收租之王，负债上牌桌

导读

甲骨文为什么疯狂借钱建云和 AI，市场却半信半疑

先做一道算术题。假设你开了一家店，卖的东西复制一份的成本几乎是零——软件就是这样，第一份写出来花了钱，第二份、第一百万份几乎白送。更妙的是，客户得先付钱、先签多年合约才能用。你不用备货，不用盖厂房，钱先到账，货几乎不花成本。这是人类商业史上最舒服的生意之一。甲骨文（Oracle）当了三十年这样的房东，靠数据库的许可证收租，躺着就把钱收了。

那么问题来了：**一个躺着收租的房东，为什么突然要去借上千亿美元的债，把钱砸进一栋栋塞满 GPU 的数据中心，还把身家押在一个叫 OpenAI 的客户身上？**这不是换个花样收租，这是把整套生意的底层规则掉了个头——从「近乎零成本、先收钱」的旧生意，跳进一门「先烧掉几百亿、买一堆两三年就贬值的芯片、再等客户慢慢付钱」的新生意。旧生意的护城河是切换成本：数据搬起来太痛，客户跑不掉。新生意的护城河是什么，甚至还有没有护城河，没人说得准。

这本书要替你算清楚的那笔账

市场的反应是「半信半疑」——股价一边涨一边有人喊泡沫，两拨聪明人看着同一张财报得出相反结论。这不是因为谁蠢，而是因为这笔账真的难算：有一个叫 RPO（剩余履约义务，Remaining Performance Obligations）的数字暴涨到约四千五百五十亿美元，听起来像天量订单，但它不是躺在账上的现金；自由现金流已经转负，毛利率被折旧和电费一点点压薄；更麻烦的是，撑起这个故事的算力合同、芯片供应、融资，几方之间还绕着一个让人起疑的循环。

你不需要是财经专家才能看懂这些。你只需要愿意跟着一条因果链往下走。**这本书就是一条链，从收租之王的旧账开始，一环扣一环剥到今天的债务和赌注**——每一章都在回答上一章留下的疑问。为什么当年错过了云？错过之后为什么敢自己从头盖一套；盖出来的东西凭什么和别人不一样；这套东西遇上 AI，赌注为什么被放大了十倍；放大之后，账面上那些漂亮数字和口袋里的现金差在哪里；把宝押在一家客户身上，是胆识还是一颗雷。

我不会替你喊涨或喊跌。这本书的目标不是给你一个结论，而是给你一套能自己下判断的工具：一张护城河的体检表，让你自己看清楚甲骨文哪里在变硬、哪里正在悄悄塌方；以及一

个能套用到任何一场 AI 基建豪赌上的框架——因为甲骨文只是这场豪赌里最典型的一个标本。**「数据我替你保管」和「数据是我的」，是两回事**；「有订单」和「有现金」，也是两回事。看懂了这两个区别，你就看懂了这本书的一大半。

> 读完之后，下次再有人跟你说「谁谁谁又签了几千亿的算力大单」，你会本能地追问一句：这是收进来的钱，还是借出去的赌注？

第一章 · 收租之王

要弄懂甲骨文今天为什么敢背上上千亿美元的债去建数据中心、去接 OpenAI 的天量订单，你得先弄懂它过去三十年过的是什么日子。那是一种大多数公司做梦都想过、却几乎没人能过成的日子：**躺着收租**。这一章我们就把这门「收租生意」拆开,看清楚它的每一个齿轮是怎么咬合的。因为后面全书要反复回来对照的那个基准——「旧甲骨文有多爽」——就建立在这里。

先说清楚它卖的是什么

甲骨文的地基是一个叫 关系数据库 的东西。别被名字唬住,它的核心思想特别朴素:把数据存成一张一张的表格,每张表有行有列,像 Excel;不同表之间靠共同的字段互相关联(所以叫「关系」)。比如一张「客户表」和一张「订单表」,通过「客户编号」这个共同列勾连起来,你就能问出「张三这个月下了几单、总共花了多少钱」这种问题。

> 大白话
>
> 你可以把关系数据库想成一家公司的**中央账本**。银行的每一笔转账、航空公司的每一个座位、电商的每一张订单,最终都要落到这个账本上。它不炫,但它是命根子——账本错一行,可能就是几百万美元的差错,或者一架飞机上卖出两张同一个座位的票。

这里有个关键词叫 事务(transaction):一组「要么全都成功、要么全都别发生」的操作。经典例子是转账——从你账户扣 100 块、往对方账户加 100 块,这两步必须捆成一个不可分割的整体。要是扣完钱系统崩了、加钱没成功,那 100 块就凭空蒸发了。数据库能不能死死保证这种「原子性」,是它值不值钱的分水岭。甲骨文早年就是靠把这件难事做得又快又可靠,拿下了银行、电信、航空这些「账错一分钱都要命」的大客户。

为什么这门生意像收房租

现在讲钱是怎么进来的,这是理解全书的第一块基石。甲骨文卖数据库,不是像卖一台冰箱那样一手交钱一手交货、交完两不相欠。它卖的是 许可证(license)——本质上是一纸「使用权」。这纸使用权的收费结构,才是收租的精髓所在,它分成两截:

- **第一截:前期许可证费。** 客户第一次买,按你要用几颗 CPU 核心、几个用户来算,一次性掏一大笔。这笔钱数目吓人,但它是一锤子买卖。
- **第二截:每年的「软件维护支持费」。** 这才是房租。行业惯例是每年再交前期许可证费的 **约 22%**,换来补丁、安全更新和技术支持。注意这个数字:22% 意味着大约**四年半**,客户光是交的维护费就又等于当初买断的价钱——而且只要他还在用,这租金就年复一年地交下去,几乎没有尽头。

> 大白话
>
> 翻译成人话:客户等于先付了一大笔「首付」买下使用权,然后每年还要交一笔「物业费」。而这笔物业费的妙处在于——甲骨文这边几乎不用再增加什么成本。软件不是钢筋水泥,复制一份的边际成本约等于零。同一套代码卖给第一万个客户,和卖给第一个客户,公司多花的力气微乎其微。所以这笔维护费,绝大部分是**纯利润**。

这就是「收租之王」这四个字的字面来源:**钱一旦进来,就变成一条每年自动续费、成本极低、极其稳定的现金流。**它不像卖硬件那样每一台都要重新付出原料和制造成本,它更像你买下一栋楼收租——楼是软件,租客是客户,物业几乎不花钱,而租约长得看不到头。请记住这条现金流的形状:又稳、又肥、又几乎无成本。全书后面讲甲骨文转去卖 GPU 算力时,你会看到这条曲线是怎么被一点点掰弯的。

护城河的第一块石头:切换成本

你可能会问:软件维护费收这么狠,客户为什么不换一个便宜的?这就问到了甲骨文这门生意最硬的那块地基——切换成本,也就是「想换掉它得付出的代价」。

想象一家银行的所有业务系统都架在甲骨文数据库上,跑了十几年。要换掉它,不是卸载一个软件再装一个那么简单。它意味着:把几十 TB 的数据一条不错地迁走;把成千上万条依赖甲骨文特有语法写的程序全部重写;在切换的那个周末,你得让整个银行的账本「停机搬家」还不能出一分钱的差错;然后再祈祷新系统上线第一天不要崩。

大白话

这就好比给一栋住满了人、还在营业的三十层大楼换地基。理论上能换,但风险高到没人敢批这个预算。对客户的 CTO 来说,继续每年交那 22% 的租金,是**两害相权取其轻**的理性选择——换数据库可能让他丢掉工作,交租金只是让公司多花点钱。

于是甲骨文获得了一样极其值钱的东西:定价权——涨价而不太怕客户跑掉的能力。因为客户被「锁」住了,搬家的痛苦远大于涨价的痛苦。这块「锁定」的石头具体是怎么砌进客户系统里的,是下一章的主题;这里你只要先记住结论:**甲骨文的租金之所以收得稳,不是因为它便宜、也不完全是因为它好,而是因为客户走不掉。**

还有一支把这门生意打进每家公司的铁军

好产品加上高切换成本,还不足以解释甲骨文的统治力。第三块拼图,是它那支江湖闻名的 销售铁军——一支纪律严明、极度好斗、按业绩拿钱的直销队伍。

企业软件和你在 App Store 里点一下就买的软件不一样。一单动辄几百万上千万美元,买方是大公司里一层层的 IT 主管、采购、CFO。这种单子不会自己成交,得有人一年到头飞过去,请客、演示、算账、施压、逼单。甲骨文早年就是靠把这支队伍训练成一台冷酷高效的机器,一家一家地把数据库塞进世界上最大那批公司的机房里。

这里藏着一个容易被忽略的因果链,值得单独点出来:

1. 销售铁军负责把客户**第一次**拉进门——签下那笔前期许可证。
2. 切换成本负责让客户**再也走不掉**——于是那 22% 的年租金能一年年收下去。
3. 而这条几乎纯利润的租金现金流,又**回过头养着**这支昂贵的铁军去攻下一个客户。

三者咬成一个自我加强的飞轮:**先靠人把客户签进来,再靠锁定把客户变成年金,再用年金养更多的人。**转得越久,轮子越沉、越难被对手推动。这就是为什么在长达二三十年里,甲骨文的日子好到近乎不真实——高毛利、高续费率、强定价权,是软件行业里最接近「印钞机」的一种形态。

把这一章钉成一根基准线

我们花一章讲一门「旧」生意,不是怀旧。是因为后面十五章里,每当我们要判断甲骨文今天这场豪赌到底是聪明还是危险,都得回头拿它跟这根基准线量一量。所以现在把「旧甲骨文」的几个数值刻在脑子里:

- **成本结构:** 卖出去之后几乎不再花钱,复制软件的边际成本约等于零。
- **现金流:** 又稳又肥,大量维护费是纯利润,基本不用先垫一大笔钱出去。
- **风险的单点:** 主要在「客户会不会被撬走」,而切换成本让这件事极难发生。
- **护城河:** 一部分来自技术,更大一部分来自锁定和那支铁军。

记住这四条。因为当甲骨文掉头去建云、去买成千上万块 GPU、去接 OpenAI 的订单时,这四条会被逐一挑战:边际成本不再是零了(GPU 要折旧、机房要吃电),现金流要先大把大把砸出去才可能收回来,风险的单点从「客户流失」变成了「一个巨型客户+一屁股债」。

> 收租之王最舒服的地方,是它**先收钱、后几乎不花钱**。而它现在要做的事,恰恰是反过来的——**先花掉海量的钱,再祈祷未来能把租收回来。**这本书剩下的部分,就是在算这笔倒过来的账。

第二章 · 锁进去就出不来

上一章我们说，甲骨文这门生意像收房租——签下许可证，客户年年续费，钱像自来水一样流进来。但房东能安稳收租，前提是租客搬不走。这一章我们就把这句话拆开：一个用了甲骨文数据库的公司，为什么想换换不掉？这个"换不掉"，具体锁在哪几个螺丝上？把这几颗螺丝看清楚了，你就摸到了整座护城河最底下、砌得最早的那块石头。

先说清楚：数据库到底是个什么东西

数据库，你可以先粗暴地理解成一个"特别讲规矩的仓库管理员"。你把公司所有要紧的数据——订单、客户、账目、库存——都交给它保管。它负责三件事：把数据摆好、你要的时候快速找出来、以及保证在任何时候数据都不会自相矛盾。

甲骨文卖的是一种叫 关系数据库 的东西。"关系"两个字听着玄，其实就是说数据都摆成一张张表格：一张表存客户，一张表存订单，订单表里记一个"客户编号"指回客户表。表和表之间靠这种编号勾连起来，这就是"关系"。你跟它打交道用的语言叫 SQL（读作"sequel"），一种专门用来查表、改表的语言，比如"把所有欠款超过 30 天的客户挑出来"。

> 大白话
>
> 为什么这事这么要命？因为对一家银行、一家航空公司、一家电商来说，这个仓库管理员管的不是普通杂物，是命根子。转账转到一半崩了、机票同一个座位卖给两个人、双十一订单对不上账——任何一个都是灾难。所以谁一旦选定了一个靠谱的管理员，就极不愿意再换。

换一个核心数据库，难在哪

工程师最容易犯的一个直觉错误是：数据库不就是个存数据的地方吗？把数据从甲骨文导出来，再导进另一个数据库，不就搬完了？如果真这么简单，甲骨文早就没护城河了。真正难的地方，从来不是搬数据本身，而是下面这几层缠绕。

第一层：SQL 长得像但脾气不同

SQL 名义上有国际标准，听起来各家数据库应该能互换。但现实是每家都在标准之上加了自己的一堆"方言"和独门函数。甲骨文尤其如此——它有大量只有它才认的语法、只有它才有的函数、还有一种叫 PL/SQL 的东西（甲骨文自己的编程语言，让你能把一整段业务逻辑直接塞进数据库里跑）。

一家公司用了甲骨文十几年，往往在数据库里堆了成千上万行 PL/SQL——发工资的逻辑、算利息的逻辑、月底结账的逻辑，全写在里面。换数据库，意味着这些代码一行都不能直接搬，得用新数据库的语言重写一遍。而且这些代码常常是十几年前某个已经离职的人写的，没人敢保证自己完全看懂了它在干嘛。

第二层：应用程序在假设"底下是甲骨文"

数据库不是孤零零一个仓库，它上面压着一整栋楼——公司所有的业务系统都建在它之上。这些系统在写的时候，程序员会有意无意地依赖甲骨文的种种特性：它的性能脾气、它处理并发的方式、它某个特定行为的细节。你把地基换掉，上面这栋楼的每一层都可能咯吱作响，出现谁也没预料到的裂缝。

> 大白话
>
> 打个比方：这就像你想给一栋住了人的大楼换地基。理论上地基是地基、楼是楼，可以分开。实际上楼里所有的水管电线都是按老地基的位置铺的，你一动地基，得把整栋楼的水电重走一遍，还得保证换的过程中住户水电不断。

第三层：一次都不能错，还不能停

这才是最狠的一层。银行的核心数据库不能说"我们周末停三天，搬完再营业"。它必须一边正常收钱转账，一边把自己搬到新家，而且搬完之后，一分钱都不能对不上。

这里有个关键词叫 迁移（migration，就是把数据和逻辑从一个系统搬到另一个系统的整个工程）。核心系统的迁移，几乎从来不是"搬一次"，而是要让新旧两套系统并行跑上几个月甚至几年，一点点把流量从老系统切到新系统，随时准备出问题就切回来。这中间任何一次对账不平，都可能让整个项目推倒重来。

把这些难处加起来，等于什么

现在我们把上面三层难处翻译成一个 CTO 桌上的决策题。他要换掉甲骨文，需要付出的代价大致是：

- **一大笔一次性成本**：重写所有 PL/SQL、改造上层应用、买新硬件、雇一支专门团队，干上一两年。对一家中大型企业，这个数字量级上常常是几百万到上千万美元。
- **一段高风险窗口**：迁移期间系统更容易出故障，而故障砸的是最核心的业务——收钱的、结账的。
- **一个几乎看不见的收益**：就算全部搞成了，换来的也不过是"和以前一样能用"，业务上没有任何新增价值，顶多省下点许可证费。

这道题的答案，对绝大多数公司来说是显然的：**不换**。这就是经济学里说的 切换成本——不是产品本身有多贵，而是"离开它"这个动作本身太贵。切换成本一高，客户就被钉在原地，动弹不得。

切换成本怎么变成了定价权

护城河的故事到这里才讲到一半。锁定本身只是让客户走不了，甲骨文真正厉害的一步，是把"客户走不了"这件事，翻译成了自己账本上的钱。这个翻译器，就叫 定价权——你有本事涨价，而客户明知道被涨了，也只能咽下去。

逻辑链条是这样的：客户被锁定 → 换掉甲骨文的代价是一千万、还得冒系统崩的风险 → 那么只要甲骨文每年涨价、加各种费用，只要这笔账加起来还没超过"一千万加风险"这条线，客户理性的选择就是接着付。甲骨文很清楚这条线在哪，于是就把价格顶到这条线附近。

> 大白话
>
> 说得再直白点：搬走要花一千万，那我一年多收你两百万的费用，你也只能忍——因为忍五年才等于搬一次的钱，而且搬还有搬崩的风险。这不是抢，这是精确地按你的"逃跑成本"给你定价。销售铁军最擅长的谈判筹码，本质上就是替你算这笔账，然后把价格顶到你不得不留下来的那个点。

甲骨文把这种定价权用到了极致，尤其体现在两件事上。一是 软件审计——它有权来查你到底用了多少、有没有超出许可证范围，一查往往能查出一笔"欠费"，逼你补交或升级到更贵

的套餐。二是它复杂到出名的许可证条款，比如按 CPU 核心数收费、虚拟化环境怎么算等等，复杂到客户自己都算不清自己该付多少，而算不清的时候，吃亏的总是客户。这些都不是数据库技术本身，而是把"你走不掉"这个事实，变现成现金流的一整套机器。

