取下一个顺序序列号，并利用ACQUIRE 和RELEASE使其初始化步骤成为前后原子操作。这些过程又进一步从更低层级的前后原子性机制中引导，因此我们共有三层引导机制。

我们现在可以重新编写图9.15中的资金TRANSFER处理流程，使其在图9.31所示情况下同时具备故障原子性和并发原子性。相较于早期版本，主要改动在于新增了第4至6行代码——TRANSFER通过调用NEW_VERSION标记待修改变量，随后调用MARK_POINT_ANNOUNCE。该程序值得注意的特点是：实现动作"前或后"语义的大部分工作实际上由被调用过程完成。应用程序员仅需通过创建新版本标识并标记事务将修改的变量，无需额外投入精力或思考。

在简单串行化规则下会全部集中在BEGIN_TRANSACTION的延迟，在标记点规则下被分散开来。部分延迟仍可能出现在BEGIN_TRANSACTION，等待前一个事务到达其标记点。但如果标记操作先于其他任何计算完成，事务就能迅速抵达各自的标记点，因此这种延迟应该不会像等待事务提交或中止那样严重。延迟也可能发生在任何调用

```
1  procedure TRANSFER (reference debit_account, reference credit_account,
2                                                                amount)
3      my_id ← BEGIN_TRANSACTION ()
4      NEW_VERSION (debit_account, my_id)
5      NEW_VERSION (credit_account, my_id)
6      MARK_POINT_ANNOUNCE (my_id);
7      xvalue ← READ_CURRENT_VALUE (debit_account, my_id)
8      xvalue ← xvalue - amount
9      WRITE_VALUE (debit_account, xvalue, my_id)
10     yvalue ← READ_CURRENT_VALUE (credit_account, my_id)
11     yvalue ← yvalue + amount
12     WRITE_VALUE (credit_account, yvalue, my_id)
13     if xvalue > 0 then
14         COMMIT (my_id)
15     else
16         ABORT (my_id)
17         signal("Negative transfers are not allowed.")
```

图9.31

一种资金转账程序的实现，采用标记点规则确保其在故障和并发活动方面均具有原子性。

READ_CURRENT_VALUE，但仅当确实存在事务必须等待的内容时才会执行，例如提交必要输入变量的待定版本。因此，任何给定事务的总体延迟不应超过简单串行化规则所施加的限制，并且可以预期，实际延迟往往会更短。

标记点机制的一个有用特性是它永远不会产生死锁。每当出现等待时，等待的对象总是序列化中的某个事务*earlier*。该事务可能又在等待更早的事务，但由于没有人会等待排序中更靠后的事务，因此系统总能保证进展。原因在于，任何时候都必然存在某个最早待处理的事务。排序特性确保这个最早待处理事务不会因等待其他事务完成而受阻，因此它至少能取得进展。当它完成时，排序中的另一个事务就成为最早事务，此时该事务便能推进。通过这种论证，最终每个事务都能取得进展。这种关于系统进展的推理方式，是"前后"原子性机制的有益组成部分。在本章9.5节，我们将探讨一些"前后"原子性机制——它们能保证产生与串行排序相同的结果（在这个意义上是正确的），但无法保证系统进展。这类机制需要额外的方法来确保线程不会陷入永久相互等待的死锁状态。

还有两点次要内容值得注意。首先，如果事务等到准备提交或中止时才宣布其标记点{v*}，那么标记点规则就简化为简单的串行化规则。这一观察证实了某一规

pline是另一种的宽松版本。其次，在标记点（mark-point）规范中，至少存在两次机会来发现并向客户端报告协议错误。事务在宣布其标记点后绝不应调用NEW_VERSION。同样地，如果客户端尝试写入一个从未创建新版本的值，WRITE_VALUE 可以报告错误。这两种错误报告机会均在图9.29的伪代码中实现。

9.4.3 乐观原子性：读取捕获（高级主题）

简单的串行化和标记点规则都是并发控制方法，可被描述为*pessimistic*。这意味着它们假定并发事务间很可能发生干扰，并通过在可能发生干扰的任何点强制等待，主动防止任何干扰的可能性。这样做时，它们也可能阻止了一些对正确性无害的并发操作。另一种方案称为*optimistic*并发控制，它假定并发事务间的干扰不太可能发生，允许它们无需等待地继续执行。然后，监视实际的干扰情况，如果发生干扰，则采取一些恢复措施，例如中止干扰事务并使其重新启动。（有一种流行的戏谑说法形容两者区别：悲观的=“先问再做”，乐观的=“事后道歉”。）乐观并发控制的目标是在实际干扰罕见的情况下提高并发性。

图9.27所示的系统状态历史表明存在乐观处理的可能。我们可以允许事务以任意顺序、任意时间将数值写入系统状态历史，但需承担某些写入尝试可能遭遇回应的风险："抱歉，该写入会干扰其他事务。您必须中止操作，放弃当前在系统状态历史中的序列化位置，获取更新的序列化点，并从头重新运行您的事务。"

这种方法的一个具体例子是*read-capture*规则。在读取捕获规则下，存在提前标记的选项，但并非强制要求。取消提前标记的要求有一个优势，即事务无需预测它将更新的每个对象的身份——它可以在工作时发现这些对象的身份。替代提前标记的是，每当事务调用READ_CURRENT_VALUE, 时，该过程会在该线程读取对象的版本历史中的当前位置做一个标记。这个标记告诉那些在串行顺序中较早但在实际时间中较晚到达的潜在版本插入者，他们不再被允许插入——必须中止并重试，使用版本历史中更靠后的串行位置。如果预期的版本插入者能更早到达，在读取者留下标记之前，新版本本是可以接受的，而读取者则会等待版本插入者提交，并采用那个新值而非较早的值。读取捕获赋予读取者通过介入事务延长版本有效性的能力，直至读取者自身的串行化位置。这一情境的视图如图9.32所示，其版本历史与图9.27相同。

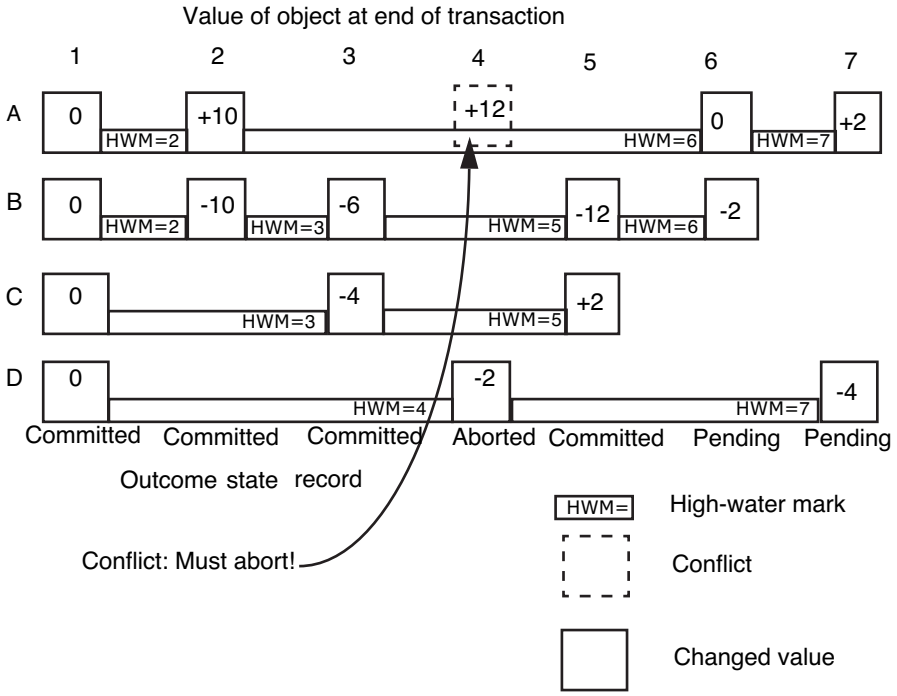


图9.32

带有高水位标记和读取捕获机制的版本历史。首先，与事务4并发运行的事务6读取了变量A，从而将A的高水位标记提升至6。接着，事务4（意图从*D*向*A*转账2）在尝试创建A的新版本时遭遇冲突，发现*A*的高水位标记已被事务6设定，因此事务4中止并以事务7的身份重试。事务7重新执行事务4的操作，将*A*和*D*的高水位标记更新至7。

读取捕获的关键特性通过图9.32中的示例得以阐明。事务4在创建对象*A*的新版本时延迟了；当它尝试执行插入操作时，事务6已经读取了旧值（+10），从而将该旧值的有效期延长至事务6的开始时刻。因此，事务4不得不被中止；它已转生为事务7重新尝试。作为事务7的新身份，其首要操作是读取对象*D*，这将最近提交的值（零）的有效期延伸至事务7的起始点。当它尝试读取对象*A*时，发现最新版本仍未提交，故必须等待事务6提交或中止。值得注意的是，若事务6此时决定创建对象*C*的新版本，可以无障碍执行；但若试图创建对象*D*的新版本，则会与现已延期的旧版本*D*产生冲突，导致事务6必须中止。

```
1   procedure READ_CURRENT_VALUE (reference data_id, value, caller_id)
2       starting at end of data_id repeat until beginning
3           v ← previous version of data_id
4           if v.action_id ≥ caller_id then skip v
5           examine v.action_id.outcome_record
6               if PENDING then
7                   WAIT for v.action_id to COMMIT or ABORT
8                   if COMMITTED then
9                       v.high_water_mark ← max(v.high_water_mark, caller_id)
10                      return v.value
11                  else skip v                          // Continue backward search
12      signal ("Tried to read an uninitialized variable!")

13  procedure NEW_VERSION (reference data_id, caller_id)
14      if (caller_id < data_id.high_water_mark)        // Conflict with later reader.
15      or (caller_id < (LATEST_VERSION[data_id].action_id))     // Blind write conflict.
16      then ABORT this transaction and terminate this thread
17      add new version v at end of data_id
18      v.value ← 0
19      v.action_id ← caller_id

20  procedure WRITE_VALUE (reference data_id, new_value, caller_id)
21      locate version v of data_id.history such that v.action_id = caller_id
22              (if not found, signal ("Tried to write without creating new version!"))
23      v.value ← new_value
```

图9.33

读取捕获形式的READ_CURRENT_VALUE、NEW_VERSION以及WRITE_VALUE。

在版本历史系统中实现读捕获相对容易。如图9.33所示，我们从在READ_CURRENT_VALUE处新增一个步骤（第9行）开始。这个新步骤会为每个数据对象记录一个*high-water mark*——即曾从该对象版本历史中读取过值的最高编号事务的序列号。这个高水位标记相当于对具有更早序列号但延迟创建新版本的其他事务发出警告：序列顺序中更靠后的事务已经读取了该对象在顺序中更早的版本，因此现在创建新版本为时已晚。为确保该警告被遵守，我们在NEW_VERSION处（第14行）增加了一个步骤，检查待写入对象的高水位标记，确认是否有更高序列号的事务已读取该对象的当前版本。如果没有，则可以无忧创建新版本。但如果高水位标记中的事务序列号大于当前事务自身的序列号，则该事务必须中止，获取一个新的更高序列号，并重新开始。

我们移除了对并发事务各组成步骤实时顺序的所有约束，因此可能出现高序号事务先创建某对象的新版本，随后低序号事务又试图创建同一对象新版本的情况。由于我们的NEW_VERSION流程只是简单地将新版本附加到对象历史记录的末尾，最终可能导致历史记录顺序错乱。避免这一错误的最简单方法是在NEW_VERSION (的第15)行加入额外检查，确保每个新版本的客户端序列号都大于前一版本的序列号。若不满足此条件，NEW_VERSION将中止该事务，如同发生读捕获冲突时一样。（此检查仅中止执行冲突性*blind writes*的事务，这类情况并不常见。若冲突事务中任一方在写入前先读取该值，通过*high_water_mark* 的设置与检测即可捕获并阻止冲突。）

对于这类算法，人们首先必须提出的问题是它是否真的有效：其结果是否总是与并发事务的某种串行顺序一致？由于读捕获机制允许比标记点更高的并发度，其正确性论证就略显复杂。论证的归纳部分如下：

1. 对于READ_CURRENT_VALUE中的PENDING值，WAIT确保如果有任何待处理事务 $k < n$修改了随后被事务n, 读取的值，事务n将等待事务k提交或中止。 2. 当事务n调用READ_CURRENT_VALUE时设置高水位标记，以及在NEW_VERSION 中对高水位标记的测试，共同保证了如果有任何事务$j < n$ 试图在事务n读取某值后修改该值，事务j将中止而不会修改该值。 3. 因此，READ_CURRENT_VALUE返回给事务n的每一个值都将包含前驱事务$1 \ldots n-1$的最终效果。 4. 因此，每个事务n的行为都将如同它串行跟随事务$n-1$之后一样。

乐观协调机制（如读捕获）可能会带来一个看似矛盾的效果：在串行顺序中较后发生的事务操作，可能导致顺序中较早的事务中止。这种效应是乐观策略的代价所在；要成为乐观机制的理想候选方案，应用场景中最好不存在大量的数据冲突。

读捕获的一个微妙之处在于，必须在过程NEW_VERSION中实现原子性前后的引导，通过添加锁以及对ACQUIRE和RELEASE的调用来完成，因为NEW_VERSION现在可能被两个并发线程调用，这些线程恰好在几乎同一时间为同一变量添加新版本。此外，NEW_VERSION 必须谨慎保持同一变量的版本按照事务顺序排列，以确保READ_CURRENT_VALUE执行的后向搜索能够正确工作。

还有一个最后的细节，涉及到全有或全无恢复的交互。高水位标记应存储在易失性内存中，以便在发生崩溃（其效果

（中止所有待处理事务时）高水位标记会自动消失，因此不会引发不必要的中止。

9.4.4 有人真的使用版本历史来实现前后原子性吗?

