# 拨开迷雾
# 看区块链

从第一笔交易，
到基于 RpcNode 的自有服务




用大白话讲 · 配有插图和示例

用大白话讲

# 拨开迷雾看区块链

从第一笔交易，到基于 RpcNode 的自有服务




这本书适合这样的你

- 听说过“区块链”这个词，却不明白它是怎么运作的；
- 想在自己的网站上收加密货币；
- 会写程序，但还没跟节点打过交道。

需要准备什么

- 好奇心——不需要任何预备知识；
- 一台带终端的电脑，用来重做书里的例子；
- 最开始的几个请求，不用注册，也不用密钥。

每一章怎么读




作者：rpcnode.dev · 2026

# 目录


## 部分 1. 为什么需要服务商

1. 想收加密货币，需要些什么？ 4
2. 什么都自己干，要花多少钱？ 8
3. 服务商能给什么，何时仍需自建 13

## 部分 2. 从零认识区块链

4. 没有中间人的钱和账 18
5. 区块与链：旧记录为何改不得 23
6. 密钥、地址和签名 28
7. 交易的旅程：从发送到区块 33
8. 网络怎样达成一致 38
9. 币、代币和智能合约 44

## 部分 3. 节点与 RPC

10. 节点有哪些种类 50
11. 怎样和节点对话 55
12. 自建节点内幕：安装、同步和故障 60
13. 怎样挑选服务商 66

## 部分 4. 各类网络如何运作

14. Bitcoin 和它的亲戚们 71
15. Ethereum：账户、Gas 和事件 76
16. Ethereum 的亲戚和“二层” 81
17. TRON：能量与稳定币 86
18. Solana：一切都分账户摆放 92
19. TON：不打电话，改写信 98
20. 别的路数：Sui、XRP Ledger 等 103
21. 该选哪个网络 109

## 部分 5. RpcNode

22. 在 RpcNode 的头五分钟 115
23. 请求经历了什么 121
24. 历史与归档数据 127
25. 套餐、调用价格和限制 133
26. 地址事件：Webhook 和套接字 139
27. 付款和交易跟踪 145

## 部分 6. 动手实践

28. 搭建店铺：从账单到确认 151
29. 事情不顺的时候 157
30. 安全与上线前检查清单 163

## 部分 7. 下一步

31. DeFi：没有中间人的金融 169

## 书末

结语 176
附录 A 术语表 178
附录 B 一页看懂各网络 180
附录 C 请求速查表 182

# 第1章 我想收加密货币，需要些什么?

## 一切从一位顾客开始

阿尔乔姆开着一家小网店：自己烘焙咖啡豆，再邮寄出去。银行卡、转账、货到付款，样样都能用。直到有一天，来了一封信：

> “你好！我想订两公斤。可以用加密货币付款吗?”

阿尔乔姆没意见。他在手机上装了个钱包，拿到一个地址，发给了顾客。几分钟后，钱就出现在屏幕上。

到了第十个订单，手动查钱包就让人烦了。要是网站能自己知道订单已经付款，该多好。




## 弄明白：什么是节点

想知道区块链上的事，就得去问保存它的人。这个“人”叫作节点。

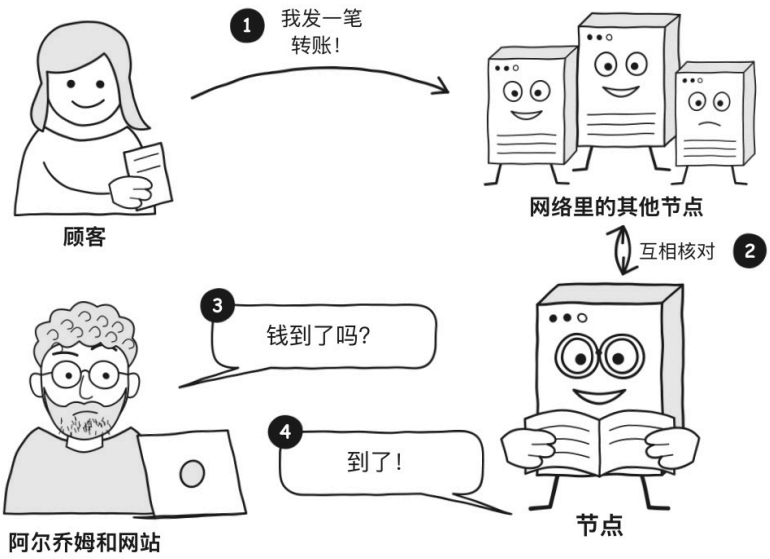
想象一本公共账本，上面记着谁给谁转了多少钱。这本账本并不锁在某一个保险柜里。世界各地有许许多多参与者各存着一份副本，并且不停地互相核对。这就是区块链，第5章我们会细讲。

**节点就是一台运行着网络程序的电脑**。它做四件事：

1. 保存自己的一份记录副本。
2. 从其他节点接收新记录，并按网络规则检查。
3. 把别人发给它的转账继续传出去。
4. 回答程序的提问：“余额是多少?”“那笔转账成功了吗?”

节点也分不同种类：有些并不保存全部历史。这个到第10章再说。

阿尔乔姆的网站并不直接翻看区块链。它向节点提问，节点作答。




## 动手试试：向节点问第一个问题

程序和节点按简单的规矩对话：发一个请求，收一个回答。其中一套规矩叫作 JSON-RPC。下面是向 Ethereum 网络节点提的一个问题：“这个地址上有多少币？”

```
curl https://节点地址 \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,"method":"eth_getBalance","params":["钱包地址","latest"]}'
```

这里的 `节点地址` 是我们要敲哪扇门，`钱包地址` 是要查谁的余额。`latest` 的意思是“以最新的时刻为准”。

节点回答：

```
{"jsonrpc":"2.0","id":1,"result":"0xde0b6b3a7640000"}
```

看着像一串乱码，其实是一个十六进制数：1 000 000 000 000 000 000。节点用币的最小单位来计数，这个单位叫 **wei**。1 个 ETH 恰好等于这么多 wei。也就是说，这个地址上有 1 ETH。

所以，全部奥妙就是一问一答。剩下的问题是：这个 `节点地址` 到底用谁的。

## 两条路：自建节点，还是找服务商

节点可以自己运行。也可以连到别人替你维护的节点上。提供这种服务的叫**服务商**。它给你一个现成的 `节点地址`。本书讲到的 RpcNode，就是这类服务之一。

| 问题 | 自建节点 | 服务商 |
|---|---|---|
| 什么时候能开始用 | 安装并同步完成之后：从几小时到几周，看是哪个网络 | 注册后几分钟 |
| 钱花在哪里 | 服务器、硬盘、流量和专业人员的时间 | 服务套餐 |
| 谁来升级和修理 | 你 | 服务商 |
| 要用多个网络 | 每个网络各要一个节点 | 通常一个控制台就够 |
| 有什么风险 | 你睡着时节点掉队了或坏了 | 依赖别人的服务 |

两条路各有用武之地。如果特别看重隐私、负载非常大，或者你要参与网络本身的运行，自建节点就是合理的。

> **阿尔乔姆**：“我是烘咖啡的，不是守着服务器值班的。不过，自己干到底要花多少钱？”

简短的回答：如果要用好几个网络，带备用服务器和完整历史，费用要以几万美元计。只跑一个小节点则便宜得多。详细的账在第2章。

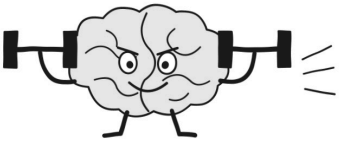
## 陷阱：“把钱包私钥交给服务商”

**永远不要**。要查余额或确认付款，节点只需要地址就够了。私钥和助记词是用来支配钱的：谁知道它们，谁就能把钱全部拿走。正经的服务商根本用不着它们。密钥的事在第6章讲。



还有第二点要说明。上面的请求只能看到网络的原生币，这里就是 ETH。如果顾客用代币付款，比如 USDT，这样问是看不到的。查代币要换一种问法，第9章会讲。

**想一想**




1. 阿尔乔姆在核对付款，而他的节点落后网络一个小时。他会看到什么？又会给顾客回什么话？
2. 店里一天三个订单。要是一天三千个呢？在自建节点和找服务商之间的选择上，会有什么变化？

*第1题提示：掉队的节点会老老实实地展示一小时前的情况。*

**本章要点**

- 程序通过向节点提问来了解区块链上的事。
- 节点可以自己养，也可以用服务商的。
- 节点需要的是地址，而不是你的私钥。

接下来：第2章。阿尔乔姆拿起计算器，算算自备服务器要花多少钱。

# 第2章 什么都自己干，要花多少钱?

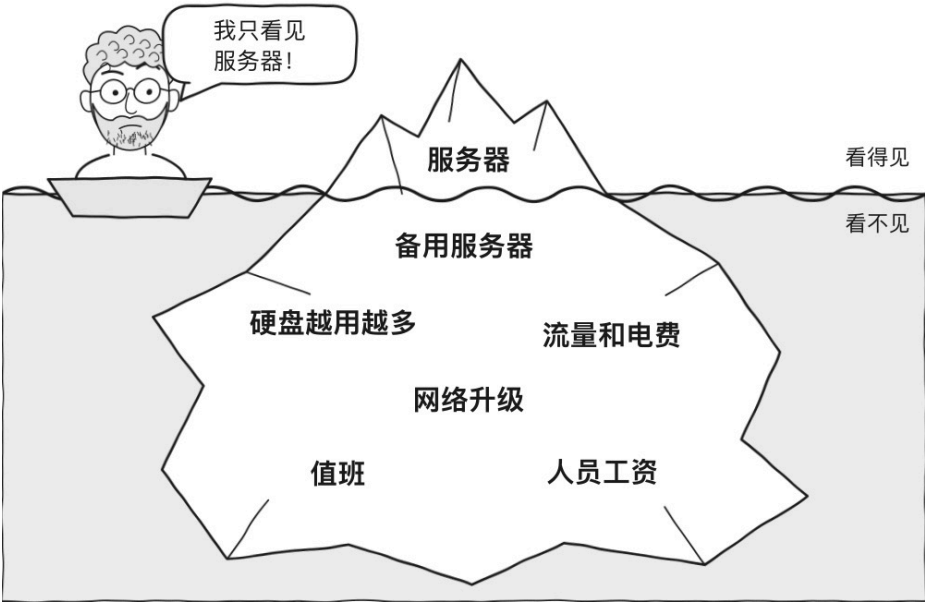
## 阿尔乔姆拿起计算器

阿尔乔姆想得很简单：“买台服务器，装上节点，收款就自动运转了。钱只花一次。”他打开计算器，开始一项项记下来。




## 弄明白：账单由哪些部分组成

服务器只是冰山一角。其余部分一开始是看不见的。




1. **服务器**。处理器、内存和高速 NVMe 硬盘。
2. **备用服务器**。一台坏了，另一台顶上回答请求，坏的那台趁机修理。
3. **硬盘占用一直涨**。区块链只会往里添新东西。Bitcoin 的链在截至 2026 年 11 月的一年里，从大约 700 GB 涨到了 786 GB。
4. **流量**。节点时刻在交换数据。TON 的轻服务器（liteserver）高峰时每月大约需要 16 TB。
5. **升级**。网络改了规则，旧程序就跟不上了。
6. **人**。总得有人配置这一切、盯着它，半夜还要爬起来。

## 每个网络要多少硬件

下面是官方文档给普通节点（不参与网络投票的节点）列出的配置要求。

| 网络 | 处理器 | 内存 | 硬盘 |
|---|---|---|---|
| Bitcoin | 1 GHz 起 | 2 GB | 建议 750 GB，而链本身已经约 786 GB |
| Ethereum | 4 核以上 | 32 GB | NVMe 4 TB |
| BNB Smart Chain | 16 核 | 64 GB | SSD 3 TB 起；完整数据库快照已约 7.4 TB，精简版约 1.9 TB |
| TRON | 16 核 | 32 GB | SSD 3.5 – 4 TB 起 |
| Solana（RPC 节点） | 16 核或更多 | 512 GB 起 | 多块 NVMe：账户数据 1 TB+，账本 1 TB+，快照 500 GB+ |

发现怪事了吗？给 Bitcoin 建议的硬盘比链本身还小。BNB 的最低要求是 3 TB，完整快照却是它的两倍还多。文档并不总能跟上网络的变化，所以硬盘要留足余量。

## 如果需要全部历史呢？

普通节点对“现在”很清楚。要问“一年前的余额是多少？”，就需要归档节点，第10章会讲。它要重得多。TON 的归档需要 16 – 20 TB。Ethereum 归档的大小很大程度上取决于用哪个程序：从 2 TB 到 14 TB 甚至更多。我们的账里没算归档：算上只会更贵。

## 算算钱

服务器可以买，但这里按租用来算。以下是 2026 年 11 月 27 日根据各主机商公开价目表做的估算，不含税费和开通费。价格是会变的。

| 服务器档次 | 配置 | 每月大约 |
|---|---|---|
| 中型 | 8 核，64 GB，2×2 TB NVMe | $180 |
| 大型 | 16 核，128 GB，4 – 8 TB NVMe | $460 – 560 |
| 超大型 | 内存 512 GB 或更多 | $1 160 – 2 350 |

现在算五个网络。价格取靠近下限的数，每个网络放两台服务器：一台干活，一台备用。

| 网络 | 档次 | 单台，每月 | 含备用，每月 |
|---|---|---|---|
| Bitcoin | 中型 | $180 | $360 |
| Ethereum | 中型 | $180 | $360 |
| BNB Smart Chain | 大型 | $500 | $1 000 |
| TRON | 大型 | $500 | $1 000 |
| Solana | 超大型 | $1 200 | $2 400 |
| 合计 | | $2 560 | $5 120 |

> **阿尔乔姆：**“每月 5 120 美元，一年就是 61 440。而且这还只是租金？！”

是的。如果按第一张表里的最高价算，一年大约 $92 000。“几万美元”就是这么来的。

只养一个网络便宜得多。Bitcoin 要求不高：比中型还低一档的服务器就够它用了。

## 那么，谁来干这些活？

服务器不会自己照顾自己。根据“Habr Career”的数据，俄罗斯中级 DevOps 工程师的月薪大约是 16 万到 29 万卢布。一个人没法全天候值班、不休周末也不休假。就算两个人，一年也要大约 380 万到 700 万卢布。

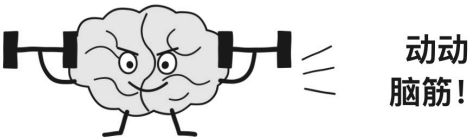
## 陷阱：“买一台服务器就完事了”

只有一台服务器，就意味着一个单点故障。硬盘满了、程序卡死了、网络升级了——网站就看不到付款了。顾客钱已经转出，订单却显示“未付款”。




还有一件常被忘记的事。付钱之前先读读主机商的条款：有些主机商禁止运行与加密货币有关的程序，可能直接把服务器关掉。

**想一想**




1. 阿尔乔姆只需要 Bitcoin，配一台中型的备用服务器。一年要多少钱？
2. 在五个网络之外又加了第六个，用每台 $500 的大型服务器，同样带备用。年度账单会增加多少？
3. 顾客一天付款三次。他们值得花这么多钱吗？

*答案：1) 360 × 12 = $4 320。2) 1 000 × 12 = $12 000。3) 由你决定，第3章会帮你。*

### 本章要点

- 钱不是付一次就完了，而是月月要付。
- 五个网络带备用，光租金一年就超过 $60 000，还不算归档和人工。
- 硬盘占用在涨，网络在升级，总得有人值班。

接下来：第3章。服务商替你扛下什么，以及什么时候还是得自建节点。

## 数字从哪里来

所有页面均于 2026 年 11 月 27 日访问。

- Bitcoin，配置要求：bitcoin.org/en/bitcoin-core/features/requirements
- Bitcoin，链的大小：api.blockchain.info/charts/blocks-size
- Ethereum：ethereum.org/developers/docs/nodes-and-clients/run-a-node
- BNB Smart Chain：docs.bnbchain.org/bnb-smart-chain/developers/node_operators/node_best_practices; github.com/bnb-chain/bsc-snapshots
- TRON：tronprotocol.github.io/documentation-en/using_javatron/installing_javatron; developers.tron.network/docs/deploy-the-fullnode-or-supernode
- Solana：docs.anza.xyz/operations/requirements
- TON：docs.ton.org/nodes/cpp/run-liteserver; docs.ton.org/nodes/cpp/run-archive-liteserver
- 薪资：career.habr.com/professions/devops（2025 – 2026 年数据）

服务器价格是作者根据几家主机商的公开价目表所做的取整估算。

# 第3章 服务商能给你什么，什么时候还是得自建节点

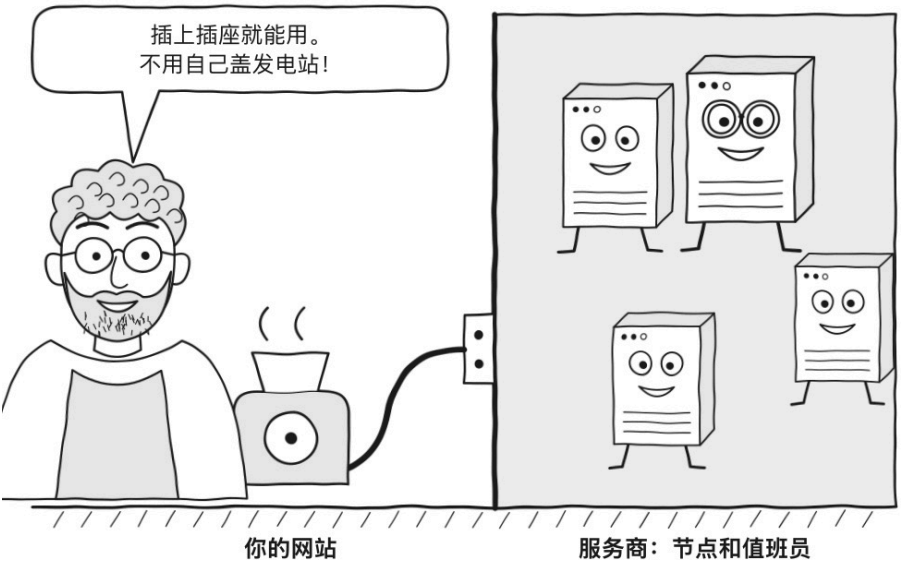
## 阿尔乔姆站在岔路口

算完第2章那笔账，计算器被屏幕朝下扣在了桌上。五个网络带备用，光租金一年就超过 60 000 美元。卖咖啡可挣不来这么多。




## 弄明白：服务商替你扛下什么

用电是从插座里取，没人会为了烧一壶水去盖一座发电站。节点也一样：服务商在自己那边养着节点，交给你一个可以提问的地址。




通常有五件操心事归它管：

1. **硬件和硬盘**。购买、租用，空间不够了就扩容。
2. **同步**。节点不能落后于网络。
3. **升级**。网络改规则时，程序要及时换。
4. **更换故障设备**。把请求转给健康的节点。
5. **值班**。半夜爬起来的人不是阿尔乔姆。

服务商各不相同，承诺的东西也不一样。这份清单里你到底能得到哪几项，要在付钱之前核实。怎么挑选，第13章会讲。

## 哪些事还得你自己负责

- **钱包密钥**。只在你手里。
- **端点地址**。服务商发给你的那个地址就叫端点。里面常常嵌着你个人的访问密钥。要像保管密码一样保管它：别人拿去发请求，耗的是你的额度。
- **你的代码**。它得经得起出错，会重试请求，并且不会把一笔付款记两次。

## 任何套餐里都有的三个词

| 词 | 意思 | 问问自己 |
|---|---|---|
| 每秒请求数（RPS） | 能问得多频繁 | 会有多少顾客同时上门? |
| 限量，也叫额度 | 一个周期里总共能问多少。不同的问题价钱不同 | 够用到月底吗? |
| 归档访问 | 能不能问很久以前的事 | 我需要老历史吗? |

## 动手试试：向真正的节点提问

第1章里地址的位置写的是 `节点地址`。现在换上一个真的。RpcNode 有免费的公共节点，不需要注册，也不需要密钥。我们来问一下 Ethereum 网络最新区块的编号：

```
curl https://public-ethereum-mainnet.rpcnode.dev \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,"method":"eth_blockNumber","params":[]}'
```

写这一章时，节点是这样回答的：

```
{"jsonrpc":"2.0","id":1,"result":"0x1946393"}
```

`0x1946393` 就是 26 502 035。你看到的数会更大：网络是不会原地踏步的。过一分钟再请求一次，自己就能看到。

公共节点是大家共用的，所以有限制：截至 2026 年 11 月 27 日，同一个地址每秒最多 5 个请求，最“贵”的那些方法不开放。用来认识一下足够了。开店则需要自己的端点。

## 要花多少钱

举个例子：这是 2026 年 11 月 27 日 RpcNode 的前两档套餐。最新价格请看官网。

| | Free | Starter |
|---|---|---|
| 月费 | $0 | $19 |
| 额度 | 1500 万 | 5000 万 |
| 每秒请求数 | 15 | 40 |
| 端点 | 3 | 5 |
| 网络 | Ethereum 和 Solana | 全部 |
| 归档访问 | 无 | 有 |

RpcNode 的节点分布在三个地区：法兰克福、美国和日本。

> **阿尔乔姆**：“每月 19 美元对 5 120 美元？这里面有什么猫腻？”

这是两样不同的东西。自己的服务器整台归你。套餐里，你得到的是公共算力中的一份，并且有限量。对小网店来说，这一份就够了，没必要为整座“发电站”买单。

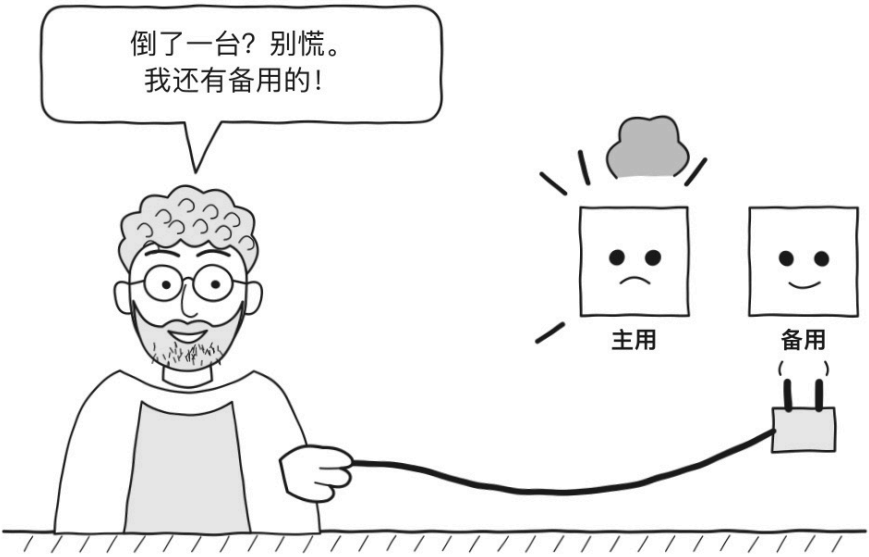
### 什么时候还是得自建节点

- **隐私**。服务商看得到你的请求：你在查哪些地址、从哪里查。
- **超大负载**。请求多到套餐费比自备服务器还贵。
- **参与网络**。你是验证者，或者在开发这个网络本身。
- **特殊配置**。需要服务商不提供的某个方法或模式。
- **法律或客户的要求**。数据只能在你自己这里处理。

也有混合的走法：主要网络用自建节点，其余网络和后备用服务商。

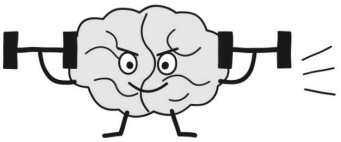
### 陷阱：“服务商永远不会挂”

什么都会挂：服务器、通信线路，服务商的服务也一样。如果收款对你很重要，就要提前想好：得不到回答时，网站该怎么办。




- 只是暂时问不到的时候，不要对顾客说“未付款”。
- 过一会儿再重试，不要一口气连发上百次。
- 准备一个备用地址：另一家服务商，或自己的节点。

### 想一想



**动动脑筋！**

1. 阿尔乔姆的网站对 20 个待付款订单每个每秒检查一次，每次检查发一个请求。每秒 15 个请求的限额装得下吗?
2. 如果节点十分钟没有回应，顾客应该看到什么?
3. 下面两家谁需要自建节点：咖啡店，还是开发网络本身的团队?

*第1题提示：20 大于 15。要么查得稀一些，要么问得聪明一些：怎么做，见第26章。*

### 本章要点

- 服务商扛下硬件、同步、升级和值班。
- 密钥、端点地址，以及出故障时网站如何表现，仍由你负责。
- 为了隐私、大负载、参与网络或特殊要求，才需要自建节点。

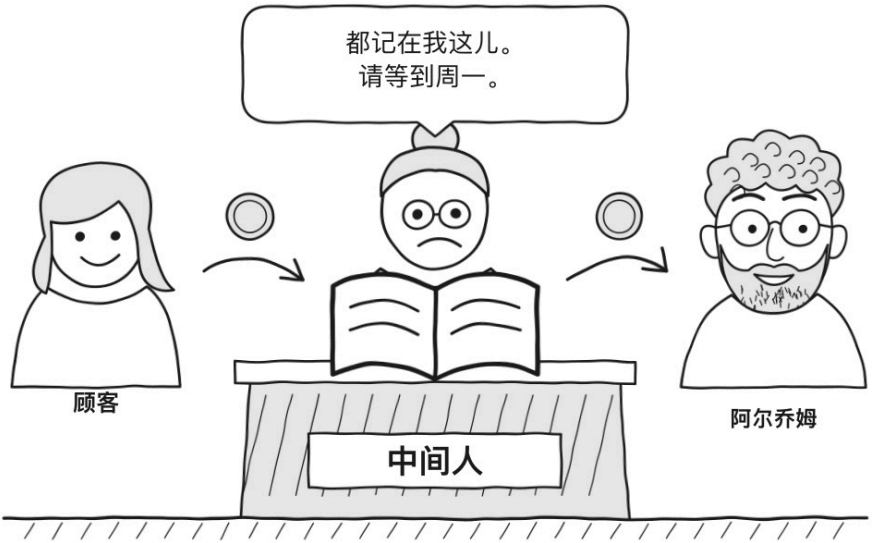
**第一部分到此结束。**阿尔乔姆选了服务商，也从真正的节点那里得到了回答。可它报出的编号为 26 502 035 的“区块”到底是什么？**接下来：**第二部分，第4章。没有中间人的钱和账。

# 第4章 没有中间人的钱和账

## 你的钱到底在哪里?

阿尔乔姆收到一笔打到银行卡上的转账。没有任何一张钞票挪过地方。那到底发生了什么?有人把顾客名字旁边的数字改小了,把阿尔乔姆名字旁边的数字改大了。

**非现金的钱,就是一条记录**。替大家记账的那一方叫中间人:银行或支付服务。




## 弄明白:为什么需要中间人,它又有什么不方便

中间人做了很多有用的事。检查钱够不够。不让同一笔钱花两次。还能处理纠纷、退回付款。

可是所有记录都攥在它手里。它可以叫停转账、收取手续费、冻结账户,或者周末不营业。顾客住在另一个国家时,这些问题尤其明显。

> 阿尔乔姆:“那就让顾客把钱当文件发给我好了。像发照片一样。什么中间人都不用!”

## 为什么不能直接发个文件

一张照片可以发一百次,你手里照样还有。钱可不能这样:否则一枚币就能付给两个人。这叫作双重支付。

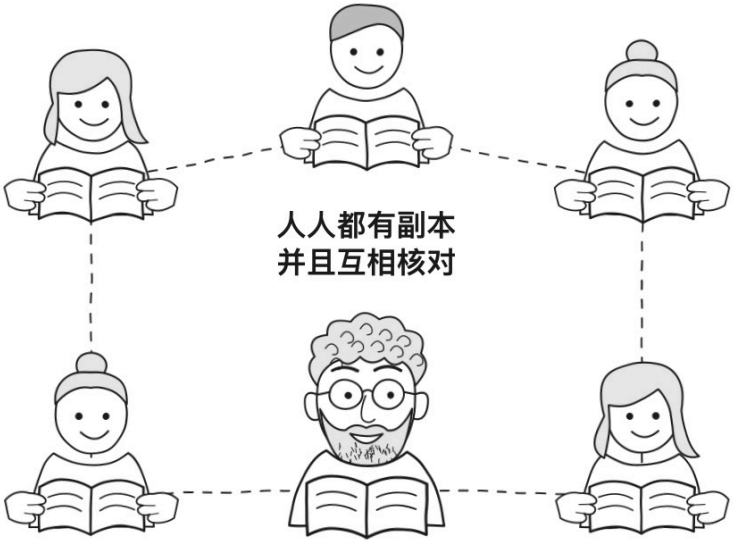



很长一段时间里，解决办法只有一个：由某个说了算的人记一份清单，宣布这枚币现在归谁。也就是说，又回到了中间人。

## 思路：让大家一起保存记录

2008 年，一位化名中本聪的作者描述了另一条路，2009 年 1 月 Bitcoin 网络开始运行。思路是这样的：

1. 转账清单不藏在某一个人手里。**很多人各有一份副本**。
2. 每笔新转账，参与者都**按共同的规则检查**：有没有钱，这钱是不是已经花过了。
3. 参与者在没有头儿的情况下**商量出一致意见**：哪些记录是对的。




这样一份由大家共同维护的公共清单，就叫作区块链。记录怎样串成链，第5章讲；参与者怎样达成一致，第8章讲。

### 动手试试：公共账本游戏

不需要写代码。叫上两位朋友，每人发一张自己的纸。开始时大家记的都一样：安雅有 10 个币，鲍里斯 5 个，薇拉 0 个。规则只有一条：**给出去的不能比手里有的多**。

| 序号 | 谁想转账 | 接受吗? | 安雅 | 鲍里斯 | 薇拉 |
|---|---|---|---|---|---|
| 开始 | | | 10 | 5 | 0 |
| 1 | 安雅 → 薇拉 4 | | | | |
| 2 | 薇拉 → 鲍里斯 1 | | | | |
| 3 | 鲍里斯 → 安雅 7 | | | | |
| 4 | 安雅 → 鲍里斯 6 | | | | |

把空格填上，然后互相对一对。如果三个人的纸一模一样，恭喜，你们刚刚在没有头儿的情况下达成了一致。答案在本章末尾。

## 区块链给什么，不给什么

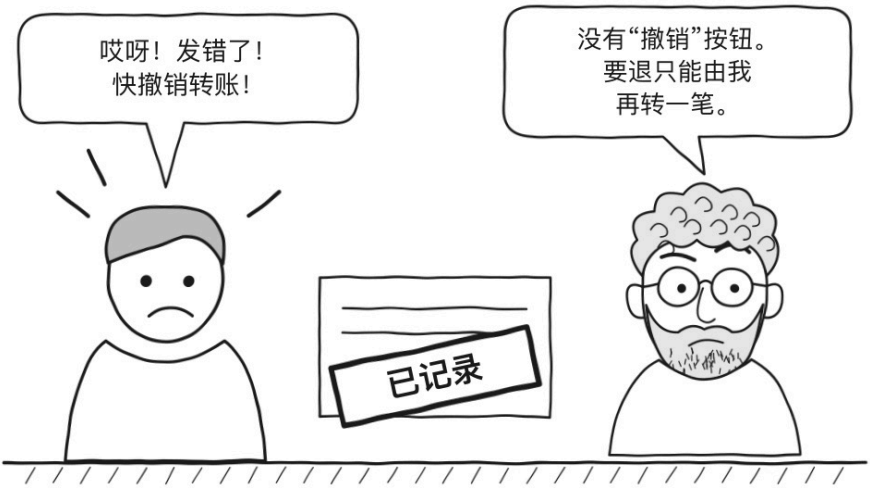
| 给 | 不给 |
| --- | --- |
| 直接转账，随便哪天几点都行 | “撤销”按钮 |
| 同一个账户在不同国家都能用 | 稳定的币价：行情可能大起大落 |
| 人人都能核查的记录 | 隐蔽性：任何地址的转账大家都看得见 |
| 不必信任某个人来保管清单 | 能帮你找回丢失密钥的客服 |

也不是所有币都上蹿下跳：有些币尽力把价格维持在一美元左右，叫作**稳定币**。但最常见的那几种背后站着一家发行公司，它可以冻结某个地址上的钱。详情见第9章。

中间人并没有消失：交易所、钱包、服务商都还在。但它们各不相同。交易所可能替你保管币。第一部分讲的服务商不碰你的钱：它只是把你的问题转给节点。

### 陷阱：“不满意我就撤销”

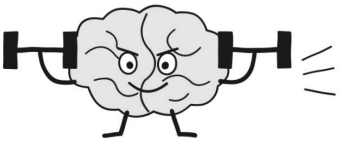
已经确认的转账无法撤销。地址写错了，钱就走了。能把钱还回来的只有收款人——前提是他愿意，而且那确实是某个人的地址。




所以地址要核对两遍，大额转账先拿一小笔试一试。

*一个小故事。2010 年 5 月，有人花 10 000 个比特币买了两块比萨。那笔转账至今还记在公共账本里。*

### 想一想



**动动脑筋！**

1. 游戏结束后薇拉有 3 个币。她同时给安雅写“我给你 3 个”，又给鲍里斯写“我给你 3 个”。参与者应该怎么办？
2. 如果一个参与者接受了第一笔转账，另一个接受了第二笔呢？谁对？
3. 对阿尔乔姆来说，付款能撤销方便，还是不能撤销方便？对他的顾客呢？

*提示：1) 只能接受其中一笔。2) 这是最难的问题，第8章专门讲它。*

### 本章要点

- 非现金的钱就是记录，总得有人记账。
- 区块链是许多人按共同规则一起维护的公共清单。
- 摆脱中间人也意味着责任自负：没有人能替你撤销转账、找回密钥。

*游戏答案：转账 1 接受（6、5、4）；2 接受（6、6、3）；3 拒绝，鲍里斯只有 6 个（6、6、3）；4 接受（0、12、3）。币的总数仍然是 15。*

**接下来：**第5章。节点报出的“区块”是什么，旧记录为什么没法悄悄改写。

# 第5章 区块与链：旧记录为什么没法悄悄改写

## 就是那个编号

第3章里节点回答阿尔乔姆：最新区块的编号是 26 502 035。区块是什么？它为什么有编号？又是什么拦着别人偷偷改掉账本里的记录？

## 弄明白：区块就是账本的一页

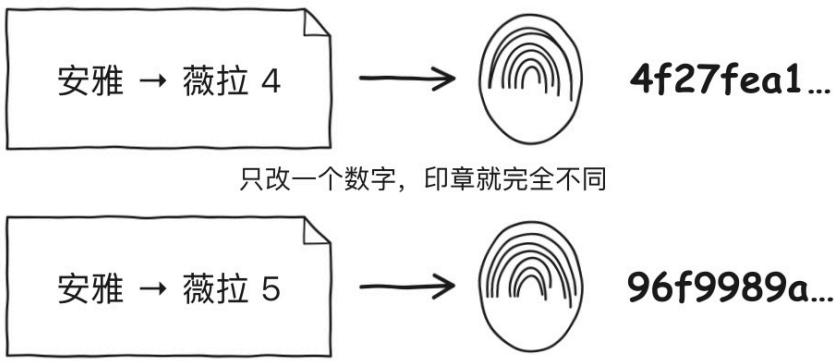
转账并不是一笔一笔单独记的。它们会攒成一批，就像一行行字凑成一页纸。这样的一页叫作区块。上面有：

1. 转账以及其他操作，统称交易。
2. 页码，按顺序排。
3. 生成的时间。
4. 上一页的印章。全部秘密都在这里。

## 印章，也就是哈希

任何一段文字，都可以算出一小串由数字和字母组成的字符。这就是哈希，我们把它叫作“印章”。它很像指纹：

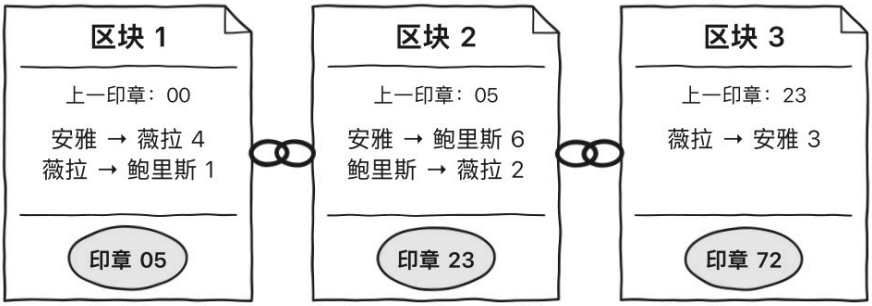
- 同一段文字，印章永远相同；
- 哪怕只改一个字母，印章就完全不一样了；
- 无法从印章反推出原文；
- 想另找一段文字盖出同样的印章，实际上办不到。



*图中两行文字下面，是它们真实的 SHA-256 哈希的开头部分。Bitcoin 用的是 SHA-256；其他网络也用别的方法盖章。*

## 链

每一页上都记着上一页的印章。而本页自己的印章，是根据页上的全部内容算出来的，其中也包括上一页的印章。

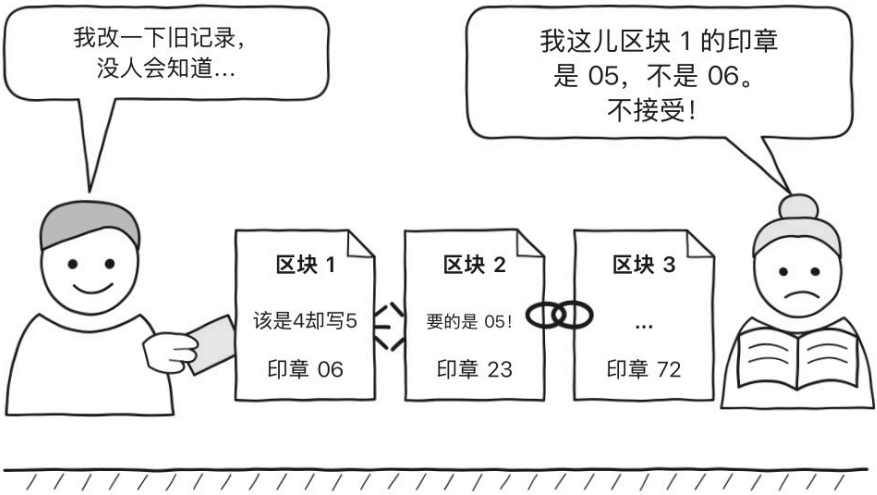



这样，一页页就扣在一起，成了区块链。英文叫 blockchain。名字就是这么来的。

*图上的印章是玩具版的，只有两位数：这样心算就能算出来。*

## 如果动了旧记录会怎样

骗子把第一个区块里的 4 改成 5。这个区块的印章随之改变。可第二个区块里记的还是旧印章。链断了，而且一眼就能看出来。




他也可以把后面所有印章都重写一遍。但别的参与者手里有自己的副本，跟他这一版对不上。要让大家接受他的版本，就得压过网络里的其他所有人。这代价非常高。网络到底怎样判定哪条链是对的，第8章讲。

## 看一看真实的链

这是 Ethereum 网络 2026 年 11 月 26 日的三个区块。印章很长，所以只显示开头和结尾。

| 区块编号 | 本块印章 | 上一块印章 | 交易数 |
|---|---|---|---|
| 26 502 033 | `0xbe58…d764` | `0x2844…c876` | 229 |
| 26 502 034 | `0x7bb7…b7a4` | `0xbe58…d764` | 229 |
| 26 502 035 | `0xeab2…c904` | `0x7bb7…b7a4` | 515 |

用手指顺着看：上一行的印章，会在下一行再次出现。这几个区块之间正好各隔 12 秒。Bitcoin 的新区块出得慢一些，平均大约 10 分钟一个。

最早的那个区块没有“上一块”。Bitcoin 的第一个区块生成于 2009 年 1 月 3 日。

## 动手试试：自己盖章

我们来定一条玩具规则：

> 印章 =（上一个印章 × 3 + 本页所有金额），结果取最后两位数。

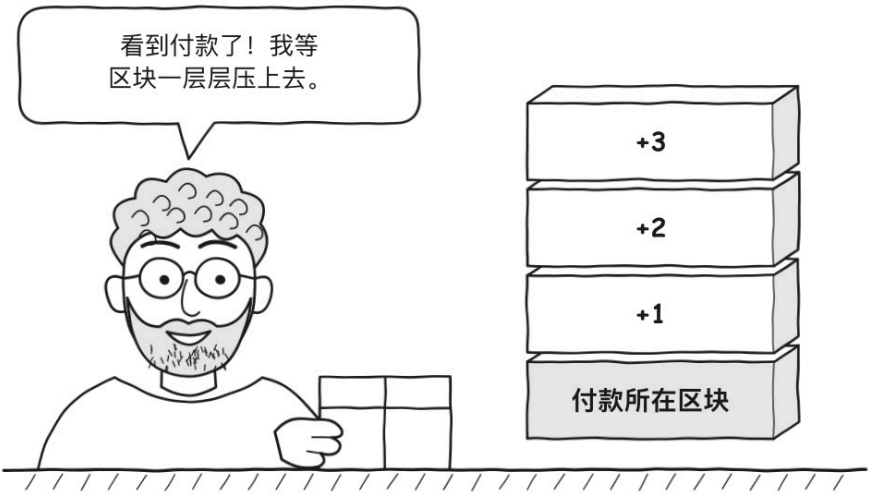
| 区块 | 转账金额 | 印章 | 如果把区块 1 里的 4 换成 5 |
|---|---|---|---|
| 1 | 4 和 1 | | |
| 2 | 6 和 2 | | |
| 3 | 3 | | |

第一个区块的“上一个印章”记为 0。把两列都填上。有几个印章变了？

## 陷阱：“进了区块，就永远在了”

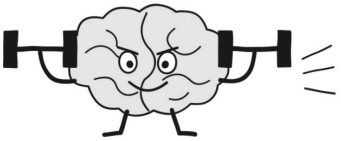
最新的那几个区块有时会被别的区块替换掉：网络的各个部分还没来得及达成一致。压在上面的区块越多，你那条记录就越难被拿掉。

这用**确认数**来计算：包含付款的区块本身算第一个确认，之后每多一个区块就加一个。




要等多久，取决于网络和金额。一杯咖啡的钱，等得少一点；一大批货款，等得久一点。详情见第8章。

### 想一想



**动动脑筋！**

1. 骗一骗我们的印章：把区块 1 里的金额 4 和 1 换成别的数，让印章保持不变。这说明我们的规则有什么问题？
2. 为什么改三年前的区块比改昨天的难？
3. 如果每 12 秒出一个区块，一小时能出多少个？

*答案：1) 比如 3 和 2：总和一样。真正的哈希能发现这种改动。2) 得把压在它上面的所有区块全部重做。3) 300 个。*

### 本章要点

- 区块就是一页账，上面有交易、编号、时间和上一页的印章。
- 改动旧记录会让印章改变，链随之断开。
- 对刚出的区块要少信几分：先等确认。

*游戏答案：印章依次是 05、23、72。改动之后是 06、26、81：三个全变了。*

**接下来：**第6章。网络怎么知道发出转账的正是钱的主人？密钥、地址和签名。

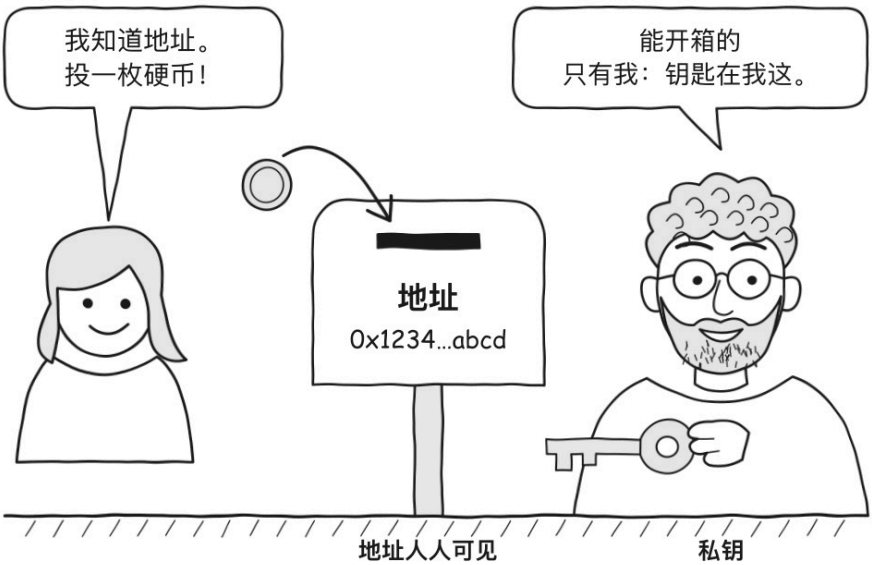
# 第6章 密钥、地址和签名

## 网络怎么知道这是你的钱?

公共账本里没有身份证。没人问姓名，也没人核对照片。那么，是什么拦着随便哪个人写上一句“阿尔乔姆把所有钱都给我”?