这块石头，砌得有多牢

回头看，甲骨文护城河最底下那块石头，是这样砌成的：数据库管的是企业命根子，所以选型极其谨慎、极其黏人；而它的技术特性、专有语言、和上层应用的深度缠绕，又让"离开"这个动作贵到不合算；最后，它用审计和复杂条款，把这份"离开太贵"精确地换算成了自己的涨价空间。锁定、切换成本、定价权——三者环环相扣，缺一不可。

但请记住一件事，我们后面几乎每一章都会回来敲打它：这块石头之所以牢，靠的是一个前提——**数据非在你自己家的机房里、你自己买的甲骨文数据库上跑不可**。整套锁定，锁的是"客户拥有并运维一个甲骨文数据库"这个动作。

那么，一旦世界变了——数据不再养在自家机房，而是按小时租用别人云上的服务；一旦上层应用可以整个打包换成别人家的 SaaS，压根不在乎底下是哪个数据库——这块石头下面的地基，还稳吗？这正是从第四章开始，我们要一层层撬开的问题。护城河最初的那块石头砌得再牢，也架不住有人把它脚下的地给挖走。

第三章 · 并购机器

上一章我们停在一个客户身上：一旦核心业务跑在甲骨文的数据库里，迁走的成本高到让你宁可年年交钱。那是一块石头。但一块石头砌不成城墙。这一章讲甲骨文怎么把这块石头变成一整道墙——不是靠自己一行一行写代码长出来，而是靠**买**。谁在企业软件这条街上开了家赚钱的铺子，它就把整家铺子连人带客户一起端走。

为什么是买，而不是自己写

先问一个工程师会问的问题：软件公司想扩张，为什么不自己写？答案很实在——自己写要时间，写完还得从零开始抢客户，而客户已经在别人手里了。收购是一条捷径：你花一笔钱，直接买到的不是代码，是**已经签了字、每年准时续费的客户名单**。

> 大白话
>
> 把它想成开餐厅。你可以在隔壁新开一家、慢慢等人上门；也可以直接把隔壁那家天天满座的老店买下来，招牌一换，第二天中午照样满座。甲骨文选的是后者——而且一口气买了一条街。

甲骨文买公司的算盘还有一层特别之处。企业软件有个性质：它的客户黏性极高，切换成本高。所以一家成熟软件公司的老客户，就算你把它买下来、砍掉研发的新花样、只维持系统能跑，这些客户短期内也走不掉，还得继续交维护费（每年按许可证价格的百分之二十上下收的"保修+更新"费，几乎是纯利润）。这意味着一家老软件公司即使不再增长,它的存量客户也是一台稳定的印钞机。甲骨文看透了这一点：它买的往往不是成长故事，是**现金流**。

第一枪：抢来的 PeopleSoft

PeopleSoft是一家做企业应用的公司——所谓企业应用，就是跑在数据库上面那一层、给人用的业务软件，比如发工资的人力资源系统、管钱的财务系统。它和甲骨文的数据库是"楼上楼下"的关系：数据库是地基和水管，企业应用是住人的房间。

2003 到 2004 年，甲骨文对 PeopleSoft 发起敌意收购——就是不经对方董事会同意、直接绕过管理层去向股东出价买股票，硬抢。这场仗打了约十八个月，闹到法庭，最后甲骨文以大约 **103 亿美元**拿下。为什么非它不可？因为 PeopleSoft 手里那批大企业客户，正是甲骨文想锁进自己生态的那种客户。买下来之后，甲骨文既拿到了这批客户的应用层，也顺手保住了他们脚下那些数据库不被对手撬走。

大白话

这一枪打的是一个信号：甲骨文不再满足于当"卖水管的"。它要往上爬，占住客户天天点开来用的那些界面。水管你可能忘了它存在，但发工资的系统你每个月都得打开——占住它，客户就更难离开。

Sun：把地基也买下来

如果说 PeopleSoft 是往上买（应用层），那 2009 到 2010 年花大约 **74 亿美元**收购Sun Microsystems就是往下买。Sun 是一家做服务器硬件的公司，同时手里攥着两样甲骨文眼馋的东西：Java（一门被无数企业软件用来编写程序的语言，等于半个行业的通用语）和MySQL（一个流行的开源数据库，理论上是甲骨文自家收费数据库的竞争者）。

买 Sun 这一手，把甲骨文的野心暴露得很清楚：它想从上到下自己全包。往上有应用，中间有 Java 这层通用语言，下面有数据库，再下面现在连服务器硬件都自己造。后来甲骨文那套引以为傲的Exadata（把数据库软件和专门为它调好的硬件打包卖的一体机，后面章节会细讲）能成立，靠的就是这次把硬件能力收进门。

> 横向买应用是把墙加宽，纵向买硬件是把墙加深。甲骨文两个方向一起动手，目的只有一个：让客户的整套系统从界面到芯片都长在自己身上。

连环收购是一门可复制的手艺

PeopleSoft 和 Sun 只是最显眼的两枪。二十年里甲骨文吞下的公司数以十计——Siebel（做客户关系管理）、Hyperion（做财务分析）、BEA（做中间件）等等，量级上加起来是几百亿美元的买买买。这不是一时冲动，是一套流水线：

1. **盯现金流，不追热点。**专挑那种有一大批黏性客户、每年稳收维护费的成熟公司，而不是烧钱的明星初创。
2. **买完就整合进自家账单。**把对方客户接进甲骨文的销售和维护体系，让"续费"这件事变成甲骨文统一收。
3. **砍掉重叠、留住合同。**研发和后台能合并的合并，但那些签好的长期合同一分不动——因为合同才是买它的理由。

> 大白话
>
> 说白了，甲骨文把"收购"本身做成了一条产品线。别的公司收购是偶尔为之的大事，它是常态操作：看到一台还在转的印钞机，估个价、买下来、接进自己的收款系统，下一台。华尔街甚至给它起了个外号——企业软件的"收割机"。

Cerner：同一套手艺，换个更大的猎场

把时间快进到 2022 年，甲骨文花了大约 **283 亿美元**——它历史上最大的一笔收购——买下Cerner，一家做医院信息系统（电子病历、医嘱、排班这些）的公司。表面看这是老手艺的又一次施展：Cerner 手里握着大量医院客户和多年合同，典型的黏性现金流。

但 Cerner 和前面那些不一样，多了一个后面几章要反复回来的关键词：**数据**。医院系统里躺着的是海量病历——这是一种极难迁移、法律上又被重重保护的资产。甲骨文买 Cerner，买的不只是又一批锁进来的应用客户，还是一座**数据金矿**。在 AI 时代，谁手里有别人拿不到、又搬不走的真实数据，谁就可能握着一种新的护城河。这颗种子这一章先埋下，第十四章我们会专门回来问：这些数据究竟能不能变成 AI 时代最硬的那块壁垒，还是只是"被托管的比特"。

这一章把护城河推到哪儿了

回头看这条线：第一章讲甲骨文靠数据库收租，第二章讲单个客户被锁在数据库里出不来。这一章要说的是——甲骨文没有停在"锁住数据库"，它用连环收购把锁定从地基一路铺到了整栋楼。你的发工资系统、财务系统、客户管理、连服务器硬件、甚至医院的病历，都可能落在甲骨文的地盘上。锁定的点从一个变成很多个，客户想整体搬家，等于要同时拆掉好几套彼此咬合的系统——难上加难。

但这里要留一个诚实的问号，它会贯穿全书。买来的城墙和自己长出来的城墙，硬度未必一样：整合几十家公司会留下技术上的裂缝，用债务和现金去买增长也有代价——这台"并购机器"越转越大，本身也在往甲骨文身上堆分量。等到后面它要一头扎进云和 AI、开始上千亿地借钱时，你会发现，"用买的来扩张"这套从二十年前就练熟的肌肉记忆，正是理解它今天为什么敢背上巨债的伏笔。护城河在应用层确实被拓宽了；至于它是变得更硬、还是在别处开始出现裂缝，我们接着往下看。

第四章 · 错过的那班车

把时间拨回到 2006 年。那一年甲骨文正处在它历史上最舒服的位置：数据库是硬通货，销售铁军所向披靡，收购机器把 PeopleSoft、Siebel 一个个吞下去。账上现金像自来水一样往里流。

也正是那一年，一家网上书店对外开放了一项奇怪的服务，叫「弹性计算云」。当时几乎没有一个甲骨文的高管把它当回事——一个卖书的，凭什么教企业怎么买服务器？

这一章要回答一个让人别扭的问题：**一门这么赚钱、这么稳的生意，为什么偏偏在下一班车来的时候，没上去。**不是它看不见,而是它太赚钱了,以至于上车这件事本身对它有害。

先说清楚，「那班车」到底是什么车

在云出现之前，一家公司想跑一套甲骨文数据库，流程大概是这样的：先找甲骨文签一张许可证（License，就是「允许你在自己机器上运行这套软件」的书面授权），一次性掏一大笔钱；然后每年再交一笔维保费（约等于许可证价格的 22%，换来打补丁和技术支持）；与此同时，你还得自己花钱买一屋子服务器，租机房、拉电、装空调、雇一队工程师看着它们别宕机。

软件的钱进了甲骨文口袋，硬件和机房的钱进了别人的口袋，但麻烦全是你自己的。

云干的事，就是把这一整套**买断+自建**，换成**按小时租**。

> 大白话
>
> 打个比方。以前你要用电，得自己在院子里盖一座小发电机房：买机器、囤柴油、请人维护，用不用都得养着。云来了，相当于电网通到了你家门口——你只管把插头插上，用多少度电付多少钱，不用了拔掉就不再计费。发电机、变压器、维护工，全在电网那头，跟你没关系了。

这里有两班车，方向不太一样，得分开说，因为它们从两个不同的地方掏空了甲骨文的老生意。

第一班车：AWS/Azure —— 把「机房」变成按小时租

IaaS（Infrastructure as a Service，基础设施即服务）是这班车的正式名字。翻成人话：**别人替你建好机房和服务器，你按小时租用算力和存储。**亚马逊的 AWS 是头一个把它做成大生意的，微软的 Azure 紧随其后。

它厉害在哪？不在技术多玄，而在于它把一件「要下重注、要等很久」的事，变成了「几乎没有门槛、随时反悔」的事。这里的关键词是两个：

- **资本开支变成了运营开支。**以前你要用数据库，得先砸几百万买服务器（这叫资本开支，CapEx——一次性买下一件能用好几年的大件资产，钱先出去，价值慢慢摊）。现在你不买了，改成每个月付租金（这叫运营开支，OpEx——像水电费一样，用多少付多少）。对一家创业公司来说，这是天壤之别：以前要先融一大笔钱买机器才能开张，现在刷张信用卡，半小时后就有一台服务器在跑。
- **弹性。**双十一流量涨十倍，你临时多租十倍机器，过完了就退掉。自建机房做不到这一点——你只能按最高峰去买机器，平时 90% 的时间它们在闲着吃灰。

注意，这第一班车表面上冲击的是戴尔、惠普这些卖服务器的，还有 EMC 这些卖存储的。甲骨文卖的是软件，好像隔了一层。**但真正致命的是它改掉了客户的采购习惯。**一旦一家公司习惯了「所有东西都按小时租、都能随时退」，它再回头看甲骨文那张几百万、签下去五年不能反悔的许可证，就会本能地皱眉。云不是抢走了甲骨文的某个客户，而是重新定义了「买软件应该是什么感觉」。

第二班车：SaaS —— 从上面把许可证整个换掉

如果说 IaaS 是从底下（机房这一层）动摇甲骨文，那 SaaS（Software as a Service，软件即服务）就是从上面，直接把「买软件许可证」这件事本身给废掉了。

SaaS 的意思是：**软件不再是你买回来装在自己机器上的东西，而是供应商在自己的服务器上跑好，你打开浏览器按月订阅就能用。**你不拥有任何软件，也不管它装在哪、怎么升级，你只是每月付费用它。

这条线上最有杀伤力的角色，是 Salesforce。它 1999 年开张，口号直白得像挑衅：**「软件的终结」**（The End of Software），发布会上还印着一个「No Software」的禁止符号。它卖的客户关系管理系统，恰恰是甲骨文和它刚收购的 Siebel 的主战场。

大白话

为什么 SaaS 对「收租之王」是釜底抽薪？回想前几章的逻辑：甲骨文的护城河，是客户把数据库和应用装进自己机房、深度绑定之后，迁移成本高到搬不动，于是甲骨文能年复一年地收租。可 SaaS 客户从第一天起就没有「自己的机房」这回事——软件在别人那儿跑，数据存在别人那儿。切换成本这块最大的石头，从一开始就没往河里砌。护城河还没来得及挖，河床先被填平了。

更微妙的是，SaaS 厂商自己在后台也要跑数据库。理论上它们可以用甲骨文——早年 Salesforce 确实大量用了甲骨文数据库。但随着它们长大，越来越多的 SaaS 公司选择开源数据库（PostgreSQL、MySQL 这些不要许可证费的）或者自研，就为了不给甲骨文交那笔「税」。所以 SaaS 这班车带来了双重打击：**它抢走了甲骨文的应用客户，同时长大后的它们还想方设法绕开甲骨文的数据库。**

关键问题：甲骨文明明看得见，为什么不上车？

一个爱因果的人到这里一定会问：这两班车又不是偷偷开走的，AWS 从 2006 就在跑，Salesforce 从 1999 就在喊。甲骨文有钱、有技术、有客户，为什么迟迟不动？

答案不是「没看见」，而是**看见了，但上车对当时的它是笔亏本买卖**。这里有三层机制，一层比一层深：

第一层：新生意的账，天生比旧生意难看

卖一张许可证，成本几乎是零——软件是复制品，多卖一份不多花一分钱，所以毛利率能到 90% 以上，钱当期一次性确认。而卖云，你得先自掏腰包建数据中心、买服务器、付电费，这些都是实打实往外流的现金；租金却要按月、按年慢慢收回来。

大白话

换句话说，从收租切到卖云，是主动把一门「近乎零成本、当场收全款」的生意，换成一门「先重金投入、再细水长流回本」的生意。对一个当期利润被华尔街盯着的上市公司，这在财报上就是自残：收入增长看着没变快，利润率和现金流却先塌一块。理性的 CFO 当然会犹豫。

第二层：经典的「创新者的窘境」

这是克里斯坦森那本书讲透的机制：**一家在旧市场里越成功的公司，越难跳上会侵蚀旧市场的新技术**——不是因为蠢，恰恰因为它在「对现有大客户负责」这件事上太称职了。

甲骨文最大的客户，是银行、电信、政府这些跑核心系统的大机构。这些客户当时并不急着上公有云——数据合规、安全、稳定性顾虑一大堆。销售铁军每天听到的反馈是「我们还要买许可证、还要自建机房」。于是公司内部所有的激励——销售提成、季度目标、产品路线——全都指向「把许可证卖得更多、更贵」，而不是「造一个会让许可证卖不动的云」。**越是维护好现有客户和现有利润，就越是在给自己造上车的阻力。**

第三层：老板亲手踩过刹车

还有一层很人的因素。甲骨文的掌门人拉里·埃里森，在相当长一段时间里公开唱衰云计算。2008 年他有一段著名的吐槽，大意是：**「云计算」这个词到底是什么意思？不就是我们本来一直在用的服务器和网络吗，改个名而已，我搞不懂大家在跟风什么。**

他没说错——云在技术底层确实还是服务器加网络。但他没抓住的是，云的革命不在技术，在**商业模式**：谁承担建机房的重注、谁背弹性和维护的包袱、客户按什么节奏付钱。当一号位公开表态「这只是炒作」，整个公司的战略惯性就更难扭过来了。

迟到的代价，和一个反转的伏笔

把这几层叠起来，「错过那班车」就不再是一句马后炮的嘲讽，而是一条清晰的因果链：**生意太赚钱 → 新生意的账天生难看 → 内部激励全指向守旧 → 老板亲自背书「云是炒作」→ 起步整整晚了近十年。**等甲骨文真正认真做云，AWS 已经是几百亿美元体量的巨兽，市场格局大局已定。

但这里要埋一个贯穿全书的伏笔，请记住它。甲骨文迟到，付出了在通用云市场几乎无缘第一梯队的代价；**可正因为它迟到，它没有像三大云那样早早锁死自己的架构。**当十几年后 AI 训练这班全新的车开来时——那是一种对机房带宽、对整机集群有着完全不同要求的负载——甲骨文反而能从一张相对干净的白纸上，专门为它盖一座数据中心。

这就引出了下一章的问题：既然租不上别人的车，那就自己造一辆。甲骨文的答案叫 OCI——它到底是什么，凭什么敢跟起步早它十年的三大云叫板？

第五章 · 自己盖，不租别人的

上一章的结尾我们停在一个尴尬的位置：甲骨文醒得晚，账面上还躺着许可证的租金，云这班车已经开走了。现在问题变得很具体——一个卖数据库的公司，想做公有云，第一步该干什么？