答案是肯定的，但最常见的应用场景不太可能被软件专家遇到。传统处理器架构通常只提供有限数量的寄存器（即“架构寄存器”），供程序员暂存临时结果，而现代大规模集成电路技术允许在物理芯片上实现比架构要求多得多的物理寄存器。更多的寄存器通常能带来更好的性能，尤其是在多发射处理器设计中——这类设计会尽可能并行执行多条顺序指令。为了利用这些额外的物理寄存器，一种名为*register renaming*的寄存器映射方案为架构寄存器实现了版本历史记录。这种版本历史使得那些原本仅因寄存器数量不足而相互干扰的指令能够并发执行。

例如，基于第5.7节描述的x86指令集架构的英特尔奔腾处理器仅有八个架构寄存器。而奔腾4则拥有128个物理寄存器，并采用了一种基于环形*reorder buffer*的寄存器重命名方案。重排序缓冲区类似于对图9.29中NEW_VERSION 和 WRITE_VALUE 过程的直接硬件实现。当每条指令发射时（对应BEGIN_TRANSACTION），它会被分配到重排序缓冲区中的下一个顺序槽位。该槽位是一个映射表，维护着两个编号之间的对应关系：程序员指定用于存放指令输出值的架构寄存器编号，以及实际将存储该输出值的128个物理寄存器之一编号。由于机器指令仅有一个输出值，在重排序缓冲区中分配槽位便以单步操作同时实现了NEW_OUTCOME_RECORD和NEW_VERSION的效果。类似地，当指令提交时，它会将其输出值存入对应的物理寄存器，从而以单步操作完成WRITE_VALUE 和COMMIT 的实现。

图9.34展示了采用重排序缓冲区的寄存器重命名机制。在该示例的程序序列中，指令n使用架构寄存器五来保存输出值，该值将被指令$n + 1$作为输入使用。指令$n + 2$从内存加载架构寄存器五的内容。寄存器重命名技术允许同时存在两个（或多个）版本的寄存器五，其中一个版本（位于物理寄存器42中）包含供指令n和$n + 1$使用的数值，而第二个版本（位于物理寄存器29中）则供指令$n + 2$使用。其性能优势在于，指令$n + 2$（以及后续任何向架构寄存器五写入的指令）能够与指令n和$n + 1$并发执行。在指令$n + 2$之后需要以架构寄存器五的新值作为输入的指令，将通过硬件实现的READ_CURRENT_VALUE来定位重排序缓冲区中该架构寄存器最近一次的前置映射。在此例中，该最近映射指向物理寄存器29。

随后指令会暂停，等待指令$n + 2$将值写入物理寄存器29。而那些重用架构寄存器五用于某些不需要该版本的后续指令则可以并发执行。

尽管寄存器重命名在概念上简单直接，但当指令间存在依赖关系时，防止干扰的机制往往比标记点或读取捕获这两种规则更为复杂，因此上述描述已作了过度简化。欲了解更多细节，读者应参考处理器架构的教科书，例如Hennessy和Patterson所著的*Computer Architecture, a Quantitative Approach*（进一步阅读建议1.1.1）。

Oracle数据库管理系统提供了多种前后原子性方法，其中一种被称为“可序列化”，尽管这一标签可能有些误导性。该方法采用了一种数据库文献中称为*snapshot isolation*的前后原子性方案。其核心思想是，当一个事务开始时，系统会在概念上对所有已提交的值进行一次快照，事务从该快照中读取所有输入值。如果两个并发事务（可能基于同一快照启动）修改了同一变量，先提交的事务将成功；系统会以“序列化错误”为由中止另一事务。此方案实质上创建了一个有限版本的

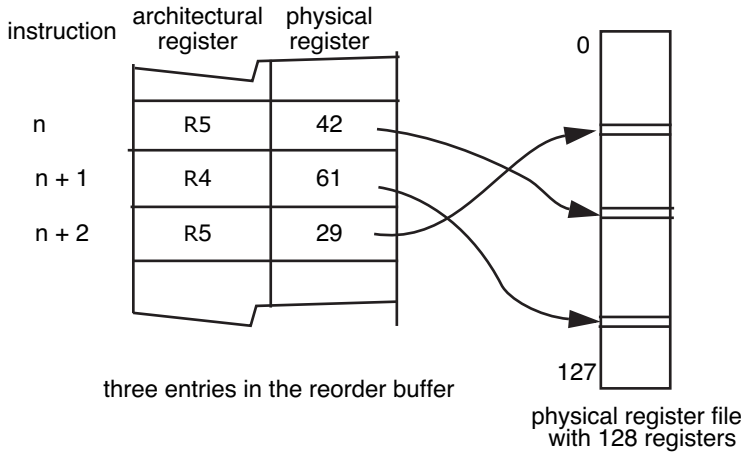


图9.34

示例展示重排序缓冲区如何将架构寄存器编号映射到物理寄存器编号。对应这三个条目的程序序列是：

n R5 ← R4 × R2 // 将结果写入寄存器五。*n* + 1 R4 ← R5 + R1 // 使用寄存器五中的结果。*n* + 2 R5 ← 读取(117492) // 将内存单元的内容写入寄存器五。

指令n和n+ 2都写入寄存器R5，因此R5有两个版本，分别映射到物理寄存器42和29。这样，指令n+ 2就可以与指令n和n+ 1并发执行。

在某些情况下，历史并不总能确保并发事务得到正确协调。

版本历史的另一种专门变体实现，称为*transac- tional memory*，是一种从任意指令序列创建原子操作的规范，这些指令序列多次引用主内存。事务内存的概念最早于1993年被提出，随着多核处理器的广泛普及，因其允许应用程序员使用并发线程而无需处理锁的问题，已成为近期不少研究的关注焦点。该规范要求在原子性指令序列起始处标记“开始事务”指令，将所有后续STORE指令导向并发线程无法读取的数据隐藏副本，并在序列结束时检查确认该序列期间读取或写入的内容未被其他已提交事务修改。若检查未发现此类早期修改，系统则通过向并发线程公开隐藏副本来提交事务；否则丢弃隐藏副本并中止事务。由于它将所有冲突发现推迟至提交点，这一规范比上文第9.4.3节所述的读取捕获规范更为乐观，因此最适用于并发线程间可能但不太可能发生冲突的场景。事务内存已在硬件和软件层面得到实验性实现：硬件实现通常涉及调整缓存或重排序缓冲区，使其延迟将隐藏副本写回主内存直至提交时刻；而软件实现则在主内存其他位置创建变更变量的隐藏副本。与指令重命名类似，此处对事务内存的描述略显简化，感兴趣的读者应查阅文献以获取更完整的解释。

其他软件实现版本历史以实现前后原子性的研究主要在科研环境中进行探索。数据库系统的设计者通常倾向于使用锁而非版本历史，因为在使用锁实现高性能方面积累了更多经验。下一节将系统性地探讨通过锁机制实现前后原子性的主题。

9.5 前后原子性II：实用考量

上一节表明，提供全有或全无原子性的版本历史系统可扩展以支持先写后读原子性。当全有或全无原子性设计采用日志并在单元存储中安装数据更新时，其他并发操作能立即看到这些更新，因此我们仍需一种方案来确保先写后读原子性。若系统利用日志实现全有或全无原子性，通常也会采用第5章—*locks*—介绍的机制来实现先写后读原子性。然而，如第5章所述，使用锁进行编程存在风险，依赖传统调试技术直至结果看似正确的方法难以捕捉所有锁错误。现在我们重新审视锁机制，此次目标是

以程式化的方式使用它们，使我们能够论证这些锁正确地实现了前后原子性。

9.5.1 锁

回顾一下，*lock*是与数据对象关联的一个标志，由某个操作设置，用于警告其他并发操作不要读取或写入该对象。传统上，锁定方案涉及两个过程：

获取（{v*}）

标记与对象*A*关联的锁变量为已获取状态。若该对象已被获取，ACQUIRE将等待直至前一个获取者释放它。

发布 (*A.lock*)

解除与*A*关联的锁变量标记，可能结束其他某个动作对该锁的等待。目前，我们假设锁的语义遵循第5章的单次获取协议：若两个或更多动作几乎同时尝试获取锁，只有一个会成功；其他动作将发现锁已被占用。在9.5.4节中，我们将探讨一些替代协议，例如允许变量在无人写入时被多个读取者同时访问的协议。

锁的最大问题在于，编程错误可能导致操作无法实现预期的前后一致性属性。这类错误会为竞态条件敞开大门——由于干扰操作具有时序依赖性，使得查明问题根源变得极其困难。因此，一个核心目标是确保并发事务的协调机制必须无可争议地正确。就锁而言，系统性地遵循三个步骤即可达成这一目标：

- 制定一个锁定规则，明确规定必须获取哪些锁以及何时获取。
- 建立一个令人信服的推理线索，表明遵循该规则的并发事务将具有{v*}之前或之后{v*}的属性。
- 在程序员与ACQUIRE和RELEASE过程之间，插入一个强制执行该规范的*lock manager*程序。

许多锁定机制已被设计和部署，其中一些未能正确协调事务（例如，参见练习9.5）。我们研究了三种成功的机制。每一种都比前一种允许更高的并发性，但即便是最优的方案也无法确保并发性达到最大化。

第一种也是最简单的、能正确协调事务的机制是*system-wide lock*。当系统首次启动运行时，它会在易失性存储器中创建一个可锁变量，例如命名为*System*。该机制的规则是：每笔事务都必须以

```
开始事务 获取(
System.lock) …
```

且每笔交易必须以

```
… 释放 (System.lock)
结束事务
```

一个系统甚至可以通过在**begin_transaction** 和 **end_transaction**生成的代码序列中包含ACQUIRE和RELEASE步骤来强制执行这一规则，无论结果是COMMIT还是ABORT。这样，任何创建新事务的程序员都能确保该事务将在其他任何事务之前或之后运行。

系统范围的锁机制只允许一个事务在同一时间执行。它按照事务调用ACQUIRE的顺序，将潜在的并发事务串行化。系统范围的锁机制在所有方面都与第9.4节中简单的串行化机制相同。实际上，简单的串行化伪代码

```
id ← 新结果记录 () preceding_id ← id - 1 等待直到
preceding_id.outcome_record.value ≠ 待处理

…… 提交（id） [或 中止（id
） ]
```

以及系统范围的锁调用

```
获取 (System.lock)
… 发布 (System.lock)
```

实际上只是同一想法的两种实现方式。

与简单串行化一样，系统级锁定在无需限制并发的情况下也施加了约束，因为它会锁定每笔事务涉及的所有数据。例如，若将系统级锁定应用于图9.16中的资金TRANSFER程序，同一时间只能执行一次转账操作——即便单次转账仅涉及数百万账户中的两个账户，这显然错失了大量可并行执行且互不干扰的转账机会。因此，开发限制更少的锁定机制成为研究重点。通常的切入点是采用更细粒度的锁*granularity*：锁定更小的对象单元，如单条数据记录、数据记录的单个页面，乃至记录内的字段。提升并发性需要权衡的是：首先，多锁机制会增加获取与释放锁的时间开销；其次，正确性论证会变得更加复杂。人们期望并发带来的性能提升能超越多锁操作的成本。幸运的是，至少还存在另外两种可实现正确性论证的机制*simple locking*与*two-phase locking*。

9.5.2 简单锁定

第二种锁定规则，称为*simple locking,*，在精神上与标记点规则相似，但并不完全相同。简单锁定规则包含两条原则：首先，每个事务在真正进行读写操作前，必须为所有打算读取或写入的共享数据对象获取锁；其次，只有在事务安装完最后一次更新并提交，或完全恢复数据并中止后，才能释放其持有的锁。与标记点类似，事务存在一个被称为*lock point*的关键时刻：即事务获取全部所需锁的第一个瞬间。当事务达到其锁定点时已获得的锁集合称为其 *lock set*。锁管理器可通过要求每个事务将其预期锁集作为参数传递给**begin_transaction**操作（该操作会获取锁集中的所有锁，必要时等待锁可用）来强制执行简单锁定。锁管理器还能拦截所有数据读取和变更日志调用，以验证这些操作是否针对锁集内的变量。此外，锁管理器会截获提交或中止调用（若应用程序采用前滚恢复，则截获记录END的日志调用），此时它将自动释放锁集中的所有锁。

简单的锁定规则正确地协调了并发事务。我们可以采用类似于证明标记点规则正确性的论证思路来支持这一主张。设想一位全知的外部观察者维护着一个有序列表，每当一个事务到达其锁定点时，观察者就将该事务标识符加入列表；当事务开始释放锁时，则将其从列表中移除。在简单锁定规则下，每个事务都承诺在自身被加入观察者列表之前不会读取或写入任何数据。我们还知道，列表中排在该事务之前的所有事务必定已经通过了它们的锁定点。由于任何数据对象都不可能同时出现在两个事务的锁定集合中，因此任一事务锁定集合中的数据对象都不会出现在列表中前驱事务的锁定集合中，通过归纳推理可知也不会出现在更早事务的锁定集合中。因此，该事务的所有输入值都与列表中前驱事务提交或中止时的状态保持一致。同样的论证适用于前驱事务的前序事务，因此任何事务的输入都与假设列表中所有前序事务按列表顺序串行执行时所能提供的输入完全相同。由此可见，简单锁定规则确保了当前事务完全在前驱事务之后执行，并完全在后续事务之前执行。并发事务产生的结果，将如同它们按照到达锁定点的顺序被串行化执行一样。

与标记点规则一样，简单的锁定机制可能会错过一些并发机会。此外，这种简单的锁定规则还会带来一个问题，在某些应用中可能相当显著。因为它要求事务对每一个将要读取*or*或写入的共享对象获取锁（回想一下，标记点规则仅要求对事务将要写入的共享对象进行标记），对于那些需要通过读取其他共享数据对象来发现需要读取哪些对象的应用来说，除了锁定所有*might*需要读取的对象外别无选择。就应用*might*需要读取的对象集合最终大于其实际操作的集合而言

*does*读取时，简单的锁定机制可能会干扰并发机会。另一方面，当事务较为直接（例如图9.16中的TRANSFER事务，它只需锁定两条记录，且这两条记录在开始时即已知晓）时，简单的锁定机制可以很有效。

9.5.3 两阶段锁定

第三种锁定规则，与读捕获规则一样被称为*two-phase locking,*，避免了事务必须预先知道要获取哪些锁的要求。两阶段锁定被广泛使用，但更难论证其正确性。两阶段锁定规则允许事务在执行过程中获取锁，一旦事务获取了某个数据对象的锁，就可以立即读取或写入该对象。主要约束是，在事务通过其锁定点之前，不得释放任何锁。此外，对于仅读取的对象，事务在达到锁定点*if*后可以随时释放其锁——即使是为了中止，也无需再次读取该对象。 该规则的名称源于事务获取的锁数量在达到锁定点之前单调递增（第一阶段），之后则单调递减（第二阶段）。与简单锁定一样，两阶段锁定通过按事务达到锁定点的顺序对并发事务进行排序，使它们产生的结果如同被串行化执行。锁管理器可以通过拦截所有读写数据的调用来实现两阶段锁定；它在首次使用每个共享变量时获取锁（可能需要等待）。与简单锁定类似，锁管理器会一直持有这些锁，直到拦截到提交、中止或记录事务END记录的调用，此时它会一次性释放所有锁。