## 弄明白：信箱和钥匙

想象街边的一个信箱。信箱号大家都知道，谁都可以从投递口往里丢硬币。可是能打开信箱把硬币取走的，只有拿着钥匙的人。



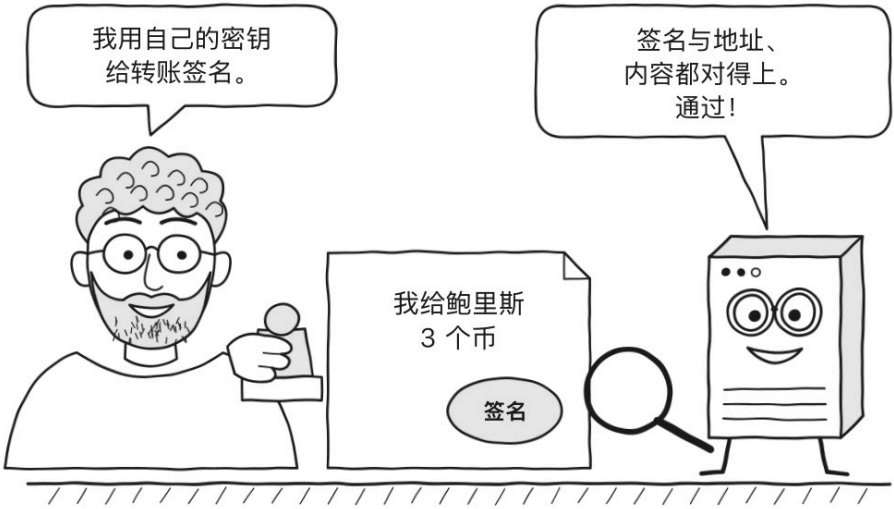

| 是什么 | 像什么 | 可以给谁看 |
| --- | --- | --- |
| 地址 | 信箱号 | 给谁都行，用来收钱 |
| 公钥 | 用来验签名的样本 | 可以公开 |
| 私钥 | 信箱钥匙 | 谁都不给 |

它们之间是单向关联的。由私钥得出公钥，由公钥得出地址。反过来推算，实际上办不到。在有些网络里，地址就是公钥本身，只是换了一种特殊的写法。

钱包是保存密钥的程序或设备。里面并没有币：币仍然只是公共账本里的记录。

## 签名

要把币给出去，主人先写好一笔转账，再用私钥给它签名。私钥本身不会发到任何地方。发进网络的只有转账和签名。




节点检查两件事：签名是不是用这个地址对应的密钥做的，签完之后内容有没有被改过。一笔转账的签名套不到另一笔上。同一笔转账，网络也不会执行第二次。

## 动手试试：玩具签名

用计算器就能算。私钥是数字 7，公钥是数字 3。只有前者需要保密。

> 规则：反复乘以选定的数，每乘一步，都只保留除以 33 的余数。

**给消息“4”签名。**把 4 连乘起来，乘几个由私钥决定，也就是 7 个：

| 几个 4 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 余数 | 4 | 16 | 31 | 25 | 1 | 4 | 16 |

每一步都是用上一步的余数乘以 4。比如第三步：16 × 4 = 64，除以 33 余 31。第七步之后得到签名：16。

**验证。**把签名 16 连乘，个数由公钥决定，也就是 3 个：先是 16，接着 16 × 16 = 256，余 25，再算 25 × 16 = 400，余 4。正好是我们的消息。签名是真的！

现在有人把消息改成“5”，签名 16 原样保留。验证算出来的还是 4，不是 5。伪造一目了然。

*这只是演示思路。这么小的数，中学生就能破解。真实网络里的密钥是巨大的数字，用的数学也不一样。*

## 助记词

私钥是一长串字符，抄起来很不方便。所以很多钱包会给出由 12 或 24 个单词组成的**助记词**。凭它可以恢复密钥和所有地址。




- 谁知道助记词，谁就是钱的主人。
- 助记词和设备都丢了，就没有人能帮你找回。
- 不要拍照，不要存在备忘录或聊天记录里。
- 不要在网站上输入，也不要念给“客服”听。真正的客服不会问它。

密码可以改，私钥改不了。一旦泄露，只能赶紧把钱转到新地址。

## 对网店意味着什么

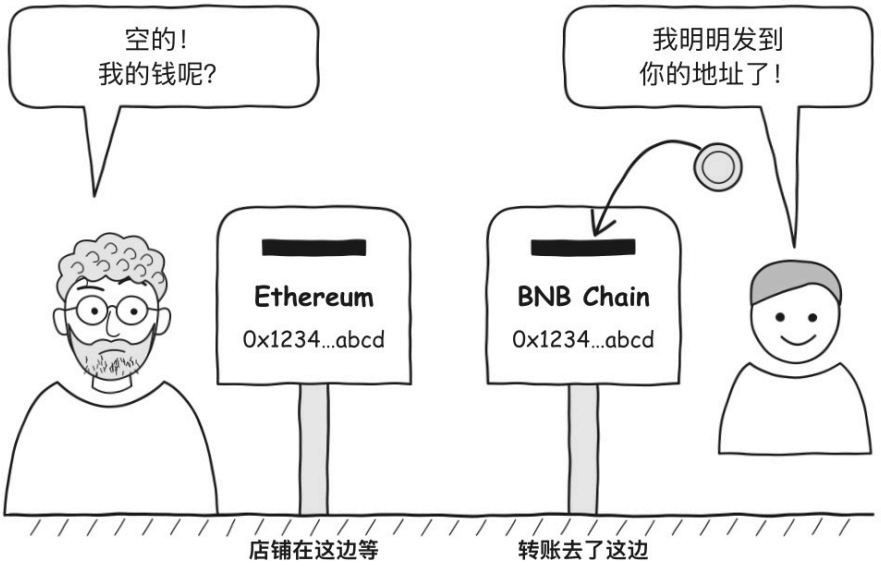
阿尔乔姆的网站要收钱、看到付款，只需要地址。店铺服务器上用不着私钥，也不该放在那里：网站一旦被攻破，钱也一起被拿走。给每个订单分配一个单独的地址很方便，第28章会讲。

## 陷阱：“地址一样，就能到账”

不同网络的地址长得不一样：

| 网络 | 地址开头 |
| --- | --- |
| Bitcoin | `1`、`3` 或 `bc1` |
| Ethereum 及类似网络 | `0x` |
| TRON | `T` |
| TON | 通常是 `EQ` 或 `UQ` |
| Solana | 没有特定开头 |

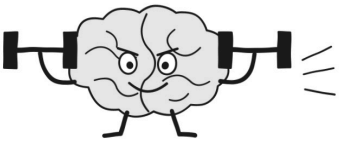
可是在 Ethereum、BNB Smart Chain、Polygon 以及许多其他网络上，地址看起来一模一样。而它们的账本各是各的。顾客转错了网络，店铺就看不到这笔付款。




所以账单上不光要写地址和金额，还要写明网络。

骗子还有一招：从一个长得很像你熟悉地址的地址，给你转一笔极小的钱。指望你以后从历史记录里把它复制过来。核对地址要看完整的一串，不能只看开头和结尾几位。

### 想一想




1. 用密钥 7 给消息“2”签名，再用密钥 3 验证。
2. 阿尔乔姆把店铺地址印在了名片上。这危险吗？
3. “钱包客服”为了追回一笔卡住的转账，向你索要助记词。怎么办？

*答案：1) 签名是 29，验证后又得到 2。2) 钱不会因此被偷，但谁都能看到这个地址收到过什么。3) 什么都别说：这是骗子。*

### 本章要点

- 地址可以给所有人看，私钥和助记词谁都不给。
- 签名证明这笔转账是密钥主人授权的，而且没有被改过。
- 网络和地址一样重要。

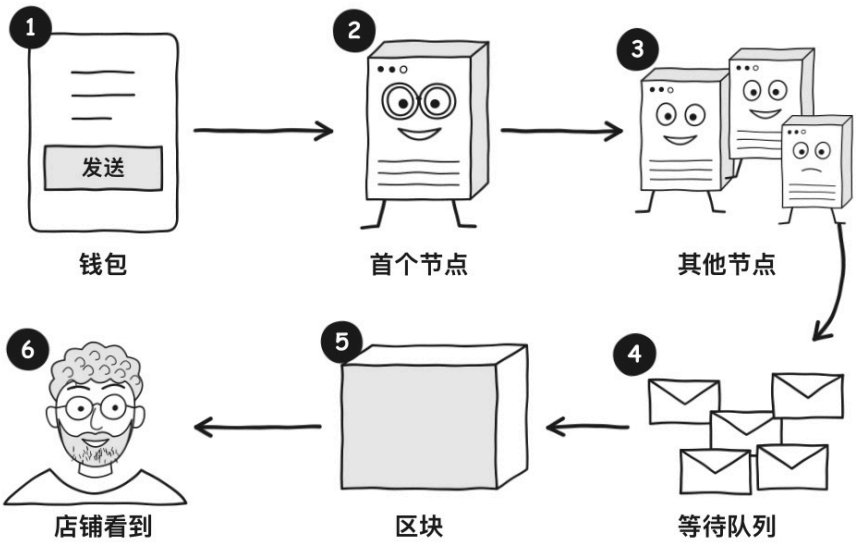
**接下来：**第7章。转账签好了。从按下“发送”到写进区块，它会经历什么？

# 第7章 交易的旅程：从“发送”按钮到区块

## “我发了！”“我这边是空的”

顾客按下按钮，发来消息：“发过去了。”阿尔乔姆一看，钱没到。有人在骗人吗？多半没有。转账还在路上。

## 弄明白：六个站点



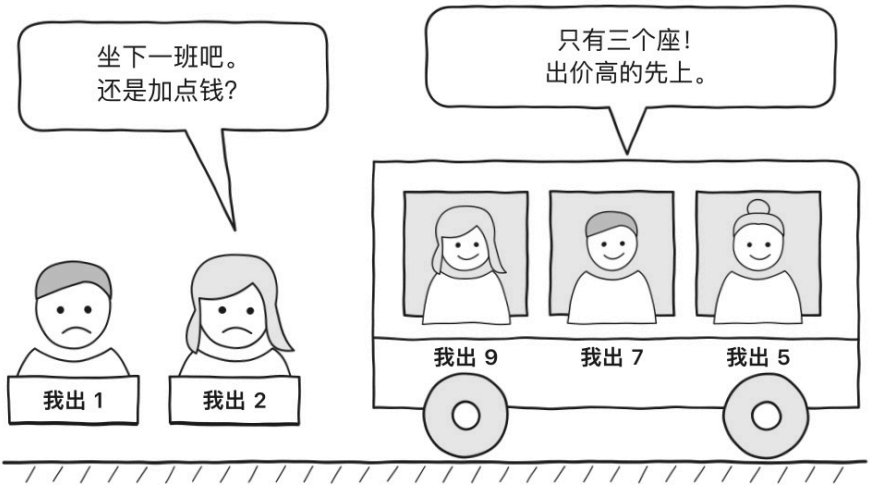

1. 钱包写好转账，用密钥签名。
2. 第一个节点收到转账，进行检查：签名对不对，钱够不够。
3. 它把转账传给其他节点，那些节点再往下传。
4. 转账在队列里排队。这个队列英文叫 mempool，即内存池。
5. 轮到谁来拼这一页，谁就把转账放进区块。
6. 店铺看到它，开始等待确认。

各个网络的叫法和细节有所不同：比如有些网络没有公共队列，转账直接送到负责打包区块的那一方手里。

每笔转账都有自己的“印章”——交易哈希。它就像快递单号：凭它可以查到转账现在到哪儿了。

## 手续费：公交车上的座位

一个区块装不下所有交易。想象一辆只有三个座位的公交车，站台上却有五个人。




要挤进区块，发送方得向网络付手续费。想上车的人越多，座位越贵。出价低的只能等下一班车，有时到最后也没等上。

- 手续费由发送方付，店铺拿不到。
- 它和转账金额无关，取决于交易的“分量”和网络的繁忙程度。
- 这个“分量”怎么算，每个网络各有一套：本书第四部分会讲。

## 看一笔真实的转账

这是 Ethereum 网络上一笔普通转账，位于区块 26 502 085。我们凭交易哈希向节点查询它：

```
curl https://public-ethereum-mainnet.rpcnode.dev \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,
       "method":"eth_getTransactionReceipt",
       "params":["转账哈希"]}'
```

回答很长。下面是其中的要点，已换算成普通数字：

| 字段 | 值 | 意思 |
| --- | --- | --- |
| `status` | `0x1` | 执行成功 |
| `blockNumber` | 26 502 085 | 在哪个区块里 |
| `gasUsed` | 21 000 | 简单转账的“分量” |
| 手续费 | 0.000043 ETH | “分量” × 单价 |
| 转账金额 | 0.000041 ETH | 取自交易本身 |

发现了吗？这一次，为转账付的钱**比转出去的钱还多**。里面装了多少钱，手续费并不在乎。

而且这个价格一直在变。自己问问看：

```
curl https://public-ethereum-mainnet.rpcnode.dev \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,"method":"eth_gasPrice","params":[]}'
```

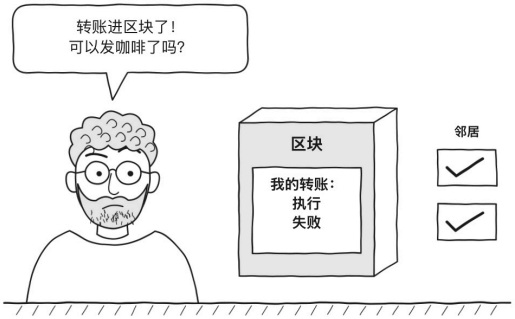
节点会报出当前每单位“分量”的价格，以 wei 计。而同一天，在 RpcNode 的 Bitcoin 公共节点上，队列里等着近 83 000 笔转账。

## 转账可能遇到的情况

| 状态 | 意思 | 店铺该怎么办 |
| --- | --- | --- |
| 等待中 | 还在队列里 | 等着，先别发货 |
| 已进区块 | 记上了，但确认数还少 | 检查是否成功，再等一等 |
| 已确认 | 上面压了足够多的区块 | 发货 |
| 执行失败 | 进了区块，但转账没有完成 | 不能算作付款 |
| 掉出队列或被替换 | 没能进区块 | 请顾客查看钱包 |

## 陷阱：“进了区块就是付过了”

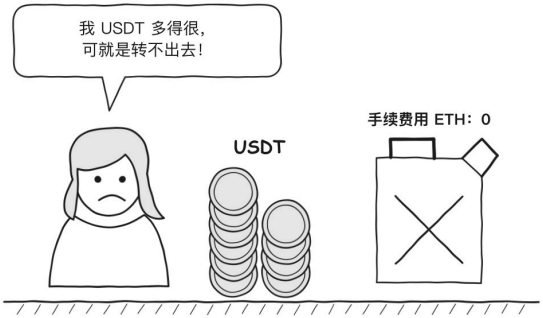
在 Ethereum 及类似网络上，转账可能进了区块，却仍然执行失败。比如顾客的代币不够。手续费照扣，钱却到不了你手里。



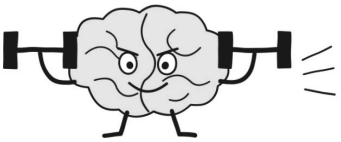

所以不能只看有没有进区块，还要看结果：上面例子里就是 `status` 字段。此外还要核对收款人、金额、币种和网络。

## 第二个陷阱：油箱空了

手续费收的是网络的原生币。在 Ethereum 上就是 ETH，哪怕你转的是 USDT。在 TRON 上，费用用 TRX 和网络特有的资源支付。



### 想一想



**动动脑筋！**

1. 一个区块能装三笔转账。队列里大家为座位出价 4、9、1、7 和 2。谁能搭上第一个区块？
2. 顾客发来交易哈希和一张显示“成功”的截图。凭这些就能发咖啡了吗？
3. 顾客在 Ethereum 网络上有 100 USDT，却一点 ETH 都没有。他该怎么办？

*答案：1）9、7 和 4。2）不能：截图可以伪造，要在自己这边查哈希。3）先弄一点 ETH 来付手续费。*

### 本章要点

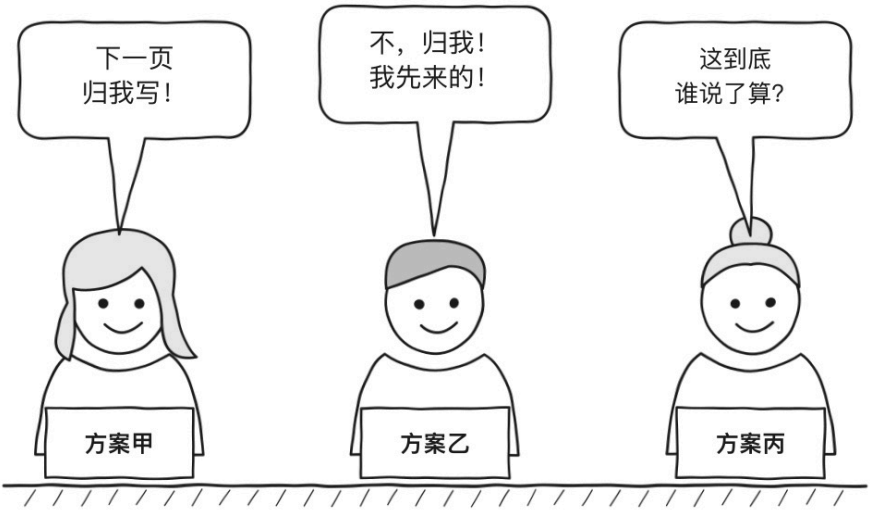
- 从“发出”到“收到”之间有一段路：节点、队列、区块、确认。
- 手续费买的是区块里的座位，和金额无关。
- 付款成立，是指在正确的网络上、用正确的币、成功完成转账，并且确认数足够。

接下来：第8章。区块由谁来打包，大家又怎么一致认定哪条链才算数？

# 第8章 网络怎样达成一致

## 下一页由谁来写?

公共账本没有头儿。成千上万的参与者各有副本，每个人都可以提出自己的“下一页”。哪一页算真的?



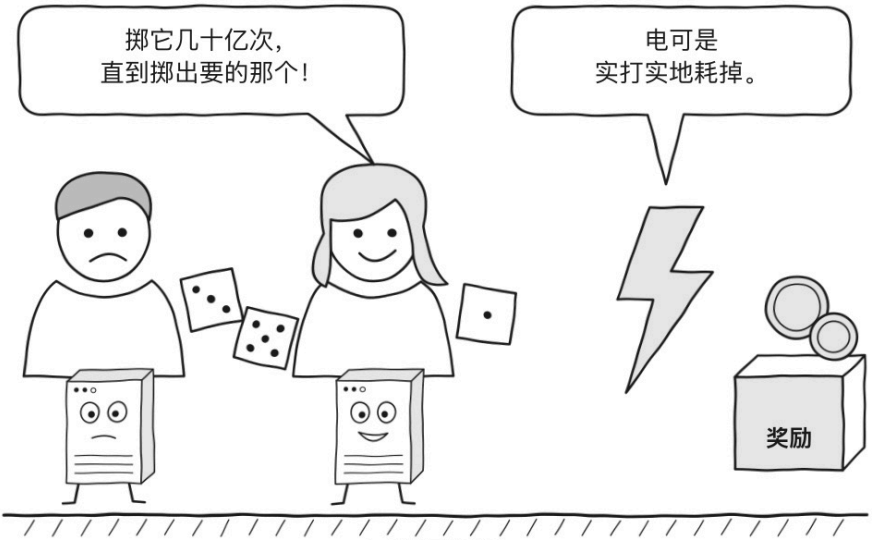

## 弄明白：共识

参与者得出统一答案所依据的规则，叫作**共识**。它的目标是：让作弊比老实参与更不划算。

单纯数人头行不通：骗子开一百万个程序，就有一百万张票。所以投票必须付出代价。主要有两种办法。

## 办法一：工作量证明

英文叫 Proof of Work，这个过程叫**挖矿**。参与者比赛解一道没法靠巧劲、只能一个个试的题。就像不停掷骰子，直到掷出一个罕见的组合。



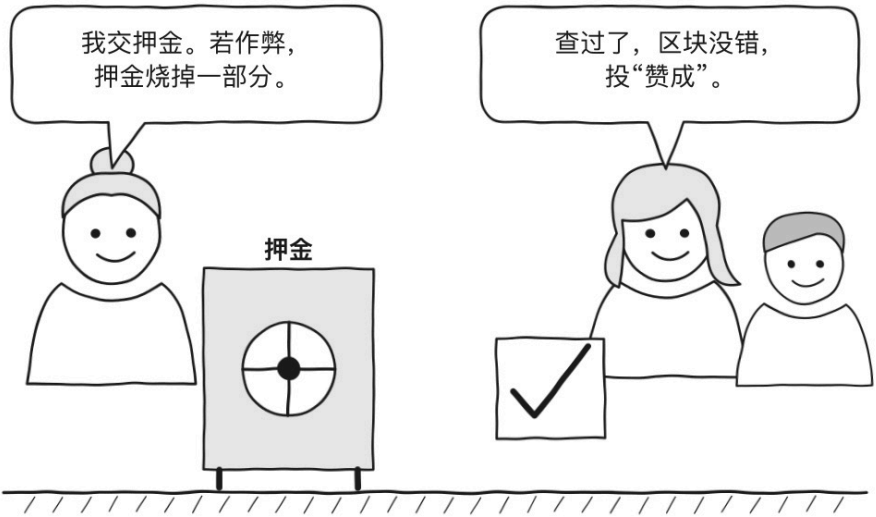

工作量证明

赢家添加区块，并获得奖励：新币和手续费。验证他的答案很容易，找出答案却很费钱。Bitcoin 就是这样运作的：

- 平均每 10 分钟出一个区块；
- 每 2016 个区块调整一次题目难度，好让节奏保持不变；
- 目前每个区块奖励 3.125 BTC 新币，大约每四年奖励减半一次。

### 办法二：权益证明

英文叫 Proof of Stake，也就是质押。参与者不耗电，而是押上自己的币。这样的参与者叫验证者。验证者轮流提出区块，其他人检查并投票。



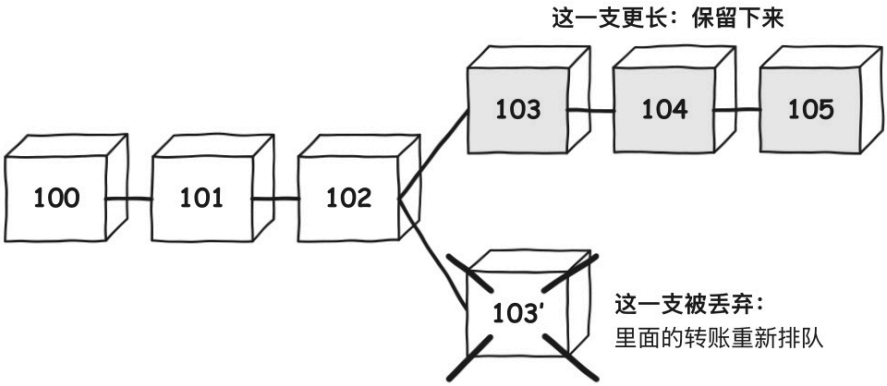

**权益证明**

严重违规时，押金会被销毁一部分，这叫罚没（slashing）。Ethereum 在 2022 年 9 月 15 日改用这种办法，网络耗电量下降了约 99.95%。要想自己当验证者，需要 32 ETH。

也有一些变体。在 TRON 上，区块由持币人投票选出的 27 位超级代表来写。BNB Smart Chain 上的验证者也只是一个人数有限的圈子。

### 两页同时出现的时候

网络很大，消息传开需要时间。有时会出现两个编号相同的区块，链暂时分成两岔。随后一岔胜出，另一岔被丢弃。这叫作链重组。




被丢弃区块里的转账不会就此无影无踪：通常它会回到队列，再进到另一个区块。但也可能进不去。第5章讲的确认数，用处就在这里。

*图上胜出的是较长的那一支。更准确地说，Bitcoin 选的是累积工作量更多的那条链，质押类网络选的是获得更多质押投票的那条。*

## 最终性：什么时候就再也撤不掉了

最终性是指一个时刻，过了它，区块就不会再被替换。有些网络会明确宣布这一时刻，另一些网络则只是随着区块增多而越来越确定。

| 网络 | 大约何时认为记录已成定局 |
|---|---|
| Bitcoin | 没有明确标记。大额建议等 6 个确认，约一小时 |
| Ethereum | 通常 13 – 15 分钟，有时更久 |
| TRON | 通常约一分钟 |
| BNB Smart Chain | 约一秒或更快 |
| Solana | 2026 年升级后，设计目标是不到一秒 |
| TON | 约一秒 |

*据各网络文档，数据截至 2026 年 11 月 27 日。这些数字会随升级而变化。*

## 动手试试：问三种“新鲜度”

向 Ethereum 节点要的不一定只是最新区块。第1章里的 `latest` 还有两个亲戚：`safe`（已经比较可靠）和 `finalized`（已成定局）。

```
curl https://public-ethereum-mainnet.rpcnode.dev \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,
       "method":"eth_getBlockByNumber",
       "params":["finalized",false]}'
```

把 `finalized` 换成 `safe` 和 `latest`，比较三个编号。写这一章时，节点是这样回答的：

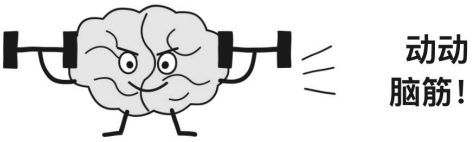
| 问的是什么 | 区块编号 | 比最新落后 |
|---|---|---|
| `latest` | 26 502 104 | — |
| `safe` | 26 502 070 | 34 个区块，约 7 分钟 |
| `finalized` | 26 502 038 | 66 个区块，约 13 分钟 |

## 陷阱：“每个网络都一样”

“我等一个确认就够了”是一条危险的规矩。不同网络里，一个确认代表的可靠程度天差地别。要按网络、按金额来决定：

- 订单越贵，等得越久；
- 大额付款，如果网络有最终性，就等到最终性；
- 不要凭刚出区块里的一笔转账，就交出收不回来的东西。

**想一想**




1. 为什么账本里不能按“一个程序一票”来投票？
2. Bitcoin 每 10 分钟出一个区块。等 6 个确认平均要多久？3 个呢？
3. 一位顾客买一杯外带咖啡，另一位买 200 公斤咖啡豆。阿尔乔姆该对哪一位等到最终性？

*答案：1) 开程序不花什么代价，想开多少开多少。2) 约一小时和约半小时。3) 买 200 公斤的那位。*

**本章要点**

- 共识是一套规则，让没有头儿的参与者选定同一条链。
- 投票要付出代价：要么是工作量（挖矿），要么是押金（质押）。
- 新区块可能被替换；成为定局要晚一些，而且每个网络各不相同。

**接下来**：第9章。币、代币和智能合约：为什么 USDT 和 ETH 不是一回事。

# 第9章 币、代币和智能合约

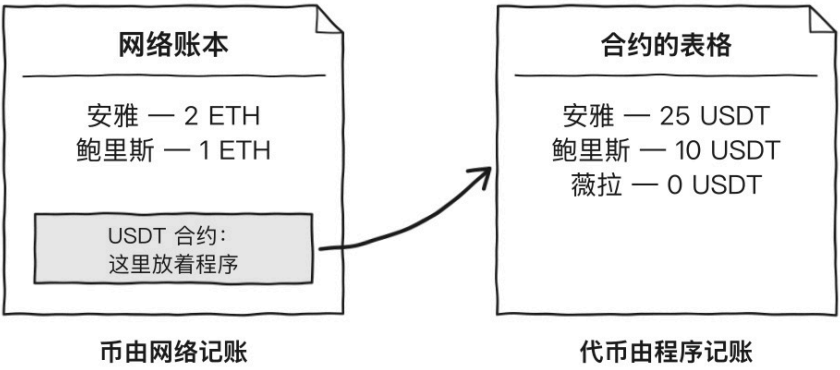
## “我付了，你那边却是空的”

顾客在 Ethereum 网络上转来 25 USDT。阿尔乔姆像第1章那样问节点，余额却没变。钱丢了？没有。只是问法不对。

## 弄明白：币和代币

**原生币**是由网络自己记账的币：Ethereum 里的 ETH、TRON 里的 TRX、BNB Smart Chain 里的 BNB。手续费就用它付。

**代币**则是由住在网络内部的某个程序来记账的。USDT 就是代币。




| | 原生币 | 代币 |
|---|---|---|
| 谁来记账 | 网络本身 | 智能合约 |
| Ethereum 里的例子 | ETH | USDT、USDC |
| 手续费用什么付 | 就用它自己 | 网络的原生币 |
| 怎么查余额 | `eth_getBalance` | 对合约做 `eth_call` |

不同网络里代币的构造方式不一样。这里看的是 Ethereum 以及和它类似的网络。

## 智能合约：没有售货员的售货机

**智能合约**是写进区块链的程序。它像一台自动售货机：投币、按键，按事先定好的规则拿到结果。每个节点执行它的方式都完全一样。




- 它不是签字盖章的合同，而就是程序。
- 通常无法修改，但有些作者给自己留了这种权限。
- 程序可能有一个拥有特殊权力的“主人”。
- 程序里的错误可能让人损失钱财，而且没人能回滚。

### 动手试试：查代币余额

方法 `eth_call` 的意思是“向合约问一下，什么都不改”。这样问网络不收手续费：账本里什么都没写。我们来问 USDT 合约：它自己的地址上放着多少 USDT：

```
curl https://public-ethereum-mainnet.rpcnode.dev \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,"method":"eth_call","params":[
  {"to":"0xdac17f958d2ee523a2206206994597c13d831ec7",
  "data":"0x70a08231000000000000000000000000dac17f958d2ee523a2206206994597c13d831ec7"},
  "latest"]}'
```

`to` 表示问哪个合约。`data` 则由三段拼成：

| 片段 | 意思 |
|---|---|
| `0x70a08231` | 问什么：“余额是多少？”（`balanceOf`） |
| 24 个零 | 补齐到规定长度 |
| 40 个字符 | 查谁的余额：去掉 `0x` 的地址 |

写这一章时，节点是这样回答的：

```
{"jsonrpc":"2.0","id":1,"result":"0x0000000000000000000000000000000000000000000000000000000014d0a9aa6c5"}
```

一串零后面是 `14d0a9aa6c5`，即 1 430 402 016 965。Ethereum 上的 USDT 有六位小数，所以合约地址上放着 1 430 402.016965 USDT。

这些钱哪来的？看样子主要是“转错地方”的钱：有人把合约地址当成收款人地址粘了进去。发送方没有通常的办法能把它们拿回来。

## 小数点点在哪里

合约只保存整数。要分出几位小数，由一个单独的属性说明——`decimals`。

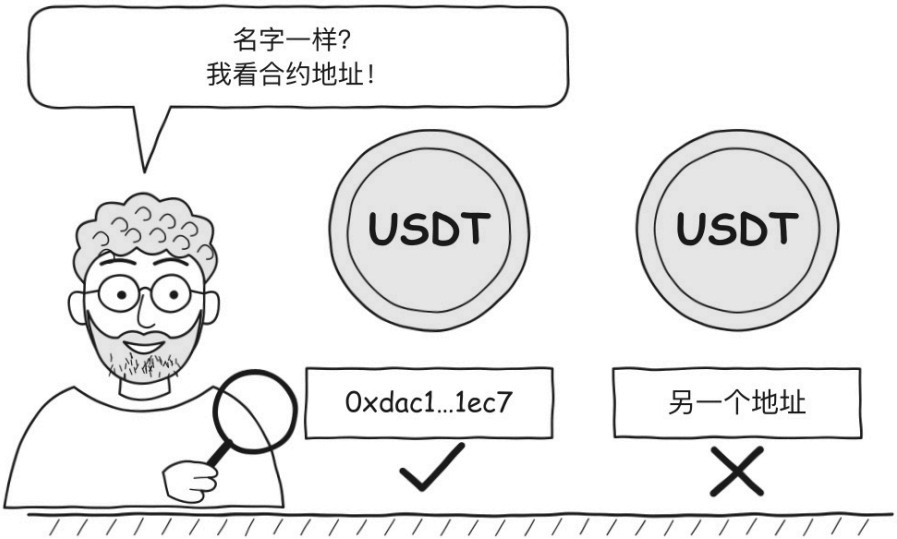
| 是什么 | 小数位数 |
|---|---|
| ETH，原生币 | 18 |
| Ethereum 上的 USDT | 6 |
| BNB Smart Chain 上的 “Tether USD”（USDT）代币 | 18 |

名字一样，位数却是三倍。一旦弄混，就差了一万亿倍。何况 BNB Smart Chain 上这个代币的地址，并不在 Tether 公司网站列出的 USDT 合约之中：名字相同，并不代表是同一家发行的。

*已于 2026 年 11 月 27 日通过查询公共节点和 tether.to/en/supported-protocols 页面核实。*

## 陷阱：假代币

谁都可以把自己的代币取名叫 “USDT”。名字什么也证明不了。真正的 USDT 要靠一对信息来认：**网络和合约地址**。



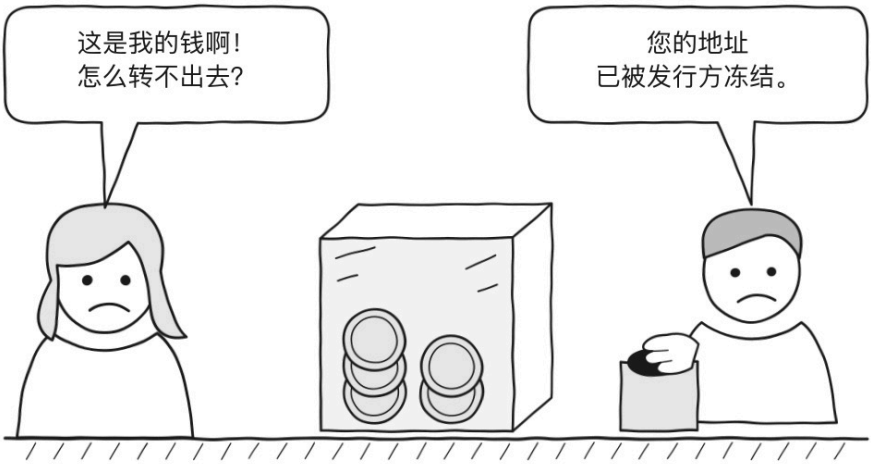

所以店铺要一次查全：

- 网络；
- 代币的合约地址；
- 收款人；
- 按正确小数位数算出的金额；
- 是否成功，确认数是否够。

### 稳定币：方便，但有主人

**稳定币**尽力把价格维持在一美元左右。就 USDT 而言，这是发行公司的承诺，不是自然规律。

发行方还握有权力：Tether 的使用条款里明文写着有权冻结代币（2026 年 2 月 26 日版本，tether.to/en/legal）。



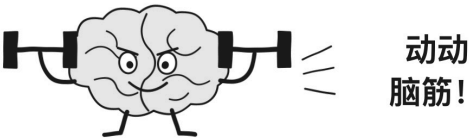

**币看得见，却动不了**

### 第二个陷阱：“请授权花费”

网站可能要你签一个授权（`approve`）：“允许这个合约花我的代币”。如果给骗子授权而不限额度，他以后就能把代币拿走，不需要你再签名。他根本不必知道你的助记词。

> 规则：看不懂要签的是什么，就别签。

**想一想**




1. 查 Ethereum 上的 USDT，节点返回数字 25 000 000。这是多少 USDT?
2. 顾客转来一个名叫 USDT、合约地址却不同的代币。订单算付款了吗?
3. 阿尔乔姆在 Ethereum 上有 100 USDT、0 ETH。他能把这些钱转给供货商吗?

