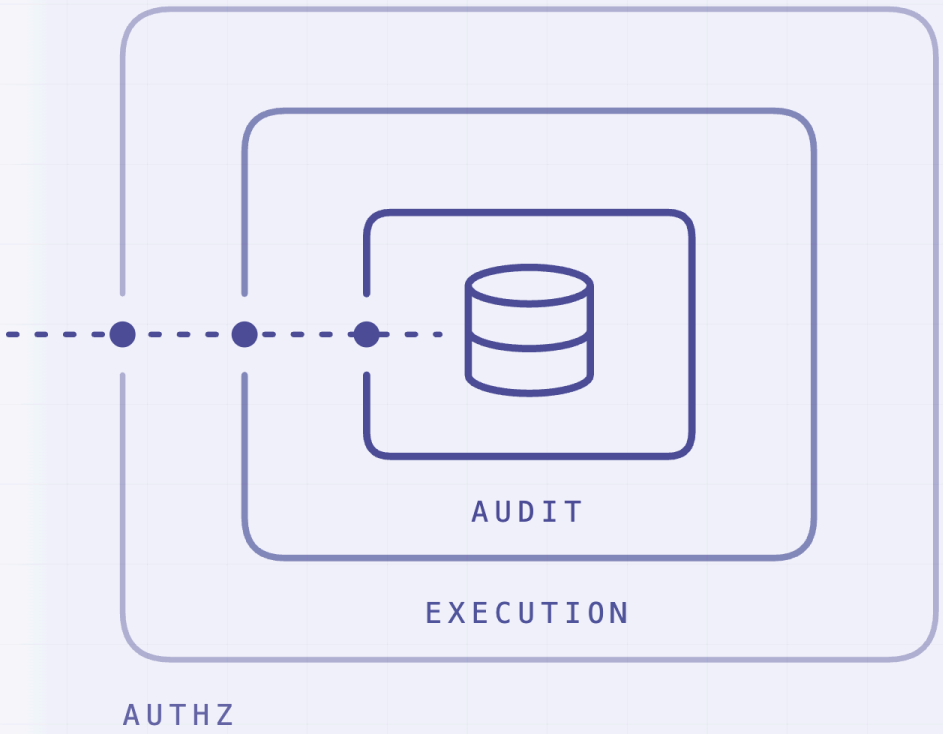
THE UNRELIABLE
SYSTEMS TRILOGY



生产数据库访问指南

Muhammet Şafak

生产数据库访问指南

身份、凭据、即时访问、受控执行与审计

Muhammet Şafak

© 2026 Muhammet Şafak
可免费下载和阅读。

目录

前言 5

第一部分 — 基础 **10**

第 1 章 — 无人设计的那扇门 11

第 2 章 — 生产数据的威胁模型 23

第 3 章 — 身份、角色与最小权限 37

第二部分 — 应用访问 **56**

第 4 章 — 凭据与机密 57

第 5 章 — 将连接作为策略 73

第 6 章 — 读取访问、副本与数据拷贝 88

第三部分 — 人员访问 **99**

第 7 章 — 访问路径：bastion、proxy 与 access gateway 100

第 8 章 — 即时访问与 break-glass 访问 117

第 9 章 — 受控执行：查询分析与危险操作 138

第 10 章 — 临时查询与数据修正 151

第 11 章 — 模式变更与迁移访问 170

第四部分 — 保护数据 **188**

第 12 章 — 敏感数据：分类、脱敏与最小化 189

第 13 章 — 备份、导出与其他旁门 207

第五部分 — 验证与运维 **225**

第 14 章 — 审计与问责 226

第 15 章 — 事故中的访问 242

第 16 章 — 访问检查清单与结语 257

附录 269

前言

设想周四晚上的 Ledgerly——一家并不存在的发票与支付公司。Robin 正在值班。一位客户的发票从下午起一直卡在“发送中”状态，修复看起来很小：一行数据，一个状态列。团队的桌面数据库客户端记住了共享的 admin 密码，连上主库，并自动提交，这类客户端常常如此。这次单行修复经由该客户端、以一个能修改每张表的角色执行，而它的 `WHERE` 子句匹配到的是一整批发票，而不是一张。

Robin 用晚上剩下的时间来修复。耗时更长的是随之而来的问题：这条语句是谁执行的？数据库日志显示的是共享的 admin 角色，每个服务以及——比方说——二十个人都在用它。密码管理器显示谁打开过该条目，而不是谁执行了什么。团队可以根据应用侧的副作用还原这次变更，最终也确实做到了。但在一天左右的时间里，没有人能有把握地说清到底发生了什么、涉及哪些行以及同一时间是否还有别的访问经由同一扇门。

这个故事里，没有哪一步单独看是不合理的。密码之所以共享，是因为共享是让第一个服务跑起来的最快办法。客户端之所以自动提交，是因为这是它的默认行为。角色之所以能修改每张表，是因为早期这比决定每个服务需要哪些表更省事。每一步在当天看来都合理。合在一起，却造出了一扇没人设计过的门。

本书要谈的，就是这扇门。

本书缘起

对生产数据库的访问往往被当作管道工程。第一个服务部署时把它搭起来，此后只有出了问题或有人离职时才会再碰。与此同时，数据库成了公司运行的最有价值的东西：客户、资金、历史。公司对它的依赖程度与其访问设计投入之少，二者之间的落差，正是本书的主题。

每个部分都已有大量著述。数据库文档讲解如何创建角色并做出授予。机密工具讲解如何签发凭据。安全指南讲解常驻访问为何是风险。更难找到的是工程师视角下的全貌：各部分如何拼合，每一部分给构建和运维系统的人带来什么成本，以及时间有限的团队应以什么顺序采用它们。

本书试图成为这样一幅全貌，并且从一个论断出发。生产数据库访问不是凭据分发问题，而是授权、执行与问责问题。

默认的看法把访问视为谁持有密码、密码如何存放、轮换和发放的问题。上面的周四晚上显示了这种看法在哪里止步。Robin 持有合法的凭据，凭据也恰好做了签发时要它做的事。失败的是围绕它的一切：没有任何机制判定这个人应当对那张表执行写入，没有任何机制在提交前检查这条语句或限定其影响，事后也没有任何机制把结果与某个人关联起来。授权决定什么可以发生、在何处发生。执行是每条语句到达数据的方式。问责是留存下来的证据。每一章都会点明它服务的支柱，第一部分为这三者建立词汇。本书把每项控制都视为有代价的决策。对支付 token 表适用的控制，对发票模板表可能毫无意义，重要时本书会明说。

有必要把立场说清楚。本书从防御方的角度写成。威胁只描述到足以说明某项控制为何存在为止；这里没有任何内容是闯入指南。本书不贩卖恐惧，也不引用数据泄露统计，因为冷静的设计评审比危言耸听的更有用。本书也如实计入摩擦。每项控制都要工程师和值班人员付出点什么，增加摩擦却不降低暴露的控制就是安全剧场。当我认为某种流行做法是剧场时，会直说并解释原因。

本书面向的读者

本书面向的读者是：构建与生产数据库通信的服务的**后端和平台工程师**，运维该数据库的 **SRE 和 DBA 角色**，必须决定访问应如何运作的**技术负责人**，以及在从未设立专职安全职能的团队中**新近承担安全职责的工程师**。凡是曾怀疑共享密码是否还是那个共享密码、在授予同事写入访问前犹豫过，或者被问到“谁能看到这张表？”而意识到要如实回答得花一天去翻查的人，都是我心目中的读者。

不需要安全背景，但应当熟悉以下几点：

- 关系型数据库：表、角色与授予、事务以及副本是什么。
- 通过网络连接数据库的服务，以及连接池和超时的相关词汇。
- 值班工作：被呼叫、在压力下修复问题，并在事后写下发生了什么。

熟悉某种具体的数据库引擎有助于理解示例，但并非必需，后文有一节会解释。

本书不是什么

本书不是数据库管理类的书。它不会教人调优查询规划器、设定缓冲区大小或选择复制拓扑。管理问题与访问相关之处，如副本和迁移，本书会涵盖那一部分，其余留给各数据库自己的文档。

本书也不是合规手册。GDPR、KVKK、PCI DSS、SOC 2 和 HIPAA 等监管制度之所以出现，是因为它们左右审计人员和客户的提问，本书也只在这个意义上描述它们：这类制度往往会问谁能看到什么、访问如何审批、事后留有什么记录。本书不解释任何制度，不引用其条文，也不告知读者被要求做什么。这类解释由法务和合规同事负责，本书的目标是给他们一个可以据以推理的架构。

本书同样不是攻击手册。书中不会有闯入的技术。某一章需要引入威胁时，只用一句话点明风险，例如凭据泄露、查询落到了错误的数据库上或供应商会话被遗留开启，然后转入控制措施。如果把威胁描述得更精确也不会改变设计，细节就略去。

本书的结构

本书共五个部分、十六章，后附一个附录。

第一部分，基础（第 1 章至第 3 章）论述上文的论点，并为其余各章建立词汇。第 1 章考察通向数据库的门是如何累积起来的。第 2 章为生产数据建立威胁模型，其中既包含误用，也包含失误。第 3 章讲身份、角色与最小权限，后面每一章都依赖它们。

第二部分，应用访问（第 4 章至第 6 章）讲服务。第 4 章讲凭据与机密，从静态密码到动态凭据和 workload identity。第 5 章将连接作为策略来处理：连接池化、上限与超时。第 6 章讲读取、副本以及生产环境悄悄产生的其他数据拷贝。

第三部分，人员访问（第 7 章至第 11 章）讲人。第 7 章比较 bastion、proxy 和 access gateway。第 8 章讲即时访问与 break-glass 访问。第 9 章讲执行本身：gateway 如何检查每条语句、限定其影响并提交。第 10 章讲临时查询与数据修正。第 11 章讲模式变更，以及谁有权做这些变更。

第四部分，保护数据（第 12 章和第 13 章）的关注点从谁能连接转向能看到什么。第 12 章讲分类、脱敏与最小化。第 13 章讲备份、导出以及绕过前面所建全部控制的其他旁门。

第五部分，验证与运维（第 14 章至第 16 章）讲团队如何得知其控制有效，并让它们持续有效。第 14 章讲审计日志与访问评审。第 15 章讲事故中的访问，此时压力最大，走捷径的诱惑也最大。第 16 章给出一份检查清单、一个带退出测试的成熟度阶梯以及关于文化的一些结语。

附录汇集了术语表、模式速查、危险操作矩阵、合规对照表以及访问评审和事故后评审的模板。

每一章的结构相同。开篇是一个简短、明示为虚构的场景。随后解释问题，以及显而易见的做法为何不够，介绍控制措施，并说明其成本和不适用的情形。接着列出常见错误，这些错误被当作设计中的陷阱，而不是个人过失，最后以五到七条要点和通往下一章的衔接收尾。统一的结构意味着既可以按顺序阅读各章，也可以跳到与本周问题相符的那一章，仍能在预期的位置找到成本一节。

认识 Ledgerly

Ledgerly 是一家虚构的企业对企业发票与支付服务。公司通过它开具发票、收款，并保存资金流转的账本。它运行五个服务：`billing-api`、`invoice-worker`、`payments-service`、`reporting-job` 和 `admin-console`。其主库有两个副本，另有一份供数据团队使用的分析副本。它存储客户联系方式、银行账户信息和支付 token，这正是它的访问设计举足轻重的原因。

围绕它的人员担任着常见的角色。有值班的后端工程师、支持团队、数据团队、运行数据库的小型平台组以及接收导出文件的外部会计供应商。

本书开始时，Ledgerly 有一个共享的 admin 密码，大部分后端团队拥有常驻访问，笔记本电脑上装着桌面客户端，也没有任何谁做了什么的记录。到本书结束时，它为每个服务设置了角色，使用短期凭据，为人员设置了 gateway，有经过评审的数据修正方式，语句在运行前经过检查与限定，有脱敏的副本，还有一份无法被悄悄改写的日志。它是一步一步走到这里的，每一章都会说明增加了哪一步，以及这一步给团队带来什么成本。

Ledgerly 是虚构的，发生在它身上的一切也是。每一章都以标明为虚构的场景开篇，其中的任何数字都是假设的。Robin、Sam、Jordan 和 Alex 等名字是团队中各角色的占位符。场景出错时，教训在于让它出错的设计，而绝不在于故事里的人。

如何阅读代码

本书中没有任何示例使用真实的编程语言。过程以朴素的伪代码书写，角色、授予和表使用一种类 SQL 的记法，不属于任何特定引擎。书中没有什么内容是用来粘贴进终端的。也不会出现某个具体 proxy、vault 或云产品的配置语法，因为语法会变，而其背后的模式不会。

图表使用 Mermaid。每幅图之前都有一句话，说明应当看什么，被拒绝或失败的路径都有标注。示例中绝不包含逼真的机密、账号或电子邮件地址；用 `<secret>` 和 `IBAN-PLACEHOLDER` 这类占位符代替。

为什么用英文，以及为什么与引擎无关

本书以英文写成，因为这一领域的大部分词汇，从 least privilege 到 break-glass，即使在以其他语言工作的团队中也以英文使用，而我希望这些术语保持易于辨认。本书与引擎无关，原因与此相关。团队可能运行 PostgreSQL、MySQL 或别的数据库，可能使用云身份服务、机密管理器或自制脚本。各种情形下模式都相同：把所有者与使用者分开，给每个 workload 单独的身份，让凭据过期，在人员前面放置感知身份的入口，在运行前检查将要运行的内容，让写入经由能够拒绝它的环节，记录发生的一切，并加以评审。产品及其语法常常变化。书中点名真实产品时，只是作为示例，并且对其行为的描述应当跨版本成立。

关于确定性的说明

我尽力说明哪些建议是明确的，哪些只是偏好。有些建议是明确的，因为判断错误的代价是客户数据暴露或账本丢失，属于这种情况时我会明说。多数建议是判断性的取舍，一个深思熟虑的团队可能依其规模、监管方、客户以及被呼叫的频率作出不同的权衡。我不知道的地方会说“我不知道”，并指出能够决定问题的变量。

如果不同意某项建议，我希望读者能够找到自己不同意的那个前提。所以每一章都写明其推理和成本。一个理解透彻、并按自身约束调整过的设计，比从书里照抄来的设计更有价值。

第 1 章从这扇门本身开始。

第一部分

基础

第 1 章 — 无人设计的那扇门

来看 Ledgerly 早年的情形。一位潜在客户发来一份安全问卷，其中一行看似例行公事：“列出所有能连接到生产数据库的人员和系统。”除其他职责外还负责照看数据库的 Alex，在一个安静的下午接手了这份问卷。

Alex 从密码管理器开始，里面有一个 admin 条目，整个后端团队都知道。接着是五个服务，各自保存着同一个密码，然后是笔记本电脑，桌面数据库客户端在那里记住了它。再往后是分析副本——每晚由未脱敏的转储恢复而成——以及共享存储里几份没人记得是谁创建的旧转储。Alex 查的每个地方，都指向另一个地方。

问卷的第二行是：“如何得知他们做了什么？”Alex 打开数据库日志，发现每一行都是同一个角色名，无论语句来自某个服务、某位修复数据行的工程师，还是一份本该读取副本的报表。对第一个问题的如实回答是一堆地方，对第二个问题的如实回答是：Ledgerly 并不知道。

Alex 不需要更好的密码。Alex 需要弄清访问由什么构成，以及一个简短、站得住脚的回答应当是什么样子。这就是本章的问题：对生产数据库的访问由什么构成，Ledgerly 要怎样才能不花一周翻查就回答这份问卷？

本章和第 2 章一样，描述 Ledgerly 的起始状态，且尚未加入任何目标状态的步骤。它的任务是为随后的各步骤提供词汇和地图。

问题及其难点

对生产数据库的访问很少是被设计出来的。它是累积出来的。第一个服务需要一条通路，于是有人创建了一个 admin 角色和密码。第二个服务因为第一个行得通而复用它。某位工程师要查看一行数据，于是密码进了桌面客户端。一个定时作业需要数据，拿到的连接字符串和其他所有人一样。这些步骤没有一步是关于访问的决策，而它们的总和，就是本章标题所说的那扇门。

来看 Alex 去清点时 Ledgerly 的门是什么样子。共享的 admin 密码存放在团队密码管理器里，不妨作为示例假设大约有二十个人知道它。它所解锁的 admin 角色拥有每一张表。五个服务全都用这个角色连接，密码相同，存放在各服务的配置中，部署流水线也用它运行迁移。工程师从笔记本电脑经公司 VPN 访问主库，客户端会记住密码，并且每条语句一运行就立即提交。访问是常驻的：没有人会失去它，包括一年前已调到其他团队的人。

副本接受同一个 admin 登录，而报表查询有时跑在主库上。分析副本是由未脱敏的转储恢复而成，转储文件还留着。一家外部会计供应商从不连接任何 Ledgerly 数据库，但它接收发票和账本数据的导出文件，而 `reporting-job` 的导出步骤用共享 admin 登录写出这些文件。数据库日志只显示一个角色。这种安排中没有任何东西记录是哪个人执行了哪条语句。

很容易把这当作纪律问题：叮嘱大家小心，更频繁地改密码。我不会从这里入手。只有人人都记得才有效的规则，在值班的人第一次疲惫时就会失效。系统赋予每个 principal 相同的能力和相同的名字，因此想要小心的人无从小心：没有更窄的角色可选，无法向数据库表明“只读”，也没有带名字的会话。

人们最先想到的补救办法也是如此，那就是把密码处理得更好：存入 vault、轮换、只发给提出申请的人。这是凭据分发，值得去做，但它只回答一个问题：谁持有机密。生产数据库访问不是凭据分发问题，而是授权、执行与问责问题。授权问的是每个 principal 可以做什么、经由哪条路径。执行问的是每条语句如何到达数据、是否有东西先检查它、它能波及多远。问责问的是证据是否把每条语句与某个人和某项申请关联起来。一个存放得完美的密码只解决谁持有机密，别的什么都解决不了：持有者仍可以执行一条没人检查过的语句，日志仍可能显示一个共享的名字。

这就是为什么这是一个工程问题，而不是政策问题。政策陈述应当发生什么，设计决定能够发生什么，以及发生时留下什么记录。本章余下部分具有设计问题的形态：访问由什么构成，我们希望它做什么，以及本书在哪里涵盖各个部分。

访问由五个部分构成

在评判任何设计之前，先对所设计的对象有一个精确的定义会有帮助。我发现可以这样说：对生产数据库的访问有五个部分：身份、授予、路径、执行和记录。下表说明每个部分回答哪个问题，以及属于本书论点的哪个

支柱。

部分	问题	支柱
身份	谁或什么在行动：一个人，还是一个 workload 实例？	问责
授予	该 principal 究竟可以做什么：角色、权限、上限？	授权
路径	连接如何到达：网络可达性、gateway、pooler？	授权（何处）
执行	语句如何到达数据？由谁或什么提交，运行前是否经过检查，能波及多远（事务、row guard、超时），以及如何提交。	执行
记录	留下什么证据，它是否把语句与人、申请和结果关联起来？	问责

三个支柱与各部分的对应如下。授权是授予加路径，按申请逐次决定。执行是每条语句所发生的事。问责是身份加记录。密码只触及其中第一个，而且只在门口。

身份是经过验证的声明，表明某个 principal 确实是它所自称的那个对象。**principal** 是任何能够完成认证的对象：人、服务、定时作业、供应商集成。当所述内容对人和 workload 同样成立时，就用 principal 一词，本书通篇这样使用。角色不是身份。**角色**是数据库内部一组具名的权限，可以被许多 principal 共享，共享的 admin 角色恰恰如此。

授予是 principal 进入之后被允许做的事。这里的授予，是指把某项权限给予某个角色、或把某个角色给予某个**登录**的行为及其产生的记录；第 3 章会扩展这一定义。**权限**是某一项具体的事，例如读取一张表，而登录是 principal 用来连接的数据库级账号。**路径**是连接到达数据库所走的路线：网络、proxy、gateway、笔记本电脑。

执行是语句到达数据的方式：由谁或什么提交，运行前是否有东西检查，什么限定其影响，以及如何提交。某项授予表示某个角色可以更新 `invoices`。它并不表示这次更新应当只触及一行，不表示有人事先看过，也不表示它运行在可以回滚的事务之内。在起始状态下，没有任何东西回答这些问题：桌面客户端发出键入的任何内容，并当场提交。第 9 章讲的就是这一部分。

记录是事后的证据，是对“谁在何时做了什么？”的回答，对于写入，还包括“涉及哪些行，出自哪项申请？”

每个部分都可能单独失败，但每个部分又都依赖其他部分才有用。没有明确的身份，精确的授予就没有可以依附的对象。没有选定的路径，身份就可能被一条从不询问身份的路线绕过。没有受控的执行，窄的授予仍然允许一条比任何人预期都更宽的语句。没有记录，就无从判断其余四个部分是否起了作用。这种依赖关系就是为什么只修复一个部分（通常是密码）会让问题的其余部分原封不动。

一项测试：日志能回答这个问题吗？

要找出缺了哪些部分，一个办法是向日志提出一个问题，看答案在哪里中断。下面的伪代码仅用起始状态所记录的内容，让一条语句依次通过五个部分。

```
function explainStatement(logLine):
    login = logLine.login
    // identity: does this login map to exactly one principal?
    principals = principalsUsing(login)
    if count(principals) != 1:
        return { answer: "unknown", stoppedAt: "identity" }
    // path: did the connection arrive by a route we chose?
    if not isKnownPath(logLine.source):
        return { answer: "unknown", stoppedAt: "path" }
    // grant: was this statement within what the principal was meant
to do?
    if not withinIntendedGrant(principals[0], logLine.statement):
        return { answer: principals[0], stoppedAt: "grant" }
    // execution: was a write reviewed and bounded before it
committed?
    if isWrite(logLine.statement):
        if not logLine.reviewedStatement or not
logLine.countShownBeforeCommit:
            return { answer: principals[0], stoppedAt: "execution" }
        if logLine.rowsAffected is unknown or logLine.rowsAffected >
logLine.expectedRows:
            return { answer: principals[0], stoppedAt: "execution" }
    // record: does the evidence tie the statement to the person, the
request and the result?
    if logLine.correlationId is empty or logLine.result is unknown:
        return { answer: principals[0], stoppedAt: "record" }
    return { answer: principals[0], stoppedAt: null }
```

在 Ledgerly 的起始状态下，`principalsUsing("admin")` 返回大约二十个人、五个服务和部署流水线，所以对每条语句，这个函数在第一项检查就停下。这不是日志的缺陷。日志如实记录了它所见到的东西，也就是一个登录。缺口在于把许多 principal 放在一个登录背后的设计。修复了身份，同一个函数接下来会停在执行检查上。开篇的事故正是它要处理的情形：一次单行修复，其 `WHERE` 子句匹配了整批数据，`rowsAffected` 远

高于 `expectedRows`，从未经过评审，也从未在 `COMMIT` 前展示行数，因此即使已知是谁、授予也允许，它仍停在“execution”处。通过该检查的语句，若没有任何东西把它与申请和结果关联起来，仍可能停在“record”处。

应用与人员，同一个问题

人们听到“数据库访问”，常常想到带着查询工具的工程师。这幅图景不完整，且不完整得很要紧。Ledgerly 主库的大部分流量来自软件：为 Web 应用提供服务的 `billing-api`，发送提醒的 `invoice-worker`，与支付提供方通信的 `payments-service`，夜间运行的 `reporting-job`，以及代表支持人员行事的 `admin-console`。软件比任何人连接得更频繁、更持久，受到的监督也更少。

服务和人员的故障方式不同，所以第二部分讲应用，第三部分讲人员。对比很具体：

	服务	人员
频率	持续不断，全天候	偶尔，集中爆发
监督	运行期间无	有人在盯着结果
凭据生命周期	短期，由 broker 签发（第 4 章）	一次会话或一个获批的时间窗口（第 7 章和第 8 章）
典型故障	机密泄露、连接池耗尽、部署指向了错误的数据库	一条比本意更宽的语句，在压力下手工执行

二者都建立在第一部分的基础之上。

还有一类容易被忽视的 principal：代表某个人行事的服务。`admin-console` 由支持人员使用，但在起始状态下，它和其他所有东西一样用共享 admin 登录连接，所以数据库看到的是这个控制台，而从来看不到点击的那个人。支持部门的 Sam 一次只查看一位客户，而这个设计让控制台以一个不属于任何人的名字触达每位客户的数据。第 3 章说明控制台如何随每个操作传递这个人的身份，第 14 章说明审计记录如何关联起来，使这个人在这一跳之后依然保留。

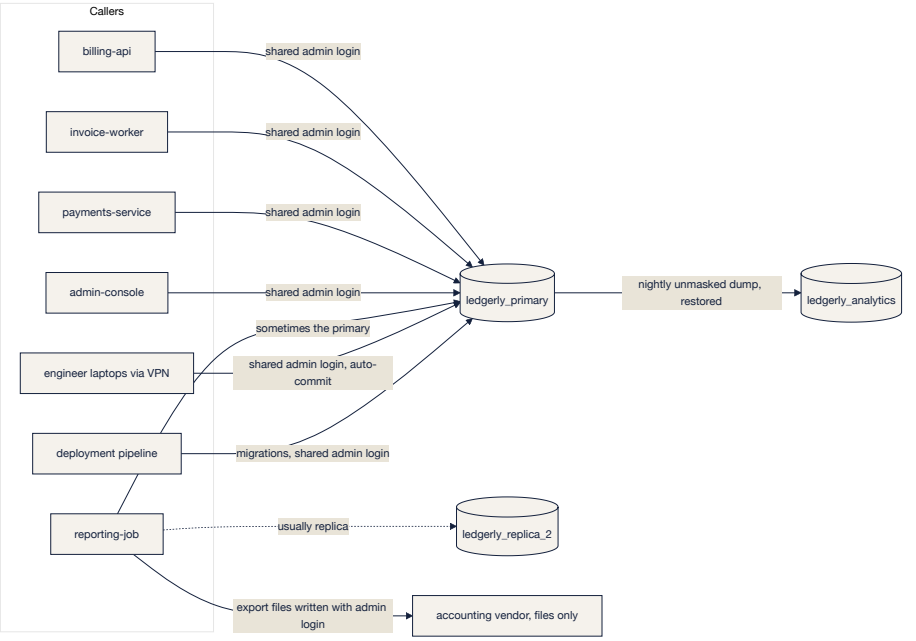
安全与可靠性是同一个问题

团队常常保留两份清单。一份关于安全：谁可能窃取数据，谁可能滥用访问。另一份关于运维：什么会让数据库宕机或损坏。它们通常由不同的人负责，在不同的时间评审。对于访问，我会把二者合并，因为问题是同一个。

来看三个听上去不相干的事件。一条 `WHERE` 子句错误或缺失的语句，更新了每一张发票而不是一张。笔记本电脑上的某个客户端让连接保持打开，整个晚上占着一个连接槽位。一份生产数据库的转储被复制到共享存储，被一个无权读取它的人读了。第一个是数据完整性问题，第二个是可用性问题，第三个是机密性问题。然而每一个对设计提出的问题都相同：为什么单个 principal 能做到如此宽泛的事，却没有更窄的路线、没有限制、没有记录？

这三个事件有共同的设计原因：授予、生命周期或路径都没有收窄，事后也没有任何东西可以展示。错误的 `WHERE` 子句主要是授予和执行的缺口，打开的连接是生命周期和上限的缺口，被复制的转储是路径的缺口。收窄授予、限制生命周期、选定路径、限定语句并保留记录，对这三个事件同时有益。这就是单一设计同时服务两份清单的原因。读不了 `payment_tokens` 的角色保护客户免于泄露，写不了 `ledger_entries` 的角色保护账本免于一次糟糕的批量更新。两者都源自同一个习惯：窄授予。

下图展示 Ledgerly 的起始状态。请留意有多少箭头终止在同一个节点，以及唯一一条从该节点通向分析副本的箭头。



有两点很突出。第一，通向主库的每条路径都带着同一个名字，所以数据库分不清是 worker 还是人。第二，数据还经由一条旁路离开：每晚的恢复把所有内容复制出去，包括没有任何分析人员需要的列，复制到一个有着自己受众的存储中。

进入的各种方式

在共享密码与完全受控的设计之间有一系列模式，后面各章会把每一种放在这个序列上。下图从对每条语句控制最少，排到控制最多。请留意每一步增加了什么，以及第一个语句级检查出现在哪里。



向右的每一步都增加一项属性，也要付出一点代价。bastion 收窄了路径，但对岸的会话仍然无人监督。session proxy 把连接与某个人绑定并记录下来，但它交出的是一个会话，授予所允许的任何内容都可以在里面键入。受控执行的 gateway 在每条语句运行前查看它，并可以拒绝它。产品路径则彻底取消会话：人请求一个已定义的操作，例如“重新发送这张发票”，从不书写 SQL。这并不意味着最右边的模式始终最好；它同样是需要构建的最多、改动最慢的一个。第 7 章在具体路径上比较前四种，第 9 章讲语句级检查，最后一章按支柱为各模式评分。

好的访问是什么样子

在选择任何工具之前，我会先把目标写下来。我用五个属性，每个都有代价。

最小权限是指只授予任务所需的内容，且只在所需的时间内授予。代价是弄清每项任务需要什么所花的功夫，以及任务被发现需要更多时的摩擦。采用它却没有快速申请更多权限的途径的团队，会看到人们绕开它。

短生命周期意味着访问会自行结束。持续一小时的授予不需要任何人记得它。与之相对的是**常驻访问**，不论用不用都持续存在。常驻授予是一个承诺：没有人需要再次申请。第一个月它很方便，此后每个月都代价高昂，因为没人记得这个承诺涵盖了什么。短生命周期的代价是更多的申请，以及到期可能在修复进行到一半时到来。

可观测性意味着对访问的每次使用都留下一条点名 principal 的记录，且记录所在之处是该 principal 无法更改的。代价是存储，更重要的是注意力：没有人读的日志是安全剧场，因为它耗费存储却不提供保护。第 14 章会回到这一点。

可逆性有两层含义，二者都要紧。访问应当易于撤销，而用访问犯下的错误应当可以撤回，或至少可以遏制。撤销访问必须成为例行，而撤回错误需要经过评审的修正和能够恢复的备份。一种廉价的可逆性是在显式事务中执行写入，并在 `COMMIT` 前展示行数：本意只触及一行却报告了比如四百行的更新，会在仍可回滚时被拦下。第 9 章讲 gateway 如何将其作为默认。代价是花在演练上的时间。

低摩擦是团队忘记写下的那一个。如果正当的路径慢，人们会找到不正当的路径，而且他们这样做没有错，因为凌晨 2 点的事故不会等人。低摩擦的代价是它约束了其余四个：每项控制都必须足够廉价，才能在一周中最糟糕的那个小时里使用。

这五个相互拉扯。短生命周期增加摩擦。可观测性增加一跳：被记录的会话必须经过某个负责记录的东西，而这个东西可能很慢，也可能宕掉。最小权限使角色成倍增加。我认为不存在能让五者同时最大化的设置。有用的问题不是“谁应当拥有访问？”，而是“哪些语句可以运行，经由什么，留下什么记录？”后面每一章回答其中一部分，并说明答案的代价。

第一份清单

在采用任何控制之前，我会先写下现有的东西。下面的草稿展示 Ledgerly 起始状态下清单的形态。要点是每扇门一行，附带具名的负责人和理由，使没有负责人的行作为缺口显现出来。

```
door:        desktop client on engineer laptops
principals:  about twenty people (illustrative)
login:       shared admin
reaches:     ledgerly_primary, ledgerly_replica_1, ledgerly_replica_2
network source: company VPN, any laptop
grants:      owns every table
can write?:  yes, auto-commit, no row limit
record:      database log shows the shared role only
owner:       nobody
reason:      "needed to fix rows during incidents"
ends:        never
```

每扇门填写一个块：每个服务、定时作业、部署流水线、笔记本电脑、分析副本的恢复、共享存储里的转储以及导出文件。多数团队会发现 `owner` 和 `ends` 两行是被留空的。这样的清单不是项目计划。它是把“谁能触达主库？”从一天的翻查变成一份可以直接阅读的文档的东西。

本书地图

后面各章依次处理访问的每个部分和上述每种故障。下表把本章提出的每个问题对应到本书回答它的位置。

问题	支柱	回答所在位置
我们要保护数据免受谁和什么的侵害，面对哪几类事件？	全部三个	第 2 章
如何给每个 principal 独立的身份和能用的最窄角色？	授权	第 3 章
服务如何获得凭据，而不必在文件中放一个共享密码？	问责（身份）	第 4 章
连接数上限和超时如何成为保护数据库的策略？	授权与执行（限定）	第 5 章
读取如何到达副本，其他数据拷贝会怎样？	授权	第 6 章
人员经由哪些路径到达数据库，每条路径上有什么？	授权（路径）、问责	第 7 章
如何为某项任务获得写入访问，又如何结束？	授权	第 8 章

问题	支柱	回答所在位置
语句在运行前如何检查，又如何提交？	执行	第 9 章
临时查询和数据修正如何保持可评审？	执行	第 10 章
谁来做模式变更，凭什么访问？	执行	第 11 章
哪些数据需要额外保护，如何脱敏或不予提供？	授权	第 12 章
备份、导出和其他绕过控制的路线又如何？	授权（路径）	第 13 章
如何保存经得起检验的记录并加以评审？	问责	第 14 章
事故期间有什么变化？	全部三个，在压力之下	第 15 章
首先做什么，又如何持续做下去？	全部三个	第 16 章

本书有五个部分：基础、应用访问、人员访问、保护数据及其拷贝以及验证并运维这一切。每一章在开头都点明它服务的支柱。不必按顺序阅读。如果眼下的问题是五个配置文件里的共享密码，第 4 章是起点，但第 3 章能避免不经意间又创建出下一个共享角色。

权衡与成本

第一项成本是**小团队的时间和注意力**。有人得建立清单、定义角色、设置凭据并开展评审。在小团队里，这往往是某个人的兼职工作，要与产品工作争时间。如果这项工作始终排不上日程，设计就停留在纸面上。我宁愿看到一个小而完成的第一步，也不要一个庞大的计划。

第二项成本是**值班人员的摩擦**。每项控制都在疲惫的人和修复之间加了一步。窄角色意味着第一次尝试有时会在最糟糕的时刻因权限错误而失败。短期访问意味着再次申请。应当事先确定快速路径是什么，并加以演练。每次事故都被绕过的控制不是控制，它只是绕过之前的一次延迟。

第三项成本是**必须在需要时可用的新系统**。gateway、机密存储和 log store，每一个都是另一个可能宕掉的东西。采用之前，要问在那次停机期间需要数据库的人会怎样，并备好答案。

也有完整设计不是正确起点的情形。一个没有客户数据、由两个人运行的单服务原型，不需要 gateway 或评审日程。但它确实需要一个与任何人的登录分开的、自己的登录，因为这很廉价；还需要一条备注，写明什么情形会让团队重新审视这个问题：第一个客户、第三位工程师或第一个外

部集成。我不知道某家公司的合适时机在何时。这要视情况而定，而所取决的是：数据有多敏感，有多少 principal 接触它，以及一条糟糕的语句多快会造成无法撤回的损害。

最后，好访问的这些属性是有限度的。它们缩小影响范围，让行为可见。它们不会让数据库变得安全，本书没有哪一章声称过相反的话。**影响范围**是指当某个 principal、凭据或操作出错时，可能受到影响的程度，本书的工作是让这个范围更小、更可见，而不是让它归零。

常见错误

这些是设计陷阱，不是个人过失。每一个都有诱人之处，每一个都有不依赖任何人更加小心的补救办法。

- **把改密码当作补救办法**。新密码让人觉得有进展，也使一位前同事不再知道密码。但如果仍有，比如说，二十个人和五个服务共享它，设计就没有变。正确的做法是一种没有人需要共享密码的设计，这正是第 3 章和第 4 章所构建的。
- **修了授予却忘了路径**。窄角色只有在连接必须使用它们时才有用。如果笔记本电脑仍能用旧 admin 登录触达主库，窄角色就是可选的。要盘点每条路径，而不只是每个角色。
- **漏掉应用**。“谁有数据库访问权”的评审常常只列出人，因为人在组织架构图上。服务持有能力更强的凭据，并且用得更多。要纳入每个 workload、每个定时作业和每个脚本。
- **让安全路径比不安全路径更慢**。如果读取生产数据需要一张工单和一天时间，而共享密码只要，比如说，十秒，共享密码每次都会赢。目标是让安全路径不比它慢。

> **警告**：共享存储中未脱敏的转储，是起始状态里通向客户数据最直接的路线，任何能读取该文件夹的人都可以访问。应当把查找并删除这些文件作为第一份清单的一部分，放在对角色做任何重新设计之前。

要点

- 生产数据库访问不是凭据分发问题，而是授权、执行与问责问题，密码只回答第一个问题，而且只在门口。访问是由五个部分构成的设

计，即身份、授予、路径、执行和记录，任何一个部分的薄弱都会削弱其他部分。

- 共享 admin 账号不是任何人做出的决定；它是累积出来的，补救办法是一种没有人需要共享密码的设计。
- 应用比人连接得更频繁、受到的监督更少，所以对访问的任何处理，都必须把服务、作业和工具与人一起作为 principal 来涵盖。
- 对访问而言，安全与运维可靠性是同一个问题，因为错误的 `WHERE` 子句、泄露的连接和被复制的转储有共同的设计原因：授予、生命周期或路径都没有收窄，也没有记录。
- 好的访问有五个属性，即最小权限、短生命周期、可观测性、可逆性和低摩擦，它们相互拉扯，所以每项控制都应连同其成本一起呈现。
- 第一步是一份每扇门一行的清单，涵盖服务、笔记本电脑和数据拷贝，缺失的负责人或缺失的结束日期会成为可见的缺口。
- 控制措施缩小影响范围并使行为可见，但不会让数据库变得安全，诚实的目标是更小的范围和更清晰的视野。

下一章从访问的形态转向保护它的理由。第 2 章追问这些控制是为谁、为什么而设，把可能出错的事件分为三类，并为 Ledgerly 写出一份威胁模型，供本书其余部分回指。

第 2 章 — 生产数据的威胁模型

设想 Ledgerly 的安全联系人在季度评审前向平台团队发来一个简短的问题：“我们要保护生产数据库免受谁的侵害？”负责数据库的 Alex 一分钟内就给出了回答：主要是外部人员。随后 Alex 停了下来，因为这个回答与团队过去一年的工单对不上。一次支持请求涉及了错误的客户。一条报表查询让发票写入慢了一个下午。一份转储被恢复到笔记本电脑上，只为调试一天，却没有人提醒删除。一份供应商导出比预期多带了一列。

这些事件都与外部人员无关，也都没有恶意。每一起都是一扇门向某个有合理任务的人敞开，而其角色允许的操作超出了任务所需。Alex 的第一个回答听起来像安全问题；第二份清单才描述了这个系统。

本章如第 1 章那样描述处于初始状态的 Ledgerly，自身不添加任何控制措施。它添加的是一份书面的威胁模型，本书后面的每项控制措施都以它为参照来选择。它围绕全部三根支柱来界定威胁：principal 被允许做什么，每条语句如何到达数据，以及留下什么证据。它回答的正是 Alex 一分钟内答不上来的那个问题：谁和什么能够损害数据，通过哪些门，损害会扩散多远？

问题及其难点

没有威胁模型的访问设计，往往顺着声音最大的那个担忧走。如果担忧的是外部攻击者，团队就会建一道边界防线：VPN、防火墙规则、网络允许列表。这些都重要。但边界防线回答的是“谁可以接近数据库？”，对于人员或服务进入之后能做什么，它什么也没说。在 Ledgerly 的初始状态下，凡是在 VPN 上并持有共享密码的人，都能做管理员角色能做的一切，也就是一切操作。

显而易见的替代做法，是照着检查清单上的控制措施依次落实。检查清单并不知道哪些门的风险最高。它会要求轮换一个不停机就无法轮换的密码，却对每晚的恢复只字不提，而那次恢复会把每一个 Restricted 列复制到受众广得多的存储里。不依据模型选出的控制措施，是凭是否现成可

用和是否流行选出的。有些确实降低了暴露；有些是安全剧场，即带来摩擦却没有相应降低风险，而团队分不清哪些是哪些，因为从来没有写下这项控制措施是为了什么。

需要一份书面模型，有三个原因。第一，门是技术对象：一个登录、一条网络路径、共享存储中的一个文件、一个流水线变量。策略可以说“访问必须恰当”，但一个登录能触达什么，只能由设计来决定。第二，最大的风险来自普通选择之间的相互作用，正如前言所述：一个共享密码、一个自动提交的客户端，加上一个拥有每张表的角色。单看任何一个选择都不显得错，所以逐个审查选择是发现不了问题的。第三，模型必须便宜到足以维护，否则写完一次就会任其偏离实际。

本书所说的威胁模型很小。它列出谁和什么能对数据采取行动、通过哪条路径、可能出什么问题、影响扩散多远以及由哪项控制措施应对。它不是风险评分，也不是预测。它是对每一扇门提出同一个问题的方式，使答案可以相互比较。

> **注意**：本书描述威胁，只到足以说明某项控制措施合理为止。本章指出可能出什么问题，例如凭据泄露，或查询落到了错误的数据库上，而不描述任何人如何促成这些问题。如果更多细节不会改变设计，就略去不写。

谁与什么能触达数据库

第 1 章指出，对生产数据库的访问有五个部分：身份、授予、路径、执行和记录。威胁模型首先列出各个 principal，并对每一个提出这五个问题。Ledgerly 有九类值得列出的参与方。其中两类，即供应商和外部攻击者，严格来说不是 principal，这正是它们应当出现在清单上的原因：它们很容易被漏掉。

服务

这些服务（`billing-api`、`invoice-worker`、`payments-service` 和 `admin-console`，其中 `reporting-job` 接下来作为作业处理）是数据库最主要的使用者。它们无人值守地运行，把凭据保存在配置中，其行为的正确程度仅取决于代码。服务即使没有任何人做错事，也可能出问题：一个 retry 循环不断打开连接直到触及上限，一条查询因为索引被删除而扫描整张表，一次部署把 staging 构建指向了主库。

在初始状态下，每个服务都使用同一个管理员登录连接，因此每个服务的影响范围都是整个数据库，尽管 `billing-api` 和 `invoice-worker` 各自只需要少数几张表。最小的服务中的一个缺陷，就拥有最大的服务的触达范围。

批处理作业

`reporting-job` 值得单独一行，因为作业按计划运行，常在夜间，无人看守。它往往读取大量数据、持有长事务并生成文件。在初始状态下，它有时读取主库，其导出步骤使用共享管理员登录写出供应商文件。它的正常行为本来就像批量访问，所以它的异常行为比人的异常行为更难发现。

值班工程师

值班轮换承受着最大的压力，要求拥有宽泛的访问权限，这有充分的理由：夜间收到呼叫时，不是等待审批的时候，因为没有人醒着来批。在初始状态下，他们通过会记住密码、并对每条语句都提交的桌面客户端使用共享管理员密码。急于修复的紧迫感，加上更改任何东西都没有阻力，使这个群体成为最容易发生意外的群体，也是设计最能保护的群体。

支持人员

支持人员使用 `admin-console` 一次查看一位客户。这里的威胁在于范围，而不是恶意。一个能显示每位客户的工具，背后是一个能读取每一列的角色，于是用一个可以展示所有发票的系统去回答只涉及一张发票的问题。设计上的问题是：当只需要窄的能力时，如何让宽的能力不可用。

数据团队

数据团队在 `ledgerly_analytics` 上工作，该库在初始状态下是每晚对未脱敏转储的一次恢复：每一个 Restricted 和 Confidential 列都在其中，而访问者是工作内容就是批量查看数据的群体。他们的访问是正当的。问题在于，这份拷贝的受众比主库更广，处理更松散，日志更少，所以主库上的控制措施保护不了这份拷贝中的数据。

供应商与集成

外部会计供应商从不连接 Ledgerly 的任何数据库。它接收文件。这改变了威胁的形态：暴露的不是登录，而是一次导出，其影响范围取决于导出包含哪些列，以及这份拷贝在供应商一侧存在多久。相关的问题是：文件里有什么，之后由谁负责，以及文件多出一列时是否有人会发现。

CI/CD

部署流水线同样是一个 principal，尽管它是基础设施，而不是人或服务。在初始状态下，它使用同一个管理员登录运行迁移，因为管理员角色拥有每张表，别的角色都改不了这些表，而该凭据就保存在流水线的配置中。一条既能修改任何表、又持有常驻凭据的流水线，同时是失陷风险和错误风险，因为一次有问题的迁移会以所有者的全部权限运行。

平台与 DBA 人员

规模不大的平台小组运行数据库，它是唯一能够改变控制措施本身的参与方：角色、授予、人员经过的入口、日志。在初始状态下，它和所有人一样使用共享管理员登录，所以没有任何东西把它的工作与其他任何人的工作区分开。在目标设计中，它持有一个用于管理访问的角色 `ledgerly_access_admin`，由三个 principal 通过各自的登录使用。这并没有消除威胁，只是转移了威胁。持有该角色的人一次错误的授予，或关闭一项日志设置，都会改变模型中其他每一行的含义。

因此，模型把**控制平面**本身列为一项资产：access gateway、凭据 broker、`ledgerly_access_admin` 和审计存储。数据由控制措施保护，而控制措施由一个小得多的人员和系统集合来保护。这种暴露体量小，影响大，需要问的问题与其他任何一扇门相同：谁能触达它，它能改变什么，谁会注意到。

外部攻击者

外部攻击者在获得凭据之前不是 principal，所以有用的建模方式是看他们需要获得什么，而不是看他们可能做什么。防御方的问题是：哪些门接受可能泄露的凭据，它能触达什么，我们如何发现？答案来自其他八类参与方。泄露的凭据会继承其原属者的影响范围。

三类威胁

同样的参与方可以通过三种不同的方式造成损害，每一种所需的控制措施不同。把几类混在一起，会导致控制措施错配：针对滥用而设计的意图审批，对意外几乎没有作用，因为审批人认可任务是正当的，而语句依然出了错。

错误

错误是本意正确却没做对的行为：对错误的环境执行的语句，匹配的行数超出预期的条件，对主库而言过重的查询，因配置过期而指向了生产的脚本。行为者是正在做本职工作的 principal。

我预计在大多数团队中，这一类最为常见，原因是结构性的，而不是统计上的。滥用需要一个有动机的人，失陷需要一个有机会的攻击者，而错误只需要一个有任务的人。我手里没有错误究竟多出多少倍的数字，即使有人给出，我也不会相信。无论如何，对设计的结论都成立：针对错误的控制措施，保护的是每个团队都有的那一类，不论它是否曾遇到另外两类。

应对错误，最好的办法是缩小单条语句能触达的范围，因为不能要求人永远小心谨慎。凡是此类的有：无法触及不需要的表的角色（第 3 章），能终止失控查询的超时（第 5 章），在每条语句运行之前对它的分析（第 9 章），带有四眼评审的、独立的写入路由（第 8 章和第 10 章），以及事后经过评审的数据修复方式（第 10 章）。

四眼步骤只在一个条件下对此有帮助：评审人看到的是语句及其预期效果。批准意图（“修复发票 1042”）是针对滥用类别的控制措施，对错误类别毫无触动。批准语句以及它应当修改的行数，才是在应对错误。

滥用与好奇

这里的滥用，是指拥有合法访问权限的 principal 把它用于任务之外的事情：出于兴趣查看某位客户的记录，导出超出工作所需的数据，查看同事的数据。它包括单纯的好奇，这大概是最常见的形式，也最难称之为不当行为。它还包括少见的蓄意情形。

设计上的应对并不是怀疑团队。团队无需假定有恶意，也能注意到：常驻授予使浏览成为可能，并且不留痕迹。真正起作用的控制措施，把访问与任务绑定并使其可见：不提供 Restricted 列的只读角色，对 Confidential 数据的脱敏视图（第 12 章），限时授予（第 8 章），以及评审人真正会读的审计日志（第 14 章）。没有证据，“这次访问是否恰当？”对任何人都无法回答，包括没有做错任何事的人。

失陷

失陷是指凭据或机器落入了并非预定 principal 的人手中：通过配置文件泄露的密码，丢失的笔记本电脑，被不该信任的对象读取的流水线变量。防御方不需要知道它是怎么发生的，就能针对“它可能发生”这一事实来

设计。设计控制的是：该凭据能做什么，持续多久，以及是否有人会知道。

凭据的属性在失陷时最为关键。共享、静态且不受限的凭据，会把丢失一台笔记本电脑变成丢失每一张表。限定在一个角色、短期有效并只签发给一个会话的凭据，则把同一事件变成有界且可见的事件。第 4 章和第 7 章讲凭据如何签发；第 3 章讲凭据应当携带得多么少；第 14 章讲如何发现。

> **注意**：对一个事件进行分类，往往需要判断。一条只改一行却波及整批数据的修复，是错误。同一条语句如果由没有任务的人有意运行，就是滥用；如果用从笔记本电脑上取得的凭据运行，就是失陷。数据库每次看到的都是同一条语句，这正是日志要写明登录、控制措施不依赖猜测意图的理由。

需要保护什么

数据有三个属性值得保护，模型应当说明每个威胁触及哪一个。它们就是人们熟悉的机密性、完整性和可用性，每一项在 Ledgerly 中都有特定含义。

机密性关乎谁能读取什么。对 Ledgerly 而言，分量并不均等。`iban`、`account_holder_name` 和 `token_value` 是 Restricted，因此一旦暴露就难以挽回。`contact_name`、`contact_email`、`contact_phone` 和 `billing_address` 是 Confidential 的客户 PII（个人身份信息）。`invoices` 和 `payments` 的金额是 Internal。触及 Internal 数据的威胁，值得投入的设计精力比触及 Restricted 数据的威胁要少，模型应当显示出这种差异，而不是把所有列一视同仁。此类数据保护法规通常会问：持有哪些个人数据，谁能看到，这正是同一种差异；对其的解释由法务和合规同事负责。

完整性关乎数据是否保持正确。对大多数表来说，错误可以通过之后的写入来纠正，但 `ledger_entries` 和 `audit_events` 是 append-only 的，而对账本来说，完整性比保密更重要：错误的条目通过 `payments-service` 以一条新的更正条目来修复，绝不通过编辑。因此，对这两张表的完整性威胁在性质上不同。这也是证据最为关键的地方，因为能被悄悄修改的账本，是没有人能依赖的账本。

可用性关乎数据库是否为依赖它的服务提供服务。它是安全讨论中最常被遗忘的属性，也是意外占主导的属性。长锁、在主库上扫描大表的查询，或来自 retry 循环的连接洪流，都可能像攻击一样彻底地让开票和收款停摆。可用性在模型中占有专门的行，与连接策略（第 5 章）以及允许读取去往哪里（第 6 章）相关联。

在这三项之外还有第四项属性：**问责**，即记录能够说明哪个 principal 做了什么。这是访问中的记录部分。在初始状态下它是缺失的，因为日志只显示共享角色，而这种缺失抬高了其他每一种威胁的成本。

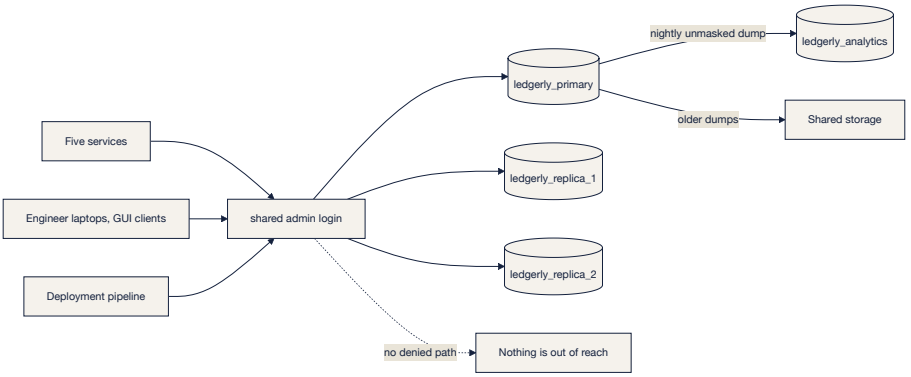
影响范围

模型还需要一个工具来比较威胁：一种说明单次故障扩散多远的方式。**影响范围**是指，当一个 principal、凭据或操作出错时，可能受到影响的范围有多大。这个术语只有在针对某个具体对象时才有用，所以问题始终是“这个登录的影响范围是什么？”，而绝不是“我们的影响范围是什么？”

影响范围有七个维度，分别考察每一个会有帮助：

- **数据触达范围**：principal 能触及哪些表、列和行。
- **操作触达范围**：它能读取、写入、删除还是修改 schema。
- **环境触达范围**：它能连接哪些数据库：主库、副本、分析拷贝。
- **时间触达范围**：访问保持有效多久，从一个会话到永远。
- **拷贝触达范围**：它能读取的数据还被复制到了别处的哪些地方，以及谁能触达这些拷贝。
- **语句触达范围**：单条语句在任何机制介入之前能改变多少：它触及的行数，它是否在可回滚的事务中运行，以及是否有超时加以限制。
- **网络来源**：哪些网络和机器完全可以出示该凭据。

在 Ledgerly 的初始状态下，共享管理员登录在每个维度上都达到了最大。它触达所有表和所有列，可以读取、写入、删除和修改，连接主库和两个副本，没有到期时间，其数据在主库之外已有拷贝。任何语句都不受限制并立即提交，VPN 上的任何一台笔记本电脑都可以出示这份凭据。图中展示了需要关注的地方：左侧的一切都持有同一份凭据，因此右侧的一切都处在同一个影响范围之内。



可与目标状态的角色 `app_reporting_job` 对比：只读，位于副本上，无法访问 Restricted 列。在初始状态下可能暴露 Restricted 数据的缺陷，对该角色而言只是副本上的一次重查询。缺陷是相同的；影响范围变了。

为了让比较保持诚实，应当按登录写下影响范围，写成一份问题清单而不是一个分数，并把两个登录并排放置。下面的草图仅作示意，是评审的辅助手段，不是本书所依赖的工具。未知的答案记录为未知，因为把未知向下舍入为“没问题”的模型，会掩盖它本应发现的那些情形。

```
dimension          shared admin login          app_reporting_job
data reach         every table                 unknown until Chapter 3
lists the tables
restricted cols    all                         none
action reach       read, write, delete, DDL    read
environment        primary, both replicas      one replica
time reach         no expiry                   unknown until Chapter 4
decides
copy reach         unmasked restore, dumps     the export files it
writes
statement reach    unbounded, commits at once  read-only transaction,
statement timeout
network origin     company VPN, any laptop     unknown until Chapter 7
decides
```

这种比较之所以有用，正是因为它并不均匀。第二列写着“unknown”的地方，说明模型发现了设计尚未回答的问题，这是一个发现项，而不是表中的空缺。

有两点限制。影响范围描述的是一个登录能够做什么，而不是它将会做什么，所以它是规划上的界限，而不是预测。而且它没有说明每种威胁发生的可能性有多大。一扇整天敞开的门，即使影响范围窄，也可能比一扇每

季度只开一次的门、即使影响范围宽，更值得关注，这就是模型必须把影响范围与暴露结合起来的原因。

Ledgerly 威胁模型

有了参与方、类别、属性和影响范围，模型就可以放进一张表。下表涵盖 Ledgerly 的初始状态。每一行是一个参与方和一条路径，列出该组合最主要的威胁类别、对上述属性的影响、提供控制措施的章节以及**残余风险**：这些控制措施到位之后仍然成立的情况。“暴露程度”是对该门使用频率和被滥用难易程度的定性判断（高、中或低），之所以写下来，是为了让人可以提出异议。

参与方	路径	主要类别	影响	暴露程度	控制措施所在章节	残余风险
服务（全部五个）	来自各服务配置的共享管理员登录	错误、失陷	任何表、所有列；连接耗尽时影响可用性	高	第 3 章、第 4 章、第 5 章	缺陷仍然拥有其自身角色的触达范围
`reporting-job`	部分运行读取主库	错误	重读取与写入争用；主库的可用性	中	第 5 章、第 6 章	副本上的重读取仍会拖慢该副本上的其他读取方
值班工程师	笔记本电脑、GUI 客户端、共享管理员密码、自动提交	错误、失陷	范围不受限的写入；任何表的完整性	高	第 7 章、第 8 章、第 10 章	在已授予的角色内手工输入的语句仍可能出错
支持人员	`admin-console` 以共享登录行事	滥用、错误	浏览 Confidential 客户数据；改错客户	中	第 3 章、第 10 章、第 12 章	支持人员仍能看到某位客户的案例所需的列
数据团队	未脱敏的分析恢复库和较早的转储	滥用、失陷	每个 Restricted 和 Confidential 列的机密性	高	第 12 章、第 13 章	脱敏后的数据仍可能通过推断泄露取值
会计供应商	导出的文件	错误、滥用	导出列的机密性及其存续期	中	第 12 章、第 13 章	供应商在交付后仍保留文件
部署流水线	作为流水线变量保存的管理员登录	错误、失陷	schema 与数据的完整性；所有表	中	第 4 章、第 11 章	经过评审的迁移仍以所有者权限运行

参与方	路径	主要类别	影响	暴露程度	控制措施所在章节	残余风险
平台与 DBA 人员	现为共享管理员登录；目标设计中为 `ledgerly_access_admin`	错误、滥用、失陷	授予、控制措施和证据可被更改	中	第 3 章、第 7 章、第 14 章	该角色存在，因此对它的滥用是可能的；只剩下检测
控制平面（资产）	gateway、凭据 broker、审计存储	错误、失陷	其他每一行都会改变含义	中	第 4 章、第 7 章、第 14 章	一个组件失陷，会削弱它所保存的证据
任何人员会话	在已授予的角色内手工输入的语句	错误	不受限的语句，立即提交	高	第 9 章、第 10 章	一条语句可以有效、已获批准，但效果仍然是错的
任何网络来源	公司 VPN 直接连通主库	失陷	泄露的凭据可在 VPN 上的任何笔记本电脑上使用	中	第 7 章	从允许的来源窃取的凭据仍然可在该处使用
外部攻击者	以上各行中任何泄露的凭据	失陷	继承该凭据所有者的影响范围	低到中	第 4 章、第 7 章、第 14 章	在有效期内被窃取的凭据，在到期前一直可用
每一个参与方	数据库日志中没有具名登录	所有类别	其他每一行的问责	高	第 7 章、第 14 章	记录的可信程度，仅取决于保存它的存储

这张表带来四点观察，它们正是要构建这张表的原因。

第一，共享管理员登录出现在大多数行中。这张表显示的是一个问题的多种面貌，而第一项控制措施，即为每个 principal 分配单独的角色（第 3 章），一次就解决了其中很大一部分。能揭示工作顺序的模型，就是在发挥它的作用。

第二，暴露程度最高的门中，有些从来就没打算成为门：分析恢复库和较早的转储。在任何人脑中的数据库示意图里，它们都不是访问路径。这就是第 6 章和第 13 章把每一份数据拷贝都当作有负责人和到期时间的门来对待的原因。

第三，残余风险这一列是表中最诚实的部分。每项控制措施都会留下些什么：已授予的角色仍可能被滥用，有效的语句仍可能是错的，记录的可信程度只能到其存储所能达到的程度。没有写下残余风险的行，就是没有人追问过的行。

第四，控制平面、人员会话和网络来源这几行指出的是资产或路径，而不是参与方，最后一行也没有指向任何单一的参与方。缺失问责是整个系统的属性，它抬高了其他每一行的成本，因为没有具名登录，任何事故都无法界定范围。第 14 章补上这一行。前面的章节构建身份，使补上它成为可能。

> **警告**：一份完成的威胁模型，就是一幅 Ledgerly 最薄弱之处的地图。应当把它放在访问限制与其所描述的系统相同的位置，不要粘贴到公开工单或受众广泛的 wiki 中。一份列出未脱敏拷贝和共享凭据的文档，对维护它的人有用，对其他任何人都没有用。

保持模型有效

只写一次的威胁模型就是一张快照，上面的表在 Ledgerly 新增一个集成、或数据团队得到一份新拷贝的当天就过时了。我会把它当作有明确负责人的活文档，像 runbook 一样对待，保存在读取限制与其所描述的系统相同的仓库中，因为这张表列出的是弱点。这是有成本的。访问配置通常位于读取范围更广的仓库中，所以那里的变更不会自动出现在模型的评审中。我愿意接受这一成本，并用一条规则来弥补这个缺口：修改访问配置的 pull request 要链接到对应的模型变更，或说明为什么无需变更。

有些事件应当触发更新，而不必依赖任何人记得：

- 新的服务、作业、供应商或工具触达生产数据。
- 新的数据拷贝：新的副本、导出、staging 刷新，或为一次性分析做的转储。
- 谁能触达什么发生变化：机构调整、新的值班轮换、供应商合同终止。
- 事故或未遂事件，它们是关于哪一行有误的证据。
- 访问评审中的发现项（第 14 章），它是关于哪项假定发生了漂移的证据。

每一行应当记录足够多的内容，使陌生人也能提出质疑：参与方、路径、类别、影响、暴露程度判断、控制措施及其负责人、上次评审的日期以及该行所依赖的假定。最后一项是团队最常跳过的，也是最有用的：“供应商导出只包含这些列”是一个可以检查的断言，而依赖未经检查的断言的行，就是猜测。

一个小小的检查可以找出已经过时的行。它不做任何决定。它只是把人指向最可能有误的行，并把缺失的评审日期视为过时。它还把上面列出的事件作为输入：当流水线创建了登录、角色或数据拷贝时，它会记录一个事件，而没有任何行覆盖的事件会开启一个评审项，这样更新就不依赖任何人记得。

```
function findStaleThreatRows(rows, maxAgeDays, accessEvents):
    stale = []
    // an event from the pipeline: a new login, role or data copy
    for each event in accessEvents:
        if not anyRowCovers(rows, event):
            stale.append(newReviewItem(event))
    for each row in rows:
        // a row with no review date has never been reviewed
        if row.last_reviewed is null:
            stale.append(row)
        else if now() - row.last_reviewed > maxAgeDays:
            stale.append(row)
        // a control with no owner is a control nobody maintains
        else if row.control_owner is null:
            stale.append(row)
    if count(stale) > 0:
        notify("platform", "threat model rows need review")
    return stale
```

评审节奏是一种权衡。每季度过一遍，是小团队能够坚持的，每年一次则很少足够，但我会根据 Ledgerly 的门变化得有多快来选定间隔。安全和合规联系人确定评审提出的问题，平台团队用系统的事实来回答。

权衡与成本

威胁模型的编写要花时间，维护要花更多时间。对小团队来说，第一版大概要花几天仔细讨论，而维护是一项周期性事务，要与功能开发争夺时间。如果没有人负责，成本只付一次，收益却在衰减。在只有一个服务和一个数据库的很小的系统中，完整的表超出了问题所需；半页纸，列出这些门和每扇门的最坏情况影响范围，就够了。

模型也可能制造虚假的信心：一张整齐的表会让人觉得每个威胁都在上面。它列出的是有人想到过的威胁，没有列出的门对它来说是不可见的。最好的防御是上面的更新触发条件，以及每次事故之后追问的习惯：“哪一行本可以预测到这件事？如果没有，缺失的是哪一行？”

暴露程度的判断是主观的。两位工程师对同一扇门的评级会不同，这种分歧本身是有信息量的，但评级暗示的精确程度超出了证据所能支撑的程度。我会只保留三个等级，并抵制计算分数的做法。

把人的群体列为参与方，有一种人的代价。把值班工程师或支持人员称为威胁参与方，读起来可能像指控。解决办法在于表述方式：模型描述的是路径和角色，而不是个人，其条目说明的是设计允许什么。表中出现这些团队，是因为他们有最正当的触达数据的需要。

何时不使用完整模型：如果目前没有人能回答“谁有访问权限？”，那就应当先盘点登录和数据拷贝，再写威胁，因为建立在不准确清单之上的模型，描述的是一个并不存在的系统。也不要把它当作决策的替代品。它排序并解释；团队仍然要选择先构建哪些控制措施，以及以书面形式接受哪些风险。

常见错误

- **只对外部攻击者建模**。它符合人们对安全的常见想象，控制措施也是熟悉的。代价是错误和好奇得不到应对。应当在表中为三个类别各自设置行。
- **列出控制措施而不是威胁**。“我们有 VPN”记录了答案，却没有问题。先说明可能出什么问题，再说明控制措施，这样当威胁不再适用时，就可以撤下该控制措施。
- **漏掉不是人的参与方**。流水线、作业和导出很容易被忽视，因为没有人以它们的身份登录，而它们往往握有最多的常驻权力。应当列出每个 principal 和每一项导出。
- **把数据一律视为同等敏感**。如果每一列都是“关键”，模型就无法指明从何处入手。使用四个数据类别（Restricted、Confidential、Internal、Public），让威胁的分量与数据的分量相符。
- **跳过可用性**。失控的查询看起来不像安全事件，所以会被安全评审漏掉。对支付业务来说，它的重要性可能不相上下。应当为可用性单独设置行。

- **只写一次**。模型在几周内是准确的，之后就变成口耳相传的说法。应当指定负责人并添加更新触发条件。

要点

- 威胁模型是一份简短的结构化清单，包含参与方、路径、威胁类别、影响和控制措施，它的作用是让团队能把各扇门相互比较。
- 错误、滥用和失陷需要不同的控制措施，只考虑外部人员的设计，会让前两类得不到应对；批准意图应对的是滥用，批准语句及其预期效果应对的才是错误。
- 每个参与方都应当在清单上，包括流水线、批处理作业、供应商导出以及运行控制措施的平台人员，因为没有人以其身份登录的 principal 往往握有最多的常驻权力，控制平面同样是一项资产。
- 机密性、完整性和可用性各自需要专门的行，而问责，即说明哪个 principal 做了什么的能力，一旦缺失就会抬高其他每一种威胁的成本。
- 影响范围始终针对某个具体的登录、凭据或操作，并沿七个维度来描述：数据、操作、环境、时间、拷贝、语句和网络来源。
- 在 Ledgerly 的初始状态下，一个共享管理员登录出现在几乎每一行中，因此为每个 principal 设置单独的角色，是一次就能缩小表中最多内容的控制措施。
- 模型的好坏取决于维护：为它指定负责人，让更新与新服务、新数据拷贝等事件挂钩，并把它保存在只有维护系统的人才能读取的地方。

模型指明了哪些门最重要，而对 Ledgerly 来说，几乎每一扇都通向同一个登录。第 3 章开始拆解这个登录：principal 有哪些，角色应当包含什么，以及每一方实际需要的权限有多少。

第 3 章 — 身份、角色与最小权限

设想 Ledgerly 的一个周四傍晚。一张新表 `invoice_disputes` 周二刚上线，安全联系人向平台团队提了一个简短而合理的问题：谁能读它？Alex 先去查授予记录，一条也没有，因为所有调用方都以 `admin` 连接，而它是这张表的所有者，什么都能读。Alex 接着去查实际是谁读过它。数据库日志里与这张表有关的有几千行，全部出自同一个登录：worker、夜间报表、两位工程师的笔记本电脑以及支持工具。

没有人能说清这张新表的去向。对“谁能读它？”的诚实回答是“凡是能连上数据库的人”，对“谁读过？”的诚实回答是“某个人”。没有人做错什么。设计从未要求数据库区分这些调用方，也从未要求它限制其中任何一个。

第 1 章把访问分成五个部分：**身份**、**授予**、**路径**、**执行**和**记录**。本书的论点是：生产数据库访问不是凭据分发问题，而是授权、执行与问责问题。本章服务于授权这根支柱。它涵盖前两个部分，即谁在请求（身份）和该 principal 可以做什么（授予），并标出授予在哪里终止，因为执行这一部分从那里接手（第 9 章）。

本章加入本书贯穿示例中目标状态的第一批步骤。每张表都归 `ledgerly_owner` 所有，没有人以所有者身份连接。每个服务获得自己的应用角色，授予尽可能窄但能正常工作。本章还介绍完整的角色 schema、principal 到达角色的方式、列级授予以及即时访问的上限。后面的章节在这些定义之上展开，不再重复。