两阶段锁的额外灵活性使得更难论证它能确保前后原子性。非正式地说，一旦事务获取了数据对象上的锁，该对象的值将与事务到达其锁定点时相同，因此现在读取该值必然与等到那时再读取得到相同结果。此外，如果事务之后不再查看该对象（即使是回滚时），释放未修改对象上的锁必定无害。关于两阶段锁能实现正确前后原子性的形式化论证，可在多数关于并发控制与事务处理的高级文献中找到，例如Gray和Reuter所著的*Trans-action Processing*（参见[进一步阅读建议1.1.5]）。

两阶段锁定协议相比简单锁定协议可能允许更高的并发度，但它仍然会不必要地阻塞某些可串行化（因而正确）的操作序列。例如，假设事务T1读取*x*并写入*y*，而事务T2仅对*y*执行（盲目）写入。由于T1和T2的锁集在变量*y*处相交，两阶段锁定协议将强制事务T2要么完全在T1之前运行，要么完全在T1之后运行。但序列

T1: 读取 *X*
T2: 写入 *Y* T
1: 写入 *Y*

其中，T2的写入操作发生在T1的两个步骤之间，产生的结果与在T1之前完全运行T2相同，因此结果始终是正确的，尽管这种顺序会被两阶段锁定所阻止。要设计出既能允许所有可能的并发，又能确保前后原子性的规则是相当困难的。（理论家们将这一问题识别为NP完全问题。）

锁与日志之间存在两种需要仔细考虑的交互情况：(1) 单个事务的中止，(2) 系统恢复。其中中止事务最容易处理。由于我们要求中止事务在释放任何锁之前，必须将其修改的数据对象恢复为原始值，因此无需对中止事务进行特殊处理。就"全有或全无"原子性而言，它们看起来就像未更改任何内容的已提交事务。关于在事务结束前不释放任何已修改数据上的锁这一规则，是实现中止操作的关键。如果某个已修改对象上的锁被释放，而后事务决定中止，它可能会发现其他事务已获取该锁并再次更改了对象。除非持有已修改对象上的锁，否则回滚中止的更改很可能是无法实现的。

基于日志的恢复与锁之间的交互关系不那么显而易见。问题在于锁本身是否应被视为需要记录变更的数据对象。为分析这一问题，假设系统发生崩溃。在崩溃恢复完成后，不应存在任何挂起事务——因为在崩溃时处于挂起状态的所有事务都应由恢复程序回滚，且恢复过程完成前不允许启动新事务。由于锁的存在仅用于协调挂起事务，若崩溃恢复完成后仍有锁被持有，显然属于错误状态。这一观察表明，锁应存放在易失性存储中（崩溃时自动消失），而非非易失性存储（恢复程序需主动追踪并释放它们）。但更核心的问题是：基于日志的恢复算法能否构建出正确的系统状态——这里的"正确"指该状态可能由崩溃前已提交事务的某种串行排序所产生。

继续假设锁位于易失性内存中，在崩溃瞬间所有锁记录都会丢失。某些事务集合——那些已记录BEGIN但尚未记录END的事务——可能尚未完成。但我们知道，在崩溃瞬间未完成的事务，在锁值消失时其锁集合是不重叠的。图9.23的恢复算法会系统性地为未完成事务执行UNDO或REDO安装，但每个这样的UNDO或REDO操作必须修改一个在崩溃时属于某事务锁集合的变量。由于这些锁集合必然互不重叠，这些特定操作可以安全地重做或撤销，无需担心恢复期间的先后原子性。换言之，锁确立了事务的一种特定串行顺序，而日志已捕获该顺序。由于RECOVER按日志指定的逆序执行UNDO操作，并按日志指定的正序执行REDO操作，RECOVER便精确重建了相同的串行顺序。因此，即便是重构该顺序的恢复算法...

从日志中恢复整个数据库可以确保产生与事务最初执行时相同的序列化结果。只要在恢复完成前不开始新的事务，尽管恢复期间没有锁机制，也不会出现协调错误。

9.5.4 性能优化

大多数日志锁定系统的实际复杂度远超目前描述所可能引发的预期。这种复杂性主要源于对性能提升的追求。在9.3.6节中，我们已看到如何通过易失性内存缓存对磁盘I/O进行缓冲——以允许读取、写入和计算并发进行——这会使日志系统变得复杂。设计者有时会为锁定系统引入两种提升性能的复杂机制：物理锁定和增加锁兼容模式。

一种通过磁盘I/O缓冲和物理介质考量驱动的性能优化技术，是选择一种特定的锁粒度，称为*physical locking*。如果事务对一个位于1000字节磁盘扇区中间的六字节对象进行修改，或对一个占据两个磁盘扇区部分空间的1500字节对象进行修改，就会产生一个问题：应该锁定哪个“变量”——是对象本身，还是磁盘扇区？若两个并发线程对恰好存储在同一磁盘扇区中的无关数据对象进行更新，那么这两个磁盘写入操作必须协调进行。选择合适的锁粒度能显著影响性能表现。

不考虑应用定义的对象与物理磁盘扇区的映射关系而直接锁定这些对象，对应用程序编写者来说易于理解，因此颇具吸引力。正因如此，这种方式通常被称为*logical locking.* 。此外，若对象体积较小，它显然能支持更高的并发度：若另一事务对同一磁盘扇区内的不同对象感兴趣，便可并行执行。然而，逻辑锁定的一个必然结果是，日志记录也必须以相同的逻辑对象为单位进行。同一磁盘扇区的不同部分可能被并发运行的不同事务修改，若一个事务提交而另一个中止，则崩溃后既不能恢复为旧扇区状态也不能采用新扇区状态；日志条目必须记录扇区内各数据对象的旧值和新值。最后需注意，采用缓存的高性能日志系统在提交时，必须强制将日志写入磁盘，并追踪缓存中哪些对象可安全写入磁盘而不违反预写日志协议。因此，对小对象实施逻辑锁定可能加剧缓存记录维护的复杂度。

从细节上退一步来看，高性能磁盘管理系统通常要求PUT调用的参数是一个与磁盘扇区大小相匹配的块。因此，逻辑锁定的真正作用是在应用程序与磁盘管理系统之间创建一层，向其事务客户端提供一个逻辑接口而非物理接口；诸如磁盘扇区内的数据对象管理和垃圾回收等功能将被纳入这一层。另一种选择是调整日志记录和锁定设计，以匹配磁盘管理系统的原生粒度。由于将日志记录和锁定的粒度与磁盘写入粒度对齐

可以减少磁盘操作次数，将更改记录和锁定块对应于磁盘扇区而非单个数据对象是一种常见做法。

大多数锁定系统中还存在另一种性能优化：即*lock compatibility modes*的设定。其核心思想在于，当一个事务获取锁时，可以声明它打算对被锁数据项执行何种操作（例如READ或WRITE）。若该操作是兼容的——即并发事务的结果等同于这些事务的某种串行顺序执行结果——那么即使其他事务已对同一数据对象持有锁，当前事务仍可获准加锁。

最常见的例子是用*multiple-reader, single-writer protocol*替代单获取锁定协议。根据该协议，允许多个读取者同时为同一对象获取读模式锁。读模式锁的作用是确保在持有锁期间，其他线程无法更改数据。由于并发读取者不会构成更新威胁，因此允许任意数量的读取者是安全的。如果另一个事务需要为已被多个线程持有读模式锁的对象获取写模式锁，该新事务必须等待所有读取者释放其读模式锁。在许多应用中，大部分数据访问都是读取操作，对于这些应用，提供读模式锁兼容性可以将等待锁的时间减少数个数量级。同时，该方案增加了复杂性，既包括锁定机制本身，也涉及策略问题，例如，当一个潜在的写入者正在等待读取者释放读模式锁时，另一个线程请求获取读模式锁该如何处理。如果不断有新的读取者到来，写入者可能会被无限期延迟。

这段关于性能优化及其复杂性的描述仅是示例性的，旨在说明它们所带来的各种机会和复杂类型；还有许多其他性能提升技术，其中一些可能有效，而另一些则价值存疑；大多数技术的价值因应用场景而异。例如，某些锁定机制会通过允许事务读取尚未提交的数据值，从而在"前或后"原子性上做出妥协。正如人们所料，在这种情况下，关于哪些可能出错、哪些不会出错的推理复杂性会急剧上升。如果设计者打算利用诸如缓冲、锁兼容模式或妥协的"前或后"原子性等性能增强技术来实现系统，建议仔细研读Gray和Reuter的著作，以及已实现类似增强的现有系统。

9.5.5 死锁；确保进展

第五章的5.2.5节介绍了*deadlock*这一突发现象，等待图作为分析死锁的一种方法，以及锁排序作为预防死锁的手段。随着事务的出现，以及能够撤销单个操作甚至完全中止事务的能力，我们现在拥有了更多应对死锁的工具，因此值得重新审视这一讨论。

使用锁来协调并发活动不可避免地会导致死锁的可能性。任意数量的并发事务都可能陷入死锁状态，要么相互等待，要么只是等待某个已经陷入死锁的事务释放锁。死锁给我们留下了一个重大的未解问题：正确性论证确保我们，任何完成的事务都会产生如同它们串行运行一样的结果，但它们并未说明任何事务是否最终会完成。换句话说，我们的系统可以确保*correctness*，即永远不会产生错误的答案，但它不能确保*progress*——可能根本不会有任何答案输出。

与并发控制方法类似，处理死锁的方法也可分为悲观或乐观两类。悲观方法采取*a priori*措施预先阻止死锁发生。乐观方法则允许多个线程并发执行，一旦检测到死锁便采取行动进行修复。以下列举了几种最常用的方法：

1. *Lock ordering*(悲观)。如第5章所述，为锁分配唯一编号，并要求事务按数字升序获取锁。采用此方案时，当一个事务遇到已被占用的锁时，等待总是安全的，因为先前获取该锁的事务不可能正在等待当前事务已持有的任何锁——那些锁的编号都比当前锁小。因此可以保证至少有一个事务（持有最大编号锁的事务）总能取得进展。当该事务完成时，它将释放所有锁，此时另一个事务将成为被确保能取得进展的事务。锁排序的一种广义化改进是将其组织为格结构，并要求按某种格遍历顺序获取锁，这可能会消除不必要的等待。但锁排序与简单锁机制存在相同问题：某些应用可能无法在获取第一个锁之前预知所需的所有锁。

2. *Backing out*（乐观策略）： Andre Bensoussan于1966年设计的一种优雅策略允许事务以任意顺序获取锁，但如果遇到一个已被获取且编号低于自身已获取锁的锁时，该事务必须回退（根据本章术语，即UNDO之前的操作）至足以释放其高编号锁的程度，等待低编号锁变为可用，获取该锁后，再REDO重新执行被回退的操作。

3. *Timer expiration*（乐观）。当新事务开始时，锁管理器会设置一个中断计时器，其值略大于该事务预计完成所需的时间。如果事务陷入死锁，其计时器将到期，此时系统会中止该事务，回滚其更改并释放其持有的锁，以期参与死锁的其他事务能够继续执行。若仍无法解决，另一个事务将超时，进而释放更多锁。虽然死锁超时机制行之有效，但它也存在常见缺陷：

很难选择一个合适的计时器值，既能保持进程顺利推进，又能适应正常的延迟和操作时间的变化。如果环境或系统负载发生变化，可能需要重新调整所有这些计时器的值，这在大型系统中可能是一项相当繁琐的工作。

4. *Cycle detection*（乐观策略）。在锁管理器中维护一个等待图（如5.2.5节所述），该图显示哪些事务已获取哪些锁，以及哪些事务正在等待哪些锁。每当另一个事务尝试获取锁时，发现锁已被占用并提议等待，锁管理器会检查该图以判断等待是否会导致循环，从而引发死锁。如果会，锁管理器会选择循环中的某个成员作为牺牲品，单方面中止该事务，以便其他事务可以继续执行。被中止的事务随后会重试，希望其他事务已取得足够进展而不再阻塞，从而避免再次发生死锁。

当系统采用锁排序、回退或循环检测机制时，通常还会设置定时器作为安全网。这是因为硬件故障或编程错误（如无限循环）可能导致进度阻塞的情况，而这些情况是任何死锁检测方法都无法捕捉到的。

由于死锁检测算法可能引入额外的事务中止原因，可以设想一种极端情况：无论调用者重试多少次，该算法都会中止执行某特定事务的每一次尝试。例如，假设名为Alphonse和Gaston的两个线程在尝试获取名为Apple和Banana的两个对象的锁时陷入死锁：Alphonse先获取Apple的锁，Gaston获取Banana的锁；接着Alphonse尝试获取Banana的锁并进入等待，而Gaston尝试获取Apple的锁也陷入等待，从而形成死锁。最终，Alphonse因超时开始回滚更新以准备释放锁，与此同时Gaston也因超时执行相同操作。两者重启后再次陷入死锁，且计时器设置与之前完全相同，很可能无限重复这一过程。因此，我们仍无法确保系统能取得进展。这正是第5章所称的*live-lock*涌现特性——从形式上看从未发生实际死锁，两个线程表面上都在执行看似有用的操作。

处理活锁的一种方法是应用第七章[在线]中熟悉技术的随机化版本：*exponential random backoff*。当计时器到期导致事务中止时，锁管理器在清除锁后，会随机延迟该线程一段时间，这个时间从某个初始区间中选取，目的是通过随机性改变活锁事务的相对时序，使得下一次尝试时其中一个事务能够成功，另一个事务随后也能无干扰地继续执行。如果事务再次遭遇干扰，它会重试，但每次重试时，锁管理器不仅会选取一个新的随机延迟，还会以某个乘法常数（通常为2）扩大延迟选取的区间。由于每次重试成功的概率都会增加，通过持续重试，可以将成功概率推至任意接近1的水平，预期干扰事务最终会...