最省事的答案是：别自己盖。租亚马逊的机器，把 Oracle 数据库打个包扔上去，收个软件钱就好。很多软件公司就是这么干的。但甲骨文偏偏选了最重、最费钱、最容易被人笑话的那条路——**自己盖数据中心，一砖一瓦，从网络到机柜全都自己来。**这一章就讲清楚：OCI 到底是什么，它为什么跟三大云长得不一样，以及「自己盖」这个看似犯傻的决定，怎么在几年后变成了甲骨文手里最值钱的一张牌。

OCI 是什么：不是一个产品，是一整片地基

OCI（Oracle Cloud Infrastructure，甲骨文云基础设施）就是甲骨文自己的公有云。你可以把它理解成甲骨文版的 AWS：你不用买服务器，按用量租它的计算、存储、网络。听起来平平无奇，跟别人没区别。

但这里有个时间上的关键细节，也是第四章埋的伏笔。甲骨文云做过两代。第一代（大约 2016 年前后）基本是照着别人抄，做出来不温不火。真正重要的是**第二代 OCI**——甲骨文把第一代几乎推倒重来，2018 年前后重新设计了整套底层架构。迟到有个隐藏的好处：**它没有历史包袱**。亚马逊 2006 年就开始搭 AWS，那套架构是为「一堆小网站、小应用各跑各的」设计的，早期的很多技术选择固化在了地基里，改不动了。甲骨文晚了十年动手，反而可以看着别人踩过的坑，从一张白纸开始画。

> 大白话
>
> 类比一下：早期的云像老城区，路是几百年前按马车宽度修的，现在堵车也没法拓宽，因为两边全是楼。甲骨文相当于在旁边一块空地上重新规划一座新城，一开始就按高峰车流量把主干道修到八车道。平时看不出区别，等真来了大车队（后面会讲，就是 AI 的 GPU 集群），差距立刻拉开。

第一个不一样：把虚拟化搬出服务器

要讲清楚差别，得先讲一个云计算里最基础的东西：虚拟化。一台物理服务器很贵，云厂商不会把整台机器只租给你一个人，而是用软件把它切成好多份「虚拟机」，租给不同的客户，每个客户都感觉自己有一台独立的机器。干这个切分和隔离工作的软件层，叫 Hypervisor（虚拟机管理程序，可以理解成给虚拟机当房东兼保安的那层软件）。

传统做法是：这个 Hypervisor 跟你的程序住在同一台物理服务器上。问题来了——它自己也要吃 CPU、吃内存来干活。等于你租了一台机器，却有个「二房东」常驻在里面，占着你的资源，还能看到你的一举一动。更麻烦的是安全：房东和租客住一屋，一旦房东那层软件有漏洞，理论上就可能从一个客户串到另一个客户。

甲骨文第二代 OCI 的选择是 off-box 虚拟化（把虚拟化从服务器上挪走）。它把「当房东、管网络、管存储」这些活儿，从服务器主板上搬到了一块专门的网卡硬件里去做。

> 大白话
>
> 说人话：以前房东住在你家客厅，现在房东搬到了大门口的门卫室（那块专用网卡）。结果有两个：第一，你租的整台服务器的 CPU 和内存,现在真的全归你，没人分你的资源；第二，房东根本不进你的屋子，就算门卫室出事也波及不到你,客户之间的隔离更干净。这对普通小应用可能感觉不到,但对「我要一整台裸机满血跑」的高性能客户,是实打实的东西。

第二个不一样：扁平的、高带宽的网络

数据中心里成千上万台服务器要用网络连起来。传统大云的网络更像一棵树：机器先连到机架顶上的小交换机，小交换机再连到中层,中层再连到核心。这种分层网络的问题是——两台机器如果挂在不同的树枝上，数据要先爬到树顶再爬下来，中间每一层都可能堵车、都会加延迟。而且越往上，主干越可能成为瓶颈（这叫「网络收敛」，就是上层带宽故意做得比下层加起来小，赌你们不会同时满负荷跑）。

甲骨文走的是扁平网络（flat network）加高带宽，尽量做到**任意两台机器之间都有充足的、稳定的直连带宽**，不搞那种「赌你们不会一起用」的超额认购。为什么敢这么砸？因为它自己盖楼，网络拓扑是它自己定的，可以从第一天就按「会有客户需要几千台机器一起满速对话」来铺线。

第三个不一样：RDMA，让服务器之间近乎直连

这是最硬核、也最关键的一块。平时两台服务器通信，数据要经过一长串手续：你的程序把数据交给操作系统，操作系统打包成网络协议（就是 TCP/IP 那一套），经过 CPU 处理，再发出去；对面收到再反着拆一遍。每一步都要 CPU 插手、都要复制来复制去，延迟就是这么攒出来的。数据不大时无所谓，但如果几千台机器每秒要疯狂交换海量数据，这套手续就成了灾难。

RDMA（Remote Direct Memory Access，远程直接内存访问）是绕开这一切的办法：让一台服务器的网卡，直接去读写另一台服务器的内存，**几乎不惊动两边的 CPU，也不走那套繁琐的协议栈**。

> 大白话
>
> 打个比方：普通网络通信像两个部门传文件，得先交给收发室、登记、装信封、送到对方收发室、再登记、再拆封、才到人手里。RDMA 相当于在两个人桌子之间直接开了个传送管道，文件「唰」地就过去了，谁的秘书都不用惊动。延迟从毫秒级压到微秒级，差着一两个数量级。

甲骨文把 RDMA 做成了 OCI 的一等公民——不是某个高端选项，而是整个高性能网络的默认底子。而 off-box 虚拟化、扁平高带宽网络、RDMA 这三样，其实是**互相咬合的一套设计**：虚拟化搬走了，服务器才没有二房东拖后腿；网络扁平了，任意两台才谈得上直连；有了 RDMA，这种直连才快到近乎「一堆机器合体成一台」。缺一个都不成立。

为什么这套东西天生适合「大规模并行」

到这里可以把三件事串成一句话了。甲骨文这套架构，专门优化的是一种特定负载：**成千上万台机器，需要低延迟、高带宽地紧密协作、频繁交换数据，像一台超级计算机那样一起算一件大事。**

这类负载在计算机领域有个老名字，叫HPC（High Performance Computing，高性能计算）——传统上是气象模拟、流体力学、基因分析这些「一道超大题目拆给一万台机器一起解」的科学计算。它们的共同特征是：机器之间不是各干各的，而是每算一步就要互相对一次答案，通信一慢，全场都得等最慢的那台，整体效率就崩了。所以它们对网络延迟的敏感度，远高于普通网站。

普通公有云是为「一堆互不相干的小应用」优化的——你的购物车崩了不影响我的博客，机器之间聊不聊天都无所谓，所以早期那种分层、超额认购、二房东虚拟化的架构完全够用，还更省钱。但 HPC 这种负载放进去就水土不服：延迟高、带宽不稳、CPU 被虚拟化偷走一截。

甲骨文当年下这个重注，公开的理由是要抢企业里那些「跑得动 Oracle 数据库、也跑得动重型计算」的高价值客户——数据库本身就是个对延迟极度敏感的负载，它太懂这种活儿了。于是它反其道而行：别人为「多而杂」优化，它为「少而重」优化，盖了一座专门跑重型负载的新城。

把赌注摆上桌

现在回头看这个「自己盖」的决定，账其实很拧巴。租别人的机器，轻资产，来钱快，报表好看；自己盖数据中心，要买地、买楼、买机器、铺网络、付电费，全是沉甸甸的重资产，回本还慢。单从做云软件生意的角度,这几乎是笨办法。

但请记住这套架构的形状：**它天生就是为「一大堆机器用超高带宽紧密互联、合体成一台超级计算机」而生的。**在 2018 年,这个形状对应的是一个不大不小的 HPC 市场,看起来性价比存疑。

可就在没几年后,世界上冒出了一种胃口大到离谱的新负载,它的技术需求跟 HPC 几乎一模一样,只是规模要再放大十倍、百倍——**训练大模型**。成千上万块 GPU，要用 RDMA 高带宽织成一张网、当成一台超级计算机来喂,谁的机房网络扁平、带宽足、延迟低,谁就天生占便宜。

甲骨文当年为了「少而重」的企业客户,咬牙盖的那座新城,地基恰好就是为这种负载打的。这是运气,还是眼光,后面几章会慢慢算。但有一点这一章必须先钉死：**正因为它迟到、没被旧架构锁死,才有机会照着高性能负载重画一张图;也正因为它选了最重的「自己盖」,才真握着地皮和网络,能在 AI 来敲门时把赌注一口气加到千亿。**下一章,我们就看这个赌注是怎么被放大十倍的。

第六章 · AI 把赌注放大十倍

把上一章的画面接过来：甲骨文花了十年，赌了一件当时看起来有点固执的事——它不去租别人的机房，而是自己盖，还盖成一种别人不太理解的样子：机器与机器之间用超高带宽的网络扁平地连在一起，任意两台之间几乎是「直线距离」。在 2019 年那会儿，你要问这有什么用，答案有点尴尬：绝大多数生意其实用不上。跑个网站、开个数据库、算算报表，机器之间根本不需要那么快地互相说话。甲骨文像是修了一条八车道的高速公路，通往一个只有几辆车的小镇。

然后 AI 来了。突然之间，小镇变成了要疏散的百万人口城市，而全世界只有几条八车道公路。这一章讲的就是：为什么训练大模型这件事，恰好是那种「机器必须疯狂地互相说话」的活儿，为什么它把甲骨文原本冷门的长处变成了稀缺资源，以及为什么赌注会因此从「百亿」这个量级，一脚踩到「千亿」。

为什么训练大模型，是一件「机器必须互相说话」的事

先说清楚训练一个大模型到底在干嘛。你可以把它想成：有一个上万亿个数字组成的巨大参数表（就是模型的「大脑」），你不断喂给它文本，让它猜下一个词，猜错了就把这上万亿个数字往「猜得更对」的方向轻轻推一下。这个「推一下」要重复几百万、几千万次。

问题是，这个大脑太大了，装不进一块 GPU（图形处理器，本来是给游戏算画面的芯片，因为特别擅长同时做海量简单乘加，被拿来算神经网络）。一块顶级 GPU 的高速显存也就几十到一两百 GB，而一个前沿模型的参数、加上训练时要临时存的中间结果，动辄需要几百上千块 GPU 的显存加起来才装得下。

于是你被迫把一个大脑拆开，摊在成千上万块 GPU 上。麻烦就在这：这些 GPU 不是各干各的，它们算的是同一个大脑的不同部分。每走完一步，它们必须把各自算出来的梯度（那个「往哪个方向推」的信息）互相同步一遍，凑成一个完整的更新，然后才能走下一步。

大白话

打个比方。一万个人合抄一本上万亿字的账本，每人负责一小段。抄完一页，所有人必须停下来，把这一页每个数字的修改互相通报、对齐，确认大家改的是同一个版本，才能翻到下一页。如果通报环节慢，那一万个人抄字的手速再快也没用——大家都在干等着对账。训练大模型就是这样：算得快不算本事，**对账（GPU 之间的通信）不能慢，才是本事。**

这就是关键的因果：GPU 越多，算力越强，但「对账」的负担也越重。如果 GPU 之间的网络不够快，你多插的那些 GPU 大部分时间都在空等通信，白白烧电。业内管这个叫 GPU 被网络「饿死」。所以训练集群不是 GPU 越多越好，而是 **GPU 加上把它们连起来的网络，共同决定了你能不能真的组成一台「超级计算机」**。

把上万块 GPU 连成「一台机器」，难在哪

让 GPU 互相说话，有两个层次。

第一层是一台服务器机箱内部的几块（通常八块）GPU，靠一种叫 NVLink 的专用高速通道直连——这个 Nvidia 已经帮你解决了，箱子内部的八块 GPU 之间快得像一块。

第二层才是难点：机箱和机箱之间、机架和机架之间，成百上千个箱子怎么连。这里用的是 RDMA（远程直接内存访问）——它的意思是，一台机器可以直接读写另一台机器的内存，不用惊动对方的 CPU 和操作系统去中转。普通网络传数据像寄快递：打包、贴单、进分拣中心、CPU 一层层拆封；RDMA 像在两台机器的内存之间打通一根直达管道，数据自己流过去，延迟低、还不占用 CPU。承载它的物理网络，通常是 InfiniBand（一种专为高性能计算设计的超高速网络，带宽和延迟都远好于普通以太网），或者是特意调校过、能跑 RDMA 的高端以太网。

但光有好网卡不够。真正决定成败的是**整个机房的网络拓扑**——也就是这几万个端口是按什么结构连起来的。这里有个残酷的物理事实：距离和层级都要收费。两台 GPU 如果只隔一个交换机，通信快；如果信号要往上爬三四层交换机、横穿半个机房再下来，延迟和拥塞就上来了。当一万块 GPU 每一步都要全体对账时，那个最慢的、绕最远的路径，会拖累所有人——木桶效应，短板是那根最长的网线。

大白话

所以「把一万块 GPU 连成一台超级计算机」，真正的工程活儿不是买 GPU，而是：设计一种网络结构，让任意两块 GPU 之间都尽量「近」、都有足够宽的路、还不会在全体同时对账时堵成一锅粥。这活儿必须在盖机房、拉线、排机架的时候就想好——事后是改不动的。

现在回头看第五章讲的甲骨文那套「off-box 虚拟化、扁平高带宽网络、RDMA 集群」的第二代 OCI，你会发现一件事：**它当年为「高性能计算」这个冷门小众市场设计的东西，几乎就是照着 AI 训练的需求提前长出来的。**扁平、高带宽、任意两点都近、原生支持 RDMA——这不是甲骨文预见了 AI，而是它为了别的原因修的八车道公路，恰好通向了这座突然爆发的城市。这在商业史上有个朴素的名字：运气，但是修在了实力之上的运气。你得先真的把公路修好，运气来了才接得住。

为什么赌注从「百亿」跳到「千亿」

理解了上面这些，赌注的量级跳变就顺理成章了。

过去甲骨文（和别的云）扩容，是「需求涨一点，加一点机器」——线性的、可控的，一年资本开支在百亿美元量级。但 AI 训练客户的胃口不是这么涨的。一个前沿实验室来谈，开口就是「我要几万块、甚至十几万块最新 GPU 连成的集群，而且我要签好几年」。这不是加几台机器，这是要你专门为它盖一整片、甚至好几片数据中心园区。

这里要引入这几年最能代表这种量级的名字。Stargate（星际之门）是 OpenAI 牵头、联合软银、甲骨文等方推动的超大规模 AI 基础设施计划，公开口径是要在数年内投入数千亿美元、在美国多地建设专供 AI 的巨型数据中心园区。而甲骨文与 OpenAI（做出 ChatGPT 的那家 AI 公司）之间，据多方报道签下了量级达数千亿美元、跨越数年的算力合同——具体数字随报道口径浮动，但公认在「数千亿美元级」这个量级，是软件行业闻所未闻的单笔体量。

大白话

翻译成人话：以前甲骨文接的是「租几千台机器」的订单，现在接的是「请你从零盖一座发电厂规模的算力工厂，专门供我一家用几年」的订单。订单的单位从「机架」变成了「园区」，从「季度」变成了「数年」，钱的量级自然从百亿跳到千亿。**不是甲骨文突然变激进，是客户的需求本身跳了一个数量级，逼着供给方也跳一个数量级。**

为什么偏偏轮到甲骨文接这种单？把因果串起来就清楚了：训练前沿模型需要「几万块 GPU 用 RDMA 连成一台超级计算机」这种极端形态的算力；能把这种集群盖得又快又扁平又不「饿死」GPU 的供给方，本来就没几家；甲骨文恰好有一套为高带宽负载而生的自建机房设计，还有一支敢在资产负债表上压重注、执行力凶悍的班子。于是那些天量算力合同，有很大一部分落到了它头上。OCI 就这样从第四章里那个「起步太晚的追赶者」，被 AI 浪潮一把推到了「AI 算力主要供应方之一」的位置。

把这一章钉住：主角是怎么换的

我们从头捋一遍这条因果链，因为它是理解后面所有财务争议的地基：

- 训练大模型，本质是让成千上万块 GPU 每走一步都全体「对账」——所以**网络（对账速度）和 GPU（算力）同样是瓶颈**。
- 要让对账不慢，就得把 GPU 用 RDMA / InfiniBand 连成任意两点都「近」的扁平高带宽集群——这活儿必须在盖机房时就设计好，事后改不动。
- 甲骨文当年为高性能计算自建的第二代 OCI，恰好就是这种形态——冷门长处一夜变成稀缺资源。
- AI 客户的订单单位从「机架」跳到「园区」、从「季度」跳到「数年」，于是赌注从百亿量级跳到千亿量级，Stargate 和 OpenAI 的天量合同就是这个跳变的具体形状。
- 结果：OCI 从追赶者变成主角之一。