问题及其难点

数据库对每条语句都要回答两个问题：这是谁，以及这个 principal 是否被允许这样做？共享账号使第一个答案失去意义，宽泛的账号使第二个答案永远是“是”。两者都很容易走到。一个登录、一个密码、全部权限，是团队创建第一个数据库那天最省事的可行做法。

代价在之后以证据缺失的形式出现。共享管理员登录的有二十个人、五个服务、部署流水线和导出步骤，当它们全都使用这个登录时，数据库日志说不出是其中哪一方执行了某条语句。应用日志描述的是应用打算做什么，而不是实际到达数据库的是什么。这是设计问题，不是纪律问题，因为个人再小心，也找不回设计已经丢弃的信息。

显而易见的补救办法是写一份策略：只授予所需的权限。策略在这里行不通，因为数据库让宽泛的选择成为容易的选择。授予全部权限只需一条语句，而且永远不会失败。授予能正常工作的最窄集合，则需要了解一个工作负载运行的每一条查询，包括没人记得的月度作业，以及一年只走两次的 retry 路径。授予过窄，会在最糟糕的时刻让生产出问题。授予过宽，什么也不会坏，所以没有人注意到。在这种不对称之下，权限只会增加。

所有权带来第二个陷阱。在大多数引擎中，表的所有者可以修改其结构、删除它，并把访问权限授予他人。如果某个服务因为启动时要创建表而以所有者身份连接，这个服务就掌握了改写数据库访问模型的权力。如果所有者是某个人，这个模型就取决于此人的任职状态。所有权是一种结构性权限，不应放在任何处理流量的东西里，也不应放在任何关乎职业生涯的东西里。

补救办法是一种结构，而不是一条指示：每个 principal 一个独立的登录，一小组承载权限的角色，以及让二者与团队意图保持一致的办法。

Principal、登录与角色

本章处理身份和授予，并在角色规定可以从哪里连接时涉及路径。**principal** 是任何能够进行认证的对象。在 Ledgerly 的世界里，有四类值得区分。

- **人员**。值班后端工程师、支持人员、数据团队和平台团队。他们来来去去，会换团队，也会在不同的地点工作。
- **服务**。`billing-api`、`invoice-worker` 和 `payments-service`。它们长期运行，以多个实例的形式运行，承载最重的流量。
- **作业**。`reporting-job` 及其导出步骤。它们按计划运行，运行结束即退出。作业所需的通常比服务少，需要的时间也更短。
- **工具**。`admin-console` 是最清楚的例子。它连接的方式像服务，但它代表一个人行事，因此每个操作都涉及两个 principal：工具，以及工具背后的人。

外部会计供应商从不连接任何 Ledgerly 数据库，它接收文件，这条路径由第 13 章处理。

身份、登录与角色是三个不同的概念

本书把三个词分开使用，因为混用它们是访问设计中大多数困惑的来源。

身份是对 principal 确为其所称之人的已验证声明。**登录**是 principal 用来连接的数据库级账号。**角色**是一组有名称的权限的集合。一个 principal 有一个身份，通过一个登录连接，而这个登录是一个或多个角色的成员。出现在数据库日志中的是登录，决定登录可以做什么的是角色。

这给出了一句值得记住的话：没有人以角色身份登录。角色的存在是为了授予给登录并持有权限，它绝不是一个人。如果日志在应当出现人员或工作负载实例的地方出现了角色名，说明模型在某处被绕过了。

对人员而言，access gateway（第 7 章）为每个会话创建一个携带此人身份的登录，该登录默认获得 `human_readonly`。对服务而言，每个实例以一个登录连接，该登录是其 `app_*` 角色的成员。在目标状态下，这个登录是动态的、短期有效的（第 4 章）。在初始状态下，它就是共享管理员登录。对核心问题的影响很清楚：日志中记录的是登录，一个登录对应一个人或一个工作负载实例，所以“是谁执行了那条语句？”有了答案。

为什么共享账号会毁掉证据

第 1 章已经完整论证过这一点；简而言之，共享账号无法归属一条语句，无法只针对一个人撤销，也无法把范围收窄到其最宽的调用方所需之下。

有一条路径需要留意：`app_admin_console` 是一个工作负载，但它的使用者是人。控制台在每个操作中传递人员 principal 的身份，下面的草图展示了做法。

```
function consoleAction(person, action):
    correlationId = newCorrelationId()
    outcome = "failed"
    // set on the session the console borrowed, cleared when it is
returned
    setSessionContext({ actor: person.id, correlation: correlationId
})
    try:
        run(action)
        outcome = "done"
    finally:
        clearSessionContext()
        // a failed or half-finished action is recorded too
        auditLogAppend({ actor: person.id, correlation: correlationId,
                         workload: "admin-console", action:
action.name,
                         result: outcome })
```

数据库登录指明工作负载，会话上下文和 `audit_events` 携带人员和 correlation id，因此应用日志、审计轨迹和语句自身的记录共用同一个键。第 14 章讲这些记录如何关联。上下文由控制台设置，所以它是工作负载作出的声明，而不是数据库验证过的事实；行规则一节中关于 pooler 的提醒同样适用于它。

四层角色层级

我会分四层来构建角色 schema，从无人使用的那一层向外，到人员使用的那几层。每一层只承担一项职责。

第一层：所有者

`ledgerly_owner` 拥有每一个 schema 对象：schema、表、视图和序列。没有人用它登录。它的存在是为了让所有权既不属于某个人，也不属于某个服务。

```
CREATE ROLE ledgerly_owner;
// no login is ever created for this role; objects are created
// under it only by a migration that assumes it

CREATE ROLE ledgerly_migrator;
GRANT ledgerly_owner TO ledgerly_migrator;
// membership grants the owner's privileges, not ownership: objects
// belong to the active role, so the migration always assumes
// ledgerly_owner explicitly, on every engine
```

第二层：migrator

`ledgerly_migrator` 执行模式变更。部署流水线使用它的登录，人员不使用，而且它自身不持有任何数据权限。它是 `ledgerly_owner` 的成员。在某些引擎上，这种成员关系不需要任何显式步骤就会传递所有者的权限，但不传递所有权：新对象属于创建时处于活动状态的那个角色，所以迁移总是显式地切换为 `ledgerly_owner`。

migrator 持有常驻的所有者权限，并通过短期有效的凭据来行使。权限本身从不过期，只有每一份凭据会过期。有三点对它形成约束：凭据只存在于流水线中，它是短期有效的（第 4 章和第 11 章），而流水线流程是使用它唯一被认可的方式。无需任何人或服务持有所有权，也能完成模式变更，而且如果这些约束成立，对所有者的每一次使用都对应一次流水线运行。第 11 章讲迁移如何评审。

属于模式变更的 backfill，在迁移内部以 `ledgerly_migrator` 的身份运行。其他任何对数据的批量变更都是数据修正，遵循第 10 章。我划出这条界线，是因为 migrator 的权力很大，不应当成为每一项棘手变更的通道。

第三层：应用角色

每个工作负载有自己的角色。每个服务一个角色，意味着要让五份定义与五个代码库保持一致。作为回报，每份凭据的影响范围与该服务的工作一样大：如果 `invoice-worker` 的那份泄露，暴露的只是 worker 能做的事，而 worker 从不需要 `payment_tokens`。

这些角色是 `app_billing_api`、`app_invoice_worker`、`app_payments_service`、`app_reporting_job` 和 `app_admin_console`。前缀说明使用者是谁：`app_` 用于工作负载，`human_` 用于人员。角色名中不含人名，也不含环境名。

第四层：人员角色

人员有三个角色，它们的不同之处在于持有方式。

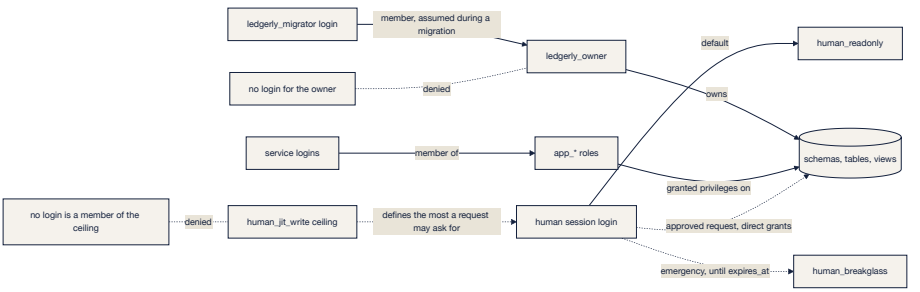
- `human_readonly` 是默认角色。它用于调查性读取，不提供 Restricted 列，并按第 12 章的规定对 Confidential 列脱敏。
- `human_jit_write` 是一个上限。它从不授予给任何登录。每份获批的申请，都会把它的一个子集直接授予会话登录，限于指定的表和指定的列，随后再收回。

- `human_breakglass` 是宽泛的应急访问，带有大量日志记录（由第 8 章定义），事后必须评审。

这三者中，上限最不直观，我在讲完授予机制之后再回到它。

示意图

下面的图显示谁是什么的成员，以及谁拥有什么。请看两个没有连线的地方：`ledgerly_owner` 没有登录，也没有任何登录是 `human_jit_write` 的成员。



授予：schema、表与列

授予既指把权限给予某个角色、把角色给予某个登录或另一个角色（或者仅在 JIT 访问中，把权限直接给予会话登录）的行为，也指由此产生的记录。**权限**是授予所允许的具体事项，例如对某一张表的 `SELECT`。这三层含义使词汇保持精确，本章中我会说明指的是哪一种。

从零开始

任何角色的起点都是完全没有权限。逐步添加授予，直到工作负载能够运行，然后停止。各个引擎对新登录默认能做什么并不相同：在有些引擎上，任何登录都可以看到对象名称，或在默认 schema 中创建对象。应当列出每个登录自动获得的权限，并去掉角色不需要的部分。

权限在三个层级上生效，并且会叠加：角色持有在任何层级授予它的所有权限的并集，较窄的授予不会收回较宽的授予。

- **Schema 级**。角色是否可以看到某个 schema 中的对象。在需要它的引擎上，没有它，表级授予不起作用，这使 schema 访问成为一个有用的粗粒度开关。
- **表级**。对整张表的 `SELECT`、`INSERT`、`UPDATE` 和 `DELETE`。
- **列级**。仅对指定列的同样这些权限。

列级授予之所以重要，是因为敏感数据的分布并不均匀：`customers` 中有 Confidential 的联系方式，`bank_accounts` 中有 Restricted 的 `iban` 和 `account_holder_name`，`payment_tokens` 中有 Restricted 的 `token_value`。表级授予对该表的每一列都说“是”。由于权限会叠加，要限制列，就意味着只在列级授予，并且该角色在这张表上不持有任何表级授予，不论是直接持有，还是通过它所属的另一个角色持有。

应用角色的授予

前两个应用角色，在表级：

```
GRANT SELECT, INSERT, UPDATE ON customers, invoices, invoice_lines TO
app_billing_api;
GRANT SELECT ON payments, ledger_entries TO app_billing_api;
GRANT INSERT ON audit_events TO app_billing_api;

GRANT SELECT, INSERT, UPDATE ON invoices, invoice_lines TO
app_invoice_worker;
GRANT SELECT ON customers TO app_invoice_worker;
GRANT INSERT ON audit_events TO app_invoice_worker;

// not granted to either role: bank_accounts, payment_tokens,
// UPDATE or DELETE on ledger_entries and audit_events
```

`app_billing_api` 可以读取 `ledger_entries` 和 `payments`，但不能写入，因为只有 `payments-service` 写 `payments`、`bank_accounts` 和 `payment_tokens`，并插入 `ledger_entries`。两个角色都不持有对任何表的 `DELETE`；如果某个服务需要，就为那张表添加授予并写明理由，评审会注意到它。

其他角色的列级授予

矩阵中带有剑号的角色受列限制。下面的列清单仅作示意：出现哪些 Confidential 列、以什么脱敏形式出现，由第 12 章决定。

```
// illustrative column lists
GRANT SELECT (id, created_at, status) ON customers TO
app_reporting_job;
GRANT SELECT (id, customer_id, amount_minor, currency, status)
    ON payments TO app_reporting_job;
GRANT SELECT ON invoices, ledger_entries TO app_reporting_job;

GRANT SELECT (id, created_at, status) ON customers TO
app_admin_console;
GRANT SELECT, UPDATE (status) ON invoices TO app_admin_console;
// UPDATE (status) is a column-level grant: the tool can change an
invoice's
// status and no other column of the row
```

最后一条授予是我最喜欢的。`app_admin_console` 可以更改发票的 `status`，而不能更改该行的其他任何内容，所以这条授予表达了意图，即使以后工具的代码发生变化，数据库也会强制执行。`app_reporting_job` 对任何主表都不持有写权限：它把结果写入 `ledgerly_analytics`。

> **注意**：在做物理复制的引擎上，角色在整个集群中只定义一次，所以每个角色可以从哪里连接，是连接层的规则（第 5 章和第 7 章），而不是授予所能表达的内容。第 6 章给出细节。

作为参照的矩阵

下表是读取或写入主表的六个角色在目标状态下的参照。S 表示 `SELECT`，I 表示 `INSERT`，U 表示 `UPDATE`，D 表示 `DELETE`，短横线表示无访问权限，剑号（†）表示受列限制：不提供 Restricted 列，并按第 12 章的规定对 Confidential 列脱敏。

表	app_billing_api	app_invoice_worker	app_payments_service	app_reporting_job	app_admin_console	human_readonly
customers	S, I, U	S	S	S†	S†	S†
bank_accounts	—	—	S, I, U	—	—	—
payment_tokens	—	—	S, I, U, D	—	—	S†
payments	S	—	S, I, U	S†	S†	S†
invoices	S, I, U	S, I, U	S	S	S, U（仅 status）	S

表	app_billing_api	app_invoice_worker	app_payments_service	app_reporting_job	app_admin_console	human_readonly
invoice_lines	S, I, U	S, I, U	—	S†	S†	S†
ledger_entries	S	—	S, I	S	—	S
audit_events	I	I	I	—	I	S

我按列来读这张矩阵。每一列都是以权限写成的某个工作负载的岗位说明，评审者可以问它是否依然属实。纵观整张矩阵，只有 `payments-service` 能触达 `bank_accounts`；除 `human_readonly` 的一次受列限制的读取之外，只有 `payments-service` 接触 `payment_tokens`；而且默认情况下，没有任何人员角色可以写入。

Append-only 是结构性的

没有任何应用角色、人员角色或 JIT 授予持有对 `ledger_entries` 或 `audit_events` 的 `UPDATE` 或 `DELETE`，这包括 `human_breakglass`。更正是一条新记录，绝不是编辑。账本更正只通过 `payments-service` 的更正路径发出，绝不由人员登录发出。

有一个结构性的例外：`ledgerly_owner`，因此还有正在运行的迁移，可以修改这些表，因为所有者可以修改或删除表，并把它上面的任何权限授予自己。这就是第 11 章要评审每一项涉及这些表的迁移的原因。控制措施在评审中，而不在权限里。

默认权限、所有权与超级用户

默认权限

许多引擎可以让一条授予适用于以后创建的表，形式可以是附加在某个所有者上的默认权限，也可以是覆盖整个 schema 的授予。这在起初很方便，之后却是个陷阱，因为新表会悄悄继承授予所规定的任何内容。

我倾向于使用空默认值。新表除了所有者之外无人可读，创建它的迁移在同一个文件里写明每一条授予。代价是每个迁移多几行；好处是一次评审就能看到一项变更对访问的全部影响。

> **警告**：把未来表的 `SELECT` 授予给 `human_readonly` 之类宽泛角色的默认权限，会让下一张敏感表在创建的那一刻就暴露。应当首先评审这类默认值，并删除所有覆盖未来表的默认值。

这里的记法仅作示意，因为默认权限没有中立的语法。

```
// illustrative notation: defaults for objects created by
ledgerly_owner
DEFAULT PRIVILEGES FOR ledgerly_owner ON TABLES: none

// each migration that creates a table states its own grants
CREATE TABLE <new_table> (...);
GRANT SELECT ON <new_table> TO app_billing_api;
```

所有权的危险

在常见引擎中，所有者可以删除对象、修改其定义、把权限授予任何人，并且在很多情况下，可以无视对其他人适用的行级限制。持有所有权的人可以更改访问模型本身。

所以围绕 `ledgerly_owner` 的规则很严格，而这种严格几乎不花成本。它没有登录。应用角色从不拥有表，即使是它们为自身使用而创建的表也不例外，因为“服务在启动时创建它的表”正是工作负载最终拥有部分 schema 的途径。所有者是系统中使用最少的角色，这是正确的状态。

超级用户

大多数引擎都有一个内置的管理角色，它绕过权限检查，不过托管服务可能不提供它。我在这里称它为超级用户；它在 Ledgerly 的角色 schema 之外，因为它是引擎自带的。

我会把它当作封存的凭证。没有应用、作业或人员会话在日常工作中使用它。它的凭据保存在平台团队可以取得的地方，并且每次取得都会留下记录，它用于任何角色都无法完成的引擎级操作，例如修复复制。诚实的局限在于，它可以撤销本章中的每一项控制措施，而我不知道有哪种设计能在消除这种权力的同时不使恢复变得不可能。应当让它的使用少见、可见且经过评审；第 15 章会再次讨论它。例行管理，例如应用角色定义或签发登录，改用 `ledgerly_access_admin`。如果超级用户被用于例行任务，说明某个角色缺少它本应拥有的权限。

`ledgerly_access_admin` 是一个由三个 principal 使用的角色，每个都通过自己的登录：应用访问定义的流水线步骤、凭据 broker（第 4 章）和 access gateway（第 7 章）。它创建登录并管理成员关系，不拥有任何表，也不持有任何数据权限，而这仍然是很宽的权力：持有它的人可以在它所管理的角色之内授予任何东西。独立的登录让三者在日志中保持区分，但不改变它们各自可以做什么。

因此，约束控制平面自身权力的不是角色，而是周边环境。三个登录中的每一个，只能从各自的组件到达。每个操作都进入审计日志（第 14 章），下面的漂移检查会把结果与代码比较。对访问定义的变更，经过它们自己的一小组评审者。这些限制使滥用可见，但不使滥用不可能，我会在访问评审中如实说明。

进一步收窄：视图、行规则与租户

视图

视图是一条行为类似表的存储查询。获得视图访问权限的角色，可以读取查询所暴露的内容，而无需持有底层表上的权限。

```
// illustrative: a narrowed projection owned by ledgerly_owner
CREATE VIEW customer_support_summary AS
    SELECT id, created_at, status FROM customers;
GRANT SELECT ON customer_support_summary TO app_admin_console;
```

> **注意**：在许多引擎中，视图以其所有者的权限运行，而不是以调用方的权限运行。收窄之所以有效，正是由于这一点；同样的原因也使得选取过多内容的视图，会把这些内容泄露给被授予该视图的每一个角色。有些引擎也允许选择另一种行为。应当查明所用引擎的默认行为，并用一个对基表不持有任何权限的角色来测试视图。

只要引擎支持，我更倾向于列级授予，因为授予就放在表的旁边。当形态本身就是目的时，视图更合适：连接、计算值，或者在不断变化的表之上提供稳定的接口。本书不把按申请创建的视图用于 JIT 访问，原因有二：视图必须有所有者，所以创建它的 gateway 就需要所有者权力，而为每个申请创建一个视图会破坏所有权模型。

行规则与租户：一条决策规则

行规则在矩阵的某个单元格内收窄行，并不改变单元格本身所表达的内容。支持人员一次只查看一位客户，所以像“`app_admin_console` 只能看到 `invoices` 中 `customer_id` 等于该案例客户的行”这样的规则，适用于列过滤不适用的地方。各引擎在语法上互不一致，所以本书不给出记法。

Ledgerly 的客户是企业，每一家都期望看不到别家的发票。这种隔离主要是应用的工作：`billing-api` 在每条查询中都按客户过滤，没有任何角色把一个租户与另一个租户分开，因为应用角色要为任何一个租户提供服务，就必须能读取所有租户。

只有同时满足两个条件时，才使用行规则，无论用于支持场景还是作为第二层租户隔离：错误的行后果严重，并且应用能为每个会话可靠地设置上下文值。代价是每条语句都带着谓词，而且必须有东西为每个会话设置上下文。在 pooler 之后，在一个会话上设置的值可能传给同一连接的下一个借用者（第 5 章会回到这一点）。当上下文缺失时，规则必须返回零行，而第一个测试就是这种情况。失败时开放的规则，比没有规则更糟。独立的数据库是更强的边界，运维代价则高得多；哪一种合适，取决于租户有多少，以及对他们作出了什么承诺。

即时访问的上限

`human_jit_write` 定义上限：一份即时（JIT）申请最多可以请求的范围，以对指定表和指定列的权限来表达。JIT 访问是为某项具体任务、在有限时间内授予的访问；审批和有效期由第 8 章讲解，本章只定义上限。

这个角色从不授予给任何登录。一份申请是一组（权限、表、列）三元组，位于上限之内，并与申请一起存储。获批时，恰好这些三元组被直接授予申请人的会话登录；到期时，恰好这些三元组被撤销。共享角色无法按申请划定范围，所以按申请的范围体现在授予中，而角色划出外边界。

```
// the ceiling: what a JIT request may ask for, never granted to a
login
GRANT SELECT ON customers, payments, invoices, invoice_lines
    TO human_jit_write;
// id and created_at are assigned by the database, never changed or
supplied
// by a data fix, so the write privileges below name every other
column
GRANT UPDATE (<every column except id, created_at>)
    ON customers, payments, invoices, invoice_lines TO
human_jit_write;
GRANT INSERT (<every column except id, created_at>)
    ON customers, payments, invoices, invoice_lines TO
human_jit_write;
// DELETE only where a request names it, and never on a table with
// Restricted columns
GRANT DELETE ON customers, payments, invoices, invoice_lines
    TO human_jit_write;

// illustrative column lists; Restricted columns are not named
anywhere
GRANT SELECT (customer_id), UPDATE (customer_id)
    ON bank_accounts TO human_jit_write;
GRANT SELECT (customer_id, provider, last_four, expires_on),
      UPDATE (customer_id, provider, last_four, expires_on)
    ON payment_tokens TO human_jit_write;
// no table-level SELECT on either table: it would include the
Restricted columns
// no INSERT or DELETE on either: a new or removed row sets or
destroys
// Restricted values
```

应当留意这个代码块略去了什么。`bank_accounts` 和 `payment_tokens` 只对少数几个指定的非 Restricted 列开放，对行标识符或时间戳从不开放：`iban`、`account_holder_name` 和 `token_value` 永远不可能出现在 JIT 申请中。`INSERT` 仅限前四张表，而且不能指明 `id` 或 `created_at`，所以恢复的或新建的行总是获得由数据库分配的值。两张 append-only 表以及属于 gateway 的访问申请记录，完全在上限之外。除 `DELETE` 之外，每个三元组都指明了列。

有两项成本，在此一次说明。把行与其所有者关联起来的列，例如 `customer_id`，不是 Restricted，所以 JIT 的 `UPDATE` 可以改变它们。这类所有者关联变更仍在上限之内，但它属于危险操作类别：执行策略对其设置关卡，并将其路由到产品路径（第 10 章），而不是作为原始 SQL 运行。关卡和产品路径才是控制措施，不能只靠四眼审批。此外，JIT 的 `SELECT` 会不脱敏地显示它所指明的任何 Confidential 列，这是审批人接受的内容的一部分。

让上限名实相符的，是检查，而不是信任。gateway 拒绝上限之外的任何申请（第 8 章展示这项检查），而下面的漂移测试断言没有任何登录是 `human_jit_write` 的成员，这种失败会把上限变成常驻授予。

引擎不支持列级授予时

并非每个引擎都支持列级授予。本书只使用一种后备方案：在引擎无法按列授予的地方，把 Restricted 列移到单独的表 `bank_account_secrets` 和 `payment_token_values` 中。

```
table bank_account_secrets:      // owned by ledgerly_owner
    bank_account_id     text primary key
    iban                text not null
    account_holder_name text not null

table payment_token_values:      // owned by ledgerly_owner
    payment_token_id    text primary key
    token_value         text not null

// reachable by app_payments_service and human_breakglass only;
// outside the ceiling, never part of a JIT request
```

有了这个后备方案，主表中不再含有任何 Restricted 列，所以对它们的表级 JIT 授予无法触及 Restricted 数据。代价是：申请的细节降到表一级，审批人批准的是该表的每一个非 Restricted 列，包括 Confidential 列，而不仅仅是申请人指明的那些。

Break-glass 与位于 schema 之上的操作

`human_breakglass` 是唯一能读取 Restricted 列的人员角色。它读取每一张表，并对除两张 append-only 表之外的每一张表拥有 `INSERT` 和 `UPDATE` 写权限；对于 `bank_accounts` 和 `payment_tokens`，它只写 `UPDATE`，而且只能用于非 Restricted 列。它用于应急，而不是例行修复，并且它不出现在矩阵中，因为只有会话在短暂的窗口内持有它；没有任何常驻的工作负载持有它。

Restricted 列只通过应用路径更改。人员绝不会用原始 SQL 写入它们，无论是在 JIT 会话中还是其他情形：`admin-console` 在第二个人批准之后向 `payments-service` 提出请求，后者是唯一的写入方，并且该变更会记录在 `audit_events` 中。第 10 章描述这条路径。这些事实标出了授予所能做到的边缘。授予说明会话可以接触哪些表和列；它无法说明是哪条

语句、多少行，或者这次具体的变更是否就是被批准的那一次。这是执行这一部分的工作，第 9 章讲 gateway 如何检查每条语句。知道授予止于何处，有助于判断一项新能力应当放在哪里。

角色即代码，以及漂移

只存在于数据库中的角色 schema，是一份无人能评审的 schema。上面的授予足够短，可以阅读，它们应当放在版本控制中，通过评审来变更，并由流水线应用，就像它们所保护的 schema 一样。

定义存放在哪里

我会保留两类文件。迁移创建表并写明每张表的授予。另有一份单独的、经过评审的访问定义，列出角色、角色之间的成员关系和上限。流水线把二者作为单独的步骤来应用。访问定义通过该流水线步骤自己的登录、以 `ledgerly_access_admin` 的身份应用，这是 broker 和 gateway 通过各自登录所使用的同一个角色；它不持有任何数据权限。创建角色是比更改表更强的权力，所以它的步骤应当有自己的一小组评审者。评审者的问题不是 SQL 是否有效，而是这条授予是否是能让变更生效的最窄授予。评审者打开矩阵，一分钟就能回答。

收窄现有系统

Ledgerly 起初只有一个拥有一切的共享登录。我不会一次变更就从那里迁到矩阵，因为过窄的授予会让生产出问题，而且团队要等到失败才会知道漏掉了什么。一条分四步的路径：

1. **先给每个工作负载各自的登录，权限与之前相同。**什么都不会坏，日志也立即开始记录工作负载的名字。
2. **观察。**用每个登录下记录的语句，列出工作负载实际接触的内容，观察的时间要足够长，以涵盖运行缓慢的月度作业。语句日志可能含有字面值，所以应当将其视为敏感：其中可能包含客户数据，或被粘贴进查询里的 Restricted 值。应当限制谁能读它，设定它的保留期，并且在引擎提供时，优先记录带参数的语句形态。
3. **定义窄角色，并把登录迁入其中**，先在运行相同作业的 staging 环境中进行，然后在生产中进行，并在短期内保留旧权限以备不时之需。
4. **移除旧权限**，并留意被拒绝的语句。

成本是日历时间，而不是精力。团队倾向于跳过第二步，可预见的后果是一条很少使用的路径在几个月后出问题。

漂移

漂移是代码所声明的访问与数据库实际持有的访问之间的差距。它出现在夜里用手写的 `GRANT` 修复问题的时候：授予生效了，事故结束了，没有人把它记下来。检查按计划比较声明与实际，并且在无法读取其中一方时必须明确失败。

```
function checkRoleDrift():
    declared = loadDeclaredAccess()
    try:
        actual = readActualAccess()
        requests = loadAccessRequests()
        known = loadGatewayLogins()
    on ReadFailed:
        // an unreadable state is not a clean state
        notify("platform", "role drift check could not read a source")
        raise DriftCheckFailed

    for each extra in difference(actual.roleGrants,
declared.roleGrants):
        notify("platform", "grant not in code: " + extra)
    for each login in actual.loginsOwningObjects():
        notify("platform", "login owns objects: " + login)
    for each login in actual.membersOf("human_jit_write"):
        notify("platform", "login is a member of the ceiling: " +
login)
    for each login in actual.allLogins():
        if login is human and login not in known:
            notify("platform", "login unknown to the gateway: " +
login)
        // memberships must match what the login was issued for
        for each role in difference(actual.rolesOf(login),
expectedRoles(login)):
            notify("platform", "unexpected membership: " + role + "
for " + login)
        // a direct grant is legitimate only as part of a live request
        live = unexpiredTriples(requests, login)    // empty for
service logins
        for each g in difference(actual.directGrantsOf(login), live):
            notify("platform", "direct grant without a live request: "
+ g + " to " + login)
    for each login in actual.membersOf("human_breakglass"):
        if not hasOpenBreakGlass(requests, login):
            notify("platform", "break-glass member without a request:
" + login)
```

`expectedRoles(login)` 对 gateway 会话登录是 `human_readonly`，在 break-glass 申请处于打开状态期间还包括 `human_breakglass`，对服务登录则是 broker 为该登录签发的那一个 `app_*` 角色。因此，这项检查

既能发现手写的 `GRANT app_payments_service TO <session login>` 或 `GRANT ledgerly_owner TO <login>`，也能发现直接的权限授予。

这里的 `roleGrants` 只涵盖角色到角色的成员关系，以及权限到角色的授予，这些是代码所声明的。登录不在代码中：它们随会话而来去。人员登录按照 gateway 的记录来检查。每个登录（包括服务登录）的直接授予，都与尚未到期的 `access_request_privilege` 三元组进行比较。服务登录没有申请，所以其上的任何直接授予都会被标记；夜里手写的把某项权限授予服务登录的 `GRANT`，就是这样被发现的。比其申请存活得更久的授予，或没有处于打开状态的申请的 break-glass 成员，即使撤销作业报告成功，也会被标记。JIT 访问靠直接授予运作，到期靠撤销作业；如果作业停止，授予仍然存在，而这项检查独立于作业发现这一点（第 8 章讲这种配对）。这项检查只读取授予和成员关系。它看不到会话用这些做了什么，那又是执行这一部分的事了。

这项检查不修复任何东西，因为在生产中自动撤销未知的授予，是在夜里让生产出问题的一种新方式。它也不判断声明的状态是否正确：一条错误但声明无误的授予会通过检查。这种判断属于第 14 章的访问评审。

权衡与成本

本章的每项控制措施都有代价，而代价由特定的人来承担。

按工作负载划分的角色需要维护。五个角色意味着要让五份定义与五个代码库保持一致。当某个服务需要一张新表时，必须有人在同一次变更中更新角色；如果授予晚于代码到达，部署就会失败。缓解办法是让授予成为迁移的一部分。只有一个服务和两名工程师时，这笔开销并非微不足道，一个更小的 schema，即一个所有者、一个应用角色和一个只读角色，是合理的第一步。

窄授予会造成被拒绝的语句。有些拒绝是授予中的错误，有些是代码伸得比应有的更远。当拒绝出现在事故期间，两者都由值班的人来承担。让 staging 使用与生产相同的角色，可以减少意外；无视这笔成本，会让团队匆忙放宽授予，而匆忙正是漂移的开端。

列级授予增加评审负担。表发生变化时，每一份列清单都要维护。在表级授予之下，新的敏感列默认就是暴露的，所以对敏感表，我在指定的列上授予，并接受多出来的那几行。在引擎缺少该功能的地方，后备方案要付出连接、额外需要迁移的表以及更粗的 JIT 审批的代价。

视图和行规则增加活动部件。当形态或隔离本身就是目的时，它们物有所值；为了显得谨慎而加上它们，则是一笔糟糕的交易。上下文可能丢失的行规则是安全剧场：它给每条查询增加摩擦，却在它为之而写的那种情形下失败时开放。

所有权纪律会拖慢一些人。习惯了在控制台里创建表的工程师，现在必须走迁移。这种摩擦是有意为之，它在一次性的开发数据库上的价值，低于在生产上的价值。应当先把这个模型应用到生产，只有在能说清楚放弃了什么的时候，才在别处放宽。

何时不需要这么多结构。只有一名开发者、没有客户数据的原型，不需要完整的角色 schema。所有者、应用和只读的划分，在出现第二个服务或第一个敏感列之前就够用了。当“是谁执行的？”无法从日志中回答，或者某个工作负载的凭据可能造成的损害超出其工作所能说明的程度时，再扩展这个 schema。

常见错误

- **把应用角色当作所有者**。这很诱人，因为这样迁移可以在启动时少一步就运行。服务随即掌握了删除和重新授予的权力。应当使用单独的 migrator 和一个没有登录的所有者。
- **按去年的查询来授予**。从现有最宽的登录复制来的角色，会保留曾经方便过的每一项权限。应当根据工作负载现在所做的事来推导角色，并在其自己的登录下观察。
- **每个人一个角色**。角色是权限的集合，登录才是每人一个。以个人命名的角色，会把 schema 变成一份员工名单。角色应当围绕职责。
- **因为几个服务“相似”就共用一个角色**。这些服务会各自分化，而角色保留的是各自所需的并集。每个服务一个角色，只多几行。
- **把默认值留给未来的表**。默认权限使每个迁移省一行，却把下一张敏感表暴露给宽泛的角色。应当保持默认值为空，并显式授予。
- **为了忙碌的一周把上限角色授予某个登录**。上限随即变成一个带着友好名称的常驻授予。应当让上限保持不附着于任何登录：只有申请检查读取它，漂移测试会标记任何成员关系。
- **把敏感列藏在视图后面，却让表保持可读**。对于能够触达基表的角色，视图什么也没有收窄。应当在表上撤销权限，并用一个不持有任何表权限的角色来测试。

- **相信一份无人比较的声明式 schema**。版本控制中的角色，挡不住凌晨 2 点的那条 `GRANT`。应当按计划比较声明与实际，并让比较在无法读取时明确失败。
- **把超级用户用于例行工作**。它每次都管用，这正是它蔓延的原因。当例行任务需要它时，应当把缺少的权限添加给合适的角色。

要点

- 共享登录同时毁掉证据和范围：日志说不出谁采取了行动，而登录必须被允许去做其最宽的调用方所做的任何事。
- 把身份、登录和角色分开：一个 principal 有一个身份，通过一个登录连接，而登录是角色的成员；没有人以角色身份登录。
- `ledgerly_owner` 拥有一切且没有登录，`ledgerly_migrator` 仅在迁移期间切换为它，每个工作负载获得与其工作相称的自己的 `app_*` 角色。
- 每个角色都从没有权限开始；在敏感列与普通列并存的地方使用列级授予；保持默认权限为空；只有在引擎无法按列授予的地方，才退回到单独的机密表。
- `human_jit_write` 是一个从不授予给任何登录的上限：每份获批的申请把一个子集直接授予会话登录，没有任何申请可以包含 Restricted 列。
- 角色和授予属于版本控制和经过评审的流水线步骤，而按计划运行、在无法读取数据库时明确失败的漂移检查，使声明状态与实际状态保持一致。
- 每一层都让某个人付出代价：角色要维护，窄授予会造成被拒绝的语句，列清单增加评审负担，而在引擎缺少列级授予的地方，审批会更粗。

角色说明了 principal 连上之后可以做什么，但对服务如何证明自己是谁，还没有说任何话。第 4 章迈出下一步：凭据如何到达工作负载，为什么短期有效的动态凭据改变了问题的形态，以及配置文件里的密码由什么来取代。

第二部分

应用访问

第 4 章 — 凭据与机密

设想 Ledgerly 的一个周五下午。后端团队有人调去了另一个小组，安全联络人客气地问，共享的管理员密码能否更换。平台团队的 Alex 同意应当更换。随后 Alex 开始列出这个密码存放的位置：全部五个服务的配置、流水线的变量、`reporting-job` 的导出步骤、一个团队密码管理器以及若干桌面客户端里保存的连接。

在数据库里修改密码只需一秒。若要在所有地方同时修改，就得在工作时间协调重启每一个服务；本周末值班的 Robin 提出了一个合理的问题：处理中的发票会怎样？这次修改被推迟到“一个安静的窗口期”，而这个窗口期始终没有到来。密码比这个人活得更久，接着又比下一个人活得更久。

本章回答那个周五引出的问题：服务如何获得凭据，又如何在不中断服务的情况下更换凭据？本章加入示例主线中目标状态的下一步：服务使用的凭据变为短期凭据或动态凭据，并按身份送达工作负载，而不是经由共享文件。

凭据属于五部分模型中的“身份”部分，它们必不可少，但并不是答案。本书的论点是：“生产数据库访问不是凭据分发问题，而是授权、执行与问责问题。”更好的凭据管理能缩小可以取得登录的人群，却无法说明登录进入之后可以做什么，也无法说明它做过什么；这些是关于授予、执行与记录的各章的任务。

问题及其难点

凭据是 principal 用来证明身份的东西；**机密**是任何一旦泄露就能获得访问权的值。密码两者皆是。第 3 章给每个服务分配了各自的角色，使登录能够说明它可以做什么。本章讨论另一半：登录如何证明自己是谁，以及这份证明能维持多久。

即使无人设计，密码也有生命周期。它被创建，被复制到服务能读取的各个位置，使用数月，最终被更换或被遗忘。每一份副本都是一个人或一个进程可以读取它的地方，“谁能看到这个密码？”这个问题有多少份副

本，就有多少种答案。在初始状态下，答案是“后端团队几乎所有人，以及一切能读取服务配置的东西”。

显而易见的补救办法是换一个更好的密码：更长、随机、存放得更整齐。这有助于对付猜测，而猜测从来不是主要风险。风险在于这个值所处位置的数量，以及其中任何一处都无法在不触动其他各处的前提下清空。五个配置文件里的强密码仍然是五份副本。

第二个难点是**轮换**，即按计划或因怀疑而用新凭据替换旧凭据。轮换之所以困难，是因为凭据是两个系统之间的共享状态：数据库知道这个值，每个使用方也知道。先改动其中一侧，另一侧就会失效。一个无法在不中断服务的情况下更换密码的团队，理所当然会不再更换，而一条无人能够执行的轮换策略，只是多了一个日历条目，并没有消除任何暴露。

凭据做不到什么

设想报表作业的一个登录，有效期一小时，被泄露到了聊天频道。按本章的各项标准衡量，它都是一个好凭据：唯一、短期，并按身份获取。在这一小时里，它仍然可以读取该角色被授予的 `customers` 三列（`id`、`created_at` 和 `status`），涵盖每一行，因为 `app_reporting_job` 就被允许读取这些。短期缩小的是时间窗口，而不是角色。限定泄露范围的有三样东西，没有一样是凭据本身：它所携带的**角色**（第 3 章）、决定它可以从何处使用的**网络触达范围**（第 7 章），以及记录它做过什么的**日志**（第 14 章）。一个只能从 pooler 的网络出示、所带角色看不到 Restricted 列的登录，造成的事故比任何密码策略所能做到的都要小。

下面最强的设计同样如此。broker、租约和 workload identity 回答的都是谁可以获得登录。它们不决定登录可以做什么，也不会阻止一个被许可的登录运行有害的语句。

这是一个工程问题，而不是策略问题。一条写着“每 90 天轮换一次”的规则，对五个服务如何在同一时刻获知新值只字未提。答案在于凭据如何签发、交付和替换，这是团队必须自己构建的设计。

凭据的生命周期

不妨沿着一份凭据从诞生到退役的过程逐站追踪，并在每一站问：谁能读取它，什么会暴露它。常见的位置如下。

位置	谁能读取	通常如何泄露	更好的存放处
仓库中的配置文件	所有有仓库访问权的人，且通过历史记录永久可读	被复制到 fork、聊天或工单中	机密管理器，按名称引用
环境变量	进程及其子进程，以及一切能检查该进程或其崩溃转储的人	被调试 endpoint 或错误报告打印出来	启动时交付的文件或内存，并设置严格的文件权限
容器镜像或构建制品	所有能拉取该镜像的人	在构建时被固化进去，于是每份副本都带着它	镜像中不放任何内容；启动时获取
CI/CD 变量与作业日志	流水线管理员；有时是每一个作业	被脚本回显，被失败的命令打印出来	由作业自身身份获取的短期值
团队密码管理器	共享保险库中的所有人	出于习惯而共享；没有谁读取过的记录	对剩下的少数情形，采用带记录的个人访问
桌面客户端保存的连接	使用那台笔记本电脑的任何人	笔记本电脑丢失或被备份	不保存凭据；使用来自 gateway 的会话凭据（第 7 章）

表中有两点很突出。第一，泄露几乎从来不是什么高明手法，而是出于方便而产生的副本：为了让同事复现问题，把一个值粘贴进工单；为了调试一次部署，回显了某个变量。第二，“更好的存放处”一栏大多是同一个答案：这个值存放在一个受管理的地方，使用方在需要时获取。

> **警告**：凡是出现在镜像层、仓库历史或作业日志中的机密，都必须视为已经泄露，即使无人读取过。删掉那一行并不会删掉副本。补救办法是更换凭据，这也是团队需要在需要之前就备好一条可用轮换路径的原因。

机密还有一个值得点明的属性：它保持有效的时间有多长。假设一个密码有效期为三年：它就背负着历来每一份副本带来的三年暴露。只持续一小时的凭据，其旧副本一文不值。这些时长仅为示意。这正是有效期比强度更重要的原因。

机密管理器与服务如何读取机密

机密管理器是一种服务，它存放机密，控制谁可以读取每一个机密，并记录每一次读取。HashiCorp Vault 和大型云服务商的机密存储就是例子。模式比产品更重要：由一处持有这个值，按 principal 决定访问，每次读取都留下记录。

把密码移入机密管理器，本身并不会减少读取者的数量。它改变的是读取者是谁，并让每一次读取都有了名字：某个特定的服务身份读取某个特定的路径，机密管理器会说明读取发生在何时。这对调查和撤销是实在的收益，对降低暴露则是有限的收益。

第一个机密

服务在读取任何内容之前，需要有某样东西向机密管理器证明自己。如果这样东西是另一个存储的密码，问题只是上移了一层，什么也没有解决。这就是引导问题，有时称为第一个机密问题。

答案是 **workload identity**：由平台赋予运行中工作负载（例如服务实例或作业）的身份，使其无需存储密码即可认证。启动该工作负载的平台为它担保，机密管理器信任这一声明。服务在这一步从不持有属于自己的长期值。我把 workload identity 视为使整个设计其余部分成为可能的关键，因为它终结了无穷回溯。

启动时读取与再次读取

服务读取数据库凭据有两个合理的时机：启动时读取一次，或每次打开连接时读取。只读一次很简单，对很少变化的静态凭据也行得通。一旦凭据真的变化，它就失效了，因为运行中的进程会保留旧值直到重启。在连接时读取，并通过一个在认证失败后再次向机密管理器查询的客户端，则无需重启就能应对轮换。

```
function connectWithCredential(service):
    attempts = 0
    while attempts < maxAttempts:
        try:
            slot = readSecret(path: "db/" + service + "/active")
            credential = readSecret(path: "db/" + service + "/" +
slot)
            return openConnection(host: "db-pooler.internal.example",
                                  login: credential.login,
                                  password: credential.password)
        on AuthenticationFailed:
            // the value may have rotated since the last read; read
again
            attempts = attempts + 1
        on SecretStoreUnavailable:
            // fail the start; never fall back to a value baked into
the image
            raise StartRefused
    raise StartRefused
```

这段代码展示了两种要紧的失败。认证失败会触发重新读取，因此一次轮换不会变成一次中断。机密管理器不可达时则拒绝启动，因为退回到某份备用副本，会重新造出这个设计已经移除的那个文件。这次拒绝是有代价的，“权衡与成本”一节会回到这一点。

> **注意**：机密管理器本身就是一扇门。谁可以读取路径 `db/billing-api`，谁可以修改它，读取是否记入日志，这些问题与数据库面临的问题相同。让每个服务都能读取每一条路径，等于多绕一步，重建了那个共享密码。

无中断轮换

要解决“先改一侧，另一侧就失效”，办法是永远不出现只有一个值有效的时刻。这就是**双凭据模式**，常称 A/B 轮换。每个服务有两个登录 A 和 B，二者都是同一个 `app_*` 角色的成员。任一时刻，一个处于活动状态，另一个闲置。

轮换时，先修改闲置的登录，发布新值，通知使用方切换，等到旧登录不再被使用，然后使其失效。在整个过程中，没有任何使用方持有数据库已不再接受的值。下面用伪代码表示，并标出了失败点：

```
function rotateServiceCredential(service):
    active = readSecret(path: "db/" + service + "/active")   // "a" or
"b"
    idle = otherSlot(active)
    if openSessions(loginFor(service, idle)) > 0:
        // an earlier rotation did not finish; changing this login
cuts live sessions
        raise RotationIncomplete
    newPassword = generatePassword()
    // the idle login has no users, so changing it breaks nothing
    setLoginPassword(login: loginFor(service, idle), password:
newPassword)
    try:
        writeSecret(path: "db/" + service + "/" + idle,
                    value: { login: loginFor(service, idle), password:
newPassword })
        writeSecret(path: "db/" + service + "/active", value: idle)
    on SecretStoreUnavailable:
        // the active login is untouched; the idle one holds a value
nobody can read
        notify("platform", "rotation stopped before publish: " +
service)
        raise RotationAborted
    // consumers re-read on their next connection; wait for the old
login to drain
    waitUntil(openSessions(loginFor(service, active)) == 0, deadline:
drainDeadline)
    if openSessions(loginFor(service, active)) > 0:
        // the old value still works; the rotation is not finished
        notify("platform", "old login still in use after rotation: " +
service)
        raise RotationIncomplete
    setLoginPassword(login: loginFor(service, active), password:
generatePassword())
    auditLogAppend({ action: "rotate", subject: service, actor:
"rotation-job" })
```

有三个细节支撑着这个设计。第一，数据库与机密管理器是两个独立的系统，没有共享事务，因此步骤的顺序决定了失败的影响范围：正在生效的值，在它的替代值可被读取之前绝不会被改动。任何一步失败，都会让先前的凭据继续有效。第二，消化检查。只有在没有任何东西使用旧登录时，轮换才算完成，而一个从不重新读取机密的服务，会在此刻被发现，而不是在三周后的一次中断中被发现。第三，最后一步用一个无人保存的值覆盖旧登录，但这一步只在消化之后才运行。只要还有任何会话持有旧登录，旧值就仍然有效。

代价是，现在每个服务有两个登录，还有一个同时了解这两者的作业，并且使用方必须在失败后重新读取机密。只在启动时读取凭据的服务，轮换时需要滚动重启，而这次重启正是 A/B 轮换再次变成日历事件的地方。

在初始状态下轮换共享的管理员密码之所以困难，原因就在这里：没有第二个登录，也没有任何使用方能够重新读取任何东西。

我会按团队能够坚持的节奏来轮换，而不是按能写下来的最短周期。怀疑泄露是另一条路径，见“凭据泄露时”。

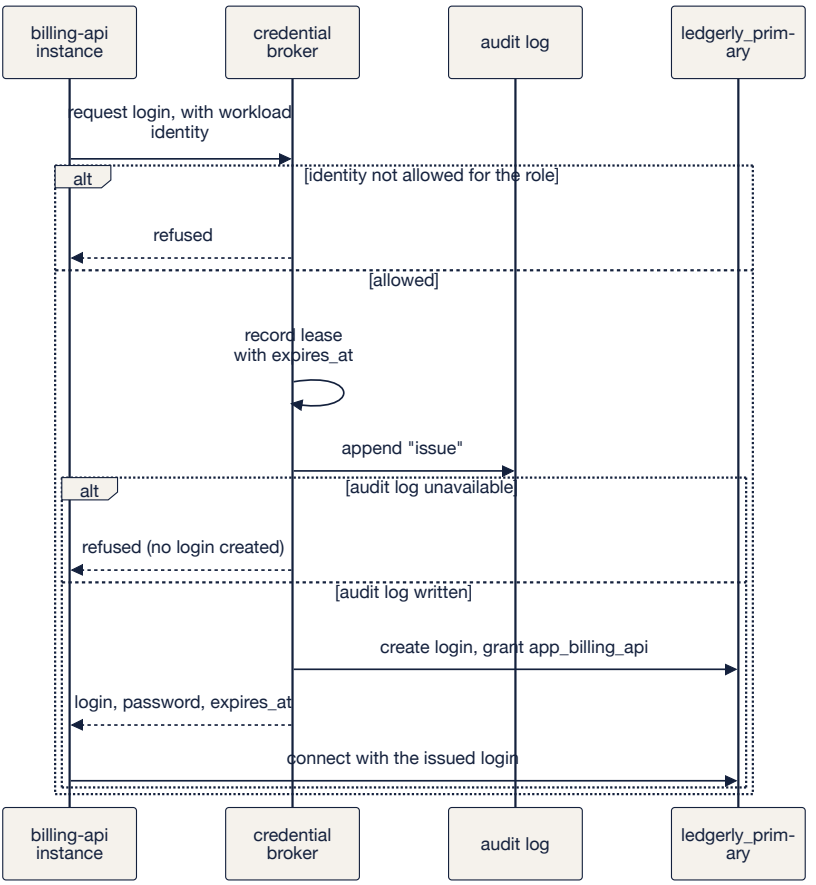
动态凭据与短期凭据

经过轮换的静态凭据，在两次轮换之间仍然有很长的有效期。下一步是把有效期缩短到让轮换不再成为一个事件。**短期凭据**会在一段简短且明确的时间内自行到期。**动态凭据**是应请求为某一个 principal 生成的短期凭据，因此两次请求永远不会得到相同的值。所有动态凭据都是短期的；并非所有短期凭据都是动态的。

通常的机制是**凭据 broker**：这是一种服务，它应请求在数据库中创建一个登录，授予其正确的 `app_*` 角色的成员资格，并把该登录的凭据连同一份**租约**一起交给发出请求的工作负载。租约是一条记录，说明谁持有该凭据、对应哪个角色、到何时为止。租约到期或有人撤销时，broker 会删除该登录。

在 Ledgerly 中，broker 用自己的登录连接，它是 `ledgerly_access_admin` 的成员，这个角色负责创建和删除登录并管理角色成员资格。它不拥有任何表，不是所有者，也无人用它做数据工作。这比所有权的权力要窄，但仍然很宽：持有该角色的人，可以在它所管理的角色之内授予任何权限，应用角色和人员角色都在其中。broker 只需要这份权力的一小部分。给它更少的权力就意味着拆分这个角色，而这有它自己的成本；这里 broker 按现状使用该角色。这个角色是一个由三个 principal 使用的角色，各自通过自己的登录：流水线步骤、这个 broker 和 gateway。第 3 章说明了是什么约束这份权力。对 broker 而言，防护措施就是这里所作的那一条声明：它的凭据与所有者的凭据同样妥善保管，只有它自己的 workload identity 才能获取，它采取的每一项动作都会写入审计日志（第 14 章）。

下面的时序展示了一个 `billing-api` 实例请求登录的过程。请留意请求被拒绝的两处：在任何东西存在之前，以及审计日志无法访问时。



在伪代码中，操作的顺序保证了这个时序的正确。`createLogin` 之类的函数名只是示意，与本书其他地方一样。

```
function issueServiceLogin(request):
    identity = verifyWorkloadIdentity(request.proof)
    if request.role not in rolesAllowedFor(identity):
        raise IssueRefused
    password = generatePassword()
    login = identity.workload + "-" + randomSuffix()
    expiresAt = now() + ttlFor(request.role)
    // the lease comes first: the sweeper reads it, so no login can
exist
    // without a record of when it must end
    leaseId = recordLease({ login: login, identity: identity.workload,
                            instance: request.instance, role:
request.role, expires_at: expiresAt })
    try:
        auditLogAppend({ action: "issue", subject: login,
                         actor: "credential-broker", lease: leaseId })
    on AuditLogUnavailable:
        markLeaseVoid(leaseId)
        raise IssueRefused
    // a failure here leaves a lease with no login; dropping a missing
login is harmless
    createLogin(login, password: password, valid_until: expiresAt)
    grant(request.role, to: login)
    return { login: login, password: password, expires_at: expiresAt }
```

租约在登录存在之前写入，审计条目在登录创建之前写入。如果数据库在租约记录之后失败，清扫器会发现一份没有登录的租约，并把它视为已经撤销。如果审计日志不可用，就根本不会签发任何凭据。这是一个有意的选择，意味着审计日志不可达会使新实例无法启动，这一成本在第 14 章中会从另一侧讨论。

到期与撤销

到期只会让新的连接无法通过认证。已经打开的会话在密码到期之后仍会继续运行，因此到期包含两部分：使登录失效，并关闭其上已打开的会话。这也是轮换与**撤销**的区别所在，撤销是让凭据或授予立即停止生效。轮换是替换，撤销是终结。

```
function revokeExpiredLeases():
    for each lease in expiredLeases(before: now()):
        try:
            closeSessions(login: lease.login)
            dropLogin(lease.login)
        on LoginNotFound:
            // the login is already gone, which is the goal
        on RevokeFailed:
            // an expired credential that still works is an incident
            notify("platform", "expired login still active: " +
lease.login)
            continue
        markLeaseRevoked(lease.id)
        try:
            auditLogAppend({ action: "revoke", actor: "lease-sweeper",
                             subject: lease.login, lease: lease.id })
        on AuditLogUnavailable:
            // the revoke is done; the log catches up from the retry
queue
            queueForRetry(lease.id)

function checkOverdueLeases():
    // scheduled separately, so one failing job cannot hide the other
    if count(expiredLeases(before: now() - graceMinutes)) > 0:
        notify("platform", "leases past expiry and not revoked")
    if now() - lastSuccessfulRun("revokeExpiredLeases") > maxRunAge:
        notify("platform", "lease sweeper has not completed recently")
```

续期遵循同样的纪律。运行中的实例在租约结束之前请求 broker 延长租约，只有在租约仍然有效且总有效期低于上限时，broker 才会同意。没有上限，续期就会把短期凭据变成一份碰巧被反复重新签发的长期凭据。有效期本身需要判断。我不知道适合某个具体系统的数字，它取决于三件事：一个实例在两次重启之间运行多久，每个实例持有多少连接，以及 broker 在负载下完成一次续期要多久。仅作示意，可以设想一小时的租约，每三十分钟续期一次。

相比 A/B 轮换，它的优势在于泄露的凭据会自行到期。登录名指明了实例，因为它就是为该实例创建的，所以数据库日志能把“这条语句是谁执行的？”回答到具体的工作负载。这兑现了第 3 章的承诺：登录对应到一个工作负载实例。

Workload identity 与基于 IAM 的数据库认证

有一种办法可以彻底去掉存储的密码。某些平台允许数据库信任平台自己的身份系统：工作负载向平台请求一个短期 token，数据库接受该 token 来代替密码。云服务商以 IAM 数据库认证之类的名称提供这种能力，有

些数据库也能验证来自外部身份提供方的 token。

它的吸引力在于没有需要存储、获取或轮换的机密。token 在需要时为工作负载签发，有效期很短（通常是几分钟），并绑定到平台已经知道的身份。撤销是平台的工作：取消该身份请求 token 的权利，就能阻止新 token，而已经签发的 token 在到期前可能仍被接受。数据库中的登录依然存在，依然是其 `app_*` 角色的成员，依然会出现在日志里。

它的代价是具体的：

- **平台耦合**。这个设计在平台支持的地方行得通，别处则不行，因此在两个平台上都有工作负载的团队，要把这个模式构建两遍。
- **映射**。必须有人说明哪个平台身份对应哪个登录和角色。这份映射就是一次授予，值得和其他授予同样的评审（第 14 章）。
- **已打开的连接**。在常见的引擎上，token 是在连接打开时检查的。与任何到期一样，已打开的连接会比它的 token 活得更久（见“到期与撤销”），因此仍然需要对连接的存续时间设置上限（第 5 章）。

三种做法中该用哪一种，与其说取决于潮流，不如说取决于平台提供什么以及团队能运维什么。

做法	是否存储机密？	有效期	需要	主要成本
机密管理器中的静态凭据，A/B 轮换	是，每个服务一对	直到下一次轮换	机密管理器、轮换作业	使用方必须重新读取；轮换作业必须持续运行
来自 broker 的动态凭据	仅 broker 自己的	几分钟到几小时，按实例	broker、`ledgerly_access_admin`、租约清扫器	broker 是高价值、必须始终可用的组件
Workload identity token	数据库侧没有	短期，通常几分钟	平台与数据库的支持	平台耦合；token 只在连接时检查

这些做法都没有消除凭据问题，每一种只是把问题挪了个位置。broker 自己的登录是一份静态凭据，存放在机密管理器中，由 broker 的 workload identity 获取，并按 A/B 轮换。token 方案则依赖平台的身份系统是正确的。变化的是机密有多少个，每个机密有用多久，以及谁能读取它们。

TLS 与证书验证

上面的一切都关乎凭据存放在哪里以及能持续多久。凭据在到达数据库之前要穿过网络，而未加密的连接会以路径上任何东西都能读取的形式，携带着密码和数据。传输加密堵住了这一点，它是基线。它必须在每一段链路上都成立。请求在到达数据库之前，往往要经过 pooler 或 gateway，每一段都是单独的连接，有各自的加密和各自的证书检查。对客户端到 pooler 这一段加密，并不能说明 pooler 到数据库那一段的情况。

还有第二项任务更容易被跳过：确认服务器是对的那一台。一个连接可能被加密到一个并非数据库的目的地。**证书验证**是指客户端检查服务器的证书由其信任的机构签发，且证书上写明的主机就是它想要连接的那一台。没有它，加密能防止有人读取流量，却防不住有人应答这个连接。

起作用的设置是客户端的验证模式。多数数据库客户端提供一种只加密不验证的模式，它之所以流行，是因为它能配合任何证书使用，包括已过期的和自签名的。我认为这种模式只做了一半：它向被动观察者隐藏了流量，却带来服务器是对的那一台的错觉。补救办法是把证书颁发机构分发给每个服务，并验证名称。

这又带来它自己的成本。证书会过期，服务器证书过期就是一次带着确切日期的中断，所以到期需要监控，续期流程需要演练。把机构证书分发给每个使用方是一项工作，机构变更时还得更新。客户端证书（即证书本身就是凭据）要想在短有效期下使用，必须有东西自动签发它们。

凭据泄露时

即使有了上述所有控制，凭据仍可能泄露，设计应当预先说明接下来会发生什么。第一步是把恐慌会混为一谈的两个问题分开：如何让这份凭据停止生效，以及它在生效期间做了什么？

对于动态凭据，第一个答案是撤销：终止租约，关闭会话，登录会在 broker 采取行动所需的时间内消失。对于 A/B 设置中的静态凭据，怀疑泄露并不是计划内的轮换。计划内的作业会等旧登录消化完毕，而泄露的值只要还有人持有它的会话，就一直有效。发生泄露时，应先切断泄露的登录：移除它的角色，关闭它的会话，不等消化就更改它的密码，然后向闲置的登录发布新值，并切换活动指针。仍然持有旧登录的实例会失去连接并出现故障，直到重新读取机密。这是可用性上的代价，我愿意付出：继续生效的泄露凭据才是更大的问题。无法重新读取机密的服务需要重

启，团队应当在事到临头之前就知道是哪些服务。对于初始状态下共享的管理员密码，第一个答案就是本章开头的那场中断，这正是在事故发生之前就做好这项工作的最有力的理由。

第二个问题由日志来回答。对应到一个工作负载实例的登录，能让人看到它执行过的语句、触及过的表以及活跃的时段。这把评审范围缩小到一个角色的影响范围，而不是整个数据库。具体的处理步骤、行动的先后顺序以及如何演练，属于第 15 章，该章讲事故中的访问。这里的要点是，把撤销建成例行步骤的团队，可以在糟糕的日子里直接使用它，而不必临时发明。

数据保护和支付卡方面的合规制度，往往会询问访问是如何授予的、持续多久、如何撤销。这里描述的租约、撤销步骤和审计条目，就是回答这些问题的记录。

Ledgerly 的迁移计划

从共享管理员密码迁移到短期凭据，是一连串小而可逆的步骤。我会按每一步消除多少风险、又最不容易破坏什么来排序。

1. **盘点管理员密码的每一份副本**。服务、流水线变量、导出步骤、密码管理器、客户端工具中保存的连接。这份清单就是工作计划。第 1 章的清单是起点。
2. **为每个服务创建一个登录，归入其 `app_*` 角色**。使用第 3 章的角色。在这一阶段凭据仍是静态的，但每一份都是独立的，存放在机密管理器中，并按身份读取。
3. **逐个把服务从管理员登录迁走**。从影响范围最小的 `reporting-job` 开始，然后是 `invoice-worker`、`admin-console` 和 `billing-api`。最后处理 `payments-service`，因为它是唯一访问 `payment_tokens` 和 `bank_accounts` 的服务，那里出错的代价最大。每一次迁移都是一次可以回滚的部署。
4. **给迁移和导出步骤各自的登录**。流水线使用 `ledgerly_migrator` 中的短期登录（第 11 章）；该角色本身持有常驻的所有者权限，只有凭据是短期的。导出步骤使用 `app_vendor_export`（第 13 章）。这一步之后，两者都不再使用管理员登录。
5. **把每个服务切换为 A/B 轮换**。让每个服务在认证失败后重新读取机密，在安静时段运行第一次轮换，并观察消化检查。不消化的服务就是一个发现项，而这个发现项正是这一步的意义所在。

6. **为易于续期的服务引入 broker**。从经常重启且持有连接很少的服务开始。随着团队了解租约的行为，再加入其他服务。保留静态 A/B 作为 fallback。
7. **退役共享的管理员登录**。把它从所有配置和密码管理器中移除。所有权移交给 `ledgerly_owner` 之后（第 3 章），剥夺它的权限，然后删除或禁用它。如果这个管理员恰好是引擎的超级用户，托管服务的主用户往往如此，它就成为第 3 章所说的封存的凭证。这是一份日常工作中无人使用的凭据。它被保管在平台团队可以取回的地方，每次取回都有记录。待整份清单清理完毕之后，再更换一次它的凭据。

团队即使在第五步之后停下，处境也已好得多：每个服务都有自己的凭据，通过身份读取它，并且轮换时不会中断。broker 是为能够运维它的团队准备的补充。

权衡与成本

本章的每一步都把成本从一次事故转移到一项例行工作。下面是这项例行工作变得昂贵的地方。

对机密管理器的依赖。一旦凭据来自机密管理器，机密管理器不可达就会使新实例无法启动，在租约很短的情况下，最终还会使运行中的实例无法续期。这让它成为数据库路径可用性的一部分。运行它时应当给予对待数据库同样的用心：不止一个实例、经过测试的恢复以及一个知道它存在的值班轮换。把凭据缓存一个租约的时长，能缓解短暂的中断，同时也延长了已撤销凭据仍可被使用的时间。

连接风暴。如果一个服务的所有实例在同一分钟续期，或在部署后一起重启，数据库就会看到一阵突发的新连接和认证。认证不是免费的，自己造成的突发会拖慢数据库上的所有其他角色，这是没有任何外部原因的连接风暴。用抖动错开续期，让续期之间的连接池保持稳定，并由第 5 章来规定一个角色可以持有多少连接。

调试更困难。只存在一小时的登录，在第二天早上有人读日志时已经不在了，而像 `billing-api-4f2a` 这样的名称，没有租约记录就毫无意义。租约及其身份的保留时间应与审计日志的保留时间一致，并给值班人员一个从登录查到实例的查询途径。如果缺少这种查询，短期凭据只会让调查更慢，而不是更快。

高价值的 broker。broker 的登录可以在每个应用角色中创建登录，因此值得攻击者下功夫。让它的代码保持精简并经过评审，并给予它与它所保护的东西同样的关注；得到的关注较少的 broker 本身就是一种风险。

何时不要使用。只有一个服务、少数几位工程师、并有托管机密存储的团队，也许只需一份单独的静态凭据，存放在机密管理器中，每年手工按 A/B 轮换两次就够了。broker 是又一个要运行的服务，是团队要学习的又一个系统；低于某个规模，增加的活动部件引发的中断会多于它们预防的。如果说不出 broker 宕机时该呼叫谁，我会等一等。平台提供时，workload identity token 是成本最低的一步，而且仍需要评审映射。

常见错误

- **把机密管理器当作全部答案**。这很诱人，因为这一步显而易见且迅速。如果每个服务都读取同一条路径，共享密码仍然是共享的。应当给每个服务各自的路径和各自的登录，并评审谁能读取每一个。
- **无人使用的轮换**。一个作业轮换了值，但服务只在启动时读取而不再读取，于是新值无人读取地放着，直到旧值被退役、某处出了故障。应当测试使用方是否会采用新的轮换值，并把消化检查纳入作业。
- **退回到固化的值**。镜像里一份“最后已知良好”的副本让启动变得可靠，却带回了这个设计已经移除的那个文件。应当拒绝启动、告警并修复依赖。
- **只加密而不验证**。开启 TLS 满足了检查清单，而只加密的模式让连接能够工作。应当验证证书和主机名，并监控到期日期。
- **没有上限的有效期**。不设限的续期会重建出长期凭据。应当为租约的总有效期设置上限，并让实例用它的身份重新认证。
- **把 broker 的角色当作小角色**。`ledgerly_access_admin` 比 broker 的工作宽得多，所以被接管的 broker 能到达它所管理的各个角色那么远。应当限制哪些 principal 可以调用它，以及每个 principal 可以请求哪些 `app_*` 角色，并评审它的审计条目。应把这个宽角色视为已接受的成本，而不是一道界限。

要点

- 凭据的每一份副本都是一个读取者，因此最有用的改变是减少副本的数量，并让剩下的每一次读取都有名字。

- 轮换之所以困难，是因为凭据是共享状态；每个服务两个登录，使得在另一个继续工作的同时修改其中一个成为可能。
- 使用方不采用的轮换，或无人运行的轮换，只是一个日历条目而不是控制措施，因此消化检查，以及对已停止运行的轮换作业的告警，属于设计的一部分。
- 动态凭据让每个工作负载实例拥有自己的登录，并内置终点，代价是一个 broker、一份租约记录以及对机密管理器的依赖。
- 在平台支持的地方，workload identity 去除了存储的密码，并把信任转移到平台以及身份与登录之间的映射上。
- 验证服务器的证书和名称，才使 TLS 真正保护凭据；没有验证的加密只是隐藏了流量，而服务器的身份仍未经检查。
- 凭据泄露时，撤销的速度和日志的清晰程度决定事件被控制到什么程度，而这取决于泄露之前所做的工作。

凭据决定谁能进入；至于它们会打开多少连接，或一条语句可以占用主库多久，凭据则不置一词。第 5 章把连接本身变成策略：按角色的限制、连接池化、超时以及便于找出失控查询的标签。

第 5 章 — 将连接作为策略

设想 Ledgerly 一个安静的周二夜晚。一位队友调高了 `reporting-job` 夜间聚合的并行度，好让它在早晨之前跑完，这个改动在评审时看起来无伤大雅。01:40 作业启动，为每个工作线程打开一个连接，并开始使用共享的 `admin` 登录从主库读取 `payments` 和 `invoices`。几分钟之内，`billing-api` 开始向一个已是早晨的时区的客户返回错误。值班的 Robin 因一次面向客户的中断而被呼叫。

Robin 打开数据库，发现什么都没有坏。主库在运行，磁盘活动正常，没有任何一条语句做错了事。服务器的连接槽位数量是固定的，报表占去了大部分，API 正在门口等待。由于所有调用方使用同一个登录，数据库无法说出报表就是原因，只能说槽位满了。