*答案：1) 25。2) 不算。3) 不能：没东西付手续费，见第7章。*

**本章要点**

- 币由网络记账，代币由智能合约记账。
- 认代币要看网络和合约地址，而不是名字。
- 读金额时要算上小数位数。

**第二部分到此结束**。阿尔乔姆知道了什么是区块、密钥、手续费和代币。可节点对他来说仍然是个黑箱。**接下来**：第三部分，第10章。节点有哪些种类。

# 第10章 节点有哪些种类

## “那时候到底有多少？”

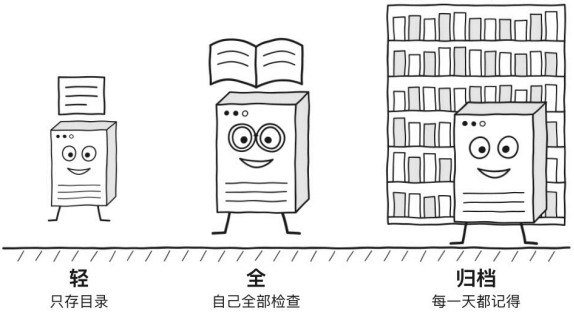
阿尔乔姆在对账，想知道几个月前店铺地址上有多少钱。节点回答今天的事对答如流，问到过去却拒绝了。区块链不是什么都存着吗?

## 弄明白：记录和余额

这里藏着两样不同的东西：

- **历史**——记录本身：区块和转账。
- **状态**——某个特定时刻谁拥有什么：余额和合约数据。

就像在银行：流水清单是有的，但去年每一分钟的余额并没有单独存着。要知道得重新算。节点之间的区别，就在于这些东西记住了多少。

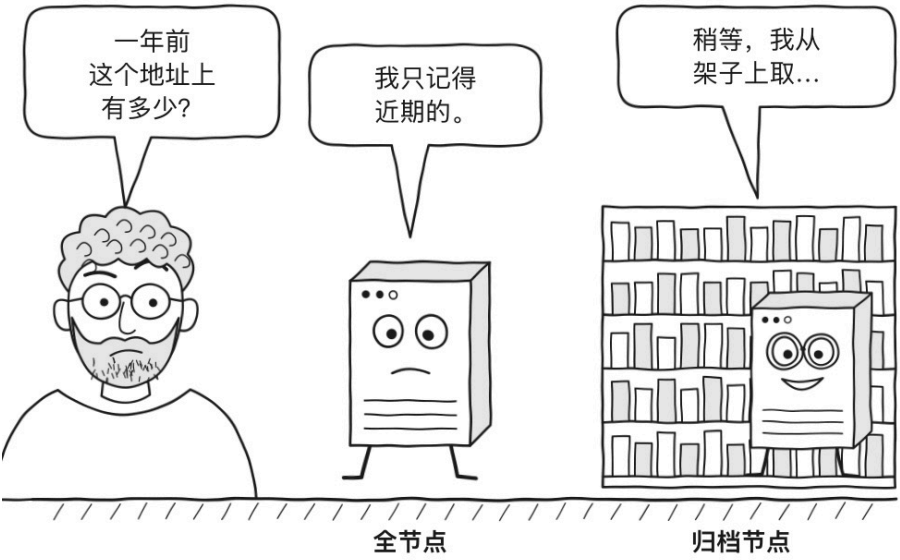



## 三个主要种类

**全节点**按网络规则检查区块，并保存近期状态。在 Ethereum 上通常是最近大约 128 个区块，也就是约 25 分钟。具体深度取决于程序和配置。

**归档节点**从头检查一切，下载过的东西不删。它记得任意区块时的状态，代价是需要大得多的硬盘。

**轻节点**只下载区块头，细节向别人打听。想法很好，但截至 2026 年 11 月 27 日，Ethereum 上还没有可以日常使用的轻节点。




### 裁剪节点：查完就扔

Bitcoin 有一种省硬盘的模式。**裁剪节点**（pruned）什么都检查，之后把旧区块删掉。它仍然是名副其实的全节点，只是删掉的东西没法再提供给别人。据 Bitcoin Core 开发者的说法，10 GB 就够它用了。

Ethereum 也有了类似的功能：从 2025 年 7 月起，节点程序可以不保存最老那批区块的内容——也就是 2022 年 9 月改用质押之前产生的区块。这是一个配置项，并非处处开启。所以就连“全”节点，也可能记不得很老的记录。

### 动手试试：问问过去

我们把同一个地址的余额查两次。第一次查现在：

```
curl https://public-ethereum-mainnet.rpcnode.dev \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,"method":"eth_getBalance",
  "params":["0xdac17f958d2ee523a2206206994597c13d831ec7","latest"]}'
```

```
{"jsonrpc":"2.0","id":1,"result":"0x2a"}
```

`0x2a` 就是 42 wei，少得可怜。现在把 `"latest"` 换成 `"0xe4e1c0"`，也就是编号 15 000 000 的区块。公共节点会这样回答：

```
{"jsonrpc":"2.0","id":1,"error":{"code":-32601,
 "message":"archive data is not available on the public endpoint,
 use a personal endpoint"}}
```

翻成中文是：“公共端点不提供归档数据，请使用个人端点”。这不是故障，而是免费服务的边界。在 RpcNode，付费套餐有归档访问，Free 套餐没有（见第3章）。

## 它们有多大

| 节点 | 硬盘占用，大约 |
|---|---|
| Bitcoin 裁剪节点 | 10 GB 起 |
| Bitcoin 全节点 | 约 900 GB |
| Ethereum 全节点 | 约 2 TB 或更多 |
| Ethereum 归档节点 | 2 到 14 TB 甚至更多，看用哪个程序 |

*Ethereum 的数据来自 ethereum.org 关于运行节点的页面，截至 2026 年 11 月 27 日。*

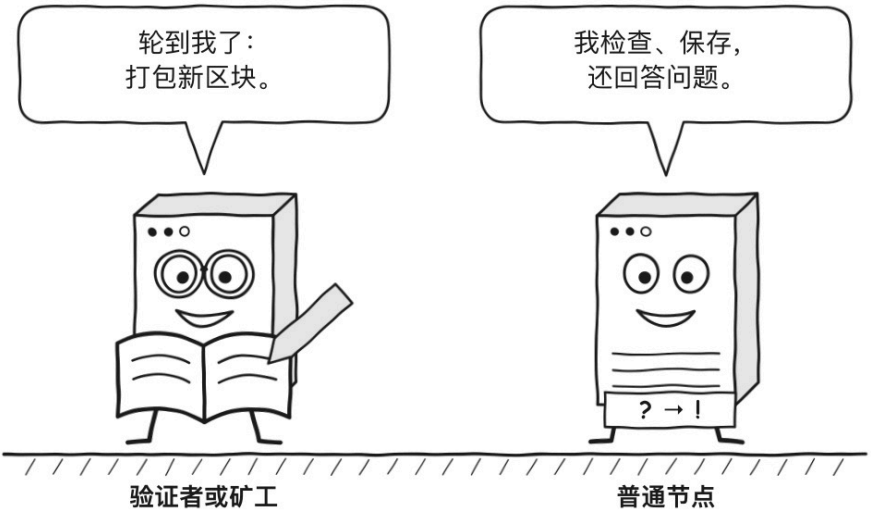
第二行可以自己验证：Bitcoin 节点会老实报出自己的大小。

```
curl https://public-bitcoin-mainnet.rpcnode.dev \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,"method":"getblockchaininfo","params":[]}'
```

在长长的回答里找两行。写这一章时，里面是：`"size_on_disk":896027659764`（连同辅助文件约 896 GB）和 `"pruned":false`——节点没有裁剪。

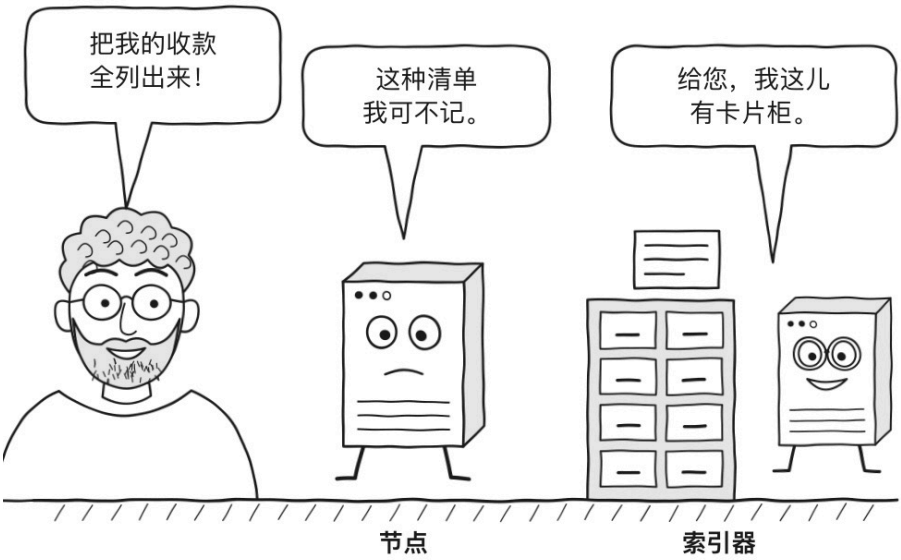
## 那区块由谁来写?

这是另一个角色。验证者或矿工参与生成区块（见第8章）。要核对付款，用不着当它们：一个能读取、检查、回答的节点就够了。




### 陷阱：“让节点列出我所有的收款”

Ethereum 节点的标准问题清单里，没有“列出这个地址的所有转账”这一项。归档节点也做不到。方法 `eth_getTransactionCount` 只会告诉你这个地址发出过多少笔交易。

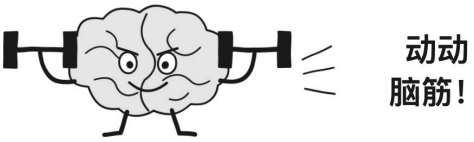


要得到这种清单，需要**索引器**：一个读取区块、把记录分门别类归进卡片柜的程序。区块浏览器网站就是这么工作的。对店铺来说，更省事的办法是一开始就记下自己的收款，并盯着新的收款，这个第26章讲。

## 你的任务需要哪一种

| 任务 | 适合用什么 |
|---|---|
| 核对刚到的付款 | 普通全节点 |
| 查过去某天的余额 | 归档节点 |
| 拿到某个地址的全部转账清单 | 索引器 |
| 参与生成区块 | 验证者或矿工 |

### 想一想




1. 阿尔乔姆想画一张店铺一年的余额曲线图，每天一个点。需要什么节点？
2. Bitcoin 裁剪节点删掉了旧区块。还能信任它来检查新区块吗？
3. 要给顾客展示他过去所有订单及付款情况，需要什么：归档节点，还是店铺自己的数据库？

*答案：1) 归档节点。2) 能，删之前它已经全部检查过。3) 自己的数据库：节点不维护这种清单。*

### 本章要点

- 历史是记录，状态是某一时刻的余额。
- 全节点记得近期状态，归档节点记得任何时候的状态。
- 节点给不出某个地址的转账清单，这需要索引器。

**接下来**：第11章。我们已经向节点问了十来个问题。该弄清楚用的是什么语言了：JSON-RPC、REST、gRPC 和 WebSocket。

# 第11章 怎样和节点对话

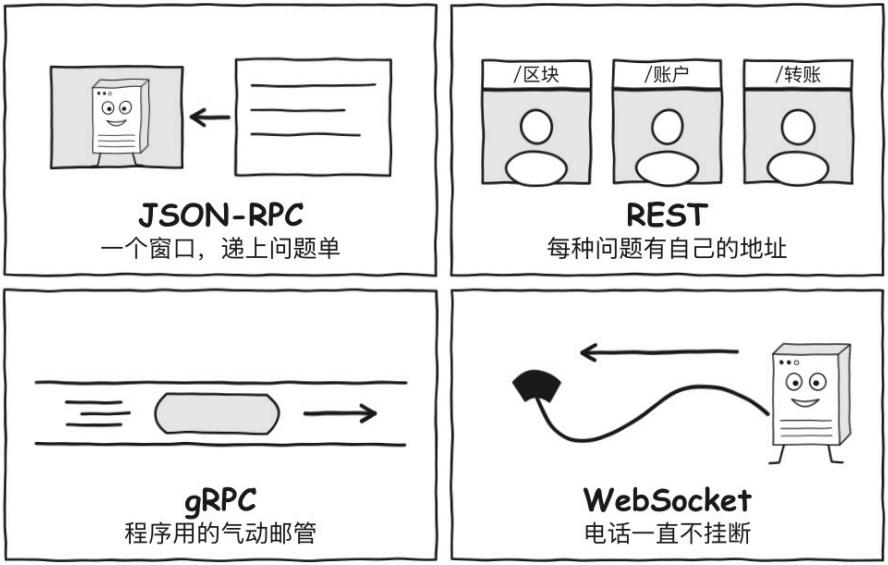
## 请求里的那些字母是什么?

阿尔乔姆已经从书里复制过十来次命令，每次都管用。该弄明白里面写的是什么了：RPC 是什么，为什么有那么多括号，为什么不同网络文档里的请求长得不一样。

## 弄明白：什么是 API 和 RPC

API 是一个窗口，一个程序通过它按事先约定的规则找另一个程序办事。RPC（远程过程调用）是让另一台电脑上的程序干活的方式，就好像它在身边一样。节点接收的正是这类请求。

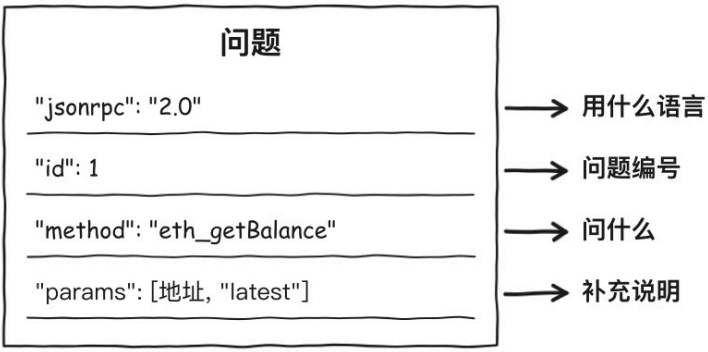
对话方式有几种：




| 方式 | 长什么样 | 在哪儿会遇到 |
|---|---|---|
| JSON-RPC | 地址只有一个，请求里写明方法和参数 | Ethereum 及类似网络、Solana、Bitcoin |
| REST | 每种问题在地址里有自己的路径 | TRON、TON、Aptos |
| gRPC | 按严格格式编排的紧凑二进制消息 | 程序之间交换数据，例如 Sui |
| WebSocket | 一条不关闭的连接 | 需要节点主动通报新消息时 |

WebSocket 不是一种语言，而是一条**通信线路**：在 Ethereum 里，线上跑的还是同样的 JSON-RPC 消息。有些网络同时提供好几种方式。

## 拆开看 JSON-RPC




JSON 是一种简单的数据写法：花括号、带引号的名称和对应的值。下面把第3章的命令逐行拆开：

| 命令片段 | 意思 |
|---|---|
| `curl` | 从终端发送请求的程序 |
| `https://…` | 端点地址 |
| `-H '…'` | 标注“里面是 JSON” |
| `-d '{…}'` | 问题本身 |

回答里要看 `result`（得到了什么）和 `error`（什么没办成）。有些节点只返回这两个字段中的一个，有些两个都返回，这时空着的那个是 `null`。回答里的 `id` 和提问时相同：靠它来对号。

## 一次问好几个问题

可以把多个问题放进方括号里的一个列表。这叫**批量请求**（batch）：

```
curl https://public-ethereum-mainnet.rpcnode.dev \
  -H 'Content-Type: application/json' \
  -d '[{"jsonrpc":"2.0","id":1,"method":"eth_blockNumber","params":[]},
       {"jsonrpc":"2.0","id":2,"method":"eth_chainId","params":[]}]'
```

```
[{"jsonrpc":"2.0","id":1,"result":"0x1946420"},
 {"jsonrpc":"2.0","id":2,"result":"0x1"}]
```

第一个回答是区块号 26 502 176，第二个是网络编号：Ethereum 的编号是 1。按 JSON-RPC 的规则，批量里的回答可以是任意顺序，所以要看 `id`。还要记住：批量省的是往返次数，不省限量。在 RpcNode，十个问题的批量按十个请求计算，并且整批一起接受或一起拒绝。

### 再看 REST

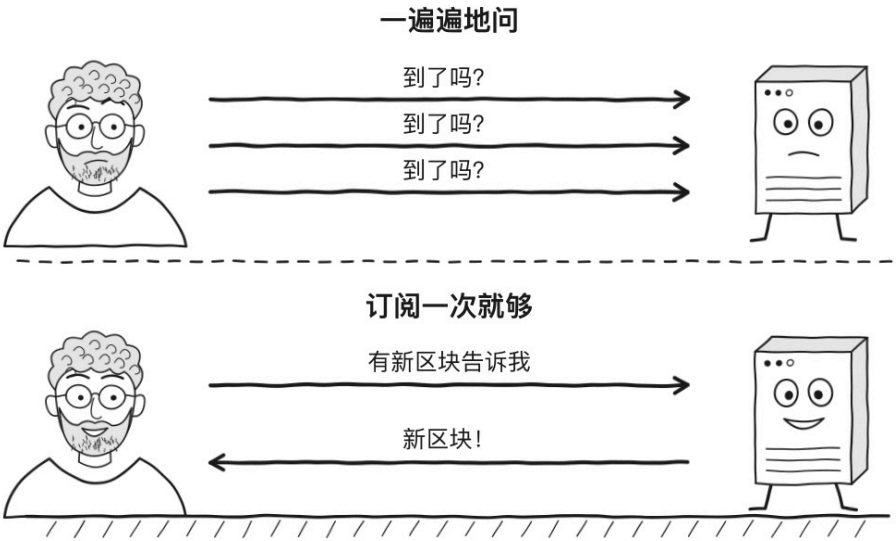
这里的问题直接写在地址里。我们向 TRON 网络要最新区块：

```
curl https://public-tron-mainnet.rpcnode.dev/wallet/getnowblock
```

会返回一大段关于区块及其转账的内容。写这一章时，那是编号 88 351 889 的区块，里面有 194 笔交易。换一个问题，就换地址后面的路径。

### WebSocket：不追问，只倾听

每秒问一次“到了没有？”太浪费。在一条保持打开的线路上，订阅一次就行，节点会自己把消息推过来。




RpcNode 的公共节点不支持 WebSocket，需要个人端点。它的地址以 `wss://` 开头。下面是一段 JavaScript 草稿：

```
const ws = new WebSocket('wss://你的端点')

ws.onopen = () => ws.send(JSON.stringify({
  jsonrpc: '2.0', id: 1,
  method: 'eth_subscribe', params: ['newHeads'],
}))

ws.onmessage = (event) => console.log(event.data)
ws.onclose = () => console.log('连接断开了')
```

线路是会断的。程序得会重新连接、重新订阅，并检查漏掉了什么。订阅同样消耗额度。

### gRPC

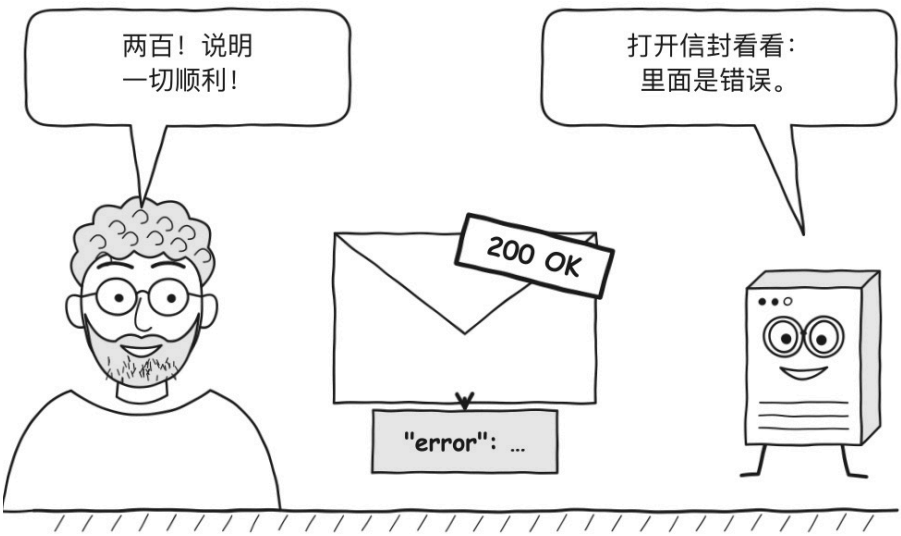
这是给看重速度和严格格式的程序用的方式。消息没法用肉眼阅读，字段事先在单独的文件里定义好。刚起步的网店通常用不着。在 RpcNode，并非所有网络都提供 gRPC：看看端点的传输方式列表。

## 陷阱：“返回 200 就一切正常”

200 是投递状态码：信封送到了。里面装的却可能是个错误。我们来查地址 “hello” 的余额：

```
curl https://public-ethereum-mainnet.rpcnode.dev \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,"method":"eth_getBalance","params":["hello","latest"]}'
```

响应状态码是 200，里面却是编号 `-32602` 的 `error`：参数无效。



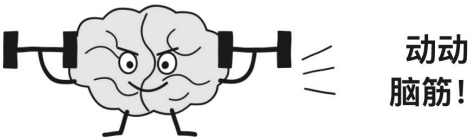
所以程序永远要同时检查状态码和 `error` 字段。

### 常见错误

| 代码 | 发生了什么 | 怎么办 |
|---|---|---|
| `-32700` | 请求内容无法解析 | 检查括号和引号 |
| `-32600` | 请求不符合规则 | 检查各个字段 |
| `-32601` | 方法不存在或不可用 | 检查名称和套餐 |
| `-32602` | 参数无效 | 检查参数值 |
| `-32005` (HTTP 429) | RpcNode：请求太频繁 | 等一等，问得稀一些 |
| `-32006` (HTTP 402) | RpcNode：额度不足 | 检查用量和套餐 |

*前四个代码来自 JSON-RPC 2.0 标准，后两个是 RpcNode 的代码。*

### 想一想




1. 把批量里第一个请求的 `eth_blockNumber` 写成 `eth_blockNumbr`。会收到什么代码？第二个正确的问题会得到回答吗？
2. 一块要自动刷新的“最新区块”显示屏，用哪个更好：每秒轮询还是订阅？
3. 程序收到 HTTP 200，马上就把订单标成已付款。哪里不对？

*答案：1) `-32601`，而且整批只回这一个：RpcNode 会拒绝整个批量，第二个问题同样得不到回答。2) 订阅。3) 既没检查 `error` 字段，也没检查转账本身（见第7章）。*

### 本章要点

- JSON-RPC 在请求里指明方法，REST 在地址里指明。
- WebSocket 让线路保持打开，好让节点主动报消息。
- 成功不等于状态码 200，而是回答里没有 `error`，并且内容符合预期。

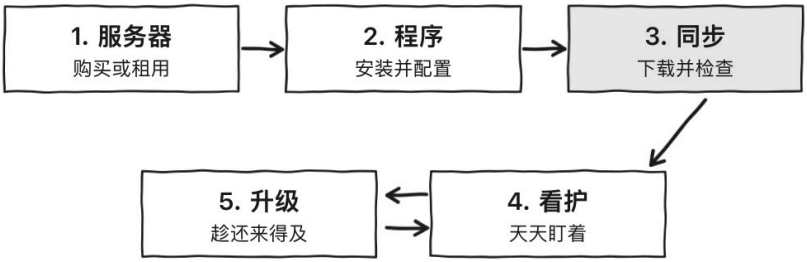
接下来：第12章。到窗口的另一边看看：自建节点怎么过日子，什么是同步，它为什么会坏。

# 第12章 自建节点内幕：安装、同步和故障

## “一个晚上就搞定了！”

一位程序员朋友劝阿尔乔姆：“下载，运行，你就有自己的节点了。”启动确实花不了多久。花时间的是之后的事。

## 弄明白：节点的一生

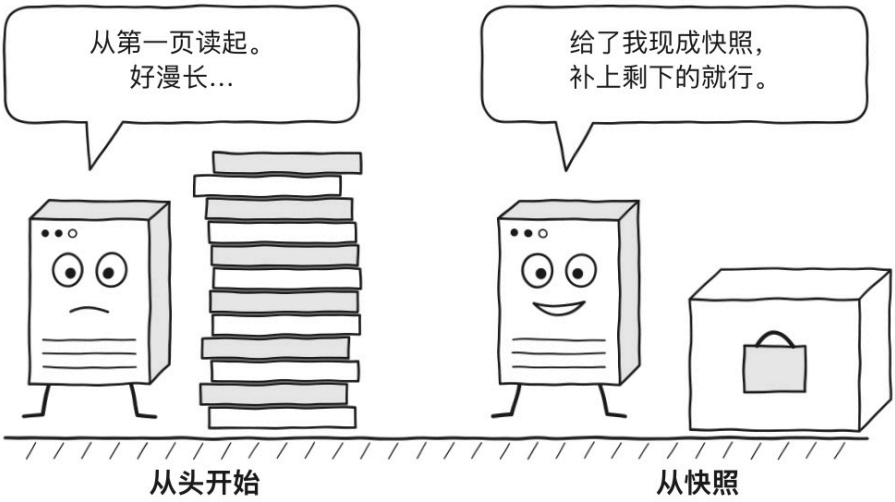



节点活着一天，第 4、5 步就重复一天

1. **服务器**。配置要求在第2章看过了。
2. **程序**。每个网络有自己的程序，往往还有好几种可选。Ethereum 需要同时跑两个：**执行客户端**（执行转账和合约）和**共识客户端**（盯着哪条链是对的）。
3. **同步**。节点追赶网络。
4. **看护**。有没有掉队，空间够不够。
5. **升级**。网络要改规则，程序得提前换。

## 同步：追上网络

新节点一无所知。在追上其他节点之前，向它查余额没有用：它只会拿过去的情况回答你。




- **从头开始**。节点亲自检查每一个区块。可靠，但很慢。Bitcoin 首次下载约 740 GB。
- **从快照开始**。拿一份现成的数据库副本，再补上剩下的部分。快一些，但你得信任做快照的人。
- **折中办法**。Ethereum 的程序通常下载近期状态，再用区块头核对。

耗时从几个小时到许多天不等：取决于网络、方式、硬盘和带宽。

### 动手试试：两行配置

配置文件 `bitcoin.conf` 的开头可以长这样：

```
server=1
prune=5000
```

| 行 | 意思 |
|---|---|
| `server=1` | 接收 RPC 提问；除非另行放开，只接收来自本机的 |
| `prune=5000` | 裁剪节点（见第10章）：删掉旧区块，区块数据大约保留 5 GB；辅助数据另外占空间 |

*这不是完整的安装说明。安装前请阅读你所用程序的文档。*

### 怎么知道节点追上了没有

可以直接问节点。在 Ethereum 上：

```
curl https://public-ethereum-mainnet.rpcnode.dev \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,"method":"eth_syncing","params":[]}'
```

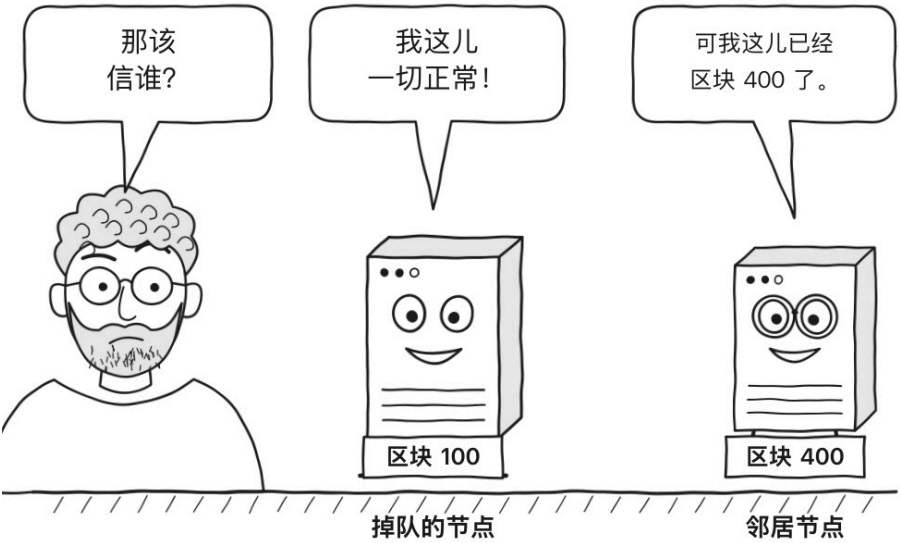
```
{"jsonrpc":"2.0","id":1,"result":false}
```

`false` 的意思是“我现在不在追赶”。节点还在追赶时，返回的则是几个编号：它现在到哪儿、要到哪儿。Bitcoin 要看第10章那个 `getblockchaininfo` 的回答：

| 字段 | 检查时的值 | 意思 |
|---|---|---|
| `blocks` | 977 715 | 已经检查了多少区块 |
| `headers` | 977 715 | 已知有多少区块 |
| `initialblockdownload` | `false` | 首次下载已经完成 |

## 陷阱：“它自己说一切正常”

节点只能根据自己知道的情况下判断。如果它和其他节点断了联系，可能根本察觉不到自己落后了。




所以要查两样：**最新区块有多新**，以及**和另一个来源对比**。新不新很容易查。要一下最新区块的时间：

```
curl -s https://public-ethereum-mainnet.rpcnode.dev \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,"method":"eth_getBlockByNumber",
  "params":["latest",false]}' | grep -o '"timestamp":"0x[0-9a-f]*"'
```

```
"timestamp":"0x6b08aacb"
```

这是从 1970 年起经过的秒数。用当前时间减去它（换上你拿到的数）：

```
echo $(( $(date +%s) - 0x6b08aacb ))
```

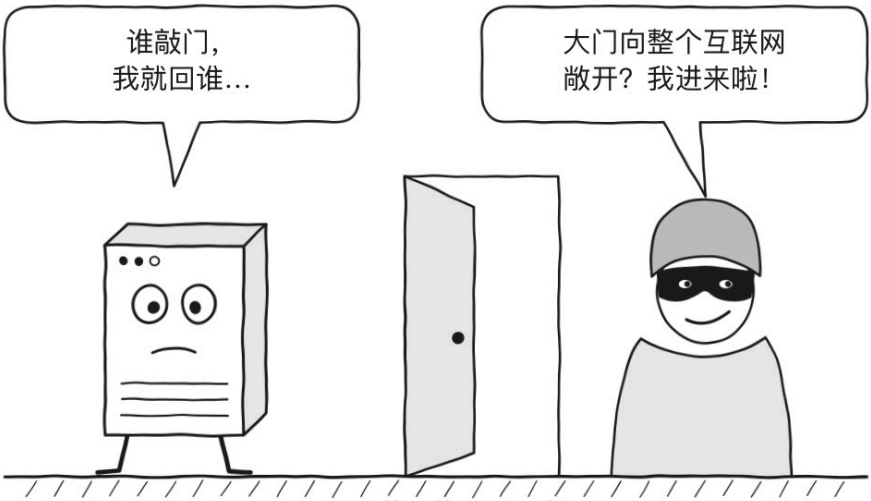
得到的是区块的“年龄”，单位是秒。写这一章时是 7 秒：对 12 秒出一个区块的 Ethereum 来说很正常。要是你的节点算出来有几千秒，那它掉队了。还记得第1章那个“节点落后一小时”的问题吗?

## 会坏在哪里

| 看到的现象 | 常见原因 | 怎么办 |
|---|---|---|
| 节点停了 | 硬盘空间用完了 | 盯着空间，提前扩容 |
| 一直落后 | 硬盘或网络太弱，相邻节点太少 | 检查硬件和网络 |
| 网络升级后“卡住” | 程序版本过旧 | 在公布的日期之前升级 |
| 断电后启动不了 | 数据库损坏 | 从备份恢复或重新同步 |
| 无缘无故变慢 | 被别人的请求压着 | 关掉外人的访问 |

## 第二个陷阱：敞开的大门

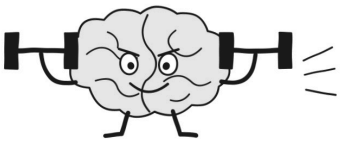
Bitcoin Core 和 Geth 默认只接收来自本机的 RPC 提问。这样设计是有道理的。把端口向整个互联网敞开，就等于谁都可以进来；如果节点上还放着钱包，你的钱就有危险。




**敞开的 RPC 端口**

- 只允许自己的地址访问（防火墙）。
- 在节点前面加一层密钥校验。
- 不要在给网站回答问题的节点上存放装着钱的钱包。

**想一想**



**动动脑筋！**

1. `eth_syncing` 回答 `false`，而最新区块已经是 40 分钟前的了。节点健康吗?
2. 硬盘 2000 GB，已用 1900 GB，每天增加 5 GB。多少天后全部停摆?
3. 网站放在另一台服务器上。能不能把 RPC 端口向所有人开放，“好让网站能提问”?

*答案：1) 不健康。2) 最多 20 天，而修理应该今天就动手。3) 不能：只放行网站的地址。*

**本章要点**

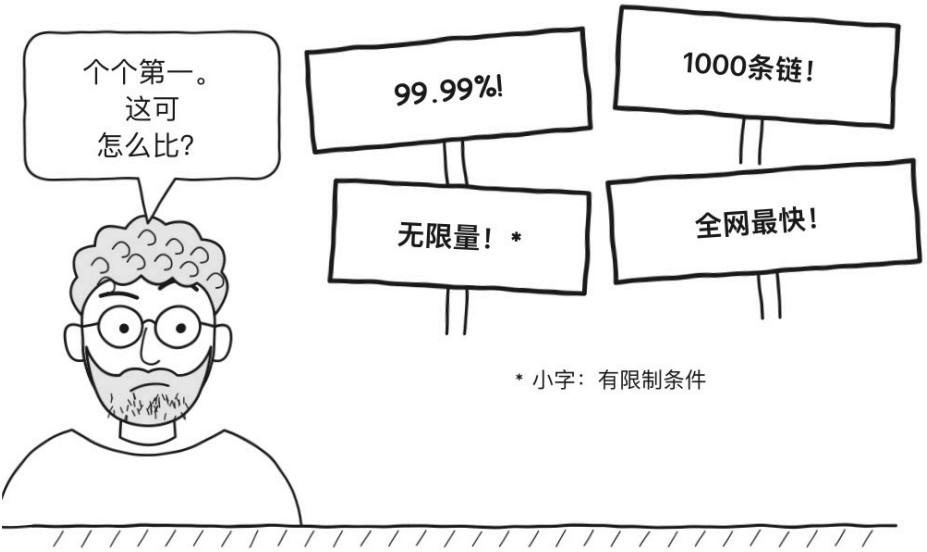
- 节点启动只是工作的开始，不是结束。
- 检查健康要看区块新不新，并和另一个来源对比。
- RPC 端口不要向整个互联网开放。

**接下来：**第13章。如果这些事不是你的爱好：怎样挑服务商，该问它些什么。

# 第13章 怎样挑选服务商

## 个个都说自己最好

阿尔乔姆想好了：自建节点不适合他。打开各家服务商的网站，满屏都是感叹号。




## 弄明白：先想清楚自己

不知道自己需要什么就去比较服务商，是白费功夫。先回答：

- 需要哪些网络？要不要测试网？
- 每秒和每月各有多少请求？
- 需要老历史（归档）吗？
- 需要第一时间知道事件吗（WebSocket、通知）？

## 算一算自己的请求量

阿尔乔姆有 20 个订单在等付款，网站对每个订单每 10 秒查一次：

> 20 个订单 × 每分钟 6 次 = 每分钟 120 个请求，即每秒 2 个。
> 30 天下来：5 184 000 个请求。

接着乘以每个方法的额度价格：不同方法价格不同。再想一想，有没有必要查得这么勤。改成 30 秒一次，请求量就只有三分之一。

## 提问清单




| 要问的 | 值得警惕的 |
| --- | --- |
| **我要的网络**有吗？这些网络里有我需要的方法吗？ | 网络列了一长串，却没有方法清单 |
| **限制**：每秒多少请求、总共多少？超了会怎样？ | 只写“无限量”不给数字；额外收费说不清 |
| **归档、WebSocket、gRPC**：哪个套餐有？哪些网络有？ | 说“应有尽有”，却没有细节 |
| **节点故障**：会切换到别的节点吗？服务器在哪里？ | 没有回答 |
| **可用性**：承诺多少、怎么测量、没做到怎么办？ | 只有百分比，不说周期和条件 |
| **防护**：能限制来源地址和网站吗？记录请求的哪些信息？ | 密钥谁拿到都能用 |
| **支持**：往哪儿写？用什么语言回复？ | 只有一个匿名表单 |
| **退出**：用的是标准方法吗？能按月付费吗？ | 强制绑定很长的期限 |

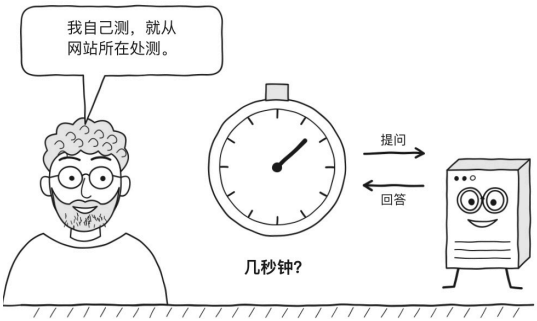
## 百分比背后藏着什么

“可用性 99%”听起来几乎等于“一直可用”。我们算算 30 天里它允许停机多久：

| 承诺 | 允许不工作 |
| --- | --- |
| 99% | 7 小时 12 分钟 |
| 99.9% | 约 43 分钟 |
| 99.99% | 约 4 分钟 |

还要问清楚到底什么算“在工作”：是服务商的网站，还是你在你那个网络上的端点。

## 动手试试：自己测速度




别人的测速是从别人那里测的。请在你网站将来所在的服务器上发五个请求：

```
for i in 1 2 3 4 5; do
  curl -s -o /dev/null -w '%{time_total} s\n' \
    https://public-ethereum-mainnet.rpcnode.dev \
    -H 'Content-Type: application/json' \
    -d '{"jsonrpc":"2.0","id":1,"method":"eth_blockNumber","params":[]}'
  sleep 1
done
```

你会看到五个数字——以秒计的响应时间。别只看最好的那个：最差的那个顾客也会碰上。这里每次都重新建立连接，保持长连接的程序通常更快。另外不光比速度，还要比区块的新鲜度（见第12章）。

## 这些问题也拿来问我们

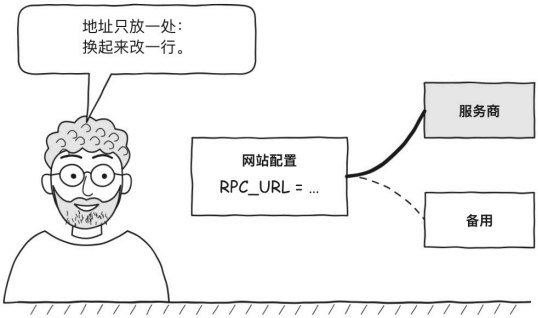
这本书讲的是 RpcNode，那就把它也查一查。数据截至 2026 年 11 月 27 日。

| 问题 | RpcNode |
| --- | --- |
| 网络 | 网站列表上有 25 个 |
| 免费试用 | Free 套餐，以及无需密钥的公共节点 |
| 限制 | Free：每秒 15 个请求、1500 万额度；Starter：40 个和 5000 万 |
| 超限时 | 不收超额费用：请求会被拒绝（429 或 402，见第11章） |
| 归档 | 付费套餐提供 |
| WebSocket 和 gRPC | 通过个人端点，并非所有网络都有 |
| 服务器位置 | 法兰克福、美国、日本 |
| 防护 | 端点可按 IP 地址和网站限制访问 |
| 隐私 | 服务商看得到你的请求（见第3章） |
| 可用性和支持 | 请查看 rpcnode.dev 网站上的现行条款 |

## 陷阱：挑数字最大的那家

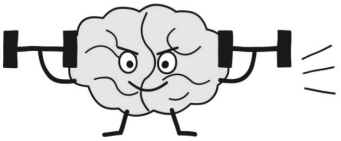
如果你要的那两个网络用起来很差，支持一千个网络也帮不了你。如果需要的方法很贵或用不了，价格最低也帮不了你。先用免费或最便宜的套餐，拿自己的真实请求跑上几天。

## 别让自己扎下根




- 端点地址放在一个配置项里，不要写死在代码中。
- 尽可能用网络的标准方法。
- 准备一个备用方案，并实际切换一次验证它能用。

### 想一想



**动动脑筋！**

1. 20 个订单，每 30 秒查一次。30 天是多少个请求?
2. 承诺 99.9%，端点一个月里停了两小时。承诺兑现了吗?
3. 端点地址写在代码的 14 个地方。这有什么不好?