但请记住，这一章讲的全是「机会」这一面：需求为什么爆、长处为什么值钱、订单为什么这么大。它把甲骨文推上了牌桌，也把赌注推到了一个前所未有的高度。而牌桌的另一面——为了盖这些千亿级园区，钱要从哪里借、资产负债表要承受什么、这些天量订单里有多少是真金白银、又有多少只是「已签未交付」的一串数字——那才是这本书真正要拆的账。

下一章，我们就从那个突然冲上约 4550 亿美元、让埃里森短暂登顶世界首富的数字开始：**RPO 到底是什么，为什么一夜暴涨，它离你口袋里的现金又有多远。**

第七章 · RPO 是什么，为什么一夜暴涨

上一章我们看到甲骨文把赌注从百亿加到了千亿——它签下了 OpenAI 那份天量算力合同。签完之后，2025 年 9 月的那份财报里，冒出来一个平时没人关心、这次却让整个华尔街炸锅的数字：**RPO 约 4550 亿美元**。当天股价跳涨了大约三成，Ellison 的身家一度冲到全球第一。

但你如果是工程师，第一反应应该是：这个 4550 亿到底是什么？是钱吗？在谁的账上？为什么一个数字能让市值一天涨掉几千亿美元？这一章我们就把这个词拆开，拆到你能用它自己判断这场狂欢里有多少是真的。

RPO 到底是什么：它是一张欠条，不是一沓钞票

RPO（Remaining Performance Obligations，剩余履约义务）说人话就是：**客户已经签了合同、答应要买，但公司还没交货、所以还没能算成收入的那部分金额**。它是「已签约、未交付」的订单总额。

> 大白话
>
> 想象你开了家装修公司。老王跟你签了合同，五年内让你给他装修 100 套房子，总价 4550 万，白纸黑字。但你现在一套都还没开工。这 4550 万就是你的 RPO——它是一个承诺的总额，不是你银行卡里的余额。你卡里现在可能一分钱都没多。

为什么会计上不让你把这 4550 万直接算成收入？因为有一条很朴素的规则：收入确认（revenue recognition，什么时候能把一笔钱记成"营业收入"）要等你真的把东西交付了才行。你装完第一套房、老王验收了，你才能确认这一套的钱。剩下 99 套还躺在合同里，它们是义务，不是业绩。

对甲骨文来说，这份"义务"就是：它答应在未来若干年里，给 OpenAI 这样的客户提供成千上万块 GPU 组成的算力。客户按合同分期用、分期付。甲骨文得先把数据中心盖出来、GPU 插上、电通上、网络连好，客户真的跑起了训练任务、消耗了算力，甲骨文才能一段一段地把 RPO 里的钱"搬"到收入表上。

为什么这个数字会"一夜暴涨"

平时 RPO 是个慢变量。软件公司卖许可证、卖订阅，合同一份份签，积压订单像水位一样缓慢上涨，季度之间变化不大。可这一季，甲骨文的 RPO 从上一季的一千多亿，跳到了约 4550 亿——**量级上翻了三四倍，主要就是几笔超大合同砸进来的**，其中 OpenAI 那份是主力。

这里藏着理解一切的关键：**RPO 的暴涨，本质是"签约动作"的暴涨，不是"交付"或"收钱"的暴涨**。签一份 3000 亿的合同，和交付 3000 亿的算力、收到 3000 亿的现金，是三件完全不同的事，中间隔着好几年和好几道坎。那天市场狂欢的，是第一件事——签约。

> 大白话
>
> 回到装修公司：你今天签下老王那份 4550 万的合同，你的"未来订单总额"确实一夜之间暴涨了。你可以拿这个数字去朋友圈发喜报，银行看了也愿意多借你点钱。但你家的冰箱里今晚还是没多一块肉。暴涨的是承诺，不是到账。

市场为什么先狂欢：RPO 是"未来收入的可见度"

那问题来了——既然只是欠条，市场为什么这么激动，甚至激动到让 Ellison 短暂登顶首富？

因为对一家云公司来说，RPO 是难得的一样东西：**对未来收入的"可见度"**。一般公司下个季度能卖多少，靠猜、靠预测。而 RPO 是已经签字画押的订单，它相当于把未来好几年的收入提前"点亮"了一大块。华尔街给公司估值，本质是在给"未来能赚多少钱"定价——现在突然有 4550 亿的合同摆在明面上，市场的算盘立刻就是：

- 甲骨文的云业务不再是"追赶者的故事"，而是有实打实的天量订单托底；
- 这些订单会在未来几年逐季转成收入，收入增速的天花板一下被顶高了；
- 管理层还顺势喊出了未来几年云收入要涨到几千亿的指引。

于是市场先按"这些钱大概率会兑现"来给股价重新定价，一天涨三成。市值（一家公司所有股票的总价格，等于股价乘以总股数）跳一大截，持股最多的 Ellison 身家自然跟着蹦上世界第一。这不是有人真给了他 4550 亿现金，而是市场"预期"变了，纸面财富随之膨胀。

RPO 和真钱之间，隔着三道坎

狂欢可以理解。但这本书的态度是：数字要负责任，得看清 RPO 和"真正到手的现金和利润"之间到底隔着什么。答案是三道坎，一道比一道硬。

第一道坎：能不能交付

合同签了，甲骨文得先把货造出来。这里的"货"不是软件光盘,是**实打实的物理世界**：几百万平方英尺的数据中心、几十万上百万块 GPU、配套的供电（一个大型 AI 机房动辄要几百兆瓦到上吉瓦的电，相当于一座中型城市的用电）、冷却、还有把 GPU 连成超算的高带宽网络。这些要花几年、花上千亿资本开支去建。**建不出来、建慢了、GPU 供不上货、电网批不下来，RPO 就卡在那里变不成收入。**它是承诺，兑现要靠工程和供应链。

第二道坎：客户会不会付、付不付得起

就算甲骨文如期交付了算力，还得客户真的用、真的按合同付钱。这里有个绕不开的名字：客户集中度（一家公司的收入有多依赖少数几个大客户）。这 4550 亿里很大一块押在 OpenAI 身上。OpenAI 是一家还在烧钱、自身也在靠融资输血的公司——它未来几年的付款能力，取决于它自己能不能持续融到钱、能不能把 AI 变现。

> 大白话
>
> 你那 4550 万的装修合同全是老王一个人签的。老王要是生意垮了、付不出钱，你这份漂亮的订单簿一夜之间就成了废纸。合同金额再大，也大不过"对手方会不会赖账"这件事。RPO 不区分客户信用——4550 亿写在纸上一样粗，但背后一个财力雄厚的政府客户和一个烧钱的创业公司，含金量差着量级。这颗雷，本书第十二章会专门拆。

第三道坎：交付了、收了钱，未必等于赚钱

最后一道坎最反直觉。假设前两道都过了：交付了，钱也收了。RPO 变成了收入——但收入不等于利润。为了赚这笔钱，甲骨文得为每一块 GPU 付出折旧、电费、机房维护。**卖算力的成本结构，天然比当年躺着收许可证租金要重得多。**所以 RPO 里的 1 块钱，最后能落成多少利润、甚至能不能覆盖当初借钱建机房的成本，是另一笔完全不同的账。这道坎，是第八、第九章的主题。

把这一章钉死的一句话

> RPO 是一张"未来会付钱"的巨额欠条集合。它涨得越猛，说明公司押下的赌注越大——但赌注大不等于赢，它同时意味着要交付的越多、要靠少数客户兑现的越多、要垫进去的成本越多。

所以那天股价一天涨三成、Ellison 短暂登顶首富，市场买的其实是**"这 4550 亿大概率能兑现"这个假设**。假设成立，甲骨文就完成了从收租之王到 AI 云主力的惊险一跃；假设动摇，纸面财富会缩得和涨上去时一样快。

RPO 告诉我们赌注下了多大，却一个字都没说这个赌能不能赢。要判断能不能赢，我们得翻到账本的另一面：为了造出这 4550 亿背后的机房，钱从哪来？下一章，我们去看甲骨文的资产负债表，正在承受什么。

第八章 · 钱从哪来

上一章我们讲了 RPO——那张厚厚的、已经签了字的订单簿。签了单，客户答应未来几年会付你几千亿。听起来是天大的好事。但这一章我们要问一个更朴素、也更要命的问题：**为了兑现这些订单，甲骨文得先把机房建起来、把 GPU 买回来、把电通上——这些钱，现在就得花。它从哪来?**

订单是「未来别人欠你的」，机房是「现在你欠别人的」。这两笔账在时间轴上错开了好几年，而中间这段空档，就是这一章的全部张力。

先算一笔工程师能摸到的账

想象你接了一个大工程：客户承诺五年内分期付给你 3000 亿。但施工得先垫钱——买地、盖楼、买设备、雇人。这里最烧钱的一项叫 资本开支（capex，capital expenditure）。

> 大白话
>
> 资本开支就是「买那种能用好几年的大件东西」花的一次性重投入：地、楼、服务器、GPU、变电站。区别于每个月都要付的水电工资（那叫运营开支）。capex 的特点是——钱今天一次性砸出去，好处却摊在未来好几年里慢慢收回。

甲骨文过去是什么样的公司？卖数据库许可证的公司。软件是代码，复制一份的边际成本几乎是零，所以它常年 capex 很低——一年几十亿美元，主要是些办公楼和普通机房。这是「收租之王」的财务长相：进来的是现金,几乎不用往外砸大件。

可一旦决定自己盖 AI 数据中心，这张脸就变了。买地、建楼、拉光纤、买成千上万块 GPU（一块高端 AI 加速卡量级上要几万美元，一个集群就是几万块），capex 从过去一年一百来亿美元的量级，往每年**三四百亿、乃至市场预期的五六百亿甚至冲向千亿**的方向猛涨。翻了大约四五倍还不止。

自由现金流为什么会转负

公司真正「能自己支配的钱」，有个专门的指标。

自由现金流（free cash flow）= 经营活动赚回来的现金 − 资本开支。翻成人话——你做生意收进来的真金白银，减去买大件花掉的钱，剩下的才是你能拿去还债、分红、或者存起来的「净剩」。

> 大白话
>
> 打个比方：你开餐馆，一年经营下来手里多出 100 万现金。这一年你没大动作，那 100 万就是你的自由现金流，可以还房贷、可以分给合伙人。但如果你这一年决定砸 150 万开三家分店（这就是 capex），那你不但没剩钱，还倒欠 50 万——自由现金流就是负的。分店明年才开始赚钱，可 150 万今天就得掏。

甲骨文现在正是这个状态。它的经营现金流依然强劲（收租的老本行还在源源不断进钱，量级上一年好几百亿），但 capex 涨得比经营现金流更快、更猛，一减之下，**自由现金流从常年为正，翻到了负值**。这不是生意变差了——生意反而在扩张——而是它主动选择把「今天的钱 + 借来的钱」全押到「明天的机房」上。

缺的口子，靠发债补

经营赚的钱不够填 capex 这个大坑，差额从哪来？答案是借。具体方式叫 发债（发行债券）。

> 大白话
>
> 发债就是公司公开打一张巨型借条：「我借你 X 亿，约定利率，几年后连本带息还。」买这张借条的是养老基金、保险公司、各种大机构。它和找银行贷款类似，只是规模更大、更标准化、能拆成很多份卖给一大堆人。

2025 年，甲骨文做了近年来规模最大的几轮发债之一，量级上一次就是几百亿美元。加上原有的债务，它的总负债堆到了一个相当高的位置——净负债（借的钱减去手头现金）跟着往上顶。

这里要引入一个工程师最该盯住的概念：杠杆。

大白话

杠杆就是「你借的钱相对于你赚钱能力的倍数」。常用的一个量法是「净负债 ÷ 每年的经营性盈利」。这个数字是 2 倍，意思是把你两年的核心盈利全拿去还债才还得清；是 4 倍，就是四年。数字越大，说明你欠得越重、腾挪空间越小、越经不起一次意外。

过去收租时代，甲骨文虽然也借钱（比如用来回购股票），但杠杆是可控的，因为收租现金流又稳又肥。现在的问题是：**债在快速加，而这些债换来的资产（GPU 机房）还没开始大规模产出现金**——分店还在装修，房贷已经压上来了。分子（债）先涨，分母（这些新资产带来的盈利）要过几年才涨。中间这段错位期，账面上的杠杆看起来会格外扎眼。

评级机构和 CDS：市场在替它体检

当一家公司借这么多钱，外面有两双眼睛在盯着它「会不会还不上」。

第一双眼睛：信用评级

信用评级就是穆迪、标普这类专业机构给公司还债能力打的分，像 AAA、AA、A、BBB 这样一级级往下排。分越高，说明机构认为你越靠谱，你借钱的利息就越低；分往下掉，利息就变贵。

甲骨文目前还稳在「投资级」（也就是被认为大概率还得起的那一档）里，但评级机构已经明确表达了担忧的方向：**债务在快速上升、自由现金流转负、且大量投入押在少数几个客户身上**。有机构把它的评级展望调成了偏负面。评级往下动一格,看似只是一个字母,实际后果是——以后每一轮新借的钱,利息都更贵,而它偏偏还要一轮接一轮地借。这是个会自我加压的循环。

第二双眼睛：CDS

还有一个更实时、更诚实的市场信号，叫 CDS。

大白话

CDS（信用违约互换，credit default swap）本质就是「给一家公司会不会赖账买的保险」。你手里握着甲骨文的债，怕它违约，就付一笔保费买 CDS；万一它真违约，卖保险的赔你。关键在于——这个保费是公开交易的价格。市场越担心它违约,愿意为这份保险付的价就越高。所以 CDS 价格就是一个实时的「体温计」：数字往上蹿，说明市场的恐惧在升温。

2025 年，随着债务和 capex 的消息出来，甲骨文的 CDS 价格明显走高——不是到「要违约」的地步，而是到「值得警惕」的地步。翻译一下市场的潜台词：*「我信你现在还得起，但你正在用越来越大的赌注、越来越薄的现金缓冲，去赌一个几年后才见分晓的未来。我要为这份不确定多收点保费。」*

工程师视角：这张资产负债表在承受什么

把上面的机制拼起来，一张清晰的受力图就出来了。甲骨文这张表，正在同时承受三个方向的拉力：

- **时间错位**：capex 是今天一次性砸出去，RPO 那些钱要未来几年才慢慢回来。中间的缺口只能靠借，于是负债先鼓起来。
- **缓冲变薄**：自由现金流转负，意味着公司暂时没有「净剩的钱」当安全垫，一切腾挪都依赖持续借新债。融资的门只要一有风吹草动收紧，压力立刻传导。
- **信号变紧**：评级展望和 CDS 都在往「更贵、更谨慎」的方向走，而它恰恰是最需要便宜资金的时候。这两者互相拉扯。

说到底，这不是「甲骨文快不行了」的故事，而是一次**财务性质的彻底切换**：从一个「不用怎么花钱、稳稳收租、账上现金充裕」的低 capex 公司，变成一个「像重资产基建商一样、先借重金砸下去、赌未来收回来」的高 capex、高杠杆公司。它把资产负债表从「保险箱」改造成了「杠杆」。

杠杆能撬起更大的东西，这是它的全部意义；但杠杆也会把每一个意外放大——利率、评级、某个大客户的变卦，任何一点松动，都会被这根越绷越紧的杠杆传导、放大。钱从发债来，债从对未来的信心来。而信心值多少钱，市场已经在 CDS 的价格上，一分一厘地标出来了。

下一章我们接着追问：就算钱借到了、机房建起来了、GPU 转起来了——卖算力这门新生意，赚的钱真有收租那么厚吗？为什么它的毛利率，天生就矮了一截。

第九章 · 毛利率为什么被压

接着上一章。我们已经知道甲骨文要借几千亿去买 GPU、盖机房。这一章只回答一个问题：**同样是一块钱的收入，为什么以前能剩下九毛，现在可能只剩两三毛？**钱去哪了？我们把成本这条线,一根一根拆开看。

先回到第一章那个"太舒服"的基准

第一章讲过,收租之王最爽的地方是零边际成本——多卖一份东西,几乎不用多花一分钱去生产它。

> 大白话
>
> 软件不是钢铁。第一套 Oracle 数据库,公司砸了几千个工程师、好多年写出来,这叫固定成本,一次性的。可写完之后,再多卖给第 1001 个客户,只是多拷一份文件、多发一个授权码。多这一单的**额外成本几乎是零**——这就是"边际成本为零"。

所以软件公司的账才那么好看。业内常看一个数叫毛利率:一块钱收入,扣掉"为了交付这份东西直接花的成本"之后,还剩多少。剩得越多,毛利率越高。甲骨文卖许可证和后续的软件支持服务,毛利率能到**约 90%**——一块钱收进来,直接成本只有一毛,九毛是毛利。这不是本事大,是生意的物理属性:你卖的是可以无限复制、几乎不磨损的比特。