本章回答一个介于身份与数据之间的问题：如何让连接本身成为应用策略的地方？在第 1 章的五部分模型中，它作用于两个部分。连接是路径，因为 pooler 决定服务如何到达，以及允许多少个到达。它也是对执行的第一道界限，因为限制和超时规定了一个调用方、一条语句可以从其他所有人那里占用多少。示例主线中的这一步是：连接由 pooler 以及按角色设置的限制和超时来塑造。

问题及其难点

连接首先是一种资源，然后才是别的东西。在大多数引擎上，每个连接都要消耗服务器的内存和一份调度开销，而服务器只允许固定数量的连接。这个上限是服务器的属性，而不是调用方的属性，所以第一个要求很多连接的调用方就能得到很多。默认配置中没有任何东西规定，夜间报表应当比面向客户的 API 获得更少的槽位。

密码与这一切无关。它决定谁能进入；至于他们占用多少槽位、占用多久，它只字未提。这就是为什么一本不止讲分发凭据的书，需要有一章讲连接。

三个习惯使问题更糟。第一，连接数是乘积，而不是总和：一个服务的实例数乘以每个实例连接池的大小，每当自动扩缩容组件增加一个实例时又要再乘一次。假设一个团队给一个会扩展到十个实例的服务设置了二十个连接的连接池。它就是在请求两百个连接，而且多半并非有意为之。第二，连接在其语句运行期间一直被占用，所以一条慢查询会全程占着一个槽位，而槽位正是下一个调用方需要的。第三，出了故障的调用方往往会更加用力地重试。一个超时后立即重试的客户端，会把一个缓慢的数据库变成门口的一群人。

显而易见的应对是提高服务器上限。这样做能管一周，然后让下一次事故更大。更多的连接意味着同一个主库上有更多并发工作，而数据库并不会因为允许一百条语句同时运行，就能更快地同时运行一百条语句。

这些控制分散在没有人统一负责的多个位置：角色定义、应用的连接池、可能存在的 pooler 以及决定一条语句可以运行多久的设置。可用性是第 2 章威胁模型中处于风险之中的属性之一，而意外在其中占主导。共享登录会把每一次意外都变成所有人的意外。

因此，在初始状态下，全部五个服务和每一位通过 VPN 接入的工程师，都在同一个名字下从同一个服务器槽位池中取用。除了各客户端库的默认值之外没有任何超时，也没有记录说明哪个调用方持有哪个连接。接下来的各节将用以下内容取而代之：按角色的预算、强制执行预算的 pooler、为任何东西占用槽位的时间设定上限的超时以及每个连接上说明归属者的标签。

按角色的连接预算

第一步成本最低。第 3 章给每个服务分配了各自的应用角色，这意味着限制终于可以附着在一个名字上。按角色的限制是在说：这个 principal 最多可以持有这么多连接，无论它运行多少实例，也无论它请求得多么用力。当它达到上限时，来自该角色的新连接会被拒绝，而其他所有角色继续工作。

这个拒绝就是策略。它把故障从“数据库对所有人都满了”转为“这个调用方没有槽位了”，而后者是 Robin 能收到的好得多的呼叫。贪婪调用方的影响范围缩小为调用方自身，并且拒绝会指出是哪个调用方，这就是同一步骤中问责的那一半。

数字需要推理，而不是公式。我会从服务器的上限向下推算。首先减去一份其他任何东西都不得使用的储备，因为修复系统的人和工具也需要连接。这份储备涵盖平台会话，以及使用 `ledgerly_access_admin` 的三个 principal 的登录（应用角色定义的流水线步骤、凭据 broker 和 access gateway）。然后把其余部分按各应用角色在最忙时的需求加上一些余地，分配给它们。下表是针对主库的示意。这些数字是假设的，并非建议。

Principal	角色	连接数上限（示意）	为何是这个大小
`billing-api`	`app_billing_api`	80	最繁忙的服务；为 Web 应用和公共 API 提供服务
`invoice-worker`	`app_invoice_worker`	30	由队列驱动；并发度由 worker 数量决定
`payments-service`	`app_payments_service`	30	平稳；在语句之间与外部服务商通信
`admin-console`	`app_admin_console`	15	少数人员执行的、范围狭窄且受审计的操作
迁移	`ledgerly_migrator`	3	一次一个迁移，来自流水线
主库上的人员会话	JIT 和 break-glass 类型的会话登录，由 gateway 创建	8	只有带已批准申请的会话才在这里运行（第 6 章）；常规读取转到 `ledgerly_replica_2`。角色 `human_jit_write` 和 `human_breakglass` 不带上限，因为没有任何登录是前者的成员，而后者是按会话授予的
`reporting-job`	`app_reporting_job`	10	仅限过渡期：它在初始状态下以及迁移期间读取主库，到目标状态时这一行归零
储备	访问管理与平台	10	仅用于修复和签发登录
未分配	无	14	余量，使一次舍入误差不至于成为一次中断

假设服务器允许 200 个连接。已分配的各行合计 186，其余 14 个被有意留作未分配。常规的人员会话不在这张表里，因为它们不在主库上运行。它们的预算设在 `ledgerly_replica_2` 上，常规读取转到那里；假设那里是 20 个槽位。`reporting-job` 这一行只在报表仍然读取主库时

存在。在目标状态下，它读取 `ledgerly_replica_2`，并在那台主机上有自己的预算，在 pooler 的连接层设置（第 6 章讲述这一迁移）。角色定义可以用类似下面的记法携带这些限制，该记法仅为示意，不属于任何引擎：

```
// illustrative notation: the limit belongs to the role, not to a
person
ROLE LIMIT app_billing_api      CONNECTIONS 80;
ROLE LIMIT app_invoice_worker   CONNECTIONS 30;
ROLE LIMIT app_payments_service CONNECTIONS 30;
ROLE LIMIT app_admin_console    CONNECTIONS 15;
ROLE LIMIT ledgerly_migrator    CONNECTIONS 3;
ROLE LIMIT app_reporting_job    CONNECTIONS 10;
```

> **注意**：不同引擎中，限制或设置附着的对象各不相同。有的按角色统计连接数，有的按登录，有的只按数据库或按服务器。在某些引擎上，写在角色上的限制或默认值只适用于以该角色本身登录的连接，作为其成员的登录不会继承。第 4 章给每个服务实例各自的动态登录，是 `app_*` 角色的成员，所以在这些引擎上，角色层面的限制和下面角色层面的超时会悄无声息地不生效。

有两个地方可以改为承载它们：broker，它在创建每个登录时可以设置按登录的限制和相同的设置；以及 pooler，它可以承载这些设置，但未必能把一个角色的各个登录加总成一个总数。下面关于 pooler 的一节列出了在 pooler 做不到时获得角色总数的三种办法。

有两条性质使这张表有用。限制在连接开始的地方检查，所以调用方会立即得知被拒绝，而不是在第一条慢语句之后。储备位于每个应用角色之外，所以在最糟糕的夜晚，平台团队仍然能登录进去修复问题。一份把整台服务器都占满的预算，与共享管理员密码有同样的缺陷：直到有人最需要它的那一刻之前，它都有效。

加总不必保守。如果 `billing-api` 和 `payments-service` 很少同时达到峰值，可以让应用限制加起来超过服务器允许的数量，并接受两个峰值同时出现时可能发生冲突。我只会在保留储备的前提下这样做，并且只在团队观察了一段时间实际使用情况之后。默认做法是预算加总得上，因为在凌晨三点，一份不会被违反的预算更容易推理。

连接池化：连接在哪里共享

连接池保持一组打开的连接并把它们借给调用方，这样每个请求不必为新连接付出代价，数据库看到的连接数也有界。放置连接池的地方有两处。

几乎每个服务的数据库库里都已经带有连接池，其大小应当依据一个实例一次能完成的工作来确定，而不是依据它希望服务的流量。让调用方短暂等待的小连接池，对数据库比大连接池更友善，并且它应当在短暂等待之后放弃，返回一个服务能转化为明确响应的错误。从小处开始，观察调用方等待多久，并依据证据调大。

数据库前的 pooler

pooler 是一种数据库 proxy，其主要工作是接受许多客户端连接，并把它们多路复用到数量更少的服务器连接上。PgBouncer 和 ProxySQL 是常见的例子。服务像连接数据库一样连接 pooler，由 pooler 决定何时借出一个真实连接。

本书不断追问的那个问题在这里同样适用：数据库看到的是谁的名字？第 4 章给每个服务实例各自的动态登录，由凭据 broker 创建，是该服务 `app_*` 角色的成员。pooler 必须契合这个模型，我会按这样的形态来构建。实例向 pooler 出示自己的登录和凭据。pooler 针对数据库验证它们，并在同一个登录下保持服务器连接。它自己不持有密码存储，也不持有一个所有客户端共享的、属于自己的登录。

被轮换或已到期的凭据通过数据库而不是通过 pooler 到达 pooler。租约结束或被撤销时，broker 会删除该登录（第 4 章），因此 pooler 下一次尝试在该登录下认证或打开服务器连接时会被拒绝，并且它会关闭为此登录持有的空闲服务器连接。

按角色的限制如何仍然成立，没有通用的答案，我也不会假定所用的 pooler 能给出答案。许多 pooler 按“登录与数据库”的组合保存限制，有的还增加按登录或按数据库的限制。我不知道有什么通用能力能把许多不同登录的连接加总成一个角色层面的上限，所以应当阅读该 pooler 的文档，而不是轻信本章。如果它做不到，角色总数就得从别处获得。有三个地方，各有代价。

- **pooler 为每个角色用一个稳定的登录向上游连接。**实例的动态凭据于是只用来向 pooler 认证客户端，而 pooler 按登录的限制恰好就是该角色的限制。代价是第 3 章的身份模型：

    - 数据库日志里每个角色只显示一个名字，因此对实例的归属转移到 pooler 的日志和下文所述的应用标签上。
    - pooler 必须自行验证动态凭据，这使它成为 broker 租约状态的使用方。
    - 它还要为每个角色持有一份长期凭据，需要保管，并按第 4 章所述进行 A/B 轮换。
    - 在 broker 处撤销某个实例的登录，不再会关闭它的服务器连接；必须由 pooler 终止该客户端的会话。

- **数据库在引擎对成员登录支持的情况下，强制执行按角色的限制**。它是精确的，位于无法绕过的地方，实例登录及其归属保持完整。代价是支持程度因引擎而异，如上面的注意事项所述。在它可用的地方，拒绝来自数据库，此时连接已经付出了一次认证的代价，并且没有任何多路复用。
- **broker 限制每个角色签发的登录数量**。在创建每个登录时设置按登录的限制，则角色的总数不会超过登录数乘以该限制。代价是这只是一个上界而不是精确计数，并且这个上限必须与自动扩缩容的实例上限相匹配。超出上限的扩容会变成一次拿不到凭据的部署，这是可用性问题；而旧登录与新登录短暂重叠的续期，也需要在上限之下留出余地。

在引擎对成员登录有效支持数据库侧限制的情况下，我会先尝试数据库侧的限制。在它不支持的地方，broker 上限保留了身份模型，代价是要维护一个上限。只有当连接数迫使跨实例多路复用时，我才会采用每个角色一个上游登录的做法，并且会写明日志从此不再说明的内容。这三种都比把每个角色的每个客户端映射到一个共享登录的设计要好。后者在更下一层重建了共享管理员账号：日志再次只显示一个名字，一个限制同时适用于所有人。

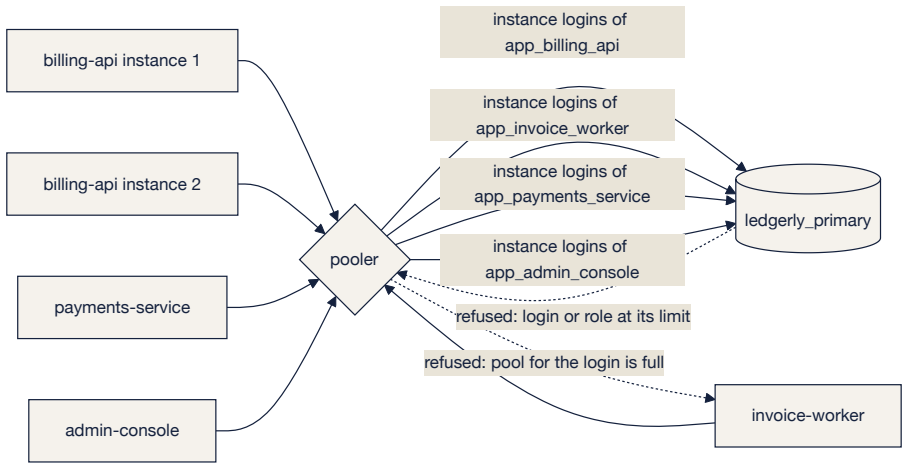
在所推荐的透传形态中，pooler 带来的多路复用很少。在事务模式下，同一个实例的各个客户端共享该实例的服务器连接，但两个实例之间互不共享，所以一个角色的总数至少是存活实例数乘以一个很小的最小值。pooler 带来的是另外三样东西。

- **集中拒绝**。由一处说不，按登录、按连接池，对每个客户端库都以相同方式进行。
- **对每个服务相同的限制**。各服务中的连接池大小和等待时间不再是唯一的防线。

- **网络咽喉点**。服务只能通过 pooler 进入数据库。网络策略规定哪些主机可以打开 TCP 连接，由第 6 章定义、第 7 章编写，它只准许 pooler 的主机，别无他人，因此从某个服务泄露的凭据，在 pooler 不可达的任何地方都毫无用处。如第 4 章所述，触达范围限定了泄露的影响，而 pooler 正是这一界限易于表述的地方。

代价是路径上多了一个新系统，下面的权衡中会谈到。对于只有一个实例的单个小服务，应用的连接池加上角色限制就够了，在实例数或对单一咽喉点的需要另有说法之前，不值得加入 pooler。

下图展示了透传形态，它让数据库仍能看到实例登录。请留意调用方被拒绝的两处：在 pooler 处，当某一登录的连接池已满；以及在数据库处，当某个登录（或在引擎支持时某个角色）达到其上限。



会话模式与事务模式

pooler 在一段时间内借出一个服务器连接，这段时间的长度就是它的**模式**。在会话模式下，客户端从连接到断开始终保持同一个服务器连接；pooler 除了省下连接的开销之外，什么也没有节省。在事务模式下，客户端只在一个事务的时长内持有服务器连接，下一个事务可能运行在另一个连接上。事务模式能把多得多的客户端多路复用到很少的连接上，这正是它的吸引力所在。

它带来一项容易被忽略的代价，因为这关系到授权而不是性能。客户端在会话上设置的任何内容都存在于一个服务器连接上，而在事务模式下，客户端并不拥有这样一个连接。在事务之外所作的设置，可能在下一条语句

时消失，或者更糟，可能留在 pooler 借给别人的连接上。依赖会话状态的东西包括临时表、许多配置下的预处理语句、咨询锁、会话级超时以及自己的设计存放在会话中的任何上下文值。

最后一项直接触及本书。在第 3 章中，行规则可能依赖诸如 `current_case_customer()` 这样的值，而 `app_admin_console` 会把人员 principal 的身份带入审计日志和 `audit_events`。如果任何一个值在每个会话中设置一次、之后再读取，那么在事务模式下它可能缺失，或者在糟糕的情形下属于前一个客户端。

> **警告**：在事务模式下，会话级设置的上下文值可能留在服务器连接上，并被下一个客户端的事务读取。读取它的行规则或审计记录就可能为错误的客户或错误的人执行操作。应当在事务内部设置这类值，并将其作用域限于该事务，并让读取它们的代码在值缺失时失败。

经得起检验的模式是：在事务内部设置上下文，并让它随事务一起结束。下面的代码块展示了这种形态，其写法确保在工作中途失败时仍然正确。

```
function runAsPrincipal(principal, customerId, work):
    transaction:
        // transaction-scoped: it disappears at commit or rollback, so
        // the next client cannot inherit it from this connection
        setLocalContext("acting_principal", principal.id)
        setLocalContext("case_customer", customerId)
        try:
            work()
        on ContextMissing:
            // a row rule or audit write found no value: stop, do not
guess
            raise ActionRefused
        // the commit happens here, once, after all of the work
succeeded
    // on any error the transaction rolls back, and the context goes
with it
```

数据库日志仍然显示实例的登录，但在事务模式下，一个实例的各个客户端共享它的服务器连接，所以日志无法说明该实例的哪个线程或哪个请求占用了某个槽位。要查明这一点，需要 pooler 自己对其客户端的视图，并且必须在第一次事故之前就做好规划。这就是接下来关于超时和命名的各节很重要的原因。

人员不使用这个 pooler。他们通过 access gateway 到达数据库，gateway 为每个会话创建一个登录（第 7 章）。他们的限制是下一节最后一部分的主题，因为人可以做服务做不到的一件事：解除限制。

将超时作为策略

槽位在一条语句或一个事务运行期间一直被借出，而在大多数引擎上，默认没有任何东西会结束其中任何一个。超时是平台团队规定一个角色可以占用某项资源多久的方式，也是本章中界定执行的那一部分：运行过久的语句会在运行之处被取消，槽位随之归还。超时属于角色定义和连接池配置，而不取决于各客户端库的自觉，这就是本章把它们当作策略的原因。

三个超时属于数据库，两个属于连接池。

超时	归属	它阻止什么	谁会感受到
语句	数据库	运行时间远超该角色所需的查询	该语句的调用方
锁等待	数据库	在锁后面排队的语句，以及排在它后面的所有人	调用方；以及在同一行上等待的任何调用方
事务内空闲	数据库	打开后被搁置的事务，它占着锁和槽位	打开它的应用
连接存续时间	连接池	保持打开数天并保留旧凭据或旧状态的连接	连接池，它必须重新连接
连接池获取等待	连接池	无限期等待空闲连接的调用方	处理该请求的服务

不同的角色需要不同的值，因为它们做的工作不同。`billing-api` 中的一个请求是一个人在等一个页面；在那里需要三十秒的语句就是一个 bug。`invoice-worker` 在构建批次时运行较长的语句可能是合理的。`reporting-job` 按设计运行长时间读取，且是在副本上，长时间读取不会阻塞写入方。共享的默认值对它们全都不对，所以这些值附着在角色上。下面的代码块使用示意记法，数值是假设的。

```
// illustrative notation: defaults for the role; see the note above on
// members, because the broker or the pooler may have to apply them
per login
ROLE SETTING app_billing_api      statement_timeout 5s,  lock_wait 2s,
idle_in_transaction 10s;
ROLE SETTING app_invoice_worker   statement_timeout 30s, lock_wait 5s,
idle_in_transaction 30s;
ROLE SETTING app_reporting_job    statement_timeout 15m, lock_wait 5s,
idle_in_transaction 60s;
ROLE SETTING human_readonly       statement_timeout 60s, lock_wait 5s,
idle_in_transaction 5m;
ROLE SETTING ledgerly_migrator    statement_timeout none, lock_wait
3s, idle_in_transaction 60s;
```

最后一行反映了第 11 章讲述的一条迁移经验：模式变更可以运行很久，但等待锁只应等很短时间。

关于顺序的两条规则比单个数值更重要。数据库的语句超时应当短于应用的请求截止时间，因为如果应用先放弃，它就丢下了一条仍在运行的语句，槽位依旧被占着；而连接池的获取等待应当是所有超时中最短的，让调用方尽早得知拥塞。当截止时间确实先触发时，连接上可能仍带着一条正在运行的语句，所以服务要关闭它，而不是把它借给下一个调用方。

事务内空闲的情况出人意料。一个服务打开事务，调用外部支付服务商，并在事务仍然打开时等待，就会让服务商的缓慢拖住数据库上的锁和一个槽位。长期的修复办法是把外部调用放在事务之外，这是一项代码变更；在这项变更还在待办列表上时，超时让成本保持有界。

一个存活数周的连接会保留它出生时带来的一切，所以设置最大存续时间，让连接池定期替换其连接。它应当短于凭据的有效期，并且各连接之间有一些差异，使它们不会在同一分钟全部重新连接。第 4 章解释了为什么已到期的凭据不会关闭一个已经打开的会话；存续时间就是连接池关闭自己的连接的办法。

会话可以解除的默认值

作为角色默认值设置的超时，只是一个默认值。在许多引擎上，会话可以更改自己的设置，包括它的语句超时和只读模式，还可以切换角色。对服务而言这可以接受：这些限制是为意外而设的，而代码不会意外地解除自己的超时。对使用 SQL 客户端的人则不行，因为需要受限制的会话，恰恰就是能解除限制的会话。

有两项控制位于会话够不到的地方。gateway 在打开每个人员会话时就应用其限制，并拒绝更改受保护列表上设置的请求；第 9 章讲述 gateway 如何做到这一点。代价是，在连接时发送这类设置的客户端会停止工作，直到有人把它们列入清单。人员会话的限制来自 gateway：它在创建会话登录时，按会话类型设置按登录的连接数上限，并统计一个人可以持有多少会话。这两者都不是角色限制，因为 `human_jit_write` 是一个没有任何登录属于其成员的上限。两者都不留给客户端。对于这两者都不具备的任何限制，控制措施表应当说明它是默认值而不是保证。

为调用方命名

Robin 查看连接时，有用的问题是“这是谁的？”，而登录回答了前一半。角色说明某个连接属于 `billing-api`。它没有说明是哪个实例、哪个版本，或者某个定时作业的哪一次运行。大多数引擎和 pooler 允许客户端给其连接附加一个简短的标签，通常称为应用名称，它会出现在数据库日志和当前活动的视图中。

一套约定比功能本身更有价值。我会在标签中放三样东西：服务、实例或运行以及版本。例如 `reporting-job/run-0412/v31` 说明了是哪个作业、哪一晚的运行以及哪个构建。标签把“有人占着四十个连接”变成了“周二那次报表运行占着四十个连接，而且是新的构建”。

随之而来有两点提醒。第一，标签由客户端提供，可以设置成任何内容，所以它用于诊断和归属，而不是用于授权。登录才是身份；标签只是信封上的一则备注。第二，事务模式下的 pooler 在客户端之间共享服务器连接，所以服务器一侧的标签未必是客户端的。有的 pooler 会为每个事务转发它，有的不会，此时 pooler 自己关于其客户端的日志就是携带名称的记录。在依赖它之前，要先弄清所用的配置是哪一种。

标签还会输送给第 14 章：当每一行都带有服务和运行，而不只是登录时，审计日志更容易阅读。代价是每个服务的启动中多一行代码，以及新服务出现时要多维护一项约定。

连接并不是角色能消耗的唯一资源。用于排序和连接操作的内存、临时磁盘空间、CPU 和并行 worker 的数量同样是共享的，而各引擎能否按角色限制它们，差别很大。引擎做不到的地方，隔离必须来自位置安排：重型读取方转到副本，这是第 6 章的论点。在向任何人承诺任何限制之前，先查明所用的引擎能按角色设置哪些限制。

Ledgerly：报表吃掉 API 的那一夜

回到那个周二夜晚。初始状态一句话就能解释：`reporting-job` 以 `admin` 身份对主库运行，所以它与 `billing-api` 在同一个名字下争夺同样的槽位，两者都没有限制，也没有任何超时。

Alex 的计划并不从 pooler 开始。它从成本最低、保护最多的事情开始，并随着团队不断学习沿清单向下推进。

1. **按角色的限制**。有了第 3 章的应用角色，每个角色按类似上表的表格获得一个连接上限。报表立即失去占用 API 槽位的能力，尽管第一晚它仍然读取主库。
2. **按角色的超时**。`app_billing_api` 获得较短的语句超时和较短的事务内空闲限制，报表获得一个不适用于 API 的较长语句超时。
3. **标签**。每个服务和每次定时运行都设置自己的标签。下一次呼叫会指明是哪次运行，而不只是角色。
4. **pooler**。五个服务通过它连接，它成为它们唯一的入口，角色总数取自上面三个地方之一。
5. **迁移到副本**。`reporting-job` 从 `ledgerly_replica_2` 读取，并拥有自己的预算，主库不再承担它的工作。第 6 章描述这一迁移本身。
6. **拒绝之前的告警**。平台团队监控每个连接池的饱和度，而不只是拒绝次数。一个连接池连续数小时保持在其上限的百分之九十，这是在还来得及的时候在说“提高上限，或者修复调用方”。

作业本身也需要一项改动，因为只有当作业对被拒绝能作出良好反应时，拒绝作业的限制才有用。被拒绝后立即重试的作业，就是缩小版的门口人群。

```
function acquireWithBackoff(role):
    for attempt in 1..maxAttempts:
        try: return acquireConnection(role, waitUpTo: acquireWait)
        on PoolEmpty: sleep(backoffWithJitter(attempt))  // slow down,
do not retry harder
    raise ServiceBusy
```

有了这些，同样那个提高报表并行度的改动就会有不同的结果。作业达到上限而被拒绝，它自己放慢速度、花更长时间，而 API 不会因此耗尽槽位。放弃某个批次的作业还应当留下它跳过了什么的记录，以免有东西把

构建到一半的聚合当作完整的来读取；这是作业自身的设计，超出本章范围。批准那次改动的评审仍然会是通过的，因为这个设计不再依赖评审人注意到两个服务之间的相互作用。

权衡与成本

按角色的限制需要调优。过低的限制会拒绝合法的负载，而这种拒绝在接收它的服务看来就像一次中断。限制需要负责人，并且每当服务扩容或出现新的工作负载时都要评审，因为实例数乘以连接池大小的总和，可能在任何单项变更看起来都没错的情况下越过上限。平台团队成为提出这些问题的地方，这很有用，同时也是一条新的队列。

pooler 是路径上的新系统。现在每个查询都要经过它，它一旦宕机，每个服务都宕机。在最需要它的那次事故中它必须可用，所以它需要自己的冗余、监控和 runbook。在事务模式下，它还会去掉一些代码所依赖的会话特性，并让数据库日志更难阅读。如果只有一个单实例的小服务和一个十个连接的连接池，应用自己的连接池加上角色限制就够了。

超时会取消合法的工作。设得太紧的语句超时会杀死一条正常的语句，而天真的重试会重复这项工作和这次取消。服务需要处理取消，对于写入，这意味着要考虑该操作能否运行两次而不造成伤害。`ledgerly_migrator` 和数据修正（第 10 章）的取值与应用的取值不同，这意味着要维护正确的配置更多了。

无人关注的限制什么也保护不了。长到永远不会触发的语句超时，或没有任何告警覆盖的角色限制，只是在配置里多了一行，并没有增加任何界限。检验的标准是：控制措施一旦停止工作，是否会有人注意到。

标签成本低且可伪造。它们帮助人，代价是纪律，但它们什么也证明不了，依赖它们来归属的日志，依赖的是一项约定。

何时不要使用这些控制：在角色存在之前不要添加按角色的限制，因为对共享 `admin` 角色的限制只会同时限制所有人。在知道一个服务的正常语句需要多长时间之前，不要给它设置激进的超时，因为第一次被取消的发票运行就会教会团队把超时关掉。先测量正常情况，再把限制设在它之上。

常见错误

- **用提高服务器上限来修复事故**。这很诱人，因为它立刻见效，今天也不花任何成本。它把故障转移到更大的人群，并去掉了早期信号。应当给每个角色一份预算，并让服务器上限等于它们之和。
- **按希望拥有的流量来确定连接池大小**。更多的连接让人觉得容量更大，但数据库的有效并发是固定的。应当按一个实例一次能完成的工作来确定连接池大小。
- **在 pooler 中把每个客户端映射到一个登录**。这是最省事的配置，它会带回共享管理员账号。应保留每个实例自己的登录，或明确选择每个角色一个稳定登录，并接受日志从此不再说明的内容。
- **在有会话状态的情况下使用事务模式**。代码在会话模式下能工作，所以没人在事务模式下测试它。行规则的上下文、审计用的身份和临时表必须存在于事务之内，否则该角色的模式就必须保持在会话级别。
- **每个角色使用同一套超时**。单一默认值对批处理太紧，对 API 又太松，通常的结果是它被提高以适应所有人。应当把这些值附着在角色上。
- **把应用名称当作身份**。标签由客户端设置。应使用登录来认证，使用标签来解释。
- **用应用预算把服务器填满**。当各限制加起来等于整台服务器时，平台团队在事故期间无法连接。应当在每个应用角色之外保留一份储备。

要点

- 连接是一种有限资源，也是一个策略点：属于服务器而不属于角色的限制，会让第一个贪婪的调用方拿走所有人的份额。
- 按角色的连接预算把“数据库对所有人都满了”变成“这个调用方没有槽位了”，把失控作业的影响范围缩小到作业本身。
- 在每个应用角色之外保留一份连接储备，让修复系统的人和工具始终能够连接。
- pooler 是服务进入的地方，因此是集中拒绝和网络咽喉点所在之处。许多 pooler 无法对一个角色的动态登录加总，所以角色总数来自数据库、broker 或每个角色一个上游登录，各有明确的代价，如 pooler 一节所述。
- 事务模式的连接池化去掉了会话状态，因此行规则或审计记录所依赖的任何上下文，都必须在事务内部设置，并在缺失时明确地失败。

- 语句、锁等待和事务内空闲超时属于角色，数据库的语句超时应当先于应用的截止时间触发。会话可以解除默认值，所以对人员而言，限制必须在他们无法更改的地方强制执行。
- 应用名称使连接可以被解释，但什么也证明不了，所以登录仍然是身份，标签仍然只是一则备注。

连接数上限和 pooler 决定有多少调用方到达主库，但它们并不改变落在主库上的工作量。第 6 章迈出下一步：把读取转移到副本和其他数据拷贝上，以及随每份拷贝而来的访问问题。

第 6 章 — 读取访问、副本与数据拷贝

设想 Ledgerly 的一个周二上午，离月底还有几天。数据团队的 Jordan 接到一个数字需求：上个月开出了多少张发票，按客户分层，并按此前是否已有付款尝试失败来拆分。分析副本本是询问这类问题的自然去处，但它是每晚一次的恢复，而这份恢复已经是一天前的了。这个问题需要今天的数据。Jordan 打开一个桌面客户端，用共享管理员登录连上数据库，运行了一条查询，对整个月的 `invoices`、`invoice_lines` 和 `payments` 做连接。

查询本身没有任何问题。这是一个合理的问题，出自一位称职的人之手。但连接指向的是主库，因为客户端记住的就是这台主机；而这条查询要读取好几张大表，与此同时 `billing-api` 正在为即将支付发票的客户提供服务。网页应用变慢，Robin 收到呼叫，Robin 和 Jordan 花了二十分钟才弄明白，报表和变慢其实是同一个问题。没有人做过任何不寻常的事。设计只提供了一个可读的地方，于是每一次读取都去了那里。

本章补上目标状态中的一步：读取走向副本，报表从不触碰主库。在第 1 章的五部分模型中，这一步关乎授予与路径的协同，本书中的授权正是这个意思。授予决定读取者能看到什么，路径决定读取者究竟能连到哪个数据库。本章还要处理这一步带来的后果：每一个副本，以及数据的每一份其他拷贝，都是一个需要重新设计访问的新位置。问题是：当数据被读取的地方多于被写入的地方时，哪些读取去哪里，谁可以读取每一份拷贝，又如何避免拷贝不断增殖成没人负责的门？

问题及其难点

一个既接收写入、又承担全部读取的数据库，是在用同一套资源做两份工作。写入需要主库：它只有一个，是权威来源，必须始终对写入可用。读取可以重复，可以分散，其中许多读取还能容忍几秒前的数据。把读取移出主库，保护的是写入，而保护写入就是保护账本。

显而易见的做法是增加一个副本，把读取密集的工作负载指向它。这样大约能撑一周。之后会出现三个问题，副本本身一个也解决不了。

第一个问题是，副本不是主库。在大多数部署中，复制是异步的：主库提交一次变更，副本稍后才应用它。这个间隔称为**副本延迟**。落在延迟副本上的读取，返回的是关于一个略微陈旧的世界的真实答案。对月度报表而言，这看不出来。对一位刚刚付完发票、却看到页面上仍显示“待付”的客户而言，这就是一张工单。

第二个问题是，权限会随数据一起走。物理副本是完整的拷贝，包括 Restricted 列；除非授予另有规定，凡是能读取副本的人，就能读取主库的读取者所能读取的一切。下一节会解释，为什么无法只在某一个副本上让角色变得更安全。

第三个问题是，拷贝会增殖。第一个副本供应用使用，第二个供报表使用，随后是供人工排查使用，再是供导出步骤使用。接着出现数据仓库，由定时任务或变更数据捕获供给。有人把一份快照恢复到 staging 数据库中。每一份拷贝在当时都是对某个真实的负载或便利问题的合理回应。但每一份也都存放着生产数据，带着自己的凭据和自己的网络路径，而且很少出现在任何人列出的数据入口清单上。**旁门**是指绕过主库访问控制的数据拷贝或导出路径。本章讨论的是有意建立的拷贝，第 13 章再回到那些意外出现的拷贝。

一条写着“不要在主库上运行报表”的策略，帮不了 Jordan。客户端只记得一台主机，这个登录在所有主机上都能用，而主库又永远有最新的数据。控制必须落在读取的路由方式上，以及每个登录能够连接到哪里。

拆分读取路径

我会先把读取路径视为独立的访问面，有自己的清单、自己的角色和自己的规则，并对每一次读取问一个问题：这次读取需要主库吗？当读取用来做决定时需要，例如收款之前查看余额，或者必须看到同一会话中刚刚完成的写入时也需要。其余的读取都可以去副本。服务如何对自己的读取分类和路由，属于应用设计，本书不展开。

访问规则是另一半，只有两行。报表读取和人工读取永远不回退到主库：如果副本不可用，作业就等待或失败，并通知平台团队。而且用于这些读取的登录根本无法连接主库，因此一个失误也不会把它们送到那里。一个晚了两小时才跑完的夜间聚合只是一点不便。而一个因为副本宕机就悄悄切到主库的聚合，恰恰就是初始状态的由来。

只读角色并非无害

读取通常被当作负载。它同时也是主要的机密性暴露面：离开系统的结果集不会改变任何东西，也没有什么可回滚，却可以被复制到任何地方。一个不能写入的角色，仍然可以读取被授予的每一列，它的一条重查询就足以拖住主库。“只读”还对应三种不同的机制，它们的区别在于会话能否撤销。

机制	阻止什么	会话能否撤销?
不含写权限的授予	该角色的写入	其本身不能；但能够切换到具有写权限的角色的会话，就不再受它约束
只读事务默认值	该会话中默认的写入	在许多引擎上可以：会话能够把它改回去，或开启读写事务
物理副本	该主机上的一切写入	不能：主机拒绝写入，会话也没有可以更改这一点的设置

对人员而言，副本是三者中最强的，这也是日常的人工读取要去副本的原因之一。这三种机制都没有限制的，是会话可以读取多少。会话的行数上限和导出分级属于执行，第 9 章会讲到。

授予会同步到副本

安全评审人最先问的副本问题是：“在上面能做什么？”在采用物理复制的引擎上，角色和授予存放在集群的目录中，而这份目录会随数据一起复制。角色只在主库上定义一次，就会出现在每一个副本上。副本是只读的，所以无法只在副本上创建、修改角色或授予角色。由此可知，无法只在某一个副本上让角色变得更安全：要缩小某个读取者在副本上的访问面，需要借助连接层或另一份独立拷贝。第 3 章中设计的权限在每一个副本上都有效，而某个登录可以触达哪个数据库，则是连接层的规则。

这有其有利的一面。`human_readonly` 不暴露 Restricted 列，并在第 12 章要求的位置对 Confidential 列脱敏，它在副本上这样做的原因与在主库上相同。`app_reporting_job` 在副本上读取，无法访问 Restricted 列。如果授予是按最小权限的思路构建的，副本给予读取者的就不会超出角色所允许的范围。

它也有风险的一面。在初始状态下，副本接受共享管理员登录，而该登录拥有每一张表。以该登录读取副本的人能看到每一列，暴露程度与主库相同，只是不能写入。人们往往正因如此而把副本当作安全的，结果副本恰恰成了最宽泛的访问所在之处。

让每个登录指向各自的副本

目标状态赋予每个副本一个用途，并在接受连接的地方强制执行这个用途。从这里开始，有两个术语按固定含义使用。**网络策略**规定哪些主机可以向数据库主机打开 TCP 连接；第 7 章会写出一份。**连接策略**规定哪个登录可以向哪个数据库认证，以及从哪里认证。

连接策略需要一种机制，因为授予无法承载它，网络策略又看不到登录。在有此功能的引擎上，这种机制是基于主机的访问规则，每台服务器从自己的配置中读取。该规则以连接的来源以及登录模式或角色为键，不属于被复制的数据，所以主库和每个副本可以持有不同的规则。下面的代码块是这样的一份策略，写成三台服务器各自持有的规则。

```
connection policy: one rule set per server, not replicated
    ledgerly_replica_1:
        allow from pooler hosts    login app_billing_api,
app_admin_console  // browsing, support lookups
        deny  everything else

    ledgerly_replica_2:
        allow from pooler hosts    login app_reporting_job,
app_vendor_export
        allow from gateway hosts   login pattern: kind readonly   //
human_readonly, Chapter 7
        deny  everything else

    ledgerly_primary:
        deny  from anywhere        login app_reporting_job,
app_vendor_export
        allow from gateway hosts   login pattern: kind jit, kind
breakglass
        deny  from gateway hosts   login pattern: kind readonly   //
routine reads use replica_2
```

成本在于，现在要让三套规则与角色 schema 保持同步，而落在某台服务器上、却没落在另一台上的变更就是漂移。这些规则集应当放进应用角色定义的同一条流水线中（第 3 章），并且在同一次评审中加入一项检查，把每台服务器的规则与策略进行比较。在没有此类规则的引擎上，同样的决定落到 pooler 和 gateway 上，以及放行它们的网络策略上。

`human_readonly` 与 `ledgerly_replica_2` 的搭配意味着，排查问题的人读取的，与报表使用的是同一个副本。这是一项有意的权衡：只需保护、监控和确定容量的副本只有一个，代价是人工排查要与夜间作业争用容量。如果两者开始互相干扰，为人员再增设第三个副本是合理的拆分，而连接策略只需改动一个块。

主库上的拒绝与副本上的允许同样重要。`app_reporting_job` 和 `app_vendor_export` 根本不该出现在主库上。如果它们的登录无法连到那里，初始状态中“有时读取主库”的行为，就不再是一个配置失误能够重新带回来的东西。一个指向了错误主机的作业会连接失败，这是一种可见且代价低廉的失败。

反方向的例外是 JIT 或 break-glass 会话。每个人员会话登录默认都是 `human_readonly` 的成员，所以主库的规则无法针对这个角色来写：要么会切断 break-glass，要么形同虚设。它是针对 access gateway 所开启的会话类型来写的。日常会话被送到 `ledgerly_replica_2`。需要写入的会话，或者在紧急情况下需要读取副本来不及展示的内容的会话，则开在主库上，并从该会话中提出访问申请（第 7、8、10 章描述了这一模型）。

这就在主库上留下了一个窗口：在申请获批之前，这样的会话能做的，与 `human_readonly` 相同。三条规则让这个窗口保持很小，第 7 章把它们作为会话类型的一部分写明。只有策略允许申请该类型的人，才能开启它。开启会话时要记录理由，会话有一个很短的开启窗口，申请必须在此期间跟上。如果没有申请，就触发告警。只想查看数据的人不会开启这样的会话；那是副本的用途，也是不允许在主库上进行日常读取的原因。

副本也有主机

物理副本是数据文件的拷贝，所以它的主机、卷、快照和复制凭据，不经过任何数据库角色就能触及同样的数据。我会对副本应用与主库相同的主机访问规则、存储加密和保留策略，并在创建副本时就写下来，因为之后没有人会想到这件事。第 13 章会再讨论快照和备份。

报表、分析与拷贝的选择

副本解决的是廉价而频繁的读取带来的负载问题。报表则相反：读取少见、量大、速度慢。问题在于报表应当放在副本上，还是放在别的东西上，答案取决于报表需要什么。

报表可以读取的三个位置

选项	新鲜度	主库负载	访问面	适用场景
报表副本（`ledgerly_replica_2`）	数秒到数分钟	仅复制	与主库相同的数据和角色	运营报表、导出、排查
按计划加载的独立拷贝（`ledgerly_analytics`）	与上次加载一样旧	每个计划一次加载	自己的角色和自己的数据	数据团队的模型、仪表盘、批量分析
由变更数据捕获供给的拷贝	数秒到数分钟	一个变更流的读取方	一条新流水线和一个新存储	近实时分析、多个消费者

我会把运营报表送到副本，把批量分析送到独立拷贝，并且会抵制把数据团队的工作放到副本上，理由与负载无关。副本带着生产 schema 的形态和敏感性。数据团队不需要 Restricted 列，在目标状态下也看不到原始的 Confidential 列，因为分析副本在加载时就已脱敏，只有那些本就是用来展示的显示提示除外（第 12 章描述了方法并指出了这个例外）。副本无法按消费者分别脱敏，除非改变副本的本质。独立拷贝则可以面向它的受众来构建。

这也是本书中分析副本从不被称为副本的原因。它由主库按计划加载，角色不同，内容也可能与源不同。称它为副本，会让人假定它带有同样的控制，而在初始状态下，它的控制更少。

变更数据捕获是我最后才会考虑的选项。它增加一条必须持续运行的流水线、一个必须加固的存储、自身的延迟以及生产者与消费者之间的 schema 契约。对于 Ledgerly 这样规模的公司，我会先采用定时加载，并写下什么情况会让我改变主意：例如报表需要比计划更新的数据，或者有第二个团队提出同样的需求。

每份拷贝的成本

表中的每个选项都有同一个隐藏项：一件新的需要保护的东西。拷贝是数据存放的地方，是一组能读取它的登录，是一条网络路径，是一条保留规则，还有一位负责人。没有负责人的拷贝是旁门，有负责人却没有到期日的拷贝，是更晚才会被发现的旁门。

保留一份简短的拷贝登记册会有帮助，每一份拷贝都使用相同的字段。下面的代码块是示意性的记法，要点在于 `owner` 和 `ends` 两行不能留空。

```
table data_copy:
    name            text primary key    // "ledgerly_replica_2"
    kind            text not null       // replica, analytics, export,
dump, staging
    source          text not null       // what it was made from
    contains        text not null       // data classes present:
Restricted, Confidential, ...
    readers         text not null       // roles or logins that may
read it
    owner           text not null       // a team, never a person
    ends            timestamp null      // null only for replicas and
the analytics copy
    review_due      timestamp not null
```

对 Ledgerly 而言，登记册一开始有五个条目：两个副本、分析副本、发往会计供应商的导出文件以及共享存储中的旧转储。其中四个是有意为之，转储则不是，这正是登记册有用之处：转储会得到一位负责人和一个临近的结束日期。第 13 章对存放在数据库之外的拷贝，使用同一份登记册。

> **注意**：没有人打开的登记册不是控制，所以要把它与创建拷贝的那一步绑在一起。预配步骤会拒绝创建副本、快照恢复、staging 数据库或导出作业，除非存在一行带有负责人的记录，并且在该类型要求时带有结束日期。访问评审（第 14 章）随后会问：存在哪些拷贝，每一份谁可以读取。

这种拒绝是有成本的：有人要先填好一行，赶着交付的团队才能拿到它的拷贝，最初几周会招来抱怨。它也只覆盖通过预配步骤创建的拷贝，所以手工创建的拷贝仍然是旁门，第 13 章会处理这个问题。

触达范围，而不只是授予

纸面上正确的角色，如果错误的登录能够连接，仍然是错的。对每一份拷贝而言，授予会扣留受众不需要的内容（第 3 章和第 12 章），而触达范围决定哪些网络路径和 principal 能够打开连接。在扁平网络上，任何服务上泄露的一个登录都能触及每一份拷贝，而触达范围界定了一次泄露的价值，正如第 4 章所说。下表给出各个位置的触达规则；在主机一侧强制执行的网络策略属于第 7 章，登录一侧则由上面的连接策略强制执行。

位置	可从何处触达	不得从何处触达
`ledgerly_primary`	pooler 主机、用于 JIT 和 break-glass 会话的 gateway、应用迁移的流水线步骤	报表和导出作业、日常人员会话、直接来自 Web 层
`ledgerly_replica_1`	pooler 主机，供浏览的服务使用	用于日常读取的 gateway、分析加载、直接来自 Web 层
`ledgerly_replica_2`	用于报表和导出的 pooler 主机、用于日常人工读取的 gateway	运行应用的服务、直接来自 Web 层
`ledgerly_analytics`	加载作业、取其结果的 `reporting-job`、用于分析会话的 gateway	运行应用的服务

迁移 Ledgerly 的读取路径

设想 Alex 要如何把 Ledgerly 从初始状态迁移到这一设计，同时不让夜间作业第一次就出问题。顺序比速度更重要，我会分五步进行。

1. **盘点读取**。针对每个服务，列出它运行的查询，并分为需要主库的读取和不需要主库的读取。`billing-api` 的大多数列表查询属于后一类。它的写入以及写入后的重定向则不是。查询日志是路由规则的原材料。
2. **给每个副本一个用途**。为 `ledgerly_replica_1` 和 `ledgerly_replica_2` 写出连接策略，先把它应用到新的登录上，暂时保留共享登录。
3. **迁移 `reporting-job` 和导出步骤**。让两者都用各自的登录指向 `ledgerly_replica_2`，然后在主库上拒绝这些登录。在关闭旧路径之前，让两条路径并行运行几个周期，并比较输出。
4. **迁移浏览读取**。在开关控制下，把 `billing-api` 和 `admin-console` 的浏览读取路由到 `ledgerly_replica_1`，并逐步提高比例。
5. **关闭副本上的共享登录**。一旦没有任何合法用途还在使用它，就移除它连接任一副本的能力。正是这一步让其余的成果得以固化，也是团队最容易拖延的一步。

数据团队的路径也会改变。Jordan 的月末问题现在有两个答案。如果一天前的数字就够，就通过 `human_analytics_reader` 去 `ledgerly_analytics`。如果需要今天的数据，就通过 access gateway，以 `human_readonly` 去 `ledgerly_replica_2`：这是一个有日

志记录的会话，看不到 Restricted 列，也无法触碰主库。Jordan 失去的，是把桌面客户端指向任意主机并且一连就通的能力。Jordan 得到的，是一条重查询只会成为副本的问题，对其他人则影响轻微。

保护副本免受读取者影响

同时服务人员和夜间作业的副本，会遇到重查询；在某些引擎上，长查询可能延迟复制，或者根据配置被取消，以便复制继续进行。控制措施就是第 5 章的那些，应用到副本的角色上：`human_readonly` 的语句超时短于 `app_reporting_job`，由 gateway 为每个人员会话登录设置的单登录连接数上限，以及在夜间聚合窗口之外运行的导出步骤。再加上延迟告警，并把呼叫发给登记册中该副本的负责人。没有人收到的告警不是控制。

权衡与成本

副本是一个需要运行的系统。它需要监控、打补丁、确定容量和恢复；对小团队而言，一个副本加一份夜间拷贝或许就够了。即便没有它，访问方面的论证依然成立：决定哪些登录能触及主库，哪些不能。

共享 `ledgerly_replica_2` 是有代价的。人员和夜间作业争用同一个副本的容量，换来的是只需保护一个地方。拆出第三个副本，在策略层面成本很低，在运维层面成本很高。

连接策略意味着更多配置。每台服务器的规则必须与角色 schema 一致，并且经得起故障转移、重建和新增副本。服务器之间的漂移就是失败所在，而比较检查是让它保持可见的代价。

有些页面会滞后。把读取移出主库意味着延迟，总会有人注意到。应事先与支持和产品约定哪些页面可以滞后，并且在副本上的排查中遇到对不上的情况时，先检查延迟（第 15 章）。

登记册增加摩擦。要求在拷贝存在之前先有一行记录，会拖慢想要这份拷贝的团队，而这种拖慢正是预期的效果。没有人必须在创建时填写的登记册，只是一份某人还记得的东西的清单。

有些情况下，本章的建议并不适用。一个小系统，如果主库有富余余量，而且从中读取的工程师本来就经由受控路径连接，也许不需要为负载而设副本，拆分读取路径的收益也就较小。一个对每次读取都有极严格新鲜度要求的系统，可能不得不让大部分读取在主库上运行，并把精力放在限制这些读取者能做什么上。副本也不是备份；第 13 章讨论备份。

常见错误

- **把副本当作比主库更不敏感的东西**。它不能被写入，于是积累起宽泛的访问：共享登录、人人都有的常驻读取角色、更宽的防火墙规则。它存放着同样的数据，所以应当得到同样的授予和同样严格的连接策略。
- **让报表读取或人工读取回退到主库**。回退很诱人，因为副本宕机时它能让报表继续运行。应当让读取失败，并在主库上拒绝这些登录，使一个失误无法把它们送到那里。
- **把用来做决定的读取发给副本**。读取很廉价，副本又空闲，看起来是轻松的收获。基于陈旧数据做出决定、随后又跟着一次写入，余额和状态就是这样漂移的。应当按读取之后发生什么来给读取分类，而不是按它有多重。
- **通过让人员指向副本来搭建数据仓库**。这样数据团队就背上了生产的 schema、生产的敏感性和生产的容量限制。应当给分析提供它自己的拷贝和自己的角色（第 12 章和第 13 章）。
- **没有登记册就增加拷贝**。每一份都是在帮一个赶期限的团队的忙。六个月后，没有人说得清究竟存在多少份。应把负责人和结束日期作为创建拷贝的一部分。
- **忘记主机和快照**。角色是对的，而副本的卷快照却能被任何拥有基础设施访问权的人读取。应当把主机、存储和快照纳入每份拷贝的控制范围。
- **假定可以按副本收紧角色**。在采用物理复制的引擎上，角色是集群范围的。应在连接层，或借助独立拷贝，来缩小某个读取者的访问面。

要点

- 写入需要主库，大多数读取则不需要，所以读取路径的第一个问题是，这次读取是在做决定，还是紧随一次写入；报表读取和人工读取永远不回退到主库。
- “只读”是三种机制，只有副本无法被会话撤销，所以只读角色限制的是写入，而不是会话可以读取多少。
- 副本存放着与主库相同的数据，并受同样的集群范围角色约束，所以第 3 章的授予与连接策略共同决定读取者能看到什么。
- 连接策略是以来源和登录为键的每台服务器一份的规则，与角色 schema 保持同步；每个副本只有一个用途，共享管理员登录在任何

一个副本上都不被接受。

- 主库上的 JIT 或 break-glass 会话，需要一个被允许开启它的人、一条记录在案的理由、一个很短的窗口以及在没有申请跟进时的告警。
- 分析属于为其受众构建的独立拷贝，称它为副本会掩盖它带有不同控制这一事实。
- 每份数据拷贝都需要一位负责人、一份读取者名单，并且除副本和分析副本外还需要结束日期，而创建拷贝的那一步在缺少这些时会拒绝运行。

读取路径现在有了自己的角色、自己的目的地和一份简短的拷贝清单，但我们还没有说明人员如何到达其中任何一个。下一章讨论人员的访问路径，并比较 bastion、数据库 proxy 和 access gateway。

第三部分

人员访问

第 7 章 — 访问路径：bastion、proxy 与 access gateway

设想平台团队的 Alex 在一次规划会议上被问到一个直白的问题：“如果明天关掉通往生产数据库的一条路径，关掉哪一条会造成影响？”没有人能很快回答。Alex 去查个究竟，先从网络入手。VPN 给每位工程师一条通往主库的路由。还有一台 bastion 主机，是两年前为一位外包人员搭的，至今仍开着监听端口。十几台笔记本电脑上的桌面客户端都保存着连接，其中一台被配置为经由 bastion 建立隧道，另一台则直接走 VPN。

接着 Alex 问本周值班的 Robin，夜里 Robin 会怎样连到数据库。Robin 毫不犹豫地说出两条路径，想了想又说出第三条。每条路径都能用，没有一条是设计出来的，而且每一条的终点都是同一个共享管理员登录。网络团队能说出谁在 VPN 上。数据库能说出管理员登录连接过。但没有人能说出某个连接背后是哪个人，也没有人能在不关掉所有人访问的前提下，关掉某一个人的访问。

所以本章的问题是：人与数据库之间应当放什么，才能让到达网络与到达数据成为两个各自独立、各自带着名字的决定？本章加入 access gateway，这是 Ledgerly 目标状态中人员一侧的第一步。在本书的五个部分中，它承载路径，以及经由这条路径进入者的身份：网络决定谁可以敲门，gateway 给每个会话标上名字。gateway 会话获得的角色，以及人员如何申请读取之外的权限，属于第 8 章。

问题及其难点

网络路径和数据库路径看起来是同一回事，因为前者是尝试后者之前所必需的。位于 VPN 上的人可以向数据库端口打开连接。此人随后能否做任何事，由用户名和密码决定，而在 Ledgerly 的初始状态下，密码是共享的。网络表示可以敲门，而数据库无法分辨是谁在敲。

这就是问题的核心。网络控制回答的是“哪些机器可以与哪些机器通信”，而且回答得很好。它们回答不了“这个人是谁，此刻允许做什么，做过什么”。当网络规则被拿来替代这些问题的答案时，它会以一种悄无

声息的方式失效：规则本身依旧正确，它所授予的访问却已经过时。换过团队的工程师仍在 VPN 上。合同已经结束的外包人员，仍在一个没人评审的组里。

还有三点让人员路径比服务路径更难。

第一，人是最多样的客户端。服务从已知的地方、用已知的驱动连接。人则可能使用命令行客户端、桌面 GUI、notebook、浏览器工具，或者上周二刚写的脚本。每一种都有自己记住凭据的方式、自己的自动提交默认值以及把结果下载到本地磁盘的习惯。第 1 章列出了会记住密码并自动提交的客户端；这并非不寻常的配置，而是许多为方便而生的工具的默认行为。

第二，人从控制最弱的地方连接。笔记本电脑会被带着走动、会休眠、会被备份，有时还在家里被共用。保存了密码的连接会随笔记本电脑一起移动，导出到文件的结果集也一样。这一切数据库都看不到。

第三，路径是在压力之下使用的。夜里需要数据库的工程师，没有心情去学一个新工具，而缓慢或脆弱的路径，会被有办法的人绕开。一个纸面上正确、却在凌晨 3 点输给旧捷径的设计，什么也没有改变，这就是为什么捷径必须被移除，而不只是被劝阻。

一条写着“不要共享管理员登录”的策略无法检查，因为数据库日志里只显示管理员登录，别无其他。而每个会话都有自己登录的设计可以检查，一条只允许一样东西触达数据库的网络规则也可以检查。本章其余部分比较构建这种设计的各种方式。

四种访问路径的比较

在比较之前，先说明两个术语，因为业界对这些词的用法比较松散，而本书需要将其固定下来。

bastion 是一台经过加固的主机，是通往受保护系统的唯一网络路径。它转发连接。它不理解数据库协议，所以分不清 `SELECT` 与 `DROP`；除非此人登录的是 bastion 本身，否则它也不知道连接背后是谁。

access gateway 是具备身份感知能力的中介，它对人进行认证、签发短期凭据，并记录会话。**数据库 proxy** 则是另一回事：它是理解协议的中介，对数据库连接进行池化、路由或过滤。gateway 可以构建在数据库 proxy 之上，但两者回答的是不同的问题。proxy 懂协议；gateway 懂人。本章只在这个意义上使用这两个术语。

下面是团队通常会考虑的路径，列出每一种对人员了解什么，又留下什么。前四种把人置于会话之中；后两种让人留在会话之外。

路径	对人员了解什么	数据库日志显示什么	主要成本
公共 endpoint	除数据库登录外一无所知	一个登录，来自任意地址	把数据库端口暴露给整个互联网
通往数据库的 VPN	此人在网络上	共享登录或个人登录	把网络成员身份当作身份
Bastion	谁登录了 bastion 主机（若账号按人建立）	此人输入的登录，来自 bastion 的地址	多一台要打补丁的主机，而且它看不到语句
Access gateway	来自公司单点登录的身份	对应唯一一个人和一次会话的会话登录	一个新系统，在有人需要时必须可用
产品路径，不用 SQL	此人在内部工具中的角色	服务自己的登录，此人记录在工具的审计轨迹中	每项任务都要构建一个功能；自由形式的问题没有路径
提交即运行的执行器	提交的人，以及批准的评审人	执行器的登录，运行一组经过评审的语句	每项任务都要等待评审；没有任何交互

最后两行根本不是连接数据库的方式。人从不持有会话，语句由事先检查过它的组件来运行。它们之所以属于这个比较，是因为对许多任务而言，它们是更好的答案：从不开启会话的人，也就不会留下一个未关闭的会话。第 1 章介绍了这类模式的范围，第 9 章讲 gateway 如何检查语句，第 10 章讲数据修正模式；本章讲的是，在适合那些模式的任务迁出之后，仍然剩下的会话。

公共 endpoint

我不会为人员使用而把数据库暴露在公共互联网上，理由很简短。可达的端口会招来与 Ledgerly 员工毫无关系的自动化流量，该数据库的每一个凭据随即都成了值得去猜的东西。剩下的唯一防线是登录的强度，而本书的前提是登录会泄露。

在个别情况下，托管数据库只能通过公网访问，服务商提供来源地址的允许列表。如果处于这种境地，应把允许列表当作最低限度，并在服务商允许时尽快转到私有路径。检验标准很简单：从未在本组织工作过的人，能否向该端口打开连接？如果能，本书中的其他每一项控制，起点都更差。

VPN

VPN 是一个合理的网络层。它把能触及数据库的人缩小到那些被给予了路由的人，而且大多数公司已经有了。它的局限在于它能证明什么。它证明某个人，或某个人的笔记本电脑，通过了 VPN 的检查。它没有说明此人之后使用了哪个数据库登录，也不记录语句。

在 Ledgerly 的初始状态下，VPN 就是全部控制。VPN 上的每个人都能触及主库，而管理员登录对所有人都是同一个。我会保留 VPN，但给它一份更小的工作：它把人送到 gateway 和其他内部工具，不再把人送到数据库端口。

Bastion

bastion 增加了一个有用的特性：有一个单一的位置可以关上网络之门。数据库的防火墙只允许 bastion 的地址，因此关掉一个 bastion 账号，就关掉了一个人的网络路径。如果 bastion 有按人建立的账号，并记录谁登录了它，就能得到一份关于谁在何时连接的粗略记录。

bastion 给不了的，是网络层以下的任何东西。一旦有人通过它打开了隧道，到数据库的连接就是此人的客户端所做出来的样子。登录仍然由此人键入，或保存在其客户端里。语句对 bastion 不可见。所以 bastion 理顺了网络的图景，却没有解决身份的图景，这也是团队明明有了它，仍然回答不了“这条语句是谁运行的？”的原因。

bastion 还有一个悄无声息的维护问题。它是一台带 shell 的主机，因此需要打补丁、管理密钥，还需要一位记得它存在的负责人。开头那台外包人员的主机就是这种失败的常见形态：它在搭建那天是正确的，此后没有人被指派去管它。

Access gateway

access gateway 是我会把人送去的地方，理由如下。

它依据公司的单点登录对人进行认证，所以身份就是公司已经在控制的那一个。离开公司、更换团队、多因素校验失败，都会在 gateway 处生效，无需任何人去编辑数据库。它签发的凭据只属于一个会话，并随会话一起消亡，所以没有密码可保存。它还会记录会话，因此“这条语句是谁运行的？”在两个地方有答案：gateway 的会话记录，以及数据库日志，后者记录的登录恰好对应一个人。

gateway 大致有两种形态。有的位于数据路径中：人的客户端连接到 gateway，由 gateway 代表此人打开到数据库的连接。另一些只是代理凭据然后退到一旁，客户端用短期登录直接连接数据库。第一种形态可以记录语句，并对连接强制执行规则。第二种更简单，路径中的东西更少，但这样一来网络必须允许人员触达数据库端口，这就把本章起点所说的部分问题又带了回来。

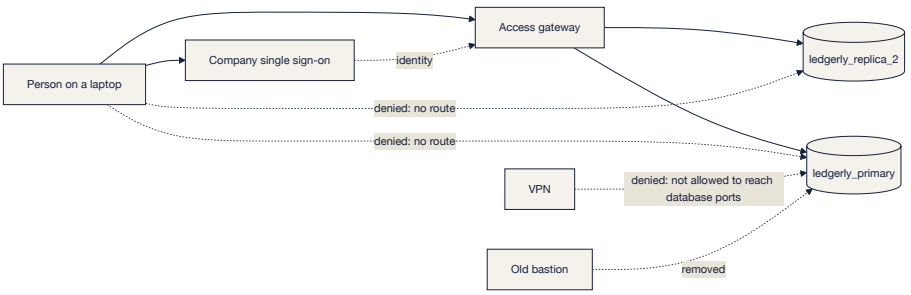
对 Ledgerly，我会选择数据路径形态，因为它能让网络规则做到绝对：只有 gateway 的地址能触及数据库主机。代价是 gateway 要处理每一条人员语句，所以它的容量和可用性很重要。下文会回到这一点。

区分网络触达与数据库访问

本章最有用的改变不是某个产品，而是把初始状态一并回答的两个问题分开。

第一个问题是**网络触达范围**：哪些机器可以向哪个数据库主机和端口打开连接。第二个是**数据库访问**：连接之后，哪个登录可以做什么。在初始状态下，VPN 回答第一个问题，共享密码回答第二个。改变之后，由一条网络规则一次性对所有人回答第一个问题，而 gateway 逐个对每个人回答第二个。

下图展示了人员的目标布局。请看那些被拒绝的箭头：它们是今天存在、此后将不复存在的路径。



这张图背后有两个强制执行点。数据库主机上的网络规则，只允许来自 gateway、服务自己的网络、流水线和凭据 broker 的连接。而 gateway 只会把会话转发给该会话类型所允许的数据库：日常读取会话去 `ledgerly_replica_2`，为提出 JIT 或 break-glass 申请而开启的会话去 `ledgerly_primary`，如第 6 章所述。

我会把网络规则写成任何人都读得懂的简短清单，因为访问评审检查的就是这份清单。第 6 章定义了其中的区别：这份清单是网络策略，决定哪些主机可以连接；连接策略则决定哪个登录可以向哪个数据库认证。如果清单里有一个没人能解释的条目，那个条目就是一条路径。

```
network policy (illustrative):
    ledgerly_primary:
        allows  access gateway
        allows  service networks of billing-api, invoice-worker,
                payments-service, admin-console
        allows  deployment pipeline
        allows  credential broker, for access administration
        denies  everything else, including the VPN

    ledgerly_replica_1:
        allows  service networks of the read-only services
        allows  replication traffic from the primary
        denies  everything else, including the VPN

    ledgerly_replica_2:
        allows  access gateway
        allows  reporting-job network
        allows  replication traffic from the primary
        denies  everything else, including the VPN

    ledgerly_analytics:
        allows  analytics-job network
        allows  access gateway, masked views only
        denies  everything else, including the VPN
```

同一份清单接着列出离开数据库主机的流量，以及什么可以触及 gateway。

```
network policy (illustrative, continued):
    egress from the database hosts:
        allows  replication, the audit log, backups
        denies  everything else

    the gateway's own exposure:
        reachable from  the VPN and the office network, nothing else
        reaches         the database hosts, the audit log, single
sign-on
```

请注意，在这一模型中 VPN 要做的事有多么少。它不再守卫数据库。它把人送到 gateway，其余由 gateway 决定。VPN 的重要性降低了；每个人的访问现在都与一个有名字的会话绑定。

清单中有两个条目容易被忘记。首先是出站流量。可以向任意地方打开出站连接的数据库主机，可能被用来把数据送出去，而一份简短的允许列表（复制、审计日志、备份）把这变成一个可见的事件。其次是 gateway 自身的暴露面：它现在是唯一可触及的门，所以它自己也在一份简短的清单上，并且不得从 VPN 本来就触及不到的任何地方被访问。

这份清单还假定网络能够区分调用方。在共享集群网络上，多个服务来自同一个地址段，网络做不到这一点：允许“服务的网络”的规则，就是允许所有服务。在这种情况下，网络规则只把既非服务也非 gateway 的一切挡在外面，区分一个服务与另一个服务的，是 pooler 和登录（第 5 章和第 6 章）。我会把这一点作为该集群的规则写下来，以免有人把网络清单读得比它实际更精确。哪个登录可以向哪个数据库认证，是第 6 章的连接策略，本章只是指向它。

Ledgerly 中的 access gateway

下面按先后顺序说明 gateway 做什么，以及某一步失败时会发生什么。这是我会仔细测试的部分，因为失败即开放的 gateway，只会变成换了个更好听名字的新共享密码。

什么是会话

会话在人完成认证时开始。gateway 检查身份，决定此人有资格申请哪种会话类型（大多数人是日常读取会话；JIT 和 break-glass 类型只能通过第 8 章的申请获得），然后创建一个**会话登录**，即此人的会话所使用的账号。它创建在 `ledgerly_primary` 上并被复制（第 6 章）。会话随后连接到其类型所属的数据库。这个登录的名字里带有此人的身份，所以它会出现在数据库日志中，并且默认是 `human_readonly` 的成员。它的凭据为本次会话生成，交给 gateway，从不向此人展示，因此没有什么可保存。

登录名有固定的形状，使得日志行不必查表就能读懂：一个表明它是会话登录的前缀，此人的简短标识符，以及会话自己的编号，如 `sess-robin-0412`。名字起到标识作用；把它映射到人和申请的，是 gateway 的记录，任何东西都不应通过解析名字来决定会话可以做什么。

为了创建和删除登录，以及授予和撤销角色成员关系，gateway 需要数据库中的权限。它通过自己的登录获得，该登录属于角色 `ledgerly_access_admin`。角色定义的流水线步骤和凭据 broker 也使用这个角色，各自通过自己的登录（第 3 章和第 4 章）。这使得 gateway

的登录成为一项敏感凭据：持有它的人，可以在其所管理的角色之内授予任何东西。我会把它存放在与所有者凭据相同的地方、遵守相同的规则，不赋予它任何数据权限，并审计对它的每一次使用（第 14 章）。为图方便把它复制进脚本或聊天，会在更高一层重新造出共享密码。

谁可以开启哪种会话

`mayRequestKind` 是策略，我会把它保存为评审人可以阅读的数据，而不是埋在 gateway 代码中的一个条件。一张表回答它：单点登录组，该组可以开启的类型，每种类型最长持续多久，落在哪个数据库，它的默认值（只读、语句超时、行数上限），以及是否允许导出。

```
session policy (illustrative):
    group backend-engineers:
        routine       max 8 hours, ledgerly_replica_2, read-only, row
cap, no export
        primary-bound max 2 hours, ledgerly_primary, only to file a
request
    group analysts:
        routine       max 8 hours, ledgerly_analytics, masked views,
export logged
    group contractors:
        routine       max 4 hours, ledgerly_replica_2, masked views,
no export
```

主库绑定的类型是敏感的，因为主库上的会话离写访问只差一个申请。四条规则让它们保持稀少：只有指定的组可以开启；开启时必须记录理由；gateway 只等待很短的时间，比如十五分钟，看是否有申请从中提出；以及没有申请就结束的会话会触发告警，因为“打开了主库却什么也没做”是值得注意的模式。两小时的上限只持续到授予发生为止：当第 8 章的 `grantJitAccess` 或 `declareBreakGlass` 批准一份申请时，gateway 会把该会话的 `ends_at` 重置为申请的 `expires_at`，下面的 `endSession` 随即在那个时间关闭会话。第 8 章从另一头开始，即从申请开始。

操作顺序

gateway、审计日志、数据库和单点登录是各自独立的系统。没有哪个事务能横跨它们，所以步骤的顺序和对半成品会话的处理必须经过设计。

```
function openSession(person, wantedKind):
    identity = verifySingleSignOn(person)
    if not mayRequestKind(identity, wantedKind):
        raise SessionRefused
    login = newLoginName(identity)
    endsAt = now() + maxSessionLength(wantedKind)

    // the record comes first: a login with no record is what the
sweep hunts
    recordSession({ login: login, identity: identity, kind:
wantedKind,
                    ends_at: endsAt, status: "opening" })
    try:
        auditLogAppend({ actor: identity, action: "session-open",
                         subject: login, ends_at: endsAt })
    on AuditLogUnavailable:
        markSessionRefused(login)
        raise SessionRefused

    try:
        // always on the primary: logins are cluster-wide and
replicate
        createLogin(login, on: "ledgerly_primary", password:
freshSecret())
        grant("human_readonly", to: login)
        // replication lag: wait until the target database knows the
login
        waitUntilLoginVisible(login, on: databaseFor(wantedKind))
    on DatabaseError:
        // a login with no role is harmless, but it must not linger
        dropLoginIfExists(login)
        markSessionRefused(login)
        raise SessionRefused
    markSessionOpen(login)
    return connectionFor(login, databaseFor(wantedKind))
```