最终会彼此让路。指数随机退避的一个有用特性是，如果重复重试持续失败，几乎可以肯定这表明存在更深层次的问题——可能是编程错误，或者对共享变量的竞争程度本质上过高，以至于系统需要重新设计。

设计更为精细的算法或编程规范以确保进展，这一项目的潜在回报相当有限，且*end-to-end argument* 表明其可能不值得投入精力。实际上，那些交易间频繁发生干扰的系统通常本就不会设计成高度并发的模式。当干扰并不频繁时，诸如安全网计时器和指数随机退避等简单技术不仅效果良好，而且通常必须予以配备，以应对可能潜入系统设计或实现中的任何竞态条件或编程错误（如无限循环）。因此，一套更为复杂的进展保障机制很可能是冗余的，且鲜有机会真正推动进展。

9.6 跨层与多站点的原子性

在我们对原子性的探索中，仍存在一些重要的空白。首先，在分层系统中，某一层实现的事务可能由一系列较低层的组件动作构成，这些动作本身也是原子的。问题在于，较低层事务的提交应如何与较高层事务的提交相关联。如果较高层事务决定中止，那么对于那些可能已经提交的较低层事务该如何处理。这里有两种可能性：

- 通过一个UNDO 操作来逆转任何已提交的下层事务的影响。这一技术要求下层事务的结果仅在高层事务内部可见。
- 设法延迟下层事务的提交，并安排它们实际上在上层事务提交的同时完成提交。

至此，我们一直假设第一种可能性。本节我们将探讨第二种情况。

另一个差距在于，如前所述，我们目前提供的原子性技术都涉及使用内存或存储中的共享变量（例如指向最新版本的指针、结果记录、日志和锁），因此隐含地假设构成事务的复合操作都在物理上紧密相邻的位置发生。当这些组合操作在物理上分散时，通信延迟、通信可靠性以及独立故障不仅使原子性变得更加重要，也使其更难以实现。

我们将从这两个问题入手，首先识别一个共同的子问题：实现嵌套事务。接着，我们会将嵌套事务问题的解决方案扩展，以创建一种被称为*two-phase commit,*的共识协议，该协议

```
procedure PAY_INTEREST (reference account)
    if account.balance > 0 then
        interest = account.balance * 0.05
        TRANSFER (bank, account, interest)
    else
        interest = account.balance * 0.15
        TRANSFER (account, bank, interest)

procedure MONTH_END_INTEREST:()
    for A ← each customer_account do
        PAY_INTEREST (A)
```

图9.35

一个关于两个过程的例子，其中一个调用了另一个，但每个过程本身都应是原子性的。

协调下层事务的提交。我们可以扩展两阶段提交协议，利用一种特殊形式的远程过程调用，来协调必须在不同地点执行的步骤。这一序列是引导过程的又一例证；我们已知如何处理的特例是单点事务，而更普遍的问题则是多点事务。进一步观察可以发现，多点事务与{v*}非常相似，但并不完全相同。

9.6.1 交易的分层组合

我们之所以讨论事务，是因为考虑到复杂解释器是以分层方式设计的，每一层都应为其更高层的客户层实现原子操作。因此，事务是嵌套的，每个事务通常由多个下层事务构成。这种嵌套要求我们对实现原子性的机制给予额外的思考。

再次考虑一个银行示例。假设第9.1.5节中的TRANSFER程序可用于将资金从一个账户转移到另一个账户，并且已将其实现为事务。现在假设我们希望创建图9.35中的两个应用程序过程。第一个过程PAY_INTEREST调用TRANSFER，将适当金额的资金转入或转出一个名为*bank*的内部账户，方向和利率取决于客户账户余额是正还是负。第二个过程MONTH_END_INTEREST通过遍历所有账户并对每个账户调用PAY_INTEREST，实现银行每月对每个客户账户支付（或收取）利息的意图。

同时运行两个MONTH_END_INTEREST 的调用可能不太合适，但在MONTH_END_INTEREST 运行的同时，很可能还有其他银行业务活动正在进行，这些活动也在调用TRANSFER。

也可能出现**for each** 语句在MONTH_END_INTEREST内部实际运行多次迭代实例（从而并发执行PAY_INTEREST)）的情况。因此，我们需要三层事务结构。最底层是TRANSFER过程，其中从一个账户扣款与向另一个账户存款必须作为原子操作执行。在更高一层，PAY_INTEREST过程应原子化执行，以确保某些并发的TRANSFER事务不会在正/负测试与利息金额计算之间更改账户余额。最终，MONTH_END_INTEREST过程应作为事务处理，以保证某些并发的TRANSFER事务不会在这两个账户的利息支付处理期间将资金从账户A转移至账户B，因为此类转账可能导致银行对同一笔资金重复支付利息。从结构上看，TRANS-FER过程的调用嵌套于PAY_INTEREST内部，而一个或多个PAY_INTEREST的并发调用则嵌套在MONTH_END_INTEREST之中。

嵌套之所以可能成为问题，源于对嵌套事务提交步骤的考量。例如，TRANSFER事务的提交点似乎必须发生在PAY_INTEREST事务提交点之前或之后，具体取决于在PAY_INTEREST的程序设计中将其提交点置于何处。然而，这两种位置都会引发问题。若TRANSFER 提交发生在PAY_INTEREST的预提交阶段，一旦系统崩溃，PAY_INTEREST将无法回滚至未尝试操作的状态，因为TRANSFER修改的两个账户值可能已被并发事务用于支付决策。但若TRANSFER提交延迟至PAY_INTEREST的后提交阶段，则存在转账本身无法完成的风险，例如因某个账户无法访问。由此得出的结论是：嵌套事务的提交点必须以某种方式与外围事务的提交点重合。 对于MONTH_END_INTEREST，则存在一个稍有不同的协调问题：在其运行期间，其他事务不得进行任何TRANSFER操作（即它应在所有并发TRANSFER事务之前或之后运行），但它自身必须能执行多次TRANSFER操作——每次调用PAY_INTEREST时皆可执行，且其自身可能并发的转账动作必须为"前或后"动作，因为它们都涉及名为“bank”的账户。

暂且假设该系统为事务提供了版本历史。我们可以通过扩展结果记录的概念来处理嵌套问题：允许结果记录按层次结构组织。每当创建一个嵌套事务时，我们会在其结果记录中同时记录新事务的初始状态（PENDING）以及外层事务的标识符。这样形成的结果记录层次结构便能精确反映事务的嵌套关系。顶层结果记录会包含一个标志，表明其不嵌套于任何其他事务内。若某结果记录包含更高层事务的标识符，我们称其为*dependent*结果记录，而其所引用的记录则称为其*superior*。

事务，无论是嵌套的还是外层的，都会照常进行它们的操作，并根据成功与否标记各自的结果记录COMMITTED或ABORTED。然而，当READ_CURRENT_VALUE（如第9.4.2节所述）检查状态时——

要查看某个版本的交易状态是否为COMMITTED，必须额外检查其成果记录是否引用了上级成果记录。若存在引用，则需追溯该引用并检查上级记录的状态。若该记录同样显示为COMMITTED，则需继续沿链向上追溯，必要时直至最高层成果记录。只有当链中所有记录均处于COMMITTED状态时，该交易才真正为COMMITTED。若链中任一记录为ABORTED，则无论其自身成果记录如何声明COMMITTED，该交易实际为ABORTED。若上述情况均不适用，则链中必存在一个或多个仍为PENDING的记录。此时该交易的成果将保持PENDING状态，直至这些记录转为COMMITTED或ABORTED。因此，表面看似COMMITTED的依赖型成果记录，其实际结果取决于所有祖先记录的结果。我们可以这样描述：在所有祖先提交前，该低层交易如同立于刀锋之上——处于即将提交的状态，但仍可能在必要时中止。为便于讨论，我们将此情形定义为成果记录和交易的一个独特虚拟状态，称之为 *tentatively committed*。

这种层次化安排带来了几个有趣的编程影响。如果嵌套事务包含任何提交后步骤，这些步骤必须等到所有更高层次的事务都提交后才能执行。例如，若某个嵌套事务在提交时触发了钱箱开启操作，那么向钱箱发送解锁指令的动作必须被暂缓，直至最外层事务确定其最终结果。

这一输出可见性后果仅是众多与暂态提交状态相关的例子之一。嵌套事务在声明自身为暂态提交后，便放弃了中止的能力——决定权已掌握在他人手中。它必须能够运行至完成*or*才能中止，同时必须能够无限期地维持暂态提交状态。保持这种进退两可的状态可能颇为棘手，因为事务可能正持有锁、将页面保留在内存中或磁带挂载状态，或是可靠地持有输出消息。一个必然结果是，设计者不能简单地选取任意事务，盲目地将其用作更大事务的嵌套组件。至少，设计者必须审查嵌套事务维持暂态提交状态所需的条件。

另一个更为复杂的后果出现在考虑同一高层事务内嵌套的不同事务之间可能的交互时。以我们之前的例子为例，TRANSFER事务嵌套在PAY_INTEREST内，而PAY_INTEREST又嵌套在MONTH_END_INTEREST中。假设MONTH_END_INTEREST首次调用PAY_INTEREST时，该调用成功提交，从而进入暂定提交状态，等待MONTH_END_INTEREST的结果。接着，MONTH_END_INTEREST在第二个银行账户上调用PAY_INTEREST。PAY_INTEREST需要能够读取银行自身利息账户的值作为输入数据，这是之前PAY_INTEREST的暂定COMMITTED,调用的待定结果。如第9.4.2节所实现的READ_CURRENT_VALUE算法，并未区分来自同一组嵌套事务内部的读取与来自其他事务的读取。

完全不相关的交易。图9.36展示了这种情况。如果READ_CURRENT_VALUE 中对已提交值的测试通过简单地跟随控制最新版本的结果记录的祖先链来扩展，无疑会迫使PAY_INTEREST的第二次调用等待PAY_INTEREST第一次调用的最终结果。但由于第一次调用的结果又依赖于

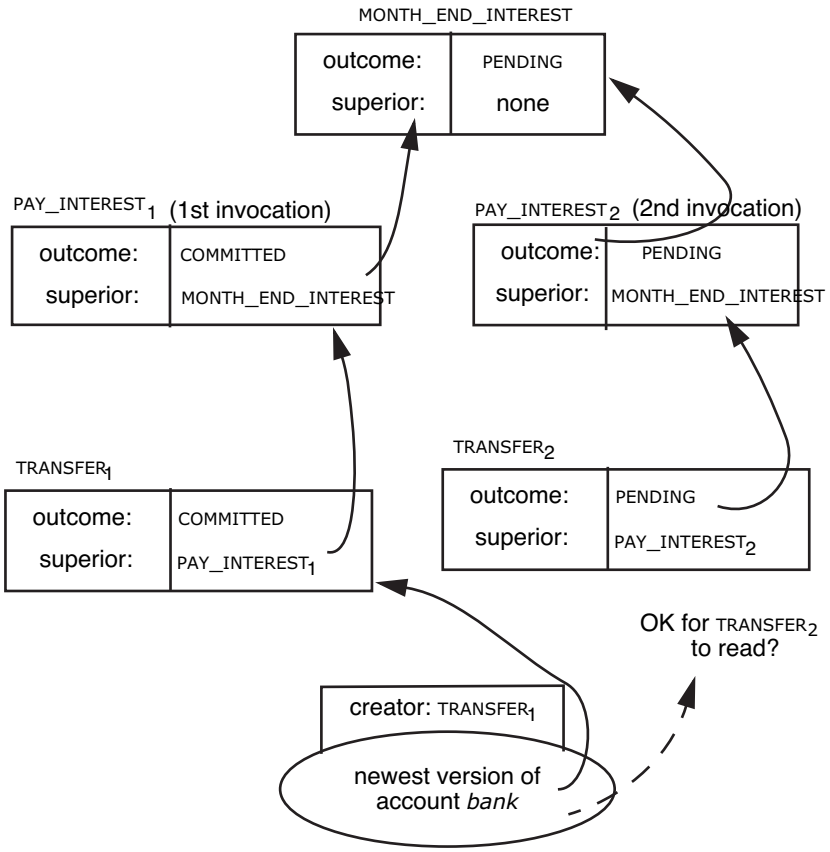


图9.36

事务TRANSFER2嵌套在事务PAY_INTEREST2中，而后者又嵌套在事务MONTH_END_INTEREST内，它想要读取账户*bank*的当前值。但*bank*最后一次是由事务TRANSFER1写入的，该事务嵌套在COMMITTED事务PAY_INTEREST1中，而后者又嵌套在仍在进行中的PENDING事务MONTH_END_INTEREST内。因此，*bank*的这个版本实际上是PENDING，而非仅通过观察TRANSFER1的结果可能得出的COMMITTED 。然而，TRANSFER1和TRANSFER2拥有共同的祖先（即MONTH_END_INTEREST），且从银行到该共同祖先的事务链已完全提交，因此对*bank* 的读取可以——并且为避免死锁，必须——被允许。

月末利息，且当前MONTH_END_INTEREST的结果取决于第二次调用PAY_INTEREST的成功与否，我们内置了一个等待循环，最多只能超时并中止。

由于阻止读取将是一个错误，何时允许读取由暂定COMMITTED事务创建的数据值才合适，这一问题需要进一步思考。原子性的前后要求是：任何可能存活的交易都不应看到由暂定COMMITTED事务做出的更新，以防该暂定COMMITTED事务最终因故中止。在此约束下，暂定COMMITTED事务的更新可以自由传递。我们可以通过以下方式实现这一目标：比较执行读取的事务的结果记录祖先与控制待读取版本的结果记录的祖先。若这些祖先未合并（即无共同祖先），则读取者必须等待该版本的祖先完全提交。若它们确实合并，且数据版本祖先中合并点以下的所有事务均为暂定提交，则无需等待。因此，在图9.36中，MONTH_END_INTEREST可能同时运行两个（或更多）PAY_INTEREST的调用。每次调用将作为其更新“bank”账户值计划的一部分调用CREATE_NEW_VERSION，从而建立调用的串行顺序。当后续PAY_INTEREST 调用READ_CURRENT_VALUE 读取“bank”账户值时，它们将被迫等待，直到所有先前PAY_INTEREST 的调用决定提交或中止。

9.6.2 两阶段提交

由于高层事务可由多个低层事务组成，我们可以将分层事务的提交描述为包含两个不同阶段。第一阶段，即所谓的*preparation*或*voting*阶段，高层事务会调用若干独立的低层事务，每个低层事务要么中止，要么通过提交进入暂态提交状态。顶层事务通过评估当前状况，确认所有（或足够数量）低层事务已暂态提交后，即可宣告高层事务成功完成。