*答案：1) 1 728 000。2) 没有：只允许约 43 分钟。3) 更换时很容易漏掉某一处，写在代码里的密钥也更容易泄露。*

### 本章要点

- 先算清自己的需求，再去比较。
- 用自己的测量和自己的请求来检验承诺。
- 给自己留一条退路。

**第三部分到此结束。**阿尔乔姆知道了节点是什么、怎么和它说话、找谁要节点。可各个网络并不一样啊！**接下来：**第四部分，第14章。Bitcoin 和它的亲戚们。

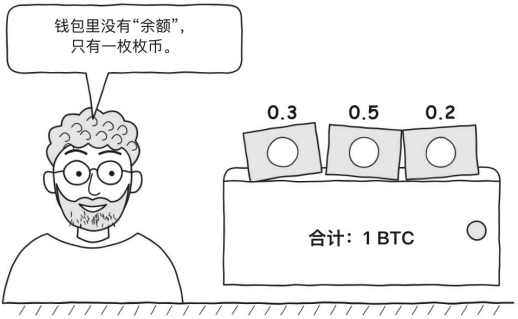
# 第14章 Bitcoin 和它的亲戚们

## “它的余额在哪儿？”

顾客想用比特币付款。阿尔乔姆已经是老手了：向节点查一下地址余额就行，跟 Ethereum 一样。可 Bitcoin 节点偏偏没有这么简单的问题。为什么？

## 弄明白：不是账户，是一枚枚钱

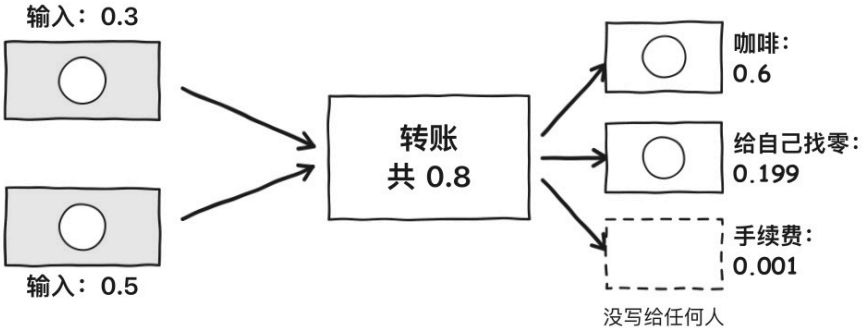
在 Ethereum 里，地址有余额，就像银行账户。Bitcoin 里没有。那里是一块块独立的“钱”，像不同面额的钞票。




每一块这样的钱叫作 UTXO（未花费的交易输出）。余额就是你所有没花掉的钱块之和。把它加起来的是钱包或索引器（见第10章）。

## 转账：输入、输出和找零

钞票不能撕下一角来用。必须整张交出去，再收找零。这里也一样：一笔转账把旧钱块整个花掉（输入），并生成新的钱块（输出）。




- 一个输出给收款人。
- 另一个是给自己的找零。
- 没有写进任何输出的部分，全归矿工。这就是手续费：输入减去输出。

> 0.8 − 0.6 − 0.199 = 0.001 BTC 手续费

手续费的单价不看金额，看交易的大小：单位是每虚拟字节多少聪。聪（satoshi）是一亿分之一个比特币。输入的钱块越多，交易越大，也就越贵。

## 人与人之间的第一笔转账

2009 年 1 月 12 日，在第 170 号区块里，一块 50 BTC 变成了两块：10 BTC 和 40 BTC。收到十个的程序员 Hal Finney 自己讲过，这是中本聪发给他试试的。40 BTC 的找零回到了发送方。没有手续费。

*区块链上没有名字：Finney 这段是从他本人的讲述中得知的（bitcointalk.org，2013 年）。*

这笔转账今天仍然查得到：

```
curl https://public-bitcoin-mainnet.rpcnode.dev \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,"method":"getrawtransaction","params":
  ["f4184fc596403b9d638783cf57adfe4c75c605f6356fbc91338530e9831e9e16",true]}'
```

在回答里找 `vin`（输入，这里有一个）和 `vout`（输出，这里有两个：`10.0` 和 `40.0`）。

## 动手试试：这枚钱花掉了吗？

方法 `gettxout` 问的是：“这块钱还在吗？”钱块用交易编号加输出序号来指认，序号从零开始。我们问问那 10 BTC：

```
curl https://public-bitcoin-mainnet.rpcnode.dev \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,"method":"gettxout","params":
  ["f4184fc596403b9d638783cf57adfe4c75c605f6356fbc91338530e9831e9e16",0]}'
```

```
{"jsonrpc":"2.0","result":null,"id":1}
```

`null`——没有这样一块未花费的钱：早就花掉了。再看创世区块之后第一个区块的奖励：

```
curl https://public-bitcoin-mainnet.rpcnode.dev \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,"method":"gettxout","params":
  ["0e3e2357e806b6cdb1f70b54c3a3a17b6714ee1f0e68bebb44a74b1efd512098",0]}'
```

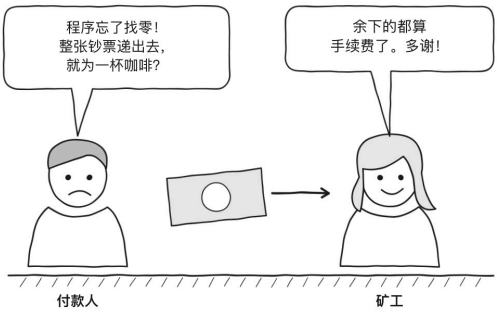
这次会返回 `"value":50.00000000` 和将近一百万个确认。写这一章时，这 50 BTC 仍然没人动过。

## 关于 Bitcoin 的另外三件事

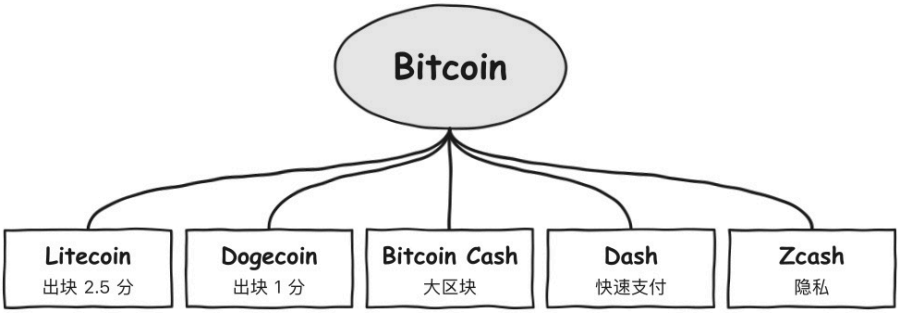
- 币的总量不会超过 2100 万枚。
- 矿工的奖励要等上面叠满 100 个区块才能花。
- 大额付款要等 6 个确认，约一小时（见第8章）。

## 陷阱：忘了找零

普通钱包会自动加上找零。但如果程序手工拼交易，漏掉了找零输出，剩下的钱就全部变成手续费。



## 亲戚们




构造上是亲戚，但各有各的网络和币

| 网络 | 出块间隔，约 | 有何不同 | 地址开头 |
|---|---|---|---|
| Bitcoin | 10 分钟 | 家里的老大 | `1`、`3`、`bc1` |
| Litecoin | 2.5 分钟 | 另一种挖矿算法 Scrypt | `L`、`M`、`ltc1` |
| Dogecoin | 1 分钟 | 发行没有上限；可以和 Litecoin 一起顺带挖 | `D` |
| Bitcoin Cash | 10 分钟 | 区块大得多 | `bitcoincash:q…` |
| Dash | 2.5 分钟 | 快速支付（InstantSend）和防止区块被替换的保护（ChainLocks） | `X` |
| Zcash | 75 秒 | 转账可以隐藏 | 公开地址 `t1`，隐蔽地址 `zs`、`u1` |

它们构造相似，所以问法也几乎一样。我们一次问五个网络的链高度：

```
for net in bitcoin litecoin doge bch dash; do
  echo -n "$net: "
  curl -s https://public-$net-mainnet.rpcnode.dev \
    -H 'Content-Type: application/json' \
    -d '{"jsonrpc":"2.0","id":1,"method":"getblockcount","params":[]}'
  echo; sleep 1
done
```

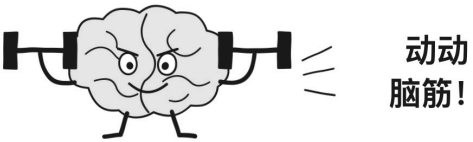
```
bitcoin: {"jsonrpc":"2.0","result":977717,"id":1}
litecoin: {"result":3219941,"error":null,"id":1}
doge: {"result":6474162,"error":null,"id":1}
bch: {"result":979125,"error":null,"id":1}
dash: {"result":2578936,"error":null,"id":1}
```

### 第二个陷阱：“有 error 就是出错了”

看看这些回答：有四个网络永远带着 `error` 字段。值为 `null` 表示“一切正常”。要检查的不是字段在不在，而是里面装着什么，正如第11章建议的那样。

也别把网络搞混。亲戚之间并不互收对方的币。Bitcoin Cash 老式地址的样子和 Bitcoin 一模一样，所以后来启用了带 `bitcoincash:` 前缀的新格式。

**想一想**




1. 钱包里有 0.3、0.5 和 0.2 BTC 三块。要付 0.4，手续费 0.001。一块够吗？找零是多少？
2. 同样发 1 BTC，用一百块零碎凑出来，为什么比用一整块贵？
3. 输入 1.0 BTC，唯一的输出是 0.4 BTC。那 0.6 去哪儿了？

*答案：1) 够，用 0.5 那块；找零 0.099。2) 输入越多，交易越大。3) 成了手续费。多半是忘了找零。*

**本章要点**

- Bitcoin 里没有账户余额，只有未花费的钱块（UTXO）。
- 手续费等于输入减输出；别忘了找零。
- 亲戚们构造相似，但是不同的网络。

**接下来：**第15章。Ethereum：账户、Gas 和事件。

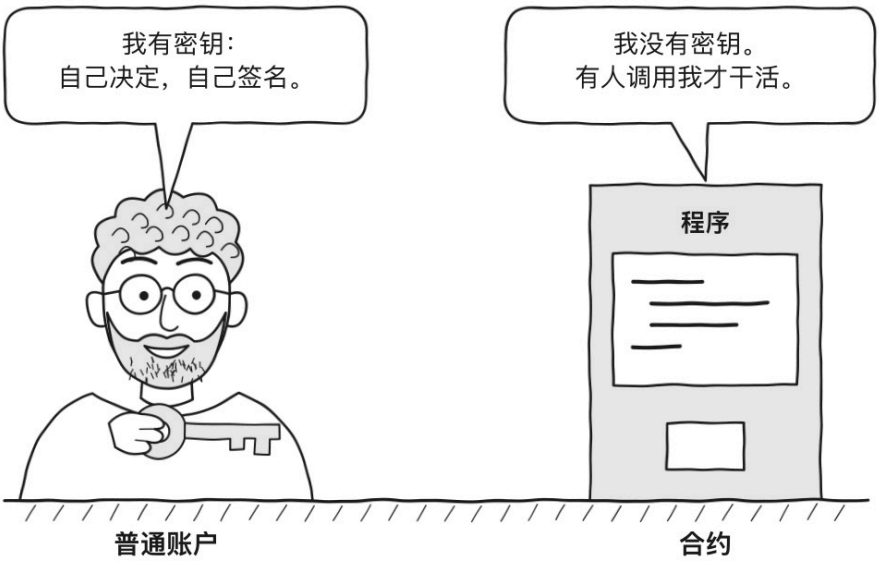
# 第15章 Ethereum：账户、Gas 和事件

## “为什么转美元更贵？”

阿尔乔姆给供货商转了一点 ETH，后来又转了 USDT。金额差不多，USDT 那笔收的“Gas”却多出一倍还不止。Gas 是什么，为什么会这样？

## 弄明白：两种账户

在 Ethereum 里，地址有余额，像银行账户一样，和 Bitcoin 的钱块（见第14章）完全不同。账户还分两种。




| | 普通账户 | 合约 |
|---|---|---|
| 谁来支配 | 密钥持有人 | 它自己的程序 |
| 能否主动发起转账 | 能 | 不能，只有被调用时才行 |
| 有没有代码 | 通常没有 | 有 |

程序由 EVM（Ethereum 虚拟机）执行。它就像装在每个节点里的一台游戏机：同一个程序，在哪儿运行结果都一样。

### 动手试试：程序放在哪里

方法 `eth_getCode` 显示某个地址上的代码。我们查 USDT 合约，只保留回答的开头：

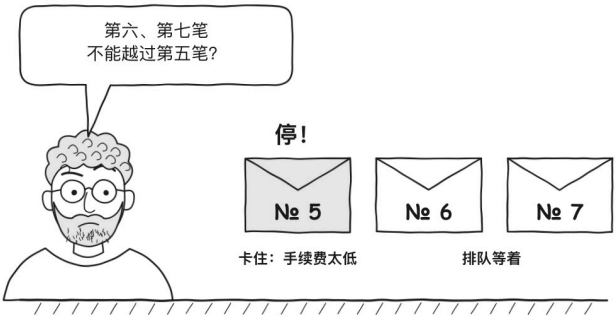
```
curl -s https://public-ethereum-mainnet.rpcnode.dev \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,"method":"eth_getCode","params":
  ["0xdac17f958d2ee523a2206206994597c13d831ec7","latest"]}' | head -c 80; echo
```

你会看到一长串字符的开头：这就是程序，超过 1.1 万字节。把地址换成 `0x000000000000000000000000000000000000dEaD`，返回的是 `"0x"`，也就是没有代码。

*从 2025 年起，普通账户也可以把自己的操作委托给某个程序（Pectra 升级）。这种情况下回答以 `0xef0100` 开头。所以“有代码”不再总是等于“这是合约”。*

### 顺序号：nonce

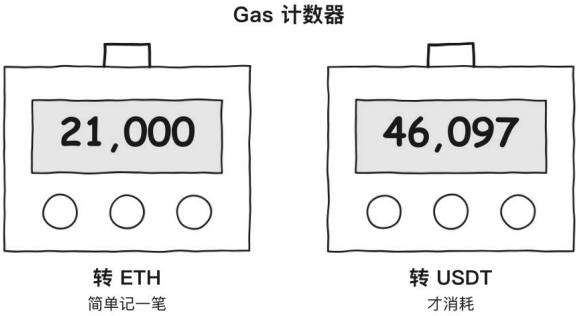
账户的每笔交易都有一个编号：0、1、2，依此类推。它叫 **nonce**。网络严格按顺序执行，同一个编号不会接受第二次。




如果一笔手续费给得少的转账卡住了，后面的都得跟着等。钱包提供“加速”功能：用同一个编号、更高的手续费再发一笔新的。

### Gas：工作量计数器

**Gas** 衡量一笔交易需要多少工作量。最简单的“转 ETH”记录，如今是 21 000。转代币要运行程序，需要的 Gas 更多。




手续费 = 消耗的 Gas ×（基础价格 + 小费）

- **基础价格**由网络自己定。这部分钱会被**销毁**。
- **小费**归验证者：给得越高，他越愿意收下你的交易。
- Gas 单价用 **gwei** 计，即十亿分之一个 ETH。

一个区块能装的 Gas 有上限。区块装得超过一半时，基础价格上涨；不到一半时下跌。例如区块 26 502 250 的容量是 6000 万 Gas，装了 93%。

钱包会设一个 **Gas 上限**。只扣实际消耗的部分。但如果上限不够，转账就执行失败，已经消耗的 Gas 不退。

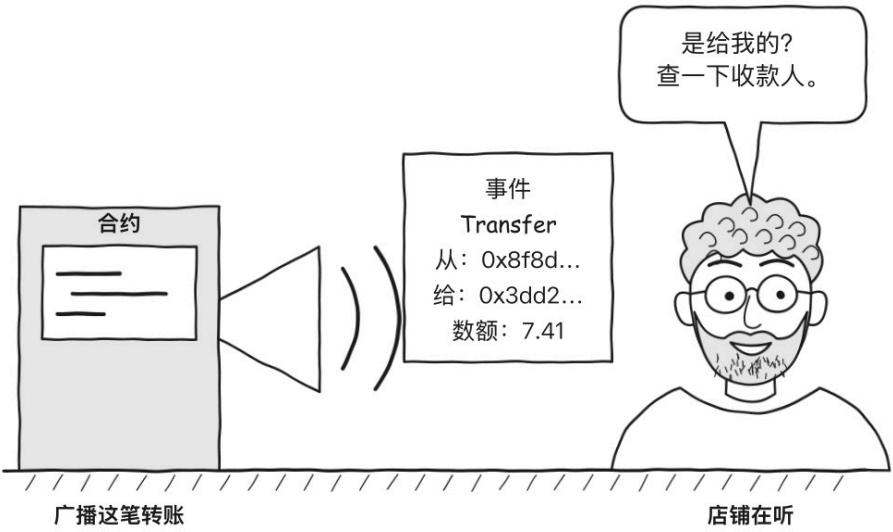
## 一笔真实的 USDT 转账

这是同一个区块里一笔 7.41 USDT 的转账。到那时，发送方已经做过 423 464 笔交易了。

| 项目 | 值 |
|---|---|
| 消耗的 Gas | 46 097 |
| 基础价格 | 0.058 gwei |
| 小费 | 2 gwei |
| 手续费 | 约 0.000095 ETH |
| 其中销毁 | 约 0.000003 ETH |

## 事件：合约怎样通报转账

代币余额放在合约内部。为了让外界知道发生了转账，合约会发出一个**事件**（log）。代币的这种事件叫 `Transfer`：谁发的、发给谁、发多少。




事件记在第7章说过的交易回执里。我们查一下这笔转账的回执：

```
curl https://public-ethereum-mainnet.rpcnode.dev \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,"method":"eth_getTransactionReceipt","params":
  ["0xc0015ff5178d72a68831b133187c3f9f1fbee2e440d3af311d0b32110e2cf016"]}'
```

在回答里找到 `logs`：

| 字段 | 开头和结尾 | 意思 |
|---|---|---|
| `address` | `0xdac1…1ec7` | 谁发出的：USDT 合约 |
| `topics[0]` | `0xddf2…b3ef` | 事件类型：Transfer |
| `topics[1]` | `…8f8d…884e` | 从谁 |
| `topics[2]` | `…3dd2…a9c0` | 给谁 |
| `data` | `…711150` | 7 410 000，即 7.41 USDT |

方法 `eth_getLogs` 则可以搜索事件。我们数一数最新区块里有多少笔 USDT 转账：

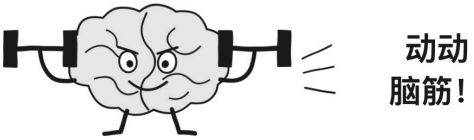
```
curl -s https://public-ethereum-mainnet.rpcnode.dev \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,"method":"eth_getLogs","params":[{
  "address":"0xdac17f958d2ee523a2206206994597c13d831ec7",
  "topics":["0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef"],
  "fromBlock":"latest","toBlock":"latest"}]}' | grep -o '"logIndex"' | wc -l
```

得到的就是一个区块里的转账笔数。过去区块的查询公共节点不会回答：那已经属于归档了（见第10章）。

### 陷阱：“我只监听事件”

- **转 ETH 本身不会产生 Transfer 事件**。只等事件的店铺会漏掉这种付款。将来的网络升级可能改变这一点。
- **ETH 可能从合约里转来**，比如某些钱包。这时在交易本身里看不到你的地址。
- **任何合约都能发出事件**。要核对 `address`：是不是真正的代币（见第9章）。

**想一想**




1. 转 ETH，消耗 21 000 Gas，单价连小费共 2 gwei。手续费是多少 gwei，多少 ETH?
2. 连续几个区块只装了 30%。基础价格会怎样?
3. 店铺只盯着 USDT 合约的 Transfer 事件。顾客用 ETH 付了款。店铺能看到吗?

*答案：1) 42 000 gwei，即 0.000042 ETH。2) 会下降。3) 看不到。*

**本章要点**

- 账户要么由密钥支配，要么由程序支配。
- 手续费是 Gas 乘以单价：基础价格加小费。
- 代币转账靠事件得知，ETH 要另外去捕捉。

**接下来：**第16章。Ethereum 亲戚很多：其他 EVM 网络和“二层”。

# 第16章 Ethereum 的亲戚和“二层”

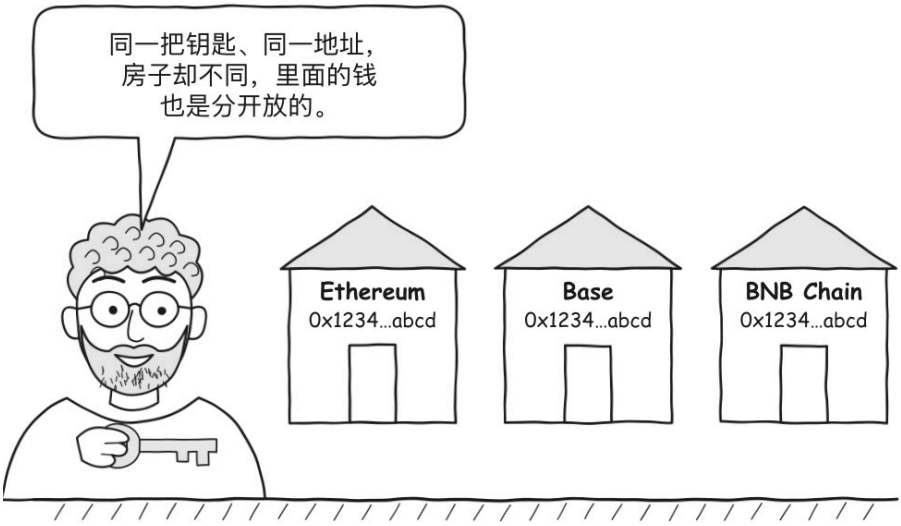
## “我发到 Base 上了”

顾客发来消息：“USDC 发到 Base 上了，地址一样。”阿尔乔姆糊涂了。这是另一种代币？另一个网络？地址又为什么一样？

### 弄明白：同一款游戏机，不同的人家

第15章的 EVM 被很多网络拿去用了。所以它们的地址格式一样，`eth_…` 方法一样，同一个合约程序可以在每个网络上运行。这类网络叫 EVM 兼容网络。

但它们的账本是各记各的。有自己的区块、自己的余额、自己的手续费。




为了不把网络弄混，每个网络都有一个编号——chain ID。它会被签进签名里，所以为一个网络签好的转账，另一个网络不会接受。

### 动手试试：向六个网络问编号

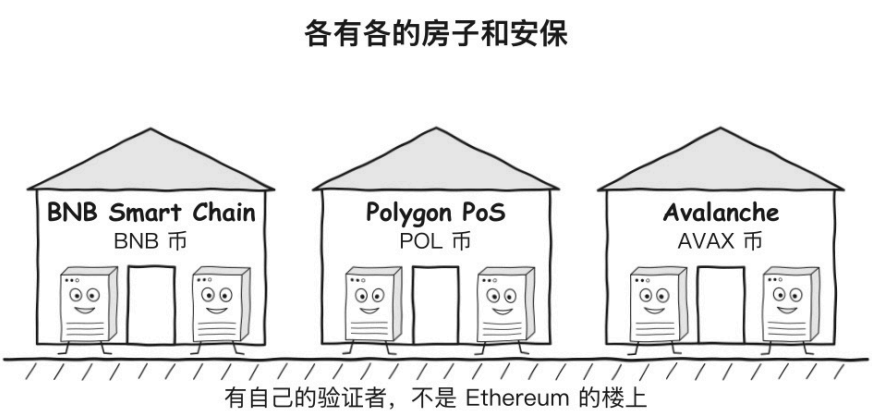
```
for net in ethereum base arb bsc mantle monad; do
  echo -n "$net: "
  curl -s https://public-$net-mainnet.rpcnode.dev \
    -H 'Content-Type: application/json' \
    -d '{"jsonrpc":"2.0","id":1,"method":"eth_chainId","params":[]}'
  echo; sleep 1
done
```

返回的是十六进制数。下面换成普通数字，同时列出检查时的出块频率：

| 网络 | Chain ID | 大约多久一个区块 | 手续费用什么付 |
|---|---|---|---|
| Ethereum | 1 | 12 秒 | ETH |
| Base | 8453 | 0.2 秒 | ETH |
| Arbitrum One | 42161 | 0.3 秒 | ETH |
| BNB Smart Chain | 56 | 0.45 秒 | BNB |
| Mantle | 5000 | 2 秒 | MNT |
| Monad | 143 | 0.3 秒 | MON |

### 第一类亲戚：自立门户的

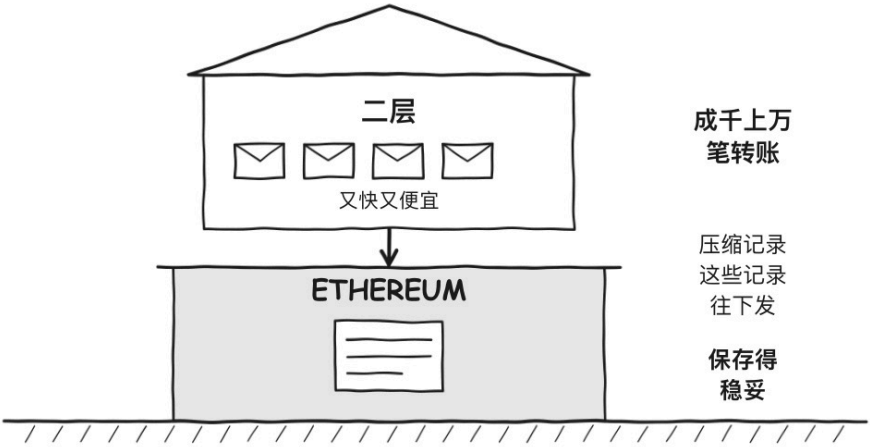
它们有自己的验证者，区块的事自己商量。




其中 Polygon PoS 还会定期在 Ethereum 上留下自己区块的记号。2025 年 11 月启动的 Monad 也属于自立门户的一类。

### 第二类亲戚：二层网络（L2）

L2 在自己这边执行转账，又快又便宜，再把压缩后的记录发到 Ethereum。最常见的一种叫 **rollup**，意思是“打包汇总”。




- 转账的先后顺序通常由一个运营方决定——**排序器**（sequencer）。所以才这么快。也正因如此，在这一环上你得信任它。
- **乐观** rollup（Arbitrum One、OP Mainnet）先认定记录是对的，但留出时间让人提出异议。
- **ZK** rollup（例如 Mantle）附带数学证明。
- Base 在 2026 年改用自己的代码，并结合了几种证明方式。

由于要留出核查时间，走官方桥把钱提回 Ethereum 快不了：

| 从哪个网络 | 提现到 Ethereum 要等多久 |
|---|---|
| Arbitrum One | 不少于 7 天 |
| OP Mainnet | 7 天 |
| Base | 5 天，个别情况 1 天 |

*据各网络文档，截至 2026 年 11 月 27 日。这些期限会变：OP Mainnet 计划在 2027 年 1 月改造。*

## “很快看到了”不等于“已成定局”

还记得第8章的 `latest`、`safe` 和 `finalized` 吗？Base 节点是这样回答的：

| 问的是什么 | 比最新区块落后 |
|---|---|
| `latest` | — |
| `safe` | 约 17 秒 |
| `finalized` | 约 19 分钟 |

区块不到一秒就出来了，可要等到关于它的记录在 Ethereum 上也成了定局，它才算定局。

## 二层的手续费

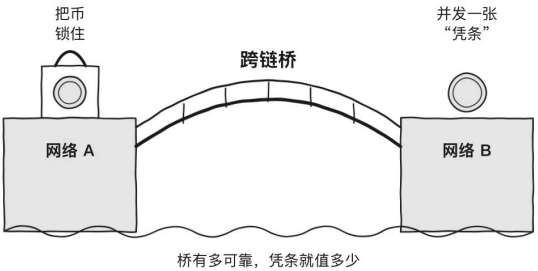
它主要由两部分组成：在 L2 上干活的费用，以及写进 Ethereum 的费用。也可能有些零头附加费。在 Base 的回执里，第二部分放在 `l1Fee` 字段：

```
curl https://public-base-mainnet.rpcnode.dev \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,"method":"eth_getTransactionReceipt","params":
  ["0xa1d4398ce62b570a2810ad2b2ed347f388f9ad85aee4b62ba3215917cf3ac7e5"]}'
```

这里在 Base 上干活收了约 0.0000035 ETH，写进 Ethereum 收了约 0.0000000006 ETH。并不总能这么便宜。

## 跨链桥

要从一个网络“搬”到另一个网络，用的是**跨链桥**。它常见的做法是在一个网络把币锁住，在另一个网络发行替代品。




所以 USDC 和 USDC.e 是两种不同的代币，合约地址也不同。前者由 Circle 公司亲自发行，后者是经由桥产生的。

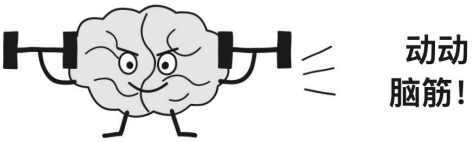
## 陷阱：发错了网络

第6章说过：这样的付款店铺看不到。这是真的。那钱丢了吗?

- **地址的密钥在你手里**——通常能把钱取出来：把钱包切换到那个网络即可。要把钱转走，需要有那个网络的原生币付手续费。
- **那是交易所的地址**——你没有密钥。如果交易所不支持那个网络，钱多半拿不回来了。
- **那是个合约钱包**——在另一个网络上它可能根本不存在。需要找行家帮忙。

> 规则：助记词不要告诉任何人，哪怕他保证能帮你把那笔钱“捞”回来（见第6章）。

### 想一想




1. 店铺等的是 Ethereum 上的 USDT，结果钱到了 BNB Smart Chain 上，地址还是自己的。密钥在阿尔乔姆手里。怎么办?
2. Base 上 0.2 秒就出了区块。大额付款能算定局了吗?
3. 顾客发来的是 USDC.e 而不是 USDC。是一回事吗?

*答案：1) 把钱包切换到 BNB Smart Chain；要转走需要 BNB。2) 不能，等 finalized。3) 不是，这是另一种代币：收不收由店铺决定。*

### 本章要点

- EVM 网络长得像，账本却各不相同：要核对 chain ID。
- 有自立门户的网络，也有架在 Ethereum 之上的二层。
- 区块出得快，还不等于已成定局。

接下来：第17章。TRON：用能量代替 Gas，稳定币最爱走的一条路。

# 第17章 TRON：用能量代替 Gas，稳定币最爱走的路

## “可以走 TRC-20 吗？”

顾客问：“收 TRC-20 的 USDT 吗？”这星期阿尔乔姆已经第三次听到这个问题了。TRC-20 是什么，为什么大家都要它？

### 弄明白：一个独立的网络

**TRON** 是一条独立的区块链，有自己的币 **TRX**。它的最小单位叫 **sun**：一个 TRX 等于一百万 sun。

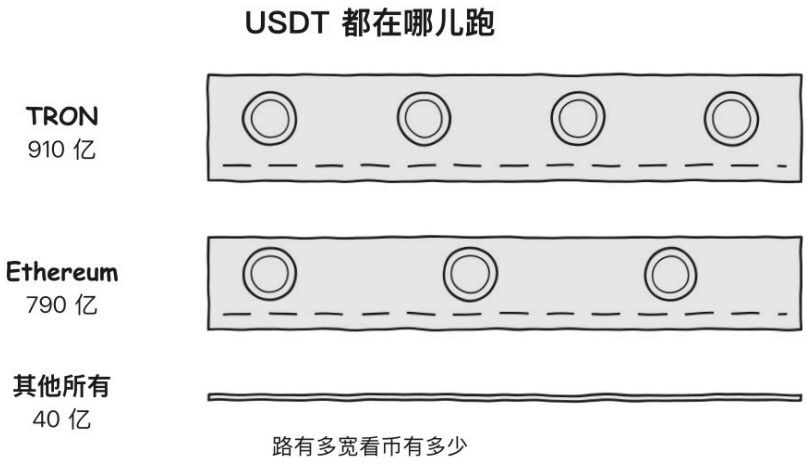
TRON 的构造在很多方面像 Ethereum：账户余额、智能合约、事件。程序由它自己的虚拟机执行，是第15章 EVM 的近亲。

**TRC-20** 是在 TRON 上制作代币所依照的一套规则，是 Ethereum 上 ERC-20 标准的兄弟。“TRC-20 的 USDT”意思就是：TRON 网络上的 USDT 代币。

区块每 3 秒出一个，由 27 位选举出来的超级代表来写（见第8章）。

### 稳定币最爱走的路

USDT 住在很多网络上，但分布极不均匀。下面是发行它的 Tether 公司自己公布的数字，访问时间是 2026 年 11 月 27 日：




全部 USDT 的一半以上在 TRON。顾客因此才这么常选这个网络。

*来源：app.tether.to/transparency.json。计算方法是已发行数量减去尚未投入流通的库存。*

### 地址以 T 开头

在内部，TRON 地址是 20 个字节，和 Ethereum 一样，前面加上前缀 `41`。给人看时则换一种写法，以字母 `T` 开头：

| 人看到的样子 | 内部的样子（USDT 合约地址） |
|---|---|
| `TR7NHqjeKQxGTCi8q8ZY4pL8otSzgjLj6t` | `41a614f803b6fd780986a42c78ec9c7f77e6ded13c` |

带 `T` 的写法里藏着校验和：打错字能被发现。我们先拿真实地址问节点，再把它最后一个字母改掉：

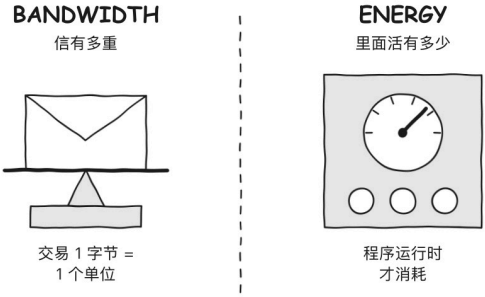
```
curl https://public-tron-mainnet.rpcnode.dev/wallet/validateaddress \
  -d '{"address":"TR7NHqjeKQxGTCi8q8ZY4pL8otSzgjLj6t","visible":true}'

curl https://public-tron-mainnet.rpcnode.dev/wallet/validateaddress \
  -d '{"address":"TR7NHqjeKQxGTCi8q8ZY4pL8otSzgjLj6u","visible":true}'
```

```
{"result":true,"message":"Base58check format"}
{"result":false,"message":"Invalid address"}
```

`"visible":true` 表示“我用带 T 的写法写地址”。

### 两种费用代替 Gas

第7章答应过要讲 TRON 怎么计算转账的“分量”。它用两把尺子来量。




- **带宽**（Bandwidth）——为占的地方付费。消耗大致等于交易的字节数。任何转账都要付。
- **能量**（Energy）——为程序的工作付费。调用合约时需要，所以转 USDT 也需要。

只要某个地址收到过一次 TRX，就能得到一份礼物：每天 600 单位带宽。用掉的部分在 24 小时内逐渐恢复。免费能量是没有的。

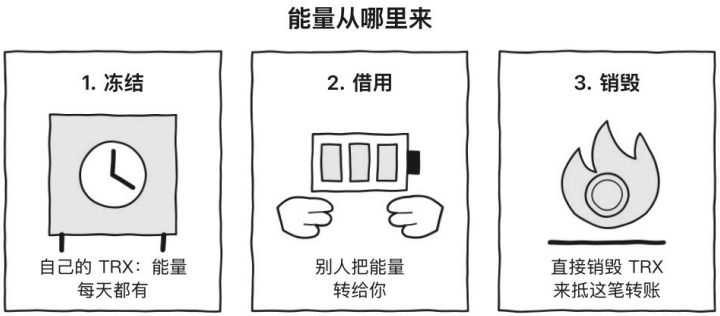
哪样不够，网络就拿 TRX 抵扣并销毁。什么价？自己问节点：

```
curl -s https://public-tron-mainnet.rpcnode.dev/wallet/getchainparameters |
  python3 -c "import json,sys
for p in json.load(sys.stdin)['chainParameter']:
    if p['key'] in ('getFreeNetLimit','getTransactionFee','getEnergyFee'):
        print(p['key'], p['value'])"
```

```
getTransactionFee 1000
getEnergyFee 100
getFreeNetLimit 600
```

也就是说，一单位带宽 1000 sun，一单位能量 100 sun。这些价格由超级代表投票调整。

## 能量从哪里来

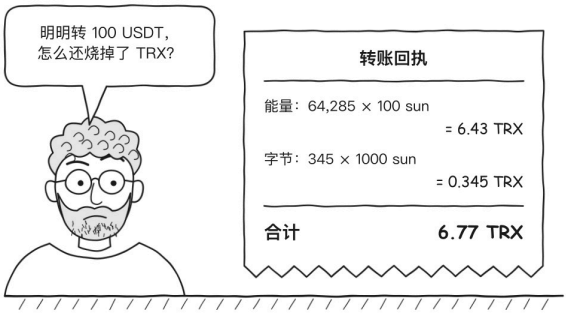



- **冻结 TRX**（英文叫 staking，质押）。币仍然是你的，地址每天分到公共能量中的一份。想取回时，解冻后要等 14 天。
- **借来用**：冻结了 TRX 的人把自己的能量转给另一个地址。
- **什么都不做**：缺的部分用销毁 TRX 来付。

要冻结多少？取决于其他所有人一共冻结了多少。检查当天，要够每天转一笔 USDT，大约需要冻结 6600 TRX。

## 一笔真实转账的回执

这是区块 88 352 467 里一笔 100 USDT 的转账。发送方既没有能量，也没有剩余的免费带宽：




回执用方法 `gettransactioninfobyid` 获取。公共节点上它是关闭的，需要个人端点，所以这里只是一段草稿：

```
curl https://你的端点/walletsolidity/gettransactioninfobyid \
  -d '{"value":"0eb7b372f5a268ce56430392de8db2fd970455bb0970c67b47a7b2673abb513c"}'
```

| 回答中的字段 | 值 | 意思 |
|---|---|---|
| `energy_usage_total` | 64 285 | 一共消耗的能量 |
| `energy_penalty_total` | 49 635 | 其中因热门而加收的部分 |
| `energy_fee` | 6 428 500 | 为能量销毁的数额，单位 sun |
| `net_fee` | 345 000 | 为字节销毁的数额，单位 sun |
| `fee` | 6 773 500 | 总计，即 6.7735 TRX |
| `result` | `SUCCESS` | 程序执行成功 |

什么是加收？当某个合约被使用得过多时，网络会让调用它变得更贵。那一天，转 USDT 消耗的能量几乎是不加收时的 4.4 倍。

还有：如果收款人此前没有持有过 USDT，需要的能量翻一倍，为 130 285。因为合约得为他新建一行记录。

## 最新区块与已固化区块

TRON 节点有两套地址。`/wallet/…` 按最新数据回答。`/walletsolidity/…` 只按已成定局的数据回答。当 27 位超级代表中有 19 位在该高度或更高处出过自己的区块后，区块就成为定局。通常约一分钟。

检查当天，最新区块是 88 352 763，已成定局的是 88 352 746。

做实验还有测试网 Nile，它有自己的公共节点：`public-tron-nile.rpcnode.dev`。

## 陷阱：“进了区块就是到账了”

看一看最新的定局区块里直接进行的 USDT 转账：

```
curl -s https://public-tron-mainnet.rpcnode.dev/walletsolidity/getnowblock > block.json
python3 <<'EOF'
import json
usdt = '41a614f803b6fd780986a42c78ec9c7f77e6ded13c'
rows = []
for t in json.load(open('block.json'))['transactions']:
    v = t['raw_data']['contract'][0]['parameter']['value']
    data = v.get('data', '')
    if v.get('contract_address') == usdt and data.startswith('a9059cbb'):
        amount = int(data[72:136], 16) / 1_000_000
        rows.append((t['txID'][:8], t['ret'][0]['contractRet'], amount))
print('USDT 转账笔数:', len(rows))
print('失败笔数:', sum(r[1] != 'SUCCESS' for r in rows))
for r in rows[:3]:
    print(*r)
EOF
```

你得到的数字会不一样。而在区块 88 352 467 里，41 笔这类转账中有两笔失败。其中一笔金额是 100 000 USDT：

```
fbfec9fa OUT_OF_ENERGY 100000.0
```

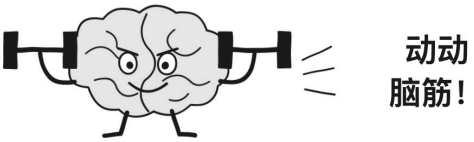
`OUT_OF_ENERGY`——能量不够，程序中止，USDT 一分都没走。可区块里确实有这条记录，截图上的金额看着也很唬人。

允许消耗多少能量由发送方设定：字段 `fee_limit`，以 sun 计的上限。如果上限不够，转账失败，已销毁的部分不退。

## 店铺要记住什么

- 等区块成为定局，并检查回执里的 `result`。
- 在回执里查找合约 `TR7NHq…Lj6t` 发出的 `Transfer` 事件。这样从别的程序发起的转账也能看到。USDT 有六位小数。
- 要从自己的地址转出 USDT，地址上需要有 TRX 或能量。每单一个地址很方便，但从一百个地址归集资金，就是一百笔要付费的转账。
- 向一个全新地址第一次转 TRX，发送方要多花至少 1 TRX。

### 想一想




1. 某地址有 600 免费带宽和大量能量。它接连发出两笔各需 345 的转账。第二笔会怎样?
2. 没有能量，收款人又从没持有过 USDT：需要 130 285 单位，每单位 100 sun。为能量要销毁多少 TRX?
3. 顾客发来哈希。交易在区块里，结果是 `OUT_OF_ENERGY`。发咖啡吗?