请从失败处理的角度来读这个代码块。记录在登录存在之前写入，所以中途崩溃留下的，要么是一条写着“opening”却没有登录的记录，要么是一个有记录可以解释的登录。审计条目先于登录：如果 log store 不可达，就不会开启任何会话。这个选择是有成本的，权衡一节会点明。对于日常读取会话，gateway 会等到 `ledgerly_replica_2` 收到该登录，否则复制延迟会让第一次连接时不时失败。如果创建登录或授予角色失败，gateway 会尝试清理并拒绝该会话。如果连清理也失败，就会有一个没有任何权限的登录存在，下面的清扫会把它移除。

结束会话

会话以三种方式之一结束：人断开连接、超过空闲上限，或者 `ends_at` 到了。后两种之所以重要，是因为人们会让标签页一直开着。已打开的连接会比已过期的凭据活得更久（第 4 章），所以 gateway 自己关闭连接，然后移除登录。同一个函数也服务于离开公司的人：来自单点登录的取消预配事件会调用 `endSession`，而最长会话时长则是该事件始终没有到达时的边界。

```
function endSession(login, reason):
    // the single owner of a session's end; Chapter 8's expiry calls
it too
    try:
        closeConnections(login)
    on CloseFailed:
        // a drop that fails on a live connection is retried by the
sweep
        notify("platform", "connections not closed: " + login)
    // requestsOf returns only requests not yet marked revoked
    for each id in requestsOf(login):
        try:
            revokeRequestAccess(id, from: login)
        on LoginNotFound:
            // nothing is left to revoke
        on RevokeFailed:
            notify("platform", "session grants not revoked: " + login)
            return
        markRevoked(id)
        auditOrQueue({ action: "revoke", actor: "gateway",
                       subject: login, request: id })
    try:
        revokeIfExists("human_readonly", from: login)
        dropLoginIfExists(login)
    on DatabaseError:
        notify("platform", "session login not removed: " + login)
        return
    markSessionEnded(login, reason)
    auditOrQueue({ action: "session-end", subject: login, reason:
reason })
```

这里的 `auditOrQueue` 把记录追加到审计日志，如果日志不可达，则把它排队等待重试，并通知平台团队。到那时撤销和删除已经完成，所以日志故障只会延迟条目，绝不会延迟访问的移除。已经不存在的登录被视为成功：由第 8 章的到期路径较早结束的会话，在这里会被干净地关闭，其记录被标记为已结束，也不会触发告警。

清扫会对每一个本应已经结束的会话调用 `endSession`。

```
function sweepSessions():
    // endSession reports a stuck login and returns, so one failure
does not stop the loop
    for each s in sessionsPastEnd(before: now()):
        endSession(s.login, "expired")
    for each s in sessionsStuckOpening(before: now() - openingGrace):
        endSession(s.login, "abandoned")
    for each login in loginsWithoutRecord():
        endSession(login, "orphaned")
```

关键在于顺序。一份申请的授予，依据其保存的记录来撤销，只有在撤销成功之后才标记为已撤销，然后才移除登录，因为有些引擎会拒绝删除仍持有权限的登录。撤销或删除失败时，会通知平台团队，并把记录留给下一次清扫；审计写入失败则会被排队，不会中断循环。清扫给仍处于“opening”的会话一个宽限期，以免移除一个才创建几秒钟的登录。第 3 章的漂移检查，会把现存的登录与 gateway 的记录对比，并把授予与保存的三元组对比，它按自己的计划运行，因此一个作业停了，不会掩盖另一个。

哪个数据库，哪个角色

每个会话都落在其类型所属的地方。日常会话是连接到 `ledgerly_replica_2` 的 `human_readonly` 登录。为提出 JIT 或 break-glass 申请而开启的会话，开在 `ledgerly_primary` 上，第 8 章会展示如何从该会话提出申请，以及授予如何随会话一同结束。gateway 强制执行第 6 章以连接策略写下的目的地规则：知道会话类型的是 gateway，拒绝为另一种类型准备的登录的是数据库主机。

一个人可以同时开启两个会话，一个日常会话，一个带申请的会话。我会保留这种可能，并让两个会话保持明显的分隔，以免有人用特权会话去做随手的查询。人的成本是客户端里多一个连接。好处是特权会话容易找到、时间短、也容易结束。

Gateway 记录什么

由于 Ledgerly 的 gateway 位于数据路径中，它可以记录会话发送的每条语句，连同开始时间、结束时间和受影响的行数。我会把这份记录存放在它所描述的那些人无法编辑的地方；第 14 章讲在哪里。一条记录如下所示：

```
statement record (illustrative):
    login:         sess-robin-0412
    person:        robin
    kind:          jit
    request:       req-0412
    fingerprint:   UPDATE invoices SET status = ? WHERE id IN (?)   //
literals redacted
    param_hash:    9c1e...
    rows:          12
    started_at:    03:21:07
    ended_at:      03:21:07
```

支付数据和财务报告方面的监管制度往往会问：谁接触过生产数据，经由什么路径；会话记录就是这个问题的答案，至于如何解读，由读者所在组织的法务与合规人员负责。有两点局限值得了解。语句日志说明问了什么，而不是返回了什么，我不会存储完整的结果集，因为那会把日志变成数据的拷贝。出于同样的原因，指纹携带的是语句的形状和参数的哈希，而不是参数的值。此外，没有人读的日志只是记录，并不保护任何东西，所以要决定谁来查看它，以及何时查看：对持有 JIT 申请的会话做每周抽样是一个开端，访问评审则是另一个场合。

容量与连接预算

每个人员会话同时也是一个数据库连接，而第 5 章的预算给了人员一个固定的份额，比如二十个连接，仅作示意。工程师的日常会话全都落在 `ledgerly_replica_2` 上，所以该副本就是人员连接槽位的所在主机，它的连接数上限是这个数字最终被强制执行的地方。gateway 应当自己强制执行这份份额：每人的上限，每种会话类型的上限，以及预算用完时明确的拒绝。gateway 处的拒绝，是给人的一条消息。预算耗尽时，数据库处的拒绝，则可能挤掉一个服务。把上限放在 gateway，可以避免人员的预算变成服务的问题。

笔记本电脑与本地拷贝

gateway 控制通往数据库的路径。数据一旦到达笔记本电脑，之后会发生什么，它就控制不了了，风险中悄无声息的一部分就在这里。

笔记本电脑上留下两样东西。一样是连接设置：保存的主机、保存的登录以及在初始状态下保存的密码。另一样是人们对结果所做的一切：导出的文件、粘贴的表格、为了“玩一玩”而留在本地的几千行拷贝。每一样都是第 6 章意义上的数据拷贝，是生产数据所在的、没有负责人也没有到期日的地方。

我会分三层来处理，从成本最低到最高。

移除客户端能记住的东西。由于会话凭据为一个会话生成且从不展示，桌面客户端没有可保存的密码。保存的连接现在存放的是 gateway 地址和此人的单点登录，而指向主库的旧保存条目，可以在迁移期间删除，并附一份简短的清单，说明删什么、在哪里删。这一层几乎不花成本，却能消除最常见的泄露。

选择适合任务的默认值。gateway 可以用只读默认值和第 5 章的语句超时来开启会话。它还可以限制一条语句返回的行数：在流式传输时对行数计数，达到上限时关闭游标，告知此人，并把该事件写入语句记录；反复触及上限的会话，是每周抽样首先关注的对象。

人能够解除的默认值不会一直是默认值。客户端可以发送 `SET` 来调高超时或关闭只读，所以 gateway 维护一份受保护设置的清单，并拒绝任何修改它们的语句。代价是，有些客户端在连接时会发送这类语句，因此会失效；这是需要在迁移中点明的成本，而第 9 章讲 gateway 如何决定拒绝哪些语句。当会话没有可供提交的写权限时，自动提交的客户端就不那么危险。由 gateway 自己运行的基于浏览器的控制台则更进一步，因为除非存在并记录了导出按钮，否则什么也不会被下载。这种控制台比原生客户端更慢、体验更差，我会把它作为一个选项，提供给只看、从不需要自己工具的人，而不是作为一项要求。

明确决定什么可以带走。结果是否可以从会话中导出，是单独的一条规则，它与第 12 章的脱敏和最小化工作，以及第 13 章的导出路径放在一起。gateway 可以应用这条规则，但无法让它变得不必要。人可以读屏幕、给屏幕拍照，或者重新键入一个值，而我不知道对此有什么性价比高的控制。gateway 减少意外带走的数据量，让有意的导出变得可见；它不会让带走变得不可能。

> **注意**：如果工程师为了分析需要大量提取数据，答案通常不是调高 gateway 的导出上限，而是为此专门设计的脱敏分析拷贝，第 6 章和第 12 章对此有描述。gateway 不是做批量数据搬运的合适工具。

一扇门，一个单点故障

单一入口容易解释，容易记录，也容易关闭。它同时也是单点故障，在推荐它之前，值得把这一点直说。如果 gateway 宕机，没有人能通过前门触及数据库，包括那位在别的东西已经出问题时最需要数据库的值班工程师。

没有哪个答案能消除这一成本。有的答案能把它转移到能够应对的地方。

保持 gateway 小。它负责认证、创建登录、转发和记录。每多一个功能，就多一个让它宕机的理由。我会克制自己，不把结果转换和审批界面加进同一个服务，而把它们作为独立组件来运行，gateway 会调用它们，但启动日常读取会话不依赖它们。在语句运行之前对其进行检查的地方，这项检查也是一个独立的组件（第 9 章）。

运行多个实例，并把状态放在持久的地方。两个实例位于同一地址之后、共享会话记录，意味着一次重启或一次糟糕的部署不会结束所有会话。会话存储和审计日志同样是依赖项，它们的可用性就是 gateway 可用性的一部分。这就是上面的开启函数在日志不可达时拒绝会话的原因：这是一个有意的选择，把人员访问与日志绑在一起，而且应当是团队有意做出并写下来的选择。

决定降级模式是什么。有两种故障需要规划。一种是 gateway 宕机而数据库正常。另一种是审计日志宕机而 gateway 正常。对每一种，必须有负责的人事先决定：人员访问是停止，还是经由更窄的路径继续，以及这期间记录什么。本书对 gateway 故障的决定是，没有第二扇门：人员访问停止，包括 break-glass，因为 break-glass 会话是角色更重的 gateway 会话。让这种情况保持罕见的办法，是上面的冗余，而不是 gateway 之外的紧急登录。第 8 章给出了这个理由，以及团队若另作决定该怎么做，第 15 章描述了对这类故障的演练。对于审计日志故障，副本上的读取可以缓冲条目，写入则不可以（见下面的权衡）。

把 gateway 当作生产系统。它有负责人、值班路径、人能读懂的健康检查以及带回滚的部署流程。平台团队负责它。如果 gateway 是某一位工程师的业余项目，单点故障就是这位工程师。

gateway 还集中了另一种风险：gateway 被攻破，就意味着它能触及的一切访问都被攻破。这就是为什么它的 `ledgerly_access_admin` 登录要按前文所述严加保护，而这并不是回避 gateway 的理由：初始状态有一个共享密码，被复制给团队里每一个人，处在同样的位置上。

Ledgerly 从多条路径迁移到一条

处于 Ledgerly 这种状况的团队不会在某一天把所有东西都切换。我会按一个顺序执行这些步骤，使每个时间点上都有一条可用的路径。

1. **盘点路径**。列出今天人员触及任何 Ledgerly 数据库的所有方式：VPN 路由、bastion、客户端中保存的条目以及任何使用管理员登录连接的脚本。开头 Alex 的那番查访就是这一步，它应当以平台团队保管的一份书面清单结束。
2. **在旧路径旁边搭起 gateway**。起初它在定义登录的主库上创建会话登录，授予它们 `human_readonly`，并把它们连接到 `ledgerly_replica_2`，仅此而已。旧路径仍然可用。少数人用 gateway 做日常读取，并反馈哪里别扭。
3. **先迁移读取**。读取占了人们所做事情的大部分，且尝试的风险最小。待别扭的地方修复后，对 VPN 关闭副本，并删除指向它的保存条目。
4. **最后迁移主库**。直接访问主库是人们在事故期间所依赖的，所以我会等第 8 章的 JIT 和 break-glass 路线已经可用并经过演练，才关闭它。更早关闭，会在替代方案就绪之前拿走应急路径。
5. **移除旧 bastion，以及管理员登录的触达范围**。外包人员的 bastion 被关掉。共享管理员登录不再被人员网络接受。服务保留自己的登录，直到第 4 章的工作替换它们。
6. **测试拒绝**。对每一条旧路径，有人去试一下，确认它失败。没有人测试过的被拒绝路径，只是一个假设。

迁移的成本主要在人。人们必须学习新的连接流程，他们的一些脚本会失效，因为这些脚本假定存在常驻登录。应当规划一个两者并行可用的窗口，并写下旧路径关闭的日期。没有日期，旧路径就会一直开着，因为留着它总是显得更省事。

权衡与成本

也许够用的更小设计。gateway 是关键路径上的一个新系统，如果无法为其值班路径和部署配备人手，更小的设计是更好的起点。如果四位工程师通过私有网络访问同一个数据库，使用的是由团队已有的东西签发的个人短期登录，那么可能已经获得了大部分价值。我会写下什么会促使改变

这一判断：一位外包人员加入、出现第二个数据库、有人要求会话录制，或者第一次因日志说不清是谁做的而出现的事故。gateway 是对这些条件的回应，不是一个里程碑。

失败即关闭是有代价的。在审计日志不可达时拒绝会话，保证了记录的完整，也让 log store 成为每个人员会话的依赖项。有些团队更愿意用本地缓冲来开启读取会话，稍后再发送条目。这对副本上的只读会话是站得住脚的选择，对会写入的会话则较弱。如果选择这样做，要说明它适用于哪些会话，并监控缓冲区。

语句记录是一份数据拷贝。指纹记录对字面量做了遮蔽，但它仍然会指明表和列，任何未被遮蔽的字段，例如自由文本备注或客户端提供的注释，都可能携带值。应当按一条记录可能包含的最敏感内容的分类来对待这份日志，限制谁可以读取它，并设定一个有人负责的保留期限。

延迟与保真度。位于数据路径中的 gateway 会增加一跳，并且可能不支持每一种客户端特性或每一个协议扩展。有些工具通过中介时表现怪异：超大传输、长期存活的游标以及不常见的认证流程，是常见的出问题之处。在承诺旧路径将要关闭之前，要测试团队成员实际使用的工具。

何时不使用 gateway。没有客户数据的单服务原型不需要它。如果系统唯一的人员访问是通过一个托管控制台，而该控制台已经提供了身份、短期凭据和日志，那么在它前面可能不需要再加一层。当真正的需求是有一个可以关闭的网络之门，而且受保护的系统自身具备强大的、按人的认证时，bastion 是合理的答案。在这两种情况下，适用的问题是相同的：此人是谁，访问持续多久，记录在哪里。

常见错误

- **把 VPN 当作身份层**。这很诱人，因为 VPN 已经知道谁在上面，而且人人都有。它证明的是网络成员身份，而不是用了哪个数据库登录或做了什么。应保留 VPN 作为到达 gateway 的方式，再增加一层，把人附着到每个会话上。
- **在数据库前面放一个 bastion 就当作大功告成**。它理顺了网络，却让登录依旧共享、语句依旧不可见。选择 bastion 应当看它能做什么，即一个可以关闭的网络之门，而身份和记录要到别处去找。
- **迁移期间让旧路径一直开着**。暂时保留两条路径看似审慎，事实上也确实审慎，但没有关闭日期，旧路径就会比新路径活得更久。应写下日期，公布它，并在之后测试旧路径确实失败。

- **开启一个没人强制其结束的会话**。只存在于文档里的会话上限，就是名字更长的常驻授予。应当在 `ends_at` 结束会话，既关闭连接也删除登录，并运行清扫，找出不该存在的会话和登录。
- **跳过降级模式的决定**。构建了 gateway 却没有决定它宕机时怎么办的团队，会在压力下做决定，通常是“就一会儿”地重新打开旧的共享路径。应事先决定应急路线，写下来，并加以演练。
- **什么都记录却什么都不读**。记录每条语句感觉像是尽职。如果没有负责人去抽样、没有访问评审去查看，这份日志就是一份没有用途的数据拷贝。应决定谁来读它，以及要找什么。
- **把网络清单读得比实际更精确**。在共享集群网络上，规则分不清一个服务和另一个服务，以为它能分清的评审人，会批准一条比看上去更宽的路径。应写下是哪一项控制在区分调用方。

要点

- 网络触达范围与数据库访问是两个独立的问题，用同一项控制回答两者的设计，说不清谁做了什么。
- bastion 让网络路径变得唯一，但不理解数据库协议，所以它本身会让登录依旧共享、语句依旧不可见。
- access gateway 通过单点登录认证一个人，创建携带该身份的会话登录，并在会话结束时移除它，这正是让数据库日志指名到一个人的原因。
- 只允许 gateway 触及数据库主机的网络规则，是使 gateway 成为唯一入口的控制，它应当是访问评审能够读懂的简短清单。
- 单一入口就是单点故障，所以 gateway 应当小，运行多个实例，由平台团队负责，并配有在事故之前就做出的降级模式决定。
- gateway 无法控制人对已到达笔记本电脑的数据做了什么；它可以移除保存的凭据、设置只读默认值并让导出可见，其余属于脱敏和最小化。
- 先迁移读取，最后关闭主库，为每条旧路径设定关闭日期，并测试被拒绝的路径确实被拒绝。

gateway 给了每个人一个带名字的会话，并默认只读，但读取很少是全部工作。下一章讨论有人在有限时间内需要超出 `human_readonly` 的权限时会发生什么：即时访问、break-glass 访问以及审批给等待它的人带来的成本。

第 8 章 — 即时访问与 break-glass 访问

设想 Ledgerly 的一个周二夜里，凌晨三点刚过。一条告警显示，自 `invoice-worker` 重启以来，一批发票一直停在 `sending` 状态，这批发票不清理，早间批次的提醒邮件就发不出去。没有任何数据暴露，也没有任何数据丢失，问题的代价只是延迟。值班的后端工程师 Robin 通过 access gateway 打开了一个常规会话。该会话只读，运行在副本上，这正合 Robin 前十分钟的需要：看、数、弄清情况。

查看确认了问题，而修复需要对 `invoices` 的一列执行一条 `UPDATE`。Robin 把这条单语句修复写下来，并为它提交了申请。审批人是第二值班工程师，此人正在睡觉，呼叫也没有被确认。Robin 手边有一台笔记本电脑、一份 runbook，还有一个要做的决定：是等一个答复，还是绕开这个问题。在旧的 Ledgerly 里，这个决定根本不会出现，因为共享的管理员密码一粘贴就能用。在新的 Ledgerly 里，绕行的路要经过设计，回头的路也一样。

本章加入运行示例中目标状态的下一批步骤：人没有常驻写权限，写操作以 JIT 授予的形式出现，且位于第 3 章的上限之内；另有一个应急角色，很少使用，每次使用都要评审。在本书的五个部分中，这里主要是授予：谁审批什么、针对哪条语句、持续多久。本章回答 Robin 的问题：访问如何在需要时出现，如何随任务一同结束，以及正常路径太慢时会发生什么。

问题及其难点

常驻访问是一个承诺：没有人需要再次提出申请。这在一个月内很方便，之后就没有人记得这个承诺涵盖了什么。人换了团队，授予还留着；为某次事故放宽的角色一直宽着。访问评审到来时（第 14 章），评审人面对的是没有任务可对应的授予，而“这还需要吗？”的答案是“大概”。常驻授予的代价是它没有被使用的每一天，因为它仍然带着所涵盖内容的影响范围。

显而易见的替代方案是 **即时（JIT）访问**：为特定任务、在有限时间内授予的访问。它容易表述，却很容易在五个地方出现细微的错误。

范围。最容易的实现是把一个角色授予某个登录一小时。如果这个角色是 `human_jit_write`，该登录得到的就是整个上限：对六张表的更新权限，而任务只需要其中一张表的一列，这样的授予只是带了计时器的常驻授予。范围必须体现在授予本身上，一次申请一份。

到期。在常见引擎上，授予不会自行到期。必须有东西把它收回，通常是一个定时作业，而作业可能停止、丢失凭据、在某个登录上失败后继续运行，或者针对错误的清单运行。

审批。由第二个人阅读每份申请，有人力成本：有人被叫醒，有人在等待。如果审批太慢，人们会找更快的路径，也就是本书想要淘汰的共享密码。如果审批人总是在几秒内回复，审批就不再减少暴露。

应急。没有获准的快速通道的访问方案，会长出一条未获准的通道。问题在于，这条快速路径是经过设计、记录和评审的，还是临时拼凑的。

操作。授予指明一张表和一列，却没有说明哪些行或哪条语句。审批人读到“`invoices` 上的 `UPDATE`，`status`”时，批准的是一种能力，而在这种能力之内运行的任何东西都在审批之外。要让审批意味着“第二个人看到了将要发生什么”，申请就必须指明操作，授予也必须由此而来。

一条写着“访问必须是临时的”的策略，并没有告诉系统在撤销作业宕机时该怎么办。本章其余部分把授予、记录和计时器连接起来，使系统知道该怎么办。

即时访问：指明操作的申请

一份 JIT 申请由五个要素构成，每个要素的存在都对应上述某个失败。每个要素都标注了它服务的是本书模型的哪一部分。

- 上限（授予）`human_jit_write` 定义了一份申请最多可以要求什么。它绝不授予任何登录（第 3 章）。
- 写申请（授予）在 `fix_id` 中指明它所针对的已评审修复。它的（权限、表、列）三元组由该修复推导而来，位于上限之内，并存储在 `access_request_privilege` 中。
- 审批（授予）针对的是语句，并由授予来强制其边界：审批人阅读语句，gateway 把推导出的三元组恰好授予申请人的会话登录，已批准

的语句集记录为 `approved_hash`。到期时撤销的恰好是这些三元组，读取自同一份记录。

- JIT 会话中的写操作（执行）只从 runner 转发，runner 拒绝任何与 `approved_hash` 不匹配的内容。
- 独立的对账（记录）把数据库持有的授予与为其提供依据的申请进行比较，这样作业停止时，发现问题的就是作业之外的东西。

我将按申请遇到它们的顺序逐一说明。

申请包含什么

申请由将要使用它的会话发起，而该会话运行在主库上。一个会话最多携带一份有效申请；超出时间窗口的任务在新会话中以新申请继续。常规排查留在副本上，使用 `human_readonly`（第 6 章），而谁可以打开绑定主库的会话，由第 7 章说明。gateway 知道哪个登录属于哪个人，所以没有人需要键入 principal 或登录名。

下面是 Robin 的申请在任何人答复之前的形态：

```
access_request:
    id:               "req-0412"
    kind:             "jit"
    requester:        "robin"
    requester_login: "sess-robin-0412"
    approver:         null
    fix_id:           "fix-0093"
    approved_hash:    null            // set at approval, never by the
runner
    reason:           "alert-7731: invoices stuck in sending"
    expires_at:       now() + 1 hour        // illustrative window
    status:           "pending"

access_request_privilege:             // derived from fix-0093, not
typed
    ("UPDATE", "invoices", "status")
```

有三点值得注意。理由指向可检查的东西，即告警或工单。授予很窄，Robin 不需要 `SELECT` 三元组，因为会话已经持有 `human_readonly`。而且三元组不是 Robin 键入的：它由 `fix-0093` 推导而来，所以授予与审批人阅读的语句不可能不一致。只持有 `SELECT` 三元组的申请，例如为了读取未脱敏的 Confidential 列，没有修复，也没有哈希。

授予限制的是权限和列，而不是行。上面的 `UPDATE` 可以修改会话能看到的每张发票的 `status`。JIT 授予按列和按时间限定错误能触及的范围；按行限定的，是审批人阅读、runner 执行的那条语句以及第 10 章中先计数的纪律。

把申请绑定到语句

修复是一组经过评审的语句，带有过滤条件和预期行数，形式如第 10 章所述；申请通过 `fix_id` 指明它，并携带它的 `approved_hash`。哈希来自修复的合并记录或已批准的申请记录，绝不来自 runner 的输入，而且在 JIT 写会话中，gateway 只从 runner 转发写操作。只持有 `SELECT` 三元组的申请没有修复、哈希或 runner，审批人阅读的是列清单。第 9 章完整陈述这条规则，并说明 gateway 如何检查每条语句。

检查申请

gateway 在请求任何人审批之前先检查申请的形态，并在授予时与审批人一起再检查一次。

```
function checkRequestShape(request):
    // runs before anyone is asked to approve
    // live = approved and not yet revoked; this request is not yet
counted
    if request.status != "pending" or
hasLiveRequest(request.requester_login):
        raise GrantRefused
    if request.expires_at is null or request.expires_at <= now():
        raise GrantRefused
    // a window longer than the gateway allows is refused, not clamped
    if request.expires_at > now() + maxJitWindow:
        raise GrantRefused
    for each p in request.privileges:
        // an empty column list is never read as "all columns"; table-
level
        // triples exist only on engines using the secrets-tables
fallback
        if p.privilege != "DELETE" and isEmpty(p.columns):
            if not tableLevelAllowed(p.table):
                raise GrantRefused
    // refuses a table outside the ceiling and any Restricted column
    if not withinCeiling("human_jit_write", request.privileges):
        raise GrantRefused
    checkOperationBinding(request)
```

```
function checkOperationBinding(request):
    if hasWriteTriple(request.privileges):
        fix = fixRecord(request.fix_id)    // null when fix_id is null
        if fix is null:
            raise GrantRefused
        // the triples come from the fix; a typed list that differs is
refused
        if request.privileges != triplesFromFix(fix):
            raise GrantRefused
    else if request.fix_id is not null or request.approved_hash is not
null:
        // a SELECT-only request has no fix and no hash
        raise GrantRefused

function checkJitRequest(request):
    // runs again at grant time, once an approver is named
    checkRequestShape(request)
    if request.approver is null or request.approver ==
request.requester:
        raise SelfApprovalNotAllowed
    if request.fix_id is not null:
        if request.approver == fixRecord(request.fix_id).author:
            raise SelfApprovalNotAllowed
```

自我审批的几行，正是四眼原则在代码中强制执行的地方：提出申请的人，或编写修复的人，不能是答复的人。待处理的申请没有审批人，所以形态检查不包含这几行；审批人在审批的那一刻才填入，完整检查在授予时运行。

我会拒绝过长的时间窗口，而不是悄悄缩短它：缩短之后，申请人以为自己有一小时，实际只有二十分钟，意外会在修复进行到一半时到来。

`withinCeiling` 是第 3 章的上限变得可以强制执行的地方。它拒绝这样的三元组：表在上限之外，权限在该表上没有提供，或者列清单包含 `iban`、`account_holder_name` 或 `token_value`。它拒绝对 `bank_accounts` 和 `payment_tokens` 的 `INSERT` 与 `DELETE`，因为新增或删除一行会设置或销毁 Restricted 值。它拒绝除 `DELETE` 之外任何没有列清单的三元组，唯一例外是使用 secrets-tables 回退的引擎上的表级三元组。它还拒绝任何触及 `ledger_entries`、`audit_events` 或访问申请记录的操作，因为没有任何人类角色写入这些表。

有一样东西被有意留在上限之内。对 `bank_accounts` 或 `payment_tokens` 的 `UPDATE (customer_id)` 能通过 `withinCeiling`，因为 `customer_id` 只是把一行链接到其所有者，并不是 Restricted。但把

行移到另一位客户名下，仍然改变了这个账户或 token 属于谁，所以执行策略把它路由到产品路径（第 9 章）。审批人的检查清单则靠人工把它拦下来。

谁审批、多长时间、什么范围

一份 JIT 申请有三个变量：审批人、时间窗口和范围。

审批人 是理解申请中各表的人，且不是申请人，也不是修复的作者。对 Ledgerly 而言，`invoices` 的自然审批人是另一位后端工程师，通常是第二值班工程师。`payments` 或 `payment_tokens` 上任何操作的自然审批人，是拥有 `payments-service` 的团队的人，因为该团队知道那里被修改的 `status` 会在下游触发什么。platform 团队运营访问工具，我不会让它成为自己并不拥有的数据的默认审批人：它会成为瓶颈，成为无法判断该变更的审批人。

```
function chooseApproverPool(request):
    owners = emptySet()
    for each p in request.privileges:
        owners.add(owningTeam(p.table))
    pool = eligibleApprovers(owners)
    // neither the requester nor the author of the fix may answer
    pool.remove(request.requester)
    if request.fix_id is not null:
        pool.remove(fixRecord(request.fix_id).author)
    if pool is empty:
        raise NoApproverAvailable
    return pool
```

空的审批人池会明显地失败，绝不回退到更宽的池。除此之外，池的顺序就是升级顺序：呼叫先发给第一个人，再发给下一个，全部发生在同一份待处理申请之内，只有当它到达 `expires_at` 仍无人答复时才被拒绝。

时间窗口 应当与典型任务一样长，而不是与最坏的任务一样长。假设一次数据修正通常需要二十分钟，表单提供一小时，gateway 的最大值是四小时。延长授予是一份新申请，带有新的审批，而不是对旧申请的编辑，因为编辑会让一次审批被无限拉长。修复超出窗口时，会话结束，任何打开的事务随之回滚，所以写成单个事务的修复（第 10 章）会干净地丢掉已做的工作，而不是留下一半已经应用的结果。

范围是三元组，以及它们所来自的语句。审批人两者都读；审批界面把三元组显示为它们将成为的授予：“对 `invoices`（`status`）的 UPDATE，至 04:12”。在 `SELECT` 中指明 Confidential 列的申请，会显示该列未脱

敏，界面用文字说明这一点，而不只是列在列清单里。

审批在 gateway 自己的界面上进行，通过单点登录，并在点击时进行一次新的多因素检查，而不是在聊天消息下作出反应：审批本身就是控制，所以记录它的东西必须知道是谁给出的。

审批人检查什么

审批人的决定比“这是个好主意吗？”要窄：批准语句，让授予来强制其边界。这份检查清单从访问一侧，对应第 10 章中评审人对修复的检查清单。

1. 理由是可检查的告警或工单吗，它还处于打开状态吗？
2. 界面上的语句做的事和理由所述一致吗？阅读语句，而不是申请人对它的概述。
3. 过滤条件限定了行吗，预期行数与申请人看到的一致吗？
4. 推导出的三元组涵盖了语句，且没有更多吗？在 secrets-tables 回退下，“没有更多”指的是“该表所有非 Restricted 列”，界面会这样说明。
5. 时间窗口与这项任务大致一样长，并且低于最大值吗？
6. 语句触及 `bank_accounts` 或 `payment_tokens` 的 `customer_id`，或任何 Restricted 列吗？那就不是 JIT 数据修正。把它送到产品路径，并拒绝该申请。
7. 我既不是申请人也不是作者吗，我能自在地向下周的评审人解释这次审批吗？

无法回答其中某一项的审批人，应在申请保持待处理的同时询问申请人。沉默不是批准，列清单也不是。

授予

审批是四份记录必须一致的时刻：gateway 的申请记录、审计日志条目、数据库中的授予和会话。没有事务横跨它们，所以顺序很重要，每种失败留下什么也很重要。

```
function grantJitAccess(request):
    checkJitRequest(request)

    // the hash comes from the records, as "Binding the request"
explains
    hash = approvedHashFor(request)
    // the approved record comes first: the revoke job reads it, so a
grant
    // can never exist without its triples and its end
    markApproved(request.id, approved_hash: hash)

    // audit next: if the log is unreachable, no grant is made
    try:
        auditLogAppend({ actor: request.approver, action: "grant",
                         subject: request.requester_login,
                         request: request.id, fix: request.fix_id,
                         approved_hash: hash, triples:
request.privileges,
                         expires_at: request.expires_at })
    on AuditLogUnavailable:
        markRefused(request.id)
        raise GrantRefused

    // the session's end follows the approved window, as "Ending the
grant" says
    setSessionEnd(request.requester_login, request.expires_at)

    // separate systems, no shared transaction: a failure part-way
leaves an
    // approved record with no grant or only some triples
    for each p in request.privileges:
        grant(p.privilege, on: p.table, columns: p.columns,
              to: request.requester_login)

    // the window may end between the check and the grants: end it now
    if request.expires_at <= now():
        revokeRequestAccess(request.id, from: request.requester_login)
```

按失败模式来读它。如果审计日志宕机，申请被拒绝，什么都不授予：无法记录的授予无法被评审，我宁可停下一次修复，也不愿造出一次不可见的修复。如果 gateway 在记录和日志条目之后、授予之前崩溃，申请就是已批准但没有任何效果，这不会暴露任何东西。如果它在三个三元组中的两个之后崩溃，登录持有两个，记录里是三个，之后的撤销会收回全部三个，因为撤销一个从未授予的三元组是无害的。

最后几行处理一个竞态。时间窗口不会因审批而重新开始，所以形态检查会拒绝在 `expires_at` 之后到达的审批；剩下的是在窗口关闭前一刻通过的审批，这几行会立即收回它的授予。

审计条目除三元组外还携带修复和哈希，所以审批人、语句和时间窗口都在同一条记录里。支付数据和财务报告方面的监管制度往往会问：谁批准了对生产数据的变更，依据是什么，持续多久；这条记录能回答这三个问题，而法务与合规负责对此作出解释。

结束授予

授予的结束分三层，没有任何一层被信任为唯一的一层。

第一层是会话。gateway 在 `expires_at` 结束会话并移除登录，不涉及任何定时作业：限定访问的时钟，属于创建访问的那个组件。绑定主库的会话开启时有第 7 章规定的两小时上限，所以在授予时，无论来自 `grantJitAccess` 还是 `declareBreakGlass`，gateway 都会把会话的 `ends_at` 重置为申请的 `expires_at`，由第 7 章的 `endSession` 在该时刻关闭会话；因此超过两小时的时间窗口不会被截短。

```
function endSessionAtExpiry(session):
    // one live request per session, so its end is the session's end
    waitUntil(session.request.expires_at)
    // never raises; a failure leaves the record for the next sweep
    endSession(session.login, "expired")
```

真正的结束动作是 `endSession`，它是唯一关闭连接、撤销会话各申请所授予的内容并移除登录的地方，`sweepSessions` 调用的是同一个函数（第 7 章）。

第二层是撤销作业，它读取同一份记录，并移除申请所授予的内容。`revokeRequestAccess` 的工作由 `access_request_privilege` 推导而来：对于 JIT 申请，它撤销每个存储的三元组；对于 break-glass 申请，它撤销角色 `human_breakglass`。它不读取可能与记录不一致的另一份清单，并把缺失的登录视为已撤销。

```
function revokeExpiredGrants():
    // expiredGrants: kind jit or breakglass, status approved,
expires_at
    // passed, not yet revoked
    for each item in expiredGrants(before: now()):
        try:
            // revokes what the request granted: its stored triples
for "jit",
            // the role human_breakglass for "breakglass"
            revokeRequestAccess(item.request_id, from: item.login)
        on LoginNotFound:
            // the login went with the session; nothing is left to
revoke
        on RevokeFailed:
            // an expired grant still live is an incident; retried
next run
            notify("platform", "expired grant still active: " +
item.id)
            continue
        markRevoked(item.request_id)
        try:
            auditLogAppend({ action: "revoke", actor: "revoke-job",
                             subject: item.login, request:
item.request_id })
        on AuditLogUnavailable:
            // the revoke is done; the log catches up from the retry
queue
            queueForRetry(item.request_id)
            notify("platform", "revoke not yet logged: " + item.id)
```

审计日志宕机时，撤销仍然继续。这种不对称是有意的：没有日志条目的授予是一种暴露，而没有日志条目的撤销只是一个之后可以填补的缺口。

第三层是对账，它是另一种性质的东西。它不收回访问，而是从外部询问前两层是否完成了它们的工作：

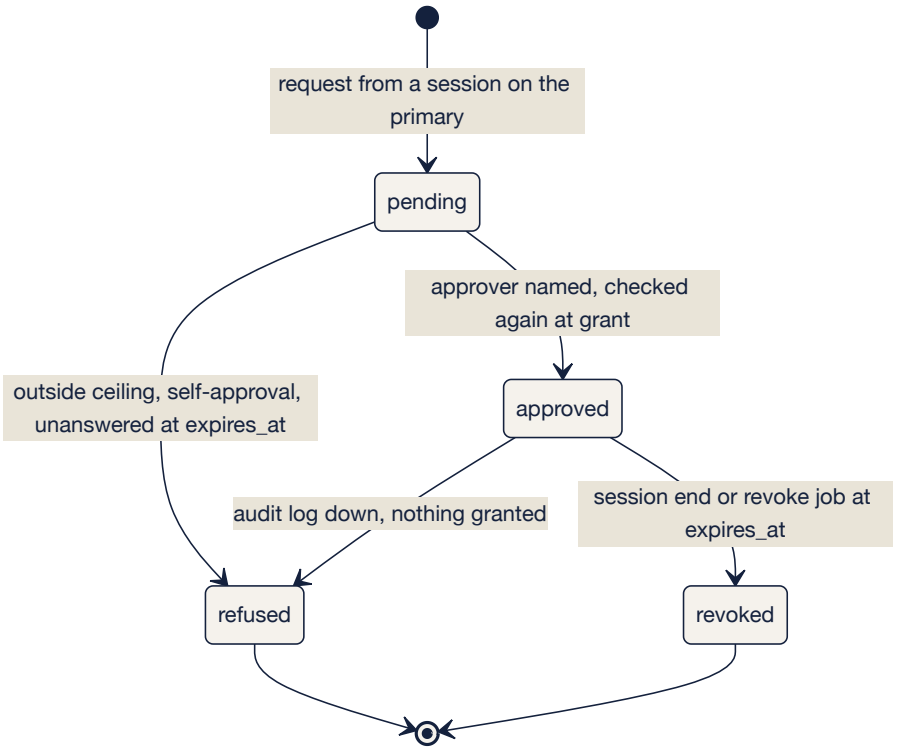
```
function checkOverdueGrants():
    // scheduled separately from revokeExpiredGrants, so one failure
    // cannot hide the other
    if count(expiredGrants(before: now() - graceMinutes)) > 0:
        notify("platform", "grants past expiry and not revoked")
    if now() - lastSuccessfulRun("revokeExpiredGrants") > maxRunAge:
        notify("platform", "revoke job has not completed recently")
```

与之并行的是第 3 章的漂移检查，它读取数据库实际持有的授予，并与申请中未到期的三元组比较。背后没有有效申请的直接授予、成为上限成员的登录、没有未结 break-glass 申请的 `human_breakglass` 成员：每

一种都会被标记，即使撤销作业报告成功。逾期检查关注作业本应处理的记录；漂移检查关注作业本应修改的数据库。一个报告“完成”却什么都没改的作业，只有第二种检查能抓住。

对账自身不修复任何东西，我会一直保持这样：自动撤销未知的授予，是在夜里弄坏生产环境的办法。它通知 platform 团队，由人来决定。

申请经历的状态很少，把它画出来就能看出每一层在哪里起作用。请看 `approved` 的出口：一次撤销，或者审计日志宕机、未授予任何东西时的一次拒绝；从 `revoked` 不会有任何东西回来。



自我审批，以及值班不设自动审批

自我审批一律拒绝。`checkJitRequest` 抛出 `SelfApprovalNotAllowed`，没有任何东西可以覆盖它，资历和紧急程度也不行。紧急情况有另一条路径，那是本章的后半部分。

自动审批是另一条诱人的捷径，因为夜里的摩擦最大。对于读取，我不会去构建它：每个会话登录已经持有 `human_readonly`，而 JIT `SELECT` 额外带来的只有未脱敏显示的 Confidential 列，这恰恰是审批人应当查看

的。写操作会改变服务接下来据以行动的内容，而机器无法判断 `status` 是不是正确的列。写操作交给人类审批人，把精力花在让审批人容易联系上。

日后若增加自动审批，用于确实带来新内容的读取，规则必须要求明确的列清单，而空的或缺失的列清单必须明显地拒绝申请。在 secrets-tables 回退下，表级读取根本没有列清单，所以把“没有点名的 Confidential 列”视为可接受的规则，会批准该表中的每一个 Confidential 列。

来自自动化的申请，以及作为代码的策略

想要写入的作业、runbook 脚本或 AI agent，与任何其他 principal 一样：它在自己的 workload identity 下打开会话，指明一个修复，并等待人类审批人，记录中同时写明自动化本身和触发它的人（第 9 章说明它在会话中可以做什么）。审批逻辑本身就是代码。`withinCeiling`、`chooseApproverPool`、`maxJitWindow` 和形态检查都存放在仓库里，经过评审，并位于漂移检查的基线之中；在控制台里手工编辑的上限，就是更高一层的原始问题。

滥用情形

有三种情形被上述控制所限定，却没有被其中任何一个消除。申请人与审批人之间的串通，被缩小到记录中的列、语句和时间窗口，而这一对人在审计条目中可见；访问评审（第 14 章）则是发现互相审批的频率远超随机的一对人的地方。被社会工程的审批人（“现在批准 req-0412，这批数据卡住了发薪”）由检查清单应对：阅读 gateway 界面上的语句，核对理由中的告警，对通过其他渠道发来的内容一概不予答复。被劫持的单点登录 token 无法在 JIT 会话中写入，因为 gateway 只从 runner 转发写操作，而 runner 会拒绝从未被批准的哈希。残余风险存在于已授予三元组内的读取，以及没有审批人的 break-glass；对其设限的，是声明时的升级检查和告警。

Break-glass 访问

break-glass 是带有明确日志集和强制评审的应急访问。这个名字来自玻璃罩着的消防开关：任何需要的人都够得着，而打破它，所有人都看得见。

在本书的模型中，它就是角色 `human_breakglass`，也是唯一能读取 Restricted 列的人类角色。它读取每一张表，并在除两张append-only 表之外的每一张表上写入 `INSERT` 和 `UPDATE`；在 `bank_accounts` 和 `payment_tokens` 上它只写入 `UPDATE`，且只针对非 Restricted 列。它从不写入 `ledger_entries` 或 `audit_events`，因为没有任何人类角色写入它们。它不是常规修复的途径，也不是数据修正的路径：数据修正是有计划、经评审、有边界的（第 10 章），而 break-glass 在设计上这几点都不是。

break-glass 申请是自我声明的，没有审批人。会话中的人声明存在紧急情况，给出理由，gateway 把申请标记为已批准，并把该角色授予同一个会话登录，直到 `expires_at`。同一个撤销作业和同一个会话计时器会把它收回。

对 JIT 而言，审批人就是全部控制，所以没有审批人的应急角色看起来可能有些奇怪。break-glass 存在的意义，是审批人池没有答复的情形，而增加一个审批步骤只会重现这种失败。控制从访问之前移到了访问周围：声明时的告警、下文的日志集、简短而固定的时间窗口以及无一例外随后进行的评审。

```
function declareBreakGlass(session, reason):
    // a hijacked token alone must not be enough to declare
    requireFreshMultiFactor(session.principal)
    // one live request per session: a break-glass declaration
supersedes it
    supersedeLiveRequest(session.login)
    // the window is the gateway's fixed maximum; there is no argument
for it
    window = breakGlassMaxWindow
    request = newRequest(kind: "breakglass", requester:
session.principal,
                         requester_login: session.login, approver:
null,
                         reason: reason, expires_at: now() + window)
    markApproved(request.id)
    try:
        auditLogAppend({ actor: session.principal, action:
"breakglass",
                         subject: session.login, request: request.id,
                         expires_at: request.expires_at })
    on AuditLogUnavailable:
        markRefused(request.id)
        raise GrantRefused
    setSessionEnd(session.login, request.expires_at)
    grant(human_breakglass, to: session.login)
    // paged after the grant: a paging outage must not block an
emergency;
    // the review queue below is the net for a page that never arrived
    queueReview(request.id, reviewer: "not-the-requester")
    notify("security", "break-glass declared: " + request.id)
    notify("platform", "break-glass declared: " + request.id)
```

该代码块中有三个决定需要说明。

时间窗口由 gateway 固定，在 Ledgerly 的示例中是两小时。不提供更长的窗口：因为“我还需要久一点”会把例外变成习惯。如果紧急情况超出了窗口，当事人再次声明，第二次声明会再次告警；这就是有意设置的摩擦。

对审计日志的检查与 JIT 相同。如果日志宕机，声明被拒绝。我接受这个代价，因为没有记录的 break-glass 会话，是评审唯一无法还原的东西，而评审正是让这个角色可以容忍的原因。当正常路径和日志同时宕机时值班该做什么，属于第 15 章。

告警是向安全联系人和 platform 团队发出的通知，告知有人打开了 Ledgerly 所拥有的最宽的访问，而不是一次许可请求。在健康的系统中，它罕见到足以被人阅读；如果它每周都触发，就不会再有人读它，说明 break-glass 被用在了本应有一条专门设计的路径的事情上。

Break-glass 日志集

在被定义之前，“详尽日志”不是一个标准，所以对 break-glass 会话，日志集是这四项：

- 每条语句，以指纹形式记录，字面量已遮蔽，另有参数的哈希，以及开始时间、结束时间和受影响行数。遮蔽在这里最重要：该会话读取 Restricted 列，日志中的一个字面量会把它们复制到一个没有列级保护的地方。
- 随事件发生而写入的条目。每条语句在运行时就追加到审计日志，而不是在结束时汇集，所以被切断的会话也会留下记录。
- 每次写入附一行说明。gateway 拒绝没有说明的写入，这只花几秒钟，却让评审人在语句旁边看到意图。
- 与任何其他会话相同的记录：登录、人、申请和理由，这样第 14 章的三份记录可以连接起来。

还有一个风险，在不同状态下有所不同。今天，在没有 gateway 或 gateway 只经手凭据的情况下，break-glass 会话是唯一仍可能对其可写的表执行无界 `UPDATE` 的地方。时间窗口、行数、条目和评审限定并暴露了损害，但没有阻止它。在目标状态下，第 9 章的 gateway 以关闭自动提交和行数上限的方式运行写会话，它的矩阵在这条通道中拒绝无界写入，没有说明的写入也被拒绝。仍然可能发生的，是一次落在上限之内却命中了错误行的写入。这里的上限最难确定大小，因为应急按定义不是计划好的任务；在策略设定上限之前，无界写入是一个被接受的风险，也是评审存在的理由。

事后评审

每一次 break-glass 的使用都要评审，评审人不是使用者本人。评审不是聆讯；它提出的是设计评审会提出的问题：这是正常路径无法应对的紧急情况吗；会话读取或写入了什么；它读取了 Restricted 列吗，有必要吗；JIT 申请能否满足同样的需要，如果可以，什么能让该申请足够快？

最后一个问题是有回报的那个。每一次评审，要么确认 break-glass 是正确的工具，要么发现正常路径中使它成为唯一工具的缺口。修补这个缺口（缺少审批人、窗口太短、上限没有提供的表）会减少下一次使用。评审也需要自己的时钟：

```
function checkBreakGlassReviews():
    for each r in breakGlassRequests(endedBefore: now()):
        review = reviewFor(r.id)
        if review is null and now() - r.expires_at > reviewDeadline:
            notify("security", "break-glass review overdue: " + r.id)
        if review is not null and review.reviewer == r.requester:
            notify("security", "break-glass reviewed by its own user:
" + r.id)
```

无人执行的评审是一个悄悄不再存在的控制，所以它的截止期限会触发告警。

Gateway 不在时

人只能通过 access gateway 访问数据库（第 7 章），而 JIT 和 break-glass 的授予都是由 gateway 授予它所创建的会话登录。因此 gateway 宕机同时也是 break-glass 宕机。这是设计中的一项权衡，第 7 章从 gateway 一侧陈述了同样的决定。

另一种做法是开第二道门：gateway 之外的应急登录。在需要它的那个夜晚，它很有吸引力，而在其他每个夜晚都很危险，因为不经 gateway 也能工作的凭据，也就不经过它的日志、身份和到期机制；它就是换了个好听名字的共享管理员密码。

我宁可让 gateway 难以丢失，也不愿通过削弱其他一切来让它的缺席可以承受：不止一个实例，依赖很少（审计日志、申请记录），故障经过演练，以及值班人员最先看到的健康信号。如果所有实例同时宕机，就没有任何人类数据路径了，我不会在这里发明一条。第 3 章的封存的凭证用于引擎层面的工作，它需要一条第 7 章对人关闭的网络路由，而且它绕过append-only 和 Restricted 的保护，所以它不是数据访问路径。第 15 章会回到这种情况下该怎么做。

这个选择并不是没有代价的。gateway 脆弱的团队，在它成熟之前，完全可能想要第二条路径。那就写下它是什么、谁能取得它的凭据、它的使用如何告警、什么会让它退役：有负责人和结束日期的路径是一个决定，没有的则是遗留物。

引擎没有列级授予时

以上一切都假定引擎能够对具名的列授予权限。第 3 章介绍了本书为无法做到这一点的引擎使用的唯一回退：Restricted 列存放在单独的表 `bank_account_secrets` 和 `payment_token_values` 中，位于上限之外，

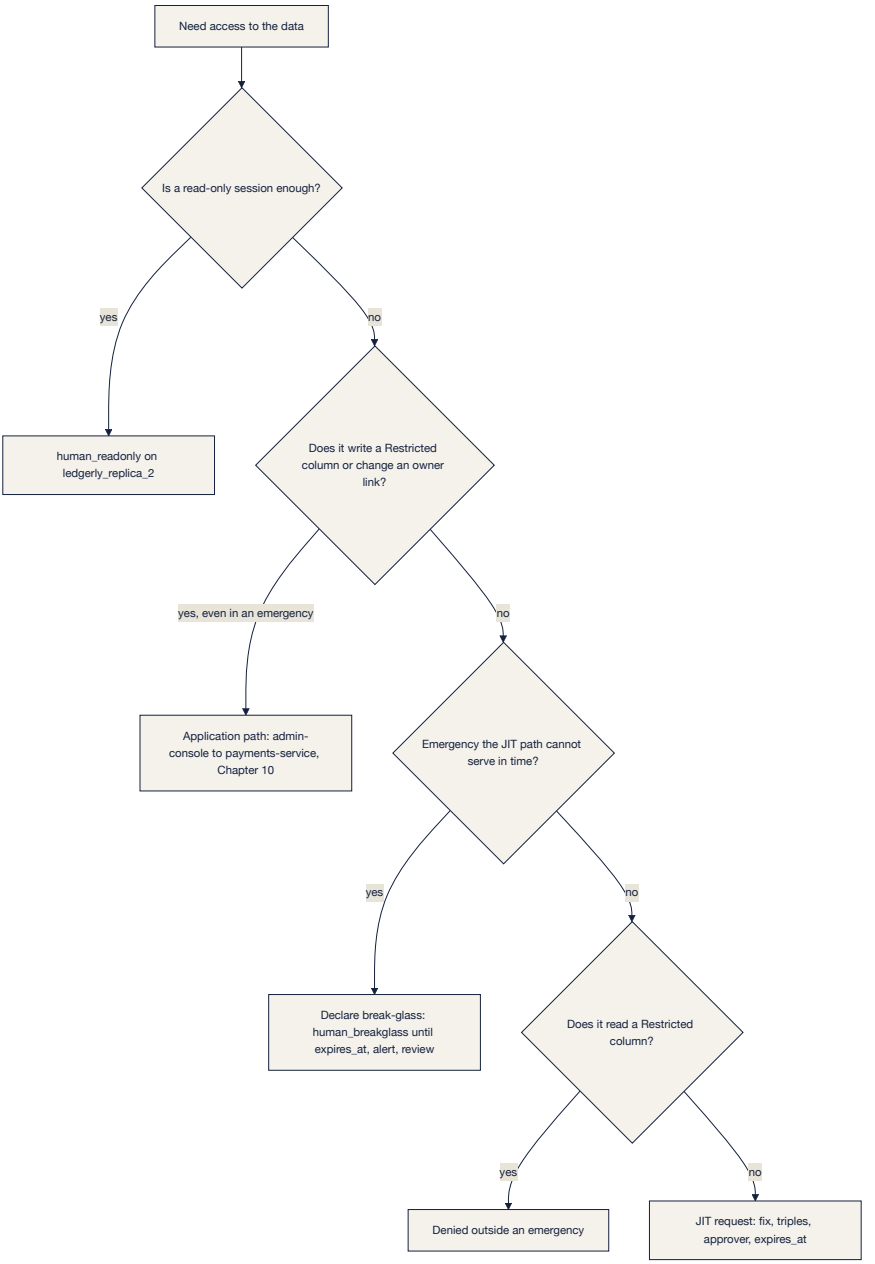
只有 `app_payments_service` 和 `human_breakglass` 能够访问。这样主表不含任何 Restricted 列，所以对 `bank_accounts` 或 `payment_tokens` 的表级 JIT 授予无法触及其中任何一列。这样的申请没有列清单：`columns` 为 null，三元组读作“对 `payment_tokens` 的 UPDATE”。

代价落在审批人身上，他批准了该表所有非 Restricted 列，包括 Confidential 列。如果申请人想修改 `expires_on`，审批人同时也在批准 `provider` 和 `last_four`。审批界面必须用文字写出“该表所有非 Restricted 列”，而审批人阅读的语句承担了更多的分量。团队预期会经常打开的列组，值得拥有自己的小表。

这个回退不改变 Restricted 数据的规则。人仍然绝不用原始 SQL 写入它；变更要经过 `admin-console` 和 `payments-service`，并在第二个人批准之后进行（第 10 章）。

选择路径

凌晨三点的人需要知道该走哪扇门，而答案应当以一个简短的流程写进 runbook。下图就是它的形态。请看没有任何人类数据路径服务的两端：写入 Restricted 列，即使在应急中也走产品路径；而在应急之外读取它，则被拒绝。设计把这点写明，而不是把决定留给身处压力之下的人。



凌晨三点的 Ledgerly

回到 Robin。卡住的行已在常规会话中数过，数字与 runbook 相符。写操作就是绑定到 `fix-0093` 的申请，期限一小时，引用那条未关闭的告警。`chooseApproverPool` 依次选出第二值班工程师和团队负责人，呼叫

发给第一个人。旧的 Ledgerly 在这里会打开共享密码。新的 Ledgerly 有一段等待，而这段等待就是设计在起作用。

如果无人应答，runbook 会提出那个决定一切的问题：这是审批人池无法及时应对的紧急情况吗？卡住的提醒可以等到早晨，所以 Robin 在告警上留下一条备注，然后停手，这是旧密码从不允许的结果。

假设情形相反：`payments-service` 正在故障，所有客户的支付收款都已停止，而诊断需要一个任何 JIT 申请都不得包含的 Restricted 列。这就是 break-glass 的情形。Robin 以那条告警作为理由声明；gateway 在其固定的时间窗口内授予 `human_breakglass`；安全联系人和 platform 团队收到通知。第二天早晨，不是 Robin 的某个人阅读日志，提出评审问题，其中之一是为什么诊断需要一个 Restricted 列。答案也许会促成下个季度一个更安全的诊断视图。

权衡与成本

摩擦是代价，必须花在有回报的地方。一份 JIT 申请，最好的情况下花申请人一分钟来写、花审批人一分钟来读，最坏的情况下要叫醒一位工程师。对于每季度出现两次的任务，这很便宜。对于每天出现两次的任务，它就是披着运维外衣的产品功能，第 10 章说明了如何把它移入 `admin-console`。

绑定到修复需要准备。修复必须先写下来才能提交申请，这比单独的授予要慢。这是批准将要运行的内容，而不是可能运行的内容所付出的代价。

审批疲劳是设计的失败，而不是审批人的失败。每次都在几秒内、由同一个人给出的审批是表演：要么申请微不足道，`human_readonly` 就应当涵盖它们，要么审批人已经不再阅读。补救办法是更小的申请、显示授予将会是什么的界面、了解这些表的审批人以及大到没有人是默认人选的池。有两个信号属于访问评审：申请被拒绝或退回的频率，以及同一个审批人答复的频率。拒绝率为零，既可能意味着申请很好，也同样可能意味着没有人在看。

值班工程师最先承担成本。审批人被叫醒，申请人在等待，团队学习着一份很少使用的 runbook。把审批人池做成轮值，公布出来，并确保池中的人具备阅读 `invoices` 或 `payments` 相关申请所需的背景。

gateway 是每一次写入的依赖。撤销作业、审计日志和申请记录都是授予的依赖；其中任何一个宕机，授予就会停止，这限制了暴露，也拖慢了事故响应。第 7 章说明了如何降低这种概率。

对账会产生噪声。漂移检查和逾期检查会标记合法的情形，例如晚了一分钟运行的撤销。调整宽限期，使一次标记有意义；被静音的检查比没有检查更糟，因为团队以为自己被覆盖了。

Break-glass 有社交成本。声明 break-glass 的人，是在告诉安全联系人正常路径失败了。如果评审感觉像审判，人们在真正的紧急情况中就会犹豫；如果它是设计评审，他们就不会。无责评审让这个角色保持可用。

何时不该使用。一个只有四名工程师、一个低风险数据库、没有 Restricted 数据的团队，第一天并不需要审批人池和对账作业。一个带有理由、时间限制、确认人和记录的手工流程，在一段时间内可能就够了，前提是时间一到授予就被移除，并且有人检查。按失败出现的顺序来构建这个模式：先是到期，然后是记录，然后是审批人，然后是告警。

常见错误

- **把上限角色“为了这一项任务”授予某个登录**。这样每个 JIT 会话都持有整个上限，申请的范围就成了摆设。把三元组授予会话登录，让 `human_jit_write` 保持为没有任何登录属于它的上限；漂移检查会标记其成员。
- **批准三元组而不批准语句**。列清单简短，容易阅读，而批准它就是批准一种能力。把写申请绑定到已评审的修复，由修复推导三元组，并从修复的记录中取得 `approved_hash`。
- **从单独的清单撤销**。硬编码的表清单迟早会漏掉申请所授予的某张表。撤销应由 `access_request_privilege` 推导，也就是驱动授予的那份记录。
- **只信任撤销作业**。一个连续运行一年每晚都正常的作业，很容易让人不再关注。给会话设置它自己的结束时间，并运行独立的检查，把授予与申请进行比较。
- **在授予之后才写审计条目**。如果日志宕机，就会存在一个没有人能看到的授予。先写记录，再写日志条目，然后授予，让失败的日志条目阻止授予。
- **对值班申请自动审批**。读取不需要申请，写入需要第二个人。用一个好的审批人池来缩短等待。
- **把 break-glass 做得长到令人舒适**。在一个工作日之内永远不会过期的窗口，会变成一个日常使用的角色。在 gateway 中固定最大值，并让第二次声明再次告警。

- **不定义日志集**。评审人无法判断一个从未规定过记录内容的会话。在第一次使用之前就指明日志集。
- **因为没出问题就跳过评审**。评审是修补正常路径的地方。给评审设定截止期限，过期就触发告警。
- **为 gateway 的糟糕日子建造第二道门**。gateway 之外的应急登录会绕过它的身份、日志和到期机制。先投入于可用性；如果第二条路径不可避免，就给它一个负责人、一个告警和一个结束日期。

要点

- 常驻访问的成本在它没有被使用的日子里最高，所以有用的问题是：现在谁需要访问，针对哪项任务，以及系统如何知道任务已经结束。
- `human_jit_write` 是上限，绝不授予任何登录；写申请指明其已评审的修复（`fix_id`），三元组由修复推导，审批人批准的是语句，runner 必须与其哈希（`approved_hash`）匹配。
- 撤销由驱动授予的同一份 `access_request_privilege` 记录推导，由会话自己的结束时间强制执行，由撤销作业兜底，并由一个两者都不信任的独立对账来检查。
- 写入需要四眼原则：审批人既不是申请人也不是修复的作者，在新的多因素检查之后于 gateway 的界面上答复。值班不设自动审批。
- break-glass 是自我声明的 `human_breakglass` 会话，具有固定的最大时间窗口、明确的日志集、声明时的告警以及由使用者之外的人执行的强制评审。它既不是常规路径，也不是数据修正的路径，它被接受的唯一风险，今天是无界 `UPDATE`，而在目标状态下，一旦第 9 章的 gateway 强制执行上限，则是上限之内的错误行。
- 没有获准的快速路径的设计，会长出一条未获准的路径，所以应急路径要经过设计、记录和演练，它对 gateway 的依赖是需要点明的成本。
- 引擎无法按列授予时，secrets-tables 回退把 Restricted 列挡在每一份申请之外，而审批人批准的是整张表的非 Restricted 列。

授予，即使是被绑定的授予，说明的也是会话可以触及什么、持续多久，而不是其背后的语句被检查得多仔细。第 9 章转向这一点：gateway 在语句运行之前对它做了什么。第 10 章随后转向工作本身：如何执行数据修正，使正确的行被修改、错误的行不被修改，并且第二个人能看到原因。

第 9 章 — 受控执行：查询分析与危险操作

Ledgerly 的支持团队在聊天工具里有一个助手。Sam 让它作废一批因计费重试产生的重复发票，助手以自己的身份运行，起草了一个修复：对 `invoices` 的一条 `UPDATE`，`WHERE` 子句使用该次重试的批次标签。数据团队的 Jordan 评审了这条语句并批准。语句是有效的 SQL，授予覆盖了它，审批人读过它，Jordan 核对时标签也是正确的。

这条语句匹配到的行，仍然多于修复所指明的那十二行。另有一个作业在两天前复用了这个批次标签，而语句的文本中没有任何地方说明这一点。没有人疏忽：Sam 准确描述了任务，助手写下了被要求写的内容，Jordan 评审了呈现给他的内容。缺少的是“已批准”与“已提交”之间的一个步骤，一个去看语句将对数据做什么，而不只是看它写了什么的地方。

第 3 章到第 8 章决定了谁可以持有授予、经由哪条路径、持续多久。本章加入五部分模型中的执行部分：gateway 逐条处理语句的步骤，而不是交出一个会话，再信任持有会话的人。它回答的问题是：当单靠审批无法说明语句将做什么时，已批准的语句与数据之间应当有什么？

问题及其难点

session handout 给人一个登录，让其客户端直接与数据库对话。此后的一切都是客户端的事：键入什么、触及多少行、是否当场提交。授予限定了可触及哪些表和列，而在这个范围之内，单条语句做什么，没有任何东西限定。在起始状态下，桌面客户端提交键入的任何内容，并立即提交。

受控执行颠倒了这种安排。语句交给一个不是该人客户端的组件，由该组件决定它是否运行、受哪些限制、何时提交。第 7 章的 gateway 已经在数据路径上，所以它是做这件事的自然位置。语句仍然由人来写，或者由自动化来写；变化在于，没有任何语句不经过下一节的循环就能触及数据。

这是一个谱系，而不是一个开关。最廉价的形式是 session handout，加上 gateway 强制执行的几项设置，例如语句超时和行数上限。下一种形式在转发之前检查每条语句。受控程度最高的形式逐条批准语句，并由 worker 运行，这样人根本不持有用于写入的数据库会话。有些工具批准的是每条查询，而不是一个时间窗口。在撰写本书时，例如 QueryProxy，由第二个人批准语句，由 worker 运行，这样开发者从不持有数据库凭据。这用交互式会话的速度，换来了绑定到确切语句的审批，其他查询审批工具的工作方式与之类似。

受控之所以困难，有三个原因。第一，SQL 表达力强：同样的效果可以用多种方式写出，一条语句是否“危险”，往往取决于检查器没有读过的数据。第二，每项检查都有代价，体现在延迟、误拒以及学会绕过它的人身上。第三，被过度信任的检查器，会变成无人评审的东西。本章的立场是：语句分析减少错误，但不是授权边界。授予才是边界。gateway 的任务，是让常见的错误语句难以被意外运行，并让每次写入都带有一个由写入者之外的东西来强制执行的界限。

Gateway 对每条语句的循环

会话发送的每条语句，无论来自人还是其他，都经过同一个循环。顺序比任何单个步骤都重要，所以在解释各部分之前，先以伪代码呈现。

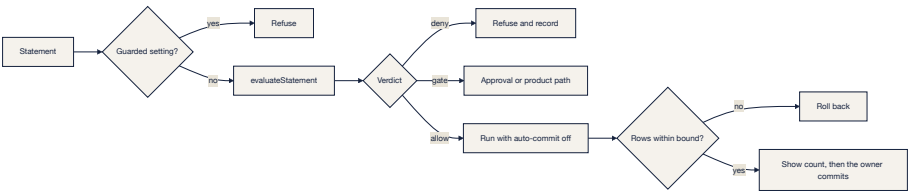
```
function handleStatement(session, statement):
    if statement.changesSetting(guardedSettings):
        return refuse(session, "guarded setting")
    verdict = evaluateStatement(session, statement)
    // the record comes first: a statement that cannot be recorded
does not run
    try:
        recordIntent(session, statement, verdict)
    on RecordUnavailable:
        return refuse(session, "record unavailable")
    if verdict.action == "deny":
        return refuse(session, verdict.reason)
    if verdict.action == "gate":
        return routeToGate(session, statement, verdict)   // approval,
or the product path
    if statement.isRead:
        return streamRows(session, statement, cap: session.rowCap)
// cursor closes at the cap
    // auto-commit is off: the runner owns the transaction of a fix,
the gateway owns the rest
    tx = transactionFor(session)
    result = execute(tx, statement)
    tx.rowsSoFar = tx.rowsSoFar + result.rows      // counted across
every statement of the fix
    if tx.rowsSoFar > verdict.bound:
        failTransaction(tx)                        // its owner can
only roll back
        return refuse(session, "more rows than approved")
    showRowCount(session, tx.rowsSoFar)            // before COMMIT,
while undo is free
    if tx.ownedByGateway:
        if not countAccepted(session, tx.rowsSoFar, verdict.bound):
            rollback(tx)
            return refuse(session, "count not accepted")
        commit(tx)                                 // the commit point
for a person or a migration
    recordOutcome(session, statement, result)      // failure is
queued and alerts; a commit stands
    return result
```

受保护设置。gateway 在打开每个会话时就应用其限制：语句超时、默认只读、按会话类型确定的角色和连接数上限（第 5 章）。客户端可以发送 `SET` 来解除其中任何一项，所以 gateway 维护一份受保护设置的清单，并拒绝修改其中任何一项的语句。“不可覆盖”这个说法，只有在有机制强制执行的地方才当之无愧。作为角色默认值设置的超时，会话可以将其解除，除非 gateway 拒绝该 `SET`，或者平台将其固定，所以两者都没有的限制只是默认值，而不是保证。代价是，有些客户端在连接时就会发送这类设置，在有人把它们列入清单之前就无法工作。

行数上限。对于读取，gateway 在流式返回行的同时计数，在上限处关闭游标，告知当事人，并记录该事件。上限并不是说一个人一天之内只能看这么多行，决心足够的人可以分片翻阅整张表。它的意思是，单条错误的查询不会把整张表拉进一台笔记本电脑。大量的合法提取属于第 13 章的导出路径和第 12 章的脱敏分析副本，由层级决定谁来审批。

写入时关闭自动提交。写会话在显式事务中运行每一次写入。对于人或迁移，事务由 gateway 打开；对于数据修正，事务由第 10 章的 runner 打开，gateway 把修复的每条语句转发进这同一个事务。无论哪种情形，gateway 都会跨所有语句统计受影响的行数，并与界限比较，而 `COMMIT` 的决定仍属于事务的所有者：一个对修复的每条语句各自提交的 gateway，会破坏 runner 的单一事务，也会让它的 dry run 变成真正的变更。当行数超过界限时，gateway 拒绝该语句，并把事务标记为失败，这样所有者只能回滚。这是第 1 章中可逆性的廉价形式：本想只触及一行却报告四百行的更新，会在撤销仍然免费时被拦下。对于交互式会话，“显示”意味着人看到行数并确认。对于 runner 和自动化，“显示”意味着 gateway 把行数与审批所记录的界限比较，超出即让事务失败。代价是多出的一步，以及事务的锁在人做决定期间一直被持有；无人应答的确认提示会占着行锁，所以该提示设有超时，超时即回滚。