基于该评估，它要么COMMIT要么ABORT更高层的事务。假设它决定提交，便进入第二个*commitment*阶段，在最简单的情况下，这一阶段仅需将其自身状态从PENDING更改为COMMITTED或ABORTED。若其为最高层事务，则在此刻所有较低层暂态提交的事务也将变为COMMITTED或ABORTED。若其本身嵌套于更高层事务中，则其转为暂态提交状态，其组成事务同样保持暂态提交状态。此处我们实现了一种称为*two-phase commit*的协调协议。下一节实现多站点原子性时，两阶段之间的区别将更加清晰。

如果系统采用版本历史记录来保证原子性，图9.36中的层级结构可以直接通过链接结果记录来实现。若系统采用日志机制，则可通过一个独立的待处理事务表来维护该层级结构，查询事务状态时需检查此表。

嵌套事务的层次化概念本身就有其价值，但我们特别关注嵌套的原因在于，它是构建多站点事务的两大基石中的第一块。为了构建第二块基石，接下来我们将探讨多站点事务与单站点事务的本质区别。

9.6.3 多站点原子性：分布式两阶段提交

如果一项事务需要在多个由尽力而为网络分隔的站点上执行组件事务，那么实现原子性将更为困难，因为用于协调各站点事务的任何消息都可能丢失、延迟或重复。在第四章中，我们了解了一种名为远程过程调用（RPC）的方法，用于在另一站点执行操作。第七章[在线]中，我们学习了如何设计如RPC这样的协议，配合持久化发送方以确保至少一次执行，并通过重复抑制确保至多一次执行。遗憾的是，这两种保证都无法精确满足确保多站点事务原子性的需求。然而，通过恰当地结合两阶段提交协议与持久化发送方、重复抑制及单站点事务，我们能够构建出正确的多站点事务。我们假设每个站点自身能够利用版本历史、日志、锁等技术实现本地事务，确保全有或全无的原子性及前后一致的原子性。若所有站点均提交或均中止，则多站点原子性协议的正确性即得以实现；反之，若部分站点提交其多站点事务部分而其他站点中止同一事务的对应部分，则意味着协议执行失败。

假设多站点事务由协调者Alice发起，她分别向工作站点Bob、Charles和Dawn请求执行组件事务X、Y和Z。简单地发出三个远程过程调用显然无法为Alice构成一个完整的事务，因为Bob可能执行了X，而Charles却报告无法完成Y。从概念上讲，协调者需要向三位工作者发送三条消息，例如发给Bob的消息如下：

> 来自：Alice　致：Bob　关于：我的交易91
>
> 如果（Charles做Y且Dawn做Z）那么请做X。

让那三位工人去处理细节。我们需要一些线索来了解鲍勃是如何完成这个奇怪请求的。

线索在于认识到协调者创建了一个更高层的事务，而每个工作者需要执行一个嵌套在该高层事务中的事务。因此，我们需要的是一种分布式版本的两阶段提交协议。复杂之处在于协调者与工作者无法可靠地

因此，问题就转化为构建一个可靠的两阶段提交协议的分布式版本。我们可以通过应用持久化发送者和重复抑制机制来实现这一点。

协议的第一阶段始于协调者Alice为整个交易创建一个顶层结果记录。然后，Alice开始持续向Bob发送类似RPC的消息：

发件人：Alice　收件人Bob　主题：关于我的交易271　请作为我交易的一部分执行X。

类似的消息从爱丽丝发送给查尔斯和道恩，同样提及交易271，并分别要求他们执行Y和Z操作。如同普通的远程过程调用一样，如果爱丽丝在合理时间内未收到一个或多个工作者的响应，她会向未响应的工作者重新发送消息，必要时重复多次，直至获得回应。

一个工作站点在收到此类请求时，会先检查是否有重复，然后创建自己的事务，但将其设为*nested*类型，其上级事务为爱丽丝的原始事务。接着，该站点会执行请求操作的预提交部分，并向爱丽丝反馈此部分进展顺利：

发件人：Bob　收件人Alice
主题：关于您的交易271
我的X部分已准备好提交。

爱丽丝在收集到一整套这样的响应后，便进入事务的两阶段提交环节，向鲍勃、查尔斯和道恩分别发送消息，例如：

T两阶段提交消息 #1:
来自：Alice　致Bob　关于：我的交易271　准备提交。

鲍勃在收到这条消息后，会进行提交——但只是试探性的——或者中止。在创建了持久的试探性版本（或将计划更新记录到日志存储中）并记录了一个结果条目，表明其状态为PREPARED（无论是提交还是中止）之后，鲍勃会持久地向爱丽丝发送一个响应，报告他的状态：

两阶段提交消息 #2： 发件人：Bob 收件人：Alice 主题：关于您的交易271 我已准备就绪（PREPARED）提交我方的部分。您是否已决定提交？此致敬礼。

或者，也可以是一条报告已中止的消息。如果Bob从Alice那里收到重复的请求，他的持久发送方会返回一个PREPARED或ABORTED 响应的副本。

此时，处于PREPARED状态的Bob陷入了孤立无援的境地。正如在局部层次嵌套中那样，Bob必须能够选择执行到底或中止操作，以无限期维持这种准备状态，并等待他人（如Alice）做出决定。此外，协调者可能独立崩溃或失去通信联系，这进一步加剧了Bob的不确定性。若协调者宕机，所有工作节点都必须等待其恢复；在该协议中，协调者构成了单点故障。

作为协调者，Alice从她的多个工作者那里收集响应消息（可能还会多次向某些工作站点重新请求PREPARED响应）。如果所有工作者都发送了PREPARED消息，两阶段提交的第一阶段就完成了。如果有任何工作者回复了中止消息，或者根本没有响应，Alice通常可以选择中止整个事务，或者尝试让另一个工作站点来执行该组件事务。第二阶段开始于Alice通过标记她自己的结果记录COMMITTED来提交整个事务。

一旦高层结果记录被标记为COMMITTED或ABORTED，Alice就会向Bob、Charles和Dawn各自发送一条完成消息：

两阶段提交消息 #3

来自：Alice 致Bob 主题：关于我的交易271 我的交易已提交。感谢你的帮助。

每个工作站点在接收到此类消息后，会将其状态从PREPARED更改为COM-MITTED，执行任何必要的提交后操作，然后退出。与此同时，Alice可以继续处理其他事务，但未来有一个重要要求：她必须可靠且无限期地记住此次事务的结果。原因在于，她发送的一个或多个完成消息可能已经丢失。处于PREPARED状态的任何工作站点都在等待完成消息以确定后续操作方向。若完成消息在合理时间内未到达，工作站点上的持久发送方将重新发送其PREPARED消息。每当Alice收到重复的PREPARED消息时，她只需返回该命名事务结果记录的当前状态即可。

如果一个使用日志和锁的工作站点崩溃，该站点的恢复过程需要采取三个额外步骤。首先，它必须将任何PREPARED事务归类为应恢复至PREPARED状态的暂定胜者。其次，如果该工作者正在使用锁来

在前后原子性操作中，恢复过程必须重新获取PREPARED 事务在故障发生时持有的所有锁。最后，恢复过程必须重启持久化发送器，以了解高层事务的当前状态。如果工作站点使用了版本历史记录，则仅需执行最后一步——重启持久化发送器。

由于工作者们作为其PREPARED消息的持久发送方持续运作，Alice可以确信每个工作者终将获悉她的交易已提交。然而，由于这些工作者的持久发送机制彼此独立，Alice无法确保它们会同步行动。她只能确定交易最终会完成。这种同步行动与最终行动之间的区别至关重要，这一点即将得到阐明。

如果一切顺利，*N*工作节点的两阶段提交将通过*3N*条消息完成，如图9.37所示：每个工作节点发送一条PREPARE消息，接收一条PREPARED响应消息，再发送一条COMMIT消息。尽管可以提出若干变体方案，但这一*3N*消息协议已是完备且充分的。

简化变体的一个例子是，初始的RPC请求和响应也可以分别携带PREPARE和PREPARED消息。然而，一旦工作节点发送了PREPARED 消息，它就失去了单方面中止的能力，必须保持在临界状态，等待协调器的指令。为了最小化这种等待，通常更可取的做法是延迟PREPARE/PREPARED消息对，直到协调器确认其他工作节点似乎已准备好执行各自的任务。

分布式两阶段提交协议的某些版本包含从工作站点到协调者的第四条确认消息。其目的是收集完整的确认消息集——协调者会持续发送完成消息，直至每个站点都予以确认。一旦所有确认到位，协调者便可安全地丢弃其结果记录，因为已知每个工作站点都已收到通知。

一个既关注结果记录存储空间又在意额外消息成本的系统，可以采用名为*presumed commit*的进一步优化方案。考虑到大多数事务预期会提交，我们可以为结果记录的值COMMITTED 采用一种略显奇特但极其节省空间的表示方式——不存在性。协调器通过发送COMMITTED 响应来答复任何关于不存在结果记录的查询。若协调器采用此表示法，它通过销毁结果记录来完成提交，因此无需来自每个工作者的第四条确认消息。作为这种在消息数量和空间上看似神奇减少的代价，我们注意到已中止事务的结果记录不易被丢弃，因为若在丢弃后收到查询，该查询将收到COMMITTED响应。然而，协调器可以持续要求对中止事务进行确认，并在收到所有确认后丢弃结果记录。这种导致结果记录被丢弃的协议，与第7章[在线]描述的关闭流并丢弃该流记录的协议完全相同。

分布式两阶段提交并不能解决所有多站点的原子性问题。例如，如果协调站点（此处为Alice）位于一艘沉没的船上，在{v*}之后

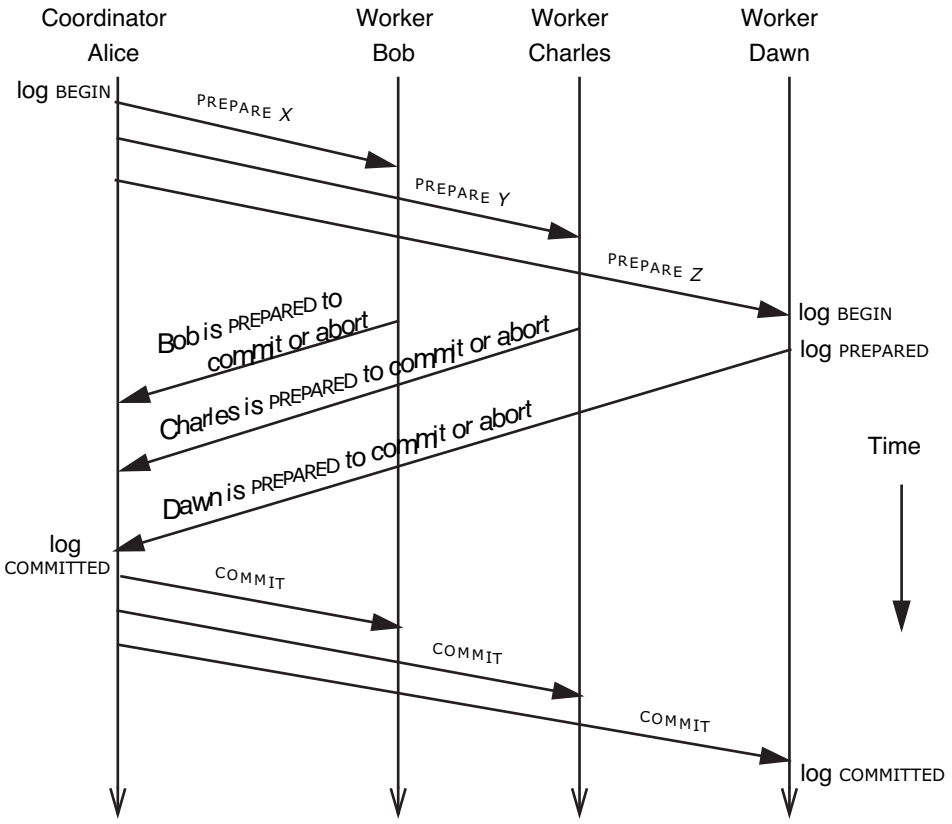


图9.37

分布式两阶段提交的时序图，使用3N条消息。（初始的RPC请求和响应消息未显示。）四位参与者各自维护其版本历史或恢复日志。图中展示了协调者与其中一位工作者的日志条目记录。

在发送PREPARE消息之后，但在发送COMMIT或ABORT消息之前，工作站点被遗留在了PREPARED状态，无法继续前进。即便不考虑这一点，Alice和她的同事们也正尴尬地站在一个多站点原子性问题边缘，这个问题至少在理论上*not*是可以解决的。唯一解救他们的是我们观察到，多个工作者最终会完成各自的部分，但不一定同时进行。如果Alice要求同时行动，那她可就麻烦大了。

这个无法解决的问题被称为*dilemma of the two generals.*

9.6.4 两将军难题

当通信不可靠时，对可能的协调协议的一个重要约束可以用一个生动的类比来概括，称为*dilemma of the two generals*。*假设两支小部队驻扎在城外两座山上。该城防御坚固，足以击退并摧毁其中任何一支军队。只有当两支军队同时发起进攻时，他们才能攻占该城。因此，指挥这两支军队的两位将军希望协调他们的进攻行动。

两位将军之间唯一的通讯方式，便是派遣信使往返于两座营地之间。然而，城中的守军在山谷间布下了哨兵，这片山谷将两座山脉分隔开来。因此，信使在试图穿越山谷时，有可能落入敌手，导致信息无法送达。

假设第一位将军发送了这条消息：

> 来自：尤利乌斯·凯撒
> 致：提图斯·拉比努斯
> 日期：1月11日
>
> 我提议渡过卢比孔河，明日拂晓发起进攻。可否?

期待第二位将军会以以下方式回应：

> 发自：提图斯·拉比努斯　致：尤利乌斯·凯撒；　日期：1月11日　是的，12日的黎明。

或者，可能：

> 发自：提图斯·拉比努斯　致：尤利乌斯·凯撒　日期1月11日　不。我正在等待来自高卢的增援。

进一步假设第一条消息未能成功传达。在这种情况下，由于未收到任何行动请求，第二位将军不会进军。此外，由于没有回应返回，第一位将军也不会采取行动，一切相安无事（除了那名不幸的信使）。

现在，假设跑者成功传递了第一条消息，而第二位将军发出了“同意”的回复，但该回复丢失了。第一位将军无法将这种情况与之前的情况区分开来，因此军队不会行动。第二位将军已同意行动，但知道除非“同意”的确认送达，否则第一位将军不会行动，所以第二位将军在没有确定第一位将军会行动的情况下也不会贸然行动。

---

* The origin of this analogy has been lost, but it was apparently first described in print in 1977 by Jim N. Gray in his “Notes on Database Operating Systems”, reprinted in *Operating Systems, Lecture Notes in Computer Science 60*, Springer Verlag, 1978. At about the same time, Danny Cohen described another analogy he called the dating protocol, which is congruent with the dilemma of the two generals.