*答案：1) 免费带宽只剩 255，不够 345；整整 345 都要付费，销毁 0.345 TRX，不会拼凑着扣。2) 13.0285 TRX。3) 不发，转账并没有发生。*

### 本章要点

- TRON 有两把尺子：带宽按字节计，能量按程序工作量计。
- 没有能量就销毁 TRX，而且转 USDT 比单纯转币贵得多。
- 付款成立，是指真正的 USDT 在已成定局的区块里转账成功。

接下来：第18章。Solana：一切都分门别类放在账户里。

# 第18章 Solana：一切都分账户摆放

## “我的 USDC 哪去了？”

顾客用 USDC 付了款。阿尔乔姆向节点查自己钱包的余额，只看到一点点 SOL。USDC 一个都没有。被骗了？

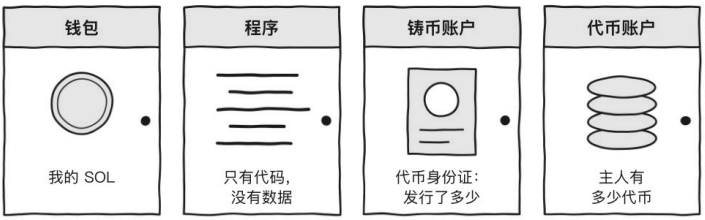
## 弄明白：由账户组成的网络

**Solana** 是一个独立的网络，它的币是 **SOL**。最小单位叫 **lamport**：一个 SOL 等于十亿 lamport。

Solana 地址是用字母和数字写出来的 32 个字节。它没有特定开头（见第6章），内部也没有校验和。

网络保存的一切都放在**账户**（account）里。这里的账户不只是放钱的，而是任何一个有地址的格子。在 Ethereum，合约把数据存在自己内部。在 Solana，**程序放一处，数据放另一处**。

**四个账户，各不相同**




英文：wallet、program、mint account、token account

每个账户都有一个主人程序（字段 `owner`）。只有它能修改这个账户的数据。

## 动手试试：代币的“身份证”

每种代币都有一个相当于身份证的账户，叫 **mint**（铸币账户）。我们查一下 USDC 的身份证：

```
curl https://public-solana-mainnet.rpcnode.dev \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,"method":"getAccountInfo","params":
  ["EPjFWdd5AufqSSqeM2qN1xzybapC8G4wEGGkZwyTDt1v",{"encoding":"jsonParsed"}]}'
```

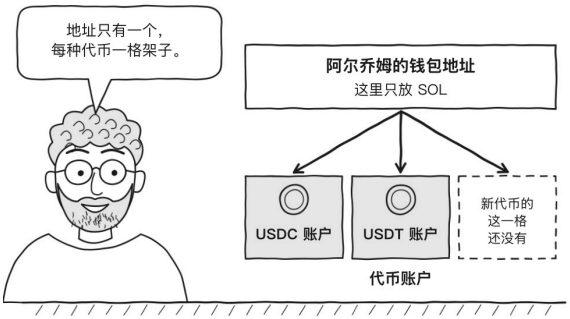
回答里的要点：

| 字段 | 值 | 意思 |
| --- | --- | --- |
| `owner` | `Tokenkeg…Q5DA` | 这个账户由代币程序管理 |
| `supply` | `8513455604386747` | 发行总量，按最小单位计 |
| `decimals` | 6 | 也就是约 85 亿 USDC |
| `mintAuthority` | `BJE5…5ruG` | 谁能增发 |
| `freezeAuthority` | `7dGb…Crar` | 谁能冻结（见第9章） |

USDC 没有自己的程序。一个公共的代币程序服务着成千上万种代币，它们靠各自的身份证来区分。这样的公共程序有两个：普通的 **Token**（USDC 和 USDT 用它）和功能更多的新版 **Token-2022**。

## 代币账户：单独的一格架子

阿尔乔姆有多少 USDC，既不记在钱包里，也不记在身份证里，而是记在**代币账户**里。每种代币各有一个。




主要的代币账户叫 **ATA**（associated token account，关联代币账户）。它的地址由三样东西算出来：钱包地址、代币的 mint 和代币程序。所以给顾客的是**钱包地址**，那一格架子由他的程序自己找到。

看一个真实的代币账户。先问上面有多少代币，再问上面有多少 SOL：

```
curl https://public-solana-mainnet.rpcnode.dev \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,"method":"getTokenAccountBalance","params":
  ["BnGihgNUu4X1JdBMNhbWnXUcpib8UQQC154isHTYwDNK"]}'

curl https://public-solana-mainnet.rpcnode.dev \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,"method":"getBalance","params":
  ["BnGihgNUu4X1JdBMNhbWnXUcpib8UQQC154isHTYwDNK"]}'
```

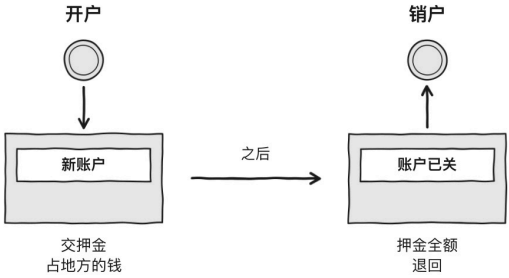
第一个回答里的 `"uiAmountString":"207637.336362"` 是 USDT。第二个里的 `"value":2039280` 是 lamport，总共才 0.00203928 SOL。方法 `getBalance` **看不见代币**。阿尔乔

姆的 USDC 就是这么“丢”的。

*顺便一提，这格架子的主人是 USDT 身份证自己的地址。看样子是有人误把 USDT 发到了“USDT 的地址”上，就像第9章那样。*

## 占地方要交押金

每个账户都占用所有节点的存储。所以账户上要压一笔 SOL **押金**（英文叫 rent，“租金”，其实并不是租金）。




不是费用而是押金：一个代币账户约 0.0015 SOL

一个代币账户大小是 165 字节。这么大要多少押金?

```
curl https://public-solana-mainnet.rpcnode.dev \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,
  "method":"getMinimumBalanceForRentExemption","params":[165]}'
```

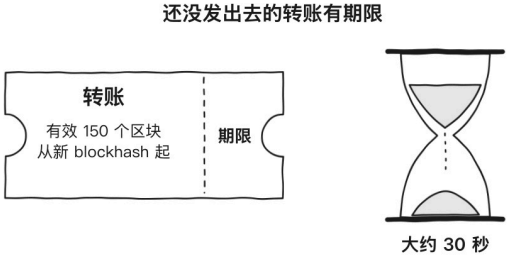
```
{"jsonrpc":"2.0","result":1488440,"id":1}
```

大约 0.0015 SOL。2026 年押金正在分阶段下调，所以程序里不要把这个数写死，要现问。如果收款人还没有某种代币的架子，通常由发送方在同一笔转账里建好，押金也由他出。

## 手续费和有效期

基础手续费是**每个签名 5000 lamport**：只有一个签名的普通转账就是 0.000005 SOL。其中一半销毁。还可以另外加一笔优先费，全部归验证者。即使转账失败，手续费照收。

每笔转账里都要放一个 **blockhash**，即某个近期区块的指纹。它同时决定了有效期。




没赶进区块，就永远进不去了

```
curl https://public-solana-mainnet.rpcnode.dev \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,"method":"getLatestBlockhash",
  "params":[{"commitment":"finalized"}]}'

curl https://public-solana-mainnet.rpcnode.dev \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,"method":"getBlockHeight",
  "params":[{"commitment":"finalized"}]}'
```

第一个回答里是 `"lastValidBlockHeight":452330099`，一秒后第二个回答给出高度 452 329 960。还剩 139 个区块。如果有效期过了，区块里仍没有这笔转账，它就再也过不去了。这时可以重新签一笔。例外是特殊的延迟转账（durable nonce），它们没有这种期限。

## 2026 年：更快了

| 变化的内容 | 原来 | 现在 |
|---|---|---|
| slot 长度，即轮到出一个区块的时段（10 月） | 0.4 秒 | 0.2 秒 |
| 最终确定时间，新投票机制 Alpenglow（10 月） | 约 13 秒 | 约 0.15 秒 |
| 交易最大尺寸，新格式 v1（9 月） | 1232 字节 | 4096 字节 |

速度可以自己验证。这个方法返回一分钟里过了多少个 slot：

```
curl https://public-solana-mainnet.rpcnode.dev \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,"method":"getRecentPerformanceSamples","params":[1]}'
```

我们拿到的是 60 秒内 `numSlots` = 277：每个 slot 约 0.22 秒。有些 slot 是空的，所以 slot 编号和区块高度是两个不同的数，别弄混。

节点回答时可以有不同的严格程度：`processed`——看到了区块，但它还可能被跳过；`finalized`——已成定局。还有一个中间词 `confirmed`：Alpenglow 之后它和 `finalized` 是一个意思。发货要以 `finalized` 为准。

### 陷阱：“交易读不出来”

由于有了新格式 v1，在 `getTransaction` 和 `getBlock` 方法里需要加上 `maxSupportedTransactionVersion`，值为 1。否则遇到新格式交易会报错。公共节点上 `getTransaction` 是关闭的，所以这是给个人端点用的草稿：

```
curl https://你的端点 \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,"method":"getTransaction","params":
  ["转账签名",{"encoding":"jsonParsed","commitment":"finalized",
  "maxSupportedTransactionVersion":1}]}'
```

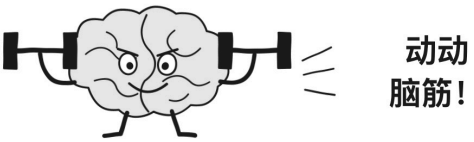
在 Solana，“转账签名”就充当交易编号，相当于别处的交易哈希。

### 店铺要核对什么

- 交易成功了：`meta.err` 等于 `null`。
- **你的钱包**名下、对应正确 mint 的代币账户增加了正确的金额：比较 `preTokenBalances` 和 `postTokenBalances`。
- 这笔钱付的正是这个订单。为此要在转账里加一个 **reference**——订单专用的一次性标记地址。
- 这个签名以前没有记过账。

收普通的 SOL 更要小心：发到错误地址的转账照样会成功。这样的钱多半没有人能帮忙追回。

**想一想**




1. 钱包签好了转账，网络两分钟后才看到它。结果会怎样？
2. 店铺为了捕捉 USDC 付款，对自己的钱包调用 `getBalance`。错在哪里？
3. 阿尔乔姆给供货商转 USDC，对方还没有这一格架子。除了手续费，还得出什么？

*答案：1）过期了，转账通不过。2）`getBalance` 只显示 SOL，USDC 在代币账户里。3）新代币账户的押金，约 0.0015 SOL。*

**本章要点**

- 在 Solana 一切都是账户；程序和它的数据分开放。
- 代币放在代币账户里：每种代币一个，还要交押金。
- 读交易时带上 `maxSupportedTransactionVersion`，并等到 `finalized`。

**接下来：**第19章。TON：不打电话，改写信。

# 第19章 TON：不打电话，改写信

## “发出去了，然后呢？”

顾客在 TON 网络上发来 USDT。在区块浏览器里，阿尔乔姆看到的不是一条记录，而是一长串：五笔交易、几个不认识的地址，还有一点零头找零。这是一笔付款还是五笔？

## 弄明白：个个都是合约，人人都在写信

**TON**（The Open Network）是一个独立的网络。它的币叫 **Gram**（GRAM）。以前叫 Toncoin，这个名字现在还常能见到。一个 Gram 等于十亿 **nanogram**。

从 2026 年 4 月起，区块大约每 0.4 秒出一个，记录大约一秒后成为定局。

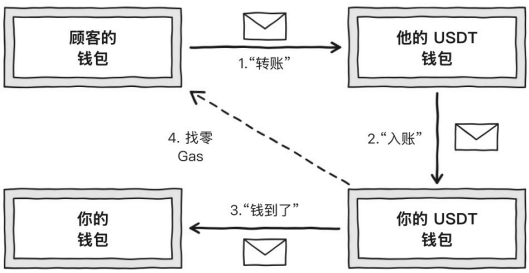
最大的不同在这里。这里的每个账户都是智能合约，**钱包也是**。合约之间不直接调用，而是互发**消息**。

在 Ethereum，一笔转账引发的所有动作都发生在同一笔交易里：要么全部成功，要么什么都不发生。这像打电话。TON 则是通信往来：信寄出去就不在线上等了。每封信由收件人随后处理，单独算一笔交易。这种方式叫**异步**。

由一个指令衍生出来的全部往来信件叫 **trace**（追踪链）。其中某一笔交易成功，还不代表整件事办成了。

## USDT 是怎么走的

TON 里的代币叫 **jetton**。每种 jetton 有一个主合约（**master**），而每个持有人有自己单独的 **jetton 钱包**。类似第18章的代币账户，只是它本身是一个独立的合约。USDT 的主合约是 `EQCxE6mUtQJKFnGfaROTKOt1lZbDiiX1kCixRv7Nw2Id_sDs`。




每封信单独用一笔交易处理

这不是编的例子，而是 2026 年 11 月 27 日凌晨一笔真实的 11.99 USDT 转账：

| 谁的交易 | 收到了什么 | 接着发出了什么 |
| --- | --- | --- |
| 发送方钱包 | 签好名的指令 | “转账”和 0.05 Gram 作费用 |
| 他的 USDT 钱包 | “转账” | “入账” |
| 收款方的 USDT 钱包 | “入账” | 通知和找零 |
| 收款方钱包 | 通知和 1 nanogram | 无 |
| 发送方钱包 | 找零，约 0.0494 Gram | 无 |

整个过程一秒钟之内完成。费用会多带一些，用剩的退回来。

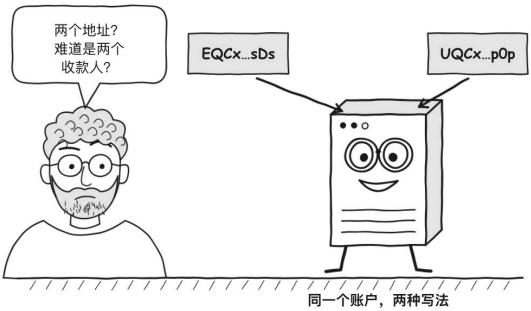
有个有趣的细节：第四笔交易被标记为中止（`aborted`）。收款方执行程序的 Gas 不够：随消息只来了一个 nanogram。而 USDT 早在第三笔交易里就已经入账了。只看某一条记录，无论判成功还是判失败都靠不住。

要看整条链，可以向索引器发请求。公共节点上它是关闭的，所以这是给个人端点用的草稿：

```
curl -G https://你的端点/api/v3/traces \
  --data-urlencode "trace_id=7rg6DNoAZeR1KPVJHd1FWIc/dobTITAsXqK1Y2kuQlM="
```

## 一个账户，两个地址

账户的真名是“原始格式”：网络分区编号加 64 个字符。给人看的是另一种写法，48 个字符，带校验和。而且同一个账户的这种写法不止一种。



同一个账户，两种写法

向节点查一下 USDT 的地址：

```
curl -s "https://public-ton-mainnet.rpcnode.dev/api/v2/detectAddress?address=EQCxE6mUtQJKFnGfaROTKOt1lZbDiiX1kCixRv7Nw2Id_sDs" |
  python3 -c "import json,sys
r = json.load(sys.stdin)['result']
print(r['raw_form'])
print(r['bounceable']['b64url'])
print(r['non_bounceable']['b64url'])"
```

```
0:b113a994b5024a16719f69139328eb759596c38a25f59028b146fecdc3621dfe
EQCxE6mUtQJKFnGfaROTKOt1lZbDiiX1kCixRv7Nw2Id_sDs
UQCxE6mUtQJKFnGfaROTKOt1lZbDiiX1kCixRv7Nw2Id_p0p
```

- `EQ…`——可退回（bounceable）地址，普通合约用这种写法。
- `UQ…`——不可退回地址，钱包建议用这种写法。
- 在测试网上，地址以 `kQ` 和 `0Q` 开头。

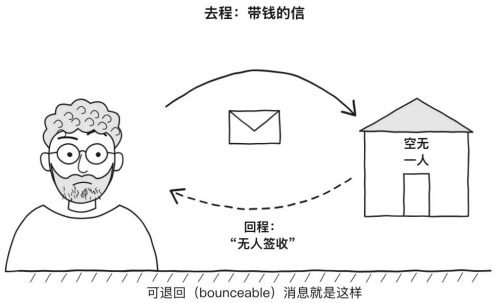
所以店铺数据库里**要比较原始格式**，而不是带 `EQ` 和 `UQ` 的字符串。

校验和能抓出打字错误。如果把末尾的 `sDs` 写成 `sDt`，节点会返回状态码 422：

```
curl -s -o /dev/null -w '%{http_code}\n' \
  "https://public-ton-mainnet.rpcnode.dev/api/v2/detectAddress?address=EQCxE6mUtQJKFnGfaROTKOt1lZbDiiX1kCixRv7Nw2Id_sDt"
```

## 寄到空房子的信

如果收件人处理不了这封信，或者他的合约还不存在，会怎样？这取决于消息上的标记。




可退回消息会带着剩下的钱退给发送方，前提是剩的钱够付回程。不可退回消息则留在收款方那里。新钱包可能还没有“部署”，也就是它的程序还没写进网络。所以给钱包付款要用不可退回地址。

### 动手试试：测一测速度

查出最新主链区块的编号，以及它和往前第一千个区块的出块时间：

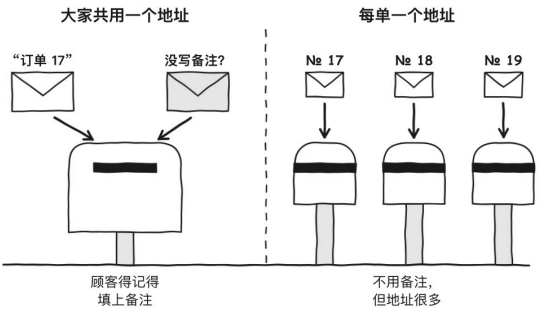
```
B=https://public-ton-mainnet.rpcnode.dev
last=$(curl -s $B/api/v2/getMasterchainInfo |
  python3 -c "import json,sys; print(json.load(sys.stdin)['result']['last']['seqno'])")
for s in $last $((last-1000)); do
  sleep 2
  curl -s "$B/api/v2/getBlockHeader?workchain=-1&shard=-9223372036854775808&seqno=$s" |
    python3 -c "import json,sys
r = json.load(sys.stdin)['result']
print(r['id']['seqno'], r['gen_utime'])"
done
```

```
108011078 1795732480
108010078 1795732076
```

第二列是以秒计的时间。1000 个区块相差 404 秒，即每个区块约 0.4 秒。

### 用备注代替单独地址

TON 的转账可以附带一段文字**备注**（comment）。于是辨认付款归属有两种办法：




用公共地址省事，但顾客可能忘了写备注，或者抄错。况且不是所有钱包都能添加备注。这类付款只能人工处理。

### 陷阱：“收到通知就发货”

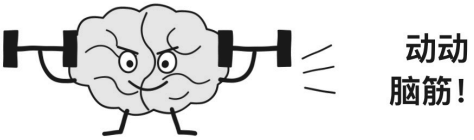
任何人都能写一个合约，给你的钱包发一条一模一样的通知。也能发行自己的、名叫“USDT”的 jetton。所以要核对：

- 通知来自**你的** USDT 钱包。它的地址要事先向真正的主合约查询（`get_wallet_address`）；
- 金额相符（USDT 有六位小数）；
- 备注或地址对应这个订单，并且这笔付款还没有记过账；

- 记录已经进入主链（masterchain），也就是已成定局。

只有发送方为通知附带了至少一点 Gram，通知才会发出。通常大家都会这么做，但自己写的程序可能忘记。那样代币到了，却没有通知。

**想一想**




1. 数据库里记的是 `UQ` 开头的地址，浏览器显示的是中间部分相同的 `EQ` 地址。怎样确认是不是同一个账户？
2. 链上第一笔交易成功了。能说明 USDT 到账了吗？
3. 顾客付到了公共地址，却忘了写备注。店铺怎么办？

*答案：1）把两个都换成原始格式（`detectAddress`）再比较。2）还不能：要等收款方的 USDT 钱包把钱入账。3）不要自动记账，人工核对：金额、发送方、转账编号。*

**本章要点**

- 在 TON，合约靠通信往来：一个指令会生出一串交易。
- 同一账户有好几种地址写法，要比较原始格式。
- 核对 jetton 要看主合约和你自己的 jetton 钱包。

**接下来：**第20章。别的路数：Sui 和 Aptos、XRP Ledger 和 Stellar、Cardano、Polkadot、Cosmos。

# 第20章 别的路数：Sui 和 Aptos、XRP Ledger 和 Stellar、Cardano、Polkadot、Cosmos

“XLM 收不收？”

一位顾客问能不能走 Stellar 付款。另一位想在 Sui 上发 USDC。阿尔乔姆已经搞懂了 Bitcoin 和 Ethereum。这些知识够用吗？不太够：这里另有一套规矩。

## 弄明白：对新网络提五个问题

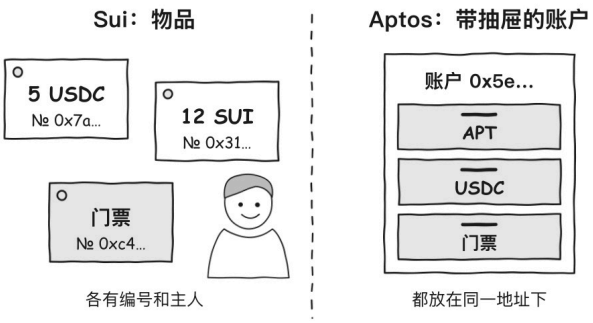
陌生的网络不必整个学一遍。起步阶段，弄清五个答案就够了：

1. 钱放在哪里：记在账户上，还是一块块分开？
2. 要事先做什么，才能收得到转账？
3. 怎么知道它付的是哪个订单？
4. 它什么时候成为定局？
5. 以后要把钱再转出去时，手续费用什么付？

## Sui 和 Aptos：语言相通，构造不同

这两个网络的程序都用 Move 语言编写。它保证贵重的东西不能随手复制。但钱的存放方式不一样。

- **Sui** 把一切都存成对象：它们像一件件物品，各有编号。钱可以是一个个独立的“币对象”，几乎像 Bitcoin 的钱块。从 2026 年起，地址还可以有一个总余额。完整余额是两者之和。
- **Aptos** 更接近我们熟悉的账户。财产堆放在地址名下，叫作资源（resource）。不需要事先开户。




两个网络通常都能在一秒内让转账成为定局。手续费用各自的币付：SUI 或 APT。Sui 有个例外：从 2026 年起，某些稳定币（包括 USDC）可以免手续费直接转账（0.01 起）。

两者的地址都是 `0x` 加通常 64 个字符。开头像 Ethereum，长度却不同：Ethereum 的 `0x` 后面是 40 个字符。

## 动手试试：在两个网络里找 USDC

本章涉及的网络，RpcNode 没有免费公共节点。所以我们向开发这些网络的机构提供的开放地址提问。这些地址是公用的，有限制：用来认识可以，开店不行。

Sui，用的方式叫 GraphQL（你列出需要哪些字段）：

```
curl -s https://graphql.mainnet.sui.io/graphql \
  -H 'Content-Type: application/json' \
  -d '{"query":"{ coinMetadata(coinType: \"0xdba34672e30cb065b1f93e3ab55318768fd6fef66c15942c9f7cb846e2f900e7::usdc::USDC\") { symbol decimals } }"}'
```

```
{"data":{"coinMetadata":{"symbol":"USDC","decimals":6}}}
```

Aptos，用第11章讲过的普通 REST：

```
curl -s https://api.mainnet.aptoslabs.com/v1/accounts/0xbae207659db88bea0cbead6da0ed00aac12edcdda169e591cd41c94180b46f3b/resource/0x1::fungible_asset::Metadata
```

```
{"type":"0x1::fungible_asset::Metadata","data":{"decimals":6,"icon_uri":"https://circle.com/usdc-icon","name":"USDC","project_uri":"https://circle.com/usdc","symbol":"USDC"}}
```

两边都不是靠“USDC”这个词找代币，而是靠发行方 Circle 公司清单上的那串长地址。这就是第9章的规则。两边的小数位都是六位。

**小心旧示例。**以前查 Sui 用的是 JSON-RPC。2026 年 7 月，Sui 基金会的节点把它关掉了，现在提供的是 gRPC 和 GraphQL。照抄文章里的示例之前，先看看文章的日期。

## XRP Ledger 和 Stellar：为转账而生的网络

这两个网络早就不只能转自己的币，也能记别人的代币。链上的“一页”在这里叫 **ledger**（账本）。XRP Ledger 每 3 – 5 秒确认一页新账本，Stellar 大约每 5 秒一页。确认过的不再改写。

但有两条不寻常的规矩。

**第一条：占地方要押钱。**账户上没有存够最低金额——**储备金**——之前，账户就不存在。要接收某种代币，比如 USDC，账户主人还得事先开一条**信任线**（trust line）：“是的，我愿意持有它。”每增加这样一条记录，储备金要求就提高一些。

| | XRP Ledger | Stellar |
|---|---|---|
| 地址开头 | `r` | `G` |
| 账户最低余额 | 1 XRP | 1 XLM |
| 每条 trust line 增加 | 0.2 XRP（前两条可能不用加） | 0.5 XLM |
| 最低手续费 | 0.00001 XRP | 每个操作 0.00001 XLM |
| 订单编号 | destination tag | memo，或 `M` 开头的地址 |

这些数字网络可以投票修改，所以要向节点查询：

```
curl -s https://s1.ripple.com:51234/ -H 'Content-Type: application/json' \
  -d '{"method":"server_info","params":[{}]}' | grep -o '"validated_ledger":{[^}]*}'
```

```
"validated_ledger":{"age":10,"hash":"AD65FEE0E665A2CA5D08D3757A9E5A65047ECC2391B5E118F3912F25C76AB64F","seq":10859
3026,"base_fee_xrp":1E-5,"reserve_base_xrp":1E0,"reserve_inc_xrp":2E-1}
```

`1E-5` 就是 0.00001，`1E0` 是 1，`2E-1` 是 0.2。再看 Stellar：

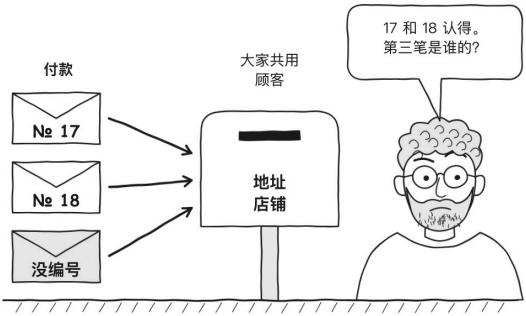
```
curl -s 'https://horizon.stellar.org/ledgers?order=desc&limit=1' \
  | grep -E '"(base_fee|base_reserve)_in_stroops"'
```

```
      "base_fee_in_stroops": 100,
      "base_reserve_in_stroops": 5000000,
```

一个 XLM 等于一千万 **stroop**。也就是说，手续费 0.00001 XLM，一份储备金 0.5 XLM（每个账户至少需要两份）。

## 大家共用一个地址

**第二条规矩。**在这里给每个订单开一个单独地址很不划算：每个地址上都得押储备金。所以店铺只展示一个地址，请顾客附上一个编号：XRP Ledger 里叫 **destination tag**（目标标签），Stellar 里叫 **memo**（备注）。

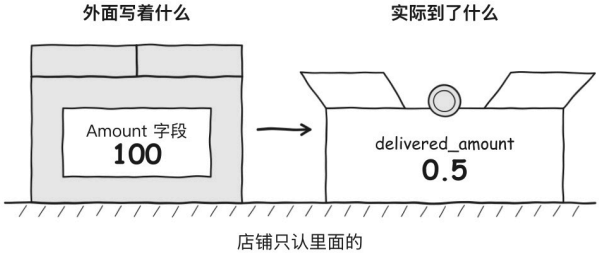


顾客忘了写编号——钱到了，付的是什么却说不清。几个有用的办法：

- XRP Ledger 里可以打开 `RequireDest` 设置：不带编号的转账网络不放行；
- 编号可以藏进地址本身：XRP Ledger 的 X 地址、Stellar 的 `M` 开头地址——前提是顾客的钱包认得。

### 陷阱：标签上写 100，里面只有 0.5

在 XRP Ledger，发送方可以允许**部分支付**（partial payment）：即使到账的少，转账也算成功。这时字段 `Amount` 表示的不是“到了多少”，而是“最多多少”。实际到账数额记在 `delivered_amount` 里。




下面是网络上一条真实记录，我们凭哈希查它：

```
curl -s https://s2.ripple.com:51234/ -H 'Content-Type: application/json' \
  -d '{"method":"tx","params":[{"transaction":"18F797F64D0255522223B1A404BA9CFF7EAE28E2ABB74CC91DD986E162D42DE
C"}]}' \
  | grep -o '"Amount":{[^}]*}\|"delivered_amount":{[^}]*}\|"TransactionResult":"[^"]*"'
```

```
"Amount":{"currency":"XXX","issuer":"rPXeYRpA9Ne3H7fwcM26ZNe98M3rkzTeg6","value":"9.552474017069"}
"TransactionResult":"tesSUCCESS"
"delivered_amount":{"currency":"XXX","issuer":"rPXeYRpA9Ne3H7fwcM26ZNe98M3rkzTeg6","value":"4.5344917829"}
```

结果是 `tesSUCCESS`，`Amount` 约 9.55，实际到账约 4.53。这样的记录本身并不是欺诈。但骗子可以写一个大数目，只发一点零头，然后等着收货。**只认** `delivered_amount`，并且核对是什么代币、由谁发行。

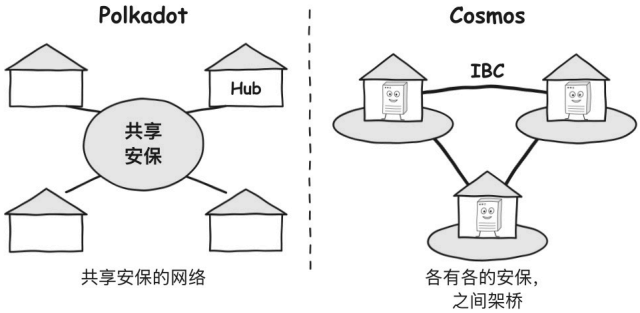
## Cardano：钱块里还能夹东西

Cardano 的构造更接近 Bitcoin（见第14章）：没有余额，只有未花费的钱块。不过它的钱块本事更大，这种模型叫**扩展 UTXO**（eUTXO）。

- 钱块里除了 ADA 币，还可以放别的代币。它们不需要单独的合约。
- 代币不能单独上路：同一个钱块里必须带一点 ADA。所以收到代币时会顺带收到一点点币。

- 手续费取决于交易的字节大小。
- 确定性是逐渐增长的：上面压的区块越多越可靠。

## Polkadot 和 Cosmos：不是一个网络，而是一群




**Polkadot** 是一条中心链，守护着接入它的各个网络。一个重要变化：从 2025 年 11 月起，用户的余额不再放在中心链上，而是放在 **Polkadot Hub** 网络（也叫 Asset Hub）。

- USDT（资产编号 1984）和 USDC（编号 1337）也在那里。
- 账户有一条生存门槛 **existential deposit**（生存押金）：0.01 DOT。如果 DOT 不够，USDT 或 USDC 也没有达到所需数量，账户就会被清除。
- 地址以 `1` 开头，和 Bitcoin 的老地址一样。别弄混。

**Cosmos** 是用通用部件搭成的一批独立网络。每个都有自己的安保、币和地址前缀（Cosmos Hub 是 `cosmos1…`）。区块很快成为定局。转账里有一个 **memo** 字段可以放订单号。

这些网络靠 **IBC** 协议相连。从邻居网络过来的代币，名字不是通常那个词，而是 `ibc/` 后面跟一长串印章，例如 `ibc/7F1D3FCF…`。印章说明它是沿哪条路过来的。

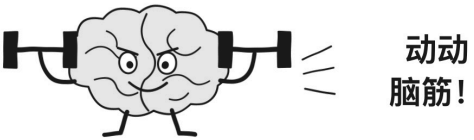
稳定币也可能撤走。Circle 宣布停止在 Cosmos 家族的 Noble 网络上发行 USDC：从 2026 年 10 月 13 日起停止新增发行，2027 年 1 月 12 日将停掉那里的 USDC。建议持有人提前转走。

## 汇总

| 网络 | 地址 | 需要事先做 | 何时成定局 |
|---|---|---|---|
| Sui | `0x…` | 无 | 不到一秒 |
| Aptos | `0x…` | 无 | 不到一秒 |
| XRP Ledger | `r…` | 储备金、trust line | 3 – 5 秒 |
| Stellar | `G…` | 储备金、trust line | 约 5 秒 |
| Cardano | `addr1…` | 无 | 逐渐确定 |
| Polkadot Hub | `1…` | 余额高于门槛 | 以标记为准 |
| Cosmos Hub | `cosmos1…` | 无 | 很快 |

*只有代币才需要 trust line。核实于 2026 年 11 月 27 日；规则会变。*

### 想一想




1. 阿尔乔姆新建了一个空的 Stellar 地址，马上开出一张 USDC 账单。哪里不对？最少需要多少 XLM？
2. 收到一笔转账：`Amount` 是 100 USDC，`delivered_amount` 是 0.5，结果 `tesSUCCESS`。按 100 发咖啡吗？
3. XRP Ledger 的顾客总忘记 destination tag。该打开什么？

*答案：1) 账户还不存在，trust line 也没开；至少需要 1.5 XLM 储备金，外加一点手续费。2) 不发：到账只有 0.5。3) RequireDest 设置。*

### 本章要点

- 面对陌生网络，先问完五个问题，再把地址给顾客看。
- XRP Ledger 和 Stellar 需要储备金、trust line 和订单号；XRP Ledger 只认 `delivered_amount`。
- Ethereum 里养成的习惯和旧示例，不能照搬到别的网络。

接下来：第21章。该选哪个网络。

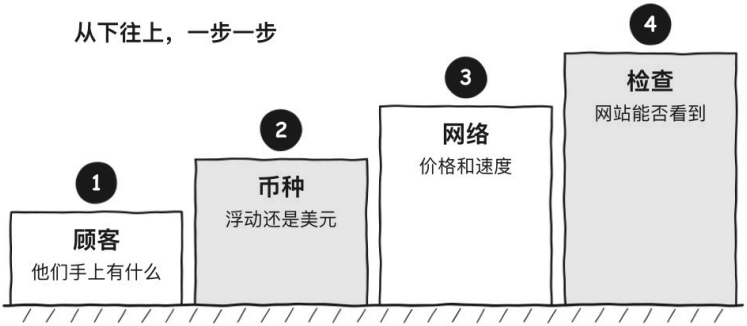
# 第21章 该选哪个网络

## “这么多，到底选哪个？”

七章下来攒了一大堆名字：Bitcoin、Ethereum、TRON、Solana、TON，还有另外十来个。阿尔乔姆只有一家网店和一间咖啡烘焙坊。全部接上？只挑一个？按什么标准挑？