先记录。gateway 在语句运行之前写入意图，在运行之后写入结果。没有横跨 gateway 的日志和数据库的事务，所以保证靠顺序来承担。如果意图无法写入，语句就不运行。如果提交之后结果无法写入，提交依然有效，条目排入队列，并通知 platform 团队；只有在一切正常时才能工作的日志，当不起这个名字。日志存放在哪里，见第 14 章。



对语句分类

gateway 在 `evaluateStatement` 中决定如何处理一条语句。它按语句做什么、在哪里做对其分类，查阅隔一节之后的矩阵，对于矩阵允许但带界限的写入，找出界限并将 planner 的估计与之比较。每一次拒绝和每一次 gate 都来自矩阵；除解析器的返回之外，该函数没有自己的提前返回。

```
function evaluateStatement(session, statement):
    parsed = parse(statement)
    if parsed.failed or parsed.isMultiStatement:
        return deny("cannot be analyzed")            // fail closed
    // nine classes, one per matrix row; the most specific class wins
    class = classify(parsed)
    cell = matrix[class][session.kind]               // deny, gate,
allow or allow with a bound
    if cell.action != "allow":
        return cell                                  // a deny, or a
gate with its route
    if class == "read":
        return allow(cap: session.rowCap)
    if cell.bound == "fix":
        binding = checkBinding(session, statement)   // the next
statement of the approved fix
        if binding.action == "deny":
            return binding
        bound = binding.bound
    else:
        bound = session.writeBound()   // the break-glass row cap, or
the backfill rule's limit
    if bound is none:
        return deny("no approved count")
    estimate = plannerEstimate(parsed)               // asks the
planner; runs nothing
    if estimate > bound * tolerance:
        return deny("estimate exceeds the approved count")
    return allow(bound: bound)
```

有三个选择值得辩护。第一，解析器读不懂的语句会被拒绝。如果 gateway 运行了自己无法分析的内容，分析就只适用于那些从来不成问题的语句。把成批语句放在一个字符串里发送的客户端，会因这条规则而失效，这是迁移中需要点明的成本；对策是逐条发送。

第二，分类看的是效果，而不是人或工具的名字。迁移中的 `DELETE` 与 JIT 会话中的 `DELETE` 属于同一个类别，由矩阵决定每条通道对它可以做什么。用一个分类器和一张表，而不是每条路径各有一套规则，才使策略小到足以被评审。

第三，planner 的估计与预期行数做比较，但只作为提前拒绝。planner 根据统计信息作估计，而统计信息会过时；估计偏低与偏高同样容易。估计让 gateway 能在取锁之前拒绝明显错误的语句。真正的控制是实际行数，在 `COMMIT` 之前于事务内部比较。容差是 platform 团队设定并记录的参数，我会从严格开始，只在测得误拒之后才放宽，而不是靠猜。

静态分析看不到什么

语句的文本，对它在行上的效果说得很少。解析器可以看出缺少 `WHERE`、谓词恒为真、语句触及了不该触及的表，或者调用了某个函数。它看不出一个谓词今天匹配多少行，而这正是开头的问题：一个有效的过滤条件，作用在一个如今匹配到的行数超出任何人预期的标签上。

还有若干情形超出分析所能解决的范围。过滤条件可以有效却很宽，因为数据而不是语法，匹配了一张表的大部分。连接可以使行数成倍增加。视图或函数可以把写入或更宽的读取藏在看似无害的名字后面，而 gateway 看不到角色被允许调用的函数体内部。触发的副作用发生在语句之后，在其文本中不可见。语句还可能在形态上完全正确、在含义上却是错的：按过滤条件是对的行，按意图却是错的行，这是任何工具都无法判断的。

QueryProxy 这样的开源工具在提交时检查语句，拒绝缺少 `WHERE` 或 `WHERE` 恒为真的 `UPDATE` 或 `DELETE`，并限制无界的 `SELECT`；在撰写本书时，它自己的文档把这个解析器描述为一道防御层，而不是授权边界，这也正是本节的要点，其他 gateway 和审批工具采取的方法与之类似。

由此得出的是一种分工。分析廉价而尽早地抓住常见的错误。授予（第 3 章）决定了究竟什么是可触及的，所以被漏掉的语句仍然受角色的限定。事务中的行数检查，抓住数据与作者想法不一致的情形。审批，有第二个人阅读语句和预期行数，在有些时候能抓住形态正确而意图错误的情形。这些都不是完备的，设计依靠的是它们的重叠，而不把其中任何一个宣称为边界。

同样的局限也塑造了第 13 章的导出层级。gateway 根据查询所读取的源列，而不是结果列的名称，来计算导出的层级，无法解析的源按最高的可导出层级计；如果连这一点也无法确定，导出就被拒绝。这与解析器遵循的失败即关闭规则相同：分析读不懂的东西，不予放行。

Deny、gate 和 allow 矩阵

每个语句类别在每条通道中都得到三种答案之一。**Deny** 表示 gateway 拒绝并记录该拒绝。**Gate** 表示语句不按发送时的样子运行：它需要一个尚未获得的审批，或者应当走另一条路线。**Allow** 表示它在上述循环的限制下运行；某个单元格写着“bound”时，写入只在界限之内才被允许，该界限来自单元格所指明的来源，实际行数在 `COMMIT` 之前与之比较。列是本书已经描述过的通道，行是分类器产生的语句类别。

语句类别	人工读会话	JIT 写入	Break-glass	数据修正 runner	迁移
读取，在行数上限之内	Allow	Allow	Allow	Allow	Allow
有界写入，在已批准行数之内	Deny	Allow，绑定到修复（仅限 runner）	Allow，绑定到行数上限（需附说明）	Allow，绑定到已批准哈希和行数	Allow，绑定到 backfill 规则
无界写入	Deny	Deny	Deny	Deny	Gate：指名审批人
所有者关联变更	Deny	Gate：产品路径	Gate：产品路径	Gate：产品路径	Gate：指名审批人
写入 Restricted 列	Deny	Gate：产品路径	Gate：产品路径	Gate：产品路径	Gate：指名审批人
导出	Gate：按层级	Deny	Deny	Deny	Gate：指名审批人
模式变更（DDL）	Deny	Deny	Deny	Deny	Allow：已评审通道
权限变更	Deny	Deny	Deny	Deny	Gate：platform 评审
写入 `ledger_entries` 或 `audit_events`	Deny	Deny	Deny	Deny	Gate：platform 评审

逐行阅读，就能看出设计。人工读会话没有写入路径，所以每个写入类别在那里都是 deny，唯一的 gate 是导出层级。JIT 写会话只从 runner 转发写入，所以它的有界写入单元格是带有绑定的 allow，而不是一扇敞开的门。break-glass 是唯一一条由人直接写入的通道，所以它的有界写入只在行数上限之下并附有说明时才被允许，所以这个上限的大小是本书中最难确定的大小：应急不是计划好的任务，所以没有任何上限能从任务中选出。上限应根据 break-glass 角色可能需要修复的内容来设定，并在每次使用之后评审。无界写入在那里被 deny。剩下的是一次停留在上限之内却命中错误行的写入，这是一个被接受的风险，评审的存在就是为了将其暴露，如第 8 章所述；第 8 章也说明了在 gateway 强制执行上限之前，今天的情形是怎样的。

所有者关联变更值得单独成行。对 `bank_accounts` 或 `payment_tokens` 的 `customer_id` 执行 `UPDATE`，能通过 JIT 上限，因为该列只是把一行链接到其所有者，并不是 Restricted。但把行移到另一位客户名下，仍然改变了这个付款账户或支付 token 属于谁。因此分类器把它命名为独立的类别，除迁移之外的每条通道，都以一个 gate 来回应，把它路由到产品路径：`admin-console` 询问 `payments-service`，第二个人批准，变更记录在 `audit_events` 中。常规通道中的迁移，对它设 gate，交给指名审批人，如同第 11 章对任何被标记的语句所做的那样，而加急通道会拒绝它；模式变更没有什么理由去重新分配所有权，如果将来确实需要批量重新分配，它是一个有待构建的产品功能，而不是一条有待运行的语句。对 Restricted 列（`iban`、`account_holder_name` 和 `token_value`）的写入走同一条路线，这就是没有任何通道允许人用原始 SQL 写入它们的原因。唯一的迁移情形，是把 Restricted 列移入第 11 章单独的 secrets 表，这会交给指名审批人，而不是被拒绝。

迁移列与第 11 章的描述一致。它的分类器把语句标记出来，交给指名审批人或 platform 评审，仅在加急通道中才拒绝。在这张表里，被标记的语句显示为需要有人点头的 gate，而 allow 的是已评审的模式变更和停留在其规则之内的 backfill。append-only 的表也一样：触及它们的迁移交给 platform 团队，其他通道则 deny。针对禁止语句的第二套规则，会是需要与这套保持一致的第二份策略，所以迁移流水线复用分类器和矩阵，只加上自己的通道规则。

矩阵被刻意保持得很小。九个类别和五条通道可以排在一页上，排得进一页的策略才会被人读。每增加一个新的类别或通道，都是必须有人为之辩护的决定，常见的失败是矩阵为每次事故长出一个例外，直到没有人能说出它禁止什么。我会把它放在版本控制中，像代码一样评审，由 platform 团队担任所有者，数据团队作为写入各行的必需评审人。

把审批绑定到语句

涵盖“对 `invoices` 的一次写入”的审批，会放行对 `invoices` 的任何写入。绑定语句的审批意味着审批指名的是这条语句，gateway 拒绝任何其他语句。第 8 章描述申请，第 10 章描述 runner；本节是把两者连接起来的规则，只陈述一次。

写入申请携带 `fix_id` 和 `approved_hash`。哈希是被评审语句的哈希，修复含有多条语句时，是对整个集合取的哈希，它的来源比检查更重要。它来自修复的合并记录，或来自已批准的申请记录，绝不来自 runner 的

输入。如果 runner 自己提供哈希，runner 就是在检查自己的工作，对语句的更改会连同一个匹配的哈希一起到来。源自第二个人批准的记录的哈希，是写入者之外的某个人作过担保的哈希。

```
function checkBinding(session, statement):
    request = loadApprovedRequest(session.requestId)     // from the
store, not from the caller
    if request is none or request.status != "approved":
        return deny("no approved request")
    if request.fixId is none:
        return deny("no fix named")                      // a SELECT-
only request has no fix and no hash
    fix = loadMergeRecord(request.fixId)                 // the
statements, read from the record
    if hashOf(fix.statements) != request.approvedHash:
        return deny("record differs from the approved hash")
    next = fix.statements[session.transaction.statementCount]
    if normalize(statement) != normalize(next):
        return deny("statement differs from the approved one")
    return allow(bound: fix.expectedRows)                // the count
comes from the hashed record
```

gateway 一次检查一条语句，但哈希涵盖整个集合：它从合并记录中加载语句，确认它们的哈希就是已批准的那个，并且只接受这个顺序中的下一条语句。界限是同一条记录中的预期行数，而不是申请所携带的数字，所以哈希也保护了行数。提交仍属于 runner 的事务，所以含有多条语句的修复要么整体落地，要么完全不落地。

还有两条规则完成这一绑定。在 JIT 写会话中，gateway 只从 runner 的连接转发写入。人自己的客户端仍然可以在已授予的三元组之内读取，但从它发出的写入会被拒绝，否则哈希就成了客户端可以绕开的检查。只有会话无法以任何其他方式写入时，守卫才是一种机制。第二条规则是相反的情形：只持有 `SELECT` 三元组的 JIT 申请没有修复也没有哈希，所以写入没有可以绑定的东西，写入被 deny。读取申请不能通过发送另一条语句变成写入申请。

在 gateway 只经手凭据、客户端直接连接的地方，这一切都无法强制执行，本书对此直言不讳：此时的哈希只记录一种意图，什么也防不了。绑定是数据路径的属性，只存在于 gateway 在其中的地方。

runbook 的情形是带有模板的那种。runbook 的合并记录存储语句模板的哈希，申请只携带键清单。runner 把键代入存储的模板。对这种情形，gateway 把哈希与模板核对，而不是与代入后的文本核对：它确认

提交的语句就是已批准的模板，只是填入了键清单，并以键的数量作为界限，连同申请的三元组一起使用。gateway 绝不会从想要完成这项工作的调用方那里接受语句或哈希。

不是人的调用方

作业、runbook 脚本、ChatOps 机器人和 AI agent 都会提交语句，而它们都不是人。本章的规则是，它们作为各自独立的 principal，走同一个循环。自动化没有更轻的路径，因为自动化正是错误以机器速度重复的地方。

自动化调用方拥有来自平台的 workload identity，而不是人的会话，并且默认只读。它在自己的 principal 下打开会话，所以它的读取与任何人一样，受同样的行数上限、超时和受保护设置约束。如果它想写入，就走与人相同的路线：指明一个修复，由并非触发该自动化的那个人的另一个人担任审批人来批准，它的写入通过带有预期行数的 runner 进行。第 8 章的双人规则在这里同样成立：要求自动化起草修复的人，不能是批准它的人，不是人的调用方也没有自动审批。提出语句的 agent 只是在提出，而审批是人在阅读它。

记录写明双方：提交语句的自动化，以及触发它的人。这是问责的那一半。只写“助手做的”的记录，会让线索止于一台机器；只写“Sam 要求的”的记录，会掩盖是哪个工具把请求变成了 SQL。两个名字都存储，访问评审对两者都要询问。

AI agent 增加了其他调用方没有的一项关切：它的语句由文本生成，而其中有些文本并非它自己所写。工单、日志行或客户消息，都可能进入产生语句的上下文。防御的办法不是更仔细地阅读文本，而是让 agent 的权限不随它读到的内容而改变。它的授予不变，它的写入仍需要人批准确切的语句，通过它不应拥有的路线到达的语句，例如 runner 之外的写入，会被同一个矩阵拒绝。无论 agent 被说服去提出什么，它都不能批准，也不能在没有界限的情况下运行。

对于只读的 agent，有用的限制就是普通的那些：严格的行数上限、较短的超时、以视图或脱敏副本代替 Restricted 列，以及每个 principal 的预算，这样出错的循环耗尽的是它自己的额度，而不是共享的额度。对于会写入的 agent，有用的限制是它的写入少而小，因为每一次都需要一个人。如果这使 agent 慢到对写入没有用处，正确的结论是该任务应当成为 agent 调用的产品功能，而不是放松审批。

权衡与成本

受控有时间成本。每条语句都多经过一个组件，被 gate 的写入要等一个人。对于副本上的交互式读取，增加的延迟很小，大多不可见。对于写入，这正是目的所在：延迟就是第二次审视发生的地方。我不会为紧急情况取消它；我会让 break-glass 通道的 gate 又快又响，并在事后评审。

它有兼容性成本。解析语句并拒绝无法解析内容的 gateway，会使一些客户端、一些驱动和一些习惯失效。拒绝 `SET` 语句、多语句字符串和不常见的语法，会产生一份不再可用之物的清单。这份清单，如果在上线之前就从现有会话的语句记录中收集，会比通过支持工单发现的要短。预留一段时期，让 gateway 记录它的裁决，但只拒绝最明确的情形，并随着误拒下降，逐个类别地转入强制执行。

它需要一个必须保持可用的组件。语句检查位于每次写入的路径上，第 7 章说明了该组件为何应当保持精简。当检查宕机时，限定损害的默认做法是写入停止，副本上的读取继续。失败即开放的检查，教会团队这项检查是可选的，而第一次宕机就会终结这项策略。

它需要一份有人维护的策略。矩阵、受保护设置的清单、容差和行数上限，都是有人负责的参数。没有人负责的策略，会漂向上一次事故所需要的样子。矩阵是需要保持精简、经过评审和纳入版本管理的那一件制品。

还有一种成本是工程无法支付的：分析带来虚假的安全感。拥有会拒绝没有 `WHERE` 的 `UPDATE` 的解析器的团队，可能开始把通过的语句视为不会造成危害的语句。矩阵说明 gateway 拒绝什么，而本章开头展示了通过的语句仍然能做什么。行数检查和第二个人就是为这种情形而存在的，不应因为解析器很好就削减它们。

常见错误

把分析器当作边界。分析器是抓住常见错误语句的一层。授予决定什么是可触及的，让分析器承担这份重量的设计，在语句有效而错误的第一次就会失败。

让 runner 或调用方提供哈希。与语句一同到达的哈希，只能证明语句与它自己匹配。哈希来自合并记录或已批准的申请记录，检查在 gateway 中运行，调用方无法编辑。

写会话中保留自动提交。开启自动提交时，行数只是事后的报告。关闭它，并在 `COMMIT` 之前显示行数，才使错误的行数代价低廉。

解析器读不懂语句时失败即开放。运行无法分析的内容，会使分析只覆盖无害的情形。拒绝它，并保留一份需要更改的客户端的简短清单。

为自动化开一条更轻的路径。因为作业和 agent 受信任，或者因为它们赶时间，就给它们一条绕过循环的捷径，会在量最大的地方拿掉控制。自动化在自己的 principal 下走同一个循环。

矩阵为每次事故增加一个例外。每个例外都是没有负责人的决定。让表只包含现存的类别和通道，遇到不适合的情形，就更改设计或构建功能。

把行数上限当作对暴露的限制。上限限定的是一条查询，而不是一个人一天能读多少。如果关切的是暴露，它属于第 12 章和第 13 章的脱敏与分层工作。

要点

- 执行是语句到达数据的方式：谁提交它、是否有东西先检查它、什么限定它的效果以及如何提交。授予回答不了这些问题。
- 受控执行在语句与数据之间放入一个组件。最廉价的形式是几项被强制执行的设置；最强的形式逐条批准语句，并由 worker 运行。
- 每条语句，无论来自人还是自动化，都经过同一个循环：受保护设置、评估、先记录、读取的行数上限以及写入时关闭自动提交并在 `COMMIT` 之前显示行数。
- 分析减少错误，但不是授权边界。授予才是边界，而在事务内部检查的实际行数，抓住解析器和 planner 抓不住的东西。
- 一张按语句类别与通道划分的小矩阵，给出 deny、gate 或 allow 的答案。所有者关联变更和对 Restricted 列的写入被 gate，并路由到产品路径。
- 审批绑定到语句：`approved_hash` 来自合并记录或已批准的申请记录，绝不来自 runner，而在 JIT 写会话中，gateway 只从 runner 转发写入。
- 作业、runbook、ChatOps 和 AI agent 是 principal，拥有自己的身份、默认只读，并走同一个循环。它们的记录同时写明自动化本身和触发它的人。

循环和矩阵说明 gateway 如何处理语句。它们没有说明如何写出一个值得批准的修复：如何证明行数、保留快照、规划回滚，并让第二个人评审。第 10 章转向这项工作。

第 10 章 — 临时查询与数据修正

设想 Ledgerly 的某个周四下午。`invoice-worker` 中的一个 bug 已经修复并部署，但它在线上运行期间，把一批发票留在了 `sending` 状态，尽管邮件早已发出。支持团队正在询问这些发票，客户的提醒已被暂停，而值班后端工程师 Robin 一行就能看出修复的样子：把受影响发票的状态改回 `sent`，其他一概不动。

Robin 通过 access gateway 打开一个会话。默认会话是只读的，所以这一行无法执行。Robin 需要申请写入访问，而这份申请必须写明表和列。在语句触及主库之前，还得有另一个人先看过它。与此同时时间在流逝，而这次修复小到让周围的流程显得比工作本身还要庞大。

本章为目标状态增加一个步骤：写入通过经过评审的数据修正流程完成，并使用 `human_jit_write` 上限之内的即时授予。第 7 章给人员提供了唯一的一扇门，第 8 章构建了在限定时间内打开这扇门的授予。这里要问的是，门打开之后一个人应当怎么做：如何读，如何写，以及如何让第二类工作足够安全，以至于可以在周四下午从容完成。

在第 1 章的五部分模型中，本章讲的是**执行**。Robin 已经持有这次修复所需的全部凭据，所以风险不在于谁持有凭据。风险在于授予允许做什么，一条语句运行时会做什么，谁先读过它，以及留下了什么证据。本章要回答的问题是：一个人修改生产数据，最小且最易于评审的方式是什么，以及这种修改何时应当彻底不再由人来做？

问题及其难点

每个保存真实数据的系统，最终都会到达一种自身代码无法修复的状态。某个 worker 在两个步骤之间崩溃了。某次迁移让一列只填充了一半。某个客户被误合并到另一个客户名下。应用里没有“把它纠正过来”的 endpoint，因为没人预料到会需要，而编写这个 endpoint 比修复本身耗时更长。于是有人写一条语句，在生产环境上运行。这就是**数据修正**：由人在正常应用路径之外对生产数据做出的变更，用来修复应用无法修复的状态。

第一个难点是，危险的工具和有用的工具是同一个工具。一个能回答“有多少发票卡住了？”的 SQL 会话，只要再多一个关键字，就能改动所有这些发票。在第 1 章的初始状态中，GUI 客户端持有共享的管理员登录，自动提交每条语句，并执行被选中的任何文本。一个问题与一次不可逆的变更之间，只隔着一个词。

第二个难点是，做这项工作的人通常处在压力之下，而且通常是对的。把每次修正都当作威胁的流程，会教会人们绕开它，而绕行的做法比原本的风险更糟：常驻授予、共享密码、笔记本电脑上的一份生产数据副本。把每次修正都当作例行公事的流程，则会让人们不再多看一眼，于是一百条语句里有一条是错的，它就会波及每一行。安全的路径必须同时是最容易的路径。

第三个难点是，风险的单位不是人，而是语句，而语句本身几乎不携带自身规模的信息。`UPDATE invoices SET status = 'sent' WHERE id = 'inv-0042'` 与去掉 `WHERE` 子句的同一行，只差几个字符。文本里没有任何地方说“这会触及 9,000 行”；只有数据能说明这一点，而除非流程先提出这个问题，数据是事后才被读取的。

第四个难点是，答案不能只是一条策略。“更新之前先数一下行数”读过一次，会被遵守到第一个繁忙的日子为止，所以它必须内建到路径之中：一个在计数不符时拒绝提交的 runner，以及一个无法以其他方式写入的会话。

临时查询有一个更隐蔽的问题。**临时查询**是指由人编写、在任何应用之外只运行一次的查询。在只读会话中它无法改动任何东西，但它可以读取大量数据，运行起来可能给数据库带来很高的开销，而且它的结果会以截图、电子表格或聊天消息的形式离开系统。

默认：只读会话

人员会话始于一个属于 `human_readonly` 的登录，经由 access gateway 连接到 `ledgerly_replica_2`。它可以读取该角色所覆盖的表，其中 Restricted 列被隐去，Confidential 列按第 12 章的规定被脱敏。它不能写入，而且无论输入什么，都无法改变这一点。

大多数调查都能在这个会话内完成，人员无需申请任何东西，也无需等待任何人。仅从成本角度，我就会为这个默认做法辩护：常见任务无需审批的设计，才能让不常见任务的审批仍然有意义。

三个习惯可以保证共享副本及其结果的安全。**限定查询范围**：角色级的语句超时（第 5 章）和 gateway 的行数上限（第 7 章）能拦住失控的情形，而先查询计数和样本，可以让意外导出的数据量保持很小。**只选择能回答问题的最少内容**：`id, status, due_on` 优于 `SELECT *`，一部分是因为更快，一部分是因为除非问题需要，否则它能让 Confidential 列不进入结果。**把结果视为数据拷贝**：粘贴到工单或电子表格中的结果，是处于另一套访问规则之下的一份生产数据拷贝（第 13 章把这类拷贝视为旁门），所以应当粘贴标识符和计数，而不是带有个人内容的行，并写明产生这些结果的查询。

有一点告诫对下一节很重要。副本落后于主库一定的量，通常很小，有时并不小。在副本上取得的计数，只是对主库当前所持内容的一个估计。对调查而言这没有问题；但要判断一次写入会触及多少行，就不行了，所以下文的数据修正流程在主库上、在执行写入的事务之内取得权威计数。

写入是另一种行为

读取可以重复、放弃和忽略。写入则会改变其他每个 principal 所看到的内容，包括此刻正在运行的服务，这些服务会在任何人检查之前就依据新状态采取行动。一个被错误更新的 `status` 可能触发一封提醒邮件；一个被错误更新的 `due_on` 可能改变客户被告知的应付金额。因此本书让写入拥有自己的路径、自己的审批、自己的时间限制和自己的记录。

这条路径就是第 8 章的即时访问，只是带有一个在此处很关键的特性：写入申请绑定到一项经过评审的修正。申请携带该修正的 `fix_id` 和一个 `approved_hash`，其（权限、表、列）三元组由修正推导而来，而不是由申请人输入。任何申请最多只能请求 `human_jit_write` 所定义的上限。三个后果决定了修正应当如何编写。

第一，**申请由修正构建**。更新 `invoices.status` 的修正恰好需要一个三元组：对 `invoices` 的 `status` 列的 `UPDATE`。下一节中的 guard 先读后写，而这次读取使用的是 `human_readonly` 已经持有的、对 `invoices` 的普通 `SELECT`，所以不需要 `SELECT` 三元组。例外是对 Confidential 列做快照的修正，例如 `customers` 上的 `billing_address`：在 `human_readonly` 之下该列是被脱敏的，而脱敏值的快照对回滚毫无用处，所以申请会为它写明一个 `SELECT` 三元组，审批人也能看到它。

第二，**Restricted 列绝不出现在申请中**。`iban`、`account_holder_name` 和 `token_value` 不能被指名，对 `bank_accounts` 和 `payment_tokens` 的 `INSERT` 与 `DELETE` 也不可用，因为插入或删除一行会设置或销毁

Restricted 值。上限之内还有一种变更，它仍然不属于普通的数据修正：对 `bank_accounts` 或 `payment_tokens` 的 `customer_id` 做 `UPDATE`。该列并非 Restricted，但移动一行会改变这个付款账户或支付 token 属于谁。执行策略对这种所有者关联变更设置 gate，并将其转到下文所述的产品路径（第 9 章）。如果一项修正需要其中任何一种操作，它就不是数据修正。

第三，**审批人看到的是语句，而不只是三元组**。第 9 章阐述了这条绑定语句的审批规则，以及 gateway 如何强制执行。对于数据修正，审批界面会在三元组旁边显示语句、过滤条件和预期计数，而请求所有列，就是在请求另一种信任。范围写得窄的修正会更快获得批准，这正是应当鼓励的激励。

卡住发票这项修正的申请记录，采用第 3 章中与引擎无关的记法：

```
// request record for the stuck-invoice fix; values are illustrative
access_request:
    id              "req-0187"
    kind            "jit"
    requester       "robin"
    approver        "alex"
    fix_id          "FIX-0031"
    approved_hash   "a41c…07"   // set at approval, from the fix's
merge record
    reason          "reset invoices left in 'sending' by worker bug"
    expires_at      now() + 1 hour

access_request_privilege:       // derived from FIX-0031, not typed
    ("req-0187", "UPDATE", "invoices", "status")
```

在没有列级授予的引擎上，Restricted 列本来就放在上限之外的独立表中（第 3 章）。此时对主表的 JIT 申请就是表级授予，审批人批准了该表的每一个非 Restricted 列，其中包括 Confidential 列。授予不再限制语句的涉及范围，所以语句本身要承担更多的评审工作。

数据修正的剖析

数据修正是一个顺序固定的简短流程。顺序比任何单独的步骤都更重要，因为每个步骤存在的目的，都是让下一步更容易做对。

1. **陈述意图**。一句话：哪种状态是错的，哪种状态是对的，涉及哪些行。如果一句话写不完，这项修正多半应当拆成两项修正。

2. **用 `SELECT` 统计受影响的行数**。“四十张发票”和“9,000 张”是两种不同的对话，而计数是错误最先暴露的地方。
3. **写下预期行数**。修正声明它预期改动的数字。这就是 guard。
4. **取变更前快照**。修正将要改动的那些列的主键和旧值，保存在能比会话存活更久的地方。
5. **在事务内执行变更**。将实际改动的行数与预期行数比较。如果不同，就回滚。
6. **在同一事务内检查后置条件**。用一个查询来证明状态现在符合意图所说，例如“列表中没有任何发票的状态仍是 `sending`”。
7. **先提交，再记录**。提交是其他 principal 看到变更的时间点。在事务内读取的变更后状态，连同变更前快照一起写入记录。

伪代码把 guard 放在 runner 里，而不是依赖人的记忆。它将数据库、快照存储和审计日志视为彼此独立的系统，所以没有任何事务覆盖数据库之外的部分。runner 只会得到一个申请 id，别无其他；第一个函数决定该申请是否允许运行。

```
function loadApprovedFix(request_id, session):
    request = loadRequest(request_id)       // the gateway's record,
not the runner's input
    if request.status != "approved" or request.expires_at <= now():
        raise RequestNotLive
    if request.requester_login != session.login:    // granted to this
session's login only
        raise RequestNotForThisSession
    fix = loadFixScript(request.fix_id)
    // approved_hash was set at approval from the fix's merge record,
so
    // the script is compared with the record, not with itself
    if hashOf(fix.statements) != request.approved_hash:
        raise ScriptNotReviewed
    if request.privileges != fix.privileges:
        raise GrantsDoNotMatchFix
    return fix
```

第二个函数负责运行，它所做的一切都受第一个函数的约束：

```
function runDataFix(request_id, session):
    fix = loadApprovedFix(request_id, session)
    // the authoritative count, on the primary, before anything is
written
    before = readRows(fix.target_filter, fix.columns, using: session)
    if count(before) != fix.expected_rows:
        raise RowCountMismatch
    saveSnapshot(fix.id, before)
    transaction:
        // re-read under the lock: a changed value must fail, not pass
        locked = readRowsLocked(fix.target_filter, fix.columns, using:
session)
        if locked != before:
            rollback()
            raise RowsChangedSinceSnapshot
        // one source: the change runs on the same filter that was
counted
        changed = execute(fix.change, where: fix.target_filter, using:
session)
        if changed != fix.expected_rows or not
fix.postcondition(session):
            rollback()
            raise FixGuardFailed
        after = readRows(keysOf(before), fix.columns, using: session)
        // commit point: other sessions see the change only after this
line
        commit()

    try:
        saveAfterState(fix.id, after)
        auditLogAppend({ actor: session.login, action: "data-fix",
                         request: request_id, fix: fix.id, rows:
changed })
    on AuditLogUnavailable or SnapshotStoreUnavailable:
        queueForRetry(fix.id)
        notify("platform", "fix committed, record pending: " + fix.id)
```

这些代码块中有几处细节是刻意为之。

runner 从不自行提供哈希。它加载申请所指名的修正，把修正语句的哈希与申请记录中的 `approved_hash` 比较，而第 9 章会把这个哈希追溯到合并记录。重新输入语句，会让评审与执行之间的文本发生变化，而评审是附着在先前那份文本上的。

快照在事务之前保存。这样，快照存储出现故障时，修正会在未改动任何内容的情况下停止。代价是，在快照与写入之间，这些行可能发生变化：worker 可能在这段间隙里把一张发票从 `sending` 改为 `failed`，而其键仍然匹配过滤条件。

事务在加锁状态下重新读取这些行。它把整个结果（连同修正所要改动的列）与快照比较。行集合或某个值上的任何差异，都会以 `RowsChangedSinceSnapshot` 中止本次运行，作者需要从一次新的计数重新开始。在无法按过滤条件锁定行的引擎上，重新读取仍在事务内运行；它缩小了这段间隙，却没有将其关闭，剩下的就是预期计数和后置条件。

记录在提交之后写入。快照存储和审计日志属于其他系统，所以变更后状态要在提交完成后保存。这与第 8 章中的 JIT 授予顺序相反，那里授予不得在没有日志条目的情况下存在。

这里的顺序之所以站得住脚，是因为 Ledgerly 的 gateway 位于数据路径中（见下文）：它转发 runner 的语句并记录会话，所以即使审计追加失败，已提交的变更也有记录可查。该条目会被重试，并通知平台团队。在只代理凭据的 gateway 上，就不会有这些语句的会话日志，审计条目必须先写。

dry run 能显示计数无法显示的内容。计数是一条 `SELECT`，所以它看不到语句所引发的后续效应：在别处写入的触发器、级联、规则。真正的 dry run 会在事务内执行变更，然后将其回滚：

```
function dryRunDataFix(request_id, session):
    fix = loadApprovedFix(request_id, session)
    if count(readRows(fix.target_filter, fix.columns, using: session))
!= fix.expected_rows:
        raise RowCountMismatch      // a wrong filter never reaches
the locks
    transaction:
        countsBefore = countsIn(fix.watch_tables, using: session)
        changed = execute(fix.change, where: fix.target_filter, using:
session)
        report = { rows: changed,
                   sample: sampleAfterValues(fix, using: session),
                   related: diff(countsBefore,
countsIn(fix.watch_tables, using: session)) }
        rollback()          // nothing is kept; the row locks are held
until here
    return report
```

报告包含行数、变更后值的样本以及修正头部 `watch_tables` 中各表行数的变化。它需要一份已批准的申请和写入授予，所以在 PR 处于评审期间无法存在；它在已批准的时间窗口内运行，运行期间持有同样的行锁，并且看不到数据库之外的效应，例如邮件。在意外代价很高的场合，作者会在时间窗口的最初几分钟运行它，把报告附在申请上，由审批人在提交步骤之前阅读，代价是一次停顿。

只有当会话无法以其他方式写入时，guard 才算一种机制。Ledgerly 的 gateway 位于数据路径中（第 7 章），在 JIT 写入会话中只转发来自 runner 连接的写入，因此哈希检查、计数和快照都无法被跳过。事务仍归 runner 所有：gateway 把每条语句转发到事务中，并在整项修正范围内统计行数，但绝不代 runner 提交，这正是后置条件失败时会回滚每一条语句的原因。第 9 章讲述 gateway 如何依据合并记录检查每条语句，以及它的检查止步于何处。代价是，每一次 JIT 写入都依赖 runner 和 gateway 可用。如果 gateway 只代理凭据，runner 就只是一种约定，无法防范缺少 `WHERE` 子句的语句。

快照存放在哪里

变更前快照是一份数据拷贝，和其他拷贝一样，需要负责人和到期时间。`invoices` 上 `status` 值的快照属于 Internal 数据。对 Confidential 列（例如 `customers` 上的 `billing_address`）的快照属于 Confidential，其存储的访问规则至少要与源表一样严格。

我会把快照与修正的记录放在一起，并设定一个由平台团队确定的保留期限：长到足以回答“这项修正是否正确？”，短到不让快照变成客户数据的第二份、不受治理的拷贝。对发票状态而言是几天，对账单地址而言或许是一个完整的计费周期。选定一个数字，写下来，让它到期。

回滚计划

变更前快照，在有人写下如何使用它之前，不能算回滚计划。计划就是修正头部的一两句话，例如“从快照中为相同的键恢复 `status`”，而且应当可以机械地把快照转换成修正脚本。

回滚本身就是一项数据修正。它要经过同样的申请和评审，因为其间其他 principal 可能已经基于新状态采取了行动，恢复旧值可能覆盖合法的更新。因此，回滚的 runner 会检查当前值是否仍与原修正记录的变更后状态相符，并拒绝触碰此后已经发生变化的行。一项无法说出“这些行已经向前发展，请另行处理”的反向修正，会引出下一次修复。

有些变更没有回滚，头部应当明说。邮件、webhook 或对支付提供方的调用，无法通过恢复一列来收回。对这些情况，讨论发生在修正运行之前：这次状态变更会触发哪些效应，作出反应的服务是否应当先暂停？

有两张表在通常意义上没有回滚，因为没有任何人员角色可以编辑它们。`ledger_entries` 和 `audit_events` 是 append-only 的。对账本的更正是一条新条目，只能通过 `payments-service` 的更正路径签发，绝不能由

人员登录签发，所以一项看似需要改动这两张表的修正，应当去请求那条路径，而不是请求授予。一个拒绝这两张表上所有 `UPDATE` 和 `DELETE` 的触发器，让这条规则即使在授予被误扩大时也依然成立。它有代价：`ledgerly_owner` 仍然可以删除该触发器，所以真正的评审点仍然落在触及这些表的迁移上（第 11 章）。

DELETE 是同一流程，只是回滚更难

上面的剖析是针对 `UPDATE` 写的。假设 `invoice-worker` 中的一次 retry 向 `invoice_lines` 插入了重复的行，修复就是把它们删除。`invoice_lines` 上的 `DELETE` 处于 `human_jit_write` 上限之内，它在三个方面有所不同。

授予约束的是表而不是行，因为 `DELETE` 三元组没有列，所以约束来自过滤条件、预期计数和评审，而过滤条件应当是一个键列表。快照是整行，因为之后没有别的东西保存这些值，而带有 Confidential 列的表会使快照与该表一样敏感。回滚是重新插入，这是另一项数据修正，有它自己的申请。`INSERT` 三元组会写明列，所以重新插入的申请决定了 `id` 和 `created_at` 是否在其中。如果不在，恢复的行会为这两列获得新值，凡是引用旧标识符的内容都不会得到恢复。如果在，审批人会看到一个键和一个时间戳被写入，这是任何普通修正都不会做的，而引擎如何处理所提供的键，取决于它如何生成键。头部写明回滚采用哪种选择。

```
fix:
    id              "FIX-0044"
    intent          "Remove the duplicate invoice_lines rows created
by a retry.
                     No other row is deleted."
    target_filter   "id IN (<ids of the duplicates>)"
    change          "DELETE"
    expected_rows   12                          // illustrative
    snapshot        "whole rows"
    privileges      ("DELETE", "invoice_lines")  // no columns for
DELETE
    rollback        "re-insert from the snapshot as a new fix; its
INSERT triple
                     leaves out id and created_at, so the rows get new
values and
                     references to the old ids are lost"
    effects         "invoice totals are recomputed by the worker on
next run"
    watch_tables    ("invoices")
```

如果丢失标识符不可接受，评审人会要求比回滚更廉价的保护：一个外键，在仍有其他行指向该明细行时拒绝删除；或者一种不删除该行的不同修复方式。

评审：四眼原则，在代码仓库中

数据修正是只运行一次的代码，所以它采用工程师们早已信赖的控制手段：pull request。修正是仓库中的一个脚本，带有头部、评审人和历史记录，而六个月后，这段历史会回答是谁改了某个状态、为什么改。

```
// data-fixes/2026-10-06-FIX-0031-reset-sending-invoices.fix
fix:
    id              "FIX-0031"
    intent          "Invoices stuck in 'sending' after the worker bug
                     are set to 'sent'. No other column changes."
    target_filter   "status = 'sending' AND updated_at < '<cutoff>'"
    change          "SET status = 'sent'"   // applied to
target_filter only
    expected_rows   40
    postcondition   "no invoice matches target_filter"
    privileges      ("UPDATE", "invoices", "status")
    rollback        "restore status from snapshot for the same ids,
                     only where status is still 'sent'"
    effects         "reminders resume for these invoices"
    watch_tables    ("invoices")            // read by the dry run
    author          "robin"
    reviewer        "alex"
```

`privileges` 字段把仓库与第 8 章连接起来。申请的三元组是从中复制的，所以审批人把申请与修正并排查看，而 `loadApprovedFix` 会拒绝三元组与修正不一致的申请。触及其 privileges 未指名之列的语句会在运行时失败，这正是预期的结果；授予是位于评审之后的第二道防线。

评审人读什么

评审人的工作比“这是个好主意吗？”要窄。

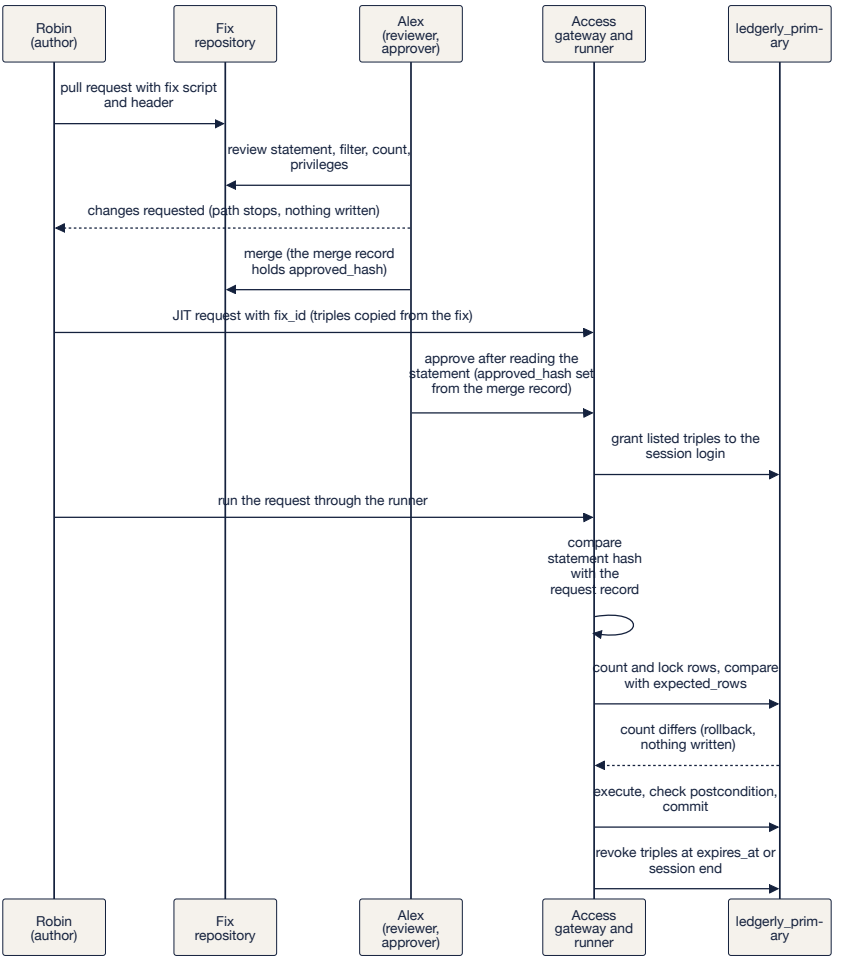
- **语句与意图相符吗**？二者在表、列和过滤条件上应当一致。
- **过滤条件够具体吗**？按键选择的过滤条件最安全；按状态加时间截点选择的过滤条件，在计数吻合时可以接受；完全没有限制的过滤条件，一眼即拒。
- **预期计数合理吗，它从何而来**？看到 `expected_rows: 40` 的评审人，可以要求在 pull request 中给出计数查询的输出。没有人能说出来源的数字，就是没有人核对过的数字。

- **privileges 只指名了语句所需的内容吗**？逐行比对。
- **修正触及 Restricted 列、append-only 表，或上限之外的表吗**？如果是，它就不是数据修正，评审以指向正确路径作结。
- **这次变更会触发什么**？提醒、邮件、读取该列的作业。这个问题能抓住那些悄无声息的情形。
- **预览与意图相符吗**？pull request 中包含计数查询的输出，以及行和变更后值的 `SELECT` 预览；dry run 报告稍后附到申请上，触发器或级联只会在那里显现，别处看不到。
- **回滚那句话可用吗**？对于 `DELETE`，它要说明重新插入无法恢复什么。

评审是本书意义上的四眼原则：在有风险的操作运行之前，由第二个人评审。合并固定了文本：评审之后对脚本的任何改动都会产生不同的哈希，runner 会拒绝它。JIT 申请的审批人与修正的评审人是两个行为。在大型团队中，我更倾向于由两个人担任，这样审批人是对照修正核对访问，而不是重新阅读修正；在小团队中，他们往往是同一个人，只要两者都不是作者，这就可以接受。作者绝不批准自己的修正或自己的授予。

流程时序

下图展示从申请到已提交变更的路径。请留意路径可以在未写入任何内容的情况下停止的几个位置：评审、哈希比较和 runner 的 guard。



当变更针对 Restricted 列时

上限禁止在任何 JIT 申请中出现 Restricted 列（原因见第 3 章），因此有一整类修复没有 SQL 路径。设想某个客户在 `bank_accounts` 上的 `account_holder_name` 录入时有笔误。没有人能在 JIT 会话中修复它，因为申请中无法指名该列。break-glass 会话也做不到：无论流程怎么说，`human_breakglass` 对 Restricted 列都不持有写入权限。

路径是应用。`admin-console` 请 `payments-service`（`bank_accounts` 的唯一写入方）做出这项变更；`admin-console` 要求在 `payments-service` 执行之前由第二个人批准，变更则记录在 `audit_events` 中。这是一条产品路径，而不是数据修正：它有表单、校验、原因字段、审批人和记录，并且以服务自己的角色运行，而不是人员的角色。

这比一条 SQL 语句的成本更高，我认为理应如此。这些值一旦暴露，会以难以挽回的方式伤害客户，而应用是校验和全部副作用唯一所在的地方。成本是需要这项变更的人要多花几分钟；节省的是手工输入的修复再也不会引入格式错误的银行账号。`payment_tokens` 中错误的 token 同样由 `payments-service` 通过提供方自己的流程来替换，而不是打补丁修补。

Break-glass 不是数据修正的途径

`human_breakglass` 在 JIT 上限够不到的地方写入，并且是唯一能读取 Restricted 列的人员角色（其确切权限见第 3 章）。纸面上看，它似乎是完成 JIT 路径拒绝的修复的便利方式。其实不是，原因在于这个角色缺少什么。

break-glass 是自行声明的。它没有审批人，没有 `fix_id`，也没有 runner，其保障措施是第 8 章定义的日志集和强制评审。这个角色的用途，是采取遏制持续危害的限制性动作，并在事后评审（事故一侧见第 15 章）。对数据的持久修复是另一种行为，而且始终是数据修正：在 break-glass 会话中运行，就跳过了在语句运行之前阅读它的那唯一一道控制，而且是在一个可能出现没有行保护的 `UPDATE` 的角色中。第 8 章把这一点表述为已接受的风险，而经由该角色路由修复会使其变成常态。

第二个原因是激励。如果 break-glass 能做修复，它就会变成快车道，其告警也就失去意义。一项“需要”break-glass 的数据修正，说明的是别的问题：夜间审批周期太慢，上限缺少该修正正当需要的某一列，或者该修正应当走应用路径。如果修复等不得，就用可用的最小动作止住危害，待压力过去之后，再通过常规路径运行数据修正。

把重复出现的修正变成产品功能

数据修正是一种症状：存在应用无法撤销的可达状态。第一次出现时是修正；第三次出现时，就是一个没人提交的功能需求。我会保留一个修正脚本目录，按修复类型打标签，同一个标签再次出现时，就有人问它是否应当属于 `admin-console`。“三次”并没有什么精确之处：一年两次、每次十行的修复，也许永远不足以支撑一个界面，而每周都要运行、每次都需要评审、支持人员一分钟就能完成的修复，就是一个等待被写出来的功能。

以开头那些卡住的发票为例，并假设这种状态会再次出现。第 3 章的授予表已经展示了 `app_admin_console` 可以对 `invoices` 做什么：只能对 `status` 做 `SELECT` 和 `UPDATE`。产品版本是发票页面上的一个操作，面向一个范围很窄的群体，为单张发票重置状态，要求填写原因，并以人员 principal 的身份将该操作写入 `audit_events`。它自带检查：只处理处于卡住状态的发票，一次只处理一张，并限制一个人每小时可重置的数量。

这一转换把风险从语句转移到了代码上：语句每次修复都要付出申请、评审和 runner 的代价，而代码只需评审、测试和版本化一次。支持人员无需数据库会话就能使用这个操作，所以每一个变成功能的修复，都是一项不再需要人员访问的周期性写入。

```
function resetStuckInvoice(actor, invoice_id, reason):
    if not actor.canResetInvoices:
        raise NotAuthorized
    if reason is empty:
        raise ReasonRequired
    if countResetsBy(actor, within: "1 hour") >= resetLimit:
        raise RateLimited

    transaction:
        invoice = readInvoiceLocked(invoice_id)
        if invoice.status != "sending" or invoice.updated_at >
cutoffForStuck():
            rollback()
            raise NotStuck
        setInvoiceStatus(invoice_id, "sent")
        // commit point: the status change and the business record
agree
        appendAuditEvent({ actor_principal: actor.id, action: "reset-
stuck-invoice",
                           subject_id: invoice_id, reason: reason })
        commit()
```

审计事件和状态变更都在 `ledgerly_primary` 中，所以一个事务就能覆盖二者；`audit_events` 是应用的业务记录，而不是第 14 章的审计日志。在上面的 runner 中，审计日志是一个独立的系统，安全性靠操作顺序来承载。

并非每项修复都应当变成功能。任何人都能按下的按钮，与两个人评审过的脚本，风险特征不同，所以涉及金钱、token 或银行详情的修复，即使频繁发生，也依然留在四眼原则的路径上。只有当某项修复的最坏情形受操作设计所约束时，它才会成为自助操作；否则它就成为产品内带审批的

申请。判断标准是：这项修复是否会再次出现，以及目前是否只有人才能做。如果两个答案都是肯定的，那么这个 sprint 就是一项投资，而不是成本。

针对可预见修复的 runbook

有些修正即使细节在变，类型却会重复出现：把发票重新分配给正确的客户，重放一次失败的导出，修复一条重复的行。对这些情形，合适的制品是 **runbook**：针对可重复任务的、成文且被遵循的流程。它与修正脚本的骨架相同，只是参数留空。它写明触发它的症状、确认症状的诊断查询、修正的模板及其所需的权限、预期计数的来源、需要检查的效应、回滚以及谁可以编写和评审。它还写明何时停下并升级：“如果诊断计数超过本段中的数字，这就不是已知问题”。

runbook 还可以携带预先审批，使可预见的修复在夜间不必等待审批人。由五个条件决定，并且由 runner 而不是人来检查：

- 过滤条件按键选择，即一个 id 列表，而不是按状态加截点选择；
- 预期行数至多为 N，这是 runbook 为自己的修复设定的数字（例如卡住的发票设为 20；仅作示意）；
- 三元组恰好是 runbook 的固定集合；
- 修复不触发任何出站效应，如 runbook 的 `effects` 字段所记录；
- 语句就是 runbook 模板，只替换了键列表，且模板的哈希在 runbook 评审时已获批准。

runbook 的合并记录存储该模板哈希。键列表是申请携带的唯一部分，runner 把它代入存储的模板，并对照申请的三元组检查结果；它从不接受来自其调用方的哈希或语句（第 9 章讲述 gateway 如何检查代入后的语句）。如果任何条件不满足，该申请就回退到由审批人完整评审。

代价是，预先批准的运行在执行时，那天夜里没有人读过键列表，所以 runbook 的行数上限约束着一份错误的列表，并且由 runner 之外的人在下一个工作日阅读记录。runbook 也会老化：对一张此后改变了结构的表而言，它会让一条过期的语句显得好像评审过。要把每份 runbook 与一个日期和一位负责人绑定，并把一次失败的运行视为修订它的提示，而不是需要绕过的麻烦。

没有 WHERE 的 UPDATE，无责分析

设想处于初始状态的 Ledgerly。一名工程师使用桌面客户端连接，客户端记住了共享管理员密码，并且自动提交。任务是处理卡住的发票。语句在编辑器里写成了两行：`UPDATE` 在第一行，`WHERE` 在第二行。客户端执行被选中的部分，而选区只覆盖第一行。语句运行了，`invoices` 中的每一行，比方说 9,000 行，状态都变成了 `sent`。

这个故事的一种讲法，是以有人被告知要更小心收场。这什么也教不了，因为坐在同一把椅子上的下一位工程师，面对的仍是一个问题与一次不可逆变更之间的同样距离。有用的问题是：设计中的什么让一条语句得以触及每一行。

- **这个登录可以在所有地方写入**。共享的管理员角色拥有每一张表，所以错误没有被限制在任务所需的表或列上。
- **客户端自行提交**。在语句成为最终结果之前，没有人看到“影响了 9,000 行”。
- **从未询问过计数**。没有任何东西要求这个人知道应当预期多少行，所以事后打印出来的数字毫无意义。
- **没有别人读过这条语句**。唯一的读者是作者，而作者是把意图读进了文本里。
- **没有变更前快照**。旧的状态没了，对修复的修复没有起点。
- **日志写的是“共享角色”**。“这是谁运行的？”这个问题，只能通过聊天记录得到答案。

恢复从备份开始，核对此后发生的一切，并通知所有下游（第 13 章和第 15 章讨论这些机制）。设计在语句被输入之前，就已经定下了它的成本。

现在把同一个下午放到目标状态中重演。Robin 写下带头部的修正，预期计数为 40，Alex 对其评审。JIT 申请指名对 `invoices` 的 `status` 列做 `UPDATE`，授予持续一小时。如果评审过的脚本中丢掉了过滤条件，runner 会数出 9,000 而预期是 40，并在任何语句运行之前拒绝。输入到个人自己客户端里的语句，不会被当作写入转发（第 9 章的 gateway 规则），而在普通的 `human_readonly` 会话中，该 `UPDATE` 根本无法运行。

guard 并不能让修正变得正确。它只是让一类错误变得响亮：语句触及的行数与作者所说的不同。一条恰好触及 40 行、但是错的那 40 行的语句会通过，这就是为什么评审、后置条件和快照作为独立的层次存在，也

是第 9 章要说明分析看不到什么的原因。guard 的好坏也取决于它的数字。从运行自身输出中复制来的预期计数，会教 runner 与自己保持一致，所以这个数字必须来自评审人可以独立运行的查询，最好是在语句编写之前就得到。在引擎具备列级授予的情况下（第 3 章），范围窄的授予限制了 guard 所漏掉的那些情形能够触及的范围。

权衡与成本

这个流程比直接输入一条语句慢。申请、评审、审批和 runner，在顺利的日子里会增加几分钟，在审批人睡着时则会多得多。在周四下午，这可以接受。在夜里、告警之下，这个成本会诱使人们走向 break-glass，它值得设计上的关注：值班轮换中有一名评审人，上文针对符合条件的修复设立的 runbook 预先审批，以及在第一个需要的夜晚到来之前就写下的、团队愿意为夜间审批等待的最长时间限制。

评审可能变成条件反射。几秒钟内给出的批准只会增加摩擦，却不会减少暴露；第 8 章给出了完整的论证。本章特有的信号是 pull request：如果评审人从不索要计数查询的输出，也从不要求修改语句，那么他们要么什么都没读，要么只评审过完美的修正。防御手段是那些需要制品的检查项：pull request 中的计数输出和预览，申请上的 dry run 报告，以及把语句与 privileges 做明确比对。

事务会持有锁。runner 在评审之后启动事务，并在数秒内结束它。如果流程先打开事务，“以求稳妥”，再让人去决定，就会让其他写入方等待，占满 pooler 的连接，并让第 5 章的超时在不相关的工作上触发。要让人的决定保持在事务之外。

大型修正需要分批。触及数百万行的修正，不能作为一个事务运行，否则会长时间持锁并产生很大的事务日志。通常的做法是把它拆成批次，比如每批几千行（这是示意的规模，不是建议），每批都有自己的计数和 guard，并由 runner 的批处理模式重复同样的检查。中途失败会让数据处于混合状态，所以修正必须写成可以重新运行的：过滤条件只选择仍处于旧状态的行。大型修复仍然是数据修正；只有为新结构做首次填充，才属于迁移（第 11 章）。

快照和修正仓库都需要负责人。没有到期时间的快照，是以一种风险换了另一种风险，而对于 Confidential 列，后一种风险可能更大。修正脚本目录是生产环境如何漂移的记录，而如果没有按标签建立的索引，就没有人会去查找某个问题上一次出现的情况。

何时不必全部采用。在数据库不含任何敏感内容的小团队中，为每次状态变更都开一个 pull request，仪式感超出了风险所需；先计数并使用事务，仍然物有所值。在没有生产数据的开发或预发布数据库上，这一切都不适用；而对每晚重建的拷贝，例如 `ledgerly_analytics`，一次错误的写入只需付出一次重建的代价。这个流程随出错的代价而伸缩，而主库正是这种代价最高的地方。

常见错误

- **把 `WHERE` 子句当作 guard。**这很诱人，因为作者的意图就寄存在这个子句里。但子句是文本，文本可能被截断、输错或只运行一部分。补救办法是一个将变更行数与声明的预期比较、不符就回滚的 guard。
- **在副本上计数，在主库上更新。**副本有延迟，所以两个数字可能因正当原因而不同。补救办法是在主库上、在事务内取得权威计数，只把副本上的计数用于探索。
- **重新输入评审过的语句。**评审人批准的是一个文件，而有人把它复制进编辑器，并“顺手”改了一处。补救办法是一个 runner，将脚本的哈希与申请记录中的 `approved_hash` 比较，同时配合一个只转发来自该 runner 的会话写入的 gateway。在只代理凭据的 gateway 上，这一 guard 只是约定，评审必须承担它。
- **“以求稳妥”而申请宽泛的授予。**这使批准成为对信任的判断，而不是对照语句的核对，并且扩大了一个错误所能触及的范围。补救办法是申请语句所需的三元组，情况有变时再提第二份申请。
- **因为审批人太慢而使用 break-glass。**路径更短，修正又很紧迫。补救办法是解决夜间审批过慢的问题，把 break-glass 留给它被定义用来应对的紧急情形。
- **通过“特殊”会话写入 Restricted 列。**直接更新 `iban` 看起来像是对客户的善意，却一步移除了校验、审批和应用的记录。补救办法是走应用路径，即使它更慢。
- **保存一份无人治理的快照。**快照用来防范一次错误的修正，却变成了共享文件夹里一份不受治理的 Confidential 数据拷贝。补救办法是设定负责人、与源表一致的访问规则和到期日期。
- **因为修正很小就省掉回滚那句话。**小修正恰恰是会被匆忙回退的那种。补救办法是在运行之前写下一行计划，说明快照恢复什么，又不能恢复到哪里。

- **让重复出现的修正永远停留在脚本阶段。**结果是一项周期性写入，每次都需要有人在主库上操作。补救办法是定期查看修正目录中的标签，并作出决定，即使决定是“暂不处理”。
- **评审意图是否正确，而不是语句是否正确。**争论这项修复是否值得做的评审人，可能根本没有读过语句。补救办法是一份按顺序提出狭窄问题的检查清单。

要点

- 只读会话是调查的正确默认，而且只有在常见任务无需审批时才行得通。
- 写入是不同于读取的行为，所以它要经过自己的申请、评审、时间限制和记录。申请携带 `fix_id` 和取自合并记录的 `approved_hash`，其三元组由修正推导而来，审批人看到的是语句。
- 数据修正的安全顺序是：计数、快照、事务、行数 guard、后置条件、提交；并且应当由 runner 而不是人的记忆来强制执行，同时由 gateway 只转发来自 runner 的会话写入。
- guard 让一类错误变得响亮，却不会让修正变得正确，这就是评审、窄授予和变更前快照作为独立层次存在的原因。
- 对 Restricted 列的变更不是数据修正：它经由 `admin-console` 到 `payments-service`，带有第二位审批人和 `audit_events` 中的一条记录，而 break-glass 不是绕过这一点的途径。
- 重复出现的手工修正是缺失的产品功能，每一个变成 `admin-console` 操作的修正，都是一项不再需要人在主库上操作的周期性写入。
- 当一条语句触及每一行时，有用的问题是设计中的什么允许了这一点，而答案是一份缺失的 guard 清单，而不是某个人。

数据修正改变的是表中的内容；下一类变更改变的是表本身。第 11 章转向模式变更，讨论由谁执行、使用何种身份以及使其向后兼容的习惯，从而让某一天的修复不会成为下一天的故障。

第 11 章 — 模式变更与迁移访问

设想 Ledgerly 的某个周四下午。开票团队希望在 `invoices` 上新增一列，以便把税额与总额分开显示，这项变更很小：一列、一个默认值，再为旧行做一次 backfill。本周值班的 Robin 评审了 pull request 并批准。流水线以 `ledgerly_migrator` 的身份运行迁移，部署完成，随后约九十秒内，所有触及发票的 `billing-api` 请求都挂起，然后失败。

迁移本身没有任何问题，也没有人做过任何异常的事。一个长时间的 `invoice-worker` 批处理事务在等待外部邮件服务时一直未关闭，持有这张表上的一把锁，迁移排在它后面，而迁移之后到达的每条语句又排在迁移后面。平台团队的 Alex 很快找到了原因。随后是更难的问题：这次迁移带着每张表所有者的权力运行，无人值守，而横亘在这份权力与账本之间的，只有对一份 diff 的评审。模式变更应当如何到达主库，使用哪种身份，又如何避免它既成为访问模型中的漏洞，又成为一次中断？

本章为目标状态增加一个步骤：迁移由流水线以 `ledgerly_migrator` 的身份运行，除此之外没有任何主体有权更改模式。

在第 1 章的五部分模型中，迁移是边界最宽的**执行**。流水线按设计持有所有者的权限，所以风险不在于谁持有凭据，而在于迁移被授权做什么，它的语句如何执行，以及留下什么记录。对迁移而言，这意味着：每条通道上允许的语句类别构成一份简短清单（授权），一条决定语句何时、以何种方式运行的路径，带有较短的锁等待和重试（执行），以及一份所有者无法改写的记录（责任归属）。

问题及其难点

模式变更是大多数团队在寻常日子里对数据库所做的权限最高的事。所有权让所有者可以删除表、更改列类型、改写授予，并且在许多引擎上绕过适用于其他所有人的限制。第 3 章把这份权力交给了没有登录能力的角色 `ledgerly_owner`，并让 `ledgerly_migrator` 只在迁移运行期间承接它。这样的安排回答了谁拥有模式，却没有回答谁可以启动迁移，运行中的迁移被允许做什么，以及迁移运行期间其他所有人会怎样。

同一项活动里藏着三种彼此独立的风险。

第一种是**权限**。能够触发流水线迁移步骤的人，实际上可以在生产环境上运行所有者级别的语句。常驻的是这份权限，其含义与第 3 章所述一致：一份常驻的所有者权限，通过短期凭据来行使。对它的约束属于流程层面：凭据只存在于流水线中，随作业一同过期，而流水线是使用它的唯一获准途径。这些约束的可靠程度，只取决于流水线自身的访问控制，这使得流水线成为数据库安全边界的一部分，无论是否有人把它画进去。

第二种是**可用性**。许多模式变更需要锁，而损害就发生在锁上。必须等待锁的语句不只是在等待：在许多引擎上，排在它后面到达的语句也会一同排队，所以一个第一秒钟拿不到锁的迁移，可能让这张表对所有人关闭，直到排在它前面的最老事务结束为止。变更的大小几乎无关紧要。只要挡路的是错误的事务，一列的变更也可能像整表重写一样造成破坏。

第三种是**漂移**。一个既由迁移定义、又被人手工编辑的模式，是没有人能完整描述的模式。某个糟糕的夜晚从控制台加上的应急索引很有用，一年之后，它就成了环境之间没人记得的差异，还附带一些对不上任何文件的授予。

每一种风险都可以通过设计来降低。受流水线约束的迁移角色、快速失败的锁等待、不需要长时间持锁的变更形态以及把线上模式与预期模式做对比的检查，这些都是可以构建出来的东西。“迁移时要小心”这样的策略，不在其列。

谁来运行 DDL，以哪个角色

DDL（数据定义语言）是创建、修改和删除模式对象的一类语句，与读写行的语句相对。本章的规则很简短：DDL 以 `ledgerly_migrator` 的身份、从流水线运行，任何应用角色都不能运行它。

应用角色不持有任何 DDL

第 3 章中的 `app_*` 角色，没有一个拥有表，也没有一个可以创建、修改或删除表。这句话容易说，坚持起来却要付出一点代价，因为许多框架可以在启动时创建或更改表，一些团队会启用这一点，因为它省去了部署中的一个步骤。这种便利的代价是服务随之需要所有权，而被攻破或出了差错的服务可以删除并重新授予。我会在生产环境中关闭自动模式管理，只在开发数据库中保留它，因为在那里无人需要解释一张被删掉的表。

可见的副作用是，包含模式变更的部署现在分为两步：先迁移，再发布服务。二者之间的先后顺序是一个设计问题，下文的扩展与收缩模式给出了回答。

迁移者与所有者

`ledgerly_migrator` 有一个供流水线使用的登录；它是 `ledgerly_owner` 的成员，自身不持有任何数据权限。成员关系让迁移者获得所有者的权限，却不会使迁移者成为所有者。在许多引擎上，新对象归创建时处于活动状态的角色所有，所以在迁移者自己的登录下运行的迁移，会让它创建的表归迁移者所有。因此规则在每种引擎上都一样：迁移始终显式切换到 `ledgerly_owner`，因为所有权跟随的是活动角色，而不是继承来的权限。它创建的每个对象于是都归 `ledgerly_owner` 所有，迁移结束时再回到迁移者自己的角色。用不属于任何引擎的记法表示：

```
function runMigration(migration):
    login = fetchMigratorLogin(identity: pipeline.workloadIdentity)
    connection = connect(login: login, database: "ledgerly_primary")
    // lock_wait 3s, statement timeout none (Chapter 5)
    applyRoleSettings(connection, role: "ledgerly_migrator")
    if isApplied(connection, migration.id):
        close(connection)
        return                          // a retry never re-runs
applied work
    // external record first: the owner can rewrite the database's own
record
    auditLogAppend({ actor: login.name, action: "migration-start",
                     migration: migration.id, commit: migration.commit
})

    try:
        assumeRole(connection, "ledgerly_owner")
        execute(connection, migration.statements)
        recordApplied(connection, migration.id)
    on LockWaitExceeded or DatabaseError as error:
        // owner rights end with the connection on every path
        resetRole(connection)
        close(connection)
        auditLogAppend({ actor: login.name, action: "migration-
failed",
                         migration: migration.id, error: error.kind })
        raise error
    resetRole(connection)
    close(connection)
    auditLogAppend({ actor: login.name, action: "migration-applied",
                     migration: migration.id })
```

有四个细节值得注意。凭据由流水线的 workload identity 在运行时签发给这个作业（沿用第 4 章的模式），并随作业一同过期，所以被复制的流水线变量无法在数月里持有一份可用的所有者凭据，仓库中也不存放任何这类凭据。切换到所有者角色的范围，限定在一次迁移的语句之内。迁移已应用的记录在同一次运行中写入，所以数据库知道自己的版本，下文的漂移检查依赖这一点。而外部条目在第一条所有者语句之前写入，因为 `recordApplied` 所在的数据库是所有者可以改写的。代价是，审计存储一旦不可用，迁移在开始之前就会停止。这与第 10 章的数据修正顺序相反，那里 gateway 的会话日志已经记录了语句；而在这里，除了流水线之外没有任何东西能看到所有者会话做了什么，所以记录要先写。每个开始条目都有对应的结束条目，即 `migration-failed` 或 `migration-applied`，而 `isApplied` 使重试即便在最后一次审计追加是失败的那一步时也依然安全。

在一些引擎上，记录版本与运行 DDL 可以共用一个事务，而在另一些引擎上，一条 DDL 语句会结束它所在的事务。需要清楚自己的引擎属于哪一种。如果一次失败可能让模式已变而版本未被记录，迁移就需要写成第二次运行也无害的形式。

人员不持有迁移者

人员从不以 `ledgerly_migrator` 的身份连接。如果有人需要一项模式变更，就由这个人编写迁移，再由流水线运行它。这听起来是摩擦，事实也确实如此，但它换来一个有用的性质：所有者权力的每一次使用，都对应一次流水线运行，而这次运行有提交、有评审、有日志。当迁移者凭据同时也存放在团队的密码管理器中，这一性质就消失了，设立迁移者的理由也随之消失。

流水线能触达什么

把迁移者凭据交给流水线，是把一项巨大的权力转移到一个许多人都能施加影响的系统中。有必要把这一点说清楚，因为这个设计可能正是在这里悄悄出错。

流水线能做什么

运行 `runMigration` 的作业，可以创建、修改或删除模式中的任何对象，更改所有者所拥有对象上的任何授予，也可以更改两张 append-only 表 `ledger_entries` 和 `audit_events`。任何应用角色和人员角色都不能编