将军收到了确认。第二位将军的犹豫表明，第一位将军应当发回一份确认收到的回执：

> 来自：尤利乌斯·凯撒
> 致：提图斯·拉比努斯
> 日期1月11日　骰子已掷出。

遗憾的是，这无济于事，因为携带这一确认信息的信使可能会丢失，而第二位将军未收到确认，依然不会进军。这就是困境所在。

我们现在可以直接得出一个结论：不存在一种消息数量有限的协议，能够同时让两位将军确信发起进攻是安全的。如果存在这样的协议，那么在该协议的任何一次具体执行中，*last*消息对于安全协调而言必然是非必需的，因为它可能无法察觉地丢失。既然最后一条消息必须是非必需的，那么我们可以删除这条消息，从而产生另一个更短的消息序列，该序列仍必须保证安全协调。我们可以对缩短后的消息序列重复应用相同的推理，进一步缩短序列长度，最终得出结论：若存在这样一种安全协议，它要么生成零长度的消息序列，要么生成无限长度的序列。零长度的协议无法传递任何信息，而无限长度的协议对将军们毫无用处，因为他们必须选择一个具体的进攻时机。

一位务实的将军，在战场上遇到数学家提出的这一困境时，会将该数学家调任为传令兵的新职务，并派遣侦察兵去探查山谷，报告在指定时间内成功穿越的概率。得知这一概率后，将军将派出多名（最好是互不影响的）传令兵，每人携带一份信息副本，所选传令兵的数量要足够多，以确保所有人在约定时间前都未能送达信息的概率微乎其微。（所有传令兵全部失联的情况，将被视为第8章[在线]所述的不可容忍错误。）同理，第二位将军也会派出多名传令兵，每人携带“是”或“否”的确认回复副本。这一流程为问题提供了实际解决方案，因此该困境并无实质影响。然而，发现一个在原则上无法完全确定解决的问题，仍不失为一件趣事。

我们可以更普遍且简洁地阐述这一理论结论：如果消息可能丢失，那么任何有限的协议都无法完全确保两位将军能同时确认他们将一起进军。他们所能达到的最佳效果，是接受一个非零的失败概率，这个概率等于他们最后一条消息未能送达的概率。

分析为何不能采用分布式两阶段提交协议来解决两位将军的困境，这一探讨颇为有趣。正如最初所指出的，关键在于*when*事情可能发生或必须发生的微妙差异。两位将军为了攻克城池的防御，必须在*same*同时进军。

分布式两阶段提交协议中的持久发送方确保，如果协调者决定提交，所有工作节点最终也将提交，但并不保证它们会同时进行。如果其中一条通信链路中断一天，当它恢复时，位于该链路另一端的工作节点随后会收到提交通知，但这一动作可能比其同事们的动作晚一天发生。因此，分布式两阶段提交所解决的问题与两将军困境相比略有放宽。这种放宽对两将军并无帮助，但事实证明，这种放宽恰好足以让我们设计出一个确保正确性的协议。

通过类似的推理方式，我们无法完全确保仅通过尽力而为网络通信的两个站点能同步执行操作。因此，分布式两阶段提交可以安全地打开东京一台ATM的现金抽屉，并确信慕尼黑的计算机终将更新该账户余额。但若因某些原因必须同时在不同地点开启两个现金抽屉，唯一解决方案要么采用概率性方法，要么设法用可靠网络替代尽力而为网络。房地产交易和婚礼（两者均为两阶段提交协议的实例）通常要求所有相关方共处一室，正是出于对可靠通信的需求。

9.7 更完整的磁盘故障模型（高级主题）

本章第9.2节为日历管理程序开发了一个故障分析模型，该系统崩溃时最多可能损坏一个磁盘扇区——即崩溃瞬间正在被写入的那个扇区（如果有的话）。该节还针对该问题提出了一种掩蔽策略，实现了全有或全无的磁盘存储。为了使该方案保持简洁，策略中未考虑衰变事件。本节将重新审视该模型，探讨如何同时掩蔽衰变事件。最终将实现全有或全无的持久存储，这意味着它既能在系统崩溃时保持全有或全无特性，又能在衰变事件面前保持持久性。

9.7.1 全有或全无且持久的存储

在第8章[在线]中，我们了解到要实现持久存储，应当为每个磁盘扇区写入两个或多个副本。而在本章中，我们认识到要从写入磁盘扇区时的系统崩溃中恢复，绝不应覆盖该扇区的旧版本，而应在不同位置写入新版本。为了获得既持久又具备全有或全无特性的存储，我们将这两点观察结合起来：创建多个副本，并且不覆盖先前版本。一个简单的方法就是直接在第八章[在线]的持久存储层之上构建本章的全有或全无存储层。这种方法固然可行，但略显粗放：仅以副本数为二为例，它会导致存——

为每个实际数据扇区分配六个磁盘扇区。这是模块化带来过高成本的一个例子。

回想一下，第八章[在线]用于确定磁盘存储完整性检查频率的参数是预期衰减时间T_d。假设目前仅需维护两份副本即可满足耐久性要求，那么T_d必须远大于在两块磁盘上写入一个扇区的两份副本所需时间。换言之，较大的T_d意味着短期内发生衰减事件的概率足够低，设计者可以安全地忽略它。我们可以利用这一观察结果，设计一种略带风险但经济得多的存储实现方法——仅用两个副本就能同时满足持久性和全有或全无特性。其核心理念是：若确信某数据有两个完好副本保障持久性，那么覆盖其中一个是安全的（就全有或全无原子性而言）；若系统在写入第一个副本时意外崩溃，第二个副本可作为备份确保全有或全无原子性。一旦确认第一个副本已正确写入新数据，便可安全覆盖第二个副本以恢复长期耐久性。若两次写入完成时间相较于T_d足够短，衰减事件干扰该算法的概率将微乎其微。图9.38展示了该算法及数据的两个副本，此处命名为*D0*和*D1*。

一个有趣的点是，ALL_OR_NOTHING_DURABLE_GET并不费心去检查读取*D1*时返回的状态——它只是将状态值传递给其调用者。原因在于，在没有衰减的情况下，CAREFUL_GET在读取CAREFUL_PUT已完成写入的数据时，预期会有*no*个错误。因此，返回的状态BAD仅在两种情况下出现：

1. CAREFUL_PUT的*D1*在操作过程中被中断，或
2. *D1* 遭遇了意外的衰减。

该算法确保第一种情况不会发生。ALL_OR_NOTHING_DURABLE_PUT在数据*D1*上不会开始CAREFUL_PUT，直到其在数据*D0*上的CAREFUL_PUT完成。由于在CAREFUL_PUT期间系统崩溃，最多只有其中一个副本可能被BAD。因此，如果第一个副本（*D0*）被BAD，那么我们预期第二个副本（*D1*）是OK。

第二种情况的风险确实存在，但我们假设其发生的概率很小：只有当*D1* 在远短于T_d的时间内发生随机衰减时才会出现。在读取*D1*时，我们有机会通过状态值*detect*来纠正该错误，但当两份数据副本都受损时，我们便无计可施，因此这种可检测到的错误必须被归类为不可容忍错误。我们所能做的就是向应用程序传递一份状态报告，让其知晓发生了不可容忍错误。

在SALVAGE程序中隐藏着一个目前不必要的步骤：如果*D0*等于BAD，将*D1*复制到*D0*上并无实际收益，因为调用SALVAGE的ALL_OR_NOTHING_DURABLE_PUT会立即用新数据覆盖*D0*。保留这一步骤的原因是它允许SALVAGE在算法的优化版本中被使用。

在没有衰减事件的情况下，该算法将与图9.6和9.7中的全有或全无程序同样有效，且由于仅涉及两份副本，其表现会略胜一筹。假设错误发生频率足够低，以至于恢复操作不会主导性能，ALL_OR_NOTHING_DURABLE_GET的常规成本仅为一次磁盘读取，而ALL_OR_NOTHING_GET算法则需要三次。ALL_OR_NOTHING_DURABLE_PUT的成本为两次磁盘读取（在SALVAGE中）和两次磁盘写入，相比之下，ALL_OR_NOTHING_PUT算法需要三次磁盘读取和三次磁盘写入。

该分析基于一个无衰减系统。为了处理衰减事件，从而使方案具备全有或全无*and*的持久性，设计者采用了第8章[在线]关于持久性讨论中的两个观点，其中第二个观点会消耗部分更优的性能：

1. 将两份副本*D0*和*D1*分别存放在独立的衰变集中（例如将它们写入两个不同的磁盘驱动器，最好来自不同厂商）。
2. 让一名职员至少每T_d秒在每个原子扇区上运行一次SALVAGE程序。

```
1   procedure ALL_OR_NOTHING_DURABLE_GET (reference data, atomic_sector)
2       ds ← CAREFUL_GET (data, atomic_sector.D0)
3       if ds = BAD then
4           ds ← CAREFUL_GET (data, atomic_sector.D1)
5       return ds

6   procedure ALL_OR_NOTHING_DURABLE_PUT (new_data, atomic_sector)
7       SALVAGE(atomic_sector)
8       ds ← CAREFUL_PUT (new_data, atomic_sector.D0)
9       ds ← CAREFUL_PUT (new_data, atomic_sector.D1)
10      return ds

11  procedure SALVAGE(atomic_sector)        //Run this program every T_d seconds.
12      ds0 ← CAREFUL_GET (data0, atomic_sector.D0)
13      ds1 ← CAREFUL_GET (data1, atomic_sector.D1)
14      if ds0 = BAD then
15          CAREFUL_PUT (data1, atomic_sector.D0)
16      else if ds1 = BAD then
17          CAREFUL_PUT (data0, atomic_sector.D1)
18      if data0 ≠ data1 then
19          CAREFUL_PUT (data0, atomic_sector.D1)
```

图9.38

数据排列与算法，用于在图8.12的谨慎存储层之上实现全有或全无的持久存储。

运行SALVAGE程序的职员每T_d秒执行2N次磁盘读取，以维护N个持久化扇区。这一额外开销是为了抵御磁盘衰变而付出的持久性代价。职员的性能成本取决于T_d的选择、N的值以及职员的优先级。由于硬盘的预期使用寿命通常为几年，将T_d设置为数周应能使因衰变导致的不可容忍故障概率微乎其微，尤其是在运营实践中还会在磁盘远未达到预期寿命前就定期更换的情况下。一块容量为一太字节的现代硬盘大约包含N = 109个千字节大小的扇区。若读取一个扇区需10毫秒，则职员读取两块一太字节硬盘的全部内容约需2×10^7秒，即两天时间。若将职员的工作安排在夜间执行，或采用优先级系统在系统空闲时运行该程序，则读取操作可分散在数周内完成，对性能的影响微乎其微。

前文曾提到存在一种优化的可能性：如果在每次系统崩溃后立即对每个原子扇区运行SALVAGE程序，那么就不必在每次ALL_OR_NOTHING_DURABLE_PUT开始时进行这一操作。这种变体由巴特勒·兰普森和霍华德·斯特吉斯[进一步阅读建议1.8.7]提出，在崩溃不频繁且磁盘容量不大的情况下更为经济。但它引发了一个小问题：该方案依赖于两种故障极少同时发生——即一个数据副本自然衰变的同时，CAREFUL_PUT在重写该扇区另一副本的过程中崩溃。若能确信此类巧合极为罕见，我们可将其归为不可容忍的误差，从而获得一个自洽且更经济的算法。此方案下，ALL_OR_NOTHING_DURABLE_PUT的成本可降至仅两次磁盘写入。

9.8 案例研究：机器语言原子性

9.8.1 复杂指令集：通用电气600系列

在大型计算机的早期，大多数制造商热衷于提供复杂的指令集，却很少关注原子性问题。通用电气600系列——后来演变为霍尼韦尔信息系统公司（Honeywell Information System, Inc.）的68系列计算机架构——拥有一项名为“间接与计数”的功能。通过将任意间接地址中未使用的高位比特（即“计数”标志位）设置为ON，即可启用该功能。该指令

从Y间接加载寄存器A。

被解释为意味着地址为Y的单元的低位部分包含另一个地址，称为间接地址，该间接地址应被用来获取要加载到寄存器A中的操作数。此外，如果单元Y中的计数标志为ON，处理器会将Y中的间接地址加一，并将结果存回Y。其目的是当下次Y被用作间接地址时，它将指向一个不同的

操作数——内存中下一个连续地址中的那个。因此，间接寻址与计数特性可用于遍历整个表格。这一特性对设计者而言似乎很有用，但实际上仅偶尔派上用场，因为大多数应用程序是用高级语言编写的，而编译器编写者发现难以利用该特性。另一方面，当产品线后期引入虚拟内存时，这一特性却带来了无尽的麻烦。

假设正在使用虚拟内存，且间接字位于主存中的一个页面内，而实际的操作数却位于已被移出至辅助存储器的另一个页面。当执行上述指令时，处理器会从*Y*中获取间接地址，对其进行递增，并将新值存回*Y*。随后，它将尝试获取实际的操作数，此时发现该操作数不在主存中，于是触发缺页异常。由于处理器已经修改了*Y*的内容（而此时*Y*可能已被其他处理器读取，甚至被运行在其他处理器上的缺页异常处理程序从内存中移除），回退并假装该指令从未执行已不可行。　　异常处理程序的设计者希望在等待缺失页面加载期间，能够通过调用诸如AWAIT这样的函数将处理器让渡给其他线程。实际上，重新分配处理器可能是唯一能将处理器指派去获取缺失页面的方式。然而，要重新分配处理器，必须保存其当前的执行状态。不幸的是，其执行状态正处于“程序计数器最后指向的指令执行到一半”的状态。保存这一状态并在之后从此状态重启处理器颇具挑战性。间接计数特性只是虚拟内存引入该处理器后涌现的多个原子性问题来源之一。