现在卖的东西,物理属性变了

租 GPU 算力不是卖比特了,是卖"一台机器在一段时间里替你算东西"的能力。这东西有实体、会磨损、要喂电。三笔以前几乎不存在的成本,现在结结实实压在每一块钱收入上。

第一笔:折旧——最狠的一刀

折旧说的是:你花大钱买的设备会随时间贬值、变旧,会计上要把这笔钱按使用年限一年一年摊进成本里。

大白话

你花 3 万块买张 GPU 卡。它不是买来就一次性记 3 万块支出,而是假设能用 5 年,那每年就"用掉"6000 块,这 6000 块记成当年的成本。这就是把一大笔钱摊成好几年的小刀,每年割一次。

问题出在两点。第一,量太大。数据中心里的加速卡不是一张两张,是**成千上万张**,一栋楼的卡就是几十上百亿的采购。这些钱一摊,每年的折旧成本就是个巨大的数字,直接从收入里扣掉。

第二,也是更要命的——**GPU 贬值得比普通机器快**。数据库软件写好了能卖十几年不过时;可 AI 加速卡两三年就出新一代,新卡更快更省电,旧卡的算力立刻不值钱。所以这资产的"保质期"很短,你得在它还值钱的两三年里把本赚回来。折旧的年限一压短,每年摊的成本就更重。一张卡若按 3 年摊,比按 6 年摊,每年的成本直接翻倍。

> 收租时代:设备(服务器)是小头,产品(软件)不磨损。AI 时代:设备就是产品本身,而且它在你眼皮底下一天天变旧。

第二笔:电费——一直在烧的钱

许可证卖出去,不用为它持续供电。GPU 不一样:它要算东西就得通电,而且是出了名的电老虎。上万张卡同时满载跑训练,一个大机房的耗电量可以顶得上一座小城。这笔电费是**持续的、随用量涨的经营成本**——客户用得越狠,你电费掏得越多。它不会像固定成本那样摊薄,而是牢牢贴着收入一起长。

第三笔:机房和运维——散热、场地、人

上万张卡挤在一起,发的热量惊人,冷却系统本身又是一大笔电和设备。再加上厂房、网络、维护它们的工程师。这些合在一起,是收租生意里几乎可以忽略、现在却省不掉的一大块**持续成本**。

大白话

一句话对比:以前的成本主要是"当年写软件的人工",写完就不再花了;现在的成本是折旧+电+散热+运维,机器开机的每一天都在花。收入是新赚的,成本却是天天在滴的。

把三笔加起来:毛利结构天然就薄

把折旧、电费、机房运维都算成"为了交付这份算力直接花的成本",再回头看毛利率——一块钱算力收入里,这些直接成本可能就吃掉一大半甚至更多。所以云基础设施(IaaS)这门生意——就是把机器算力当水电一样租给别人用——它的毛利率**天然远低于卖软件**,通常在几成的量级,而不是软件那种九成。这不是甲骨文经营不善,是所有卖算力的人都逃不掉的物理账:你卖的是会折旧、要喂电的铁疙瘩。

为什么租给 OpenAI 这种大客户,毛利还要更薄

更扎心的是:同样卖算力,卖给巨型客户往往还得**打折**,毛利比零售更薄。原因很工程师:

- **议价权在买方手里。**OpenAI 这种客户一签就是天量、长期的量。量越大,对方越有底气压价——就像批发永远比零售便宜。为了锁住这块巨单,你得让利。
- **为它专门建、专门包。**这么大的量,往往要为它规划整片机房、整批卡,很多产能是为这一个客户预留的。你把一大块资产押在一个合同上,却拿着更低的单价。
- **成本没因为量大而变便宜多少。**软件多卖一份边际成本是零,所以给大客户打折无所谓,反正没成本。可算力多卖一份,折旧、电、散热一分不少地照付。**打了折,底下的硬成本却纹丝不动**——这一压一不动,毛利就被夹薄了。

大白话

软件打折:反正没成本,打折等于白赚少一点,还是赚。算力打折:成本是硬的,打折是直接从本就不厚的那层毛利里割肉。同样是"给大客户优惠",两门生意的痛感完全不同。

比毛利率更值得盯的:利润质量

光看毛利率还不够。真正变差的是利润质量——一块钱收入,最后有多少变成公司能自由支配、能拿去还债或分红的真钱。

这里藏着一个会计上的错位,得说清楚。折旧在利润表上是"成本",可它不是当年真流出的现金——现金是**当年买卡时一次性砸出去的**。于是出现一个尴尬局面:

- 买卡那年,几百亿现金真金白银流出去了,但账面上只记了其中一年的折旧,利润看着还行;
- 可你为了跟上 AI,几乎**每年都在大规模买新卡**——旧卡还在摊折旧,新卡的现金又流出去了。折旧摊的是过去买的,现金花的是今年买的,两头叠加。

> 大白话
>
> 这就是为什么上一章会看到"利润表上还有利润,自由现金流却转负"。利润是按折旧慢慢摊出来的账面数,现金是实打实往外掏的。当你年年都在加倍买卡,现金流出的速度会跑在账面成本前头——赚的是"账上的利润",花的是"兜里的现金"。

所以利润质量下降,说的是同一块钱收入的**含金量**变了:收租时代,一块钱毛利几乎就等于一块钱能自由支配的现金,因为没什么要持续再投入的;算力时代,一块钱毛利背后,拴着必须年年续上的巨额资本开支——不接着买新卡,你的算力两三年就落伍,合同就交付不了。这块钱看着是利润,却大半得再投回到下一批很快又会贬值的铁疙瘩里。

把这一章收成一句话

收租之王最舒服的东西,是"零边际成本"——多卖一份不多花钱,九成是毛利,毛利几乎全是自由现金。而 AI 算力这门生意,正在把这条基准线一点点掰弯:

- 多卖一份算力,要实打实付折旧、电费、散热,**边际成本不再是零**;
- 卡两三年就贬值,逼你年年重投,毛利被折旧啃、被大客户折扣压,**结构上就薄**;
- 账面利润和口袋现金分了家,一块钱收入的**含金量下降**。

这不是说这生意不能做——云和 AI 的盘子够大,薄毛利乘上巨大的量,绝对利润未必小。但市场半信半疑的一个根子就在这:甲骨文是拿一门**九成毛利、现金极干净**的老生意做底,去换一门**毛利薄、要持续烧现金**的新生意的规模。护城河也许在变宽,可河水的成色,确实和收租时

代不一样了。下一章我们去看那个更刺耳的质疑——这些需求里,有多少是几家公司左手倒右手转出来的。

第十章 · 左手倒右手的疑云

市场对甲骨文最刺耳的质疑,不是"赚得少",而是四个字:**左手倒右手**。意思是——它签下的那些天量订单,可能不是真实需求,而是几家公司互相把钱和算力递来递去,转出来的一个热闹圈子。

先把这个圈子画出来

你只要盯住三家公司和它们之间的三笔钱,就能看懂整件事:

- **OpenAI** 跟甲骨文签了大合同,要租用甲骨文建的数据中心算力(据披露量级在数千亿美元,分很多年)。
- **甲骨文** 为了交付这份合同,得去 **Nvidia** 那里买几十万张 GPU,建成机房。
- **Nvidia** 又反过来宣布要向 **OpenAI** 投资(公开表态量级达千亿美元级)。

把这三笔连起来看:Nvidia 的钱流向 OpenAI → OpenAI 拿钱去租甲骨文的算力 → 甲骨文拿这笔收入去买 Nvidia 的 GPU → Nvidia 的芯片卖出去了,收入变好看,又更有底气投 OpenAI。钱绕了一圈,几乎回到起点。

> 大白话
>
> 想象三个人围一张桌子打牌。A 借给 B 一百块,B 用这一百块向 C 买东西,C 又拿这一百块投给 A。桌上看起来发生了三笔"生意",GDP 涨了、大家账面都热闹,可桌子外面并没有真的多进来一个客户、多卖出一件东西。钱只是在三个人手里转了一圈。

为什么这种结构有个专门的坏名字

金融圈管这叫 循环融资,也叫 供应商融资(vendor financing)——卖东西的人,自己出钱(或者变相出钱)让客户来买自己的东西。

为什么这值得警惕?因为它把两件本该分开的事搅在了一起:**"有人真的需要我的东西"**和**"我自己造了一个买家"**。健康的需求,是外面的世界拿它自己挣的钱来买你;可疑的需求,是你先把钱塞给买家,买家再拿这笔钱来买你——收入是真的记进了报表,但那笔钱的源头是你自己。

对工程师来说,这就像一个系统的自测:如果你写的压力测试,负载是测试进程自己发给自己的,QPS 曲线再漂亮也说明不了真实用户扛不扛得住。**需求的信噪比,取决于信号是不是从系统外部来的。**

历史上这么玩过,结局很难看

这不是新剧本。上一次大规模上演,是 2000 年前后的电信泡沫。

当时市场笃信"互联网流量要爆炸,光纤永远不够铺",于是一堆新兴电信公司疯狂下单买设备。但这些新公司很多没钱。卖设备的巨头——比如 朗讯(Lucent,当年做电信设备的顶级供应商)、北电(Nortel)——为了把货卖出去,干脆**借钱给这些客户,让它们拿这笔钱回来买自己的设备**。

账面上,朗讯的订单和收入一路飙升,股价冲天。可这些订单背后的客户,本身就是靠借来的钱在运转,真实的网络流量根本没跟上。等到资本市场一收紧,客户还不上钱、纷纷倒闭,朗讯手里的"应收账款"变成了收不回的坏账,之前记进去的"收入"又得反过来冲销。泡沫破的时候,不只是股价跌,是**之前那些漂亮数字被证明有一部分从来就不是真的**。

> 大白话
>
> 供应商融资最阴险的地方在于:它在上行期让你的数字比真实需求更好看,又在下行期让你的窟窿比真实亏损更大。它是个放大器,顺风时放大繁荣,逆风时放大崩塌。审慎的投资人怕的不是这个圈子今天转不转得动,而是它一旦卡住,反噬会比普通生意猛得多。

所以当投资人看到"OpenAI 租甲骨文、甲骨文买 Nvidia、Nvidia 投 OpenAI"这张图时,脑子里立刻蹦出来的就是朗讯。结构太像了。

但是——别把洗澡水和孩子一起倒掉

说到这儿必须踩一脚刹车,否则就成了另一种偷懒。**"结构像泡沫"不等于"就是泡沫"。**2000 年电信最致命的问题,是那些光纤铺下去**根本没人用**——需求是纯虚构的。今天的 AI 算力,情况不一样:GPU 建好之后,是真的被烧得发烫、排队都排不上的。这是本质区别。

所以真正该做的,不是喊"这是泡沫"或"这是革命",而是拿把刀,把这堆收入切成两半:**哪一部分是外部真实需求,哪一部分是被合同和叙事放大出来的。**

怎么分辨真的那部分

给几个工程师能自己动手验的判据:

1. **钱的源头是不是外部的。** OpenAI 的收入里,有一大块是几亿真实用户和企业付的订阅费、API 调用费——这是从圈子外面进来的现金,是真信号。这部分越大,循环的成分越小。反过来,如果 OpenAI 花在算力上的钱,主要来自 Nvidia/微软这些同时又是它上下游的股东,那循环的味道就重。
2. **GPU 是不是真的在满负荷跑。** 循环交易造出来的是"签约"和"付款",但造不出真实的**利用率**。如果机房建好后 GPU 的算力被真实推理和训练任务占满、还得扩容,那需求就是硬的;如果建好后闲置、靠合同撑着账面,那才是虚的。利用率是骗不了人的物理量。
3. **合同能不能扛住降速。** 前面第七章讲过 RPO(已经签了、但还没交付确认的收入)。RPO 不是现金,是承诺。一个健康的承诺,是客户就算增速放缓也照样要用、照样付得起;一个脆弱的承诺,是它成立的前提是"客户必须一直高速增长、一直能融到下一轮钱"。后者一旦融资断档,承诺就会像多米诺骨牌一样往回倒。

那么甲骨文这笔,到底算哪种

诚实的答案是:两种成分都有,而且现在还没到能盖章的时候。

真的部分:AI 训练和推理的算力需求是实打实存在的,OpenAI 的产品有真实的、付费的外部用户,甲骨文的机房不是建了没人用。这跟朗讯当年"铺了没人点亮的暗光纤"有质的不同。

被放大的部分:这份需求里,有相当一块是**用一份长达数年、绑定单一大客户的巨额合同,一次性锁进了报表和 RPO**——它把"未来很多年可能发生的需求"提前变成了"今天的天量订单数字",而这个数字能不能兑现,又高度依赖 OpenAI 自己能不能持续烧下去、融下去。链条上任何一环(Nvidia 的投资、OpenAI 的融资、真实用户的付费)慢下来,这个看起来自我强化的循环,就会反过来自我削弱。

这也是为什么市场对甲骨文的态度是"半信半疑"而不是"看空"或"看多"——它不是在质疑 AI 算力有没有需求,而是在质疑:**甲骨文用这么重的债务、押在这么一个循环结构里的订单上,万一那个大客户和它背后的资金圈转慢了,谁来接盘。**

这个"大客户会不会成为单点故障"的问题,本身分量太重,我们留到后面两章专门算——先是下一章,为什么同样是循环、同样是重债,市场却愿意给纯 AI 云更高的估值,偏偏怀疑甲骨

文。

第十一章 · 同样建云，凭什么它们估值更高

把前面几章的账摆到一起，会冒出一个特别扎眼的问题。同样是借钱建 GPU 数据中心、同样是押注 AI 算力，市场却给出了两套完全不同的脸色：对 CoreWeave、Nebius 这类"纯 AI 云"，投资人愿意用很高的估值倍数买单；对甲骨文做同一件事，却半信半疑。这一章我们就把这件不公平的事拆开,看看市场那把尺子到底是怎么量的,以及它量得对不对。

先说清楚，"估值倍数"是把什么尺子

要谈"谁更值钱",得先统一单位。这里最常用的是市销率(P/S,Price-to-Sales),意思是"市场愿意为这家公司每一块钱的年营收,付多少块钱的股价总和"。

大白话

打个比方:两家煎饼摊,A 一年卖 100 万,B 一年也卖 100 万。你出价买下整摊,给 A 出了 500 万,给 B 只出了 200 万。那 A 的"市销率"是 5 倍,B 是 2 倍。你多付的那 3 倍,买的不是今天的煎饼,是你相信 A 明年、后年能卖得多得多、而且卖得干净利落的那个**未来故事**。

所以市销率高低,基本不是在给"现在"定价,是在给"未来的增长故事"定价。故事越纯、越陡、越可信,市场愿意付的倍数就越高。记住这一点,后面全通了。

先认识这两个"轻装新兵"

CoreWeave原本是挖加密货币的,手里囤了一堆 GPU,AI 浪潮一来,直接转型成"专门出租 GPU 算力的云"。Nebius脱胎于俄罗斯搜索巨头 Yandex 的海外资产重组,如今也是一家专做 AI 算力出租的欧洲云。两家的共同点是:年轻、体量小、几乎**只做 AI 算力这一件事**。

它们和甲骨文放在一起,像一头大象旁边站着两只猎豹。猎豹给人的第一印象就是快;而大象哪怕跑起来,你也很难一眼看出它在加速。这个体感差异,恰恰是估值差异的第一个来源。

第一笔账:叙事纯粹度——市场讨厌"混合体"

市场给增长故事定价时,最喜欢的是"一句话能说清"的公司。CoreWeave 是什么?"一家出租 AI 算力的云。"完。Nebius 是什么?同一句话。它们是纯 AI 云,收入几乎全来自 AI 算力出租,增长曲线陡得吓人——一个干净、单一、方向明确的故事。

甲骨文是什么?这就麻烦了。它是"一个躺着收了几十年数据库许可证租金、毛利极高的老巨头,同时背上上千亿债务、去干一件毛利薄得多、还没赚钱的新买卖"。这是一个**混合体**:老业务稳、慢、赚钱;新赌注快、烧钱、不确定。

> 大白话
>
> 为什么混合体不讨喜?因为估值是"分类定价"的。分析师心里其实有两把尺子:老收租业务用"稳定现金流"那把尺子量,值不了几倍;新 AI 赌注用"陡峭增长"那把尺子量,能量出高倍数。可这两块焊死在同一家公司、同一张财报里,谁也没法把它们干净地拆开单独估价。结果就是:高增长那部分的性感,被老业务的沉重给稀释、拖累了。市场对没法干净归类的东西,本能地打折。