辑这两张表中的行，包括 `human_breakglass`；唯一的结构性例外是，所有者以及运行中的迁移可以修改它们。对这一例外的控制并非权限，而是评审。

于是关于流水线的问题就是：谁能让它运行某些东西？这包括向部署分支合并的人，能编辑流水线定义的人，能向持有凭据的作业添加步骤的人，以及作业运行期间能读取其环境的任何人。

约束触达范围

我会用五种方式约束这个范围，大致按每一种去除的风险多少排序。

1. **只有一个作业持有凭据**。生产环境的迁移作业，是迁移者登录唯一可以被签发的地方，并且由作业的身份签发，而不是由每个作业都能读取的变量提供。未经评审的分支的构建、测试作业和 pull request 流水线从不会收到它。
2. **流水线定义与迁移一样接受评审**。对迁移作业、其权限或触发它的分支规则的更改，就是对所有者触达范围的更改，需要由本该检查迁移的同一位评审人来把关。
3. **触发条件是受保护分支**。向它合并需要一位非作者的人评审，这是 Ledgerly 针对模式变更的四眼原则。对于触及授予、所有权或 append-only 表的迁移，评审人必须来自平台团队。
4. **凭据是短期的**。一个失败后留下含有机密的日志的作业，留下的是一份已经过期的机密。
5. **每次运行都有记录**。流水线把它应用了什么、以哪个登录、来自哪个提交，写入一个流水线无法改写的存储，并且在迁移运行之前写下开始条目。第 14 章说明该存储位于何处。

评审本身可以部分机械化。一个小检查可以先于人读取迁移，并标出需要更仔细查看的语句：

```
function classifyMigration(migration, lane):
    flags = []
    refusals = []
    for each statement in migration.statements:
        if statement.touches("ledger_entries") or
statement.touches("audit_events"):
            add(flags, "append-only table: platform review required")
        if statement.kind in ["GRANT", "REVOKE", "CHANGE OWNER",
                              "CHANGE DEFAULT PRIVILEGES", "CREATE
VIEW"]:
            add(flags, "changes access: platform review required")
        if statement.kind in ["CREATE TABLE", "ADD COLUMN"] and
           (statement.table is new or statement.table in
sensitiveTables) and
           not hasExplicitGrants(migration, statement.table):
            add(flags, "new structure without a grant decision")
        if statement.kind in ["INSERT", "UPDATE", "DELETE"]:
            add(flags, "data change in a migration: backfill rule
applies")
        if statement.kind in lockHeavyKinds:
            add(flags, "long lock: scheduled window and a named
watcher")
        if statement.kind in ["DROP TABLE", "DROP COLUMN"] and not
migration.isContractStep:
            add(flags, "destructive change outside a contract step")
        if lane == "expedited" and statement.kind not in
expeditedAllowlist:
            add(refusals, "not allowed in the expedited lane: use the
normal lane")
    return flags, refusals
```

在常规通道中，这个检查不做任何决定。它让评审人的注意力落在改变访问、持有锁或销毁数据的那几行上，并让授予的遗漏变得可见。一个遇到任何标记都阻止合并、又无法确认某个标记的检查，一个月内就会被绕过，所以标记需要一位具名的审批人，而不是禁止该变更。加急通道是这个检查唯一会拒绝的地方：非空的 `refusals` 列表会让作业在运行之前失败，任何审批人都无法推翻。下一节解释其中的原因。

常规通道不会彻底拒绝任何东西，这是一种选择：一份硬性的禁止语句清单，会成为第 9 章拒绝、gate 与允许矩阵之外的第二套策略。删除 `ledger_entries` 会被标记两次，一次作为 append-only 变更，一次作为破坏性变更，平台团队必须对它说“可以”。

为什么不是持工单的人

一些团队更愿意在审批之后由 DBA 手工应用每一次迁移。对于每月只有一两次模式变更的小型数据库，这可以是合理的选择，流水线触达范围的问题也随之消失。代价会在这个人不在时、变更必须在多个环境中完全一

致地应用时，或手工步骤与评审过的内容出现偏离时显现。流水线的弱点是触达范围宽；手工路径的弱点是没有任何东西保证被应用的内容等同于被批准的内容。我倾向于采用带有上述约束的流水线，并且如果流水线自身的访问清单被证明无法保持简短，我会重新审视这个选择。

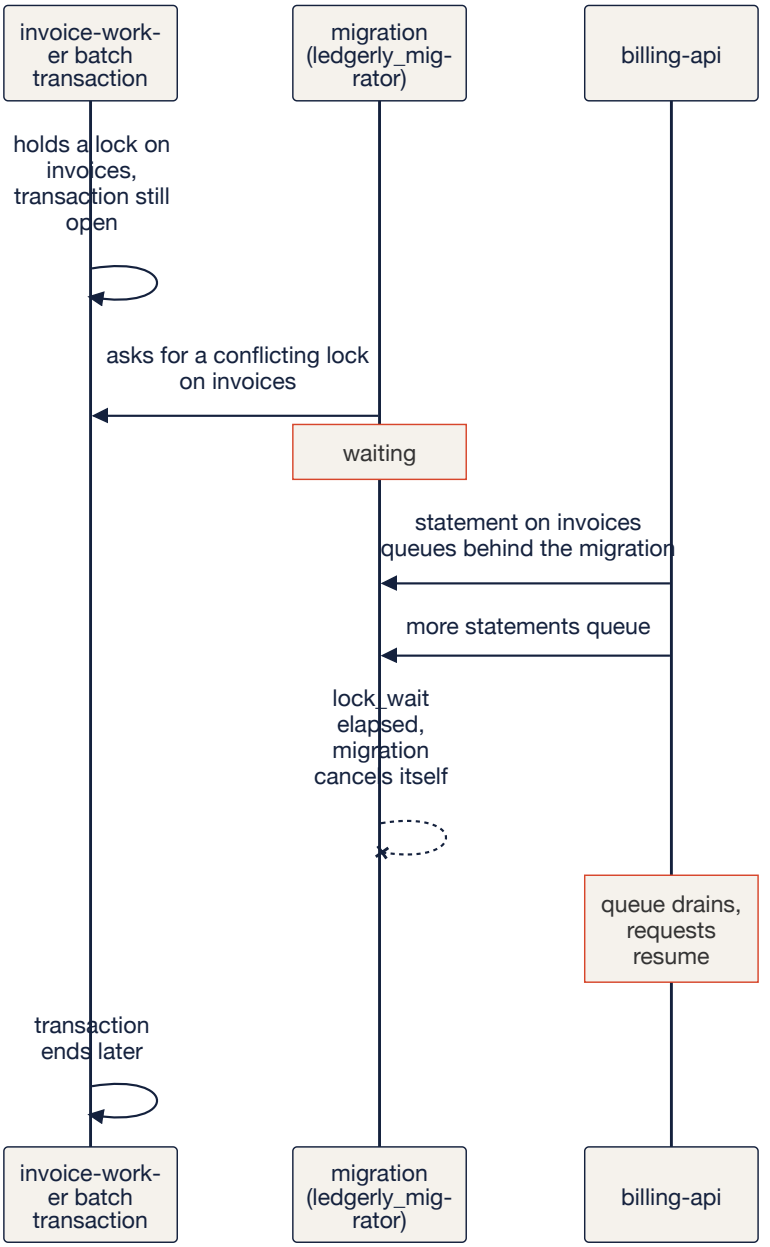
锁与可用性风险

现在回到 Ledgerly 的那九十秒。其背后的机制在各种引擎上是共通的，防御则由少数几个习惯构成。

等待中的迁移如何让一张表关闭

修改 `invoices` 的迁移需要一把与普通读写冲突的锁。只有当不再有持有冲突锁的事务处于打开状态时，它才能拿到这把锁。如果有一个长事务处于打开状态，迁移就要等待。在它等待期间，这张表上的新语句会排在迁移待决的请求之后，因为允许它们插队会让迁移饿死。于是这张表实际上处于关闭状态，直到旧事务结束或迁移放弃为止。

下面的时序图展示事件的顺序。请留意锁等待在队列增长之前给迁移留出退路的那个时间点。



关键在于自我取消。没有锁等待时，迁移会一直等到那个批处理事务结束，而 `billing-api` 在整个期间都在排队。有了较短的锁等待，迁移在几秒后失败，队列排空，流水线可以稍后再试。

较短的锁等待与耐心的重试

第 5 章为 `ledgerly_migrator` 设置了 `lock_wait 3s` 且不设语句超时。这一搭配是刻意的。模式变更可能需要很长时间才能完成工作，但它应当只等待很短的时间来开始。重试因此属于流水线：

```
function applyWithRetry(migration, maxAttempts, pauseSeconds):
    attempt = 1
    while attempt <= maxAttempts:
        try:
            runMigration(migration)
            return
        on LockWaitExceeded:
            // a clean no-op only if nothing had committed: one
transactional
            // unit, or a locking step that comes first
            if not migration.retrySafe:
                notify("platform", "partial migration, not retried: "
+ migration.id)
                raise MigrationBlocked
            notify("platform", "migration waiting on a lock: " +
migration.id)
            sleep(pauseSeconds * attempt)
            attempt = attempt + 1
    // the table is busy for longer than the retry window; a person
decides
    notify("platform", "migration blocked after retries: " +
migration.id)
    raise MigrationBlocked
```

对此有几点说明。锁等待失败只有在尚无任何提交时才是干净的空操作，这就是迁移带有 `retrySafe` 标记的原因：它是一个事务单元，或者它的加锁步骤排在最前。在每条 DDL 语句各自提交的引擎上，多步骤迁移不会被盲目重试，因为第二次尝试会遇到第一次尝试的成果。它会停下来，由人来决定。暂停时间随尝试次数增长，使得反复尝试本身不会变成一种负载模式。而最终状态 `MigrationBlocked` 是一个交给人的问题：查明是哪个事务挡了路，再决定是等待、取消它，还是重新安排时间。这个决定涉及终止某个人的会话，而这本身是一个访问问题，若发生在事故期间，则属于第 15 章。

长语句与在线变更

有些操作在运行期间一直持有锁，比如为更改列类型而重写一张大表。对这些操作，问题不在锁等待，而在持续时间。哪些语句快、哪些慢，因引擎而异，而且答案随引擎版本变化，所以需要逐条核实。在线模式变更工具和非阻塞的索引构建可以避免长时间持锁，但它们要创建和删除对象，

因此需要与任何迁移相同的所有者触达范围，并以迁移者的身份在流水线内运行。`classifyMigration` 会把这些语句标记为长锁，而长锁迁移会获得一个预定的时间窗口和一位在场盯守的具名人员，二者都写在迁移的描述中。这不是自动化的失败；有些变更本身就是事件。

扩展与收缩

最安全的模式变更，是新旧应用代码都能与之共存的变更。其模式称为**扩展与收缩**：以增量方式的步骤更改模式，把应用迁移过去，然后才移除不再使用的内容。它对本书很重要，因为每一步都是一次带有小锁的小迁移，也因为它让部署保持可逆。

设想 Ledgerly 的一项假设变更，仅用来展示其形态：把 `invoices` 上的 `total_minor` 重命名为 `amount_minor`。规范模式保留 `total_minor`，而这个例子假设存在一个重命名了该列的变体。步骤如下：

1. **扩展**。一次迁移以短暂的锁添加可为空的 `amount_minor` 列。旧代码会忽略它。
2. **双写**。`billing-api` 和 `invoice-worker` 的新版本在写旧列的同时也写新列，因此新旧版本可以并行运行。
3. **Backfill**。旧行获得新列的值，做法见下文。
4. **读取新列**。如果该版本表现异常，旧列依然是正确的，所以回滚是一次部署，而不是一次修复。
5. **收缩**。在没有任何东西读写 `total_minor` 的一段时间之后，最后一次迁移将其删除。这是唯一的破坏性步骤，并且单独设 gate：按破坏性变更评审，且绝不放在引入新列的那个版本中。

扩展与收缩需要时间和精力。本来可以是一次迁移的变更，变成了四五次，分散在若干次部署中，而在中间的步骤里，应用要同时携带两种形态的代码。对于只有一个服务且有维护窗口的小团队，单次迁移也许是更好的选择。对于 Ledgerly 这样有五个服务读取同样的表、且业务不接受停机窗口的情形，这一模式正是让锁保持短暂、让回滚保持廉价的东西。

迁移之内的 backfill，与迁移之外的数据修正

第 3 步是对数据的批量变更，同样的规则也适用于把总额拆分为净额和税额：区别在于，旧表中没有税额数字，所以那次 backfill 无法计算，只能让新列保持为空，直到数据修正把它们填上。第 10 章把数据修正定义为人员在正常应用路径之外对生产数据所做的变更，用来修复应用无法

修复的状态，因此本书需要在这里划一条线：属于模式变更的 backfill，在迁移内部以 `ledgerly_migrator` 的身份运行；其他任何对数据的批量变更都是数据修正，遵循第 10 章。

什么使一次 backfill 属于模式变更？

- 它唯一的目的，是依据迁移中写明的规则，为新结构提供第一批值。
- 它在迁移中得到描述，随迁移一同评审，并由流水线应用。
- 它写入的每个值都可以由表中已有的数据重新算出。

只要其中任何一条不成立，它就是数据修正。这次重命名展示了界线落在何处：写入 `amount_minor` 的每个值，都是同一行的 `total_minor`，这条规则由迁移写明，评审人可以核对。假设团队在 backfill 时发现一些旧发票的 `total_minor` 是错的。纠正这些总额不属于模式变更的一部分。它遵循第 10 章，要有受影响行数的计数、对预期行数的 guard 和第二位评审人，因为复制一个错误的值，仍然是一次正确的 backfill。迁移者的权力很大，它不应当成为每一项棘手变更的通道。

在大表上，backfill 分批运行，每批是对一组有界的行所做的短事务。当副本延迟增长时（第 6 章）它会暂停，延迟迟迟不消退时明确放弃，并且可以停止后重新开始而不丢失进度。

同样的规则适用于两张 append-only 表。迁移可以为 `ledger_entries` 添加一列，但不会改写已有条目、用更正后的值去填充这一列；更正是一条新条目，经由 `payments-service` 的更正路径签发。确实触及这两张表的迁移，会被上面的分类步骤标记，并交给平台团队评审。

应急 DDL 与漂移

每个团队迟早会面对这样的夜晚：加一个索引就能修复一个慢查询，而流水线需要二十分钟之类的时间。问题在于，访问模型对这条捷径是怎么说的。

没有控制台捷径，所以要建一条快速通道

在目标状态中，没有人持有所有者的登录。`human_breakglass` 可以写入 `INSERT` 和 `UPDATE`，不携带 DDL，而超级用户是一件封存的凭证，不用于日常工作（第 3 章）。所以主库上的人员没有任何正当的方式手工添加索引。这是刻意为之，它把问题转移到了流水线：当需求紧急时，它能以多快的速度运行一项经过评审的变更？

我会给流水线一条**加急通道**，而不是一个旁路。这条通道使用同一个作业、同一份凭据和同一份记录；它的差别在于时间，而不在于谁必须批准。四眼原则保持不变：非作者的评审人仍须批准，作者不能独自合并。放宽的是预定的等待期，以及对于一份简短的语句类别白名单，平台团队的要求，改为平台团队在下一个工作日跟进评审。`classifyMigration` 强制执行这份白名单：构建索引、删除索引和添加可为空的列符合条件。更改授予、所有权或视图、触及 append-only 表或销毁数据的语句则不符合，因为平台评审正是为这些变更而存在的，包含其中任何一种的迁移，会在这条通道中被拒绝，转由常规通道处理。第 15 章描述事故期间的访问；这里的要点是，紧急改变的是流程的速度，而不是运行 DDL 的身份，也不是危险语句所要经历的评审。

代价是，加急通道是一个长期存在的、可以跳过部分谨慎的选项，而这样的选项是会被用到的。要记录它被使用的频率，并把不断上升的数字视为常规通道太慢的信号，而不是团队变得不那么谨慎的信号。

漂移

漂移是迁移所描述的模式与生产环境中实际存在的模式之间的任何差异。它通过手工应用的应急修补进入，通过应用之后又被编辑的迁移进入，通过某个没人记得的工具所做的所有者级别变更进入，也通过在访问定义之外被改动的授予进入。模式中的漂移会让下一次迁移措手不及。授予中的漂移则是一次没有任何评审看到的访问变更。

一项定期运行的检查可以发现这两种漂移：

```
function detectDrift(environment):
    expected = buildExpectedSchema(appliedMigrations(environment))
    actual = readLiveSchema(environment)

    findings = []
    for each difference in compare(expected, actual):
        add(findings, difference)
    for each difference in compare(expectedGrants(),
readLiveGrants(environment)):
        add(findings, difference)

    if length(findings) > 0:
        notify("platform", "schema or grant drift in " + environment)
    recordCheckRun(environment, findings)

function checkDriftCheckAge(environment):
    // scheduled separately, so a stopped drift job cannot hide itself
    if now() - lastSuccessfulRun("detectDrift", environment) >
maxRunAge:
        notify("platform", "drift check has not completed recently")
```

有两点使它有用。预期状态由迁移和第 3 章的访问定义推导而来，所以用来对比的真相来源恰好只有一个。并且检查覆盖的是授予，而不只是表，因为对本书而言，出现在任何文件之外的授予才是更重要的一类漂移。发现差异时，处理方式有两种：如果这项变更应当保留，就编写一次迁移把它记录下来；如果不应当保留，就编写一次迁移把它撤销。通过编辑已应用内容的记录来解决漂移，是在隐藏变更，而不是评审变更。

> **注意**：不要以超级用户或所有者的身份直接应用一条纠正语句来“修复”漂移。消除差异的那条语句，本身就是一次对生产环境未经评审的变更。应当把它放进迁移，让它走通道。

新表、新列以及随之而来的授予

第 3 章为新表选择了空默认值：在迁移写明授予之前，除所有者之外没有人能读取这张表。本节是这一决定的另一半，因为这一决定要么坚持，要么被遗忘，都发生在迁移里。

授予写在迁移中

每个创建表的迁移，都在同一个文件中写明它的授予。这样评审人在一份 diff 中就能看到一项变更对访问的全部影响，这正是支持空默认值的论据。以新表 `invoice_adjustments`（一张示意用的表，不属于规范模

式）为例，文件如下所示。它的自由文本 `reason` 列按最坏情形归类为 Confidential，与 `invoice_lines.description` 相同，所以表所有者之外的读取者获得的列级授予会把它排除在外：

```
CREATE TABLE invoice_adjustments (
    id          text primary key,
    invoice_id  text not null,
    reason      text not null,
    delta_minor integer not null,
    created_at  timestamp not null
);

// the migration states every grant it means to give
GRANT SELECT, INSERT ON invoice_adjustments TO app_billing_api;
// reason is Confidential free text: left out of the broad read grants
GRANT SELECT (id, invoice_id, delta_minor, created_at) ON
invoice_adjustments TO app_reporting_job;
GRANT SELECT (id, invoice_id, delta_minor, created_at) ON
invoice_adjustments TO human_readonly;
```

忘记一条授予有两种可能的结果，二者的差别正是空默认值的理由所在。采用空默认值时，需要这张表的服务在第一次查询时就会失败。这很响亮，会在测试或第一次部署时被发现，用一次单行迁移就能修好。采用宽泛的默认权限时，被遗忘的授予会在另一个方向上悄无声息：下一张表可以被默认值所指名的任何人读取，而没人注意到，因为没有任何东西失败。响亮的失败比悄无声息的暴露更便宜。

只有在有东西运行查询时，失败才会是响亮的。这就是我会在流水线中加入迁移后冒烟步骤的原因，该步骤检查每个应用角色是否都能使用它应当使用的表。应用角色没有登录，所以这个步骤需要一种机制。在引擎支持的情况下，我会使用一个测试登录，它是每个应用角色的成员，并依次切换到每个角色；另一种做法是为每个角色各设一个测试登录。两种选择都会扩大流水线的触达范围，因为一个能变成任何服务角色的登录，对攻击者的价值高于任何单个服务凭据。因此它由作业的 workload identity 签发，随作业一同过期，并且只用于这一步骤。作为回报，这个步骤会在服务发布之前抓住被遗忘的授予。

已有授予的表上的新列

新列的表现取决于其所在的表是如何授予的。

表的授予方式	新列……	后果
表级授予	持有该授予的每个角色都可以读取	新的敏感列会立即暴露

表的授予方式	新列……	后果
列级授予	在被点名之前无人可读	新列必须被授予才能使用

对于带有敏感列的表，例如 `customers`、`bank_accounts` 和 `payment_tokens`，这就主张在引擎支持时采用列级授予，这也是第 3 章所做的选择。新列于是一开始就是关闭的，而添加它的迁移，正是团队决定哪些角色可以读取它、它属于 Confidential 还是 Restricted 的地方（分类见第 12 章）。为这些表之一添加列却没有任何授予语句的迁移，就是引出一个问题的提示，而分类检查可以提出这个问题。

在引擎缺少列级授予的地方，适用第 3 章的退路：Restricted 列存放在 `bank_account_secrets` 和 `payment_token_values` 这样的独立表中。新的 Restricted 列属于其中一张表，绝不能放在主表里，因为主表的授予是表级的，而 JIT 访问的上限正是围绕它划定的。这很容易被无意违反，因为向人人都熟悉的那张表添加一列是很自然的做法，而这类错误只有了解该退路的评审人才会注意到。

敏感表上的视图

在许多引擎上，视图以其所有者的权限运行，这意味着建立在带有 Restricted 列的表之上的视图，可能暴露其读取者自己的角色本无法看到的内容。这就是 `classifyMigration` 把 `CREATE VIEW` 标记为访问变更的原因，也是创建视图的迁移要经过平台团队的原因，以及敏感表上的视图要作为访问变更而不是作为便利来判断的原因。

权衡与成本

流水线是一个高价值目标。一份能运行所有者级别 DDL 的凭据，由一个许多人都能编辑的系统持有，这是权力的集中。经过评审的流水线并不能阻止拥有所有者级别触达范围的人做出未经评审的事；它只是让那件事在事后变得可见，而这种控制比听起来要弱。本章中的约束降低了风险，却没有消除它。如果流水线的访问清单很长而且无人负责，应当先解决这一点，再去依赖其余的一切。

评审会增加延迟，而且只能和评审人一样好。针对迁移的四眼检查会拖慢小型变更，并可能沦为形式，尤其是当评审人每周要看十个迁移而每一个都只是加一列的时候。分类检查通过指向危险的行来提供帮助。它不能取代注意力。一份总是在数秒内获批的评审，是需要留意的信号，第 8 章给出了完整的论证。

扩展与收缩会成倍增加变更的数量。一项逻辑上的变更会变成若干次部署。对于有维护窗口、只有一个服务和一张小表的团队，这也许不值得。当许多东西读取该表、而没人能在变更期间让它们停下来时，这一模式才物有所值。

较短的锁等待会让迁移失败。这是预期的行为，也是一种成本。在繁忙的表上，迁移可能需要尝试多次，而最后一次之后会发生什么，必须有人来决定。如果针对自身的工作负载把等待设置得过短，流水线会因噪声而呼叫平台团队。应依据自身的长事务是什么样子来调整它，并写入 runbook。

漂移检查产生的发现需要有人负责。一个向无人阅读的频道发出告警的检查，什么也发现不了。在开启检查之前，先确定每一类发现由谁负责。

何时不采用这种设计。对于一次性的数据库、原型或没有客户数据的单人项目，带有所有者角色的迁移者和经过评审的流水线，超出了这种情形的需要。可以从开发者的机器上运行迁移，让生产数据库远离关键路径，等到团队之外有人依赖这份数据时再加入流水线。本章的设计适用于这样的节点：模式由多个服务共享，而一次错误的变更会让客户付出代价。

常见错误

- **让服务在启动时创建或修改自己的表**。它省去了一个部署步骤，却把所有权交给了服务，其中包括删除和重新授予的权力。在生产环境中关闭自动模式管理，把迁移作为单独的一步，由迁移者来运行。
- **出于“应急”把迁移者凭据存放在团队密码管理器中**。这很诱人，因为流水线有时恰好在最糟糕的时刻失败。它把每一位持有者都变成所有者，并消除了所有权的每次使用都对应一次流水线运行这一性质。应当去建设加急通道。
- **不设锁等待就运行迁移**。迁移会等待到排在它前面的最老事务结束，而排在它后面的每条语句也一同等待。给迁移者设置较短的锁等待，并把重试放在流水线里。
- **把数据修正藏在 backfill 里**。趁着表本来就开着，顺便纠正错误的值，感觉很高效。迁移的评审是为列定义设计的，不是为客户金额的变更设计的。如果这项变更不是依据既定规则对新结构的首次填充，它就是数据修正，遵循第 10 章。
- **在引入新列的同一个版本中删除旧列**。这省下了一次迁移，却去掉了回滚。应让收缩步骤保持独立、延后，并按破坏性变更评审。

- **依赖默认权限，让新表不写授予行也能工作**。宽泛的默认值把被遗忘的授予变成悄无声息的暴露。保持默认值为空，让每个迁移写明自己的授予。
- **在没有列级授予的引擎上，向主表添加 Restricted 列**。表的授予是表级的，JIT 上限就是围绕它们构建的。应把 Restricted 列放进独立的机密表。
- **把漂移发现当作噪声**。一条存在于生产环境、代码中却无处可寻的授予，是一项没有人评审过的访问决定。用一次要么记录它、要么撤销它的迁移来解决。
- **在迁移已应用之后编辑它**。运行过旧版本的环境，现在与运行新版本的环境有所不同。应当另外添加一次新迁移。

要点

- DDL 以 `ledgerly_migrator` 的身份从流水线运行，在一次迁移的持续期间以所有者的身份行事，任何应用角色都不持有任何 DDL 权限。
- 迁移者持有一份常驻的所有者权限，通过短期凭据来行使：只有一个作业持有它，由 workload identity 签发，每次运行在开始之前就有记录，人员从不使用它。
- 因为流水线能触及所有者的权力，它自身的访问清单和定义要像迁移一样接受评审；触及授予或 append-only 表的迁移交给平台团队，加急通道只接受由 `classifyMigration` 强制执行的语句类别白名单。
- 较短的锁等待加上流水线中的重试，使受阻的迁移不会让一张表关闭，而在线工具和增量步骤处理那些原本会长时间持锁的变更。
- 扩展与收缩让新旧代码一起工作，并使回滚成为一次部署，代价是更多的迁移；依据既定规则填充新结构的 backfill 在迁移内部运行，其他任何对数据的批量变更都是数据修正，遵循第 10 章。
- 模式或授予中的漂移，通过把线上状态与迁移及访问定义所描述的内容做对比来发现，并通过迁移来解决，而不是悄悄地编辑。
- 每个迁移为它所创建的内容写明授予，并保持默认值为空，使被遗忘的授予响亮地失败，而不是悄悄暴露一张表。

模式变更是唯一一个由流水线而不是人，持有对数据最大权力的地方。下一部分从谁能触及数据库，转向数据库里面有什么：第 12 章要问的是，哪些列值得保护，如何分类，以及如何在不妨碍需要它们的工作的前提下对其脱敏或加以保留。

第四部分

保护数据

第 12 章 — 敏感数据：分类、脱敏与最小化

设想 Ledgerly 的一个周一。一位客户写信给客服，说一笔付款没有到账，并询问档案中的银行账户是否正确。客服的 Sam 在 `admin-console` 中打开该客户的页面，发现整个账号都显示在屏幕上。在聊天里把它读出来是最快的回答方式，在工单里贴一张截图是把案子交给同事最快的办法。故事里没有人做出不合理的举动，可到了下午结束时，这个号码已经留在聊天记录和工单附件里。

隔着两张桌子，数据团队的 Jordan 正在 `ledgerly_analytics` 上构建报表，注意到那里的 `customers` 表带有联系人姓名、邮箱和电话，而任何报表都用不到这些。值班的 Robin 正在翻看应用日志，想弄清一条提醒为何发送失败，结果在错误消息里发现了收件人的邮箱地址。每一处暴露都很小，三者却是同一种形态：数据来到了一个没有人决定过应当让它出现的地方。

本章为目标状态增加一步：敏感列已分类，默认脱敏或不提供，分析副本中不含原始的 Restricted 数据。本章回答的问题是：每个人和每份副本实际需要看到什么，以及如何构建系统，使其只展示这些，不多一分。

在本书的五部分模型（**身份、授予、路径、执行、记录**）中，本章作用于其中两部分。授予决定某个角色究竟可以读取哪些列，执行决定值经由路径返回时是什么形态。这一切都与凭据如何到达某个人手中无关；一个防护完善的登录若读到了完整账号，该账号照样已经暴露。

问题及其难点

前面各章的控制措施决定谁能访问哪些表、哪些列，却不知道一列为何重要。对 `customers` 的授予把状态标志和联系邮箱一视同仁，schema 里两者都叫 `text`。列定义中没有任何信息表明 `invoice_lines.description` 是自由文本，人们可能在其中填了姓名，也没有表明 `iban` 这类值一旦暴露，轮换任何东西都无法挽回。

第二个难点在于，人们确实需要其中一部分数据，而一条“任何人不得查看”的规则只会催生变通做法。客服必须回答有关付款账户的问题。值班人员必须弄清提醒为何失败。数据团队必须按客群统计客户数。当获准的路径什么都不显示时，未获准的路径就会冒出来：截图、临时导出、“直到项目结束”的常驻授予。有用的问题不是某人能否看到某个值，而是任务需要这个值的哪种形态：完整值、末几位、一个是或否，还是一个计数。

第三个难点是数据不会留在表里。它被复制进分析存储，被写进日志行，在错误消息中被回显，被粘贴进工单，被放进导出文件发出去。主库上的每一项控制都止步于数据库边缘，而值继续向前走。这就是答案必须构建在系统里、不能停留在政策层面的原因。一条写着“谨慎处理个人数据”的政策，迁移流水线、定时加载或日志格式化器是无法执行的。这些决定必须写进会运行的东西里：schema、授予、加载和日志代码。

数据保护与支付卡类制度，例如 GDPR、KVKK 或 PCI DSS，往往会提出一系列问题：持有哪些个人数据，为何持有，流向何处，谁能看到，保留多久。保存在 schema 中的分类，正是能够回答这些问题的形式。任何制度的解释权属于法务与合规人员；本章只描述让答案可查的工程做法。该目录是系统所作所为的证据，并不声称系统满足任何制度；第 14 章会回到访问评审能收集哪些证据。

存在于 schema 中的分类

数据分类把数据划分为命名的类别，每个类别决定处理方式。Ledgerly 使用四类。

类别	含义	Ledgerly 示例	默认处理方式
Restricted	一旦暴露，会以难以挽回的方式损害客户或公司	`iban`、`account_holder_name`、`token_value`	对 `human_readonly` 不提供；绝不出现在即时（JIT）申请中；只能通过应用路径更改；不出现在分析副本中

类别	含义	Ledgerly 示例	默认处理方式
Confidential	关于客户和联系人的个人数据或商业数据	`contact_email`、`billing_address`、`provider_reference`、`invoice_lines.description`	对 `human_readonly` 脱敏或不提供；仅在有审批人的 JIT `SELECT` 中才不脱敏；在分析副本中脱敏或丢弃，但专为展示而存在的展示提示除外（`provider`、`expires_on`；见下方决策表）
Internal	不含个人内容的运营数据	`amount_minor`、`status`、`due_on`、`account_code`	需要它的角色可读；复制到分析副本
Public	可安全公开	八张表中没有	仅保证完整性

Public 行刻意留空。它的存在，是为了让“这能公开吗？”这个问题有沉默以外的答案。

四条规则让这些类别可用。在选择任何工具之前，我会先把它们写下来。

- **对列分类，而不是对表分类。**`customers` 里既有状态标志也有电话号码，二者需要不同的处理。仅出于盘点目的，一张表取其各列中的最高类别。
- **自由文本按最坏情况分类。**供人输入的列，终将装下人们输入的任何东西。`invoice_lines.description` 属于 Confidential，因为它可能包含个人数据，而没有任何过滤器能分辨哪些行包含。
- **派生值继承其来源的类别，直到有人论证并非如此。**`iban` 的带密钥哈希在有人论证其并非 Restricted 之前，仍是 Restricted。`payment_tokens.last_four` 属于 Confidential 而非 Restricted，因为它本来就是用来展示的，这个决定只做过一次，并已写下。
- **未知即 Confidential。**无人分类的列按 Confidential 处理并标记为暂定，使未完成的工作可见，而默认值是封闭的那一个。

与迁移放在同一仓库的目录

存放在 wiki 中的分类会逐渐失真，因为改动 schema 的任何环节都不会去读 wiki。我会把它保存为迁移所在仓库中的一个文件，并在添加该列的同一个 PR 中修改它。有些引擎也允许给列加标签或注释；只要流水线能读到，两种形式都可以。

```
// illustrative catalog entry, one per column
table column_classification:
    table_name              text not null
    column_name             text not null
    class                   text not null    // Restricted,
Confidential, Internal, Public
    reason                  text not null    // one sentence, for the
reviewer
    owner                   text not null    // a team, never a person
    human_readonly          text not null    // readable, masked,
withheld
    analytics_copy          text not null    // copy, transform, drop
    reviewed_on             date null        // null means provisional
```

目录使两项检查成为可能。第一项针对每次迁移运行，与第 11 章所述的标志并列。那些标志是请评审人去查看，这一项则不同，它会阻止合并，因为给一列分类只需一行，而缺失的这一行正是整个控制所要捕获的对象。

```
function checkColumnClassification(migration, catalog):
    problems = []
    for each column in addedColumns(migration):
        entry = catalog.find(column.table, column.name)
        if entry is null:
            add(problems, "unclassified: " + column.table + "." +
column.name)
            continue
        if entry.class == "Restricted" and not
inRestrictedHome(column):
            // a main table whose grants are table-level cannot
withhold it
            add(problems, "Restricted outside its home: " +
column.name)
        if entry.class == "Restricted" and entry.analytics_copy !=
"drop":
            add(problems, "Restricted column would reach analytics")
    if problems is not empty:
        raise MergeBlocked(problems)
```

这里的 `inRestrictedHome` 指授予能够不提供该列的表：具有列级授予的主表，或者在没有列级授予的引擎上，第 3 章所述的单独机密表。添加该列的人拿不准时，出路不是跳过检查，而是把该列分类为 Confidential 并将 `reviewed_on` 留空，让下一次访问评审来定。

第二项检查按计划运行，将线上 schema 与目录比对，方式与第 11 章把线上 schema 与迁移比对相同。某个倒霉的夜里手工创建的列，会作为带有负责人的发现项浮现出来，而该发现项通过一次迁移和一行目录来解

决，而不是悄悄修改。

最小化：不持有的数据

最小化就是不收集、不复制、不保留无人需要的东西。它是本章中成本最低的控制，因为从未存储的值，无需维护脱敏，没有副本要找，也没有需要遮蔽的行。本章其他所有控制都以数据存在为前提；这一项则追问它是否应该存在。

对每个敏感列，我会按以下顺序问四个问题。

1. 是否有任何功能或职责需要这个值？如果回答是“它可能有用”，那么答案就是否。
2. 能否改为持有替代值？Ledgerly 持有 `payment_tokens`，即由支付提供方签发、用来代表支付工具的 token，而不持有支付工具本身。仅这一个决定，就从 schema 中去掉了一整类值。
3. 能否少持有一些？像 `last_four` 这样的展示提示，让客服和客户无需持有支付工具本身就能认出它。
4. 能否持有更短的时间？已关闭客户的联系方式，在最后一张发票结清、争议期限过去之前是有用途的，此后只保留一个简短而明确的期限，然后删除。

有两个关于数据存放位置的决定，比任何保留规则都重要。第一，**append-only 表不应携带个人数据**。`ledger_entries` 保存 `amount_minor` 和 `account_code`，二者都是 Internal，所以更正可以是一个新条目：为了落实某个人的请求，表里没有任何内容需要被抹除。`audit_events` 出于同样的原因，用不透明的 `subject_id` 指代其对象。第二是自由文本。取值固定的结构化字段可以分类、可以脱敏；备注框两者皆不行，所以在功能可以使用编码、状态或产品引用来代替一句话的地方，我会选前者。

保留是一项作业，而不是政策里的一句话。必须有东西按计划运行，找出超过期限的行，删除或匿名化它们，而这个东西需要自己范围很窄的权限、自己的审计条目和一个负责人。至于数据该保留多久，我不知道，本书也不知道。它取决于合同、监管方和自身争议处理流程的要求，必须先由法务人员回答，工程师才能编写该作业。工程师能说的是：备份、导出和分析副本里同样保存着旧数据，只触及主库的删除并未完成。第 13 章会回到这一点。

脱敏：动态与静态

脱敏对值进行替换或模糊处理，使真实值不可见，而形态仍然可用。**遮蔽**是其粗暴版本：把值从输出或日志中移除。两种脱敏的作用位置不同。

	动态脱敏	静态脱敏
变化的内容	查询返回的内容	副本本身
时间	读取时，按读取者	加载时，一次性
原始值是否仍被存储	是，在同一存储中	副本中没有
典型用途	副本上的 `human_readonly`	分析副本、staging 数据
失效方式	引擎行为、过滤条件、直接授予	过期或裁剪过度的副本

面向人员的动态脱敏

`human_readonly` 是每个会话的默认起点，所以它的脱敏是多数人最先碰到的。下面的写法仅作示意，因为各引擎表达脱敏的方式差别很大：有的提供列上的掩码，有的提供取消脱敏的权限，有的为每张表提供一个固定视图。

```
// illustrative notation. The grant lists the columns the role may
read;
// each mask rule changes what a read through the role returns
GRANT SELECT (id, created_at, status, contact_name, contact_email,
              contact_phone, billing_address) ON customers TO
human_readonly;

MASK RULE contact_name_initials   ON customers (contact_name)
    FOR human_readonly USING initials();
MASK RULE contact_email_partial   ON customers (contact_email)
    FOR human_readonly USING keep_first(1);
MASK RULE contact_phone_tail      ON customers (contact_phone)
    FOR human_readonly USING keep_last(2);
MASK RULE billing_address_hidden  ON customers (billing_address)
    FOR human_readonly USING redact();
MASK RULE provider_reference_tail ON payments (provider_reference)
    FOR human_readonly USING keep_last(4);

// free text has no shape to mask, so it is withheld: not in the
column list
// (illustrative column list)
GRANT SELECT (id, invoice_id, created_at) ON invoice_lines TO
human_readonly;
```

函数的选择取决于该列的用途。技术本身没有优劣之分；每种都有其适用之处，也有其失效之道。

技术	读取者得到什么	适用于	何时不再有用
部分显示	开头或末尾的几个字符	屏幕、粘贴的结果	较短的值几乎全部显露
遮蔽	固定标记或空	地址、自由文本	无法确认任何事
带密钥的假名	稳定的替代值	跨次加载的关联	轮换密钥会使关联失效
粗化	范围更宽的值，例如国家或月份	分析	组合起来的列可能锁定某个人（第 13 章）

清单上没有普通哈希。取值来自小到可以逐个尝试的集合（例如电话号码）时，其哈希可以通过逐个尝试而被还原，因此假名需要一个存放在数据库之外的密钥。

并非每个 Confidential 列都要脱敏。专为展示而存在的列保持可读：`payment_tokens.provider`、`last_four` 和 `expires_on` 是让人认出支付工具的展示提示，对它们脱敏只会把人们推向它们所替代的 Restricted 值。联系字段、账单地址和 `provider_reference` 能识别一个人或一笔交易，要脱敏。`invoice_lines.description` 不提供。

由此得到第 8 章所依赖的规则。`human_readonly` 本来就能读，所以只有对该角色已脱敏或不提供的 Confidential 列，才需要 JIT `SELECT` 三元组，而它所增加的，就是该列的未脱敏形态。Restricted 列绝不出现在 JIT 三元组中，所以这里对它们没有任何改变。因此，申请和审批人的界面应当单独带一行明确的说明，“未脱敏列：`contact_email`”，与表清单分开，让审批人看到申请去掉了哪些脱敏，而不仅仅是触及了哪些表。

> **警告**：应当检查引擎如何处理这样一种登录：它既通过 `human_readonly` 读取某列，又持有对该列的直接列级授予。若脱敏优先，JIT `SELECT` 不会显示任何新内容，审批也就没有意义。若对所有读取者都是原始值优先，脱敏就只是摆设。依赖任何一种结果之前，都应当在副本上测试这两种情形。

技术旁边还有两条局限。第一，在许多引擎上，脱敏作用于查询返回的内容，而不作用于其条件所比较的内容，因此允许对已脱敏列进行过滤的读取者，可以通过计数和匹配了解该列取值的一些信息。应当把动态脱敏视

为防范偶然暴露的保护，例如屏幕、截图或粘贴的结果。第二，对输出脱敏对已存储的值毫无作用，该值依然保留在主库的每一份备份、副本和转储中。

第一条局限给出一条针对推断的判定规则。要问读取者通过过滤还原该值的成本有多低：对 `contact_email` 做少量等值探测很便宜，按已脱敏电话前缀统计客户数同样便宜。成本低且读取者不得获知该值时，应当不提供该列而不是脱敏，或者对该角色禁止在已脱敏列上使用谓词。成本高时，例如通过数千条已记录的语句逐字符探测一个很长的自由值，脱敏加语句记录就是合理的控制。

Result stream 中的脱敏

有些访问路径不能指望引擎来脱敏。位于人和数据库之间的 gateway，可以在行到达屏幕或结果文件之前，对 result stream 本身套用规则。在引擎没有脱敏功能时这很有用，而且无论背后是哪种引擎，规则都可以相同。

stream 规则依赖于知道值来自哪一列，而 `SELECT api_key AS k` 这样的别名会让按列名匹配的规则失效。对结果做脱敏的工具（QueryProxy 即是其一）会加入同时按值匹配的规则，这是第二层，绝不能取代授予。截至撰写时，它自己的文档就写明了按名称匹配规则的这一局限，其他工具也采取类似做法。对 Ledgerly 而言，结论是 stream 脱敏位于第 3 章的列级授予之后，一个角色绝不能看到的列，由授予来不提供，而不是交给某种模式。

> **注意**：在没有列级授予的引擎上，Ledgerly 使用第 3 章的备用方案：Restricted 列放在 `bank_account_secrets` 和 `payment_token_values` 中。此时对 `human_readonly` 脱敏通常意味着每张表一个固定的脱敏视图，由一次迁移创建一次，归 `ledgerly_owner` 所有，该角色对视图持有 `SELECT` 而不对基表持有。对基表的表级 JIT 授予会把所有非 Restricted 列（包括 Confidential 列）以原始形态展示，而审批人接受的恰恰就是这一点。这是固定视图，不是为每个申请单独创建的视图，它是针对 Confidential 列的脱敏机制。它不是针对 Restricted 列的第二种备用方案：后者唯一的备用方案是机密表。

面向分析副本的静态脱敏

分析副本是脱敏最省成本、最有用的地方，因为原始值完全可以不进入副本。Jordan 的报表需要客群、计数和金额，不需要姓名、邮箱或电话，所以对大多数 Confidential 列，正确的处理是 `drop`，转换留给分析人员确实在用的少数列。

加载是一项定时作业，而不是服务，它以自己的登录和角色 `app_analytics_load` 运行，与访问评审中的其他 principal 一样。该角色只对目录未标记为 Restricted 的列持有 `SELECT`，所以授予是第一道防线，目录是第二道。二者并不独立，因为授予是根据目录的 Restricted 标记派生出来的：被误分类为 Internal 的列会同时通过两者。授予所增加的是：即使某条转换条目有误，被标记为 Restricted 的列也不可能被读到；独立的检查是 `verifyMasked` 中的 schema 断言，它将副本与计划做比对。在该授予范围内，加载按目录逐列决定，并且在失败时关闭。

```
function loadAnalyticsTable(table, catalog):
    plan = []
    for each column in sourceColumns(table):
        entry = catalog.find(table, column)
        if entry is null:
            // an unclassified column never travels; the load says so
            notify("platform", "not loaded, unclassified: " + table +
"." + column)
            continue
        if entry.class == "Restricted" and entry.analytics_copy !=
"drop":
            raise CatalogInvalid
        if entry.analytics_copy != "drop":
            add(plan, { column: column, treatment:
entry.analytics_copy })
    // build beside the live table, check, then swap: a failure leaves
the
    // previous load in place, so the copy never holds a half-masked
table
    next = table + "_next"
    writeMasked(next, from: table, plan: plan)
    if not verifyMasked(next, plan):
        dropTable(next)
        raise MaskingCheckFailed
    swapTable(next, into: table)
```

`verifyMasked` 做两项检查。它断言 schema：目标的列集合必须与计划相等，其中不得存在 Restricted 列或已丢弃的列。然后它抽样若干行，如果某个经过转换的列仍等于其源值，就判失败。schema 断言能发现经

由计划未列出的路径到达的列；抽样能发现配置错误的转换。二者都不能证明副本不含任何敏感内容，因为抽样只读取了部分行。

失败时关闭的代价是看得见的：开发者添加了一列却忘了目录，本该由迁移检查拦住，若没有拦住，分析表加载时就不带这一列，数据团队会发现缺口。我愿意接受。缺了一列的报表只是一张工单；带有未脱敏列的副本则是无人察觉的暴露。

用于关联的带密钥假名值得单独做一个决定。它让分析人员跨多次加载追踪同一位客户而不知道客户是谁，而它只有在密钥始终不在数据团队触达范围之内且保持不变时才有效。密钥一轮换，所有假名都变，历史就无法再关联。对许多团队而言，不透明的 `customer_id` 已经承担了这项工作，再加一个替代值只是多出一个要守护的密钥，毫无收益。

令牌化与应用层加密

令牌化把一个值替换为一个替代值，该替代值只能由受控的服务映射回去。它不是脱敏，因为映射是存在的；它通常也不是加密，因为替代值通常根本不是由值计算得出的。那些在密钥作用下由值计算出替代值的方案，不过是换了名字的加密，需要同样的密钥保管。Ledgerly 已在一处受益：`token_value` 是支付提供方的 token，映射由提供方持有，而 `payments-service` 是唯一的调用方。因此 Ledgerly 数据库从不持有支付工具本身。

为银行账号再建一个令牌化服务，是更大的一步。它给支付路径增加了一个系统，该系统必须在支付可用时同样可用，需要一份可以将 token 映射回去的调用方清单，每一次映射都是一个应当有人能够评审的事件。以 Ledgerly 的规模，我不会建。限制这些列、限制读取它们的那个服务并对值加密，就能以更少的运维负担得到大部分保护。如果受监管的合作方或支付提供方要求这样做，则由该要求决定。

应用层加密指服务在数据库看到值之前就对其加密，因此角色、副本、备份和转储里保存的都是密文。这是它独有的长处：它保护值在授予无法触及的数据副本中的安全（第 13 章）。由存储层完成的加密（在磁盘上或快照中）能防止卷丢失，对查询则毫无作用，团队常把它误当作访问控制。应用层加密更接近真正的边界，但它是更重的承诺。

- **密钥决定谁能读**。数据的保护程度只取决于它的密钥。应当只把解密权限授予 `payments-service` 的 workload identity，其他一概不授

予，这样决定就归结为谁能解密、通过什么途径解密。密钥层级、轮换和密钥备份自成一门学问，密钥服务的文档比本书讲得更好。

- **该列失去查询功能**。对密文无法按通常方式搜索、排序、范围过滤或强制唯一性，因此应当选择只通过标识符读回的列。
- **能按请求解密的服务，就可能被要求解密**。加密把边界移到了服务上，因此它的调用方、调用速率和审计条目成为控制，且审计条目要在解密之前写入，如下所示。

```
function readPayoutAccount(accountId, onBehalfOf):
    row = selectBankAccount(accountId)       // the stored value is
ciphertext
    // the audit entry comes first: if the log is unreachable, nothing
is decrypted
    try:
        auditLogAppend({ actor: "payments-service", action: "decrypt",
                         subject: row.id, on_behalf_of: onBehalfOf })
    on AuditLogUnavailable:
        raise DecryptRefused
    // the key service checks the caller's workload identity, not the
request
    return keyServiceDecrypt(row.iban, keyId: row.key_id)
```

加密之后，矩阵不变。`human_breakglass` 仍然能读 Restricted 列，但读到的是密文，解密需要该角色不具备的密钥权限。我会事先决定，需要明文的事故能否申请密钥访问、由哪位审批人批准，并写入第 8 章的 break-glass runbook。把这个选择留到事故当中，规则就是这样被临时拼凑出来的。

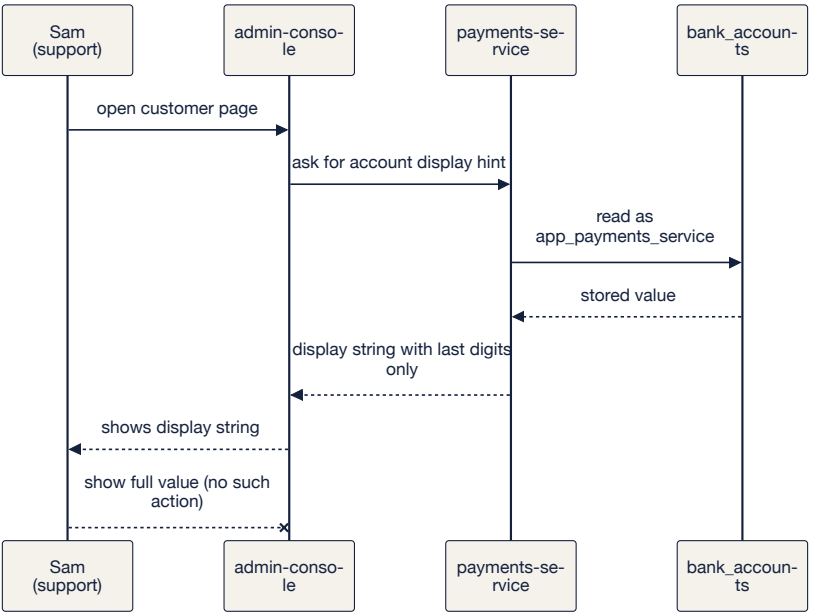
看不见也能工作：客服案例

回到 Sam。目标是让 Sam 能完整地回答客户，同时从不看到完整的值。多数客服问题最后发现，需要的比这个值本身要少。

客服要做的事	需要看到什么	如何提供
告知客户档案中是哪个账户	末几位	`payments-service` 返回展示字符串；或者在写入账户时存下提示，就像为 token 存下 `last_four` 那样
核实来信者就是联系人	一个是或否	在已持有授予的服务中做一次窄范围的检查，它把客户所述内容与存储的值比对，回答“匹配”或“不匹配”

客服要做的事	需要看到什么	如何提供
查看客户的发票	仅限该客户的状态、金额、到期日	`app_admin_console` 读取 `invoices` 的 Internal 列，第 3 章所述针对案件客户的行规则收窄行范围
修正错误的账号	什么也不需要，也不能手工操作	应用路径：`admin-console` 请求 `payments-service`，由第二个人批准，变更记录在 `audit_events` 中

图示展示了第一行。看最后一个箭头：一个不存在的操作，比一个存在却被拒绝的操作是更强的控制。



起步时我不会做“显示”按钮。要求填写理由并记录点击的显示功能听上去合理，但一个月内点击就成了习惯，它的日志也成了没人读的东西。如果出现了真实需求，应当把它做成需要审批人的申请，就像第 8 章对任何未脱敏的 Confidential 读取所做的那样，并统计使用频率。每天都在使用的功能，是在说明展示形态还缺了点什么。

值班人员面对不同的压力，遵循同样的规则。Robin 调试一条失败的提醒，需要知道收件地址被拒收了，却很少需要地址本身。Robin 确实需要未脱敏的 `contact_email` 时，那就是一次点名该列的 JIT `SELECT`，由阅读列清单的审批人审批。值班读取从不自动批准：跳过 Confidential

列的规则，批准的是一个什么也不授予的申请，而包含它们的规则，则去掉了唯一有意义的检查。代价是在糟糕的时刻等待，偿还的办法是在常见情形中消除对该值的需要，下一节就讲这一点。

日志和错误消息中的敏感数据

日志是无人设计的数据副本。它们由许多组件写入，被许多人阅读，保存很长时间，并通过从未纳入访问模型的工具来检索。四个地方造成了大部分泄露。

- **应用日志和错误消息**。由值构造的消息，例如“无法发送到该地址”，会把值放进日志、错误追踪器，有时还会出现在客户的屏幕上。开启调试日志时，驱动和 ORM 也可能记录绑定参数。
- **数据库日志**。语句日志可能记录字面量，慢查询日志会连同其中的值一起保留查询文本。
- **语句记录**。access gateway 会记录每个人发送的每条语句（第 7 章），而在条件中键入某个值的人，就把该值放进了记录。
- **工单和聊天**。为说明问题而粘贴的结果，是受众不对、没有终止日期的副本。

> **警告**：语句日志里有字面量，所以第 14 章的审计日志和 gateway 的语句记录本身就是敏感数据的存储。应当把语句记录为遮蔽了字面量的指纹，将该存储至少分类为 Confidential，限制谁能读取，并为其设定保留期限，否则监视数据的控制就会变成数据的又一份副本。机制由第 14 章负责。

对于应用日志，我偏好的规则是允许列表，而不是清洗器。清洗器查找与敏感值相似的模式并将其移除；它能抓到作者想到的形态，漏掉其余的。允许列表点名可以出现的字段，丢弃其他一切，因此新字段在有人选择记录它之前不会出声。

```
function buildLogEvent(event, allowedFields):
    safe = {}
    dropped = 0
    for each field in event.fields:
        if field.name in allowedFields:
            safe[field.name] = field.value
        else:
            dropped = dropped + 1
    // the count is the signal: a field dropped every day is either a
    // missing allowlist entry or a value somebody tried to log
    safe["dropped_field_count"] = dropped
    return safe
```

错误消息遵循同样的纪律。错误携带一个编码和若干标识符，例如发票 id、客户 id 和投递状态，而不是失败的那个值。Robin 的提醒案例也由此不再需要原始地址：如果失败记录保存的是“收件人被拒收”这样的状态和发票 id，日志无需地址就能回答问题，而 gateway 中谁读过该地址（如果有人读过的话）的记录，也只留在一处。

日志流水线中的遮蔽是兜底机制，它会漏掉一些东西。我不会跳过的检查是测试中的 canary：在每个敏感字段中放入一个形态可识别的占位符来运行应用，若它出现在任何日志输出中，就让构建失败。它在关键之处检验允许列表，只需几行代码。当日志里是标识符而不是值时，调试会变慢，由此产生的“为什么我看不到这个”的疑问，其去处是 `dropped_field_count`，而不是某位开发者在生产环境中重新打开参数日志。

Ledgerly 的列决策

下表把本章汇总起来。它是八张表中敏感列的决策记录，格式便于访问评审阅读。在该列中，“不提供”和“无访问权限”都表示该角色对该列没有任何授予，因此点名该列的查询会被拒绝。`human_readonly` 列说明会话的默认情况；JIT 列说明申请能否点名该列。

列	类别	`human_readonly`	JIT `SELECT`	分析副本
`bank_accounts.iban`、`account_holder_name`	Restricted	无访问权限	绝不	已丢弃
`payment_tokens.token_value`	Restricted	不提供	绝不	已丢弃

列	类别	human_readonly	JIT SELECT	分析副本
payment_tokens.provider、last_four、expires_on	Confidential	可读	不需要	保留 provider 和 expires_on，丢弃 last_four
customers.contact_name、contact_email、contact_phone	Confidential	已脱敏	点名该列，不脱敏	已丢弃
customers.billing_address	Confidential	已遮蔽	点名该列，不脱敏	已丢弃
payments.provider_reference	Confidential	末四个字符	点名该列，不脱敏	已丢弃
invoice_lines.description	Confidential	不提供	点名该列，不脱敏	已丢弃
amount_minor、currency、status、due_on、account_code	Internal	矩阵授予之处可读	不需要	已复制

有两点值得说明。Restricted 列从分析副本中丢弃，也不向 human_breakglass 之外的任何人员角色开放；唯一能更改它们的路径是应用路径，并由第二个人批准（第 9 章讲述执行策略如何把需经 gate 的操作转到该路径）。同一路径也负责所有者关联：bank_accounts 和 payment_tokens 上的 customer_id 因为不属于 Restricted，所以仍在 JIT 上限之内，然而把一行移到另一位客户名下，会改变这个账户属于谁，因此所有者关联变更要经过 gate，并转交产品路径，绝不作为普通的数据修正来做。而在引擎没有列级授予的地方，同一张表依然成立，只有一处不同：Restricted 列存放在机密表中，JIT 列写的是“表级”，而审批界面会用文字说明这一点。

权衡与成本

分类需要持续维护。目录在写成的当天是正确的，但随即开始老化。需要有人负责每一项类别决定，评审暂定的条目，并回答未分类列所引出的问题。如果无人评审目录，它就会变成那种完整却错误的记录，这比坦率的

缺口更糟。

脱敏消耗调试时间。Robin 看不到原始邮箱，所以依赖其确切字符的缺陷，需要一次 JIT 申请和一位审批人。为了本章所关闭的访问，我接受这一成本。它并非没有代价，每晚都要付出这一代价的团队，会开始寻找绕过的办法，所以应当度量未脱敏读取被申请的频率，并修复成因。

动态脱敏依赖引擎。过滤条件、直接授予和视图方面的行为各不相同，没有在所用引擎上测试过的脱敏，只是一种希望。基于视图的变体增加了对象，迁移必须让它们与基表保持同步。

失败时关闭会产生缺口。拒绝未分类列的分析加载，会让数据团队在有人分类之前缺少一列。应当指明谁负责这段等待，并让分类评审足够快，使缺口以小时计，而不是以周计。

应用层加密去掉功能，增加依赖。密钥服务位于该列每次读取的路径上，它的中断就是支付的中断。因为审计条目在先，审计日志不可达也会阻止这些读取，这是有意的耦合，也是又一项可用性依赖。查询功能的丧失把搜索推向其他列，而密钥备份又是一件必须测试的事。

这里的一些控制可能沦为安全剧场。一个总会被点击的取消脱敏按钮、一份无人评审的目录、一个无人测试的日志清洗器以及一个任何读取者都能用过滤条件探测的已脱敏列，增加的是摩擦或文书工作，什么也没有减少。

何时不必全部采用。一个只有寥寥几张表、没有客户个人数据的小系统，不需要目录文件、脱敏加载或密钥服务。一页纸的敏感列清单、把这些列排除在外的列级授予以及日志允许列表，就能带来大部分价值。第二类受众需要这些数据时再加脱敏，当无法通过授予绕开的某份副本成为关注点时再加加密，并写下什么事件会促使转入下一步。

常见错误

- **把静态数据加密当作访问控制**。它之所以诱人，是因为只需一项设置，还能满足检查清单。它能防止磁盘丢失，对查询毫无作用。应保留授予，并在需要保护某份副本的地方使用应用层加密。
- **在客户端脱敏而不是在数据库中脱敏**。Web 界面中的脱敏容易构建，而直接连接、导出或 API 响应会绕过它。应当在数据离开存储的地方脱敏。

- **对表分类而不是对列分类**。每张表一个标签，会对状态标志保护过度，对电话号码保护不足。应对每一列分类，并由各列推导出表的标签。
- **新列默认为 Internal**。Internal 是阻力最小的类别，于是成了一切未经评审之物的类别。应当把 Confidential 作为默认值，并通过评审来解除。
- **用模式给自由文本脱敏**。模式能找到作者预期的数字和邮箱，却漏掉记在备注里的姓名。应当对广泛的读取者不提供自由文本列。
- **用不带密钥的哈希作假名**。它看上去不可逆。对可能取值很少的值，可以通过逐个尝试还原。应当使用密钥保存在数据库之外的带密钥假名，或者直接丢弃该列。
- **把生产数据复制到 staging，“事后再脱敏”**。复制与脱敏之间的窗口，就是一份全部内容的副本。应当边走边脱敏，就像分析加载那样，并把 staging 视为第 13 章中的一扇门。
- **为省一步而把 Restricted 列加进 JIT 申请**。上限拒绝它是有原因的，而夜里诱惑最强。Restricted 值只能通过应用路径并经四眼原则审批来更改，在没有列级授予的引擎上，它们放在机密表中。
- **自动批准值班读取**。它去掉了让人检查正在取消脱敏的是哪个 Confidential 列的那次停顿。应保留审批人，并减少需要该值的次数。
- **在 error 级别记录请求体**。请求体对开发者最有用，也包含客户键入的每个字段。应当记录标识符和状态，其余能出现什么，交由允许列表决定。

要点

- 对每一列分类，自由文本按最坏情况处理，并让未分类列默认为 Confidential，使新数据从封闭状态开始，而打开它的工作是可见的。
- 把分类目录放在保存迁移的仓库中，并阻止合并任何添加了未分类列的变更，因为这项检查只需一行，缺了它就是失败。
- 最小化优先：从未存储的数据无需脱敏，而不进入 append-only 表的个人数据，不会变成删除难题。
- 动态脱敏适合 `human_readonly`，防范偶然暴露；静态脱敏适合分析副本，那里可以把原始值完全丢弃，并且加载遇到任何未分类内容都会在失败时关闭。

- 向人提供其任务所需的值形态，例如展示字符串或一个是或否，并让 Restricted 列不进入任何 JIT 申请和任何原始 SQL 写入。
- result stream 中的脱敏是授予之后的第二层，绝不能取代授予；无法读取 Restricted 列的加载角色，使目录错误以失败即关闭的方式收场。
- 日志、错误消息和语句记录都是数据副本：通过允许列表记录标识符，不让值进入消息，并对审计日志本身分类。

本章的控制措施保护的是数据停留在数据库中、以及被查询返回时的安全。下一章跟随数据向外走，进入备份、转储、导出和 staging 副本，那里这些角色一概不适用。

第 13 章 — 备份、导出与其他旁门

设想 Ledgerly 的一个周五下午。Robin 在追查发票提醒中的一个缺陷，它只在接近真实的数据上出现：到期日不寻常的发票、拥有多个联系人的客户、描述列里装着没人预料到的自由文本。staging 里只有十几个手工造出来的客户，这个缺陷在其中任何一个上都不出现。有人想起共享存储里有一份较早的生产转储，负责数据库的 Alex 把它恢复到 staging 中，好让整个团队一起看。

缺陷在一小时内就找到了。转储仍留在文件夹里，恢复出来的数据库仍留在 staging 里。staging 没有 access gateway，因为它不是生产环境。它的登录是共享的，每位工程师都能连上，CI 作业在其上运行，上个月刚加入的一位外包人员也有账号。staging 中的 `invoice-worker` 使用与生产相同的出站邮件设置，其提醒作业也是开启的。

没有人决定过暴露客户数据。每一步都是对一个下午的问题的合理回答，而前面各章构建的控制措施，保护的是一个已不再独占其内容的数据库。

本章为第五部分之前的目标状态补上最后一步：备份、导出和 staging 副本被当作有负责人、有期限的门来对待。问题是这样的：数据出于某种目的离开主库之后，无论是为了恢复、为了供应商、为了测试，还是为了需要查看的人，怎样才能让控制措施跟着数据走，而不是留在它所来自的那个数据库里？

在五部分模型（**身份、授予、路径、执行、记录**）中，副本是授予和路径不再适用的地方，因为数据是经由一条两者都不携带的路线离开的。本章要做的，是让副本自己携带一份授予：一份读取者名单、一个负责人和一个终止日期，并留有一条记录，说明是谁创建了它、又是谁删除了它。

问题及其难点

第 6 章把**旁门**定义为绕过主库访问控制的数据拷贝或导出路径。副本和分析副本是刻意构建的门，可以点名。本章讲的是其余的：备份和快照、转储、CSV 文件、staging 数据库、笔记本电脑上的几千行数据、仪表

盘缓存、供应商收件箱里的一个文件。其中大多数是出于正当理由创建的。很少有负责人。

核心难点在于，访问控制存在于系统之中，而不在数据之中。主库有 access gateway、每个会话一个登录、列级授予、超时和审计日志。主库的一份转储一样也没有，因为转储只是一个文件。能读取这个文件的人拥有其中的一切，包括角色 schema 对除 `human_breakglass` 之外的所有人员角色不提供的那些列。包含 Restricted 列的备份，对能读到它的人来说，就是对这些列的一份长期授予，既不是 gateway 签发的，也没有任何评审会把它列出来。

显而易见的对策是一条政策：不得拷贝生产数据。它失败的原因很简单。拷贝正是工作得以完成的方式。恢复在定义上就需要一份拷贝。供应商需要文件。依赖真实数据形态的缺陷，需要看起来真实的东西。一条全面禁止的政策会被绕过，由此产生的拷贝就是无人管理的拷贝。更好的问题是：需要哪些拷贝，每份由谁负责，谁可以读取，何时终止。

第二个难点是，这些门由不同团队出于不同理由创建，所以没有哪个负责人能看到全部。平台团队负责备份，数据团队负责分析副本，后端团队负责 staging，财务负责供应商关系，每位工程师负责自己笔记本电脑上的内容。盘点清单是第一项控制，它存在于存储桶、作业计划和配置之中，而不在政策文档里。

旁门地图

在讲控制措施之前，先画一张地图。下表列出本章涵盖的门，每扇门在 Ledgerly 的情形下装着什么，以及在无人另行设计的情况下默认谁能触及它。

门	装着什么	默认谁能触及	第一项控制
备份与卷快照	一切，包括 Restricted 列和机密表	任何有存储或基础设施访问权限的人	创建、恢复和删除的权力分开；加密使用独立的密钥
共享存储中的转储	一切，以可读文件的形式	任何能打开该文件夹的人	找出来、指定负责人、按备份规则删除或移走
人员导出（CSV、粘贴的表格）	查询返回的任何内容	先是本人，然后是他们发送给的任何人	分层审批、到期、不含 Restricted 数据
Staging 和测试数据库	生产的拷贝，或形似生产的数据	每位工程师、CI、外包人员	脱敏或合成数据，关闭出站消息

门	装着什么	默认谁能触及	第一项控制
笔记本电脑	结果集、本地数据库、保存的文件	笔记本电脑的使用者及其备份	消除需要；改用脱敏样本
BI 工具	缓存的提取、定时报表、共享仪表盘	工具共享给的所有人	只读取分析副本，尽可能按个人身份
供应商文件	发票与账簿导出	供应商的员工和系统	窄角色、列清单、保留期限、文件到期

每一行的问题形态都相同：一份数据拷贝，只有文件系统或 SaaS 产品的控制措施，以及一份没人写下来的读取者名单。第 6 章的登记册正是出于这个原因而存在，我会把它扩展到上表的每一行。不在登记册里的门，没有人会去评审。

```
table data_copy:                        // extended from Chapter 6
    name         text primary key       // "staging_refresh_current"
    kind         text not null          // backup, dump, export,
staging, laptop,
                                        // bi_extract, vendor_file,
replica, analytics
    source       text not null          // what it was made from
    contains     text not null          // highest data class present
    readers      text not null          // roles, logins or groups
    storage      text not null          // where the bytes live
    owner        text not null          // a team, never a person
    ends         timestamp null         // null only for replicas and
analytics
    review_due   timestamp not null     // when the row is next reviewed
    deleted_by   text null              // who removed it, once removed
```

新增的字段是 `storage` 和 `deleted_by`；`kind` 增加了取值 `backup`、`laptop`、`bi_extract` 和 `vendor_file`，而 `contains` 记录其中出现的最高数据类别。第一个新字段说明了有人问“什么能读取这份拷贝”时该去哪里找，因为答案通常是一项存储权限，而不是数据库角色。第二个新字段让闭环成立：没人执行的 `ends` 日期只是一个愿望，而谁删除了该拷贝的记录，才是这个日期确有意义的证据。