虚拟内存设计者迫切希望在中断的处理器上运行其他线程。为解决这一问题，他们扩展了当前程序状态的定义，使其不仅包含下一条指令计数器及程序可见寄存器，还囊括了处理器的完整内部状态描述——即指令执行过程中的216位快照。通过后续恢复处理器状态至先前保存的下一条指令计数器值、程序可见寄存器值及216位内部状态快照，处理器能够精确地从触发缺页警报的点继续执行。该技术虽有效，却带来两个棘手副作用：1）当程序（或程序员）查询中断处理器的当前状态时，状态描述会包含超出程序员接口范畴的内容；2）系统在重启中断程序时必须确保存储的微状态描述是有效的。若有人篡改状态描述，处理器可能试图从一个自身永远无法进入的状态继续执行，这将导致非预期行为，包括内存保护功能的失效。

9.8.2 更复杂的指令集：IBM System/370

当IBM通过在其System/360架构上添加虚拟内存来开发System/370时，某些System/360的多操作数字符编辑指令导致了

原子性问题。例如，TRANSLATE指令包含三个参数，其中两个是内存中的地址（称为*string*和*table*），第三个参数*length*是一个8位计数，该指令将其解释为*string*的长度。TRANSLATE 每次从*string*中取出一个字节，将该字节作为*table*中的偏移量，获取偏移量处的字节，并用其在*table*中找到的字节替换*string* 中的字节。设计者设想TRANSLATE可用于将字符串从一种字符集转换为另一种字符集。

添加虚拟内存的问题在于，*string*和*table*都可能长达65,536字节，因此这两个操作数中的任意一个或两者都可能跨越不止一个，而是多个页面边界。假设只有*string*的第一个页面在物理内存中。TRANS-LATE 指令会逐个处理字符串开头的字节。当它到达该第一页的末尾时，会遇到缺页异常。此时，由于所需数据缺失，指令无法完成执行。它也无法回退并表现得好像从未开始执行，因为它已经通过覆写修改了内存中的数据。在虚拟内存管理器获取缺失的页面后，问题在于如何重新启动这个半完成的指令。如果从头开始重启，它将尝试转换已经转换过的字符，这将是一个错误。为了正确操作，指令需要从中断处继续执行。

IBM处理器的设计者并未选择篡改程序状态定义，而是采取了一种*dry run*策略：利用程序可见寄存器的隐藏副本执行TRANSLATE 指令，且不对内存做任何更改。若某个操作数引发缺页异常，处理器可表现得仿佛从未尝试执行该指令，因为程序可见层面没有任何执行痕迹。存储的程序状态仅显示TRANSLATE指令即将被执行。当处理器获取缺失页后，它会通过从头重试TRANSLATE 指令来恢复被中断的线程——这又是一次空运行。若存在多个缺失页，可能会发生多次空运行，每次都将一个新增页调入主存。当某次空运行最终成功完成时，处理器会再次执行该指令，这次才是真实执行：使用程序可见寄存器并允许更新内存。由于System/370（进行此修改时）是单处理器架构，不存在其他处理器在空运行后、指令真实执行前抢走页面的可能性。该解决方案的副作用是给后来承担多处理器扩展任务的设计者增加了难度。

9.8.3 阿波罗桌面计算机与摩托罗拉M68000微处理器

当阿波罗计算机公司采用摩托罗拉68000微处理器设计一款桌面电脑时，希望加入虚拟内存功能的设计师们发现，该微处理器的指令集接口并非原子操作。更糟的是，由于该处理器完全集成在单一芯片上，无法像IBM 370那样进行试运行，也无法像通用电气600系列那样存储内部微程序状态。因此，阿波罗的设计师们采取了不同的策略：他们安装的不是一个，而是两个

摩托罗拉68000处理器。当第一个处理器遇到缺页异常时，它会在当前状态直接停止运行，等待操作数出现。第二个摩托罗拉68000处理器（其程序经过精心规划，完全驻留在主内存中）则负责获取缺失的页面，然后重新启动第一个处理器。

其他与摩托罗拉68000合作的设计师采用了一种不同且略显冒险的技巧：修改所有编译器和汇编器，使其仅生成恰好为原子操作的指令。摩托罗拉后来推出了68000的一个版本，其中微处理器的所有内部状态寄存器均可被保存，这一方法与通用电气600系列添加虚拟内存时所采用的相同。

练习

9.1 *Locking up humanities*：教务处的办公室正在升级其针对有限招生人文学科的排课程序。计划将其改为多线程运行，但有人担心多个线程同时尝试更新数据库可能会引发问题。该程序最初仅包含两项操作：

```
status ← 注册 (subject_name)
删除 (subject_name)
```

其中*subject_name* 是一个类似“21W471”的字符串。REGISTER过程会检查该科目是否还有剩余空间，若有，则将班级规模增加一，并返回状态值ZERO。若无剩余空间，则不改变班级规模，而是返回状态值-1。（这是一个简易的注册系统——仅作人数统计！）

作为升级的一部分，*subject_name* 已改为双组件结构：

```
结构 subject   字符串
subject_name
    锁定 slock
```

而注册员现在正琢磨着该在哪里应用这些锁定原语，

```
获取（subject.slock）
发布 (subject.slock)
```

这是一个典型的应用程序，它将调用者注册为两个人文领域的{v*}

主题，*hx* 和 *hy*：
procedure REGISTER_TWO (*hx*, *hy*)
status ← 注册 (*hx*) 如果 *status* = 0 则 *status* ← 注册 (*hy*) 如果 *status* = –1 则 删除 (*hx*) 返回 *status*;

9.1a. 目标是整个流程REGISTER_TWO应具备前后顺序属性。向REGISTER_TWO流程中添加对ACQUIRE和RELEASE的调用，这些调用需遵循*simple locking protocol*。

9.1b. 添加对ACQUIRE和RELEASE的调用，这些调用需遵守*two-phase locking protocol*，此外尽可能推迟所有ACQUIRE，并尽早完成所有RELEASE。

路易斯·里森纳提出了一项他认为能简化程序员创建应用程序工作的建议，比如REGISTER_TWO。他的想法是通过让两个程序REGISTER和DROP在内部完成ACQUIRE和RELEASE来对它们进行修订。也就是说，该流程：

过程 注册 (*subject*)
{ 当前代码 } 返回 *status* 将改为：

过程 注册(*subject*)
获取 (*subject.slock*) { 当前代码 } 释放 (*subject.slock*) 返回 *status*

9.1c. 和往常一样，路易斯误解了问题的某个方面。简要解释一下这个想法的问题所在。

1995–3–2a…c

9.2 本和艾丽莎正在就版本历史事务处理的某个细节进行辩论，他们希望得到你的帮助。本认为，在标记点事务处理规则下，每个事务都应尽快调用MARK_POINT_ANNOUNCE，否则该规则将失效。艾丽莎则声称，即使没有事务调用MARK_POINT_ANNOUNCE，一切仍会正确无误。谁是对的?

2006-0-1

9.3 本和艾丽莎正在就版本历史事务规则如何自举的另一个细节展开辩论。文中给出的NEW_OUTCOME_RECORD版本同时使用了TICKET、ACQUIRE和RELEASE。艾丽莎认为这有些过度——

应该可以仅通过ACQUIRE 和RELEASE来正确协调NEW_OUTCOME_RECORD。修改图9.30的伪代码，创建一个不需要票证原语的NEW_OUTCOME_RECORD版本。

9.4　你受雇于Many-MIPS公司，协助设计一款新型32寄存器RISC处理器，该处理器将实现六路多指令发射。你的职责是协调六个并行运行的算术逻辑单元（ALU）之间的交互。回顾关于协调的讨论后，你意识到首要任务是明确多指令发射系统中何为“正确”的协调。对于数据库并发操作的正确协调，曾定义为：

无论实际计算顺序如何，最终结果始终保证与某种并发操作的顺序执行所能得到的结果一致。

你有两个目标：(1) 最大化性能，(2) 不让程序员感到意外——他们编写的程序原本预期在单指令发射机器上执行。

标识为你的问题确定最佳的协调正确性标准　　米

A. 多指令发射必须限制在寄存器集不重叠的指令序列上。 B. 无论实际计算顺序如何，最终结果始终保证与某种并行发射指令的顺序执行结果一致。 C. 无论实际计算顺序如何，最终结果始终保证与并行发射指令的原始顺序执行结果一致。 D. 最终结果必须按照原始程序指定的操作顺序执行得出。 E. 无论实际计算顺序如何，最终结果始终保证与某种顺序执行的指令集结果一致。 F. 六个算术逻辑单元(ALU)无需任何协调。

1997–0–02

9.5　1968年，IBM推出了信息管理系统（IMS），它迅速成为全球使用最广泛的数据库管理系统之一。事实上，IMS至今仍在使用。在推出之初，IMS采用了一种“前或后”原子性协议，该协议包含以下两条规则：

- 一个事务只能读取由先前已提交事务写入的数据。
- 事务必须为它将写入的每个数据项获取一个锁。

考虑以下两个事务，对于所示交错执行，两者均遵循协议：

~~3~~	~~开始 (t1)~~	开始 (*t2*)	2 获取 (*y.lock*)
4		获取 (*x.lock*)	
5			*temp2* ← *y*
6			*x* ← *temp2*
7	*y* ← *temp1*	提交 (*t1*)	9 提交 (*t2*)
8			

先前提交的事务已将*x* ←设为3，*y* ←设为4。

9.5a. 在两次交易都完成后，*x* 和 *y* 的值分别是多少？从什么意义上说这个答案是错误的?

1982–3–3a

9.5b. 在20世纪70年代中期，人们注意到了这一缺陷，尽管客户并未提出投诉，但“前后”原子性协议仍被更优方案取代。试解释客户可能未就该缺陷提出抱怨的原因。

1982–3–3b

9.6 一个试图将操作设为全有或全无的系统，会向非易失性存储中维护的日志写入以下类型的记录：

- <开始 *i*> 动作 *i* 开始。
- <*i*, *x*, *old*, *new*> 动作 *i* 将值 *new* 覆盖到变量 *x* 的原值 *old* 上。

- <已提交*i*> 动作 *i* 提交。
- <已中止*i*> 动作 *i* 中止。
- <检查点 *i*, *j*,…> 在此检查点，操作 *i*、*j*…… 待处理。

动作按数字顺序开始。发生崩溃后，恢复程序发现

以下日志记录从最后一个检查点开始:

<检查点 17, 51, 52> <启动 53> <启动 54> <53, *y*, 5, 6> <53, *x*, 5, 9 > <提交 53> <54, *y*, 6, 4> <启动 55> <55, *z*, 3, 4> <中止 17> <51 , *q*, 1, 9> <启动 56> <55, *y*, 4, 3> <提交 54> <55, *y*, 3, 7> <提交 51 > <启动 57> <56, *x*, 9, 2> <56, *w*, 0, 1> <提交 56> <57, *u*, 2, 1> * ********** 此处发生崩溃 **********

9.6a. 假设系统正在使用回滚恢复程序。恢复程序应在日志中向后扫描多远? 9. 6b. 假设系统正在使用前滚恢复程序。恢复程序应在日志中向后扫描多远? 9.6c . 日志这部分提到的操作中，哪些是胜者，哪些是败者? 9.6d. 恢复程序完成后 ，*x*和*y*的值分别是多少？为什么?

1994–3–3

9.7 练习9.6的日志中包含了（可能含糊的）证据，表明有人未遵守协调规则 。这一证据是什么?

1994–3–4

9.8 前滚恢复要求在将提交（或中止）记录写入日志*before*之前，先完成对单元存储的任何安装操作。请指出这一要求的最佳理由。

A. 这样恢复管理器就能知道需要撤销哪些操作。 B. 这样恢复管理器就能知道需要重做哪些操作。 C. 因为日志比单元存储更不容易失效。 D. 为了最小化所需的磁盘寻道次数。

1994–3–5

9.9 事务内的两阶段锁定确保

A. 不会发生死锁。 B. 结果将与事务的某种串行执行相对应。 C. 资源将被锁定在尽可能短的时间间隔内。 D. 气体和液体均不会逸出。 E. 即使一次锁定尝试失败，事务仍会成功。

1997–3–03

9.10 帕特、黛安和昆西在使用电子邮件安排会议时遇到了困难。帕特建议他们从两阶段提交协议中汲取灵感。 9.10a. 以下哪种协议与两阶段提交最为相似?

一、 a. 帕特征求每个人的空闲时间安排。 b. 大家回复各自的空闲时段列表，但无法保证所有时间都能预留。 c. 帕特检查这些列表，寻找共同空闲时间。 若存在合适时段， 帕特选定会议时间并通知所有人。 否则 帕特将发送取消会议的通知。

二、 a–c，同协议一。 d. 所有人若收到第二条消息， 需确认接收。 否则 向Pat发送消息询问情况。

III a–c，同协议I。 d. 每个人，如果在选定时间其日程仍为开放 向Pat发送确认。 否则 向Pat发送致歉。 e. Pat收集确认。若全部为肯定 向所有人发送会议“如期举行”的通知。 否则 向所有人发送会议“取消”的通知。f. 每个人，若收到“举行/取消”通知， 需确认接收。 否则 向Pat发送询问情况的消息。

IV. a–f，同协议III。 g. Pat发送一条消息告知所有人均已确认。 h. 所有人对确认进行回应。 9.10b. 对于您选择的协议，哪一步骤确定了会议时间?

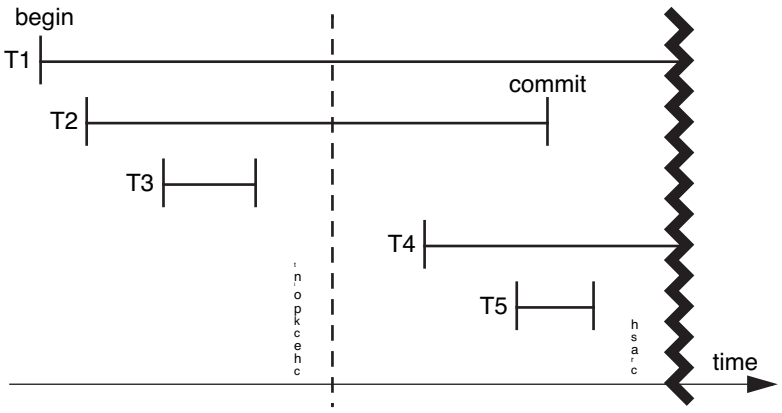
1994–3–7

9.11 Alyssa P. Hacker需要一个事务处理系统来更新她收集的97只蟑螂的相关信息。*

9.11a. 在艾丽莎的第一个设计中，她将数据库存储在磁盘上。当一个事务提交时，它直接访问磁盘，并在旧数据的位置上就地写入更改。艾丽莎的系统存在哪些主要问题?