纯 AI 云省掉了这道"分类"的麻烦。你买它,买的就是那一个纯故事,不掺水。市场愿意为这份"纯粹"多付钱——哪怕这份纯粹背后的风险,并不比甲骨文小。

第二笔账:增长质量与增速——基数大,增速天生显小

再看增速这个数字本身,这里藏着一个几乎所有人都会被骗到的陷阱。

假设 CoreWeave 今年的 AI 算力营收是"几十亿美元"量级,明年翻倍到"上百亿"量级,增速就是 100%、甚至更高。这数字一亮出来,谁看了都心跳加速。

甲骨文呢?它的 OCI 云增速同样很猛,但它总营收有"五百多亿美元"的**大基数**压着。就算它今年新增的 AI 云绝对金额比 CoreWeave 全年营收还大,摊到几百亿的总盘子里,整个公司的营收增速看起来也就"十几个百分点"——一个体面、但绝不性感的数字。

大白话

这是分母在骗你。同样往水里加一桶水,加进脸盆里水位猛涨,加进游泳池里几乎看不出变化。不是甲骨文那桶水小,是它的池子太大了。市场按"整体营收增速"这个百分比给估值倍数时,大象天然吃亏——它的绝对增量可能是猎豹的好几倍,可除以巨大的基数,百分比就被稀释得平平无奇。

更麻烦的是"增长质量"。纯 AI 云的营收增长,100% 都指向那个热门故事;甲骨文增长的营收里,一部分是老数据库业务的自然滚动,市场不会给这部分高倍数。于是同样一块钱新增营收,长在猎豹身上被当"纯 AI 增长"高价定价,长在大象身上却被摊薄进一锅"混合增长"里贱卖。

第三笔账:负债与体量——大象转身 vs 轻装新兵

负债这件事,表面看是甲骨文最吃亏、最该被扣分的地方:上千亿债务、capex 冲天、自由现金流转负(前面第八章讲过,就是"公司实际烧掉的现金多过赚回的现金")。听上去像个包袱。

但换个角度,大象的负债有大象的底气:它有几十年攒下的、极其稳定的数据库收租现金流在背后垫着——哪怕新赌注全打水漂,老业务的租金还在源源不断进账,债主至少睡得着。

轻装新兵就不一样了。CoreWeave 这类公司同样背着**惊人的高负债**——买 GPU、建机房要的钱一分不少,它们大量靠借债和把 GPU 抵押出去来融资。但它们身后**没有一条老收租现金流兜底**。一旦 AI 需求放缓、或者贷款到期续不上,它们几乎没有第二条腿站立。

大白话

所以真论"负债的安全性",大象其实更抗摔:它摔一跤有老本垫着;猎豹跑得快,可摔一跤就是硬着地。可市场在狂热期偏偏更爱猎豹的速度,对它同样致命的债务反而睁一只眼闭一只眼。这不是理性算出来的结论,是情绪挑出来的偏好。

把三笔账合起来:市场用的是两把不同的尺子

现在能收网了。市场对甲骨文和纯 AI 云,压根不是用同一把尺子在量:

- **纯 AI 云**,用的是"故事尺":故事越纯、增速百分比越高、方向越单一,倍数越高。它量的是想象力。

- **甲骨文**,用的是"混合体尺":老业务按稳定现金流估、给不了高倍数,新赌注的性感又被老业务和大基数稀释,还得为上千亿债务扣一道风险分。它量的是"这笔混合账我算不清,那就先打折"。

说穿了,市场愿意为甲骨文的 AI 赌注多付的那部分,被它自己"老巨头"的身份给对冲掉了一大半。它不是不被相信,是它那个"又老又新"的身份太难定价,市场干脆用打折来对付看不清。

但别急着替甲骨文喊冤——这偏爱本身就未必理性

到这儿你可能已经在替甲骨文抱不平了:同样的活,凭什么它被打折?这一章要诚实到底,所以反面也得说透——**市场更爱纯 AI 云,不代表纯 AI 云真的更安全,很可能只是它更爱一个干净的故事。**

把那些高倍数背后的雷点一个个亮出来,你会发现它们和甲骨文被质疑的东西,几乎一模一样:

- **同样的高负债。**纯 AI 云的资产负债表同样绷得很紧,大量靠债务和 GPU 抵押融资撑规模,而且没有老收租业务兜底,抗风险能力其实更弱。
- **同样、甚至更极端的客户集中。**甲骨文被骂"太依赖 OpenAI 一个大客户"(这是下一章的主角),可 CoreWeave 对微软这样的单一大客户的依赖度只高不低。把大半个身家押在一两个客户身上,这颗雷纯 AI 云踩得更深。
- **同样的 GPU 折旧问题。**第九章讲过,GPU 是会快速贬值的资产,几年就被新一代淘汰。这道折旧的坎,甲骨文要过,纯 AI 云一样要过,而且它们全部身家都压在这类会贬值的铁疙瘩上,更没退路。
- **同样的循环融资疑云。**第十章那个"钱在几家公司之间转圈"的结构里,纯 AI 云往往站在更靠中间、更依赖上游供应商和大客户输血的位置。

> 大白话
>
> 把结论说白了:市场给纯 AI 云的高倍数,买的不是"更低的风险",而是"更纯的故事"。这两只猎豹背着和大象一样的、甚至更重的雷——只是它们的故事一句话能讲完,听起来更热血,于是被定了高价。这更像一场对叙事清晰度的溢价,而不是对基本面安全度的奖赏。

所以回到本章标题的那个问题——"同样建云,凭什么它们估值更高"——答案有两层。**表层**是:纯 AI 云故事更纯、增速百分比更亮眼、体量小转身快,套进"故事尺"里天然吃香;甲骨文

是难以定价的混合体、大基数摊薄增速、还背着显眼的债,套进哪把尺子都被打折。**深一层**是:这套定价里有相当一部分是情绪,不是精算。市场此刻更愿意为一个听得懂、说得清、够刺激的故事付钱,而把那些藏在纯粹叙事背后、同样要命的负债、客户集中和折旧风险,暂时调成了静音。

而甲骨文最大的那颗雷——把三分之一以上的未来订单押在 OpenAI 一个还在烧钱的客户身上——到底是护城河,还是一个能让整座机房瞬间没人买单的单点故障?这就是下一章要单独算的账。

第十二章 · 客户集中度这颗雷

上一章我们把纯 AI 云和甲骨文放在一起比，末尾埋了一根刺：这些公司——包括甲骨文自己——都有一个共同的软肋，叫**客户集中**。这一章我们把这根刺单独拔出来，看清楚它到底有多扎人。

先说结论会吓你一跳的那部分：甲骨文暴涨到约 4550 亿美元的 RPO（Remaining Performance Obligations，剩余履约义务，也就是「已经签了合同、还没交付、未来要陆续确认的那笔收入」）里面，很大一块——市场普遍估计可能占三分之一甚至更多——押在**同一个客户**身上：OpenAI。一个到今天还在大幅烧钱、尚未稳定盈利的客户。

先把 SPOF 这个词讲清楚

工程师对这个概念不会陌生。SPOF（Single Point of Failure，单点故障）说的是：整个系统的命脉吊在某一个部件上，这个部件一倒，全盘皆输。

> 大白话
>
> 你搭了一套很气派的服务：三台 Web 服务器、两个负载均衡、多副本的应用层，看起来冗余做得滴水不漏。结果所有请求最后都打到同一个数据库，而这个数据库只有一台机器。那这套系统的可用性，本质上就等于那一台数据库的可用性。前面堆再多机器都是装饰——数据库一挂，服务就死。这台孤零零的数据库，就是 SPOF。

工程师的直觉训练是什么？看一个系统稳不稳，不看它顺风时跑得多快，而是问一句：**哪个部件倒了会让整个系统崩？那个部件有没有备份？**把这套直觉搬到甲骨文的商业模式上，OpenAI 这个客户，正在变成甲骨文收入结构里的那台单机数据库。

旧甲骨文：几万个客户,谁走都无所谓

要看清今天有多脆,得先回忆前几章讲的旧甲骨文有多稳。

收租之王的生意,是把数据库和应用卖给**成千上万**家企业——银行、电信、政府、制造业、零售,什么行业都有。每一家交的许可证和维护费,在总盘子里可能只占千分之几、万分之几。

这种结构在工程上叫什么?**高度冗余的分布式系统**。

> 大白话
>
> 任何一个客户明天倒闭、跑路、改用别家,对甲骨文的营收几乎无感——就像一个上万节点的集群挂掉一个节点,监控面板抖一下,系统照跑。而且前几章讲过,这些客户被切换成本牢牢锁住,想走也走不动。收入既分散又粘,这是收租生意最漂亮的地方:它的抗打击能力,是被几万个互不相关的客户摊薄出来的。

一个系统的健壮,从来不是靠单个部件多强,而是靠**没有哪个部件的死活能决定全局**。旧甲骨文就是这种健壮:去掉任何一个客户,曲线纹丝不动。

新赌注:把冗余换成了单点

现在看新故事。甲骨文为了接住 OpenAI 这张天量订单,做了三件在财务上不可逆的事:

- **举债**——发了大规模的债去筹钱(第八章);
- **建专用机房**——不是通用云机房,是为超大规模 GPU 训练量身定制的数据中心,扁平高带宽、RDMA 互联那一套(第五、六章);
- **囤成堆的 GPU**——上万张卡,买来就开始按几年报废的节奏折旧(第九章)。

这三件事的钱,已经花出去或者已经承诺要花。而支撑这笔巨额投入的需求,压在一个客户的合同上。用系统的话说:甲骨文把一套本来靠几万客户摊薄风险的**分布式冗余架构**,在 AI 这条新业务线上,亲手改成了**吊在单点上的架构**。前面借的债、建的机房、囤的卡是「前端」,OpenAI 是那台唯一的「后端数据库」。后端一旦不给力,前端堆再多资产都是装饰。

算一算这个单点倒了会怎样

SPOF 的风险分析,不是问「它会不会倒」,而是问「**如果它倒了,损失沿着哪条路径蔓延**」。我们把 OpenAI 这个单点可能出问题的几种方式,和随之而来的连锁反应,一条条摆出来。

情况一:增长不及预期

OpenAI 签的是一份未来若干年、逐年放量的巨额算力合同。这份合同能兑现,前提是它的收入和用量真能按剧本涨上去。可它今天还在烧钱——训练和推理的成本高得吓人,靠一轮又一

轮融资续命。如果增长慢下来,它未必付得起当初签下的那么多算力。

大白话

关键在于:甲骨文的机房和 GPU 是**先建好、先付钱**的,合同里的收入却是**往后逐年才确认**的。这中间有个时间差。资产的折旧不等人——GPU 一上电就在贬值,机房的电费和运维每天都在烧。如果那份未来收入没足额到账,已经砸下去的成本可不会跟着缩水。

情况二:融资断档

OpenAI 履约的能力,高度依赖它能不能持续从外部融到巨额资金。这就引出上一章、下面还要细讲的循环融资隐忧——它的钱有一部分和整个 AI 资本循环绑在一起。**一个还没自我造血、靠外部输血活着的客户,它的付款能力,本质上是别人对 AI 前景的信心。**信心在,水管就通;信心一断,水管就干。甲骨文这份合同的现金流,间接吊在市场情绪这根更细的线上。

情况三:转向自建或别家

这是最微妙的一条。OpenAI 并没有义务永远只用甲骨文的云。它完全可能自己建数据中心(它确实在推进自建),也可能把算力分散到别的供应商去,避免自己被单一云商锁住——毕竟本书前几章反复讲,谁都怕被锁定,OpenAI 比谁都懂这个道理。

大白话

这里有个残酷的不对称:旧甲骨文靠切换成本把客户锁死,是甲骨文握着锁、客户被锁。可在这份 AI 大单里,反过来了——甲骨文为一个客户举债建了专用产能,议价的筹码更多在客户手里。客户要是分散下单或转向自建,甲骨文那些为它定制的机房和成堆 GPU,一时半会儿找谁来接盘?这些资产越专用,越难转手。

护城河,还是单点故障?

那么问题来了:这么大一笔押注在一个客户身上,到底算是护城河,还是 SPOF?

乐观的说法是护城河:能拿下 OpenAI 这种量级的客户,恰恰证明 OCI 的技术(RDMA 超算网络那一套)是真能打的,别人建不出、接不住。这话有道理——第五、六章讲过,甲骨文的架构

确实为这种负载量身而生。

但护城河和单点故障,其实是同一件事的两张脸,区别只在于:**你手里护的那样东西,靠不靠一个部件撑着。**

> 旧甲骨文的护城河,是几万个客户、几万条切换成本共同垒起来的城墙——拆掉任何一块,墙还在。
>
> 新甲骨文这条 AI 护城河,更像一座吊桥,吊在 OpenAI 这一根缆绳上。缆绳绷得住,桥就气派;缆绳一断,桥连同为它借的债、建的机房、囤的卡,一起掉进河里。

工程师最忌讳的,就是把「现在跑得很好」当成「很稳」。一台没有冗余的单机数据库,在它挂掉之前,性能可能是全公司最亮眼的。稳不稳,不看它顺风时多快,只看它背后那个单点倒下时,损失会不会无处可去。

大白话

所以把两代甲骨文摆在一起,风险的形状变了。旧的是分布式集群:收入摊在几万个互不相关的客户上,单点失效被冗余吸收,曲线不抖。新的是把一大块未来收入、外加已经砸下去的债务和专用资产,全接到一个还在烧钱、尚未盈利、而且不一定非用它不可的客户上。前者的稳,是结构给的;后者的险,也是结构给的。

这不是说 OpenAI 一定会出事——它很可能一路狂奔,把合同履约得漂漂亮亮,那样甲骨文这一注就赢麻了。这一章要说的只是:**甲骨文亲手把自己收入结构里的冗余,换成了一个单点。**而一个审慎的观察者评估风险,从来不是赌那个单点不会倒,而是先算清楚——万一它倒了,谁来为那些已经建好的机房和成堆的 GPU 买单。这个问题的答案,直接决定了下一章要摊开讲的那团循环融资疑云,到底有多值得警惕。

第十三章 · 数据库锁定还硬不硬

前面十二章，我们一直在给甲骨文挑刺：借了上千亿的债、押注押在一个客户身上、循环融资像 2000 年电信泡沫。现在把镜头掉个头，做一次正面体检——甲骨文最老、最赚钱的那块地基：**数据库锁定**，今天到底还硬不硬？

先说清楚"锁定"是什么。数据库锁定不是法律条款，是一种物理事实：你的公司把核心数据放进了 Oracle 数据库，用了它的 SQL 方言、它的存储过程、它的高可用机制，写了几十万行紧贴着它的代码。想换成别家，等于给一栋住了人的大楼换地基——技术上能做，但要停业、要重写、要冒数据丢失的风险。于是绝大多数人选择每年乖乖续费。租金就是这么收上来的。

这块地基这几年被甲骨文加了三样新东西：自治数据库、Exadata、多云互联。前两样是把锁往硬里焊，第三样最微妙——它到底是加固还是开逃生门，是本章要吵清楚的那个点。

第一样：自治数据库，把 DBA 焊进服务里

自治数据库（Autonomous Database）说白了是一个"自己照顾自己"的数据库。传统上，一个数据库背后得养一群 DBA（数据库管理员，专门给数据库打补丁、调性能、做备份、半夜出事爬起来救火的人）。自治数据库把这些活儿大部分自动化了：补丁自己打、索引和内存自己调、备份自己做、出故障自己切换。卖点很实在——省人、少出错。

> 大白话
>
> 就像从"雇一个专职司机天天伺候的老爷车"换成"一辆自动驾驶车"。车还是那辆车，但你不用再养司机了。听起来是帮你省钱，可省下来的那份工资，其实一部分变成了甲骨文的订阅费——而且这辆车只有甲骨文能修。

从锁定的角度看，这一步很聪明。它把锁定从"数据和代码"这一层，往下再钉了一层："运维"。原来你还养着一队 DBA，这队人理论上哪天也能帮你把数据库迁走；现在这队人裁了，运维知识连同人一起交给了甲骨文的自动化。真要搬家，你连一个懂怎么搬的人都不剩了。**它卖的是省心，买回来的是更深的依赖。**

第二样：Exadata，把锁定焊进硬件

Exadata是甲骨文的数据库一体机：把数据库软件，和一整套专门为这个软件调过的硬件（服务器、闪存、内部高速网络、智能存储），打包成一个箱子卖给你。核心卖点是快——Oracle 数据库跑在 Exadata 上，比跑在普通服务器上快很多。

它凭什么快？关键在一个叫智能扫描（Smart Scan）的招。普通数据库查数据，是存储把一大块原始数据整个搬到 CPU，CPU 再从里面挑出你要的那几行——搬得多，算得少，网络和内存全堵在路上。Exadata 反过来：让存储层自己先过滤，只把命中的那几行送上去。相当于你去仓库取货，普通做法是把整个货架推到你面前你自己翻，Exadata 是仓库管理员先按清单挑好、只把你要的那箱递出来。