备份与快照

备份是唯一没人反对的旁门，而这种一致认同，正是它常常失去控制的原因之一。它必须包含一切，因为它的任务就是恢复一切。这就是它的全部设计，也是它的全部暴露面。

备份包含什么

备份有几种。**物理备份**或卷快照复制数据库的文件，通常还带有对变更的持续归档，以便把数据库恢复到某个时间点。**逻辑转储**通过数据库读取数据，并把它写成语句或行。两者都包含每一张表。物理备份还包含集群的角色定义，以及引擎为这些角色保存的任何凭据材料，有些逻辑工具也包含，取决于使用方式。

由此产生两个后果，二者在主库的示意图中都看不出来。第一，Restricted 列在备份里，在使用机密表的引擎上，机密表也在其中（第 3 章的 `bank_account_secrets` 和 `payment_token_values`）。角色 schema 规定只有应用角色和 `human_breakglass` 能触及这些值，这描述的是数据库，对文件则只字未提。

第二，备份保存的是它被创建那天的状态。共享 admin 登录停用之前创建的备份，仍包含该登录。它保存着此后已换团队的人的角色成员关系。轮换和撤销发生在运行中的系统里，不会回溯到已经存在的文件。有人收窄主库上的某个角色时，去年那份备份的恢复不会有任何变化：主库改善了，而各份拷贝保留着它们被创建那天的状态。

恢复即读取

我会在备份设计的最前面写下一句话：恢复的权力就是读取一切的权力。能把备份恢复到自己控制的地方的人，随后可以在那里以所有者身份登录并读取每一列，而生产环境中的 gateway、列级授予和脱敏规则一概不适用，因为读取并不发生在生产环境中。

这改变了控制归谁所有。备份访问通常由基础设施角色授予：有人能读取存储桶、挂载快照或运行恢复工具。这些角色虽然位于 `human_readonly` 之上，却很少得到同等细致的评审。因此，访问评审要问的不仅是“谁能查询主库”，还有“谁能恢复，恢复到哪里”。

我会把三种权力分开，交给不同的 principal：

权力	含义	由谁持有
创建	创建备份并写入存储	备份作业自己的身份，对存储只写
恢复	读取备份并在某处启动	平台团队，按申请，由第二个人批准（四眼原则）
删除	移除或使备份过期	保留流程；人员只能通过经过评审的申请

这样拆分的要点在于，写备份的作业无法把它们读回来，而且没有人既能恢复备份又能删除恢复点，因此恢复之后不可能紧接着删掉被恢复的对象。在存储支持在一段时间内一次写入、不可更改的保留时，我会出于同样的理由使用它，代价是：在这段期限结束之前，无法删除误创建的备份，例如从一个短暂装过不该存在的内容的数据库创建的备份。应当事先决定这种情况如何处理。

加密，以及谁持有密钥

备份应当静态加密，有意思的问题是谁持有密钥。如果同一位管理员既能读取存储桶又能使用密钥，那么加密只能防范磁盘丢失，别无其他。如果密钥由不同于存储管理员的 principal 持有，而恢复需要二者同时具备，那么存储团队单独无法读取数据，密钥持有者单独则无物可读。

这种分离有明确的代价。恢复现在需要两方，而密钥丢失的备份就不再是备份。必须存在密钥恢复路径并经过测试，这又是平台团队要维护的一项流程。第 12 章讲了应用层加密和密钥管理的局限；这里更窄的一点是，备份加密是恢复的一道 gate，而 gate 只有与存储上的那道不是同一道时才值得设置。

加密对获准恢复的人毫无帮助。它保护的是传输中和静态的文件，使其不被不属于恢复路径的人和系统读到。我会在设计文档里直白地说明这一点，因为“备份是加密的”常被听成“备份是安全的”，而这是两种不同的说法。

恢复请求

恢复是备份变成运行中数据库的时刻，所以控制应当放在这里。下面的代码块展示一个恢复请求，并写明了操作顺序。备份存储、登记册和审计日志是各自独立的系统，因此没有任何事务能涵盖它们。

```
function restoreBackup(request):
    if request.approver is null or request.approver ==
request.requester:
        raise SelfApprovalNotAllowed
    if request.target not in isolatedTargets:
        raise TargetNotIsolated
    if request.ends is null or request.ends > now() + maxRestoreLife:
        raise RestoreNeedsEnd

    // the register comes first: a copy that exists with no record is
a side door
    registerCopy({ name: request.target, kind: "backup", source:
request.backup_id,
                   contains: "Restricted", readers: "none until access
applied",
                   owner: request.owner_team, ends: request.ends })

    try:
        auditLogAppend({ actor: request.approver, action: "restore",
                         subject: request.backup_id, target:
request.target })
    on AuditLogUnavailable:
        raise RestoreRefused

    // set before the copy exists, so a failure part-way still ends;
if this
    // call fails, the sweeper on `ends` removes the copy all the same
    scheduleDeletion(request.target, at: request.ends)
    // reachable only by the restore tooling until the last step
    startRestore(request.backup_id, into: request.target, reachable:
false)
    // every login returns with its backup-day credential, the
superuser and
    // ledgerly_access_admin included: nothing may connect before this
is done
    disableAllLogins(on: request.target)
    rotateSuperuserCredential(on: request.target)
    // the backup already holds a login with this name: drop it and
create it
    // again with a new credential, so the backup-day one never works
    replaceLogin("ledgerly_access_admin", on: request.target,
credential: "new")
    applyAccessDefinition(on: request.target)
    openTarget(request.target, readers: request.readers)
```

如果流程在 `registerCopy` 之后、恢复开始之前停止，登记册里就有一条针对并不存在的拷贝的记录，每周的检查会发现并移除它。如果流程在恢复开始之后停止，则拷贝存在但无法触及，由它的定时删除将其移除；即使定时设置本身也失败了，根据登记册中 `ends` 行事的清扫器也会找到它。反过来的顺序，会留下一份存在却未列出的拷贝，而这才是要紧的故障。这种设计的代价是延迟：在任何人看到一行数据之前，要先有审批和

隔离的目标。事故期间，这种延迟是真实的，这正是恢复路径应当预先演练、事先安排好审批的原因，第 15 章在讨论事故时会谈到。如果目标是恢复几行数据而不是整个数据库，恢复仍然进入同一个隔离的目标，从中取出这些行，再按第 10 章的规则，作为数据修正放回生产环境。

恢复出的副本起初是关闭的

恢复出的数据库会带回备份当天的角色、成员关系和凭据材料。在重新应用访问定义之前，我会把它视为敌对环境。实际做法是：恢复目标没有来自各服务的网络路由，也没有人员登录，第一步是停用备份带回的每一个登录，包括 superuser 和 `ledgerly_access_admin`，因为每个登录返回时带着的，都是备份当天的凭据。然后轮换 superuser 凭据，随后才应用第 3 章的访问定义：按代码当前所描述的角色、成员关系和授予。在此之后，副本才可触及，在此之前不可。

这一步需要一个访问管理员，并且必须是属于目标的那一个。备份里已经有一个名为 `ledgerly_access_admin` 的登录，它带着备份创建当天的凭据回来了。恢复要替换它：把返回的登录删除，以同一个名称重新创建，换上新凭据，该凭据专为本次恢复生成，并随之一起移除。生产环境中该角色的当前凭据绝不会拷贝到目标中，而备份里的那一个，在任何东西能够连接之前就已消失，因为无论持有哪一个，都能在其所管理的角色范围内授予任何东西，而恢复演练恰恰是凭据被粘贴进测试环境的地方。如果恢复出的副本自己的管理员被攻破，影响范围也就是这份副本。

只有在此之后，恢复出的副本才会有读取者，读取者就是恢复申请上列出的人。用于演练的恢复根本不需要人员读取者：演练检查数据能否起来、计数是否吻合，然后删除副本。

保留与删除

保留是在两种成本之间的权衡。备份保留得更久，就能从发现得晚的问题中恢复，代价是包含一切的数据堆更大，其中包括主库早已删除的客户数据。保留得更短，持有的就更少，而有些损坏会比最老的备份还要早。我不知道某个具体组织适合什么期限。这取决于一个悄无声息的问题通常要多久才被发现，丢失一个月的数据要付出多少，以及组织自己的法务和合规人员对已删除却仍存活于备份中的数据怎么说。GDPR 或 KVKK 这类隐私制度，往往会问一个人的数据从运行系统中删除之后会怎样，而解释权属于他们，不属于本书。工程能提供的是证据，而非合规：一份登记册和一份删除记录，显示系统做了什么。

有一种设计值得与他们讨论：备份按计划到期，恢复流程则重新应用自备份创建以来记录的删除，这样恢复出的副本不会把被有意移除的数据带回来。这会给每次恢复增加一步，并且要求删除被记录在恢复时可以查阅的地方，这是把删除请求保存在持久记录中的又一个理由。这一步在应用访问定义之后、副本开放之前运行。

```
function reapplyDeletions(target, backupTakenAt):
    // by execution time: a deletion recorded before the backup but
run after
    // it is not in the backup's state, so it must be applied again
    for each request in deletionRecord.executedSince(backupTakenAt):
        auditLogAppend({ actor: "restore-tooling", action:
"reapply_deletion",
                         subject: request.id, target: target })   //
record first
        deleteRows(on: target, subject: request.subject_id)
    // a failure here leaves the copy closed: openTarget has not run
yet
```

演练会产生副本

从未恢复过的备份，没有被证明可用。恢复演练是了解备份是否可用的办法，而每一次演练都会创建一个包含 Restricted 数据的数据库。我会让演练走与真实恢复相同的请求路径，使用同样的隔离目标、同样的登记册条目以及当天之内的终止日期。演练同时也是对事故的预演：如果它需要走捷径才能完成，那么这条捷径会在凌晨三点被人使用。

> **警告**：包含 Restricted 列的备份或转储，若放在一个范围很广的群体可读的存储中，就等同于对这些列的一份长期授予，既没有 gateway，也没有日志。发现这种情况时，应当先限制存储权限，再讨论保留期限。

转储、导出与 CSV 文件

备份是有明确目的、有团队负责的门。转储和导出则通常是某个人在时间压力下打开的门，它们有两类，处理方式应当不同：按计划发生的导出，和由人做出的导出。

手工制作的转储

Ledgerly 的共享文件夹里有几份较早的转储。每一份都是因为有人需要一小时的拷贝而创建的，拷贝却留了下来。个人的转储是一份没有保留期限、没有加密政策、也没有负责人的备份。在转储通过数据库读取的引擎上，它还是证据，表明有人持有能读取每一张表的登录。

我会做三件事。第一，把转储移出常规工具的范围：access gateway 不提供整库转储，人员角色读不了转储所需的表，而 `human_breakglass` 是唯一可能读取的人员角色，并受第 8 章 break-glass 日志集的约束。第二，找出现有的转储，并且在删除任何一份之前，先检查它是否是某个人唯一的备份。这种事是有的：转储成了某张无人另行拷贝的表的备份。应先构建真正的备份，再删除转储。第三，给需要转储的人一个能完成工作的替代方案，通常是脱敏的分析副本或一套测试数据集（参见下面关于测试数据的一节）。

人员导出

人所做的导出最为常见，也最难禁止。客服需要某位客户的发票清单。财务需要上个月付款的文件。数据团队需要一份样本。我会按文件中出现的最高数据类别对它们分层，因为一条适用于所有情形的规则，对危险的导出来说太松，对无害的导出来说又太重。

层级	内容	控制
仅 Internal	聚合值、计数、状态分布，不含个人内容	自助，有日志，无需审批
含 Confidential	带有客户联系方式或自由文本的行	带理由的工单，一位审批人，文件有终止日期；这些行来自点名各列的 JIT `SELECT` 会话（第 8 章），因此审批人读到的是文件将包含的列清单
含 Restricted	银行账户或 token 的值	不导出；答案是展示形式或应用路径（第 12 章）

层级由 gateway 计算，而不是由申请人声明，并且失败时关闭：它是查询所读取的源列中的最高目录类别（第 12 章），而不是结果列名称中的最高类别，因为别名或 `lower(contact_email) AS x` 这样的表达式会隐藏源列。经由视图、CTE 或函数而无法解析来源的列，按可导出的最高层级计，而连这一点也无法判定时，导出被拒绝。静态分析的局限见第 9 章。这些层级是对一种形态的示意。要紧的是审批与风险挂钩。每次都在

几秒内对所有东西批准的审批，就是第 8 章所描述的那种形式主义。需要人工审批人的那一层级应当足够小，使审批人确实会阅读申请，我会用有没有人说过“不”来衡量这一点。

终止日期是团队最常跳过的部分。没有终止日期的导出，是一份活得比它的问题更久的拷贝。批准该文件的工单带有一个日期，文件放到一个有负责人的地方，由清扫器将其移除。第 10 章关于临时工作的规则涵盖最轻的情形：粘贴标识符和计数，而不是带有个人内容的行。第 7 章讲了 gateway 对批量结果集能做什么：限制它并使其可见，而不是阻止它。

供应商导出

外部会计供应商从不连接 Ledgerly 的任何数据库。它接收文件，这使导出步骤成为一扇可以完整设计的门，因为直到供应商的收件箱为止，路径的两端都在掌控之中。在起始状态下，`reporting-job` 的导出步骤用共享 admin 登录写出这些文件，所以文件可以包含 admin 角色能读取的任何内容。在目标状态下，它使用 `app_vendor_export`，这改变了文件能包含的内容，因为文件所含的内容不可能超出写它的登录所能读取的范围。

```
// illustrative column lists: the vendor's real needs decide them
GRANT SELECT (id, customer_id, total_minor, currency, status, due_on,
created_at)
    ON invoices TO app_vendor_export;
GRANT SELECT (id, amount_minor, currency, account_code, created_at)
    ON ledger_entries TO app_vendor_export;
```

起控制作用的是授予，而不是查询。导出查询中的错误可能漏掉行或重复行，却无法加入 `contact_email`，因为该角色读不到它。`customer_id` 是不透明的标识符，供应商没有可以与之关联的表。如果供应商的工作需要客户姓名，那是某个人做出并批准的决定，不会因为一条查询被放宽而自动出现。

导出步骤按第 6 章的连接策略所述，连接到 `ledgerly_replica_2`，并写出一个文件。随后文件经由与供应商约定的渠道送达供应商。这些步骤涉及三个独立的系统，所以下面的代码块展示顺序，以及每一种失败会留下什么。

```
function runVendorExport(period):
    file = writeExportFile(period, as: "app_vendor_export")  // local,
not yet shared
    manifest = { file: file.name, rows: file.rowCount, checksum:
file.checksum,
                 columns: file.columnNames, period: period }

    // the record exists before the file leaves: a sent file with no
    // record is the failure to avoid
    registerCopy({ name: file.name, kind: "vendor_file", source:
"ledgerly_replica_2",
                   contains: "Internal", readers: "accounting vendor",
                   owner: "platform", ends: now() + fileLife })

    try:
        sendToVendor(file, manifest)
    on SendFailed:
        // nothing has left; the file stays local and is removed with
its end date
        notify("platform", "vendor export not delivered: " +
file.name)
        raise ExportNotDelivered

    // the file has already left: a log outage delays the entry, never
the export
    auditOrQueue(manifest)
```

这里的 `auditOrQueue` 把记录追加到审计日志，日志不可达时，则将其排队等待重试并通知平台团队，做法同第 7 章。日志故障变成一个可见的、会重试的缺口，而不是无声的缺口，并且没有任何事务横跨日志与供应商渠道。

manifest 是人们最容易漏掉的部分。它让双方都能核对该文件就是原本要发的那一份，也给访问评审提供了可读的内容：这么多行，这些列，这个期间。如果某一列出现在 manifest 中，而上个月的没有，那就是一项本该有人批准的变更。

还有两项局限。文件一旦到达供应商，其保留就是合同和信任问题：可以要求供应商确认已删除，却无法核实。校验和（checksum）只能表明文件在传输中没有被改动，不能表明供应商的员工是唯一读过它的人。对于这两点，我会在协议中写入保留条款，并在访问评审中加一行，问供应商上一次确认删除是在什么时候。

测试数据：staging、脱敏副本与合成数据

Ledgerly 的故事就是从 staging 开始的，它是最常因好意而被打开的门。工程师想要真实的数据，而生产是唯一包含真实客户所造成的各种形态的数据来源。代价是 staging 拥有测试系统的访问模型，却装着生产系统的数据。

三种非生产数据

种类	真实度	暴露面	成本
生产拷贝	最高	与生产相同，控制却更弱	创建便宜，持有昂贵
脱敏副本	高，丢失一些形态	较低，但不为零	需要构建并维护脱敏流水线（第 12 章）
合成数据	较低，仅限于已建模的内容	最低	需要维护生成器，而需要未建模形态的缺陷无从复现

我不会把未脱敏的生产数据拷贝进 staging，理由并不抽象。staging 通常比生产有更多读取者、一个共享登录、保存日志的 CI 以及被配置成与生产行为一致的集成。它把最大的受众与最弱的身份控制结合在一起，却装着最敏感的数据。如果数据必须是真实的，控制措施就必须与生产一样强，到那时，staging 不过是换了名字的生产环境。

脱敏副本仍然是什么

第 12 章讲过，脱敏是替换或模糊值，同时保持形态可用。脱敏副本远好于原始副本，但它仍然是一份数据拷贝，下面几个原因值得说明，免得有人把“已脱敏”当成“可以随便发出去”的同义词。

第一，脱敏只覆盖已列出的列。`invoice_lines.description` 这样的自由文本，可能包含分类时没有预料到的个人数据，这正是本书按最坏情况对它分类的原因，而跳过它的脱敏会把它留在副本里。第二，保持形态的脱敏会让剩下的行仍可区分：不寻常的发票金额、不寻常的日期和一个小客户，合起来可以指向某一家企业。值是脱敏了，组合起来仍携带信息。第三，一致性脱敏，即相同输入得到相同输出，以便关联仍然可用，同样也使同一个人可以跨副本和跨表被识别出来。

这一切并不意味着脱敏不值得做。它的意思是：脱敏副本在有人评审过剩下的内容之前，按 Confidential 处理，其受众比“工程部所有人”要小，并且像任何其他副本一样有终止日期；持续运行的分析副本是例外，改为

在其 `review_due` 日期上评审。对此的说法是**重新识别**：根据留下的字段推断出一条记录说的是谁。我会用这样的措辞向团队描述风险，而不是用攻击的语言。设计上的对策是缩小受众并设置到期，而不是列出更长的脱敏字段清单。

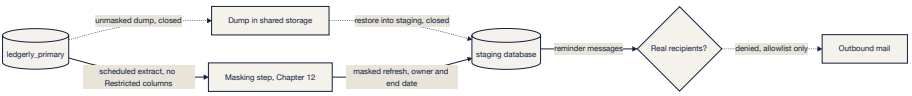
合成数据

合成数据不含任何真实的人，所以暴露面最小。它的局限是只包含有人想到去建模的形态，而依赖真实数据的缺陷，恰恰是没人建过模的那些。生成整齐客户的生成器，生成不了有六个联系人、描述用两种语言写成的那位客户。

我会让合成数据集从缺陷中生长。每当一个问题需要生产形态的数据才能复现，导致它的形态（而不是具体取值）就成为一个 fixture：一位有多个联系人的客户、一条描述为空的发票行、一个落在月底的到期日。一年下来，这套数据不再整齐，而变成有用的那种古怪。每次这样的缺陷之后，工程师要花一小时，换来的是不再需要下一份生产拷贝。

重做 Ledgerly 的 staging

回到 Robin 的周五。打开这扇门的路径，是把共享存储中的一份转储恢复进 staging。我会构建的路径用另一种刷新取代它：刷新经过脱敏步骤，落入一个寿命短、有明确负责人的数据库。在图中，请看虚线路径：它们是设计所关闭的路径。



设计的右半部分很容易被忘记。staging 环境既保存数据，也对数据采取行动。staging 中的 `invoice-worker` 读取客户并发送提醒，如果它的出站设置与生产相同，就会把提醒发往数据中的联系地址。脱敏副本里没有真实地址，这在无意中解决了问题。原始副本则会把一次测试运行变成发给真实客户的消息。我会让每个非生产环境的出站路径默认拒绝，只允许一份内部测试地址的允许列表，无论数据里有什么。

支付 token 遵循一条相关的规则。token 代表一种支付工具，它在签发它的系统之外是否有效，取决于支付提供方，我无法替该提供方作判断。默认做法是：`payment_tokens.token_value`、`bank_accounts.iban` 和

`bank_accounts.account_holder_name` 绝不出现在任何非生产副本中，无论是否脱敏（备份是例外，因为它必须包含一切），而 staging 使用提供方的测试 token。

> **注意**：应当把 staging 刷新当作有失败路径的流水线，而不是一次性的操作。如果脱敏步骤失败，刷新就停止，staging 保留之前的内容。若刷新在某一次运行中退回到未脱敏的来源，就等于换了几个步骤把共享文件夹里的转储又造了出来。

BI 工具、笔记本电脑与供应商

还有三扇门值得简短谈谈，因为每一扇都是通过看起来像一项功能的途径把数据带出数据库的。

BI 工具。仪表盘工具以一个登录连接，每个能访问仪表盘的人都能看到该登录所能读取的内容。这颠倒了第 3 章的身份模型：数据库看到的是一个 principal，而比如说一百个人看到它的数据。有两项控制会有帮助。把工具指向 `ledgerly_analytics` 而不是某个副本，让它读取脱敏数据，并使用专属于该工具的角色，即与其他任何工作负载一样的 `app_` 角色（`human_analytics_reader` 是数据团队中供人使用的角色），而不是生产角色。并且在工具支持的情况下，让它把每个人的身份传给数据库，这样行规则和列规则就按个人生效。工具还会自己保留提取、缓存和定时的电子邮件报表，这些都是数据拷贝，要进入登记册。仪表盘是一扇方便的门，而缓存的提取是同一扇门，只是控制措施没了。

笔记本电脑。gateway 去除了保存的凭据并设置只读默认值，如第 7 章所述，但它无法控制人如何处理已经到达屏幕上的行。切实可行的控制是减少保留本地拷贝的理由：一份脱敏样本、一个行为与生产相仿的 staging 数据库，以及针对引发缺陷的各种形态的 fixture。团队无法证明没有任何一台笔记本电脑持有生产数据，我也不会声称能证明。全盘加密以及能清除丢失设备的办法这类基线，可以降低丢失笔记本电脑的代价，但它是基线，不是答案。

其他供应商。会计是本书一路跟随的那一家，其他的则是顺带收到数据：捕获请求体的错误追踪器、接收粘贴截图的客服台、存储语句的日志平台。每一家持有拷贝的供应商，都在登记册里有一行，写明它包含什么以及合同何时结束。第 12 章讲了如何让敏感值不进入日志和错误，这是让它们不进入供应商存储的最便宜办法。

关闭 Ledgerly 的各扇门

设想 Alex 如何把 Ledgerly 从零散的拷贝带到一份有负责人的登记册。与第 6 章的读取路径一样，顺序很重要，我会分六步来做。

1. **盘点拷贝**。列出备份、快照、转储、导出、staging、BI 提取和供应商文件，写明存储位置、读取者和负责人。预计会发现没人记得的拷贝；登记册的第一轮主要是发现。
2. **先限制存储**。对于任何包含 Restricted 列的东西，先收窄谁能读取存储，再决定删除什么。这很快，也可以撤回，并在其余事项理清之前关闭最广的暴露面。
3. **构建备份权力**。给备份作业一个只写身份，把恢复放在需要第二位审批人的申请之后，把加密密钥与存储管理员分开持有，并通过请求路径跑一次演练。
4. **淘汰转储**。真正的备份存在、并且有一次恢复成功之后，删除旧转储，并记录每一份是谁删的。为 `ledgerly_analytics` 供数的每晚未脱敏恢复，在第 12 章的脱敏加载（由 `app_analytics_load` 运行）取而代之时结束；在此之前，它是最大的敞开之门，要给它一个终止日期。
5. **把 staging 迁移到脱敏或合成数据**，默认关闭出站消息，并移除共享的 staging 登录。
6. **让供应商导出使用自己的登录**。把导出步骤迁移到 `app_vendor_export`，加上 manifest，并设定文件寿命。

然后把登记册接到本来就会发生的事情上。第 14 章的访问评审询问存在哪些拷贝、每份谁能读取，创建拷贝需要一条登记册记录，清扫器会在宽限期并通知负责人之后，移除任何超过其 `ends` 的东西。恢复请求和导出也应当写入审计日志，那是第 14 章的主题。恢复作业和导出作业遵循与承载获批写入的 runner 相同的执行纪律：它们以自己的身份运行，来自获批的申请，并先写下记录；唯一的例外是 Internal 层级的导出，保留上述分级表中的自助规则（该纪律见第 9 章）。

权衡与成本

第一项成本是事故期间的速度。需要审批人和隔离目标的恢复，比由持有存储桶权限的任何人完成的恢复要慢，增加的时间落在最糟糕时刻的值班人员身上。应当演练这条路径并事先安排好审批（事故期间的访问见第 15 章），使控制成为人们熟悉的流程，而不是临时撞上的障碍。

第二项成本是密钥保管。把备份密钥与存储分开，意味着密钥丢失就是备份丢失，而密钥的恢复路径又是一项必须编写并测试的流程。对于很小的团队，这种拆分付出的可能多于回报；一个加密的存储、严格的存储权限和有文档的密钥恢复，可以是合理的起点。

第三项成本是更慢的调试循环。当工程师不能把生产恢复进 staging 时，有些缺陷需要更长时间才能复现，因为触发它们的形态还不在测试数据里。fixture 的习惯会随时间回本，但在开始时是一笔实实在在的开销。脱敏副本增加了一条有人负责的流水线，每当添加一列它就会出问题，这就是为什么添加列的迁移是决定其类别的地方（第 11 章）。

第四项成本是保留。更长的保留意味着更多历史可恢复，也意味着更多敏感数据存在于更多地方；更短的保留则两者相反。没有对所有人都正确的数值，而没有选定过数值的团队，选的就是“永远”。

最后，登记册及其清扫器是额外开销。清扫器删除了某人仍然需要的拷贝，本身就是一起事故，所以删除应当发生在通知和宽限期之后，负责人应当能够附上理由，把终止日期延长一次。从不被打开的登记册什么也保护不了，而与访问评审的联系，才是让它保持诚实的东西。

何时不必全部构建：只有一个数据库、寥寥几位工程师、没有值得一提的 Confidential 数据的小公司，不需要分层的导出审批，也不需要拆分的密钥。它仍然需要盘点清单、对备份的受限存储以及一条规则：生产中的任何东西都不得未经脱敏就进入 staging。做好这三件事，写下什么情况会促使做得更多，并在团队或数据变化时重新审视这份笔记。

常见错误

- **收窄了对主库的访问，却忽略它的备份**。角色、gateway 和审计日志都已到位，而备份存储桶对整个基础设施组可读。这很诱人，因为备份属于“运维”。补救办法是把恢复视为对一切的读取，配有自己的审批人和自己的日志。
- **相信“已加密”就是“安全”**。静态加密保护文件不被恢复路径之外的人读到，而密钥与存储桶放在一起，在同一位管理员手里。应把密钥分开持有，并说明它不能防范什么。
- **在真正的备份存在之前删除旧转储**。那份转储是某个人对某张表唯一的拷贝，删掉它也就删掉了唯一的退路。应先构建并测试备份，再淘汰转储。

- **恢复到旧登录仍然有效的网络中**。恢复出的副本带回它那天的角色和凭据，包括此后已经停用的。应恢复到隔离环境，重新应用访问定义，并把备份带回的 `ledgerly_access_admin` 登录替换为新的，而不是保留任何一份旧凭据。
- **把脱敏副本当作公开的**。值已经脱敏，副本被广泛共享，随后自由文本或某个罕见的字段组合指向了某一位客户。应当按 Confidential 处理，缩小受众并设置终止日期，直到评审完成。
- **把从生产刷新 staging 当作一次性的例外**。脱敏步骤失败了，发布在等着，转储就在手边。应让刷新失败时关闭，使旧数据保持原样，也没有任何东西退回到原始来源。
- **让 BI 工具用生产角色连接**。一个登录背后有，比如说，一百位查看者，还有一份无人列出的缓存。应让它用自己的角色连接到分析副本，并登记它的提取。
- **几秒钟就批准每一次导出**。审批存在，却什么也没减少。应把审批人留给值得做决定的那一层级，并观察是否有人曾经拒绝。
- **给拷贝指定了负责人，却不设终点**。登记册是完整的，却没有任何东西离开它。每一行都需要一个 `ends` 日期，以及一个据此行事的清扫器，并先发通知。

要点

- 旁门携带的数据不再带有其来源系统的控制措施，因为访问控制存在于数据库中，而不在文件里。
- 恢复备份的权力就是读取其中一切的权力，所以备份访问应当得到与权限最高的人员角色相同的评审。
- 创建、恢复和删除备份应当由不同的 principal 持有，加密密钥应当与存储管理员分开持有，并有经过测试的恢复路径。
- 恢复出的副本带回的是创建当天的角色和凭据，所以它起初处于隔离状态，重新应用访问定义，并获得自己的管理员登录。
- 供应商导出由 `app_vendor_export` 的授予控制，而不是由查询控制，因此文件不可能包含该角色读不了的列，而 manifest 让文件可以被核对。
- 测试数据应当是脱敏或合成的，脱敏副本在评审之前仍按 Confidential 处理，staging 应当默认拒绝出站消息，并在脱敏失败时关闭。

- 每份拷贝都需要一个负责人，以及登记册中的一个终止日期，该登记册与访问一起评审，因为没人负责的门，也没人去关。

拷贝、导出和恢复现在都有了负责人和终止日期，但设计的好坏，取决于是否有证据表明它被遵守。下一章转向审计：审计日志放在哪里，其中记录什么，以及本书一再提到的访问评审实际上如何进行。

第五部分

验证与运维

第 14 章 — 审计与问责

设想 Ledgerly 的一个周二上午。一位客户询问，上个月公司里有谁查看过其付款账户，安全联系人把这个问题转给了 Alex。听上去只是一次查询。Alex 先查看 access gateway 的会话列表，其中显示 Robin 在主库上持有过一个时长 40 分钟的 break-glass 会话。接着 Alex 打开数据库日志，里面写着会话登录，却没有写会话的原因。然后 Alex 搜索 `audit_events`，它记录的是应用执行的业务操作，对某个人输入过什么只字未提。

这三个来源在大部分情节上吻合，在其余部分上相互矛盾。gateway 的时钟比数据库的快 2 分钟。数据库日志里有一个会话登录，gateway 里却完全没有对应条目，没有人能说清，这是列表遗漏的某个服务登录，还是经由一条无人设计的路径到达的连接。Alex 能说出大概发生了什么，却无法证明，也无法证明没有拥有数据库管理员权限的人改动过证据。

本章服务于论点中的问责支柱，并补上事故之前目标状态的最后一步：访问记录写入被访问系统无法改写的存储，并按计划定期评审。在五部分模型（**身份、授予、路径、执行、记录**）中，本章讲的是记录。身份说明谁在行动；记录则是留下来的东西，以及它能否把语句与人、申请和结果联系起来。本章要回答的问题是：一条语句应当留下什么，这些内容应当存放在哪里，以及如何在有人发问之前，就发现证据并不完整？

问题及其难点

在初始状态下，这个问题无从回答。数据库日志里写的是共享角色，而不是具体的人，日志还存放在管理员控制的主机上。前面各章解决了身份这一半：每个会话一个登录，每项授予对应一份申请，已批准的写入由 runner 执行。它们都通过同一个调用 `auditLogAppend` 追加记录，却没有说明条目最终去往何处。本章回答这个问题。

难点在于，Ledgerly 已经有三份记录，各自了解的内容并不相同。它们都不是本书所定义的审计日志，即记录谁对哪些数据做了什么、并存放在行为人无法更改之处的那份记录。

记录	它知道什么	它无法说明什么	谁能修改它
gateway 会话记录	人、申请、原因以及提交时的语句	数据库实际做了什么；是否存在另一条路径	平台团队，经由 gateway 自己的存储
数据库日志	登录、时间、记录了的话还有语句文本、错误	登录背后的人、申请、原因	任何管理数据库主机的人
`audit_events`	业务操作、行为的 principal、对象	产生这一操作的语句；应用没有写入的任何内容	通过 `UPDATE` 无人能改，但 `ledgerly_owner` 可以在迁移中修改该表

显而易见的应对是调高日志级别：在数据库中记录每一条语句，并保留很长时间。这样只是多了第四份记录，三处缺口一处也没有补上。被访问系统记录自身行为的日志，其可信度至多等同于它的管理员；而没人阅读的日志只是剧场。语句日志本身就是敏感数据的一份副本，因为人们会把字面值写进条件里；第 12 章已就此发出警告，而本章负责给出机制。

记录的存在，是为了找出系统中的缺口，而不是给经手的人排名。当告警在 Robin 的会话上触发时，首先要问的是设计迫使他做了什么，而不是 Robin 做错了什么。

一条语句，一条记录

记录只有在能回答一份事先写好的问题清单时才有用。我会先写出这份清单，再由它推导出各个字段，因为按 gateway 碰巧产生的日志行来设计的审计存储，只能回答那些日志行碰巧允许回答的问题。

gateway 为每条语句写入两个条目，因为第 9 章要求在语句运行之前写入意图，在运行之后写入结果。两者合起来构成**语句记录**；这两个条目共用一个 `statement_id`，第二个条目填入起初未知的字段。同一种结构既用于这两个条目，也用于其他 principal 写入的记录。

```
table access_audit_record:
    stream          text not null      // one per producer instance
    sequence        integer not null   // gapless within a stream
    previous_hash   text not null      // hash of the previous entry
    entry_hash      text not null
    kind            text not null      // "intent", "outcome", "event"
    action          text null          // events only: grant, revoke,
breakglass, data-fix, contain, and so on
    statement_id    text null
    session_id      text null
    login           text not null      // what the database log shows
    principal       text not null      // the person or workload
identity
    on_behalf_of    text null          // the human behind an
automation principal
    roles           text not null      // role set at the time of the
statement
    request_id      text null          // always access_request.id; a
data fix has one too, and its fix_id resolves through it
    reason          text null
    approver        text null
    statement_class text null          // the Chapter 9 class the
classifier assigned
    fingerprint     text null          // statement shape, literals
redacted
    literals_hash   text null          // keyed hash of the literal
values
    tables          text null          // tables touched
    columns         text null          // columns touched, by data
class
    rows_affected   integer null       // outcome only
    rows_returned   integer null       // outcome only
    outcome         text null          // ok, refused, error, rolled-
back, row-cap-closed
    started_at      timestamp not null
    ended_at        timestamp null
```

`event` 类型承载所有不是语句的内容：会话开启、授予、撤销、轮换、恢复、迁移开始。它们就是第 3、4、7、8、10、11、12、13 和 15 章中的 `auditLogAppend` 与 `auditOrQueue` 调用，由 `action` 列区分授予、撤销或遏制。它们使用同一条链和同样的身份字段，因此一次查询就能沿着时间线追踪一个人：从申请到授予，再到使用该授予的语句，直至撤销。

只有当这张表能回答人们会问的问题时，才值得建立。第二份清单是我会贴在 schema 旁边的那一份。

问题	回答它的字段
谁运行了这条语句？	`principal`、`on_behalf_of`、`login`

问题	回答它的字段
凭借哪些授予和审批?	`roles`、`request_id`、`approver`、`reason`
它请求了什么?	`statement_class`、`fingerprint`、`literals_hash`、`tables`、`columns`
它做了什么?	`outcome`、`rows_affected`、`rows_returned`
它是否触及 Restricted 数据?	按数据类别分类的 `columns`，连同 `rows_returned`
何时开始，持续多久?	`started_at`、`ended_at`
周围是否缺失了什么?	`stream`、`sequence`、`previous_hash` 以及没有结果的意图
这个条目与写入时一致吗?	`entry_hash`，由下文的 `verifyChain` 检查

有两个字段承担了大部分分量。`principal` 与 `on_behalf_of` 使轨迹不会止于一台机器，这正是第 9 章对自动化的要求。`request_id` 是关联第 8 章和第 10 章所记录全部内容的连接键。

没有 `request_id` 的记录依然有效。例行的 `human_readonly` 会话没有申请，其记录通过把该字段留空来说明这一点。持有 `human_jit_write` 三元组或 `human_breakglass` 的会话若 `request_id` 为空，则属于一个发现项，本章后面的检查会查找这种情况。

被访问系统无法改写的存储

条目存放在哪里，比它们的格式更重要。第 1 章把可观测性定义为存放在 principal 无法更改之处的记录。这里重要的 principal 是权力最大的那些：数据库管理员、运行 gateway 的平台团队以及 `ledgerly_access_admin`，它的三个登录几乎可以在所管理的角色内授予任何权限。如果其中任何一方能编辑记录，那么这份记录只能证明他们选择留下的东西。

我会按如下顺序，以五项属性构建这个存储。

1. **独立的账户**。审计存储位于云账户、项目或主机组之中，数据库管理员、gateway 的运维人员以及三个 `ledgerly_access_admin` 登录在其中都没有任何权限。总得有人来管理它；这是一个不持有数据库管理员权限的小团体，而且它自己对存储的使用，要能被团体之外的人看到。

2. **只写的生产者**。每个生产者（gateway、runner、凭据 broker、流水线、撤销作业、恢复工具）各持一份自己的凭据，只能调用一个操作，即 append。它不能读取、更新或删除。泄露的生产者凭据可以加入虚假条目，而链使这一点可以被发现，但它无法删除真实条目。
3. **持有链的 append service**。生产者不计算哈希。审计账户内的一个小型服务加载流的 head，把新条目链接上去，并将两者一并存储。自己计算链接的生产者，同样可以重新计算其后的一切。
4. **账户管理员无法缩短的保留期**。存储提供一次写入的保留锁时，在保留期内使用它，这样账户内的任何人都无法在期满前删除条目。没有这种锁时，诚实的说法是管理员可以删除，而使删除变得可见的，是链和下面的锚点。
5. **锚点**。append service 每天把各个流的 head 哈希发送到第二个地方，该地方由另一个团体拥有：安全联系人的系统，或第二个存储。没有锚点，完全控制存储的人可以从某一点起改写整个流，并重新计算其后的每个哈希。有了锚点，改写后的 head 就会与锚点不一致。

```
function appendChained(stream, record):
    // runs inside the audit account; producers can call only this
    // one writer per stream at a time: the head is read and locked
    // inside the same transaction that stores the next entry
    transaction:
        head = lockHead(stream)
        entry = { stream: stream, sequence: head.sequence + 1,
                  previous_hash: head.hash, body: record,
                  stored_at: now() }
        entry.entry_hash = hashOf(entry.stream, entry.sequence,
                                  entry.previous_hash, entry.body)
        // one system, so one transaction is allowed: the entry and
the
        // new head exist together or not at all
        storeEntry(entry)
        storeHead(stream, entry.sequence, entry.entry_hash)
    // commit point: the entry is durable and the chain has moved
    return { sequence: entry.sequence, hash: entry.entry_hash }
```

两个会话可能在同一时刻向同一个流追加，所以 head 在事务内加锁读取；没有这把锁，两者会读到同一个 head，写下相同的序列号，使链产生分叉，验证器随后便会报告一次从未发生的篡改。规则是同一时刻每个流只有一个写入方，由这把锁来保证。

验证器是一个独立的作业，在同一账户内持有只读凭据。把存储变成证据的，正是它的工作：

```
function verifyChain(stream, from, to):
    previous = loadEntryBefore(stream, from)
    for each entry in entries(stream, from, to):
        if entry.sequence != previous.sequence + 1:
            notify("security", "hole in " + stream + " at " +
entry.sequence)
        if entry.previous_hash != previous.entry_hash:
            notify("security", "broken link in " + stream)
        if entry.entry_hash != hashOf(entry.stream, entry.sequence,
                                      entry.previous_hash, entry.body):
            notify("security", "entry altered in " + stream)
        previous = entry
    // compare at the anchor's own position, not at the end of the
range
    anchor = loadAnchor(stream, atOrBefore: to)
    if anchor is not null and anchor.hash != entryAt(stream,
anchor.sequence).entry_hash:
        notify("security", "head differs from anchor in " + stream)
```

现在要说诚实的局限，我会把它作为单独一句写进设计文档。这一设计提供的是证据，而不是预防。它不会阻止语句运行，也不会阻止对审计账户拥有足够权力的人销毁存储。它使篡改变得可被发现，并使有能力篡改的人成为一份简短而独立的名单。检测还取决于有人阅读 `verifyChain` 的输出，因此验证器是一个带有自身心跳的定时作业，下一节会谈到。

成本是实实在在的。审计账户是又一个需要保持可用的系统，而第 8 章和第 9 章已经接受了它最坏的后果：当意图无法写入时，授予、break-glass 声明或语句会被拒绝。因此审计存储的中断会阻塞特权工作，而且恰恰阻塞在可能有人正等着处理紧急情况的时刻。这就是不依赖善意的记录所要付出的代价。

> **警告**：不要把审计存储放进它所描述的数据库里，即使放在有独立所有者的 schema 中也不行。能管理该数据库的人就能改动条目，仅有 `INSERT` 权限的角色改变不了这一点。本书已列出的那个结构性例外，即 `ledgerly_owner` 可以在迁移中修改 `ledger_entries` 和 `audit_events`，正是 `audit_events` 只是应用的业务记录、而不是审计日志的原因。

找出缺口

链证明现存的条目没有被改动，却不能证明每条语句都产生了条目。第二类检查查找的是应当存在却不存在的东西。缺口有四种，每种含义不同，所以检查把它们分别报告。

序列空洞。流按构造是无间隙的，所以出现空洞意味着 append service 丢失了条目，或者有人移除了条目。这是最罕见的发现项，也是最严重的发现项。

没有结果的意图。gateway 在语句之前写入意图，在语句之后写入结果。如果在超时所允许的最长语句时间之后，意图仍没有结果，那就意味着 gateway 在两者之间失败了，或者结果正在 retry 队列中等待，或者语句仍在运行。只有第三种情况无害，而且它不应当超过超时时间。

只出现在一侧的会话。每个数据库登录都应当有已知的来源。对于人，来源是 gateway 会话。对于服务，来源是第 4 章中凭据 broker 签发的租约。对于迁移，来源是流水线的一次运行。数据库日志中的登录若这三者都不是，就是一条无人设计的路径；而 gateway 会话若声称有语句而数据库从未见过，这份记录描述的就是一件没有发生过的事。

沉默的生产者。一个停止写入的流，可能意味着平静的一周，也可能是生产者已经死亡，或者被人关掉了。检查会把每个流与预期的频率比较。

```
function detectGaps(window):
    gatewaySessions = sessionsFromAuditStore(window)
    databaseLogins = loginsInDatabaseLog(window)
    for each login in databaseLogins:
        if login not in gatewaySessions
           and not hasBrokerLease(login, window)
           and not isPipelineRun(login, window):
            notify("security", "login with no known origin: " + login)
    for each s in gatewaySessions:
        if s.statements > 0 and s.login not in databaseLogins:
            notify("platform", "gateway session with no database
login: " + s.id)
    for each i in intentsWithoutOutcome(olderThan: maxStatementAge):
        if not inRetryQueue(i):
            notify("platform", "intent with no outcome: " +
i.statement_id)
    for each stream in expectedStreams():
        if now() - lastEntry(stream) > expectedCadence(stream):
            notify("platform", "stream silent: " + stream)
```

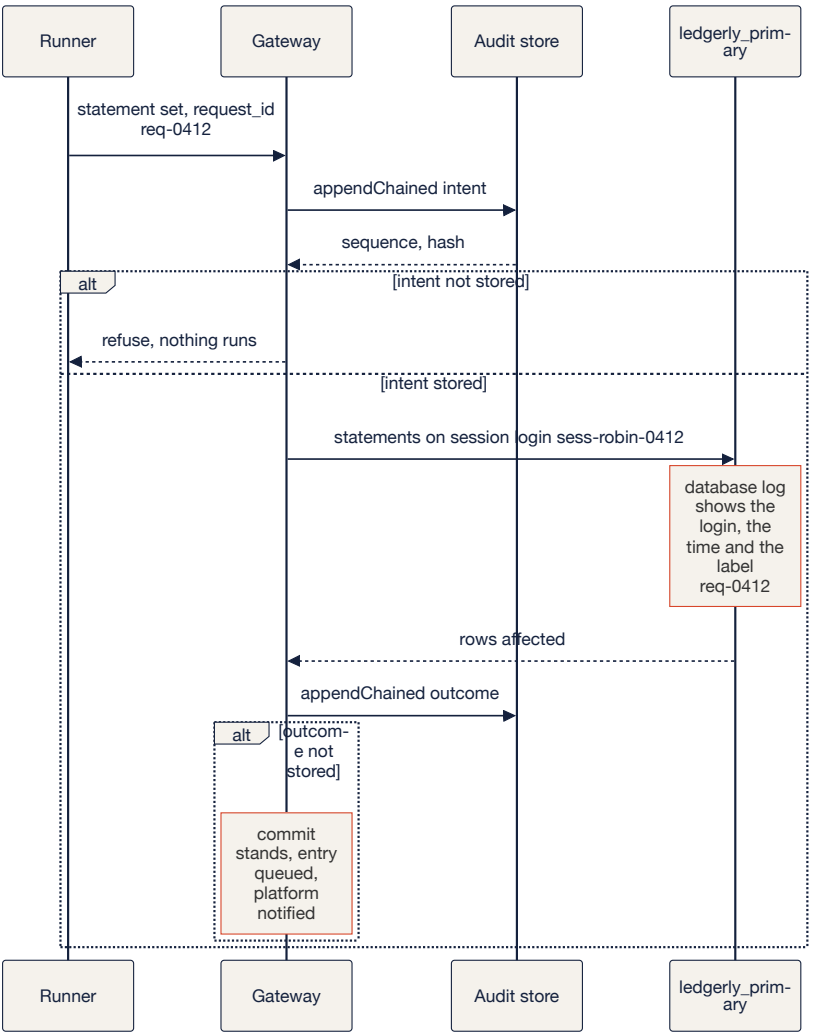
数据库日志通过一个持有自己只写凭据的传送代理到达审计账户。拥有主机访问权限的人可以停掉这个代理，沉默流检查正是为此而设：被停掉的代理会留下一个没有条目的流，这本身会引发一个发现项。

关联三类记录

gateway 记录、数据库日志和 `audit_events` 由三个从不共享事务的系统产生。把它们关联起来，才能把 Alex 那个上午的各种情形从猜测变成时间线，而且关联键必须在事故之前选定，因为事后无法再添加。

三个键承担这项工作。**登录**出现在数据库日志中，由于一个会话登录恰好对应一个人或一个 workload 实例，它把日志行与会话记录联系起来。**申请 ID** 由 gateway 写入审计记录，由 runner 写入审计存储自己的 `request_id` 字段；它把语句与允许其执行的审批联系起来。runner 从不写入 `audit_events`，第 9 章的矩阵拒绝了这一点。带有第 3 章引入的 correlation ID 的 `audit_events` 行来自产品路径，即 `admin-console` 调用 `payments-service` 之处，它们通过该 ID 关联到同一份申请。第 5 章中的**连接标签**可以把申请 ID 带入受控连接的数据库日志。这个标签由打开连接的一方设定。在受控执行中，gateway 以已批准申请的申请 ID 打开连接，所以标签是可靠的；而在 session handout 中，人的客户端可以把它设成任何值，因此登录仍然是键，标签只是提示。

图中展示了一次已批准的数据修正，以及各份记录分别写在哪里；runner 只能通过 gateway 到达主库。如果结果写入失败，提交依然有效，而缺口是可被发现的。



记录不得保存什么，以及谁来审计审计者

语句记录保存的是人们输入的内容。通过账号查找客户的人，已经把账号写进了语句，而存储该语句的日志，就把一个 Restricted 值复制到了没有列级保护的地方。第 12 章的警告在此完全适用，所以构建记录的人必须决定保留什么。

我会把语句分成两部分存储。**指纹**是把每个字面值替换为占位符后的语句，所以 `UPDATE invoices SET status = ? WHERE id IN (?)` 只记录形状，不记录别的。**字面值哈希**是对字面值的带密钥哈希，调查人员可以据此用记录来检验某个怀疑的值，而记录本身并不保存它。密钥很关键：

状态或简短的账号后缀之类的字面值熵很低，不带密钥的哈希可以通过尝试候选值被还原。密钥与审计账户放在一起，同生产者分开，调查人员的工具只请求比较，从不请求密钥。

```
function buildStatementRecord(session, statement, analysis):
    // analysis comes from the gateway's query analysis, not from the
client
    parsed = parse(statement)
    record = { kind: "intent", session_id: session.id,
               login: session.login, principal: session.principal,
               on_behalf_of: session.on_behalf_of,
               roles: session.roles, request_id: session.request_id,
               fingerprint: redactLiterals(parsed),
               literals_hash: keyedHash(auditKey, literalsOf(parsed)),
               tables: analysis.tables,
               columns: classifyColumns(analysis.columns),
               started_at: now() }
    // a statement that cannot be parsed is recorded as a refusal with
no
    // text, never stored verbatim as a fallback
    if parsed is null:
        record.fingerprint = "unparsed"
        record.literals_hash = null
    return record
```

最后几行最为重要。解析器失败时回退为存储原始语句，会保留字面值最可能出现的那种情形，因为解析器恰恰会在不寻常的语句上失败。我会什么都不存，让拒绝本身说明问题。存储本身至少归为 Confidential，因为它描述了谁查看了什么，并且有自己的读者名单和保留期。公司可以对自己的员工记录到什么程度，是需要向法务与合规人员请教的问题。

谁来审计 `ledgerly_access_admin`

`ledgerly_access_admin` 是能够创建登录、并在所管理的角色内授予任何权限的角色，通过三个登录使用：应用访问定义的流水线步骤（第 3 章）、凭据 broker（第 4 章）和 gateway（第 7 章）。第 3 章说过每项操作都会进入审计日志；机制则是这三者各是一个生产者，有自己的流，因此记录会写明是哪个登录在行动。它签发的每一条 `CREATE LOGIN`、`GRANT` 和 `REVOKE` 都是一个 `event` 条目，在变更之前写入，并附带引发它的申请或运行。

由此得到一项闭合回路的检查。数据库的系统目录是已存在授予的真相，审计存储是已发生授予的真相。系统目录中出现了没有对应条目的授予，就说明它是在三个生产者之外做出的，这正是第 3 章的漂移检查要找出

的凌晨 2 点的那项授予。这两项检查本是同一件事的两端：漂移检查把系统目录与代码比较，这一检查把系统目录与记录比较。两者都不做修复，只做报告。

迁移

迁移流水线像第 11 章要求的那样，在运行迁移之前向审计存储写入，并在迁移完成或失败时再写一次。之所以要有这条外部条目，是因为迁移工具也会把已应用的内容记录在 `ledgerly_primary` 内的一张表中，而 `ledgerly_migrator` 会承担 `ledgerly_owner`，后者可以修改那张表。行为人能够改写的记录只是一种便利，而不是证据。规则是：当外部的开始条目和提交哈希都存在于审计存储中时，迁移才算已应用；迁移表中没有对应外部条目的行，则由同样的系统目录与记录比较报告为绕过流水线的变更。

有人响应的告警

记录用于事后回答问题。其中有少数问题值得持续地问，告警就是做到这一点的形式。我对每条规则都会应用的检验是：是否指定了人来响应它，他是否知道该做什么。

Ledgerly 从三条规则开始，每一条都利用了记录所知而其他来源不知的东西。

```
function evaluateAlertRules(window):
    for each r in outcomeRecords(window):
        // 1. large reads outside the person's working hours
        if isReadClass(r.statement_class) and isHuman(r.principal)
           and (r.rows_returned > nightRowThreshold or r.outcome ==
"row-cap-closed")
           and isOutsideHours(r.started_at, r.principal):
            notify("security", "large read outside hours: " +
r.statement_id)
        // 2. break-glass in the role set without a declared request
        if "human_breakglass" in r.roles and not
hasDeclaredRequest(r):
            notify("security", "break-glass role without a
declaration: " + r.session_id)
        // 3. Restricted columns returned to a human principal
        if touchesRestricted(r.columns) and r.rows_returned > 0
           and isHuman(r.principal) and not isDeclaredBreakGlass(r):
            notify("security", "Restricted data read by a human: " +
r.statement_id)
    for each s in sessionsOpened(window):
        if "human_breakglass" in s.roles:
            notify("security", "break-glass in use: " + s.id)
```

阈值只是示意；合适的 `nightRowThreshold` 取决于什么能让每周只有少数几个事件摆到人面前。第二条规则是对第 8 章声明机制的独立检查：该章在有人声明紧急情况时告警，而这条规则找出的是持有该角色却没有声明的会话，也就是绕过了声明的授予。第三条规则几乎不应触发。`human_readonly` 无法读取 Restricted 列，JIT 申请也无法指定这类列，所以人读取 Restricted 数据，要么是已声明的 break-glass 会话，要么是与矩阵不符的授予，而后者属于设计层面的发现项。

每条规则都需要一位指定的负责人和一个首要动作：安全联系人询问任务是什么，值班的平台工程师确认声明并开出评审项，平台团队撤销被绕过的授予。被确认却没人看过的规则，就是安全剧场。

访问评审

告警盯的是事件。**访问评审**则是定期检查每个 principal 的访问是否仍然需要，它是用来发现那种没有任何事件显示的缓慢失效的控制：授予时是正确的，后来悄悄不再正确的访问。人会换团队，服务不再读某张表，角色保留着其工作负载几个月没用过的权限。第 3 章说过，错误的授予只要声明无误就能通过漂移检查，而访问评审正是做出这种判断的地方。

评审把已授予的内容与已使用的内容进行比较。第二份清单由记录提供，这使它成为一项决定，而不是一张等人签字的表格。

```
function runAccessReview(period):
    proposals = []
    for each grant in declaredGrants():
        // role to table privileges, identity provider group to role,
        // and the logins of ledgerly_access_admin
        use = lastUse(grant, inRecords: true)
        if use is null or use.at < now() - period:
            proposals.append({ grant: grant, finding: "unused",
                               proposal: "revoke" })
        if grant.owner is null or not isActivePrincipal(grant.owner):
            proposals.append({ grant: grant, finding: "no owner",
                               proposal: "assign or revoke" })
    for each r in breakGlassUses(period):
        if reviewFor(r.id) is null:
            proposals.append({ grant: r, finding: "no review",
                               proposal: "review now" })
    // the review proposes; a named reviewer decides, and is never
    // the person whose access is under review
    return assignReviewers(proposals)
```

我会按三条规则使用它的输出。第一，这个函数只提出建议，从不撤销，理由与第 3 章所说相同：自动撤销某人依赖的授予，是破坏生产的一种新方式。第二，`period` 必须长于工作负载的最长周期。季度报表所用的授予在大多数月份里都没被使用，所以一个月的评审会在三个月中有两个月建议撤销它。第三，使用情况要按权限匹配，而不仅仅按表匹配。一个每晚从某张表读取、却从未更新过它的角色，是把 `S, U` 收窄为 `S` 的候选，而记录里有所需的信息，因为每条记录都带有语句类别和所触及的表。

评审还有第二个输入，是记录无法提供的：并非授予的那些东西。第 13 章的副本登记册应当列入议程，谁可以审批 JIT 申请的名单，以及谁可以读取审计存储的名单也一样。对每一项，问题都相同：这个人或系统是否仍然需要它，是否仍有人负责它？附录中有一份模板。

是证据，而非合规

数据保护和支付卡类制度，往往会就访问提出一系列问题：谁能接触数据，谁做了什么，访问持续多久，副本在哪里，紧急情况下会怎样。在这些制度下工作的审计人员，通常想看到的是：控制措施存在，它运行过，它的结果有人看过。对任何制度的解释，都归组织自己的法务与合规人员。本节的主张不超出上述工程所能交付的范围，那就是证据：能回答问题的记录，以及证明它们未被改动的办法。

本章中没有任何内容能使一家公司在任何方面达到合规。下表列出审计人员往往会问的问题、回答每个问题的控制措施、它留下的制品以及建立它的章节；附录以对照表的形式重复了这张表。

问题	控制措施	制品	章节
谁能通过哪条路径接触生产数据?	gateway 作为唯一的门、作为代码的角色 schema、漂移检查	网络规则清单、角色定义、漂移报告	3、7
谁以哪些授予和审批运行了某条语句?	带申请 ID 和审批人的 `access_audit_record`	该记录和 `verifyChain` 报告	14
提升的访问持续多久，是谁批准的?	带审批人和 `expires_at` 的 JIT 申请、撤销作业	申请记录、撤销日志、到期授予检查	8
敏感数据如何识别和限制?	schema 中的分类、脱敏、被扣留的列	系统目录、脱敏加载验证	12
生产数据的副本在哪里，各归谁负责?	带负责人和结束日期的登记册、清扫器	登记册、清扫器日志	6、13
谁修改 schema，谁评审过?	迁移流水线、受保护分支、外部开始条目	与提交和审计条目匹配的迁移记录	11
访问是否仍然需要?	由指定评审人完成的访问评审	带决定的已完成评审	14
紧急情况下会怎样?	带告警和强制评审的 break-glass	评审记录、告警历史	8、15
谁来管理访问本身?	作为独立生产者的 `ledgerly_access_admin` 登录	它们的流，以及系统目录与记录的比较	3、14

权衡与成本

记录是有账单的，大部分花在两个地方。

存储与注意力。每条语句两条记录，随流量增长，存储还需要自己的保留期、备份和读者名单。更大的成本是注意力。没人查询的存储是代价高昂的安全剧场，因此第 7 章中每周对提升会话的抽样、告警和评审，都是代价的一部分，而不是额外之物。

关键路径上的新依赖。因为意图在语句运行之前写入，审计中断会阻塞特权工作。另一种做法是先运行语句、稍后再记录，那会使记录恰好在最重要的时候变成可有可无。我会对授予、break-glass 和写入保持拒绝；

如果团队认定读取中断的代价大于一处缺口，则可以考虑对例行的 `human_readonly` 读取采用排队写入，并限制队列允许增长多长。这是平台团队应当一次做出并写下来的决定。

什么时候不必全部构建：一家只有几名工程师、一个数据库、还没有 gateway 的公司，不应当一上来就建 append service、锚点和独立账户。先做三件事：以每个人自己的登录记录其会话，把数据库日志传送到主机之外、管理员无法从中删除的地方，并指定一个人负责每月评审。当能够改动日志的人数多到让“信任”不再是够用的回答时，再加上链和锚点。

常见错误

- **把数据库日志当作审计日志**。它是最完整的来源，却处在管理员的控制之下。应将它传送到别处，并把它作为关联的一个输入，而不是记录本身。
- **没有独立存储的哈希链**。链看起来像证明，但能写入存储的人可以重新计算它。应当把存储放在另一个账户中，让 append service 持有链，并为 head 设置锚点。
- **存储原始语句**。这很诱人，因为它让调查变得容易，但它把 Restricted 值复制进了没有列保护的存储。应当存储指纹和带密钥的哈希，并拒绝回退为原始文本。
- **没有负责人的告警**。没有人被指定响应的渠道会变成噪音。应为每条规则指定负责人和首要动作。
- **审计了一切，唯独没审计审计者**。授予访问的角色，恰恰是其行为最重要的角色。应当给它独立的生产者，并把它的工作与系统目录比较。

要点

- 审计日志是记录谁对哪些数据做了什么、并存放在行为人无法更改之处的记录，而 gateway 记录、数据库日志和 `audit_events` 是它的三个输入，不是它本身。
- 语句记录由先写下的问题清单构建，即使行为是机器做出的，其身份字段也通过 `principal` 和 `on_behalf_of` 终止于一个人。
- 防篡改证据来自独立的账户、只写的生产者、持有链的 append service 以及由他人拥有的锚点，它提供的是证据，而不是预防。

- 链只表明现存条目没有被改动，因此需要另行检查序列空洞、没有结果的意图、来源不明的数据库登录和陷入沉默的生产者；三份记录以登录和申请 ID 关联，连接标签只是提示。
- 记录绝不能变成它所描述数据的副本，所以语句存储为指纹并附带字面值的带密钥哈希，存储本身像数据一样分类并加以限制。
- 告警需要负责人和首要动作，访问评审则在长于工作负载周期的时间段内，把授予与使用情况比较，从而让撤销建议建立在证据之上。
- 证据不是合规：由问题、控制措施、制品和章节组成的对照表，展示系统能够证明什么，而这份证明在任何制度下意味着什么，归法务与合规人员判断。

记录现在已经建立、关联并受到监视。接下来要面对的，是在压力之下需要用到它的那一天，那时发问的人不再是审计人员，而是一位凭据泄露的值班工程师。下一章转向事故中的访问，并利用这里建立的记录，查明一个被入侵的 principal 实际触及了什么。

第 15 章 — 事故中的访问

设想 Ledgerly 的一个周六下午。支持团队的 Sam 在值班频道里写道：一台工作笔记本电脑从停着的车里被偷了，上面可能开着一个 gateway 会话。Robin 正在值班，把这条消息读了两遍。目前没有任何已知的问题，但“可能开着”这样的话，是必须得到回答、而不能掂量一下就算了的。

Robin 需要三件事，而且顺序很重要。第一件，是阻止该会话以及它背后的任何登录仍能做的一切。第二件，是弄清它在能操作的时候做了什么。第三件，是在前两件事进行的同时保持业务运转，这意味着其他人仍然需要使用数据库，其中有些人很急，而平时的审批人都在家里。

本章把事故中的访问当作需要设计和演练的事情来处理。它同时检验三根支柱：紧急增删授予时检验授权，在压力下执行修复时检验执行，记录必须说明发生了什么时检验问责。在五部分模型中，身份找出登录，授予与路径将其切断，执行承载紧急写入，记录衡量损害。本章补上 Ledgerly 目标状态的最后一步：事故中的访问有一条经过演练的路径，撤销成为例行动作。问题是，Ledgerly 如何在不重新打开前面各章已关闭的门的前提下，停止访问、衡量触及范围并保持运转。

问题及其难点

事故是之前每一项控制看上去都成了障碍的时刻。平静的周二里合理的审批，到了周六就成了等待；记录一切的 gateway 又成了一个可能宕机的环节；而房间里最干净利落的捷径，是一个能跳过这一切的登录。通常的应对是提前造出这样一个登录，“以备紧急情况”。这又是那个共享的管理员密码，只是换了个更好听的名字。

压力之下有三点不同。时间很短，所以需要人去回忆的步骤会被跳过。知识是片面的，所以对范围的第一次估计总会在某个方向上出错。而且行动者很特殊：很少接触数据库的人突然被要求去接触，天天接触数据库的人则已疲惫不堪。

因此设计问题是，哪些控制可以放松，哪些不可以。我会坚持一条规则：紧急程度改变速度，绝不改变身份，绝不改变所运行的语句，也绝不改变记录。等待期和审批人的顺序可以缩短。principal 以自己的登录行动、与语句哈希绑定的审批以及审计日志中的一个条目，则不可以。如果一项控制在缩短时无法不去掉这三者之一，错的是这项控制，而不是这条规则。

速度必须被构建出来。需要十几个手动步骤的撤销，会被做错顺序，或者只做一半。顺序是代码，范围查询是代码，说明事故是否留下了访问的对账也是代码。指向这些代码的 runbook 是一套流程；用判断取代这些代码的 runbook 则是一种希望。

遏制入侵

这里的**遏制**指的是停止某个 principal 的访问，无论原因是丢失的笔记本电脑、泄露的凭据，还是某个工作负载做出了无人预料的行为。它并不判定原因。第 4 章把恐慌容易混为一谈的两个问题分开了：如何让一份凭据停止生效，以及它在生效期间做了什么。遏制回答的是第一个，而下面的顺序保证第二个仍然可以回答。

顺序是：撤销授予、关闭会话、轮换凭据、从审计日志界定范围，然后记录并通知。每一步放在这个位置都有理由。

撤销授予排在第一位，因为没有权限的会话什么也做不了，重新连接也得不到任何东西。关闭会话排第二，针对的是在途的工作，以及在授予被撤销后仍然存活的打开的连接（第 4 章对已到期的凭据也指出过同一点）。轮换排第三，因为它能阻止新的登录，却对已经打开的登录无能为力。界定范围是只读的，需要的是记录而不是访问，所以要等到访问消失之后。通知排在最后，但每一步的记录不在其列：每一步完成时都会追加记录。

```
function containCompromise(incident, principal, since):
    unfinished = []
    // every step runs even if the one before failed: a stuck revoke
must
    // not leave sessions open and the credential valid
    for each step in ["revoke-grants", "close-sessions", "rotate-
credentials"]:
        outcome = "done"
        try:
            runContainmentStep(step, principal)
        on StepFailed:
            outcome = "failed"
            unfinished.append(step)
            notify("platform", "containment step failed: " + step + "
" + incident.id)
        // the log never delays containment; an unreachable log queues
the entry
        auditOrQueue({ action: "contain", actor: incident.commander,
                       subject: principal, request: incident.id,
                       step: step, outcome: outcome })
    // read-only and last: it needs the record, not the access
    scope = blastRadius(principal, since)
    saveIncidentRecord(incident.id, { principal: principal, since:
since,
                                      scope: scope, unfinished:
unfinished })
    notify("security", "contained, scope attached: " + incident.id)
    return unfinished
```

有两个细节值得说明。该函数返回未完成的步骤而不抛出异常，因为只遏制了一半的 principal 是一种需要跟踪的状态，而异常会终结唯一知道这件事的进程。`auditOrQueue` 与第 7 章中的辅助函数相同：日志中断只会推迟条目，绝不会推迟移除访问。这种不对称，与第 8 章为撤销所做的选择一致：没有日志条目的授予是一种暴露，没有日志条目的撤销则是日后可以补上的缺口。

每一步的含义取决于 principal 是什么。下表列出 Ledgerly 的四种情形。

Principal	撤销授予	关闭会话	轮换凭据
人，会话登录	从此人持有的每个会话登录上移除 JIT 三元组、`human_breakglass` 和 `human_readonly`	对每个登录执行 `endSession`；在 IdP 中停用此人，使新会话无法开启（第 7 章）	重置此人的 IdP 凭据和认证因子；会话登录会被删除，因此没有需要轮换的数据库密码

Principal	撤销授予	关闭会话	轮换凭据
服务，动态凭据	结束租约；登录随之消失（第 4 章）	broker 在结束租约时一并关闭会话	无需手动操作；下一次请求会获得新值
服务，A/B 对中的静态凭据	立即从泄露的登录上移除 `app_*` 角色，不等待旧登录排空	关闭它剩余的会话	向空闲的登录发布新值并切换指针；重启无法重新读取其机密的服务
`ledgerly_access_admin` 登录	移除它的权限，这会使使用过它的任何 principal 的配置工作停止	关闭它的会话	向该 principal 签发新凭据；范围问题变成哪些授予和登录无法由任何申请解释（第 3 章的漂移检查）

静态凭据这一行与第 4 章相符。泄露的静态凭据不是计划内的轮换，泄露的登录会在替代品就绪之前被切断，这会使使用它的服务发生中断。第 4 章已经接受了这一代价，我会保持这种接受：继续生效的泄露凭据是更大的问题。需要重启的服务，应当在关键时刻到来之前就列在一份清单上。在初始状态下，每个服务共用一个管理员登录，此时第一步就是整个公司的中断，这正是先建立例行撤销的最有力理由。

最后一行最沉重。管理访问的登录可以在所管理的角色内授予任何权限，所以遏制它，会让 gateway、流水线或 broker 在新凭据出现之前无法配置任何东西。这就是切断触达范围最广的登录所要付出的代价，团队应当事先决定由谁来承担。