9.11b. 在阿丽莎的第二个设计方案中，她存储在磁盘上的*only*结构是一个日志，其中包含易失性RAM中所有数据的参考副本。该日志记录了数据库的每一次更改，以及更改所属的事务。同样存储在日志中的提交记录，用于标识事务何时提交。当系统崩溃并恢复时，它会重放日志，重新执行每个已提交的事务，以重建RAM中的参考副本。阿丽莎的第二个设计有哪些缺点?

为了加快进程，艾丽莎会偶尔对她的数据库进行一次检查点操作。进行检查点时，艾丽莎只需将数据库的完整状态写入日志。当系统崩溃时，她从最后一个检查点状态开始，然后重做或撤销某些事务以恢复数据库。现在考虑图示中的五个事务：



事务T2、T3和T5在崩溃前已提交，但T1和T4仍处于待定状态。

9.11c. 当系统恢复时，在加载检查点状态后，需要使用日志对某些事务进行撤销或重做。对于每个事务，

* Credit for developing exercise 9.11 goes to Eddie Kohler.

在表格中标记该事务是需要撤销、重做，还是两者都不需要。

	Undone	Redone	Neither
T1			
T2			
T3			
T4			
T5			

9.11d. 现在，假设事务T2和T3实际上是*nested*事务：T2嵌套在T1中，而T3又嵌套在T2内。再次填写表格

	Undone	Redone	Neither
T1			
T2			
T3			
T4			
T5			

1996–3–3

9.12 Alice 在 两阶段提交协议 中 担任 Bob 和 Charles 的协调者。以下是他们之间传递的消息日志:

1 爱丽丝 ⇒ 鲍勃：请执行X *2* 爱丽丝 ⇒ 查尔斯：请执行Y *3* 鲍勃 ⇒ 爱丽丝：X已完成 *4* 查尔斯 ⇒ 爱丽丝：Y已完成 *5* 爱丽丝 ⇒ 鲍勃：准备提交或中止 *6* 爱丽丝 ⇒ 查尔斯：准备提交或中止 *7* 鲍勃 ⇒ 爱丽丝：已准备就绪 *8* 查尔斯 ⇒ 爱丽丝：已准备就绪 *9* 爱丽丝 ⇒ 鲍勃：提交 *10* 爱丽丝 ⇒ 查尔斯：提交

在这个序列的哪些节点上，Bob可以中止他的部分操作

交易?

A. 在Bob收到消息1之后，但在发送消息3之前。 B. 在Bob发送消息3之后，但在收到消息5之前。 C. 在Bob收到消息5之后，但在发送消息7之前。 D. 在Bob发送消息7之后，但在收到消息9之前。 E. 在Bob收到消息9之后。

2008–3–11

与第9章相关的额外练习题可以在习题集***29*** 至***40***中找到。

第9章术语表

中止（abort）——在判定一个全有或全无（all-or-nothing）动作无法或不应提交时，撤销该动作之前所做的所有更改。中止后，从实现全有或全无动作的层次之上观察，系统的状态如同该动作从未存在过。与{v*}对比。[第9章]

全有或全无原子性——一种多步骤操作的属性，指如果在操作步骤中发生预期故障，从调用者的角度看，该操作的效果要么从未开始，要么已完全完成。与*before-or-after atomicity*和*atomic*进行比较。[第9章]

存档（archive）——一种通常以日志形式保存的记录，用于存储旧数据值，目的包括审计、从应用程序错误中恢复或满足历史研究需求。[第9章]

原子性（adj.）；原子性（n.）——一种多步操作的特性，即在实现该操作的层次之上无迹象表明其具有复合性。原子操作可以是"前或后"的，这意味着其效果表现为要么完全在其他任何"前或后"操作之前发生，要么完全在其之后发生。原子操作也可以是"全有或全无"的，这意味着若操作期间发生预期故障，高层所见的操作效果要么从未开始，要么已成功完成。一个同时满足*both*"全有或全无"和"前或后"特性的原子操作被称为*transaction*。[第9章]

原子存储——一种多单元PUT存储，其仅可能产生两种结果：(1) 成功存储所有数据，或 (2) 完全不改变原有数据。因此，无论是并发线程还是（在发生故障后）后续执行GET的线程，读取到的数据要么全部为旧数据，要么全部为新数据。若计算机架构中的多单元PUT操作不具备原子性，则称其易受*write tearing*影响。[第9章]

前后原子性——并发操作的一种属性：若从调用者的视角看，并发操作的效果等同于这些操作要么完全在另一个之前执行，要么完全在另一个之后执行，则这些操作具有前后原子性。一个必然结果是，具有前后原子性的并发软件操作无法察觉彼此的复合性质（即一个操作无法识别另一个操作包含多个步骤）。在硬件层面的体现是，对同一内存单元的并发前后WRITE操作将按某种顺序执行，因此不存在该单元最终存储例如多个WRITE值的OR的风险。数据库文献使用"隔离性"和"可串行化"表述此概念，操作系统文献采用"互斥"和"临界区"，而计算机体系结构文献则直接使用"原子性"这一术语。对比*all-or-nothing atomicity*与*atomic*。[第9章]

盲写——事务对数据值*X*的更新，而该事务之前并未读取*X*。[第9章]

单元存储——一种存储方式，其中WRITE或PUT通过覆写操作运行，从而破坏先前存储的信息。包括磁盘和CMOS随机存取存储器在内的许多物理存储设备都实现了单元存储。与*journal storage*进行比较。[第9章]

检查点—1. (名词) 写入非易失性存储器的信息，旨在加速系统崩溃后的恢复。2 (动词) 执行检查点写入操作。[第9章]

关闭-打开一致性（close-to-open consistency）——一种针对文件操作的并发一致性模型。当某个线程打开文件并执行若干写入操作时，所有修改内容只有在首个线程关闭文件后，才会对其他并发线程可见。[第4章]

一致性——参见*read/write coherence*或*cache coherence*。 提交——指放弃单方面中止一项全有或全无操作的能力。通常，在将全有或全无操作的结果提供给并发或后续的全有或全无操作之前，会先提交该操作。提交前，全有或全无操作可被中止，且可当作其从未执行过；提交后，该操作必须能够完成。已提交的全有或全无操作不可中止；若能精确确定其结果的传播范围，则可能通过补偿来逆转部分或全部效果。提交通常还包含一种预期，即结果将保持所有适当的不变性，并根据应用需求确保这些属性的持久性。对比*compensate*和*abort*。[第9章]

补偿（形容词）；补偿（名词）——执行一个动作以逆转之前某个已执行动作的效果。补偿本质上依赖于具体应用；相比撤销一个不想要的钻孔，纠正一个错误的会计条目要容易得多。[第9章]

执行动作——（名词）在某些系统中用于指代*redo action*的术语。[第9章]

指数随机退避——一种*exponential backoff*形式，其中反复遭遇干扰的动作会不断将延迟区间的大小加倍（或更一般地说，乘以大于1的常数），然后从中随机选择下一次重试前的等待时间。其目的是通过相对于其他干扰动作随机调整时机，避免干扰再次发生。[第9章]

强制（动词）——当输出可能被缓冲时，确保先前的输出值已实际写入持久存储或作为消息发送。非直写式缓存通常具备一项功能，允许调用者强制将其部分或全部内容推送至二级存储介质。[第9章]

安装——在使用日志实现全有或全无原子性的系统中，将数据写入单元存储。[第9章]

日志存储——一种存储方式，其中WRITE或PUT会追加新值，而非覆盖先前存储的值。与*cell storage*进行比较。[第9章]

锁定点——在通过锁定提供前后原子性的系统中，一个前后动作中的第一个瞬间，此时其锁定集中将包含的所有锁都已被获取。

获取。[第9章]　锁集合——在执行一个“之前或之后”动作期间获取的所有锁的集合。[第9章]　日志—1.（名词）一种专门用途的日志存储，用于维护某些应用活动的仅追加记录。日志用于实现全有或全无动作、性能提升、归档及对账。2.（动词）向日志追加一条记录。[第9章]　逻辑锁定——对更高层数据对象（如数据库记录或字段）的锁定。与*physical locking*比较。[第9章]　标记点——1.（形容词）一种确保原子性的规则，其中每个新创建的动作n必须等待，直到动作$(n-1)$标记了它打算修改的所有变量后，才能开始读取共享数据对象。2.（名词）动作标记完其打算修改的所有变量的瞬间。[第9章]　乐观并发控制——一种并发控制方案，允许并发线程继续执行，即使存在它们可能相互干扰的风险，计划是检测是否实际发生干扰，并在必要时强制其中一个线程中止并重试。在干扰可能但不太可能发生的情况下，乐观并发控制是一种有效的技术。与*pessimistic concurrency control*比较。[第9章]

页面错误——参见*missing-page exception*。

配对与备用——参见*pair-and-compare*。　待定pending）——全有或全无操作的一种状态，表示该操作尚未提交或中止。也用于描述由仍处于待定状态的全有或全无操作所设置或更改的变量值。[第9章]　悲观并发控制——一种并发控制方案，当线程有可能干扰其他并发线程时，强制其等待。在并发线程间干扰概率较高的场景中，悲观并发控制是一种有效技术。对比*optimistic concurrency control*。[第9章]　物理锁定——对底层数据对象的锁定，通常锁定由存储介质物理布局决定的数据块。例如磁盘扇区或整个磁盘。对比*logical locking*。[第9章]　预分页——多级内存管理器的优化策略，管理器预测可能需要的页面并在应用程序请求前将其预加载至主存。对比*demand algorithm*。

已准备prepared）——在分层或多站点的全有或全无操作中，组件操作的一种状态，表明其已声明可根据指令提交或中止。进入此状态后，等待上层协调器作出决策。[第9章]　呈现负载——参见*offered load*。

进展（progress）——由原子性保障机制提供的一种理想保证：尽管存在并发带来的潜在干扰，系统仍能完成某些有用工作。此类保证的一个例子是，原子性保障机制至少不会中止并发操作集中的任一成员。实践中，缺乏进展保证时，有时可通过指数随机退避机制进行补救。在系统的形式化分析中，进展是称为“活性”（liveness）属性的组成部分。进展确保系统将朝着特定目标推进，而活性则保证系统最终会达成该目标。[第9章]

redo action 重做操作（redo action）——应用程序指定的一个操作，在故障恢复期间执行时，能产生某些已提交组件操作的效果（这些效果可能在故障中丢失）。（某些系统称之为“执行操作”。对比*undo action.*）[第9章]

roll-forward recovery 前滚恢复（roll-forward recovery）——一种预写日志协议，额外要求应用程序在执行任何安装操作前记录其结果记录*before*。若全有或全无动作在通过提交点前发生故障，恢复过程无需撤销任何操作；若在提交后发生故障，恢复过程可利用日志记录确保存储单元的安装操作不会丢失。亦称*redo logging*。对比*rollback recovery*。[第9章]

rollback recovery 回滚恢复（rollback recovery）——一种预写日志协议，额外要求应用程序在记录结果记录前执行所有安装操作*before*。若全有或全无动作提交前发生故障，恢复过程可利用日志记录撤销部分完成的动作。亦称*undo logging*。对比*roll-forward recovery*。[第9章]

serializable 可串行化（serializable）——先于或后于操作的属性，即使多个操作并发执行，其结果仍等同于它们按某种顺序（即串行）依次执行的效果。[第9章]

shadow copy 影子副本（shadow copy）——全有或全无动作为对象创建的工作副本，用于在保持原对象不变的同时进行多次修改。当完成所有修改后，该动作会谨慎地将工作副本与原对象交换，从而保持所有修改原子性发生的表象。根据实现方式，原对象或工作副本可能被标识为“影子”副本，但两种情况下技术原理相同。[第9章]

simple locking 简单锁（simple locking）——创建先于或后于操作的锁协议，要求在到达锁定点前不得读取或写入任何数据。若要使原子动作同时满足全有或全无性，还需满足在提交（或中止）前不得释放任何锁。对比*two-phase locking*。[第9章]

simple serialization 简单串行化（simple serialization）——一种原子性协议，要求每个新创建的原子动作必须等待所有先前启动的原子动作不再处于待处理状态后才能开始执行。[第9章]

transaction 事务（transaction）——一种多步骤动作，既具备故障原子性，又具备...

并发性的体现。也就是说，它既是全有或全无的，也是之前或之后的。[第9章] 事务性内存——一种内存模型，其中对主内存的多次引用既是全有或全无的，也是之前或之后的。[第9章]

两将军问题——一个本质性问题，即任何有限协议都无法保证在由不可靠通信网络连接的两个地点间同步协调状态值{v*}。[第9章]

两阶段提交——一种将独立的底层事务组合成高层事务的协议。该协议首先经历一个准备阶段（有时称为投票阶段），在此阶段结束时，每个底层事务会报告其无法执行自身部分，或已准备好提交或中止。随后进入提交阶段，由高层事务作为协调者做出最终决定——因此得名“两阶段”。两阶段提交与发音相似的术语*two-phase locking*无任何关联。[第9章]

两阶段锁定——一种用于实现“之前或之后”原子性的锁定协议，要求在所有锁被获取之前不得释放任何锁（即必须存在一个锁定点）。为了确保原子操作同时也是“全有或全无”的，进一步要求是：在操作提交之前，不得释放任何待写入对象的锁。与*simple locking*进行比较。两阶段锁定与听起来相似的术语*two-phase commit*并无关联。[第9章]

撤销操作——一种由应用程序指定的动作，在故障恢复或中止过程中执行时，能够逆转某些先前执行但尚未提交的组件操作的效果。其目的是使原始操作及其逆转在实现该操作的层次之上均不可见。与*redo*和*compensate*进行比较。[第9章]

版本历史——一个对象或变量曾经存在过的所有值的集合，存储在日志存储中。[第9章]　预写日志W(AL）协议——一种恢复协议，要求在将相应数据安装到单元存储之前，先在日志存储中追加一条日志记录。[第9章]　写撕裂——参见*atomic storage*。

第9章索引

设计原则和提示出现在*underlined italics*中。过程名称出现在SMALL CAPS中。粗体页码位于章节词汇表内。