> 大白话
>
> 这个"存储自己会挑货"的本事，是甲骨文把软件和硬件揉在一起才做出来的——普通硬件上跑普通 Oracle，享受不到。于是性能成了锁定的一部分：你一旦习惯了 Exadata 的快，换到别处不光要重写代码，还要接受"变慢一大截"。慢，也是一种迁移成本。

更狠的是，Exadata 现在不只是摆在你自己机房里的箱子，它进了云——甲骨文云里有 `Exadata Cloud`，你按用量租，不用买整台。这让锁定从"资本开支"（买一台几百万的机器）变成了"细水长流的订阅"，门槛更低，反而更多人踩进来。**把锁定焊进硬件的好处是：软件可以被模仿，一整套软硬协同的性能优势，别人抄起来慢得多。**

第三样：多云互联——这一章真正的问号

多云互联指的是 `Oracle Database@Azure` / `@AWS` / `@Google` 这一系列产品：甲骨文把自己的数据库硬件（就是上面说的 Exadata 那套），直接搬进微软、亚马逊、谷歌三大云的机房里，摆在人家的数据中心内部。于是一个客户主力用 AWS，也能在 AWS 里贴着用原生的 Oracle 数据库，数据库和它的应用之间是机房内部的低延迟连接，不用跨着公网来回跑。

为什么要这么做？因为过去十年发生了一件对甲骨文不利的事——客户上云，上的多半是 AWS、Azure、Google 这三家，不是甲骨文自己的 OCI。客户跟应用一起跑到了别人家。

甲骨文面前只有两条路：要么眼睁睁看着客户在别人的云里慢慢把 Oracle 换成那家云自带的数据库；要么，跟着客户过去。

它选了跟过去。这就是本章的核心张力。同一件事，有两种完全相反的读法：

读法 A：这是加固——"你跑到 AWS 上也甩不掉我"

过去客户搬去 AWS，是甩掉 Oracle 的天赐良机：既然整个技术栈都在重搭，顺手把数据库也换成 AWS 自家的（比如 Aurora），一劳永逸。多云互联堵死了这条路——现在你搬去 AWS，Oracle 数据库原封不动跟着进来，性能一样、体验一样，你连"重搭时顺手换掉"的借口都没了。锁定不但没松，还跨越了云的边界，变成"你去哪家云我跟到哪家"。从这个角度，甲骨文赢了：它把一个流失场景，改造成了留存场景。

读法 B：这是逃生门——"我留不住你，只好跟着你去别人家"

但换个角度，这动作本身就是一种认输。健康的收租之王，是让客户离不开自己的云；一旦你得把数据库拆下来，装进竞争对手的机房、按人家的规矩摆放、还要给人家分一杯羹，你其实已经承认：**留住客户的是那三家云，不是你。**你从"地主"退成了"租户的租户"。

更要命的是它给客户开的那扇门。数据库进了 AWS 机房、和 AWS 的服务贴在一起用之后，客户的技术重心是在 Oracle 这边，还是在 AWS 那边？长期看，客户越来越熟悉 AWS 的工具、AWS 的生态，Oracle 数据库慢慢变成 AWS 大盘子里的一道菜。哪天 AWS 把自家数据库的迁移工具做得足够顺滑，客户已经站在 AWS 家门口了，跨出最后一步只会更容易。多云互联可能不是关门，而是把客户送到了别人家门口。

那到底哪种读法对？一个诚实的判断

我的判断是：**短期是加固，长期是风险，具体偏哪边，取决于一个可观测的信号——谁在控制那段关系。**

看两件事就够了。第一，甲骨文在这套里是**被集成方还是集成方**。如果客户是"以 Oracle 为核心，顺便用点云资源"，那是加固；如果是"以某家云为核心，顺便挂个 Oracle 数据库"，那 Oracle 就是那道随时可能被换掉的配菜。第二，看**定价权在谁手里**。进了别人机房，甲骨文的续费涨价还硬不硬气？如果客户一句"再涨我就用 AWS 自家的",甲骨文就得让步，那锁定的成色已经在稀释了。

大白话

一句话收束这一章：自治数据库和 Exadata，是把老锁定往更深、更硬里做——省人、焊硬件，抄不动。这两块地基还很硬。真正拿不准的是多云互联。它同时是加固和逃生门，是同一个动作的两面。它救了甲骨文"客户上云就流失"的近渴，代价是把自己从"你必须住我这栋楼"，降级成了"我把家具搬进你选的楼里"。家具再好，房子是别人的。

所以回到本章的标题问题——数据库锁定还硬不硬？地基那两层，硬。往外扩的那层，甲骨文正在用"跟着客户走"换"别被甩掉",这笔交易划不划算，还要几年才见分晓。护城河没塌，但它的形状变了：从一圈围着自家城堡的水，变成了一条跟着客户到处流的河。河还在，只是不再由它决定往哪流。这条线索，我们留到第十五章那张护城河体检表里再一起结账。

第十四章 · 数据会是下一个十年的护城河吗

这本书从头到尾追的是一个问题：甲骨文那些又老又硬的锁，到底哪些还锁得住，哪些正在松。前一章我们把数据库本身的锁翻来覆去检查了一遍。这一章换一个更时髦、也更容易被吹上天的东西——**数据**。

你大概听过一句话，反复出现在各种 AI 时代的宣讲稿里：「数据是新石油」「谁握着数据，谁就赢了下一个十年」。甲骨文正好握着几堆看起来吓人的数据：Cerner 带来的美国医疗病历、TikTok 美国用户的数据、还有给各国政府建的主权云。听上去它简直坐在一座金矿上。

但一个写过系统的人，看到「我们手里有海量数据」这种话，第一反应应该是追问一句：**这数据到底是谁的？谁能拿它去干活、去赚钱？**「东西放在你机房里」和「东西的价值归你」，是两码事。这一章就干这一件事：把甲骨文手里这三堆数据，按「保管」还是「拥有」分个类，看看它到底是矿主，还是只是那个收保管费的保险柜——以及，它有没有在某一堆里，正从保险柜偷偷往矿主那边挪。

先分清两种生意：拥有数据，和保管数据

打个工程师熟悉的比方。你在云上开了个数据库实例，把公司的数据存进去。云厂商的机房里，物理上确实躺着你的数据——但没人会说这数据是云厂商的。它不能拿你的订单记录去训练模型、去卖广告、去做任何生意。它能收的，是**存储和运维的租金**。

> 大白话
>
> 拥有数据的价值 = 我能拿它喂给我自己的 AI、我自己的产品，直接变成钱。
> 保管数据 = 数据是别人的，我只负责把它安全、合规地存着，收一份保管费。哪天客户想搬走，法律上、道义上我都得让他搬。

这两种生意都值钱，但值钱的方式完全不同。拥有数据，是资产，是弹药；保管数据，是受托业务——你是被信任的托管方，权力和好处都有边界，法规和合同把你框得死死的。搞混这两者，就是「甲骨文坐在金矿上」这个叙事最大的漏洞。带着这把尺子，我们看三堆数据。

第一堆：Cerner 的医疗病历——最接近「拥有」，但被监管拴住

第三章埋过这个伏笔。2022 年 6 月，甲骨文以每股 95 美元、全现金约 **283 亿美元**收购 Cerner——美国最大的电子病历（EHR）厂商之一。电子病历就是把你去医院看病的所有记录（诊断、用药、检查结果、住院过程）从纸质病历搬进软件系统。Cerner 的系统装在全美大量医院里，背后是数以亿计病人、几十年积累的健康数据。

为什么说这堆最接近「拥有」？因为甲骨文不只是存着这些数据，它**卖的就是处理这些数据的软件本身**。医院用它的系统录入、查询、开单。理论上，把 AI 塞进这套软件——自动写病历、提示用药冲突、预测哪个病人要恶化——是顺理成章的，甲骨文也一直这么讲故事。

但「最接近」不等于「就是」。医疗数据是这个星球上被管得最严的数据之一。美国有 HIPAA（一部管医疗隐私的联邦法律，规定谁能碰病人数据、怎么碰、碰了要担什么责）。在 HIPAA 的框架里，甲骨文的角色叫 业务伙伴（Business Associate）——说白了就是替医院处理数据的外包方。数据的主人是医院和病人，不是甲骨文。它想拿这些病历去训练一个自己的、跨所有医院的大模型，然后卖给别人，中间横着一堆同意、去标识化、合同授权的关卡。

> 大白话
>
> 翻译成人话：甲骨文能不能用「张三这家医院」的数据，帮「张三这家医院」把它自己的活干得更好？能，这是本分。能不能把全美所有医院的病历汇成一锅，熬出一个它独家拥有、随便卖的 AI？这就撞上法律和信任的墙了——医院把数据交给你，是让你当保险柜，不是让你当矿主。

所以 Cerner 是真护城河：医院换 EHR 系统极其痛苦（想想第二章讲的切换成本），数据黏得要死，还有监管当门槛挡住新玩家。但它的价值上限，被「这数据终究是病人的」这条线压住了。它更像一门**受监管保护的、极难替换的软件生意**，而不是一座甲骨文可以随意开采的数据金矿。而且别忘了，Cerner 并入后系统整合、医院合同流失的麻烦一直没断，这门生意本身也没那么太平。

第二堆：TikTok 美国——保险柜正在往矿主方向挪的破例

上一稿这里写错了，得据实推翻重来——而改对之后，你会发现这才是本章最有意思的一堆。

故事的起点确实是纯保管。为了回应美国政府对「TikTok 数据可能流向中国」的担忧，早年间 TikTok 就把美国用户数据交给甲骨文托管、存在甲骨文的云上、接受审查（这套安排一度被叫作 德州计划（Project Texas））。那个阶段，甲骨文就是个装了监控摄像头的保险柜：里面的金条不是它的，它证明的恰恰是「我绝对没动过里面的东西」。

但到 2025 年 12 月，事情变了性质。TikTok 美国业务被拆出来，装进一个新的合资公司（JV）——一个专门为美国市场重新搭的壳。据披露的结构，甲骨文、私募基金 Silver Lake、阿布扎比的 MGX 三家各持约 **15%**，合计约 45%，成为控盘的一组新投资人；字节跳动保留不到 20%。也就是说，甲骨文**真金白银买下了 TikTok 美国约 15% 的股权（量级上约 20 亿美元），还拿到一个董事会席位**。这已经不是收保管费，这是当股东。

更关键的是算法。按这套方案，甲骨文要负责在自己的云 OCI 上**重建（retrain）那套推荐算法**，用美国用户的数据重新训练，并做安全审计。回想一下——推荐算法正是 TikTok 最值钱的那台发动机。上一稿说「那个真正值钱的推荐算法也是 TikTok 的、甲骨文碰都碰不到」，这句话在新结构下站不住了：甲骨文不但碰得到，还被指派去在自己机房里把它重新造一遍、拿真实用户数据喂它。

> 大白话
>
> 用保险柜的比方说：甲骨文原来是那个只保管、不开柜的老实保险柜。现在它买下了柜子所在这栋楼的一部分产权（当了股东），还被请去负责柜子里那台印钞机的重新组装和调试（重训算法）。它离「拥有价值」明显近了一大步——虽然远没到「金条全归我」，但也早就不是「碰都不能碰」了。

所以对 TikTok 这一堆，诚实的判断要改：它是甲骨文手里**唯一一个从「保管」越界到「部分拥有 + 运营核心算法」的破例**。别的地方甲骨文都被监管按着当保险柜，唯独这里，政治压力反而把它推成了股东兼运营方。

但破例不等于白捡的护城河，它换来的是另一种风险。这门生意的稳定性不掌握在甲骨文手里，而在华盛顿和北京的政治天平上。合资结构是永久性出售、不是临时托管，「随时换个

托管方」的旧判断已经不成立；可换来的新风险是：**甲骨文现在是把自己的一部分资本和声誉，压在了一桩高度政治化的交易和一台需要它亲手重训的算法上。**算法重训得不好、用户流失、或者哪天中美监管任何一边翻脸，这 15% 的股权和那个董事会席位就都要跟着承压。它从收租的旁观者，变成了下场的赌徒——赌注不大，但性质变了。

第三堆：主权云——保险柜生意的规模化，护城河在「牌照」不在「数据」

主权云（Sovereign Cloud）：很多国家立法要求，本国公民和政府的数据必须留在本国境内，由符合本国监管的方式运营，不能存到别国的服务器上。为了吃这块市场，甲骨文（和它的对手们）在各国建专门的、物理和运营都隔离的云区域。

这确实是甲骨文的一个真优势，而且和它整本书的性格一脉相承——它擅长**为特殊要求单独盖楼**（第五章讲过它爱自己建、不将就）。政府和银行这类客户，最看重的不是最便宜，而是「合规、隔离、可审计、出了事有人担责」。甲骨文能签下这些客户，护城河是实打实的：合同长、黏性高、有监管当护城墙。

但注意护城河**到底在哪一段**。主权云值钱，值钱的是「甲骨文拿到了在这个国家合规运营的牌照和信任」，是把保险柜生意做到了国家级规模。数据本身还是政府的、公民的——甲骨文照样是那个不能私自开柜的托管方。它卖的是**合规能力和信任**，不是数据的所有权。

> 大白话
>
> 把三堆数据的护城河到底扎在哪，摆在一起看：
> - Cerner：护城河在「难换的医疗软件 + 监管门槛」，数据的价值上限被隐私法压住，甲骨文还是保险柜。
> - TikTok：破例——从保管越界到持股约 15% + 重训核心算法，甲骨文正往矿主挪，但要额外担政治与算法运营的风险。
> - 主权云：护城河在「合规牌照 + 国家级信任」，不在数据本身，甲骨文仍是保险柜。

那这条「数据护城河」，到底该给几分

诚实的结论是：**数据是甲骨文一条真实、但被严重高估、而且边界极其关键的护城河。**再加一句补丁——这堵墙不是铁板一块，大多数段落是保险柜，但 TikTok 那一段已经破了例。

说它真实，是因为保险柜生意本身是好生意：黏（数据搬家极痛）、受监管保护（合规是高门槛，挡住新玩家）、难替换（信任和牌照要多年才攒得出来）。这些特性，和这本书前面讲的锁定逻辑是同一副骨架——只不过这次把客户锁住的不是切换成本，是**合规与信任的黏性**。

说它被高估，是因为市场把「数据在你机房里」直接翻译成了「数据的价值归你、你能喂给自己的 AI 变现」。而我们一堆一堆拆下来发现：**甲骨文手里绝大多数值钱的数据，所有权和使用权都归客户、病人、政府——它是被信任的保管者，不是可以随意开采的拥有者。** Cerner 和主权云都卡在这条线上。保险柜再满，里面的金条也不算你的资产。

说边界关键，是因为这条护城河的价值，完全取决于甲骨文站在「保管」还是「拥有」的哪一侧——而这条线基本不是它自己划的，是 HIPAA、是华盛顿的政治、是各国主权法律划的。TikTok 是唯一一次它被推过了这条线；而这唯一的破例告诉我们两件事：第一，甲骨文**确实**有可能从保险柜升级成矿主；第二，它每次这么升级，都不是靠自己把墙推倒，而是被外部政治硬塞进去的，代价是替别人的政治风险和算法运营买单。

所以回到章节标题的问：数据会是下一个十年的护城河吗？会，但它是那种**又硬又矮、且大部分不归你**的墙——硬，是因为受监管保护、极难替换；矮，是因为它的天花板被「这些数据终究不是你的」压住；而唯一顶破天花板的 TikTok，顶上去的同时也把政治与算法运营的雷抱进了怀里。把这堆数据整体当成甲骨文对冲千亿债务的救命金矿，是看错了这门生意的本质。它主要是个越来越大、越来越受信任的保险柜，只是最近在其中一格里，破天荒地当了回半个矿主。保险柜生意值得尊重，但它不是石油。下一章，我们把前面所有这些护城河——老的、新的、硬的、松的——放到同一张体检表上，冷静点名：到底哪些在变硬，哪些正在塌方。

第十五章 · 哪里在塌方

前面十四章，我们把甲骨文这家公司拆开来一块一块地看：收租的老生意、切换成本、并购、错过的云、自建的 OCI、AI 把赌注放大、RPO、发债、毛利率、循环融资、估值、客户集中度、数据库锁定、数据护城河。看得越细，越容易只见树木不见森林。所以这一章不引入任何新东西，只做一件事:把散落的判断汇总成一张表,冷静地问每一块——它在变硬,还是在塌方?