## 弄明白：四个步骤

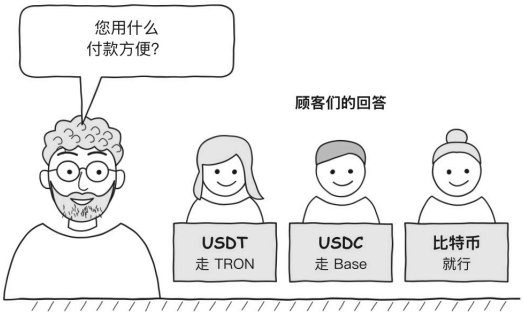
没有“最好的网络”。只有适合你的顾客、适合你订单的网络。按顺序一步步来最省事。




这里讲的是用什么收款方便，不是建议你把钱投到哪里。

## 第一步：问问顾客

付钱的是手里已经有币的人。如果他的币在一个网络上，你收的是另一个网络，这笔钱多半就付不成了。



在群发邮件里问，在聊天群里问，在下单时问。两个问题就够了：“你手上有什么币？”“在哪个网络上？”

## 第二步：用哪种币

| | 网络的原生币 | 稳定币（USDT、USDC） |
|---|---|---|
| 美元价格 | 会变，有时变得很猛 | 尽力保持在一美元左右 |
| 谁说了算 | 网络规则 | 有发行方，它可以冻结（见第9章） |
| 手续费用什么付 | 就用它自己 | 通常用网络原生币，而不是稳定币 |

如果阿尔乔姆的咖啡以美元定价或与美元挂钩，收稳定币意外会少一些。但这同样是一种信任：信任发行它的公司。

## 第三步：用哪个网络

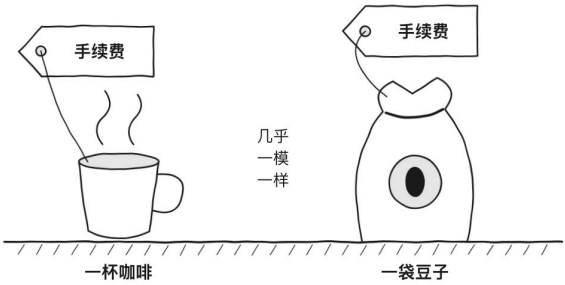
同一种稳定币并不是哪儿都有。下面是第14 – 20章讲过的网络。“是”的意思是：发行方网站上为这个网络列出了这种币本身的地址。

| 网络 | 手续费 | Tether 的 USDT | Circle 的 USDC | 别忘了 |
|---|---|---|---|---|
| Bitcoin | BTC | 否 | 否 | 确认数 |
| Litecoin、Doge 等 | 各自的币 | 否 | 否 | 网络各不相同 |
| Ethereum | ETH | 是 | 是 | Gas |
| Base、Arbitrum | ETH | 否 | 是 | finalized |
| BNB Smart Chain | BNB | 否 | 否 | 查发行方 |
| TRON | TRX | 是 | 否 | 能量 |
| Solana | SOL | 是 | 是 | 代币账户 |
| TON | Gram | 是 | 否 | 消息链 |
| Aptos | APT | 是 | 是 | 长地址 |
| Sui | SUI | 否 | 是 | gRPC、GraphQL |
| Polkadot Hub | DOT | 是 | 是 | 账户门槛 |
| XRP Ledger | XRP | 否 | 是 | 储备金、标签 |
| Stellar | XLM | 否 | 是 | 储备金、标签 |
| Cardano | ADA | 否 | 否 | 代币要带 ADA |

*据 tether.to/en/supported-protocols 和 developers.circle.com，截至 2026 年 11 月 27 日。“否”不代表那里没有叫这个名字的代币：要查清是谁发行的（见第9章）。Cosmos 的情况取决于具体网络。*

## 转账费用和订单大小

手续费不看金额，看交易的“分量”和网络繁忙程度（见第7章）。




对一袋咖啡豆来说，这是小钱。对一杯咖啡来说，手续费可能比咖啡本身还贵。所以做小额订单要看哪里转账便宜。价格是变动的，应当在发送前到钱包里看，而不是查一张旧表。

还要记着回程：要把收到的钱转出去，地址上需要有付手续费的币。一百个地址就是一百笔要付费的转账。

## 要等多久

| 多久成为定局 | 网络 |
|---|---|
| 几秒或更快 | BNB Smart Chain、Solana、TON、Aptos、Sui、XRP Ledger、Stellar |
| 约一分钟 | TRON |
| 几十分钟 | Ethereum、Base、Arbitrum One |
| 约一小时 | Bitcoin（6 个确认） |

*大致数值，依据第8章和第17 – 20章。*

## 动手试试：定局区块有多“旧”

向几个 EVM 网络要 `finalized` 区块（见第8章），算一算它是多少秒前出的：

```
for net in ethereum base arb bsc; do
  t=$(curl -s https://public-$net-mainnet.rpcnode.dev \
    -H 'Content-Type: application/json' \
    -d '{"jsonrpc":"2.0","id":1,"method":"eth_getBlockByNumber","params":["finalized",false]}' \
    | grep -o '"timestamp":"0x[0-9a-f]*"' | cut -d'"' -f4)
  echo "$net: $(( $(date +%s) - t )) 秒"
  sleep 1.5
done
```

```
ethereum: 929 秒
base: 1343 秒
arb: 990 秒
bsc: 1 秒
```

Ethereum 约 15 分钟，Base 约 22 分钟。BNB Smart Chain 只要一秒。你看到的数会不同：这个标记是一跳一跳往前走的。为小订单未必要等到最可靠的标记，但那样风险就由你来担。

### 第四步：你的网站看得到吗

在把地址印到网站上之前，先核实：

- 服务商有没有这个网络的端点（见第13章）；
- 这个网络有没有付款通知。这是两回事。在 RpcNode，能开端点的网络比能解析出事件的网络要多（见第26、27章）；
- 有没有测试网，可以不用真钱先试；
- 你的钱包和你换币的地方，是否支持在这个网络上的这种币。

还要看一眼套餐：在 RpcNode 的 Free 套餐里，只有 Ethereum 和 Solana 能开端点（见第22章）。

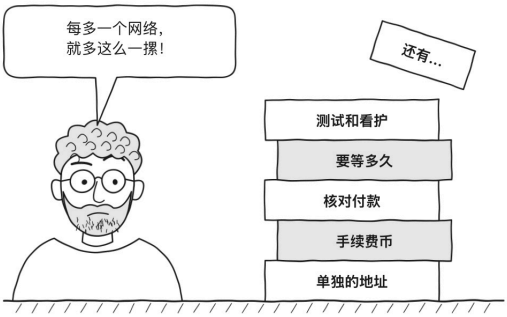
### 例子：阿尔乔姆怎么决定

假设他问了 20 位顾客。数字是教学用的，你店里的数字会是自己的。

| 想用什么付 | 人数 | 决定 |
|---|---|---|
| TRON 上的 USDT | 9 | 马上接入 |
| Base 上的 USDC | 5 | 马上接入 |
| BTC | 3 | 以后，如果还有人要 |
| ETH | 2 | 以后 |
| XLM | 1 | 暂时不接 |

两个网络覆盖了 20 人中的 14 人。对其余的人，阿尔乔姆如实写明暂时还不是什么都收。

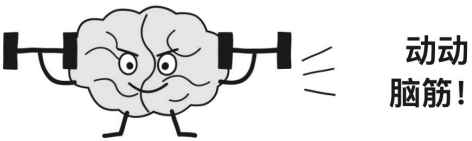
### 陷阱：“一口气全接上”




付款页面上多一行看起来不花钱。可它背后是单独的地址、单独的手续费币种、单独的核对规则，以及升级之后单独的意外。先接一两个，有人提需求再加。

还有第二个陷阱：付款页面上只写“USDT”。顾客会自己选网络，可能选错（见第16章）。币种和网络要写在一起，写大一点：“TRON 网络的 USDT”。

#### 想一想




1. 订单 5 美元，而选定网络今天的手续费是 2 美元。对谁不方便，该怎么办？
2. 顾客要求收 Base 上的 USDT。Tether 网站的清单显示什么？
3. 阿尔乔姆想第一天就接八个网络。你怎么劝他？

*答案：1) 对顾客：手续费等于在订单上又加了 40%；可以提供一个当前转账更便宜的网络。2) 清单里没有 Base：接收叫这个名字的代币之前，先弄清是谁发行的。3) 那是八摞操心事；先接顾客最需要的一两个。*

#### 本章要点

- 按顺序挑选：顾客、币种、网络、检查。
- 拿手续费和订单金额作比较，并考虑要等多久。
- 从一两个网络起步，提到币种时永远连网络一起说。

**第四部分到此结束**。阿尔乔姆知道了各个网络的区别，也决定了从哪儿起步。**接下来**：第五部分，第22章。在 RpcNode 的头五分钟。

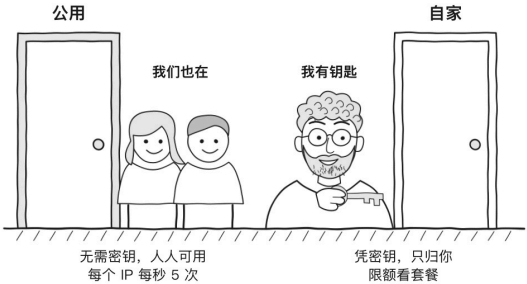
# 第22章 在 RpcNode 的头五分钟

## “我要一扇自己的门”

整本书读到现在，阿尔乔姆敲的都是公共节点的门。不用注册它们就应答，可那扇门是大家共用的。开店得有自己的。弄一扇要很久吗？要填些什么？

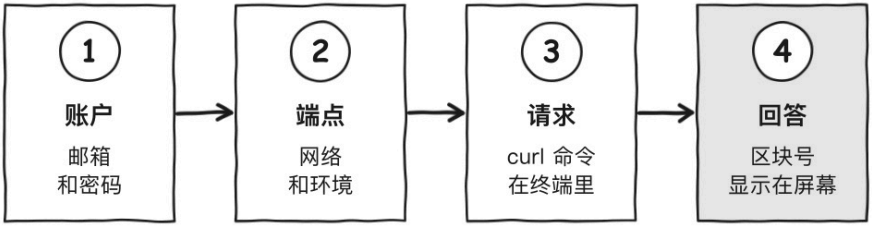
## 弄明白：公用的门和自己的门

程序向节点提问所用的地址叫端点（见第3章）。在 RpcNode 有两种。




| | 公共地址 | 个人端点 |
|---|---|---|
| 注册 | 不需要 | 需要 |
| 密钥 | 无 | 嵌在地址里 |
| 限制 | 每个 IP 每秒 5 个请求 | 按套餐 |
| 重型方法 | 关闭 | 开放 |
| 归档（见第24章） | 关闭 | 付费套餐可用 |
| WebSocket 和 gRPC | 无 | 网络支持就有 |

四步就能拿到自己的端点。




## 第一步：注册

打开 `cabinet.rpcnode.dev/register`，或者在 rpcnode.dev 网站上点“开始使用”。控制台是你在这项服务里的个人页面：端点、用量和套餐都在这儿。

1. 填写 Email，设一个密码并再输入一遍。名字可以不填。
2. 阅读并勾选“我同意服务条款”。如果出现“你不是机器人”的验证，完成它。
3. 点“继续”。
4. 邮箱会收到一个一次性验证码。输入后点“验证并继续”。

也有捷径：用 Google 或 GitHub 登录。

新账户默认是免费的 **Free** 套餐，不需要银行卡。包括每月 1500 万额度、每秒 15 个请求、3 个端点和两个网络：Ethereum 和 Solana。额度是什么，第25章讲。

## 第二步：端点

在控制台菜单里选“端点”，点“新建端点”。会弹出一个分三步的窗口：

| 窗口里的步骤 | 做什么 |
|---|---|
| 选择网络 | 比如 Ethereum |
| 选择环境 | 主网或测试网（下面会讲） |
| 确认 | 核对后点“创建” |

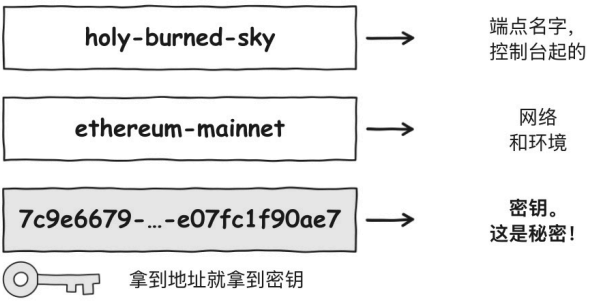
环境是网络的版本：真实的或练习用的。端点的名字由控制台用三个英文单词自动生成。

在 Free 套餐上，创建之前你会看到一条关于归档节点的提示。这不是错误，第24章会讲它。

## 地址由什么组成

下面是 RpcNode 文档里的例子。它本是一行，这里在斜杠前换了行：

```
https://holy-burned-sky-ethereum-mainnet.rpcnode.dev
/7c9e6679-7425-40de-944b-e07fc1f90ae7
```




斜杠后面那一长串叫 UUID：一个实际上猜不中的编号。它就是密钥。请求里不需要另外附加密码。

地址要用控制台里的按钮复制，不要手打。屏幕上密钥有一部分被遮住，复制到的却是完整的。

*菜单里还有“API 密钥”。那是另一回事：它用来给发给服务本身的指令签名，例如“创建一笔付款”。向节点提普通问题用不着它。*

## 第三步：动手试试第一个请求

把地址放进配置，查最新区块号。这是一段草稿：把中文占位文字换成你自己的地址。

```
export RPC_URL='你在控制台里的地址'
curl -s "$RPC_URL" \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,"method":"eth_blockNumber","params":[]}'
```

个人地址只有你自己有，所以书里展示的是同样几行换成公共地址后的回答：`RPC_URL='https://public-ethereum-mainnet.rpcnode.dev'`。

```
{"jsonrpc":"2.0","id":1,"result":"0x19464e4"}
```

`0x19464e4` 是编号 26 502 372 的区块。第3章节点报的是 26 502 035：这期间网络又多了 337 个区块。

Solana 要把 `eth_blockNumber` 换成 `getSlot`。

控制台的“概览”页面上有一个“开始使用”清单。里面放着一条已经填好你地址的现成命令，还有一行字“等待第一个请求…”。端点页面的“实时”标签页上则能看到请求本身：时间、方法、响应和延迟。

## 如果没成功

我们故意用一个全是零的假密钥试一下：

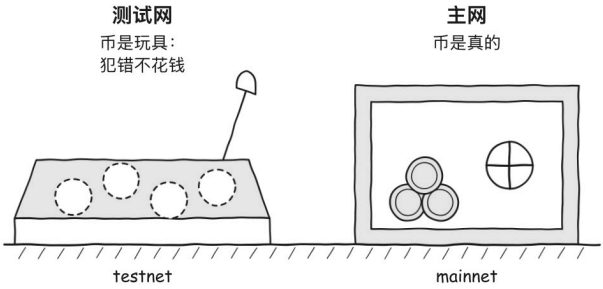
```
HOST=https://holy-burned-sky-ethereum-mainnet.rpcnode.dev
KEY=00000000-0000-0000-0000-000000000000
curl -s -w '  HTTP %{http_code}\n' $HOST/$KEY \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,"method":"eth_blockNumber","params":[]}'
```

```
{"jsonrpc":"2.0","id":1,"error":{"code":-32000,
"message":"endpoint not found"}}  HTTP 404
```

`endpoint not found`——“没有这个端点”。检查地址是否完整复制。不要拿错误地址循环重试：一直不停地试，会被暂时拒之门外。

## 测试网：用玩具钱的演练场

**测试网**（testnet）按和真实网络相同的规则运行，但它的币一文不值。真实网络叫主网（mainnet）。在测试网上犯错不花钱。




测试币从“水龙头”（faucet）领取：这是一种网页，会把测试币发到你填写的地址。

| 网络 | RpcNode 中的测试环境，例如 |
|---|---|
| Ethereum | Sepolia、Hoodi |
| Solana | testnet、devnet |
| Bitcoin | testnet4、signet |
| TRON | Nile、Shasta |

开发 Ethereum 应用，ethereum.org 网站建议用 Sepolia。Hoodi 是给测试网络本身和验证者的人用的。

也有不用注册的测试网：截至 2026 年 11 月 27 日，公共节点列表里有 Bitcoin testnet4、TRON Nile 和 TON testnet。我们分别问问 TRON 主网和测试网的网络编号（见第16章）：

```
for host in public-tron-mainnet public-tron-nile; do
  echo -n "$host: "
  curl -s https://$host.rpcnode.dev \
    -H 'Content-Type: application/json' \
    -d '{"jsonrpc":"2.0","id":1,"method":"eth_chainId","params":[]}'
  echo; sleep 2
done
```

```
public-tron-mainnet: {"jsonrpc":"2.0","id":1,"result":"0x2b6653dc"}
public-tron-nile: {"jsonrpc":"2.0","id":1,"result":"0xcd8690dc"}
```

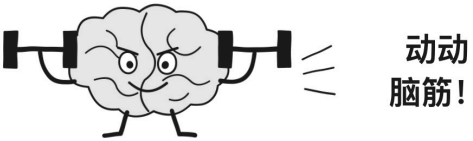
编号不同：这是两本不同的账。端点同样绑定在一个网络、一个环境上。要从测试网换到主网，得新建一个。

### 陷阱：“我把地址直接写进网页里”

凡是进了网页代码的东西，每个访客都看得见。他会拿走你的地址，花掉你的额度。

- 从自己的服务器向节点提问，不要从顾客的浏览器发。
- 地址放在服务器配置里，不要写进代码，也不要进公共代码仓库。
- 在端点页面的“设置”里配置“IP 白名单”：这样别人的电脑会被拒绝。
- “允许的 Origin”能阻止别人的网站在浏览器里调用端点。但它挡不住浏览器以外的程序。
- 地址泄露了？新建一个，让网站切换过去，再把旧的删掉：删掉后无法找回。

**想一想**




1. 阿尔乔姆为了显示区块号，把端点地址写进了店铺网页的脚本里。谁会看到密钥？
2. 在 Sepolia 上一切测试通过。能用同一个端点收真实付款吗？
3. 返回 `endpoint not found`。从哪里开始排查？

*答案：1) 任何访客。2) 不能：需要主网端点。3) 和控制台里的地址对一对：重新完整复制一次。*

**本章要点**

- 个人端点是一个内含密钥的地址：像保管密码一样保管它。
- 端点绑定一个网络和一个环境。
- 先上测试网，再上主网。

**接下来：**第23章。请求经历了什么：网关、节点选择和三个地区。

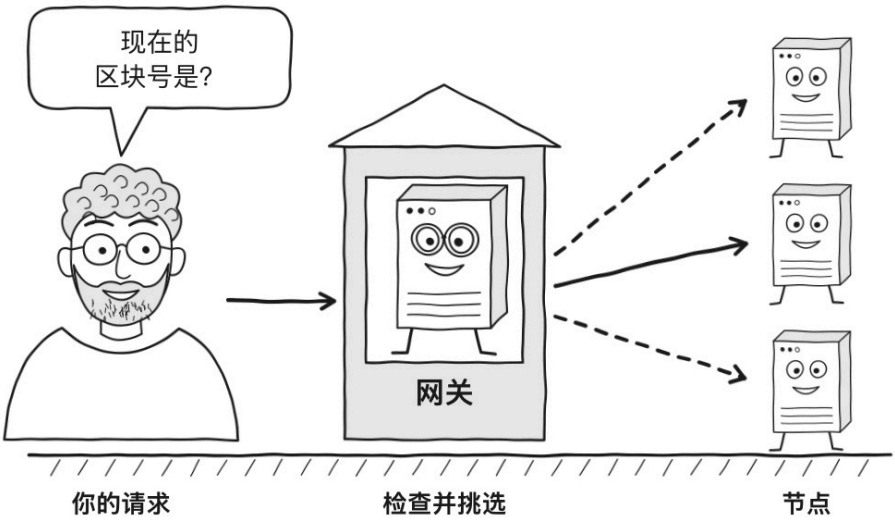
# 第23章 请求经历了什么

## “是谁回答的我？”

阿尔乔姆发出请求，收到了区块号。可是问题到哪儿去了？是哪个节点答的？要是它在大促正火的时候坏了，会怎样？

## 弄明白：入口处站着网关

你的请求并不直接进到节点。它先到达网关。网关是个调度程序：接收请求、检查，再转给合适的节点。




## 网关一路上检查什么

过程像过门禁。一项没通过，请求就不再往下走。

| 检查项 | 没通过时 |
| --- | --- |
| 端点是否存在，地址是否正确 | HTTP 404，`-32000` |
| 端点是否停用；你的 IP 和网站是否被允许 | HTTP 403，`-32000` |
| 问得是否太频繁 | HTTP 429，`-32005` |
| 这个网络有没有这个方法 | HTTP 404，`-32601` |
| 额度够不够 | HTTP 402，`-32006` |

每个网络有自己的方法清单。我们问一个清单上没有的：

```
curl -s -w '  HTTP %{http_code}\n' https://public-ethereum-mainnet.rpcnode.dev \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,"method":"eth_noSuchMethod","params":[]}'
```

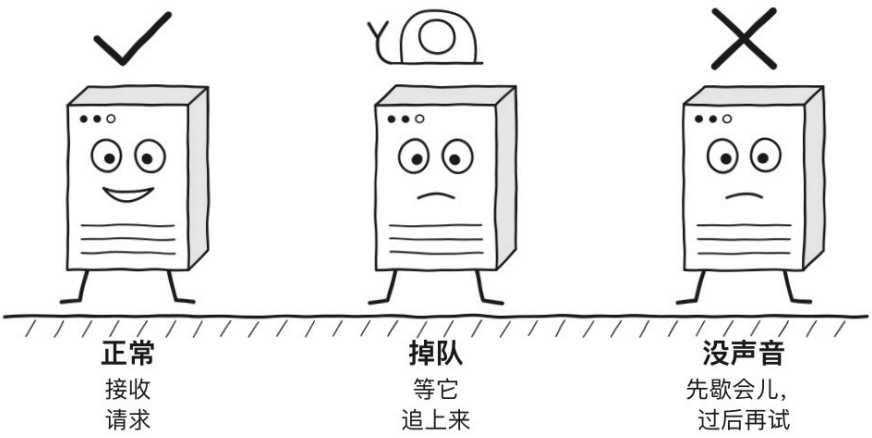
```
{"jsonrpc":"2.0","id":1,"error":{"code":-32601,
"message":"method not found: eth_noSuchMethod"}}  HTTP 404
```

这个问题根本没到节点：网关自己就回了。

## 挑节点：负载均衡

一个网络可以有好几个节点。把请求分配给它们，叫作**负载均衡**。就像超市的收银台：新来的顾客引到比较空的那一台。

网关先看节点状态是否正常。



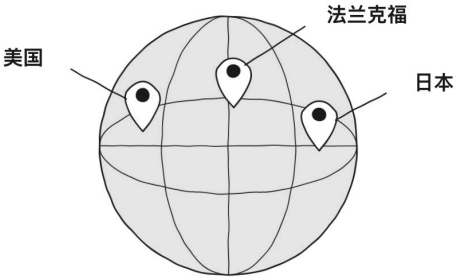

- **掉队的**节点回答得很爽快，答的却是昨天的事（见第12章）。只要有跟得上的节点，请求就发给它们。

- 健康节点里，忙的少分，闲的多分。
- 连接断了，或节点返回内部错误——同一个请求转给下一个。你察觉不到。
- 连续几次没办好事的节点，先晾一段时间，之后再试。

有一点要说明。如果健康节点一个都不剩了，网关还是会去试掉队的节点：有时旧答案总比没答案好。所以在新鲜度要紧的地方，自己查一下区块时间（见第12章）。

## 三个地区

RpcNode 在三个地区运行：法兰克福、美国和日本。节点越近，来回路程越短。网关尽量选离自己最近的健康节点，远处的留作后备。




RpcNode 的三个地区（示意图，非地图）

并非每个网络在三处都有节点。你需要的网络在哪里应答、状态如何，看 `rpcnode.dev/status` 页面。

## 动手试试：读信封上的戳记

回答会带着**响应头**——一些附加标注，像信封上的邮戳。其中两条说明网关在哪个国家、节点在哪个国家：

```
for net in ethereum base arb; do
  echo "== $net"
  curl -s -D - -o /dev/null https://public-$net-mainnet.rpcnode.dev \
    -H 'Content-Type: application/json' \
    -d '{"jsonrpc":"2.0","id":1,"method":"eth_blockNumber","params":[]}' |
    grep -iE 'x-gateway-region|x-upstream-country'
  sleep 2
done
```

```
== ethereum
x-gateway-region: DE
x-upstream-country: DE
== base
x-gateway-region: DE
x-upstream-country: JP
== arb
x-gateway-region: DE
x-upstream-country: JP
```

`DE` 是德国，`JP` 是日本。这一次所有请求都由德国的网关接收，而 Base 和 Arbitrum 的数据是到日本取的。你看到的字母可能不同。这些只是内部标注：不要让网站的功能依赖它们。

## 节点报的错，也是回答

我们问 USDT 合约：如果从一个空地址转一笔天文数字，会怎样？方法 `eth_call` 只是试一试，什么都不发送。字段 `data` 里拼着转账函数的编号、收款人（`TO`）和金额（`SUM`）。

```
TO=0000000000000000000000000000000000000000000000000000000000000dead
SUM=0000000000000000000000000000000000000000000000000fffffffffffffff
curl -s -w '  HTTP %{http_code}\n' https://public-ethereum-mainnet.rpcnode.dev \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,"method":"eth_call","params":[{
  "from":"0x000000000000000000000000000000000000dEaD",
  "to":"0xdac17f958d2ee523a2206206994597c13d831ec7",
  "data":"0xa9059cbb'$TO$SUM'"},"latest"]}'
```

```
{"jsonrpc":"2.0","id":1,"error":{"code":-32000,
"message":"invalid opcode: INVALID"}}  HTTP 200
```

这个老合约就是这么说“办不到”的。节点完成了任务：它如实报告程序中止了。换一个节点也会这么答，所以网关把这样的回答原样交给你，状态码 HTTP 200。

## 谁都没回答的时候

还有更糟的：网关把节点挨个试了一遍，还是没有回答。这时会返回代码 `-32002`、文字 `upstream unavailable`（“节点不可用”）和一个 5xx 的 HTTP 状态码。这次调用的额度会退还。

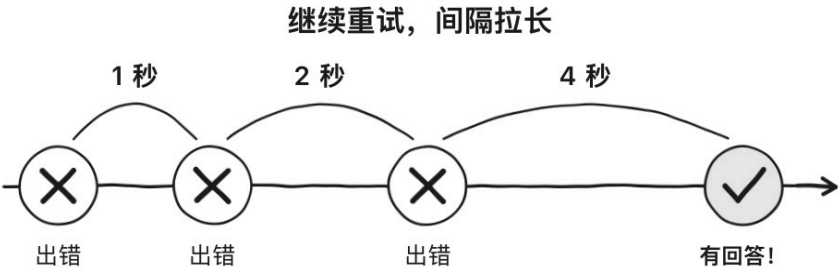
## 如果是发送资金呢？

如果节点拒绝发送交易，那就是回答。如果它干脆没有任何回应，网关可以把同一笔签好名的交易转给另一个节点。

钱不会因此转出两次：签好名的交易哈希是固定的，网络只接受一次。但如果没收到回答的是你，不要想当然地签一笔新交易。先按哈希查一查旧的（见第7章）。

### 动手试试：聪明地重试

网关会换节点，但你和网关之间的连接也可能出问题。要自己重试，间隔越来越长：1、2、4、8秒。英文叫 backoff（退避）。




文件 `ask.js`，运行命令 `node ask.js`（需要 Node.js）：

```
const url = 'https://public-ethereum-mainnet.rpcnode.dev'
const sleep = (ms) => new Promise((ok) => setTimeout(ok, ms))

async function ask(method, params = []) {
  for (const pause of [1, 2, 4, 8]) {
    try {
      const res = await fetch(url, {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify({ jsonrpc: '2.0', id: 1, method, params }),
        signal: AbortSignal.timeout(10000),
      })
      if (res.status !== 429 && res.status < 500) return await res.json()
      console.log('HTTP', res.status)
    } catch {
      console.log('无法连接')
    }
    await sleep(pause * 1000)
  }
  throw new Error('查询失败')
}

ask('eth_blockNumber').then(console.log)
```

```
{ jsonrpc: '2.0', id: 1, result: '0x19464ee' }
```

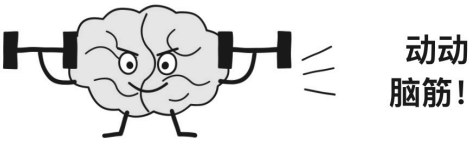
只重试那些第二次可能成功的情况：“太频繁”、断线、5xx。请求本身写错了，重试多少次都没用。拿到回答后别忘了检查 `error` 字段（见第11章）。

### 陷阱：“没回应”不等于“没付款”

如果没问成，网站并不知道顾客付没付。要写“正在核对付款”，而不是“未付款”，过一会儿再问。

还要记住：拥有多个节点的服务商比单个节点出故障的次数少，但不是“从不”（见第3章）。

### 想一想




1. 返回 HTTP 200 和 `execution reverted`。把请求重试五次有用吗?
2. 程序请求了 `eth_blokNumber`。这个请求到节点了吗?
3. 阿尔乔姆发了一笔转账，迟迟没等到回应。第一件事做什么?

*答案：1) 没用：这是真实的回答，不是故障。2) 没有，网关拒绝了不认识的方法。3) 按哈希查这笔交易。*

### 本章要点

- 网关检查请求并挑选健康节点；一个没回应就试另一个。
- 节点本身针对你的请求报的错，会原样交给你。
- 你的程序仍然应当带间隔地重试。

**接下来：**第24章。历史与归档数据。

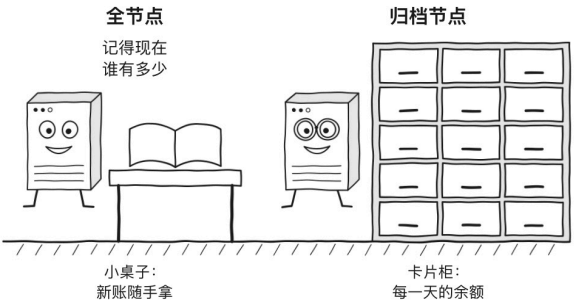
# 第24章 历史与归档数据

## “一月一号有多少？”

会计请阿尔乔姆帮忙：“说说年初的时候店铺地址上有多少钱。”问今天的事，节点秒回。问那时候呢？

## 弄明白：记录和余额

回忆一下第10章。历史是区块和转账。状态是某个特定时刻谁拥有什么。




全节点手边放着近期的余额。归档节点记得任意区块时的余额，代价是需要大得多的硬盘。

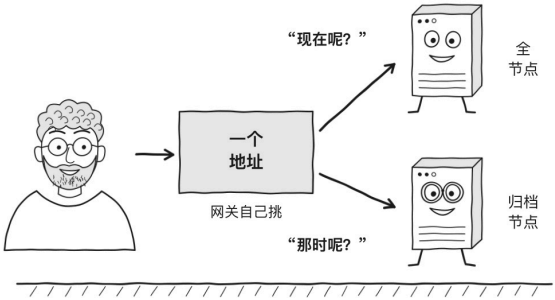
## 哪些问题需要归档

| 问题 | 例子 |
| --- | --- |
| 旧区块时的余额或合约回答 | 带区块号的 `eth_getBalance`、`eth_call` |
| 过去区块的事件 | `eth_getLogs` |
| 把交易一步步“回放”（追踪，trace） | `trace_…`、`debug_…` |
| 状态证明 | 针对旧区块的 `eth_getProof` |
| 很久以前的 Solana slot | 带旧编号的 `getBlock` |

而“账户现在有多少”以及普通的新近付款，并不需要归档。店铺的日常工作通常用不上它。

## RpcNode 是怎么做的

端点地址只有一个。不用自己挑节点：网关看问题内容，决定交给谁。




**归档包含在付费套餐里**，Free 套餐没有。在 Free 套餐上创建端点时，控制台会显示：

> “免费套餐不支持归档节点。您的请求由普通节点处理，节点保存的数据都可以返回。只有归档节点才保留的数据（旧状态、久远历史、部分追踪）无法获取，节点会返回它自己的错误。”

- 问“老问题”没有单独的价格：按方法收费（见第25章）。
- 各网络的归档深度不同。在它之上搭建重要功能之前，先在你的网络上试一试你最老的那个问题。
- 公共地址不处理归档问题。

## 动手试试：公共节点很严格

这次不问“现在”的余额，而是按区块号查，这个区块仅仅在十个区块之前，约两分钟：

```
E=https://public-ethereum-mainnet.rpcnode.dev
H='Content-Type: application/json'
N=$(curl -s $E -H "$H" \
  -d '{"jsonrpc":"2.0","id":1,"method":"eth_blockNumber","params":[]}' |
  grep -o '0x[0-9a-f]*')
OLD=$(printf '0x%x' $((N - 10)))
sleep 2
curl -s -w '  HTTP %{http_code}\n' $E -H "$H" \
  -d '{"jsonrpc":"2.0","id":1,"method":"eth_getBalance",
  "params":["0xdac17f958d2ee523a2206206994597c13d831ec7","'$OLD'"]}'
```

```
{"jsonrpc":"2.0","id":1,"error":{"code":-32601,"message":
"archive data is not available on the public endpoint,
use a personal endpoint"}}  HTTP 403
```

就连这么近的过去，公共地址也算作归档。向它提问要用第8章那几个词：`latest`、`safe` 或 `finalized`。

## 第二个实验：节点也可能已经扔掉了

区块属于历史，不属于状态。但历史也不是人人都存。从 2025 年 7 月起，Ethereum 节点程序可以不保留改用质押之前那些区块的内容（见第10章）。我们要 2015 年 8 月的第 46 147 号区块（`0xb443`）：

```
curl -s -w '  HTTP %{http_code}\n' https://public-ethereum-mainnet.rpcnode.dev \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,"method":"eth_getBlockByNumber",
  "params":["0xb443",false]}'
```

```
{"jsonrpc":"2.0","id":1,"error":{"code":4444,
"message":"pruned history unavailable"}}  HTTP 200
```

“已裁剪的历史不可用”。节点老实说：我这儿没有。在付费套餐上，网关从普通节点得到“我没有”的答复后，如果这个网络有归档节点，可以再去问归档节点。

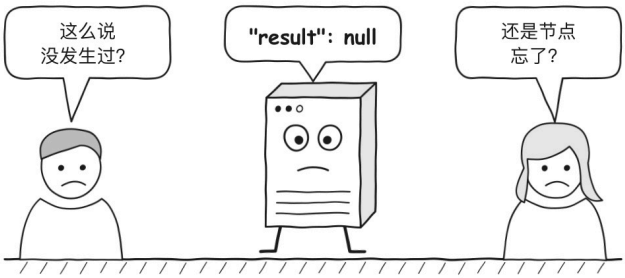
## 陷阱：“null 就是从来没有过”

就在那个区块里有一笔转账。我们按哈希查它，接着再查一个纯属瞎编的哈希：

```
for hash in \
  0x5c504ed432cb51138bcf09aa5e8a410dd4a1e204ef84bfed1be16dfba1b22060 \
  0x0000000000000000000000000000000000000000000000000000000000000001
do
  curl -s https://public-ethereum-mainnet.rpcnode.dev \
    -H 'Content-Type: application/json' \
    -d '{"jsonrpc":"2.0","id":1,"method":"eth_getTransactionByHash",
    "params":["'$hash'"]}'
  echo; sleep 2
done
```

```
{"jsonrpc":"2.0","id":1,"result":null}
{"jsonrpc":"2.0","id":1,"result":null}
```

回答一模一样！第一笔转账确实发生过，第二笔从来不存在。




光看回答分不出

不要凭一个 `null` 下结论，尤其是对老记录。它有时是“没有过”，有时是“我没存”，对刚发的转账则是“还没进区块”。

## Solana 节点记得多久

Solana 每秒出好几个区块，记录堆积如山。我们查最新的 slot 编号，以及节点还保留着的最老区块：

```
for method in getSlot getFirstAvailableBlock; do
  echo -n "$method: "
  curl -s https://public-solana-mainnet.rpcnode.dev \
    -H 'Content-Type: application/json' \
    -d '{"jsonrpc":"2.0","id":1,"method":"'$method'"}'
  echo; sleep 2
done
```

```
getSlot: {"jsonrpc":"2.0","result":474316453,"id":1}
getFirstAvailableBlock: {"jsonrpc":"2.0","result":473916707,"id":1}
```

相差约 400 000 个 slot。检查时这只相当于一天一夜。而 1000 号区块早就清掉了：

```
curl -s https://public-solana-mainnet.rpcnode.dev \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,"method":"getBlockTime","params":[1000]}'
```

```
{"jsonrpc":"2.0","error":{"code":-32001,"message":"Block 1000 cleaned up,
does not exist on node. First available block: 473916707"},"id":1}
```

## 回到会计的问题

“年初”的余额，就是旧年最后一个区块之后的余额（时间按世界协调时 UTC 算）。节点没法按日期直接报出区块号。好在区块是按顺序排的，可以用**二分查找**：取中间的区块，看它的时间，把用不着的那一半丢掉。

我们来找 2026 年的第一个区块。脚本运行大约一分钟：

```
E=https://public-ethereum-mainnet.rpcnode.dev
ask() {  # method, params
  curl -s $E -H 'Content-Type: application/json' \
    -d '{"jsonrpc":"2.0","id":1,"method":"'$1'","params":'$2'}'
  sleep 2
}
WHEN=1767225600   # 2026 年 1 月 1 日 00:00 UTC
low=20000000
high=$(( $(ask eth_blockNumber '[]' | grep -o '0x[0-9a-f]*') ))
steps=0
while [ $low -lt $high ]; do
  mid=$(( (low + high) / 2 ))
  hex=$(printf '0x%x' $mid)
  t=$(ask eth_getBlockByNumber '["'$hex'",false]' |
    grep -o '"timestamp":"0x[0-9a-f]*"' | grep -o '0x[0-9a-f]*')
  if [ -z "$t" ]; then echo "没有回应"; exit 1; fi
  if [ $(( t )) -lt $WHEN ]; then low=$(( mid + 1 )); else high=$mid; fi
  steps=$(( steps + 1 ))
done
echo "区块 $low, 提问次数: $steps"
```

```
区块 24136053, 提问次数: 23
```

在几百万个区块里，问 23 次就找到了要的那个（你那边次数可能差一次）。也就是说，2025 年最后一个区块是 24 136 052（`0x1704974`）。查那一块时的余额，问法和第一个实验相同，只是换成这个编号。公共地址会拒绝，需要付费套餐上的个人端点。查代币则把 `eth_getBalance` 换成 `eth_call`，向合约问 `balanceOf`。

## 长区间要分段

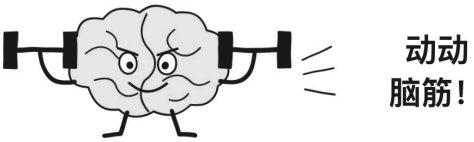
“一整年的所有事件”这种请求，节点多半吃不消：回答太大。把区块区间切成小段，依次询问，再把回答拼起来。

这是一段草稿：`rpc` 是你自己向个人端点提问的函数，类似第23章的 `ask`。

```
async function logsByChunks(rpc, filter, from, to, size = 1000) {
  const all = []
  for (let a = from; a <= to; a += size) {
    const b = Math.min(a + size - 1, to)
    const answer = await rpc('eth_getLogs', [{
      ...filter,
      fromBlock: '0x' + a.toString(16),
      toBlock: '0x' + b.toString(16),
    }])
    if (answer.error) throw new Error(answer.error.message)
    all.push(...answer.result)
  }
  return all
}
```

通不过就把每段切小一些；每一段都是一次单独计费的调用。而店铺最便宜的归档是自己的那份：付款一看到就记进数据库（见第26章）。

### 想一想




1. 节点对一笔很久以前转账的哈希返回 `null`。这能证明顾客没付过款吗?
2. 需要 7 月 1 日的 USDC 余额。Free 套餐够用吗?
3. 要按每段 1000 个区块查看 5000 个区块的事件。要调用几次?