依据记录界定影响范围

访问一旦消失，问题就变成它做了什么。答案是对 `access_audit_record` 的一次查询，其设计见第 14 章，使用的是该章定义的字段。该查询给出的是一个 principal 自时刻 T（其访问最早可能被滥用的时刻）以来的影响范围：它触及的每张表、返回了多少行、改动了多少行以及这些表中哪些包含 Restricted 数据。

```
// illustrative notation; field names as in Chapter 14
SELECT t.table_name,
       classOf(t.table_name)                  AS data_class,
       COUNT(*)                               AS statements,
       SUM(r.rows_returned)                   AS rows_returned,
       SUM(r.rows_affected)                   AS rows_affected,
       SUM(CASE WHEN r.outcome = "refused" THEN 1 ELSE 0 END) AS
refused,
       MIN(r.started_at)                      AS first_seen,
       MAX(r.ended_at)                        AS last_seen
FROM access_audit_record r
     EXPAND r.tables AS t                     // one row per table
touched
WHERE r.kind = "outcome"                   // one row per statement;
intent rows have no counts
  AND r.login IN :logins
  AND r.started_at >= :since
GROUP BY t.table_name
ORDER BY (data_class = "Restricted") DESC, rows_returned DESC;
```

选择 `:logins` 和 `:since` 是大部分判断所在之处。对于一个人，gateway 为每个会话创建新的登录，所以集合是该 principal 自 T 以来拥有过的每一个会话登录，而不只是一个。对于 T，应取访问最早可能被滥用的时刻，例如笔记本电脑最后一次确知还在主人手中的时间，而不是有人发现问题的时间。窗口过窄，会对一个错误的问题给出干净的答案。

结果的质量只取决于记录，所以在任何人阅读结果之前要先做三项检查。

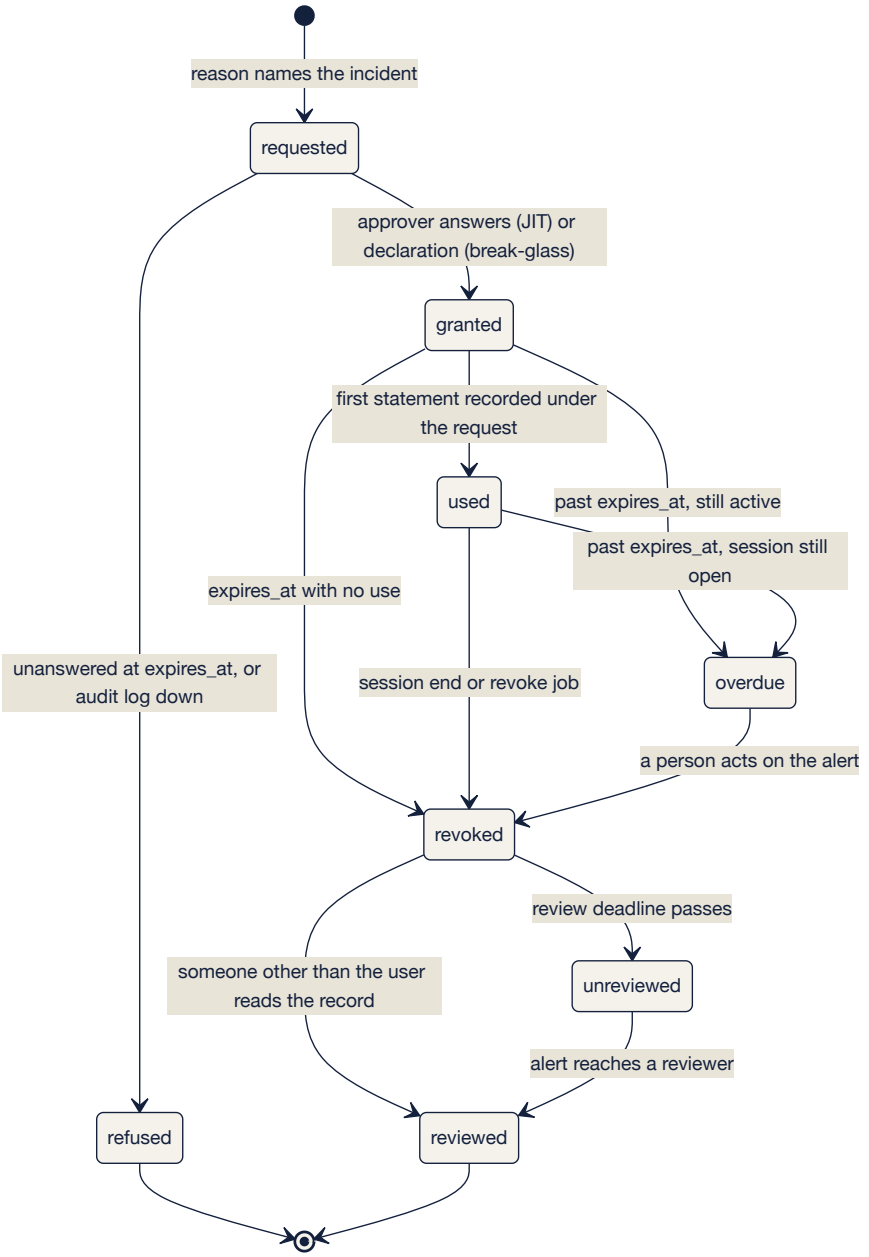
1. `verifyChain` 在该窗口内通过，如第 14 章所定义。如果失败，那么结果证明的是事故之外的某件事，而事故会得到第二份记录。
2. 窗口内没有序列空洞，也没有只在 gateway 一侧而数据库侧没有的会话，反之亦然，这就是第 14 章的缺口检测。窗口内的空洞，是答案未知的一段。
3. 该 principal 的旁路另行检查：它申请过的恢复、导出和副本（第 13 章），通过副本登记册查找，而不是通过这条查询。

如果任何一项检查失败，结果就是一个下界，而我会在事故记录中用这样的话来写明。这条查询自身也有局限。`rows_returned` 是一个计数，不是一份清单：它只说明窗口内某张 Restricted 表返回了若干行，而无法说明那是哪些客户。`fingerprint` 和 `literals_hash` 字段可以用来查询某条已知语句是否重复出现过，但字面值按设计已被遮蔽。因此记录界定的是暴露的范围，而不会逐一列举。Restricted 表上非零的 `rows_returned`，是安全联系人和读者的法务与合规人员需要的那条线，而它意味着什么的解释，包括任何通知义务，归他们所有。

事故访问的生命周期

遏制移除访问。事故同样需要增加访问：总得有人去查看，也可能有人要去修复。这个方向的风险，在于匆忙之间创建、到周一仍然存在的授予。解决办法是一个有明确终点的生命周期，事故中产生的每一份访问都要经过同样的五个状态。

该图是对第 8 章已保存记录的一种视图，而不是新的状态字段。`granted` 即申请记录中的 `approved`，`used` 表示申请名下记录了一条语句，`reviewed` 表示使用者之外的人读过记录。请留意两个到不了 `reviewed` 的出口：超过期限仍然存在的授予，以及从未进行的评审。



要使这一切成立，事故期间提出的每份申请都要在原因中带上事故的标识符。gateway 的表单可以强制要求这一点，在进行中的事故里，不带标识符的申请会被拒绝。正是这一条约定，让作业可以询问“这起事故授予的一切”。

```
function reconcileIncidentAccess(incident):
    // watches the incident's own requests; never changes access
(Chapter 8:
    // an automatic revoke of unknown grants is a way to break
production)
    for each r in requestsWithReason(incident.id):
        if r.status == "approved" and r.expires_at < now() -
graceMinutes:
            notify("platform", "incident grant past expiry: " + r.id)
        if r.status == "revoked" and reviewFor(r.id) is null
                and now() - r.expires_at > reviewDeadline:
            notify("security", "incident access not reviewed: " +
r.id)
    // the database side: grants that no request explains
    for each g in unexplainedGrants(since: incident.opened_at):
        notify("platform", "grant with no request: " + g.login)

function closeIncident(incident):
    reconcileIncidentAccess(incident)
    // an incident is not over while any access it created is still
open
    if openFindings(incident.id) > 0:
        raise IncidentHasOpenAccess
    markClosed(incident.id)
```

这个作业并不取代第 8 章和第 3 章的到期检查与漂移检查。它从事故这一侧审视同样的事实，并为事故设置了一道关卡：只要还有授予仍然有效，或者还有 break-glass 的使用未经评审，任何人都无法把事故标记为关闭。使用工单系统的团队可以把 `closeIncident` 接到工单上，把“我们忘了撤销”变成一个看得见的阻塞项。

已评审

最后一个状态是最常被跳过的，因为事故感觉已经结束了。评审就是第 8 章为 break-glass 所描述的那种，范围扩大到事故产生的每一项授予。它是无责的，问的是系统：这份访问是不是能完成工作的最小访问，有没有本该存在的更快路径？附录中有一份事故后模板。真正有价值的发现，通常是缺失的路径。

事故中的写入

修复是最容易让人想放松控制的地方，因为损害看得见，数据此刻就是错的。第 10 章已经确定，修复是一次数据修正；第 8 章已经确定，break-glass 不是用于修复的途径。在压力之下，我会保持这两项决定，并让合规的路径变快，而不是让错误的路径变得可用。

第一个选择是，这项需求属于哪条路径。检查的顺序从最窄的门走向最宽的门。

```
function chooseIncidentWritePath(need):
    // first match wins; Restricted values are never written by a
person
    if need.touches_restricted_column or need.is_owner_link_change:
        return "product path: admin-console asks payments-service
(Chapters 9, 10)"
    if need.is_schema_change:
        return "pipeline, expedited lane (Chapter 11)"
    if need.stops_ongoing_harm and need.is_small_limiting_action:
        // pausing, disabling or flagging: never the repair itself
        return "break-glass session, reviewed afterward (Chapter 8)"
    return "data fix: JIT request, runner, expedited review (Chapters
8, 10)"
```

break-glass 只用于小型的限制性动作：阻止某个作业写入错误行的标志，或者暂扣一个批次的状态。对数据的持久修复始终是数据修正，理由和之前一样：在原始 break-glass 会话中运行修复，会跳过唯一一项在语句运行前读取它的控制，而且它运行的地方仍然可能出现不受限制的 `UPDATE`（第 8 章把这一点列为已接受的风险）。在事故中，这种诱惑更强，一次错误写入的代价也更高。

数据修正保持它的形态。申请指明一份经过评审的修正，并携带取自合并记录的 `approved_hash`，runner 拒绝哈希不同的脚本，授予由修正推导而来，行数校验和快照依然适用（第 9 章和第 10 章）。事故改变的是评审。

```
function expediteFixReview(fix, incident, reviewer):
    // relaxes: the waiting period, and who may be asked first
    // never relaxes: a reviewer who is not the author, the hash, the
runner
    if incident.status != "open":
        raise NoOpenIncident
    if reviewer == fix.author:
        raise SelfApprovalNotAllowed
    mergeRecord = mergeFix(fix.id, reviewedBy: reviewer, expedited:
true,
                           incident: incident.id)
    // the hash is read from this record by the gateway, not from the
runner
    queueFollowUpReview(fix.id, reviewer: "platform", due:
nextWorkingDay)
    notify("platform", "expedited fix merged: " + fix.id)
    return mergeRecord
```

这与第 11 章给迁移提供的加急通道如出一辙：同样的作业、同样的凭据、同样的记录，只是时机不同。事后评审是偿还这条捷径的地方，也是把加急路径变成可计数事件的地方。如果团队发现一个季度中大多数修正都走了加急路径，就说明常规通道对这项工作而言太慢，正确的应对是加快常规通道，而不是停止计数。

> **警告**：一项“需要”break-glass 的修正，是关于别的问题的信号，如第 10 章所说：上限缺少某个列，审批人池对这个时段来说太小，或者某个 Restricted 列本应走产品路径。在事故中，这个信号很容易被忽略，而忽略两次代价高昂。

前面各章推迟处理的情形

有三种情形被前面的章节推到了这里。下表给出事先的决定；各小节说明理由。

情形	什么会停止	什么不变	事先决定的事项
gateway 宕机	人员的数据访问（没有第二道门）	服务仍以自己的凭据运行	谁来恢复 gateway，以及何时可以使用封存的凭证
事故期间需要恢复	如果这条路径演练过，则没有任何东西会停止	审批人、隔离的目标、登记册条目、计划的删除	指名审批人、准备好的目标、恢复 runbook
夜间审批缓慢	不属于紧急情况的修复	自我审批仍被拒绝	升级顺序，以及谁可以扩大审批人池

Gateway 宕机

第 7 章和第 8 章决定了不存在第二道门：人只能通过 gateway 到达数据库，所以 gateway 宕机同时也是 break-glass 宕机。那里的理由是，不经 gateway 就能使用的凭据，也同样不受日志、身份和到期的约束。我会在最糟糕的夜晚也坚持这一点，而宕机中的第一个动作，是留意什么还没有停。服务以自己的凭据连接，继续运行，所以许多恰好遇上 gateway 宕机的事故，根本不需要有人进入数据库。回滚部署、暂停作业、切换 feature flag、扩缩容服务：这些都不经过 gateway。

如果事故确实需要数据访问，那么此刻就是两起事故，第一起是 gateway。恢复它排在数据问题之前，而恢复它的是平台团队，使用的是部署路径和第 7 章要求的冗余。

如果所有实例都宕机，就不存在人员的数据路径，唯一剩下的工具，是第 3 章视为封存的超级用户。它是引擎层面的工具，而不是数据路径：它绕过 append-only 和 Restricted 保护，通向它的网络路由对人关闭。我会把它的使用限定于恢复 gateway 或 gateway 所依赖的东西，绝不用于读取或写入业务数据，并且会把这条规则写下来。

> **警告**：不要因为 gateway 宕机就用封存的凭证去查看数据。它绕过 append-only 和 Restricted 保护，它所给予的访问，正是前面每一章致力于防止的访问。如果必须读取数据，先恢复 gateway。

按这条规则使用它，有相应的机制。取得凭据会留下记录，第二个人在场并记下姓名，打开网络路由是一项有负责人和结束时间的变更，而且凭据在事后要轮换，因为有人见过它。这段时间 gateway 不在记录，所以引擎日志是唯一的轨迹。gateway 恢复之后，平台团队手动追加一个条目，类似第 14 章为迁移所描述的外部审计条目，并且这次使用要接受每次 break-glass 使用都要接受的评审。

事故中的恢复

第 13 章把恢复变成了一项申请：审批人不是申请人，目标是隔离的，有登记册条目，有计划的删除，并以关闭状态开始。那里指出的代价是事故期间的速度，对此的回答是演练并提前安排审批。这就是全部机制，因此有必要具体说明“提前”是什么意思。

在事故之前，runbook 会写明夜间谁可以批准恢复，目标已准备好并已隔离，恢复申请有一张可以填入事故标识符的表单。在事故期间，申请原样通过：审批人不是申请人，登记册条目先建，计划的删除在副本存在之前就已设定，而在副本打开之前，登录被停用，访问定义已应用。事故增加的是原因，而不是捷径。

副本的用途比它的制作方式更重要。事故期间，恢复出来的副本是一个读取来源，通常用于查看旧值：某一列在错误写入之前的状态，或者被误删的行。它的读者是申请上列明的人。随后的修复作为数据修正在主库上运行，从副本读到的值作为一份经过评审的脚本的输入，而第 13 章的 `reapplyDeletions` 在副本打开之前运行，这样副本就不会带回被有意删除的数据。以恢复出来的副本替换主库是一项恢复决策，归平台团队和事故指挥官。其中的访问问题就是上面那个：谁读取副本，它何时删除。

夜间审批缓慢

第 8 章的夜间场景以一个决定告终，本章保持这个决定：如果审批人池没有回应，而需求可以等待，那就留下一条说明然后停手。这是个好结果，也是旧的 Ledgerly 无法做到的。对于不能等待的需求，升级机制内置于申请之中，所以无人应答的呼叫会在同一份申请之内沿着审批人池向下移动。

```
function escalateApproval(request, pool):
    // one request, one record: the page moves, the request does not
restart
    for each approver in pool.inOrder():
        if approver == request.requester or approver ==
request.fix_author:
            continue
        page(approver, request.id)
        if waitForAnswer(request.id, within: escalationMinutes):
            return
    // pool exhausted, window still open: the incident commander
decides
    notify("platform", "approval unanswered, pool exhausted: " +
request.id)
    // never widens the pool or approves by default; the request stays
pending
    // until expires_at, when it is refused
```

围绕它有两条规则。第一，审批人池只能由拥有它的人来扩大，而扩大本身就是一条记录：在已宣布的事故期间，事故指挥官可以为这起事故增加一位指名审批人，并设定结束时间。第二，仍然不建立对值班人员自动批准的机制，因为几秒内批准每一份申请的控制，就不成其为控制。如果审批经常超出授予的时间窗口，发现项就关乎轮班表、审批人池的规模或这个时段，补救办法是向审批人池添加人手，而不是把审批人从流程中拿掉。

演练各条路径

从未运行过的路径就是未经测试的路径。本章全部由那些没人愿意在事故中第一次发现的步骤构成，所以每一步都需要演练，这里称为 game day：一场有场景、有预期结果、并对记录做检查的定期演习。四个场景涵盖了本章所构建的内容。它们在设计上都是防御性的，每个都从监控已经知道如何识别的触发条件开始，并带有可观察的通过条件。

场景	触发条件	预期行为	通过条件	涉及的部分
凭据泄露	演练用 principal 的凭据被宣布暴露	按顺序执行 `containCompromise`；范围查询附在事故上	该 principal 没有存活的登录，unfinished 为空，范围列出演练触及的每张表	身份、授予、记录
gateway 宕机	在非生产副本中先停止一个、再停止全部 gateway 实例	人员访问停止；服务继续运行；恢复路径运行	平台团队按 runbook 恢复 gateway；封存的凭证未使用，或按其规则使用并留有记录	路径、记录
可疑导出	由 `evaluateAlertRules`（第 14 章）发出的大 `rows_returned` 告警	遏制该 principal；在副本登记册中找到目的地；将其删除	副本被找到、结束并记录；范围查询与登记册一致	身份、记录、执行
误操作的批量 `UPDATE`	在演练环境中，一次符合行数上限、却改错了行的写入	停止进一步的写入；根据快照计算反向操作；作为加急数据修正运行	反向操作由非作者的评审人合并；后续评审已排队	执行、授予、记录

game day 只依据观察到的结果判定通过，绝不依据有人记得该怎么做。如果演练因为有人知道一条捷径而完成，它就是失败了，而这条捷径就成了需要构建的东西。把每一步实际花费的时间记下来；这些数字属于组织自己，而我不会给出一个并不属于它的数字。

尽可能使用非生产副本，在生产环境的演练中使用合成 principal，这样就不必为了练习读取而读取真实的客户数据。演练用的副本与其他副本一样登记并关闭（第 13 章）。

权衡与成本

第一项成本是演练本身。game day 会占用工程师做产品工作的时间，一次演练需要负责人和日程。对于三个人的团队，一年做一次覆盖凭据泄露的演练是合理的起点，其他三个场景可以等团队壮大之后再做。

第二项成本是遏制所造成的中断。立即撤销某个服务的凭据，就是该服务的一次中断，遏制 `ledgerly_access_admin` 则会使配置工作停止。这两者都是应当与承担后果的人事先做出的选择，因为凌晨 3 点做决定的人，不应同时还在了解这个选择的代价。

第三项成本是事故标识符的约定。它给每份申请增加了一个字段，也给表单被拒绝多添了一个理由，而且它只在人们愿意填写时才有效。依赖于它的对账在没有填写时会悄无声息地失败，所以 gateway 应当强制要求这个字段，而不是依赖习惯。

第四项成本是对范围保持诚实：有些事故以一个比真相更大的界限收场，我会报告这个界限，而不是声称拥有记录所缺乏的精确度。

不要把完整的机制套用到尚无审计日志的系统上。`blastRadius` 需要记录，`reconcileIncidentAccess` 需要申请记录；没有它们的团队应当先建立这些（第 8 章和第 14 章），同时写一份简短的 runbook，写明谁来撤销、谁来关闭会话以及谁负责回答“触及了什么”这个问题。

常见错误

- **只轮换，不撤销**。轮换改变机密，却让打开的会话和已授予的权限原样保留。这很诱人，因为轮换是团队已有的例行动作。应当先撤销，再关闭会话，然后轮换。
- **先界定范围，后遏制**。想弄明白的冲动先于想制止的冲动，团队可能花一个小时在查询上，而访问依然有效。应当先遏制。记录在遏制之后依然存在，因为它存放在访问无法触及的存储中。
- **不加检查就信任日志**。对带有缺口的窗口做范围查询，读起来像一个干净的答案。应当先检查链和序列，其中任何一项失败时，就报告下界。
- **把 break-glass 当作修复通道**。它快，也随时可用，但没有语句评审。应将它用于小型的限制性动作，并把修复作为加急数据修正来运行。
- **让事故访问保持打开**。原因修复后事故就感觉结束了，而授予留了下来。应当把关闭事故与对账绑定，这样访问仍然打开时，工单无法关闭。

要点

- 紧急程度改变控制的速度，绝不改变行动的身份、运行的语句或留下的记录。
- 遏制有顺序：撤销授予、关闭会话、轮换凭据、从审计日志界定范围，然后记录并通知，并且即使前一步失败，每一步也都要执行。
- 撤销不是轮换：轮换能阻止新的登录，对已经打开的登录无能为力，所以泄露的静态凭据要先切断、后替换。
- 影响范围来自对 `access_audit_record` 的一次查询，只要链或序列存在缺口，它就是一个下界。
- 事故访问要经过 requested、granted、used、revoked 和 reviewed，只要事故创建的任何授予仍然打开，事故就无法关闭。
- 事故中的修复是带加急评审的数据修正，break-glass 则留给制止损害的小型动作。
- 未经演练的路径只是一种假设，所以凭据泄露、gateway 宕机、可疑导出和误操作的批量 `UPDATE` 各有一场带有可观察通过条件的 game day。

第 16 章把全书的控制措施汇集为一份检查清单和一个成熟度阶梯，并回到每一章都在服务的那句话。

第 16 章 — 访问检查清单与结语

试想 Ledgerly 在第 1 章的安全问卷一年之后。另一位潜在客户发来同样的两行问题：“列出所有能连接到生产数据库的人员和系统”和“如何得知他们做了什么？”Alex 打开的是访问清单，而不是密码管理器，读完那份门的清单，一个下午就回复了对方。这份答复没有任何亮眼之处。它很短，每一行都有负责人和终止时间，也没有哪一项需要翻查。

本周值班的 Robin 在聊天中问道：“我们有 gateway、runner、流水线、审计存储和三个评审作业。其中哪些是真正需要的，哪些只是因为书里这么说才建的？”Alex 给不出一个干脆的回答。有些组件在第一次有授予自行到期时就证明了自己的价值；另一些从未真正派上过用场，Alex 也说不清这意味着它们运转正常，还是一直闲置。

本章同时服务于三大支柱：授权、执行与问责。它不增加任何新控制。它增加的是目标状态的各个阶段，按团队能够达到的顺序排列；并列评分的各种模式；以及一份清单，清单的每一行都是观察结果，而不是意见。它回答的正是 Robin 提出的问题：团队如何决定下一步该建什么，又如何得知某项控制确实有效？

问题及其难点

读完前十五章，读者手里有许多控制，却没有顺序。每一项控制都带着自己的论证引入，每段论证在其所在章节中都成立。放在一起看，它们像是一个耗时一年甚至更久的项目，面对这份清单的团队往往走向两个极端之一：从最便宜的做起，等工作变得牵涉政治时就停下；或者从最令人恐惧的做起，精心构建设计中的一个复杂部分，而共享密码还躺在五个配置文件里。

难点在于，在访问工作中，“完成”很难看出来。纸面上存在的角色 schema 与真正得到强制执行的角色 schema，在评审会议上看起来并无二致，于是团队按已安装了什么来衡量，而“已安装”恰恰是错误的度量单位。有用的单位是他人可以执行的测试：一条可以追踪的语句，一次被拒绝的连接，一项已经消失的授予。

这同样取决于系统如何构建，而不是策略如何规定。策略说明应当发生什么；设计则决定什么可能发生，以及发生时什么会留下记录。下面的每一行都可以通过查看系统、日志或表来回答；只能靠询问某人的意图才能回答的行，不在清单之内。

论点的落地

本书论证的是一句话，下面对照各章对它加以检验：

> 生产数据库访问不是凭据分发问题，而是授权、执行与问责问题。

这句话有三大支柱，每根支柱都立在特定的章节之上。授权是授予加路径，按每次请求逐一决定。执行是每条语句所经历的一切。问责是身份加记录。密码只触及其中的第一项，而且只在门口起作用。

支柱	模型的组成部分	构建它的章节	读者现在能够回答的问题
授权	授予、路径	3、5、6、8、12	这个 principal 可以做什么、在哪里做、持续多久，哪些列被隐藏？
执行	执行	7、9、10、11、13	这条语句是如何到达数据的，由谁或什么检查过，又受到什么约束？
问责	身份、记录	4、7、14、15	是哪个人员或 workload 实例执行了操作，依据谁的申请，哪些证据把它与结果联系起来？

第 1 章和第 2 章为三者搭建框架，第 15 章则在压力之下检验它们。第 7 章出现两次，因为 access gateway 既是路径，也是人员转化为日志能够指名的会话登录的地方；第 13 章归在执行之下，因为 `reporting-job` 的导出步骤会运行一组语句，而其他旁门则是绕开同一批控制的路线。

为什么凭据工作是必要的，却不是答案

第 4 章讲的是凭据，也是多数读者以为会成为本书中心的一章。它是必要的。从共享文件读取密码的服务没有自己的身份，而没有各不相同的身份，记录就无法指出任何人。短期凭据和动态凭据让撤销成为例行步骤。workload identity 则去掉了那个文件。

以上这些都不能决定 principal 进入之后可以做什么、语句如何检查，或者记下了什么。团队可以把第 4 章做得十全十美，却仍然有一个拥有每张表的角色，一个把每条语句按输入原样提交的笔记本电脑会话，以及一

份日志——日志里每一行都有互不相同的名字，却没有一行带有理由。密码存储提供身份部分所需的各不相同的名字，并通过撤销提供授予的一部分；密码只触及第一根支柱，而且只在门口起作用。其余部分要在别处构建，本书其余内容讲的就是在哪里。

当某个提案声称“我们将通过改进 X 来修复访问问题”时，应当在批准之前，先把 X 放到模型上：

```
function placeControl(proposal):
    parts = partsOfModelTouched(proposal)   // Identity, Grant, Path,
Execution, Record
    if "Execution" not in parts and "Record" not in parts:
        // a control that touches only the door leaves two of three
pillars as they were
        note("improves how access is obtained; says nothing about what
it does or what remains")
    for each part in ["Identity", "Grant", "Path", "Execution",
"Record"]:
        if part not in parts:
            note("unchanged: " + part)
    return notes
```

这个函数不是关卡。它指出提案让哪些部分维持原样，因为空缺的部分是一个发现项，而不是遗漏。

按支柱评分的模式目录

第 1 章勾勒了进入的各种方式的范围，第 7 章比较了具体的路径。下表把该范围内的五种模式并列，并按三大支柱为每一种打分。这些评分是我的判断，而不是测量结果。它们说明在前面各章的授予和日志记录保持相同的前提下，该模式自身能够为一根支柱提供多少证据。“强”理解为“该模式就是为证明这一点而构建的”，“部分”理解为“它能证明其中一部分，或者只有在补充其他东西之后才能证明”，“无”理解为“该模式在此保持沉默”。

模式	授权	执行	问责
直接访问（客户端到数据库）	部分：仅有授予，触达范围取决于网络允许什么	无	至多为部分：若有人预配，则按人登录，否则为无
bastion	部分：收窄网络路径	无：它只转发，分不清 `SELECT` 与 `DROP`	部分：能看到谁登录了主机，看不到运行了什么

模式	授权	执行	问责
session proxy 或 access gateway	强：按会话登录、到期、按请求授予	部分：受授予和受保护设置约束，语句可以自由键入	对谁和何时为强，对什么为部分
语句受控执行器	强	强：运行前先检查，并绑定到已批准哈希	强：语句、审批人和结果在同一条记录中
产品路径（无 SQL）	强：操作本身就是权限	强：操作是固定的	部分：人员出现在工具的审计轨迹中，数据库日志显示的是服务登录

直接访问

- **验证**：登录确实存在，且其授予与当时的状态一致。
- **无法验证**：共享登录是由哪个人使用的，或者某条语句做了什么。
- **成本**：无需构建成本，这正是它的吸引力。所有成本都在以后到来，表现为一周的翻查。
- **故障模式**：第 1 章中的共享管理员密码。
- **不应使用的情形**：不止一个人员或 workload 可以接触该登录时。对于只有一名工程师、唯一使用者就是其负责人的系统，这尚可容忍，而我会写明什么情况会使这一点不再成立。

Bastion

- **验证**：连接经由一台受控主机而来，并且，如果主机上的账号按人设置，还能验证是谁登录了那里。
- **无法验证**：任何语句是什么，因为该主机不理解数据库协议。
- **成本**：需要一台要打补丁并保持可用的主机，换来的是一个可以关闭网络路径的单一位置。
- **故障模式**：共享的 bastion 账号会把这台主机重新变成共享密码，只是提前了一跳。
- **不应使用的情形**：需要语句级证据时。它仍然有用，可作为位于某个确实留有记录的组件之前的网络层面的触达手段。

Session proxy 或 access gateway

- **验证**：一位具名人员在两个时间戳之间，在特定授予之下持有一个特定的会话登录。

- **无法验证**：在该会话中键入的语句是不是个好主意。该会话是一次 session handout（第 9 章），任何授予所允许的内容都可以键入。
- **成本**：一个新系统，在有人需要时必须可用，还有随之而来的审批路径（第 7 章和第 8 章）。
- **故障模式**：旁边还开着第二扇门的 gateway，这不过是换了个更好听的名字的共享密码。
- **不应使用的情形**：任务重复出现且形态固定时。它们属于产品路径，在那里没有人需要会话。

语句受控执行器

- **验证**：这组确切的语句（由其哈希标识）经具名人员批准、经过检查、在约束范围内运行，并产生了经过计数的结果。
- **无法验证**：分析是否理解了该语句。查询分析减少错误，但不是授权边界，授权边界仍由授予承担（第 9 章）。
- **成本**：每项任务都要等待评审，没有任何交互，而且必须有人构建并维护 runner。
- **故障模式**：总在几秒内批准的审批，这是多了几道步骤的安全剧场。
- **不应使用的情形**：工作带有探索性时。想要弄清某个状态的人首先需要一个读取会话，`human_readonly` 正是为此而设。

产品路径

- **验证**：某项已定义的操作，为某个工具中的某个角色运行，并留下该工具自己的审计轨迹。
- **无法验证**：任何无人为之构建功能的任务。自由形式的问题没有路径。
- **成本**：每项任务一项功能，且是五种模式中变更周期最慢的。
- **故障模式**：积累起“运行任意 SQL”逃生舱口的产品路径，原因是第一个紧急案例没有对应的功能。
- **不应使用的情形**：任务尚未重复出现时。第 10 章描述了把重复出现的修正转为功能的信号，而只发生过一次的任务不是那个信号。

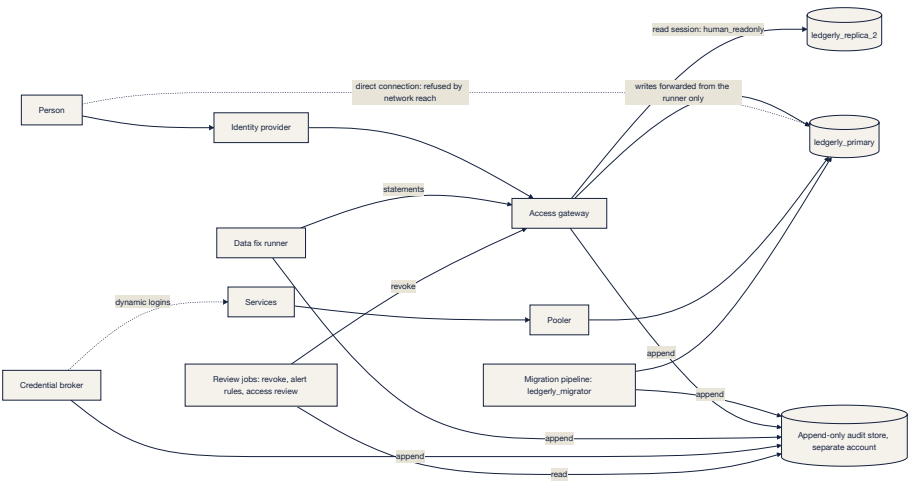
最右侧的模式并不总是最佳选择，对于关于某位客户发票的一个问题，语句受控执行器是过度的。我会逐项任务地选择，取仍能为该任务所危及的支柱提供证据的最便宜的模式：

```
function choosePattern(task):
    // a person who never opens a session cannot leave one open
    if task.repeats and task.hasFixedShape:
        return "product path"
    // a schema change takes the migration pipeline, never the runner
(Chapter 11)
    if task.changesSchema:
        return "migration pipeline"
    // Restricted columns and owner links are changed by the product,
not by SQL
    if task.writesRestrictedColumn or task.changesOwnerLink:
        return "product path"
    if task.writesData:
        // reviewed statement set, approved hash, runner executes
        return "statement-mediating executor"
    if task.readsData:
        return "access gateway: human_readonly on ledgerly_replica_2"
    // anything else is an exception with a named owner and an end
    return "exception: owner, reason, ends_on"
```

最后一行才是关键。不适合的路线要连同负责人和终止日期一起进入清单，正如第 1 章的清单块所要求的，而缺少二者之一的例外只是遗留物。

参考架构

下图展示了 Ledgerly 的目标状态，把本书的每项控制放在一张图里。请注意三点：除经由 gateway 之外，没有任何人员的箭头能够到达数据库；人员的每次写入都要经过 runner；审计存储位于数据库管理员无法写入的账户中。



该图做了简化：第 6 章的应用读取副本与其写入方共用一个框，而脱敏分析数据的加载、供应商导出和备份（第 13 章）则被略去，因为每一项都是有负责人和到期时间的拷贝，把所有拷贝都画上的图将没有可读的中心。

每个框都对应模型中的一部分：IdP 和凭据 broker 提供身份；角色、会话登录和到期作业提供授予；虚线箭头背后的网络规则和 gateway 提供路径；runner 和流水线提供执行；审计存储和评审作业提供记录。不提供任何部分的框是可以移除的候选，Robin 的问题值得用这项测试来检验。

带退出测试的成熟度阶梯

目标状态是一个方向，而不是终点线，因此这里把本书的各个步骤归为五个阶段。这种分组是我的做法。它依据的是一个团队在下一项变更变得更容易之前，能够改变多少；不同的组织会重新排序；我愿意为之辩护的唯一顺序是访问清单排在最前，因为没有它，没有人知道之后的每一步必须覆盖什么。记录也最好尽早以简单形式开始，因为有了证据，之后的每个阶段都更容易评判。

阶段在其退出测试通过时才算完成，而“我们装好了”不是测试。每项测试都是他人可以重复的观察。

阶段	增加的内容	退出测试	团队为何在此停滞
1. 认清各扇门	principal、门、拷贝的访问清单（第 1、2、13 章）	一份文档列出每扇门及其负责人和终止时间，新来的工程师仅凭它就能回答“谁能触达主库？”	访问清单被当作一个项目，因此永远完不成，也从未获得信任
2. 为每个 principal 命名	每个 principal 各有登录和角色、动态凭据、按角色的预算（第 3、4、5 章）	一天的数据库日志里没有共享登录，也没有服务以所有者或管理员身份连接	切换对每个服务都像是中断风险，因此共享密码“暂时”保留
3. 单一路径，受约束	gateway、副本、JIT 与 break-glass、分类与脱敏（第 6、7、8、12 章）	从笔记本电脑直接连接主库会失败，JIT 授予在 `expires_at` 时无人操作也会消失，读取会话不返回任何 Restricted 列	担心 gateway 是单点故障，以及夜间无法回应的审批人

阶段	增加的内容	退出测试	团队为何在此停滞
4. 受控且留痕	执行策略、runner、经评审的修正、仅限流水线的迁移（第 9、10、11 章）	来自人员自己客户端的写入被拒绝，且 runner 拒绝哈希与 `approved_hash` 不同的语句集	评审成为瓶颈，于是例外的堆积速度超过产品功能
5. 可证明且经演练	哈希链审计存储、告警规则、访问评审、game day（第 14、15 章）	从昨天随机挑出一条未事先告知的语句，无需询问任何人，就能追溯到人员、申请和结果；`verifyChain` 通过；game day 以根据记录确认已撤销访问收尾	日志存在却无人阅读，而发布延期时 game day 是最先被砍掉的

阶段被宣布完成之后，仍可能无法通过其测试，因为配置会漂移；按计划重复运行测试，能在漂移还小的时候将其发现。

> **注意**：只有一个数据库和四名工程师的小团队，完全可以合理地停在第三阶段，并写明什么情况会让它继续前进。要紧的是那份书面理由，而不是所达到的高度。

检查清单

检查清单按支柱分组，每一行都是一项观察。使用约定与退出测试相同：“否”是有负责人和日期的发现项，而不是需要辩解开脱的失败。

```
function runChecklist(items):
    findings = []
    for each item in items:
        answer = observe(item.check)     // read a system, a log or a
table, never ask what is intended
        if answer != "yes":
            findings.add({ item: item.id, owner: nominate(item), due:
dateByWhichToDecide })
    // a finding may end in "accepted, with reason, until a date"; it
may not end in silence
    return findings
```

授权

- [ ] 每张表都归 `ledgerly_owner` 所有，且没有任何人员或服务持有其登录。
- [ ] 每个服务都以属于其自身 `app_*` 角色成员的登录连接，且没有应用角色持有 DDL。

- [ ] 任何人员都不存在常驻写入访问；`human_readonly` 是默认角色，`human_jit_write` 是上限而不是已授予的角色。
- [ ] 每项 JIT 和 break-glass 授予都有 `expires_at`，且撤销作业及其独立的到期授予检查近期都已运行。
- [ ] 从笔记本电脑直接连接主库会被拒绝，且副本拒绝管理员登录。
- [ ] Restricted 列不出现在任何 JIT 申请、任何读取会话或任何分析副本中，且分类目录列出了存在的每一列。
- [ ] 每份备份、转储、导出和 staging 拷贝都有具名负责人和到期时间。

执行

- [ ] 来自人员和自动化的语句，都经由执行策略到达数据，且 deny / gate / allow 矩阵置于版本控制并有负责人。
- [ ] 写入会话在关闭 auto-commit、设有行数上限的条件下运行，且受保护设置无法在会话内部更改。
- [ ] JIT 写入指明一项经过评审的修正，且 runner 拒绝任何哈希与 `approved_hash` 不同的语句集。
- [ ] 所有者关联变更和对 Restricted 列的写入，在每条通道中都走产品路径。
- [ ] 模式变更由流水线以 `ledgerly_migrator` 身份运行，且没有任何人员持有该登录。
- [ ] 导出步骤使用权限范围窄的角色运行，其文件有具名负责人。

问责

- [ ] 每个登录恰好对应一个人员或一个 workload 实例，且数据库日志可以显示这一点。
- [ ] 每条写入记录和每条提权会话记录都带有申请、审批人、指纹和返回的行数，且字面值已做遮蔽处理。
- [ ] 审计存储位于数据库管理员和 `ledgerly_access_admin` 都无法写入的账户中，且 `verifyChain` 通过。
- [ ] 序列号缺口，或者没有数据库会话的 gateway 会话（或相反的情形），会产生一项发现。
- [ ] `evaluateAlertRules` 和 `runAccessReview` 在上一个周期内已运行，它们提出的撤销已被应用，或附理由被接受。

- [ ] `containCompromise` 已在 game day 中演练过，且影响范围查询已依据记录得到回答。

文化：安全的路径必须是最容易的路径

本书中的每项控制，对使用它的人而言都是成本，而这个人要在一周中最糟糕的夜晚决定是否付出这项成本。缓慢或经常宕机的控制会被绕过，因为设计让安全的路线变成了困难的路线，而事故不会等待。有用的做法是把绕过视为设计缺陷。

我会把一条规则放在文化的中心，并让它可以度量：

```
function reviewBypass(signal):
    // signal: someone used a second door, a shared login, or an
unplanned break-glass
    cause = ask("what made the intended path unusable at that
moment?")
    if cause in ["slow approval", "gateway unavailable", "missing
product feature", "grant too narrow"]:
        // a design defect: repair the path, and note the date by
which it should be easy
        openFinding(cause, owner: "platform", due: nextReview)
    else if cause == "unknown":
        // an unexplained bypass stays open until it is understood
        keepOpen(signal)
    // nothing here assigns the signal to a person
    return cause
```

有两个习惯支撑这条规则。统计 break-glass 角色被声明的频率，并且每次都追问是否本应由 JIT 路径或产品功能来承担；次数上升说明正常路径太慢，而整整一年为零，可能意味着这个角色有效，也可能意味着没有人相信它会被批准，这要靠 game day 来澄清。还要给审批人一些他们在夜间能够回答的东西：耗时数小时的审批教会申请人多要超出所需的权限，而总在几秒内获批的审批则让人学会不去阅读它（第 8 章涵盖审批人的检查清单）。

权衡与成本

只提出建议的结语章，会重蹈本书所警告的覆辙，所以这里说明整个设计的成本。参考架构中的系统有许多部件，每一个都必须在有人需要的那一刻可用。gateway、runner、流水线、凭据 broker、审计存储和评审作

业各有负责人、值班路径和故障模式。小型平台团队可能发现，自己的时间都花在维持这些控制运转上，这是一项真实的成本，需要与它消除的暴露相权衡。

检查清单和阶梯各有其成本。每季度读一遍的检查清单，除非发现项得到关闭，否则会沦为仪式；而当团队为阶段名称而优化时，阶梯会变成一份状态报告；退出测试是解药，因为它重新运行成本低，又难以造假。“强”的评分说明的是某个模式被构建出来要证明什么，而不是它的某一次具体部署确实做到了。

何时不应使用本章的这套机制：只有一个数据库、没有供应商的单工程师系统，需要的是一个无人使用的所有者登录、一个只读角色、一份能够恢复的备份以及一页纸的访问清单。本书面向的是 principal 的数量或错误的代价超出这一水平的时刻，而我不知道对某个具体组织而言这个时刻在哪里；这取决于有多少人可以接触数据、其中有多少是 Restricted 以及一个错误会在多长时间里不被发现。

常见错误

- **把检查清单当作证书**。做完一遍就把结果归档，是很诱人的。检查清单显示的是读取当天哪些地方缺少证据。应按计划重新运行，并让发现项保持开放，直到每一项得到解决，或附理由和日期被接受。
- **先构建最右侧的模式**。语句受控执行器之所以有吸引力，是因为它看起来控制最多。如果其下没有访问清单和各不相同的 principal，它所受控的只是共享登录的语句。应按顺序攀登阶梯，或写明为什么另一种顺序更合适。
- **为应急增设第二扇门**。它看起来像是韧性，实际重建了共享密码。如果团队在 gateway 成熟期间需要这样一扇门，它就需要负责人、使用时的告警和退役条件（第 8 章）。
- **把绕过当作行为问题**。有人绕开某项控制时，是设计让绕开比预期路径更便宜。应当追问在那一刻预期路径的成本是什么，并改变它。

要点

- 生产数据库访问是授权、执行与问责问题，而密码只触及其中的第一项，而且只在门口起作用。
- 第 4 章的凭据工作是必要的，因为身份使记录成为可能；它又是不充分的，因为它对 principal 可以做什么、语句会做什么只字未提。

- 逐项任务地在直接访问、bastion、access gateway、语句受控执行器和产品路径之间选择，取仍能为该任务所危及的支柱提供证据的最便宜的模式。
- 阶段在退出测试通过时才算完成，而他人可以重新运行的测试，是唯一能经受住漂移的“完成”形式。
- 检查清单的每一行都是对系统、日志或表的观察；“否”是有负责人和日期的发现项，绝不是对某个人的裁决。
- 安全的路径必须是最容易的路径，每一次绕过都是信息，说明设计在哪里让预期路线变得昂贵。
- 控制缩小影响范围，使行为可见，同时消耗可用性、注意力和等待时间，因此在采用之前就要点明这些成本，在付出时再点明一次。

Alex 在第二天早上以一项测试而不是一番辩护回答了 Robin 的问题：一个组件，如果它提供了模型中其他任何东西都没有提供的某一部分，并且其退出测试通过，就保留。问卷还会再来，来自另一位客户或另一位审计员，上面还会有同样的两行。本书能留给读者的，是这样一个系统：它能凭一份文档回答第一行，凭一份记录回答第二行，并且让值班人员无需英雄式的努力就能做对事。

附录

本附录是本书立论所依据的三大支柱——授权、执行与问责——的参考资料，共分六个部分：术语表；第 16 章所评分模式的速查表；第 9 章的 deny / gate / allow 矩阵；从审计人员的问题到相应制品的对照表；以及两份模板，一份用于第 14 章的访问评审，一份用于第 15 章的事故后评审。

这里没有任何新内容。表格若重复某一章中的表，其单元格即该章原文；两者若有出入，以章节为准。对任何监管制度的解释，由法务与合规人员负责；对照表列出的是工程制品，不作更多声明。

A. 术语表

术语按字母顺序列出，并注明各自由哪一章构建。这是本书词汇的最终参考：每个条目都是简短形式，所注明的章节才有完整论述。

术语	含义
access gateway	具备身份感知能力的中介，负责对人员进行身份验证、签发短期凭据并记录会话。它不是 bastion，也不是数据库 proxy（第 7 章）。
access_audit_record	为每条语句写入的条目，一条 kind 为 intent，一条 kind 为 outcome，包含 principal、登录、申请、语句 `fingerprint`、涉及的表和列，outcome 条目还包含行数和结果。它是模型中记录这一部分的载体（第 14 章）。
bastion	经过加固的主机，是通向受保护系统的唯一网络路径。它转发流量，不理解数据库协议（第 7 章）。
break-glass	带有大量日志记录并强制评审的紧急访问。角色为 `human_breakglass`，申请由申请人自行声明（第 8 章）。
deny / gate / allow	执行策略的几种答复，顺序始终如此。**Deny** 表示绝不通过此路径运行。**Gate** 表示只有获得额外审批后才运行，或只通过产品路径运行。**Allow** 表示在会话的授予范围内运行（第 9 章）。
PII	个人可识别信息。“敏感数据”是 Restricted 与 Confidential 数据的总称（第 2 章）。

术语	含义
principal	任何进行身份验证的对象：人员、服务、作业、供应商集成（第 1 章）。
runbook	针对可重复任务的、成文并被遵循的流程（第 10 章）。
runner	经评审的数据修正 runner。仅当语句集的哈希等于 `approved_hash` 时才执行，且 gateway 只转发来自它的 JIT 写入（第 10 章）。
session handout	一种模式：gateway 向 principal 提供会话登录和连接，由 principal 自己的客户端发送语句。登录上的授予限定一切（第 9 章）。
workload identity	平台赋予运行中 workload 的身份，使其无需存储密码即可完成身份验证（第 4 章）。
安全剧场	增加摩擦却没有减少暴露的控制（第 2、8 章）。
绑定语句的审批	附于将要运行的确切语句集、由 `approved_hash` 标识的审批，而不仅仅附于时间窗口或权限集（第 8、9、10 章）。
常驻访问	无论是否使用都持续存在的访问（第 1 章）。
产品路径	工具中的某项已定义操作，例如 `admin-console` 请求 `payments-service` 更改 Restricted 列，人员无需接触 SQL（第 9、10 章）。
查询分析	在语句运行之前对其检查：语句类型、涉及的表和列、写入是否有受限行数的保护、预期行数。它减少错误，但不是授权边界（第 9 章）。
撤销、轮换	撤销使凭据或授予立即失效。轮换按计划或出于怀疑替换凭据，不会关闭已经打开的会话（第 4、15 章）。
登录	principal 用来连接的数据库级账号（第 1、3 章）。
动态凭据	按请求为单个 principal 生成的短期凭据，因此两次请求绝不会得到相同的值（第 4 章）。
短期凭据	在限定的短时间内自行到期的凭据（第 4 章）。
遏制	按固定顺序停止某个 principal 的访问：撤销授予、关闭会话、轮换凭据、界定损害范围、记录并通知（第 15 章）。
访问评审	定期检查每个 principal 的访问是否仍有必要。它将已授予的内容与实际使用的内容对比，由具名评审人做出决定（第 14 章）。
分析副本	`ledgerly_analytics` 存储。它是生产数据的拷贝，而不是副本（第 6 章）。
哈希链	一种记录格式，每个条目都带有前一条目的哈希，因此被修改或删除的条目会使 `verifyChain` 失败。它提供篡改的证据，并不能阻止篡改（第 14 章）。

术语	含义
会话	principal 的一次经过身份验证的连接，或经 gateway 中转的一次交互（第 5、7 章）。
记录	模型中追问还剩下什么证据、这些证据能否把语句与人员、申请和结果联系起来的那一部分（第 14 章）。
机密	一旦泄露就能授予访问的任何值（第 4 章）。
即时（JIT）访问	为特定任务、在有限时间内授予的访问（第 8 章）。
静态凭据	在有人更改之前保持不变的凭据（第 4 章）。
角色	数据库中一组命名的权限。角色绝不是人员（第 3 章）。
列级授予	对表的指定列授予某项权限。引擎不支持时，Restricted 列放在上限之外的单独表中，JIT 授予降级到表级（第 3、8 章）。
临时查询	由人员编写、在任何应用之外运行一次的查询（第 10 章）。
令牌化	用只有受控服务才能映射回去的替代值取代原值。它既不是加密，也不是脱敏（第 12 章）。
旁门	绕过主库访问控制的数据拷贝或导出路径（第 6、13 章）。
凭据	principal 用来验证身份的东西：密码、证书、token。凭据是机密的一种（第 4 章）。
迁移	由 `ledgerly_migrator` 运行的、带版本的模式变更。“模式变更”是一般概念，“迁移”是其制品（第 11 章）。
权限	授予所允许的某项具体内容，例如某张表上的 `SELECT`（第 1 章）。
上限	JIT 申请至多可请求的内容，由 `human_jit_write` 定义。绝不包括 Restricted 列（第 3、8 章）。
身份	对 principal 确为其所称身份的已验证声明。角色不是身份（第 1 章）。
审计日志	记录谁对哪些数据做了什么，保存在操作者无法修改的地方。`audit_events` 表是应用的业务记录，不是审计日志（第 14 章）。
受保护设置	principal 无法在会话内部更改的会话设置：语句超时、行数上限、auto-commit 模式、默认只读、角色。只有在有机制强制执行的地方，才称为不可覆盖（第 5、9 章）。
受控执行	一种模式：principal 从不持有连接，而是提交语句，由 gateway 或 runner 分析，再由 runner 执行（第 9 章）。
授予	把某项权限授予角色，把角色授予登录或另一个角色，或者（仅限 JIT）把指定列上的权限直接授予会话登录。也指由此产生的记录（第 3 章）。

术语	含义
数据分类	将数据归入命名的类别，由类别决定处理规则（第 12 章）。
数据拷贝	生产数据被复制到的任何位置：副本、分析副本、备份、转储、导出、staging 数据（第 6、13 章）。
数据库 proxy	能理解协议的中介，负责池化、路由或过滤数据库连接。当话题只涉及池化时，称为 pooler（第 5、7 章）。
数据类别	四个命名的数据类别：**Restricted**（泄露后难以挽回，例如 `iban` 和 `token_value`）、**Confidential**（关于客户和联系人的个人数据或商业数据）、**Internal**（不含个人内容的运营数据）、**Public**（可安全公开）。用作类别名称时首字母大写（第 2、12 章）。
数据修正	人员在正常应用路径之外对生产数据做的更改，用于修复应用无法修复的状态（第 10 章）。
四眼原则	有风险的操作运行之前，由第二个人评审的审批。它说明谁来批准，而不说明批准的是什么（第 8 章）。
所有者关联变更	对把行关联到所属客户的列执行的 `UPDATE`，例如 `bank_accounts` 上的 `customer_id`。属于 Gate，转入产品路径（第 9 章）。
脱敏	替换或遮掩值，使真实值不可见，而形态仍可使用。静态脱敏修改拷贝，动态脱敏修改查询返回的内容（第 12 章）。
已批准哈希	访问申请的 `approved_hash`：已批准语句集的哈希，取自经评审修正的合并记录或已批准的申请记录，绝不取自 runner 的输入（第 9、10 章）。
影响范围	若某个 principal、凭据或操作出错，可能波及的范围有多大。总是针对某个具体对象来表述（第 1、2 章）。
与引擎无关	本书用这个词指称不属于任何数据库引擎的记法和建议（前言）。
允许列表、拒绝列表	一份列出允许的内容，一份列出拒绝的内容。绝不使用“whitelist”或“blacklist”（第 7 章）。
遮蔽	从输出或日志中彻底移除某个值（第 12 章）。
执行	语句如何到达数据：由谁或什么提交，运行前是否有检查，什么限定其影响，以及如何提交（第 1、9 章）。
主库、副本	可写的数据库及其只读拷贝。分析副本不是副本（第 6 章）。
自动化 principal	不是人员的 principal：作业、runbook、ChatOps 命令、AI Agent。它拥有 workload identity 和默认只读，记录中同时写明它和触发它的人员（第 9 章）。
最小化	不收集、不复制、不保留无人需要的数据（第 12 章）。

术语	含义
最小权限	只授予任务所需的权限，且只持续到任务所需的时长，并将这样做的成本计算在内（第 1、3 章）。

其中有几组术语容易混淆，用一条简短的规则即可在争论开始之前定夺。

如果问题是	使用	不要用
一台只转发网络流量、不了解语句的主机	bastion	access gateway
签发按人会话登录并记录会话的中介	access gateway	数据库 proxy
对连接做池化或路由，与人员无关	pooler 或数据库 proxy	access gateway
人员自己的客户端通过 gateway 提供的连接发送语句	session handout	受控执行
人员提交语句而从不持有连接	受控执行	session handout
让凭据立即停止生效	撤销	轮换
谁来批准有风险的操作	四眼原则	绑定语句的审批
具体批准了什么，以哈希为准	绑定语句的审批	四眼原则
应用的业务记录	`audit_events`	审计日志

B. 模式速查

第 16 章按三大支柱为五种模式评分。这些评分是本书的判断，而不是测量结果：“强”表示该模式就是为证明这根支柱而构建的，“部分”表示它能证明其中一部分，或者只有在补充其他东西之后才能证明，“无”表示它在此保持沉默。成本一栏是第 16 章给出的内容，经过缩写。

模式	解决的问题	成本	授权	执行	问责	章节
直接访问（客户端到数据库）	登录确实存在，且其授予与当时的状态一致	无需构建成本；所有成本都在以后到来，表现为一周的翻查	部分：仅有授予，触达范围取决于网络允许什么	无	至多为部分：若有人预配，则按人登录，否则为无	1
bastion	一台受控主机，一个可以关闭网络路径的位置	需要打补丁并保持可用的主机	部分：收窄网络路径	无：它只转发，分不清 `SELECT` 与 `DROP`	部分：谁登录了主机，而不是运行了什么	7

模式	解决的问题	成本	授权	执行	问责	章节
session proxy 或 access gateway	一位具名人员在两个时间戳之间，在特定授予之下持有一个特定的会话登录	一个在有人需要时必须可用的新系统，以及随之而来的审批路径	强：按会话登录、到期、按请求授予	部分：受授予和受保护设置约束，语句可以自由键入	对谁和何时为强，对什么为部分	7、8
语句受控执行器	这组确切的语句（由其哈希标识）经批准、经检查、受约束并经过计数	每项任务都要等待评审，没有任何交互，而且必须有人构建并维护 runner	强	强：运行前先检查，并绑定到已批准哈希	强：语句、审批人和结果在同一条记录中	9、10
产品路径（无 SQL）	某项已定义的操作，为某个工具中的某个角色运行	每项任务一项功能，且是五种模式中变更周期最慢的	强：操作本身就是权限	强：操作是固定的	部分：人员出现在工具的审计轨迹中，数据库日志显示的是服务登录	9、10

在它们之间选择，是逐项任务的决定，所依据的规则就是第 16 章给出的那条：取仍能为该任务所危及的支柱提供证据的最便宜的模式。

```
function choosePattern(task):
    // a person who never opens a session cannot leave one open
    if task.repeats and task.hasFixedShape:
        return "product path"
    // a schema change takes the migration pipeline, never the runner
(Chapter 11)
    if task.changesSchema:
        return "migration pipeline"
    // Restricted columns and owner links are changed by the product,
not by SQL
    if task.writesRestrictedColumn or task.changesOwnerLink:
        return "product path"
    if task.writesData:
        // reviewed statement set, approved hash, runner executes
        return "statement-mediating executor"
    if task.readsData:
        return "access gateway: human_readonly on ledgerly_replica_2"
    // anything else is an exception with a named owner and an end
    return "exception: owner, reason, ends_on"
```

C. 危险操作矩阵

下面的矩阵是第 9 章的执行策略，单元格原样重复。行是分类器产生的语句类别，列是语句可能经由的通道。每个单元格有三种答复之一：deny（绝不通过此路径运行）、gate（需要尚未获得的审批，或者属于

另一条路线）、allow（在回路的限制之内运行；单元格中写有“绑定”的，只能在所注明来源的界限之内运行）。

语句类别	人员读取会话	JIT 写入	Break-glass	数据修正 runner	迁移
读取，在行数上限之内	Allow	Allow	Allow	Allow	Allow
有界写入，在已批准的计数之内	Deny	Allow，绑定到该修正（仅限 runner）	Allow，绑定到行数上限（需附说明）	Allow，绑定到已批准哈希和计数	Allow，绑定到回填规则
无界写入	Deny	Deny	Deny	Deny	Gate：指名审批人
所有者关联变更	Deny	Gate：产品路径	Gate：产品路径	Gate：产品路径	Gate：指名审批人
写入 Restricted 列	Deny	Gate：产品路径	Gate：产品路径	Gate：产品路径	Gate：指名审批人
导出	Gate：按层级	Deny	Deny	Deny	Gate：指名审批人
模式变更（DDL）	Deny	Deny	Deny	Deny	Allow：经评审的通道
权限变更	Deny	Deny	Deny	Deny	Gate：平台评审
写入 `ledger_entries` 或 `audit_events`	Deny	Deny	Deny	Deny	Gate：平台评审

三条阅读规则能省去大部分查表。

- **落在人员手中的写入，要么被拒绝，要么受绑定。**人员读取会话没有写入路径，JIT 写入只来自 runner，而 break-glass 是人员直接写入的唯一通道，须在行数上限之内并附有说明。
- **Gate 会指明语句接下来去往何处。**“产品路径”指 `admin-console` 请求 `payments-service`，由第二个人审批，并将变更记录在 `audit_events` 中。“指名审批人”和“平台评审”指迁移通道把该语句标记出来，交给必须表态同意的人。
- **矩阵不是授权边界。**查询分析减少错误。授权边界仍是登录上的授予，分类器误读的语句依然受它们限制（第 9 章）。

对一条语句的决定，写成规则如下：

```
function answerFor(statementClass, lane):
    // the matrix above is the only source; no lane has its own rule
set
    cell = matrix[statementClass][lane]
    if cell starts with "Deny":
        refuse and record the refusal
    else if cell starts with "Gate":
        // the text after the colon names the route or the approver
        hold the statement; route it as the cell says; do not run it
as sent
    else:
        // "bound" cells compare the real count with the bound before
COMMIT
        run within the loop's limits
```

D. 合规对照表

数据保护和支付卡制度往往会就访问提出一系列问题。在其中某一制度下工作的审计人员，通常想看到：控制存在，它运行过，并且有人查看了它的结果。这份对照表列出各类问题、回答每个问题的控制、它留下的制品以及构建它的章节。它是第 14 章那张表的重复，并不声称任何控制满足任何义务。

问题	控制	制品	章节
谁可以触达生产数据，经由哪条路径？	以 gateway 为唯一的门、以代码管理的角色 schema、漂移检查	网络规则清单、角色定义、漂移报告	3、7
谁以哪些授予和审批运行了某条语句？	带有申请 id 和审批人的 `access_audit_record`	该记录以及 `verifyChain` 报告	14
提权访问持续多久，由谁审批？	带有审批人和 `expires_at` 的 JIT 申请、撤销作业	申请记录、撤销日志、逾期授予检查	8
如何识别并限制敏感数据？	schema 中的分类、脱敏、扣留的列	目录、脱敏加载的验证	12
生产数据的拷贝在哪里，各归谁负责？	带有负责人和终止日期的登记册、清理作业	登记册、清理作业日志	6、13
谁更改 schema，由谁评审？	迁移流水线、受保护分支、外部启动条目	与提交和审计条目相匹配的迁移记录	11

问题	控制	制品	章节
访问是否仍然需要?	由具名评审人执行的访问评审	带有各项决定的已完成评审	14
紧急情况下会发生什么?	带有告警和强制评审的 break-glass	评审记录、告警历史	8、15
谁来管理访问本身?	作为独立生产者的 `ledgerly_access_admin` 登录	它们的流，以及目录与记录的比对	3、14

存在却从未运行过的控制，或者运行过却无人查看的控制，只回答了问题的三分之一。下面的检查，应在把制品交给任何索取者之前运行。

```
function evidenceStatus(row):
    // three things an auditor tends to want, from Chapter 14
    if row.artifact is null:
        return "control not evidenced: build or find the artifact"
    if row.artifact.lastProducedAt is null:
        return "control exists, no evidence it ran"
    if row.artifact.lastReviewedBy is null:
        return "control ran, results not looked at"
    return "evidenced: attach artifact and its review"
```

E. 访问评审模板

这是第 14 章 `runAccessReview` 背后的模板。每个评审周期复制一份。评审只提出建议，由具名评审人做出决定；任何人都不评审自己的访问，评审期间要长于被评审 workload 最长的周期（为期一个月的评审会建议撤销季度报表使用的某项授予，而该报表每三个月中有两个月会用到它）。

表头

```
Review cycle:            <period start> to <period end>
Reviewer(s):             <names; none is the owner of an item they
review>
Record queried on:       <date>, verifyChain result: <passed | failed:
details>
Period chosen and why:   <longest workload cycle covered?>
```

第 1 部分：声明的授予与实际使用

授予（角色、权限、表）或登录	负责人	记录中的最近一次使用	发现项	决定	决定人	完成日期
`<role>`、`<privilege>`、`<table>`	`<named principal>`	`<date or none>`	未使用 / 无负责人 / 使用范围窄于授予 / 正常	保留 / 收窄为 `<privilege>` / 撤销 / 指定负责人	`<reviewer>`	`<date>`

按权限匹配使用情况，而不只是按表。一个每晚执行 select、却从未执行 update 的角色，是从 `S, U` 收窄到 `S` 的候选。纳入身份提供方的组到角色的映射，以及 `ledgerly_access_admin` 的登录。

第 2 部分：不属于授予的事项

事项	问题	答复	负责人	决定
数据拷贝登记册（第 13 章）	每份拷贝是否仍有负责人和一个未来的终止日期?	`<yes / no, which>`	`<name>`	`<keep / end by date>`
谁可以审批 JIT 申请	每位审批人是否仍处于能证明其资格的角色?	`<yes / no, which>`	`<name>`	`<keep / remove>`
谁可以读取审计存储	每位读取者是否仍然需要?	`<yes / no, which>`	`<name>`	`<keep / remove>`
自动化 principal（作业、runbook、AI Agent）	每个是否仍在运行，并具备 workload identity 和默认只读?	`<yes / no, which>`	`<name>`	`<keep / retire>`

第 3 部分：评审期内的 break-glass 使用

申请 id	使用者	事故或原因	评审人	评审日期	关于系统的发现
`<id>`	`<principal>`	`<reason>`	`<other than user>`	`<date or MISSING>`	`<for example: missing path>`

没有评审日期的一行，本身就是一项发现。

签核规则

```
function closeAccessReview(review):
    for each row in review.rows:
        // a row without a decision is not a review, it is a form
        if row.decision is null or row.decidedBy is null:
            raise ReviewIncomplete
        if row.decidedBy == row.owner:
            raise ReviewerIsOwner
    // decisions become tracked work with an owner and an end date
    for each row in review.rows where row.decision != "keep":
        createChange(row, owner: row.owner, endsOn:
review.nextCycleStart)
    markClosed(review.id)
```

F. 事故后评审模板

本模板遵循第 15 章。它是无责的：每个问题都针对允许某项操作发生、或让它不可见的系统，没有一个问题追问谁本应知道得更清楚。应依据记录填写，而不是凭记忆。记录无法回答某个问题时，就写明这一点，因为缺口本身就是一个发现项。时间以记录所显示的为准，不要估计。

1. 概要

```
Incident id:         <id carried in every request reason>
Opened / closed:     <timestamps from the incident record>
Trigger:             <alert, report or declared exposure>
Principal(s) in scope: <logins, services, automation principals>
Pillars touched:     <authorization / execution / accountability>
```

2. 遏制，按顺序

步骤	完成时间	结果	阻碍因素说明
撤销授予	`<timestamp>`	完成 / 失败	
关闭会话	`<timestamp>`	完成 / 失败	
轮换凭据	`<timestamp>`	完成 / 失败	
依据审计日志界定损害范围	`<timestamp>`	完成 / 失败	
记录并通知	`<timestamp>`	完成 / 失败	

问题：即使前面某一步失败，每一步是否仍然运行了？是否有步骤未完成，是否对其进行了跟踪？日志是否可以访问；若不可以，条目是否在排队，且没有拖延访问的撤除？

3. 影响范围

```
Query run:           <everything the principal touched since T,
Restricted rows per table>
Tables touched:      <list, with class>
Restricted columns:  <touched or not, rows returned per table>
Agreement check:     <does the scope agree with the register of copies
and the gateway session list?>
```

4. 事故访问的生命周期

申请 id	类型	申请时间	授予时间	首次使用	撤销时间	评审人
`<id>`	jit / breakglass	`<time>`	`<time>`	`<time or none>`	`<time>`	`<other than user, or MISSING>`

问题：是否有授予超过了 `expires_at`？是否有访问被授予，却无法用任何申请解释？`closeIncident` 是否发现了未关闭的访问，且该访问是否在事故关闭之前就已关闭？

5. 事故期间的写入

写入	路线（带加急评审的 runner、产品路径、原始 break-glass）	非作者本人的评审人	已批准哈希是否匹配
`<description>`	`<route>`	`<name>`	是 / 否

问题：在 runner 本可胜任的情况下，是否对某次写入使用了原始 break-glass，原因是什么？

6. 延后处理的情形（如果发生）

- **Gateway 不可用**：人员访问中断了多久，服务是否持续运行，恢复 runbook 是否照其所写发挥了作用。
- **事故期间的恢复**：恢复出的拷贝是否登记了负责人和终止日期，并已终止。
- **夜间审批缓慢**：哪项申请在等待，在等谁，以及针对未答复申请的规则起了什么作用。

7. 发现项与后续跟进

关于系统的发现项	部分（身份、授予、路径、执行、记录）	变更	负责人	终止日期
`<for example: a path that should have existed>`	`<part>`	`<change>`	`<name>`	`<date>`

最后做一项检查：每一个发现项，是否都是对某个设计的变更，且有负责人和终止日期，而不是一句“要更加小心”的提醒？

关于本书

生产数据库访问不是凭据分发问题，而是授权、执行与问责问题。

更好的密码管理器只改变机密送达某个人的方式。它并不决定这个人可以做什么，不决定对数据实际执行了什么，也不决定事后由谁来负责。本书跟随 Ledgerly，从一个共享的管理员密码走到一套经过刻意设计的访问模型：最小权限身份、即时访问与 break-glass 访问、针对危险操作的受控执行，以及一条能够说明谁做了什么、为什么这样做的审计轨迹。这些模式与数据库引擎无关，每一种都连同它所带来的成本一并给出。

关于作者

Muhammet Şafak 自 2008 年起从事软件开发，多年来一直在医疗、金融和电信领域一线任职。Muhammet 主张开发应当聚焦于问题，而不是技术。自 2024 年起，Muhammet 设计并构建可端到端扩展的高流量系统。

Muhammet 为中小型软件开发团队提供指导，主导开源项目，并把每一项决策连同其理由与代价一并写下来。

可靠的系统并非由可靠的部件构成，而是围绕会失效的部件构建而成。

I 访问

II 事件

III AGENT