先说清楚"护城河"这个词到底指什么，别让它变成一句正确的废话。

大白话

护城河不是"这家公司很厉害"的形容词。它是一个很具体的机制问题：当竞争对手想抢走它的客户时，客户要付出多大代价才能走？代价越高，河越宽。所以每一块护城河，都可以还原成同一个问题——**客户想离开，被什么东西拽住了？**拽力在变强，就是变硬；拽力在松，就是塌方。

先分清三种"硬"

护城河的硬，不是一种硬，是三种，混在一起谈就会得出糊涂结论。

- **技术硬**：别人短期做不出同样的东西。比如把上万张 GPU 用 RDMA（一种让网卡绕过 CPU 直接读写另一台机器内存的技术，延迟极低）连成一台超级计算机，这是工程能力上的硬。
- **合规硬**：法律和监管规定数据必须放在某个地方、由某个主体保管。这不是技术领先，是"别人不许碰"的硬。
- **惯性硬**：客户懒得动、迁移风险太大、数据搬不走。这是最传统的收租式硬——也是最容易被时代慢慢腐蚀的那种。

为什么要分这三种？因为它们的**保质期完全不同**。技术硬会随着别人追上而贬值；合规硬只要法律不变就一直有效；惯性硬则取决于"搬家工具"有没有变好——一旦行业造出了标准化

的迁移通道，惯性硬会以肉眼可见的速度融化。甲骨文的麻烦，恰恰是它最厚的那层河，是保质期最短的惯性硬。

正在变硬的一侧

为 AI 负载而生的自建机房

这是技术硬。别人的云是先为通用网站、通用后台设计的，GPU 是后来塞进去的；OCI 是扁平、高带宽、为把上万张卡连成一台机器而生的。在"训练一个大模型"这种极端负载下，网络拓扑的差别会直接变成训练时间的差别。这块河这两年确实在变宽——因为需求（大模型训练）正好撞上了它当初的架构选择。**但技术硬有保质期**：微软、谷歌、亚马逊都在拼命补网络，这块领先是"暂时的几个身位"，不是"别人永远追不上"。

多云互联：把数据库送进别人的云

多云互联（Oracle Database@Azure/AWS/Google，把甲骨文的数据库直接部署进竞争对手的云机房里）是这两年最聪明的一步。它把"你要用我的数据库就得来我家"改成"我把数据库搬到你家门口"。这一步同时是变硬也是认输——变硬，是因为它降低了客户"顺便把数据库也换掉"的动机；认输，是因为它承认了客户不会全搬进 OCI。它把惯性硬续了命：客户上了 Azure，但数据库还是 Oracle 的。

主权云的合规壁垒

主权云（数据和运维完全在某个国家境内、由该国主体控制的云）是合规硬。TikTok 美国数据托管、各国政府数据，这类需求不是"谁技术好选谁"，而是"法律规定只能放这里"。合规硬最稳，因为它不靠跑得比别人快，靠的是别人根本不许上跑道。这是甲骨文护城河里质量最高的一块——可惜也是量最小的一块。

正在塌方或承压的一侧

许可证：被 SaaS 从上面持续侵蚀

这是全书起点，也是最确定的塌方。许可证租金（客户买一次软件、每年再交约 22% 的维护费）是零边际成本的印钞机。但 SaaS（软件即服务，你不买软件，按月租用别人跑好的

服务）正在从上层把它掏空——新一代客户默认租、不默认买。这不是甲骨文一家的问题，是整个卖许可证的老模式在退潮。老锁定还在收着旧客户的钱，但**新增的水管越来越细**。

云算力：一片打价格战的红海

租 GPU 算力这件事，本质上是卖标准化的电力+硬件。标准化意味着什么？意味着客户只比价格。第九章讲过毛利率被压：GPU 会折旧、机房要电费、边际成本不再是零。当四五家巨头都在建同样的机房、卖同样的算力，价格战几乎是必然。**这块业务增长最快，但也最不像护城河**——它更像在一条谁都能进的赛道上抢跑。

多云化本身的离心力

这是最微妙、也最致命的一条。甲骨文用多云互联去接住客户，但多云互联的前提，是承认了一个正在成型的行业共识：**客户再也不想被单一供应商锁死**。企业现在默认要"多云"，就是为了随时能换、能比价、能不受制于人。

> 大白话
>
> 这里有个残酷的悖论。甲骨文为了留住客户，主动把数据库搬进别人的云——这一步短期是加分的。但它同时在训练客户接受"一切皆可迁移"的世界观。而惯性硬的整个前提，是"搬家太麻烦所以不搬"。当整个行业都在造标准化的搬家工具、当客户把"不被锁死"当成采购原则，惯性硬的地基就在慢慢抽走。**甲骨文最赚钱的那层河，正在被它赖以生存的时代精神反向掏空。**

巨债与薄毛利：自己造出来的新脆弱点

这一条不在传统"护城河"清单里，但它是塌方风险的放大器。为了追赶，甲骨文背上了量级上千亿美元的债、capex 拉到上千亿、自由现金流（经营赚的现金减去为维持和扩张花掉的现金）转负。护城河讨论的是"能不能守住客户"，而债务讨论的是"守的过程中会不会先失血过多"。一个高毛利、零负债的收租王，和一个薄毛利、高负债的建云者，抗风险能力完全不同。**河还在，但守城的人自己先背上了重甲下水。**

护城河实时体检表

把上面全部压成一张能一眼看懂的对照表。每一行的格式是：**这块河 —— 属于哪种硬 —— 现在的方向**。

- **为 AI 负载而生的自建机房（OCI）** —— 技术硬 —— **在变硬**，但保质期有限，领先是几个身位不是永久。
- **多云互联（Database@别人的云）** —— 续命惯性硬 —— **在变硬**，代价是承认客户不会全搬进来。
- **主权云 / 数据合规托管** —— 合规硬 —— **在变硬**，质量最高但盘子最小。
- **自治数据库 / Exadata 的技术锁定** —— 技术+惯性硬 —— **还看不清**：把老锁定做得更硬，还是顺手给了逃生门，取决于客户到底是被绑住还是被服务。
- **手里的数据（Cerner 医疗、TikTok）** —— 合规+惯性硬 —— **还看不清**：真拥有价值，还是只是保管比特、所有权归客户。
- **传统许可证租金** —— 惯性硬 —— **在塌方**，被 SaaS 从上层持续侵蚀，存量还在、增量变细。
- **云算力（租 GPU）** —— 几乎无河 —— **红海**，标准化商品、只比价格、毛利被压。
- **客户不再愿被锁死的离心力** —— 反向作用力 —— **在塌方**，直接抽走惯性硬的地基。
- **巨债 + 薄毛利** —— 不是河，是新脆弱点 —— **在承压**，放大所有其他风险。

一句话看懂强在哪、虚在哪

把这张表再压一层，答案其实很清爽：

> 甲骨文**变硬的**，是靠技术领先和法律合规撑起来的新河——为 AI 而生的机房、多云互联、主权云；**塌方的**，是靠客户惰性撑起来的老河——许可证租金、以及"客户懒得搬家"这个前提本身。而它为了从老河跳到新河，背上了一身债。

换句话说，甲骨文正在做的，是一场**护城河的乾坤大挪移**：把最赚钱但正在腐蚀的惯性硬，换成不那么赚钱但更结实的技术硬和合规硬。这笔交换在战略上说得通——旧河迟早会干，趁还有现金流的时候挖新河，是对的。

问题只剩一个：**挪移要用债务买时间，而时间是不是够用，取决于新河挖好的速度，能不能快过旧河干涸加上利息累积的速度。**这正是最后一章要交给你的判断——不是替你下结论，而是告诉你该盯哪几个数、什么信号算赌赢、什么信号算翻车。

第十六章 · 第二曲线，还是用债务买时间

我们从第一章出发，绕了一大圈：从"躺着收租"的许可证生意，到自己盖数据中心、签下天量合同、背上上千亿债务。现在到了最后一章，该回答那个从封面就悬在头顶的问题了——甲骨文这一把，到底算什么？

有两个故事版本在市场上打架。

第一个版本叫**"老巨头成功跳上第二曲线"**。第二曲线是个商业上的老比喻：一家公司靠着某条业务（第一曲线）赚钱，但那条曲线迟早会见顶下滑；聪明的做法是在它还没塌之前，从旁边长出一条新曲线接上去，公司整体就能继续往上走。经典范本是微软——纳德拉上台，把一家卖 Windows 授权、眼看要被移动互联网甩下车的公司，硬生生扭成了云计算巨头 Azure。那是一次惊险但漂亮的换轨。按这个版本，甲骨文现在做的一模一样：许可证生意迟早被 SaaS 侵蚀，趁早用 OCI 云和 AI 算力长出第二曲线。

第二个版本叫**"用债务买时间的豪赌"**。同样的动作，换个眼光看：一家现金流没那么充裕的公司，借未来的钱，去撑一个现在还兑现不了的故事。它赌的是几年后 RPO 真能变成现金、云真能规模化盈利。如果赌对了，债务只是杠杆；如果赌错了，债务就是绞索。

> 大白话
>
> 同一堆事实——同样的债、同样的合同、同样的机房——可以讲成"英明的转型"，也可以讲成"绝望的对赌"。差别不在事实，在于你信不信那些还没发生的事会发生。这就是为什么市场"半信半疑"：不是有人蠢，是这局本来就还没开牌。

我不替你下结论，我给你一套仪表盘

诚实地说：现在没人知道答案，包括甲骨文自己。区别"第二曲线"和"债务豪赌"的那些关键变量，都在未来几年才会揭晓。所以本章不喊结论，而是交给你一套**可以自己每个季度套用的判断框架**。就像看一台正在爬坡的发动机，你不需要预言它会不会到山顶，你只要盯住几个仪表盘的读数往哪个方向走。

下面是六个表盘。每个我都告诉你：它量的是什么、往哪走算**赌赢**、往哪走算**翻车**。

表盘一：RPO 到底收没收到钱

RPO（Remaining Performance Obligations，剩余履约义务）是已经签了合同、但还没交付、因此还没收到钱的那部分未来收入。甲骨文的 RPO 在 FY2026 Q1（截至 2025 年 8 月）量级约 4550 亿美元、同比暴涨约 359%，其后据报进一步升到约 5000 多亿——这一跳几乎全靠 OpenAI 那张天量大单撑起来的，也正是整个牛市故事的地基。但记住第七章的话：**RPO 不是现金，是一张写着"以后会给"的欠条。**

- **赌赢的信号**：看每个季度有多少 RPO 真正转成了已确认收入、又收到了现金，同时 RPO 净额还在健康增长。如果这两件事同时发生——有一大块欠条稳定地"落地"变现，新欠条又持续补进来——说明地基是实的。
- **翻车的信号**：RPO 数字很大很好看，但迟迟不见转成收入；或者合同交付一再延期、被重新谈判、甚至被取消。那说明这四五千亿更像 PPT 上的数字，不是银行账户里的钱。

表盘二：自由现金流什么时候转正

自由现金流是公司经营赚到的现金，减去为了维持和扩张而必须花的资本开支（capex，主要就是买 GPU、盖机房）之后，真正剩在手里的钱。第八章讲过，因为疯狂 capex，这个数已经转负——公司花出去的比赚回来的多，缺口靠借债补。

> 大白话
>
> 这是判断"第二曲线还是豪赌"最要命的一个表盘。烧钱扩张本身不可怕，亚马逊 AWS 早年也烧。可怕的是烧了很久还不回本。

- **赌赢的信号**：自由现金流的负值先收窄、再转正，并且时间点大致踩在管理层承诺的窗口里。这说明前面砸下去的机房开始产钱，投入进入了回收期。
- **翻车的信号**：转正时间一推再推，capex 却还在往上加。那就是典型的"填不满的坑"——每一轮新合同都要求更多机房，赚的永远赶不上花的。

表盘三：客户从一家扩散到几家没有

第十二章点过这颗雷：那几千亿 RPO 高度押在 OpenAI 一个客户身上，这是单点故障（SPOF，single point of failure，指整个系统的命脉系在一个部件上，它一倒全盘皆

输）。一个客户扛起大半身家，这不是护城河，这是走钢丝。

- **赌赢的信号**：新签的大额 AI 合同来自 OpenAI 之外——别的模型公司、别的大企业。客户名单越长、单一客户占比越低，这门生意就越像"一片可持续的市场"，而不是"押注一家公司的命运"。
- **翻车的信号**：增量还是几乎全靠 OpenAI 一家加码。那甲骨文的股价，本质上就成了 OpenAI 的期权——OpenAI 打个喷嚏，甲骨文就得住院。

表盘四：毛利率能不能随规模爬回来

第九章讲透了：收租时代边际成本近乎为零，多卖一份许可证几乎纯赚。但云和 AI 不一样，毛利率（收入里刨掉直接成本后剩下的比例）被 GPU 折旧、电费、机房这些实打实的成本压下来了——边际成本不再是零。

- **赌赢的信号**：随着机房利用率上升、规模摊薄固定成本，云业务毛利率逐季往上爬。这是"规模经济真实存在"的证据，说明这门重资产生意能长成一门好生意。
- **翻车的信号**：规模上去了，毛利率却卡住不动甚至下滑。那可能是被云价格战打的（第十一章），也可能是 GPU 更新换代太快、折旧永远追着你跑。重资产没换来规模红利，就只剩下重资产的负担。

表盘五：债务和利息还扛得住吗

买时间是要付利息的。第八章提到发债、CDS（信用违约互换，可以粗略理解为"给这家公司会不会赖账买的保险"，保费越贵说明市场越担心它违约）走阔。债务本身不是罪，关键是你赚的钱够不够还利息、还能不能借到新钱。

- **赌赢的信号**：经营产生的现金相对利息支出有充足的余量，债务增速慢于业务增速，CDS 价格回落。说明市场重新相信它还得起。
- **翻车的信号**：利息吃掉越来越大比例的经营现金，评级下调，再融资成本飙升。最危险的组合是——自由现金流还没转正（表盘二），债务却到了要滚动续借的时候。**那意味着它得借新债还旧债的利息，这是雪球开始往下滚的样子。**

表盘六：循环融资的成分是升还是降

第十章的疑云：循环融资指的是钱在几家关联公司之间转圈——比如 A 投钱给 B，B 拿这钱去买 A 的产品，账面上收入很好看，但外部真金白银并没进来多少。历史上 2000 年电信泡沫里，设备商借钱给客户来买自己的设备，就是这么把泡沫吹大的。Oracle-OpenAI-Nvidia 这个三角，被不少人这样质疑。

- **赌赢的信号**：新增收入越来越多来自和这个三角无关的、纯外部的客户付款。循环的成分被稀释，说明需求是真的。
- **翻车的信号**：拆开来看，增长主要还是靠这个圈子内部互相输血。那这条第二曲线可能画在沙子上——圈子里任何一环资金链一断，整幅画一起塌。

怎么用这套仪表盘

别指望某一个表盘单独给你答案，要看它们**作为一个整体往哪个方向漂移**。理想剧本是这样一条因果链：客户从一家扩散到多家（表盘三）→ RPO 稳定转成现金（表盘一）→ 毛利率随规模爬升（表盘四）→ 自由现金流转正（表盘二）→ 债务和 CDS 压力缓解（表盘五）→ 而且这一切靠的是圈外真需求（表盘六）。六个表盘朝一个方向亮绿灯，"第二曲线"就坐实了。

翻车剧本则是反过来、而且会互相踩踏：客户还是 OpenAI 一家→ RPO 迟迟不落地→ 毛利率卡死→ 自由现金流转正无限延期→ 偏偏债务到期要续借→ 一查增长又多半来自内部循环。这几个坏信号有传染性，一个恶化会拽着别的一起恶化。

> 大白话
>
> 一句话记住：**盯"钱有没有真的从外部客户手里，流进甲骨文的账户，多到能覆盖它借钱的成本"。**其余所有指标，都是在给这一句话拆解证据。

回到第一章那笔倒过来的账

第一章我们说，甲骨文过去的生意是**"先收租、再花钱"**——先靠许可证锁定收到稳稳的现金，然后从容地花。那是一门顺序无比舒服的生意：钱先到手，风险后置。

而这一整本书讲的事，是它把这笔账**整个倒了过来——"先花钱、再祈祷收租"**。先借上千亿把机房盖起来、把 GPU 买下来、把电费付掉，然后祈祷未来那几千亿的欠条真能变成现金流回来。顺序一倒，风险的性质就变了：从"我有护城河，慢慢享用"，变成"我押上护城河，去换一张更大的门票"。

这不必然是坏事。所有真正的第二曲线，都得先经历这段"先花钱、再收租"的难看时期——AWS 如此，Azure 也如此。区别只在于：有人爬了上去，成了新王；有人爬到一半掉下来，成了教科书里的警示案例。而甲骨文特殊在，它是含着几百亿真实租金收入去下这个注的，不是白手起家——这既是它的底气，也是它输不起的地方。

所以，这本书不给你一个"买"或"卖"的结论，因为诚实的答案是：**牌还没开完。**但你现在手里有了那六个表盘。往后每个季度，甲骨文都会用财报把新的读数报出来。你不必猜它心里想什么，只要看那些数字往哪个方向走——是六盏灯慢慢转绿，还是彼此拖着往红里陷。

收租之王已经离开了它舒服的牌桌，站到了灯光更亮、赌注更大的那一张前面。它到底是在跳向第二曲线，还是只是在用债务买时间——这个判断，现在交还给你。

收租之王，负债上牌桌