*答案：1) 不能：节点可能没有保存这条记录。2) Free 不含归档，指望不上。3) 五次。*

### 本章要点

- 旧余额、旧事件和追踪，是归档节点的活。
- 归档包含在付费套餐里；端点地址不变。
- `null` 和空回答还不能证明什么都没发生过。

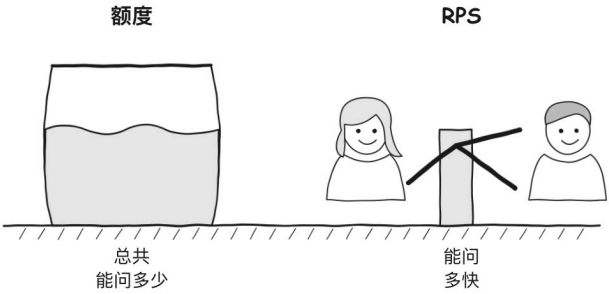
**接下来：**第25章。套餐、调用价格和限制。

# 第25章 套餐、调用价格和限制

## “那一个月要花多少？”

端点能用了。现在阿尔乔姆想知道店铺要花多少钱。定价页上有两个主要数字：额度和每秒请求数。哪个是干什么的？

### 弄明白：两把不同的尺子



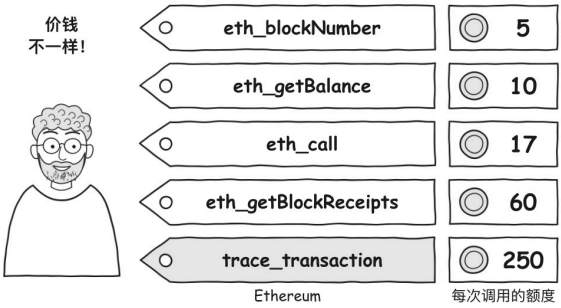

额度是服务内部的“钱”：一个月总共能问多少。RPS 是每秒请求数：能问多快。两样都得够用。

### 动手试试：查调用价格

不同的问题价钱不同：有的节点直接从内存里回答，有的要为它翻遍整个区块。价格是公开的，不用注册就能查。取几个 Ethereum 方法看看（需要 `python3`）：

```
curl -s https://rpcnode.dev/api/v1/public/networks/ethereum/methods |
python3 -c 'import json, sys
want = """eth_blockNumber eth_getBalance eth_call
eth_getLogs eth_getBlockReceipts trace_transaction""".split()
for m in json.load(sys.stdin)["methods"]:
    if m["method"] in want and m["variant"] == "default":
        print(m["method"], m["credits"])'
```

```
eth_blockNumber 5
eth_getBalance 10
eth_call 17
eth_getLogs 17
eth_getBlockReceipts 60
trace_transaction 250
```




- 一次 `trace_transaction` 的价钱相当于问 50 次区块号。
- 每个网络有自己的价目表。把地址里的 `ethereum` 换成 `solana` 或 `tron`，方法名也跟着换。
- 批量请求（见第11章）的价钱是其中各次调用之和。
- WebSocket 订阅也收费。在 Ethereum 上，开通新区块订阅是 10 额度，之后每推送一个事件 2 额度。

控制台里，价格汇总在“网络”栏目。

## 套餐

数据截至 2026 年 11 月 27 日。最新数据见 `rpcnode.dev/pricing`。

| 套餐 | 每月 $ | 每年 $ | 每月额度 | RPS | 端点数 |
|---|---|---|---|---|---|
| Free | 0 | — | 1500 万 | 15 | 3 |
| Starter | 19 | 190 | 5000 万 | 40 | 5 |
| Build | 36 | 360 | 1 亿 | 75 | 12 |
| Grow | 99 | 990 | 3 亿 | 150 | 25 |
| Scale | 219 | 2 190 | 7 亿 | 300 | 50 |
| Business | 449 | 4 490 | 15 亿 | 500 | 100 |

套餐不只是 RPC。里面还有这些（见第26、27章）：

| 套餐 | 监控地址数 | 付款：同时未结 / 每月 | 交易监控数 | 归档和全部网络 |
|---|---|---|---|---|
| Free | — | — | — | 无：只有 Ethereum 和 Solana |
| Starter | 15 | 10 / 200 | 15 | 有 |
| Build | 50 | 25 / 1 000 | 40 | 有 |
| Grow | 150 | 50 / 5 000 | 80 | 有 |
| Scale | 400 | 100 / 20 000 | 150 | 有 |
| Business | 1 000 | 250 / 100 000 | 400 | 有 |

- 新账户可以免费试用一次 Starter，为期 7 天，不需要银行卡。
- 现成套餐不合适的话，定价页上有“自行配置”。
- 买套餐点“购买”并确认。会生成一张有效期 15 分钟的账单，可以用加密货币支付。
- 到期而没有续费，账户就转回 Free。给自己设个提醒。

## 给阿尔乔姆的店算笔账

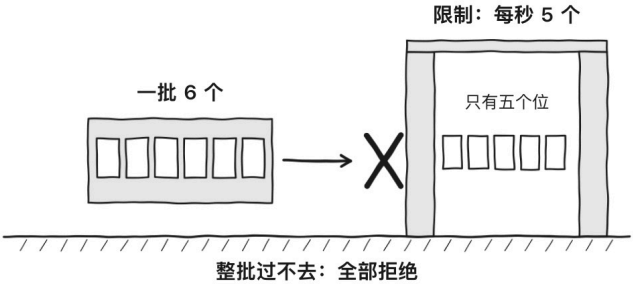
二十个订单在等付款，每个订单有自己的 Ethereum 地址。按 30 天算，并假定总有订单在等：这是上限估计。

| 网站怎么问 | 请求数 | 额度 | 够用的套餐 |
|---|---|---|---|
| 每 10 秒查一次每个地址的余额 | 5 184 000 | 51 840 000 | Build |
| 同上，每 30 秒一次 | 1 728 000 | 17 280 000 | Starter |
| 每 12 秒查一次区块号和一次 `eth_getLogs` | 432 000 | 4 752 000 | Free |

第三种办法比第一种便宜十倍还多：一个问题问遍所有地址。不过这个比较要打个折扣：前两种办法能发现 ETH 到账，第三种发现的是代币转账（见第15章）。而且 Free 只有两个网络：Ethereum 和 Solana。

## 每秒限制

RPS 计数器每秒清零，按 RpcNode 文档，每个端点各有一个。超出的请求收到 HTTP 429 和代码 `-32005`。**含 N 次调用的批量算 N 个请求**，要么整批通过，要么整批不过。




在限制为 5 的公共地址上试一下。发一批六个问题就够了：

```
q='{"jsonrpc":"2.0","id":1,"method":"eth_blockNumber","params":[]}'
curl -s -i https://public-ethereum-mainnet.rpcnode.dev \
  -H 'Content-Type: application/json' \
  -d "[$q,$q,$q,$q,$q,$q]" | grep -iE '^(HTTP|retry-after)|error'
echo
```

```
HTTP/2 429
retry-after: 1
{"jsonrpc":"2.0","id":null,"error":{"code":-32005,"message":
"rps limit exceeded: a batch of 6 calls counts as 6 requests
(the limit is 5 a second)"}}
```

`retry-after: 1` 的意思是“一秒后重试”。公共地址对昂贵的方法也有保护：

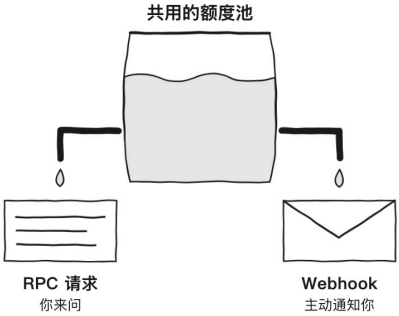
```
curl -s -w '  HTTP %{http_code}\n' https://public-ethereum-mainnet.rpcnode.dev \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,"method":"eth_getBlockReceipts","params":["latest"]}'
```

```
{"jsonrpc":"2.0","id":1,"error":{"code":-32601,"message":
"method eth_getBlockReceipts is not available on the public endpoint,
use a personal endpoint"}}  HTTP 403
```

## 什么扣费，什么不扣

| 发生了什么 | 额度 |
|---|---|
| 节点返回了结果 | 扣 |
| 节点针对你的请求返回错误，例如 `execution reverted` | 扣：这也是回答 |
| 因每秒限制被拒 | 不扣 |
| 方法不在该网络的清单里 | 不扣 |
| 没有任何节点回应 | 退还 |
| 发到公共地址的请求 | 免费 |

## 一个池子管所有




同一份额度还用来支付 Webhook——服务主动发给你网站的消息（见第26章）。在 Ethereum 主网上，一个监控地址的事件值 450 额度，一笔付款值 750。在 Ethereum 测试网上便宜一半。

剩余额度在控制台的“用量”栏目里看，那里还列出最常用的方法。

## 额度用完了

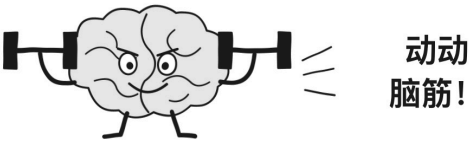
不会有超额账单：套餐之外没有加收。但也别指望服务会像什么事都没有一样继续工作。在程序里为第11章那个拒绝（`-32006`）做好准备，并提前盯着余量。

## 陷阱：看价签挑套餐

先算一算：**调用次数 × 方法价格**。几个重型调用可能比几千个轻量调用吃得还多。

方法价格和条款可能变化。按服务条款，除法律另有要求外，套餐费不予退还，也不保证服务不中断：付费套餐遇到重大故障时，以延长已付费期限来补偿。付款前请读一读。

**想一想**




1. 网站每 10 秒调用一次 `eth_call`（17 额度），整月 30 天不停。一共多少额度?
2. 在 Free 套餐上发了一批 20 次调用。会怎样?
3. 一个请求收到了 429。为它扣额度了吗?

*答案：1) 259 200 × 17 = 4 406 400。2) 整批拒绝：20 大于 15。3) 没有。*

**本章要点**

- 额度限制一个月的总量，RPS 限制速度。
- 每个方法有自己的价格；批量按调用次数计。
- 问得聪明一点，往往比换更大的套餐划算。

**接下来：**第26章。地址事件：用 Webhook 和套接字代替不停轮询。

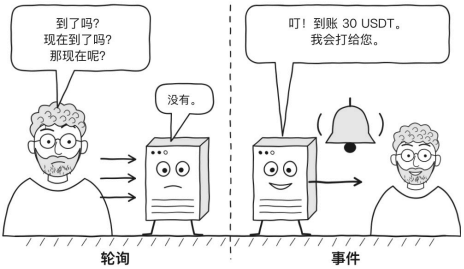
# 第26章 地址事件：用 Webhook 和套接字代替不停轮询

## “二十个订单，每秒二十个问题？”

第3章算过：二十个待付款订单，每个每秒查一次，就已经超过 Free 套餐允许的每秒 15 个请求。而几乎每个回答都一样：“还没到”。

## 弄明白：自己追问，还是等铃响

**轮询**是你的网站一遍又一遍主动问：“到了没？”**事件**是事情一发生，就有人主动告诉你。




## 动手试试：一个问题问遍所有地址

先学着问得聪明一点。不必为每个订单单独提问。第15章的 `eth_getLogs` 方法可以一次列出一串收款人：

```
curl -s https://public-ethereum-mainnet.rpcnode.dev \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,"method":"eth_getLogs","params":[{
  "address":"0xdac17f958d2ee523a2206206994597c13d831ec7",
  "topics":["0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef",
  null,
  ["0x000000000000000000000000000000000000000000000000000000000000dead",
   "0x0000000000000000000000000000000000000000000000000000000000000001",
   "0x000000000000000000000000dac17f958d2ee523a2206206994597c13d831ec7"]],
  "fromBlock":"latest","toBlock":"latest"}]}'
```

问的是：“最新区块里，有没有任何人向这三个地址中任意一个转过 USDT？”地址在左边补零凑到 64 个字符。得到的回答：

```
{"jsonrpc":"2.0","id":1,"result":[]}
```

空列表：这个区块里没人付款。地址可以是二十个，请求仍然只有一个。但是每个新区块都得问一次，确认数要自己数，ETH 本身的转账还要另外捕捉（见第15章）。

## Address Watch：让别人替你盯着

**Address Watch**（地址监控）把这份活接过去。RpcNode 解析区块，一旦有交易涉及你的地址，就通知你的服务器。

地址在控制台里添加：“通知”→“地址”→“添加地址”。程序可以用请求 `POST /v1/address-watches` 做同样的事，请求体是 `{"network_slug": "tron", "address": "你的地址"}`。

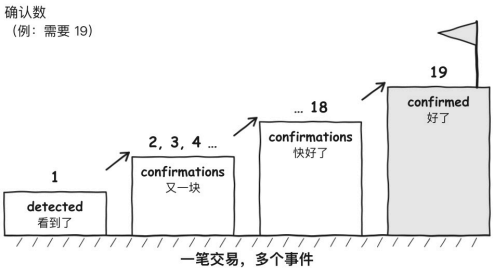
返回的是 `watch_id`——这条监控的编号。一个网络里的一个地址占套餐的一个名额。删掉监控，名额就空出来，但添加和删除本身也有次数限制。Starter 有 15 个名额。Free 在主网上没有名额，只有“沙盒”：测试网里的几个地址。

能开端点的网络比能解析出事件的网络要多。截至 2026 年 11 月 27 日，后者是 Ethereum、Bitcoin、Solana、TRON、BNB Smart Chain、Base、Arbitrum One、TON、Litecoin、Dogecoin、Bitcoin Cash、Dash、Mantle 和 Monad，测试网只有 TRON Nile。

钱进来是 `payment` 类事件，钱出去是 `payout`，其他一切（比如 approve 授权）是 `address`。新区块是 `block` 类，`tx` 和 `payment_order` 见第27章。

## 一笔交易，铃响好几次

一笔转账不是只通知一次，而是随着它逐渐“凝固”（见第8章）分次通知。具体发生了什么，看字段 `event`：




| event | 含义 | 店铺该做什么 |
|---|---|---|
| `detected` | 在区块里看到了交易 | 显示“已看到转账” |
| `confirmations` | 又多了一个确认 | 更新计数 |
| `confirmed` | 确认数足够，执行成功 | 认定钱已到账 |
| `failed` | 确认数足够，交易失败 | 不要发货 |

要等几个确认，在“设置”→“确认数”里配置；不配置则采用该网络的默认值。

在 Bitcoin、Litecoin 和 Dash，最早到达的可能是 `pending`：转账还在内存池里（见第7章）。它的类别永远是 `address` 而不是 `payment`，这还不算钱：这样的转账可能被替换。

## Webhook：投进你信箱的信

Webhook 是服务主动发到你在互联网上某个地址的请求。地址和各网络的订阅在“通知”→“推送设置”里配置。地址必须能从互联网访问：`localhost` 不行。

如果只启用 Webhook 而什么都不勾选，会收到所有网络的转入和转出，但不包括区块。

下面是文档示例的一部分——关于 TRON 上 30 USDT 的第一声铃响：

```
{ "delivery_id": "3f9c…6b3d90",
  "type": "payment", "event": "detected", "network": "tron",
  "tx_hash": "bb08…c320c", "address": "TToL…QRzRG",
  "direction": "in", "currency": "USDT",
  "amount": "30", "amount_raw": "30000000",
  "confirmations": 1, "required_confirmations": 19,
  "confirmed": false }
```

金额以字符串形式给出，避免任何舍入；`amount_raw` 是同一个数但不带小数点（见第9章）。服务目录里没有的代币只会带 `amount_raw`。请求头里有 `X-RpcNode-Delivery`（推送编号）、`X-RpcNode-Event`（类别）和 `X-RpcNode-Signature`（签名）。

### “真的是你们吗？”

Webhook 地址是敞开的：谁都能来敲门，写一句“有人给你付钱了”。所以每次推送都带签名。把时间、一个点和请求体连起来，用 Webhook 的密钥串算出指纹。这种方法叫 HMAC-SHA256。密钥串在“密钥串和 IP”区块里；它不是 API 密钥的密钥串。

### 动手试试：自己签一次

这里的密钥串和请求体是编的：

```
printf '%s' '1787347206.{"type":"test"}' |
  openssl dgst -sha256 -hmac 'demo-secret' | awk '{print $NF}'
```

```
e17efb57778b0e0c758a9bf2c5705049c4c383d2af3a2bd2783c8b665c151cde
```

请求体或密钥串里改一个字符，结果就是完全不同的一串。你的服务器算同样的东西，再和形如 `t=…,v1=…` 的请求头比对。

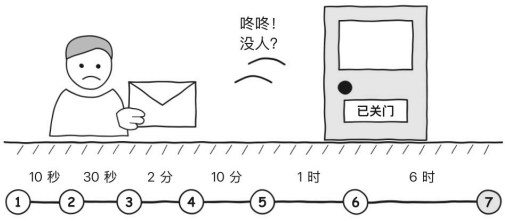
下面是一个 Node.js 接收端的草稿。我们用自己造的请求测过它，但没有用真实推送测过：那需要个人账户。300 秒的时间检查可以防止有人把昨天的推送再发一遍。

```
import { createServer } from 'node:http'
import { createHmac, timingSafeEqual } from 'node:crypto'

const SECRET = process.env.RPCNODE_WEBHOOK_SECRET
const seen = new Set()      // 真实店铺里这应是数据库
createServer(async (req, res) => {
  const parts = []
  for await (const part of req) parts.push(part)
  const body = Buffer.concat(parts)   // 原样接收的字节
  const [, t, v1] = /^t=(\d+),v1=([0-9a-f]{64})$/
    .exec(req.headers['x-rpcnode-signature'] ?? '') ?? []
  const ours = createHmac('sha256', SECRET)
    .update(`${t}.`).update(body).digest()
  const ok = v1 && Math.abs(Date.now() / 1000 - t) < 300 &&
    timingSafeEqual(ours, Buffer.from(v1, 'hex'))
  if (!ok) return res.writeHead(401).end()
  const id = req.headers['x-rpcnode-delivery']
  if (!seen.has(id)) {
    seen.add(id)
    console.log(id, body.toString())  // 在这里写你的业务
  }
  res.writeHead(200).end()
}).listen(3000)
```

## 如果你不在家

服务器在 10 秒内返回 2xx 状态码，推送才算成功。否则会重发，第七次失败后放弃。所以：

- 同一事件可能到达两次。推送编号相同——就按它去重。
- 不保证顺序：`confirmations` 可能跑到 `detected` 前面。
- 耗时的活放在回应之后做。



间隔越来越长；共七次，约七小时

事先用测试推送检查一下你的地址：测试推送免费。控制台里的推送日志保存一天。

## 套接字：一条打开的线路

第二种方式是 WebSocket：由你的程序主动建立连接并收听，不需要自己在互联网上有个地址。它不是第11章的 `eth_subscribe` 订阅：那里是节点在讲网络的事，这里是服务在讲你的地址和订单。消息体和 Webhook 一样，但你不在期间错过的不会重发。

协议与 Pusher 兼容（可以用 `pusher-js` 库），key 是你的 API 密钥编号。套接字要单独打开，开关是“推送到 WebSocket”。两种方式都开着，每条消息就推送两次、收费两次。

## 要花多少钱

事件和请求花的是同一份额度。截至 2026 年 11 月 27 日，TRON 或 Ethereum 主网上的一个地址事件值 450 额度——**每笔交易只收一次**：确认通知和最终结果免费。费用按推送收取，哪怕你的服务器后来没接收成功。

估算一下 Ethereum 上 30 天的一个月，表中每个请求 17 额度：

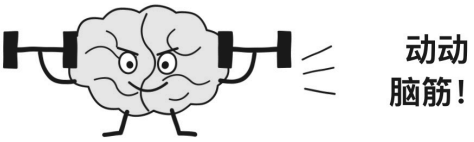
| 靠什么得知付款 | 请求数或事件数 | 额度 |
|---|---|---|
| 20 个订单，每 10 秒一次 `eth_call`（见第13章） | 5 184 000 | 88 128 000 |
| 每个区块一次 `eth_getLogs`，12 秒一次 | 216 000 | 3 672 000 |
| Address Watch，一个月 200 笔转账 | 200 | 90 000 |

Starter 有 5000 万额度：第一种办法装不下。这只是个例子。转账越多，事件费用越高，别人往你地址发的零碎小钱也算在内。

## 陷阱：“来了事件就发货”

- **不是每个事件都是付款**：`detected` 还没成定局，`payout` 是你的支出。
- **认代币要看合约地址**，也就是 `token` 字段里的地址，而不是名字（见第9章）。
- **特殊的转账要事先测试**，例如从合约内部发来的 ETH（见第15章）。
- **地址事件不是档案库**：通过 API 只能查一天之内的。

**想一想**




1. 店铺服务器停了一个小时。Webhook 推送会怎样，套接字消息又会怎样？
2. 收到两次 `X-RpcNode-Delivery` 相同的推送。钱记几次账？
3. 一个月里监控地址在 TRON 上有 1000 笔转账，只用 Webhook 推送。多少额度？

*答案：1) Webhook 会重发，直到七次机会用完；套接字错过的不会重发。2) 一次。3) 450 000。*

**本章要点**

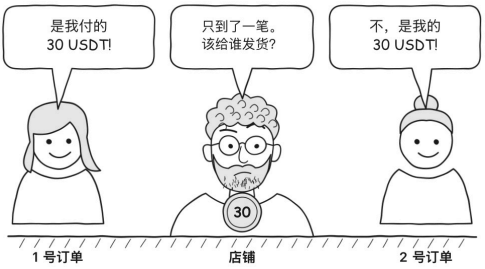
- 别没完没了地问：让转账主动通知你。
- 验签名，去重复，快回应。
- 钱到账是在 `confirmed`，而不是第一声铃响。

接下来：第27章。付款和交易跟踪。

# 第27章 付款和交易跟踪

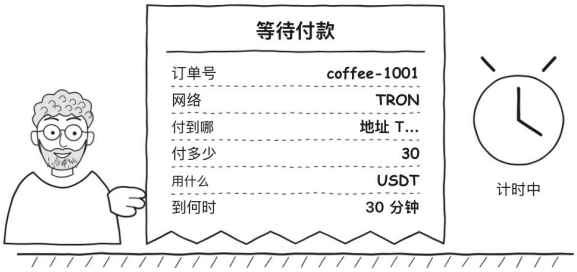
## “钱到了。可这是谁的？”

阿尔乔姆收到一个事件：地址上进了 30 USDT。打开订单列表，里面有两个三十的订单。事件知道地址和金额，却对顾客的购物车一无所知。




## 弄明白：付款就是一份“预期”

付款（控制台里的“付款”栏目）是事先记下的一份预期：某订单在某时刻之前，等着某地址收到某金额。合适的转账一到，服务直接告诉你订单号。




地址由你提供。钱从顾客直接到你手里，服务不需要助记词（见第6章）。

创建付款可以点按钮“新建付款”，或发请求 `POST /v1/payments`：

| 字段 | 是什么 |
| --- | --- |
| `order_id` | 你的订单号，不得重复 |
| `network_slug`、`address` | 网络，以及你在该网络的地址 |
| `amount` | 金额，写成字符串：`"30"`、`"12.5"` |
| `token_contract` | 代币合约；不填则等网络原生币 |
| `ttl_seconds` | 以秒计的期限；不填为一小时，最长 30 天 |

## 动手试试：不带密钥敲门

```
curl https://api.rpcnode.dev/v1/payments
```

```
{"ok":false,"error_code":"public_api.key_missing",
 "error":"Missing X-Api-Key header."}
```

这是状态码 401 的回答：“你是谁？”发给这个 API 的每个请求都要用 API 密钥的密钥串签名（“API 密钥”栏目）。签名内容是四行：时间、方法、路径和请求体的指纹。

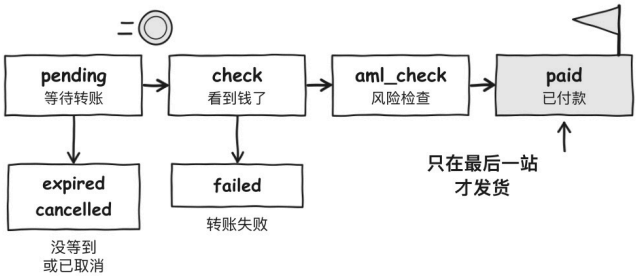
下面是一段 shell 写的草稿。我们检查过它的语法和签名计算，但没有实际发送：那需要个人密钥。

```
BODY='{"order_id":"coffee-1001","network_slug":"tron",
"address":"你的地址","amount":"30","ttl_seconds":1800,
"token_contract":"TR7NHqjeKQxGTCi8q8ZY4pL8otSzgjLj6t"}'
TS=$(date +%s)
HASH=$(printf '%s' "$BODY" | openssl dgst -sha256 | awk '{print $NF}')
SIG=$(printf '%s\nPOST\n/v1/payments\n%s' "$TS" "$HASH" |
  openssl dgst -sha256 -hmac "$SECRET" | awk '{print $NF}')

curl https://api.rpcnode.dev/v1/payments \
  -H 'Content-Type: application/json' \
  -H "X-Api-Key: $KEY_ID" -H "X-Timestamp: $TS" \
  -H "X-Signature: $SIG" -d "$BODY"
```

返回内容里有 `payment_id`、状态 `pending` 和截止时间 `expires_at`。用同一个 `order_id` 并且网络、地址、代币和金额都相同地重复请求，返回的是同一笔付款。内容不同则报错 `payment.order_exists`。

## 一笔付款的路程

每一步都以 `payment_order` 事件送达——经 Webhook 或套接字（见第26章）。




| `event` | 发生了什么 |
|---|---|
| `check` | 看到钱了，等确认 |
| `confirmations` | 确认数增加了 |
| `aml_check` | 确认数够了，正在做风险检查 |
| `paid` | 已付款：可以发货 |
| `expired`、`cancelled` | 过期了，或你取消了等待中的付款 |
| `failed` | 链上交易失败（不要把一切都建在这个状态上：付款也可能直接过期） |

AML（反洗钱检查）是核查收到的钱是否和不良记录有牵连。在支持它的网络上，`paid` 事件里的 `aml` 对象带有 0 到 100 的评分 `score`（越高越危险）和等级 `risk_level`。不另外收费。并非所有网络都有：没有的网络，付款直接变为 `paid`。

这只是提示，不是判决，也不是法律意见：有风险的转账怎么处理，由你决定。付款卡在 `aml_check`，还不算付款成功，请联系支持团队。

文档示例的一部分：

```
{ "type": "payment_order", "event": "paid", "order_id": "ord_123",
  "network": "tron", "status": "paid", "tx_hash": "bb08…c320c",
  "received_amount_raw": "30000000",
  "confirmations": 19, "required_confirmations": 19,
  "aml": { "score": 0, "risk_level": "low" } }
```

## 小字条款

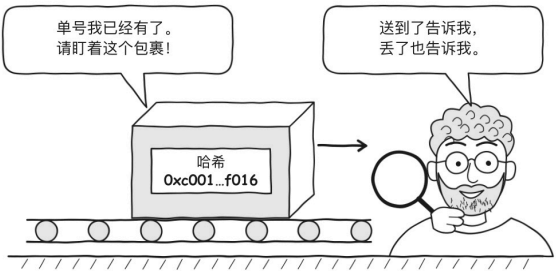
- **少付不算数。**付款继续等待，直到过期。文档里有 `underpaid` 状态，但不要靠它来搭店铺：少付要通过地址事件去发现。
- **多付算数。**实际到账多少看 `received_amount_raw`。
- **迟到的转账不算付款。**而在期限内被看到的转账，过了期限仍可从容积累确认。

- **每个订单配一个单独地址**。那样两个“三十”就不会混淆（见第28章）。
- **地址名额不会自动释放**：`paid` 或 `expired` 之后，请删除监控。取消付款时，专为它添加的地址会自动撤下。

付款功能在付费套餐里：Starter 可同时有 10 笔未结、每月 200 笔。TRON 或 Ethereum 主网上的一笔付款值 750 额度，只有看到钱时才扣。Free 有沙盒：测试网上 3 笔未结付款，目前是 TRON Nile。那里的风险检查是假的，带有 `"sample": true` 标记。

## 交易跟踪

也有反过来的情况：哈希已经在你手里。阿尔乔姆给顾客退了款，或给供货商付了钱，想知道到了没有。




“通知”→“交易”→“添加哈希”，或者 `POST /v1/tx-watches`，请求体 `{"network_slug": "tron", "tx_hash": "哈希"}`。哈希可以在它出现在链上之前就添加。

收到的是 `tx` 事件：`detected`，然后是 `confirmations`，最后是 `confirmed` 或 `failed`。如果哈希始终没找到，则是 `cancelled`。前提是这个网络启用了转入或转出通知。

进行中的跟踪占一个名额，结束后释放。Starter 有 15 个名额，Free 没有这项服务。每笔交易收一次费：在 TRON 和 Ethereum 是 150 额度。

## 动手试试：自己数确认数

任何这类跟踪的幕后做的就是下面这件事。取第15章那笔 USDT 转账：

```
URL=https://public-ethereum-mainnet.rpcnode.dev
H=0xc0015ff5178d72a68831b133187c3f9f1fbee2e440d3af311d0b32110e2cf016
curl -s $URL -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,"method":"eth_getTransactionReceipt",
  "params":["'$H'"]}' | grep -oE '"(status|blockNumber)":"0x[0-9a-f]+"' | sort -u
sleep 1
curl -s $URL -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":2,"method":"eth_blockNumber","params":[]}'
```

```
"blockNumber":"0x194646a"
"status":"0x1"
{"jsonrpc":"2.0","id":2,"result":"0x19464c5"}
```

状态 `0x1` 是成功。包含这笔转账的区块算第一个确认，所以：`echo $((0x19464c5 - 0x194646a + 1))` 得到 92。你算出来会更大。

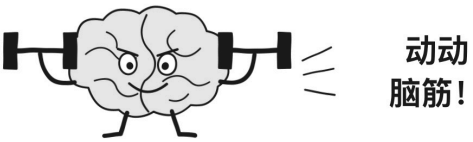
### 陷阱：同一笔钱，铃响两次

创建付款会把地址纳入监控。因此同一笔转账可能同时发来 `payment`（地址事件）和 `payment_order`（付款事件）。两者各按各的价收费。

> 规则：只有 `event` 为 `paid` 的 `payment_order` 才能结单。并且核对订单号、`payment_id`、代币和金额。

也别没完没了地查询付款：API 有自己的限制，每个账户每分钟 100 个请求。停机之后需要对账时再查：`GET /v1/payments/orders/{orderId}`。

**想一想**




1. 付款要 30 USDT，到了 29.5。这笔付款怎么样了？
2. 期限过了，过后一分钟才看到转账。钱在哪里，状态如何？
3. 收到 `check` 事件。可以发咖啡了吗？

*答案：1）没有付清，到期失效。2）钱在你的地址上，付款保持 `expired`：要人工处理。3）不可以，等 `paid`。*

**本章要点**

- 付款把转账和订单连起来，哈希跟踪把你和交易的结局连起来。
- 凭 `paid` 发货，而不是凭第一声铃响。
- 少付、迟到和被占的名额，都得你自己操心。

**第五部分到此结束。**阿尔乔姆会开端点了，明白了请求的路径和套餐，也能不靠轮询就知道钱的消息。

**接下来**：第六部分，第28章。搭建店铺：账单、地址、等待付款、确认。

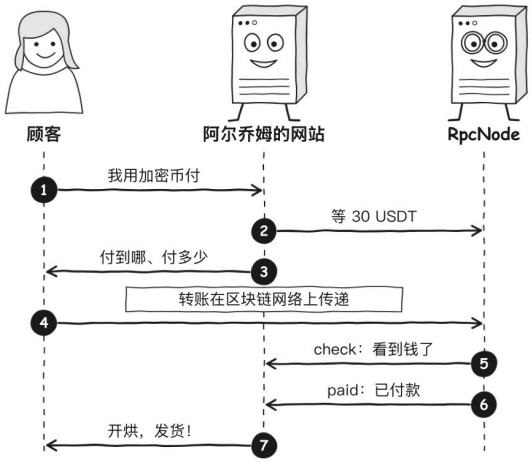
# 第28章 搭建店铺：账单、地址、等待付款、确认

## “分开看都懂了。合起来呢？”

阿尔乔姆知道了地址、确认数、Webhook 和付款。现在顾客点下“用加密货币支付”。接下来网站该一步步做什么？

### 弄明白：都有谁参与

参与者有四个。带着钱包的顾客。有订单数据库的店铺网站。盯着网络的 RpcNode。还有存放密钥的阿尔乔姆的钱包。网站和服务之间的对话，钱包并不参与。

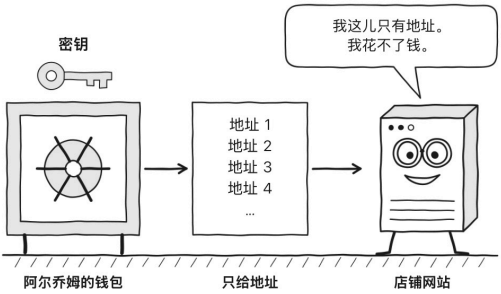



看第 4 步。钱既不到网站，也不到 RpcNode。它直接落在阿尔乔姆的地址上，RpcNode 只是在区块里发现了它。

### 服务器上没有密钥，地址从哪来

第6章许诺过：每个订单一个地址。那一章也说了：私钥不该放在店铺服务器上。这两点怎么兼顾？

地址可以和密钥分开准备。




办法有两种。

| | 地址清单 | 扩展公钥 |
|---|---|---|
| 服务器上放什么 | 现成的地址，每单用一个 | 一个密钥（常以 `xpub` 开头），程序用它自己算出地址 |
| 适用范围 | 任何网络 | Bitcoin、Ethereum、TRON 及亲戚；Solana 和 TON 行不通 |
| 如果服务器被攻破 | 清单里的地址被看到 | 你的全部地址和这些地址的全部收款都被看到 |
| 麻烦之处 | 要及时补充 | 配置一次即可 |

凭这种公钥花不了钱。但 BIP-32 标准提醒：这个公钥**再加上任意一个**同一分支里的私钥，就能算出该分支的全部私钥。所以不要把这种钱包的私钥单个导出。新手从清单起步更省事。

清单要从密钥不放在联网服务器上的钱包里导出。**硬件钱包**很方便：它是一个把密钥保存在自身内部的独立设备。上传之前，拿几个地址和设备屏幕核对一下。在 TON 上，给顾客展示 `UQ` 写法的地址（见第19章）。

## 动手试试：让数据库不许弄混

分发地址是数据库的活。下面是个玩具例子：两个空闲地址、三个订单。需要 `sqlite3` 程序，很多系统自带。

```
sqlite3 :memory: <<'SQL'
CREATE TABLE addresses(address TEXT PRIMARY KEY, order_id TEXT UNIQUE);
INSERT INTO addresses(address) VALUES ('地址-1'), ('地址-2');

UPDATE addresses SET order_id = 'coffee-1001' WHERE address =
  (SELECT address FROM addresses WHERE order_id IS NULL ORDER BY address LIMIT 1)
  RETURNING order_id, address;
UPDATE addresses SET order_id = 'coffee-1002' WHERE address =
  (SELECT address FROM addresses WHERE order_id IS NULL ORDER BY address LIMIT 1)
  RETURNING order_id, address;
UPDATE addresses SET order_id = 'coffee-1003' WHERE address =
  (SELECT address FROM addresses WHERE order_id IS NULL ORDER BY address LIMIT 1)
  RETURNING order_id, address;
SELECT '第三个订单没分到地址' WHERE changes() = 0;

UPDATE addresses SET order_id = 'coffee-1001' WHERE address = '地址-2';
SQL
```

```
coffee-1001|地址-1
coffee-1002|地址-2
第三个订单没分到地址
Runtime error near line 15: UNIQUE constraint failed: addresses.order_id (19)
```

第三张账单开不出来：清单该补充了。最后一条命令想给订单 1001 再配第二个地址，数据库拒绝了。守着这一关的是 `PRIMARY KEY` 和 `UNIQUE`：每个地址只记一次，每个订单最多一个地址。

还有，旧地址不要轮第二圈。上一个订单的迟到转账会把新订单“付清”。

## 开账单




顾客需要五样东西：网络、币种、准确金额、地址和期限。还有一个请求：全部金额一笔转清，因为两个一半不会合计。要提醒他：如果钱包或交易所从金额里扣手续费，到账就会少。

以卢布或美元标价的，要在下单时换算成币，并在期限内锁住汇率。所以期限定得短，比如半小时。

接下来需要第27章的“付款”。下面是一段 Python 草稿。其中的函数 `rpcnode` 负责发送签名请求：照第27章自己写，或者用 RpcNode 官方客户端（有 Python、TypeScript、PHP、Go 和 Kotlin 版）。数据库还是那个，两张表。

```
USDT = 'TR7NHqjeKQxGTCi8q8ZY4pL8otSzgjLj6t'    # TRON 上的 USDT 合约
# addresses(address, order_id)
# orders(id, payment_id, amount_raw, status — 初始为 'waiting')

def take_address(db, order_id):
    with db:                                    # 一个数据库事务
        db.execute('INSERT OR IGNORE INTO orders(id) VALUES (?)',
                   (order_id,))
        mine = db.execute('SELECT address FROM addresses'
                          ' WHERE order_id=?', (order_id,)).fetchone()
        if mine:
            return mine[0]                      # 重复调用——还是那个地址
        free = db.execute(
            'UPDATE addresses SET order_id=? WHERE address='
            ' (SELECT address FROM addresses WHERE order_id IS NULL'
            '  ORDER BY address LIMIT 1) RETURNING address',
            (order_id,)).fetchall()
        if not free:
            raise RuntimeError('地址用完了')
        return free[0][0]
```

```
def checkout(db, rpcnode, order_id, amount):   # amount 是字符串
    address = take_address(db, order_id)
    reply = rpcnode('POST', '/v1/payments', {
        'order_id': order_id, 'network_slug': 'tron',
        'address': address, 'amount': amount,
        'token_contract': USDT, 'ttl_seconds': 1800})
    if not reply['ok']:
        raise RuntimeError(reply['error_code'])
    payment = reply['payment']
    with db:
        db.execute('UPDATE orders SET payment_id=?, amount_raw=?'
                   ' WHERE id=?', (payment['payment_id'],
                   payment['amount_raw'], order_id))
    return payment        # 给顾客：address, amount, expires_at
```

这里的关键是顺序：地址在向服务发请求之前就绑定到订单上。连接断了，重复调用会拿到同一个地址和订单号，服务返回同一笔付款。我们用一个虚构的 `rpcnode` 函数验证过；没有拿这段草稿对接真实服务运行过。

在顾客选定用加密货币支付时才创建付款，而不是每个购物车都建：套餐的月度限额按创建的付款数计算，没付的也算。

RpcNode 凭地址、币种和金额识别付款。第19章的备注和第18章的 `reference` 标记在这里不起作用：付款没有这样的字段。第20章的 XRP Ledger 和 Stellar 不在支持付款的网络清单里（见第26章）。

## 等待付款

订单页面每隔几秒向**你的服务器**询问，服务器从自己的数据库回答。数据库由 Webhook 更新。不能从浏览器直接问 RpcNode：密钥串会落到顾客手里，而且 API 限制是每个账户每分钟 100 个请求。

| 状态 | 顾客看到什么 | 店铺做什么 |
|---|---|---|
| `pending` | “等待转账，截止 14:30” | 什么都不做 |
| `check` | “已看到转账。确认数 3/19” | 不发货 |
| `aml_check` | “正在核查转账” | 等待 |
| `paid` | “已付款，正在备货” | 发货 |
| `expired` | “已过期。请不要再向本账单转账” | 关闭账单 |
| `failed` | “转账未成功” | 请顾客重新来过 |
| `cancelled` | “账单已取消” | 关闭账单 |

转账没进区块之前，付款一声不吭。所以要写“等待中”，而不是“未付款”。

### 等几个确认

这在“设置”→“确认数”里配置，每个网络单独设。没设就用网络默认值。需要的数目在付款的 `required_confirmations` 字段里能看到。

设得比默认值少是有风险的：新区块有时会被替换（见第8章）。而且这个数是按网络设的，不是按订单。想让一袋咖啡豆比一杯咖啡等得久——在 `paid` 之后再多等一阵，像第27章那样自己查。

### 腾出名额

名额有两种。付款名额在付款变为 `paid`、`expired`、`failed` 或 `cancelled` 时自动释放。地址名额不会自动释放（取消除外，见第27章）。

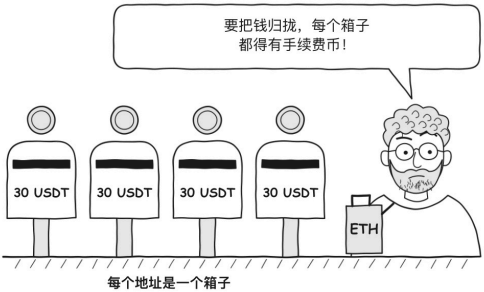
在 Starter 上是 10 笔未结付款、15 个监控地址、每月 200 笔付款。第十一笔未结付款会被拒绝：`payment.slot_limit`。地址名额全占满时，返回 `watch.address_limit`。

已关闭的订单让它先放一阵，万一转账迟到呢。然后撤掉监控：用请求 `GET /v1/address-watches?network_slug=tron&address=…` 找到它的 `watch_id`，再发 `DELETE /v1/address-watches/{watch_id}`。此后不再有这个地址的消息，尽管钱仍然可能打到上面。

添加和删除地址也有限制：按天、按周、按月，名额越多，余量越大。每个订单用一个新地址会消耗这份余量。大促当天，即使还有空名额，也可能收到 `watch.quota`。预计订单很多的话，事先弄清上限。

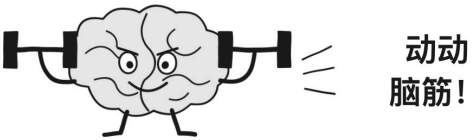
### 陷阱：钱散在一百个地址上

每单一个地址便于记账，可钱之后是分开放着的。要把 USDT 从某个地址转出去，该地址上还得有付手续费的币（见第9、17章）：Ethereum 上是 ETH，TRON 上是 TRX 或能量。一百个订单就是一百笔单独的转账。在 Bitcoin，许多零碎“钱块”同样让发送变贵（见第14章）。在为小额订单选网络之前，先把这笔账算清。




还有一点。按 BIP-44 标准的钱包连续遇到 20 个空地址后就停止查找。被放弃的订单一多，排在远处地址上的付款，钱包可能显示不出来。钱没丢，但要确认你的钱包能不能往后继续找。

### 想一想




1. 顾客刷新了付款页面。需要新地址、新付款吗?
2. 阿尔乔姆在 Starter 上已经有 10 笔未结付款。第十一位顾客来了。会怎样?
3. 付款页面从浏览器直接向 RpcNode 查询。说出两个麻烦。

*答案：1) 不需要：同一个订单号、同一个地址。2) 被拒 `payment.slot_limit`：请顾客稍等，或提供别的付款方式。3) 密钥串被顾客看到，请求限额也很快用完。*

### 本章要点

- 密钥留在钱包里，网站上只放地址：一单一地址，不重复使用。
- 顾客问你的网站，网站从 Webhook 得知消息。
- 名额和归集资金的开销要事先算好。

**接下来：**第29章。事情不顺的时候：错误、重试、卡住的交易、重复入账。

# 第29章 事情不顺的时候：错误、重试、卡住的交易、重复入账

## “咖啡发了两次！”

早上阿尔乔姆发现：一个订单发出了两个包裹。另一位顾客来信说：“我付了钱，你们那边还一直显示‘等待中’。”这不是什么怪事。程序只按晴天来写，就会这样。

## 弄明白：三类麻烦

| 麻烦 | 例子 | 药方 |
|---|---|---|
| 你的请求没得到回应 | 创建付款时连接断了 | 无害地重试 |
| 消息来得不对 | 来两次、顺序乱，或根本没来 | 数据库里加锁，加上对账 |
| 转账出了岔子 | 卡住、迟到、少付 | 定规则，人工处理 |

## 什么时候重发 API 请求

递增间隔第23章讲过了。但能放心重试的只有幂等的操作：做多少次，结果都和做一次一样。创建付款靠 `order_id` 做到了这一点。用同一个订单号、同样的数据重试。每次重试都要重新签名：签名里有时间。

RpcNode 官方客户端在断线和 5xx 之后不会自己重发 `POST`。所以重试创建付款是你的事。

API 的各种回应该怎么处理：

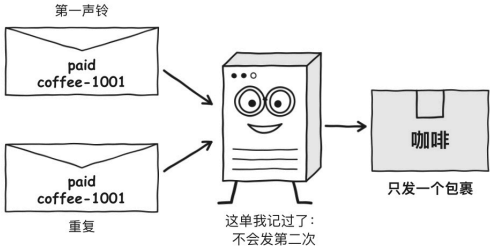
| 收到什么 | 怎么办 |
|---|---|
| 什么都没有；502、503、504 | 带间隔地重试 |
| 429，带响应头 `Retry-After` | 按它说的时间等待 |
| 429 `watch.quota`（见第28章） | 马上重试没用：余量用完了 |
| 400 | 请求写错了：`errors` 里指出了字段 |
| 401 | 检查密钥、签名和服务器时钟 |
| 403 | 套餐、名额限制或 IP 白名单 |
| 409 `payment.order_exists` | 这个订单号已被数据不同的付款占用：要查清楚，而不是随手换个号 |

付款过期而顾客想再试一次，则是另一回事。旧的 `order_id` 返回的还是那笔过期付款。需要一个新号，比如 `coffee-1001-2`。

## 一条消息，来了两次

重复消息的来源五花八门：

- 因为你 10 秒内没回应，Webhook 重发了；
- Webhook 和套接字都开着；
- 同一笔钱同时来了 `payment` 和 `payment_order`（见第27章）；
- Webhook 和你的定时检查同时触发；
- 员工手动点了“发货”。




推送编号（见第26章）能滤掉同一次推送的重复。但同一笔钱经另一条路发来的第二条消息，它挡不住。最要紧的一把锁是订单状态：从“等待”变为“已付款”只能发生一次。

## 动手试试：同一个 Webhook 来两次

下面是这把锁的全部。消息是我们自己编的。把两段代码存进同一个文件 `once.py`，运行 `python3 once.py`。

```
import sqlite3
db = sqlite3.connect(':memory:')
db.executescript("""
CREATE TABLE orders(id TEXT PRIMARY KEY, payment_id TEXT,
                    amount_raw TEXT, status TEXT);
CREATE TABLE to_ship(order_id TEXT PRIMARY KEY);
INSERT INTO orders VALUES('coffee-1001','p1','30000000','waiting');
""")

def on_news(n):
    if n['type'] != 'payment_order' or n['event'] != 'paid':
        return '这还不是付款'
    with db:                # 里面的操作是一个数据库事务
        cur = db.execute(
            "UPDATE orders SET status='paid' WHERE id=? AND"
            " payment_id=? AND amount_raw=? AND status='waiting'",
            (n['order_id'], n['payment_id'], n['amount_raw']))
        if cur.rowcount != 1:
            return '已记过账，或不是我们的'
        db.execute('INSERT INTO to_ship VALUES (?)', (n['order_id'],))
    return '发货'
```

接着是连续四条消息和程序的回答：

```
paid = {'type': 'payment_order', 'event': 'paid',
        'order_id': 'coffee-1001', 'payment_id': 'p1',
        'amount_raw': '30000000'}
print(1, on_news({**paid, 'event': 'check'}))
print(2, on_news(paid))
print(3, on_news(paid))                          # 同一个 Webhook 又来一次
print(4, on_news({**paid, 'event': 'check'}))  # 迟到的消息
print('包裹数:', db.execute('SELECT count(*) FROM to_ship').fetchone()[0])
```

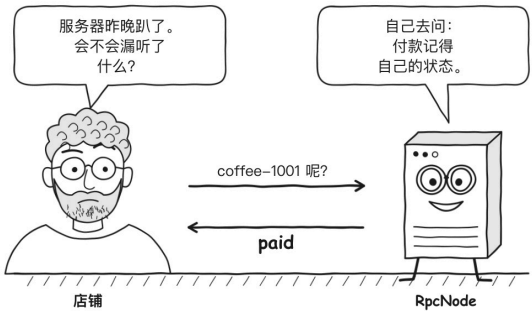
```
1 这还不是付款
2 发货
3 已记过账，或不是我们的
4 这还不是付款
包裹数: 1
```

包裹只有一个。一切都系在一条命令上：“标记为 `paid`，仅当订单还在等待，且付款和金额都对得上”。标记和发货任务一起写入：要么都写，要么都不写。这叫**数据库事务**（英文里它和区块链的“交易”是同一个词 transaction，指的却是另一回事）。我们还试过八条相同消息同时涌入：发货任务只有一个。

第26章的草稿里，推送编号放在内存中，而且在干活之前就记下了。涉及钱的事不能这样：程序中途崩溃，一切就乱套。先写进数据库，再回 200。

## 消息压根没来

Webhook 试过七次就放弃了，大约历时七小时。套接字错过的则完全不补发。所以有些消息可能丢失。药方是**对账**：对那些在你这边仍未结的订单，自己去查。




请求 `GET /v1/payments/orders/{orderId}` 返回付款信息。那里的名称略有不同：不看 `event` 而看 `status`，网络叫 `network_slug`，代币合约叫 `token_contract`。用回答拼出一条同样格式的“消息”，让它过同一把锁。

服务器停机之后、套接字重连之后，以及按定时计划，都要对账。十个未结订单每分钟查一次，是 100 个允许请求中的 10 个。

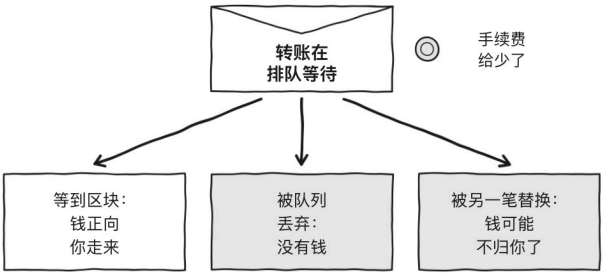
## 转账卡住了

顾客发来哈希："一小时前就发了。"付款毫无动静。我们问节点认不认识这笔交易。这里的哈希是编的：

```
curl -s https://public-ethereum-mainnet.rpcnode.dev \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,"method":"eth_getTransactionByHash","params":
  ["0x0000000000000000000000000000000000000000000000000000000000000001"]}'
```

```
{"jsonrpc":"2.0","id":1,"result":null}
```

`null` 的意思是"这个节点不认识它"。原因有多种：打错了、发错了网络、转账还没传到，或者已经被丢掉了。如果交易找到了，但它的 `blockNumber` 是 `null`，说明它在排队。




**转账没进区块，钱就还不是你的**

| 网络 | 久等不进的交易会怎样 |
|---|---|
| Ethereum 及类似网络 | 挡住发送方后面所有转账（见第15章）。可以用 `nonce` 相同、手续费更高的转账替换它：节点程序 geth 默认要求至少高 10% |
| Bitcoin | 发送方可以用手续费更高的转账替换它。Bitcoin Core 程序默认把在队列里躺了两周的交易丢掉 |
| TRON | 交易有有效期，默认一分钟。没赶上区块，就再也进不去了 |
| Solana | 普通交易在它引用的区块还"新鲜"时有效：150 个区块，也就是区区几十秒 |

别人的转账，店铺一般没法加速：需要发送方的密钥。等它进区块，不要发货。如果付款在此期间过期、转账最终又成功了，钱会落到你的地址，但付款仍是 `expired`。

卡住的是你自己发出的转账，比如退款呢？同一笔签好名的转账可以再发一次：哈希不会变。而一笔新转账就是第二笔钱，除非它是用来替换第一笔的。在 Ethereum 上，一个地址有没有排队的交易，可以这样看：

```
URL=https://public-ethereum-mainnet.rpcnode.dev
A=0x8f8d1206d1bce12ff892731f8a140ed18b93884e
for tag in latest pending; do
  curl -s $URL -H 'Content-Type: application/json' -d '{"jsonrpc":"2.0",
    "id":1,"method":"eth_getTransactionCount","params":["'$A'","'$tag'"]}'
  echo; sleep 2
done
```

```
{"jsonrpc":"2.0","id":1,"result":"0x67641"}
{"jsonrpc":"2.0","id":1,"result":"0x67641"}
```

第一个数是该地址已经进区块的转账笔数，第二个数连排队的也算上。两者相等：这个节点没看到队列。这是第15章那位发送方的地址。

用哈希跟踪（见第27章）时要小心：回答 `cancelled` 表示服务没找到这笔交易，而不是它永远不会再出现。

## 钱到了，但不对劲

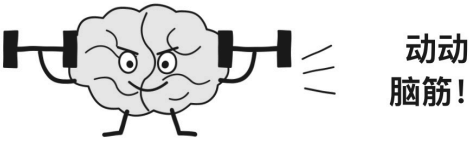
| 发生了什么 | 付款会怎样 |
|---|---|
| 到账偏少 | 未付清，到期失效；钱已经在你手里 |
| 分两笔付 | 不会合计：结果同上 |
| 到账偏多 | `paid`；实际金额看 `received_amount_raw` |
| `paid` 之后又发来一笔 | 付款已不再等待：顶多能从地址事件得知 |
| 过期之后才到 | `expired` |
| 代币或网络不对 | 不记账（见第16章） |
| 交易失败 | 钱没到，不要发货：付款会变为 `failed` 或直接过期 |

每种情况怎么办要事先定好，并在付款页面上告诉顾客。

## 陷阱：“从哪来，退回哪去”

顾客多付了，阿尔乔姆想把多余部分退到发送方地址。可那可能是交易所的公共地址：钱会归交易所，而不是顾客。要问顾客退款用的地址和网络。亲手从钱包发送，不要从网站发。

### 想一想




1. 创建付款的回应没等到。能重试吗？用哪个 `order_id`？
2. 同一个 `paid` 经 Webhook 和套接字各来一次。示例程序会发几个包裹？
3. 应付 30 USDT，对方分两笔各 15 发来。那笔 30 的付款怎样了？

*答案：1) 能，用同一个订单号和同样的数据。2) 一个。3) 没有付清：多笔转账不合计。*

### 本章要点

- 只重试可以安全重试的操作。
- 发货由数据库里的锁来把守，丢失的消息靠对账找回。
- 转账没进区块，钱就不算到；奇怪的付款要事先立好规则。

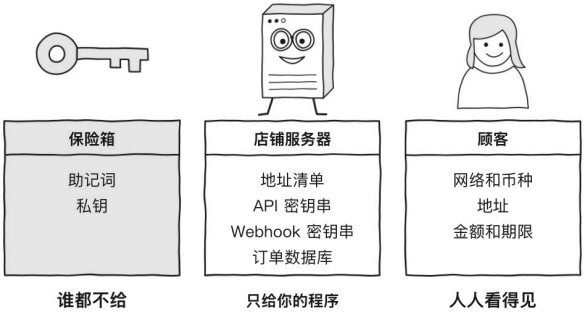
**接下来：**第30章。安全与上线前检查清单。

# 第30章 安全与上线前检查清单

## “明天开张。我忘了什么？”

测试网上一切正常。阿尔乔姆已经给顾客发消息：“我们收加密货币啦！”可他睡不踏实。等钱变成真的，还有什么会出岔子？

## 弄明白：什么东西放在哪儿




API 密钥的密钥串花不了你的币：RpcNode 那里没有私钥。但它照样能惹祸。给密钥设置“允许的 IP”：这样别人的电脑拿着偷来的密钥串也用不了。

双重验证是指除了密码还要应用里的验证码。在控制台里它位于“设置”。邮箱也要打开同样的保护：很多时候，别处的访问权限都是靠邮箱找回的。

下面这些东西一旦泄露会怎样：

| 泄露了什么 | 外人能做什么 | 怎么办 |
|---|---|---|
| 助记词、私钥 | 把钱拿走，无法追回 | 赶紧把剩下的钱转到新钱包 |
| API 密钥的密钥串 | 操纵付款、监控、端点和 Webhook，读取 Webhook 的密钥串 | 删除该密钥，签发新的，更换 Webhook 密钥串，检查各项设置 |
| Webhook 密钥串 | 伪造 `paid` 消息 | “生成新密钥串” |
| 端点地址 | 花掉你的额度 | 新建一个，删掉旧的（见第22章） |
| 控制台登录权限 | 表中几乎所有事，除了动你的币 | 改密码，启用双重验证 |

## 密钥串不住在代码里

写在代码里的密钥串，会跟着进公共代码仓库、进发给外包方的压缩包、进截图。密钥串要放在服务器配置（环境变量）里，存放它的文件要对外人关闭。

## 动手试试：在自己这儿搜一搜

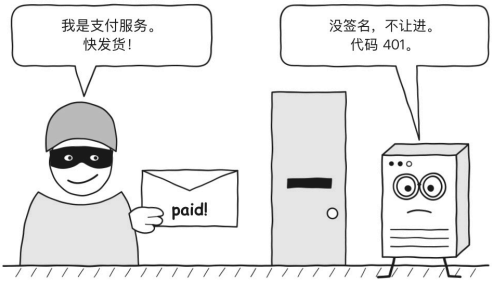
API 密钥的密钥串以 `sk_live_` 开头。我们建一个带一处错误的练习文件夹，再把它找出来：

```
mkdir demo
echo "const SECRET = 'sk_live_example'" > demo/pay.js
echo "const SECRET = process.env.RPCNODE_API_SECRET" > demo/ok.js
grep -rnE 'sk_live_|PRIVATE KEY|xprv' demo
```

```
demo/pay.js:1:const SECRET = 'sk_live_example'
```

搜索找到了第一个文件，没碰第二个。把 `demo` 换成你项目的文件夹。`xprv` 是扩展私钥的开头。如果搜出了东西，光删掉那一行还不够：它可能还留在修改历史里。要当作密钥串已经泄露，签发新的。至于助记词，这样是搜不出来的，那都是普通单词：干脆别把它存在电脑里。

## 别轻信外来消息




来验一验自家的门。把第26章的接收端草稿存为 `receiver.mjs` 并运行：

```
RPCNODE_WEBHOOK_SECRET=demo-secret node receiver.mjs
```

在另一个终端窗口里，用三种外人的方式敲门：

```
URL=http://localhost:3000
BODY='{"type":"payment_order","event":"paid"}'
# 1) 不带签名
curl -s -o /dev/null -w '%{http_code}\n' $URL -d "$BODY"
# 2) 瞎编的签名
curl -s -o /dev/null -w '%{http_code}\n' $URL -d "$BODY" \
  -H "X-RpcNode-Signature: t=$(date +%s),v1=$(printf 'a%.0s' {1..64})"
# 3) 签名是真的，但是一小时前的
T=$(( $(date +%s) - 3600 ))
SIG=$(printf '%s.%s' $T "$BODY" |
  openssl dgst -sha256 -hmac 'demo-secret' | awk '{print $NF}')
curl -s -o /dev/null -w '%{http_code}\n' $URL -d "$BODY" \
  -H "X-RpcNode-Signature: t=$T,v1=$SIG"
```

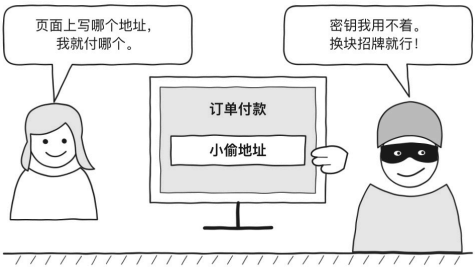
```
401
401
401
```

三次都是“不让进”。要是哪次返回了 200，门就是开着的。把 `- 3600` 改成 `- 0`，第三次敲门会得到 200：签名又新又对。

大额订单再加一把锁。收到 `paid` 后，自己查一下这笔付款（`GET /v1/payments/orders/{orderId}`），核对 `payment_id`、地址、代币、金额和状态。这样即使 Webhook 密钥串被偷，也骗不走咖啡。

另外，用 `https` 接收 Webhook，并限制来访请求的大小：第26章的草稿没做这一点。还可以只放行“密钥串和 IP”区块列出的地址。这是对签名的补充，不是替代。

## 招牌被调包



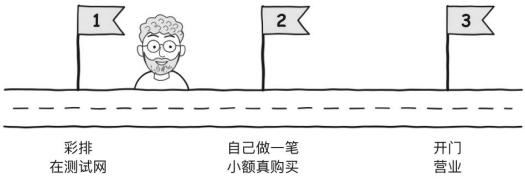

被攻破的网站会显示别人的地址

小偷未必需要弄到密钥。攻破网站，给顾客展示他自己的地址就够了。有用的做法：

- 不能通过网站后台悄悄替换地址清单；
- 时不时换一台电脑，把发出去的地址和自己的钱包核对一下；
- 偶尔自己在店里下单，看看让你付到哪里；
- 及时更新服务器上的程序，登录后台用双重验证。

小偷也会直接找你本人：“客服”、“帮你追回卡住的付款”、从相似地址发来的极小转账（见第6章）。真正的客服不会问助记词。

## 彩排




先把一切在测试网上走一遍。测试网的付款和事件在 TRON Nile 上可用（见第26章）。Free 有沙盒：默认 3 笔未结付款和 3 个地址，那里的风险检查是假的。沙盒里的地址同样不会自动释放。

| 演练什么 | 应有的结果 |
|---|---|
| 付得刚刚好 | 一个包裹 |
| 少付 | 什么都不发，付款过期 |
| 给自己发两次同一条消息 | 一个包裹 |
| 关掉自己的服务器，付款，再打开 | 付款被找到：靠 Webhook 重发或对账 |
| 不带签名敲门 | 401 |

然后在你将来做生意的那个网络上，向自己做一笔小额真实购买。顺便确认你的钱包看得到钱，并且能把它转出去：付手续费的币够不够。

## 法律和税

很多国家对接收加密货币有限制或要征税。上线前请查清自己国家的法律。这里不是法律意见。顺便问问会计，他需要哪些付款记录，从第一天起就保存好。

## 陷阱：“查过一次就忘了”

店开着，周围的一切却在变。套餐到期，账户转为 Free，而 Free 没有主网付款（见第25章）。事件也在消耗额度。地址清单快用完了。给自己设好提醒。

## 上线前检查清单

**密钥和访问权限**

- □ 助记词已抄下，且不在任何服务器和电脑上；
- □ 网站上只有地址，其中几个已和钱包核对过；
- □ 代码里没有密钥串，在项目里搜索结果为空；
- □ API 密钥设置了允许的 IP；
- □ 控制台、邮箱和网站后台都启用了双重验证。

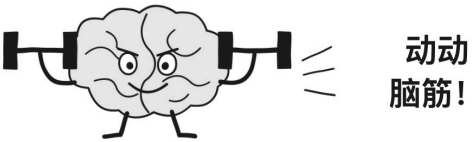
**收款**

- □ 一单一地址，没有重复；
- □ 账单上写明网络、币种、金额、地址、期限和“一笔付清”；
- □ 代币合约取自 `GET /v1/network-tokens`，并和发行方网站核对过；
- □ 确认数按你的订单金额选定；
- □ 验证 Webhook 签名，数据库里的锁守住发货；
- □ 按定时计划对未结订单对账。

**周边事务**

- □ 少付、多付、迟到和退款都有规则；
- □ 名额、额度、套餐期限和地址存量有人盯着；
- □ 清楚归集资金时用什么付手续费；
- □ 订单数据库有备份，并且至少成功恢复过一次；
- □ 彩排通过，自己的真实购买已完成；
- □ 本国法律和税务已查清。

**想一想**




1. API 密钥的密钥串泄露了。别人能把你钱包里的币拿走吗？哪些事仍然必须马上做？
2. 你的接收端对一条没有签名的消息回了 200。有什么危险？
3. TRON Nile 上的检查全部通过。能不经新的检查就开放 Ethereum 收款吗？

*答案：1) 不能，那里没有钱包密钥；但要删除 API 密钥、签发新的、更换 Webhook 密钥串并检查设置。2) 任何人都能不花钱就“付清”订单。3) 不能：网络、地址、手续费和确认数都不同。*

**本章要点**

- 管钱的密钥离网站远一点，密钥串离代码远一点。
- 外来消息、截图、邮件，都不能凭它一句话就信。
- 先彩排，再做一笔小额真实购买，然后开张。

**第六部分到此结束**。阿尔乔姆搭好了店铺，知道出故障时怎么办，也知道怎么守护它。

**接下来**：第31章。DeFi：没有中间人的金融。

# 第31章 DeFi：没有中间人的金融

## “币能不能别光躺着？”

阿尔乔姆有一部分营业收入攒成了 USDC。一个熟人建议：“放进 DeFi 吧，在那儿钱自己生钱。”这个没有柜台、没有出纳的地方到底是什么？

> 这不是理财建议，只是解释。协议名称仅作举例。在你的国家，这类服务可能受限制或要征税：请查清法律。

## 弄明白：银行和程序

DeFi，即“去中心化金融”，是由智能合约（见第9章）来执行的兑换、借贷等金融服务。一组按共同规则运作的合约叫作**协议**。

| | 银行 | DeFi |
|---|---|---|
| 钱在哪里 | 在银行 | 在地址上或合约里 |
| 谁来执行 | 员工和他们的系统 | 网络里的程序 |
| 何时营业 | 按自己的时间表 | 全天候 |
| 规则在哪里 | 合同里 | 代码里 |
| 操作错了 | 可以找银行 | 通常没人能撤销 |
| 丢了访问权 | 凭证件能解决 | 没有密钥就找不回 |

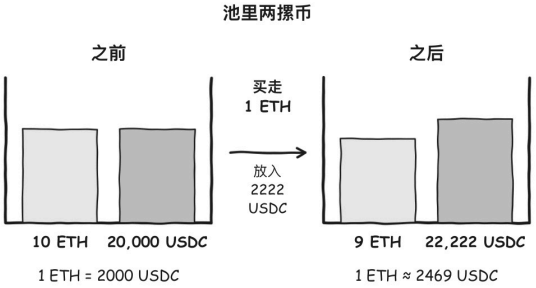
“没有中间人”不等于“不需要信任”。你要信任代码、能修改代码的人、价格提供方和币的发行方。

## 不靠交易所的兑换：流动性池

普通交易所要给买家找卖家。DeFi 里常见的做法是设一个**流动性池**——一个同时放着两种币的合约。往池子里放一种，取走另一种。不用等第二个人。

能换到多少，由公式决定。简单的池子用的是：$x \times y = k$，其中 x 和 y 是两种币的存量。兑换不能让这个乘积变小。

这里的价格就是两个存量之比。假设池中有 10 ETH 和 20 000 USDC。那么 1 ETH 约值 2000 USDC，$k = 200\,000$。阿尔乔姆想取走 1 ETH（手续费暂时不算）。剩下 9 ETH，所以 USDC 必须变成 $200\,000 \div 9 \approx 22\,222$。




他要付 2222 USDC，而不是 2000：**自己这一买把价格抬高了**。这叫**价格影响**。在大一百倍的池子里（1000 ETH 和 2 000 000 USDC），同样 1 ETH 大约只需 2002 USDC。

如果池子里的 ETH 比别处贵了，就会有人把 ETH 拿来、把 USDC 取走，直到价格靠拢。这就是**套利**。

往池子里放币的是**流动性提供者**，他们分得一部分手续费。但当价格偏离时，池子会把涨价的币交出去，换进另一种。不计手续费，价格涨一倍时，投入部分的价值比单纯拿着不动少约 5.7%。这叫**无常损失**：如果你偏在这时撤出，损失就一点也不“无常”了。

## 动手试试：问一个真实的池子

以 Ethereum 上 Uniswap v2 协议的 USDC 与 WETH 池为例。WETH 是做成代币的 ETH，一比一。

池子地址不能随便搜一个来用。我们从 Uniswap 文档取了池子“工厂”的地址，从发行方 Circle 取了 USDC 地址，从 ethereum.org 取了 WETH 地址。然后向工厂提问 `getPair`：“这一对的池子在哪？”回答是：`0xB4e16d0168e52d35CaCD2c6185b44281Ec28C9Dc`。

接下来问 `getReserves`——“存量是多少？”。这是第9章的 `eth_call`：只读不改。

```
curl https://public-ethereum-mainnet.rpcnode.dev \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,"method":"eth_call","params":[
  {"to":"0xB4e16d0168e52d35CaCD2c6185b44281Ec28C9Dc",
   "data":"0x0902f1ac"},"latest"]}'
```

2026 年 11 月 27 日凌晨，节点这样回答：

```
{"jsonrpc":"2.0","id":1,"result":"0x000000000000000000000000000000000000000000000000000000079ecc8266bc00000000000000
0000000000000000000000000000000010b762ae141ddef4f50000000000000000000000000000000000000000000000000000000006b08b4a
f"}
```

`0x` 后面是三个各 64 个字符的数：

| 数（去掉左边的零） | 是什么 |
|---|---|
| `79ecc8266bc` | USDC 存量，6 位小数：8 378 617 USDC |
| `10b762ae141ddef4f50` | WETH 存量，18 位小数：4933.80 WETH |
| `6b08b4af` | 存量上次变动的时间，自 1970 年起的秒数 |

哪种币排第一，由问题 `token0` 和 `token1` 给出：这个池子第一种是 USDC。剩下的就是做除法：

```
python3 -c 'u = 0x79ecc8266bc / 10**6; w = 0x10b762ae141ddef4f50 / 10**18; print(u, w, u / w)'
```

打印出 `8378617.325244 4933.79553339896 1698.2092728662094`。

在这个池子里，一个 ETH 约值 1698 USDC。你得到的数会不一样。

大宗兑换能拿到多少？这里的兑换手续费是 0.3%：其中 0.25% 归流动性提供者，0.05% 归协议。按同样的存量算出来是这样：

| 付出 | 得到 USDC | 每 1 ETH | 比价格 1698.21 差 |
|---|---|---|---|
| 1 ETH | 1692.77 | 1692.77 | 0.3% |
| 100 ETH | 165 957.86 | 1659.58 | 2.3% |
| 1000 ETH | 1 408 492.58 | 1408.49 | 17% |

*计算结果与 Uniswap 辅助合约（router）对 `getAmountsOut` 问题的回答一直吻合到最后一位。*

重要：用存量相除只适用于 v2 类型的池子。在 Uniswap v3 和 v4 中，资金分布在不同的价格区间里，价格算法不一样。

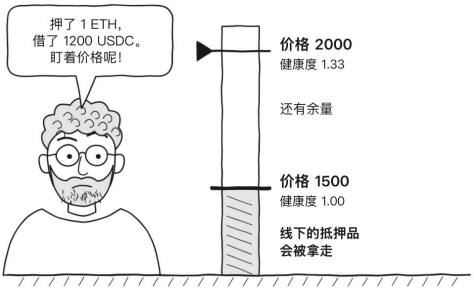
## 抵押借贷与清算

在借贷协议（例如 Aave）里，先存入**抵押品**，再借出价值低于它的钱。图什么？比如需要美元，又不想卖掉 ETH。

每笔借款有一个“健康度”（health factor）：

> 健康度 = 抵押品价值 × 清算阈值 ÷ 债务

编个例子：抵押 1 ETH，价格 2000 USDC，阈值 80%，债务 1200 USDC。健康度：2000 × 0.8 ÷ 1200 ≈ 1.33。价格跌到 1500，健康度正好是 1。




健康度低于 1 时，任何参与者都可以替你偿还一部分债务，有时是全部，并以此拿走带额外奖励的抵押品。这就是**清算**。不会有人打电话让你“快补点钱”：全由程序执行，哪怕是半夜。债务自己也在涨——因为有利息。

## 稳定币：“一美元”靠什么撑着

第9章讲过：稳定币只是**尽力**值一美元左右。撑住的办法各不相同。**发行方储备**：美元和其他资产（USDT、USDC）。合约里的**超额加密资产抵押**（DAI、USDS；抵押物里也可能有别的稳定币）。靠增减币量来调节的**算法**，没有抵押。

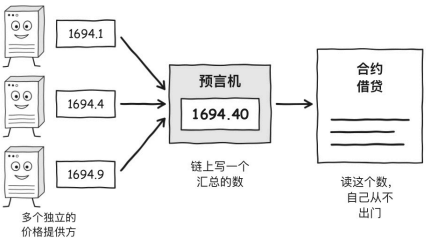
没有哪种办法能打包票。如果池子里某个这样的币跌价了，流动性提供者手里攒下的偏偏就是它。

## 质押简说

当第8章的验证者需要 32 ETH。钱不够的人通过**联合质押池**参与（这不是兑换池）。作为交换，常会给你一种凭证代币。风险有：验证者被罚、合约出错、要信任运营方。收益多少没有人保证。

## 预言机：合约怎么知道价格

借贷合约需要 ETH 的价格。可它自己不会上网：每个节点必须算出同样的结果（见第15章）。把价格写进链里的是**预言机**（oracle）。



例子是 Chainlink 预言机。它的“ETH / USD”喂价地址取自 Chainlink 自己的目录。问题 `latestRoundData` 要最新一条记录：

```
curl https://public-ethereum-mainnet.rpcnode.dev \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,"method":"eth_call","params":[
  {"to":"0x5f4eC3Df9cbd43714FE2740f5E3616155c5b8419",
   "data":"0xfeaf968c"},"latest"]}'
```

```
{"jsonrpc":"2.0","id":1,"result":"0x00000000000000000000000000000000000000000000000070000000000008e4f00000000000000
00000000000000000000000000000000000000002773693800000000000000000000000000000000000000000000000000000000006b08aefa
00000000000000000000000000000000000000000000000000000000006b08af0f00000000000000000000000000000000000000000000000700
00000000008e4f"}
```

五个数里的第二个 `2773693800` 是带 8 位小数的价格：1694.40 美元。池子显示的是 1698.21 USDC，相差约 0.2%。第四个数是写入时间：在我们提问前 25 分钟。这个喂价在价格偏离记录值 0.5% 时更新，另外还按时间表更新，大约一小时一次。

为什么不直接用池子里的价格？一笔大额交易能把它瞬间推偏。但预言机也可能出错或滞后，而清算按它给的数执行。

## DeFi 应用怎样读链

兑换或借贷网站只是个橱窗。数据是向节点要的：

- `eth_call`——现在是什么情况：池子存量、余额、预言机价格、借款健康度。
- 事件（见第15章）——什么发生了变化。v2 池子每次变动后发出带新存量的 `Sync` 事件。用 `eth_getLogs` 查，或经 WebSocket 收听（见第11章）。
- 发送——兑换会改写账本。由钱包签名，节点传出去（见第7章），要花 Gas。

下面查我们这个池子在最新区块里的 `Sync` 事件：

```
curl https://public-ethereum-mainnet.rpcnode.dev \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,"method":"eth_getLogs","params":[{
  "address":"0xB4e16d0168e52d35CaCD2c6185b44281Ec28C9Dc",
  "topics":["0x1c411e9a96e071241c2f21f7726b17ae89e3cab4c78be50e062b03a9fffbbad1"],
  "fromBlock":"latest","toBlock":"latest"}]}'
```

```
{"jsonrpc":"2.0","id":1,"result":[]}
```

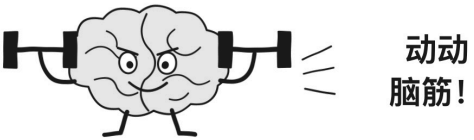
空列表也是回答：这个区块里池子没变化。`getReserves` 的第三个数可以印证：上次变动大约在我们提问前 10 分钟。过去的变化，比如要画价格曲线，那就属于归档或自己的数据库了（见第10章）。

## 陷阱：什么会出问题

| 危险 | 会发生什么 | 怎样降低风险 |
|---|---|---|
| 代码漏洞 | 钱被盗或被卡住 | 代码审计，但这不是保证 |
| 假代币 | 名字眼熟，地址是别人的 | 核对地址（见第9章） |
| 无限授权（`approve`） | 代币日后被拿走 | 少授权，多余的清零 |
| 滑点 | 到手比预期少 | 设定最低到账数量 |
| 清算 | 抵押品连同奖励被拿走 | 留足余量，勤盯着 |
| 仿冒网站 | 山寨网站要你签名 | 核对网址；不输入助记词 |
| “稳赚不赔” | 许诺一番然后消失 | 不参与 |

滑点是预期价格和实际价格之差。兑换交易在队列里等着的时候，别人的交易会推动价格，有时是故意的。最低到账数量能保护你：比它更差，兑换就不成交，不过 Gas 还是会花掉。

### 想一想




1. 池中有 100 ETH 和 200 000 USDC。有人取走了 20 ETH。他放进了多少 USDC（不计手续费）？
2. 抵押 2 ETH，价格 2000 USDC，阈值 80%，债务 2400 USDC。健康度是多少？ETH 价格到多少时它等于 1？
3. 一个网站许诺每天 3% 且无风险，要求授权无限额花费你的 USDT。你怎么做？

*答案：1) 剩下 80 ETH，所以 USDC 变成 250 000：他放进 50 000，每个 ETH 合 2500。2) 约 1.33；价格 1500 时。3) 什么都不签，关掉网站。*

### 本章要点

- 在 DeFi 里，服务由合约执行，风险和责任很大程度上落在你身上。
- v2 类型池子的价格由两个存量决定，大额交易会推动它。
- 应用读链用的请求和你一样：`eth_call` 和事件。

接下来：结语。从顾客的一封信到自己的店铺。

核实于 2026 年 11 月 27 日：developers.uniswap.org、developers.circle.com/stablecoins/usdc-contract-addresses、ethereum.org（wrapped-eth、defi、stablecoins、staking、developers/docs/oracles）、aave.com/help/borrowing/liquidations、docs.chain.link。

# 结语 从顾客的一封信到自己的店铺

一切始于信里的一句话：“可以用加密货币付款吗？”那时阿尔乔姆还不知道什么是节点，靠手动查钱包。看看你和他一起走过了怎样一段路。

## 阿尔乔姆现在会什么

| 部分 | 学到了什么 |
| --- | --- |
| 1. 为什么需要服务商 | 计算自备服务器的开销，决定节点自己养还是用服务商的 |
| 2. 从零认识区块链 | 理解区块、密钥、签名、手续费、确认数和代币 |
| 3. 节点与 RPC | 向节点提问，区分节点种类，检查它有没有掉队 |
| 4. 各类网络如何运作 | 看出网络之间的区别，挑选适合顾客的网络 |
| 5. RpcNode | 开端点，计算调用价格，不靠轮询得知付款 |
| 6. 动手实践 | 搭建收款功能，扛住故障，保护店铺 |
| 7. 下一步：DeFi | 直接从链上读取池子，诚实地评估风险 |

## 值得记住的七条规则

1. 地址可以给所有人看，私钥和助记词谁都不给。
2. 要一次查全：网络、代币合约地址、收款人和金额。
3. 交易成功且攒够确认数，钱才算到账，而不是收到截图就算。
4. 状态码 200 还不等于成功：要看回答里面装着什么。
5. 一笔付款发一次货，不管关于它的消息来了多少次。
6. 像保管密码一样保管端点和密钥串。
7. 别轻信：向网络查询，并检查它的回答。

## 明天从哪儿开始

1. 重做第3章向公共节点发的请求。
2. 开一个自己的端点，用上测试网（见第22章）。
3. 用玩具钱给自己付一次款，演练一遍（见第28、30章）。
4. 过一遍第30章的检查清单。
5. 查清自己国家的法律和税。

## 如果有内容过时了

本书描述的是各网络和这项服务在 2026 年 11 月 27 日的样子。网络在升级，价格和套餐在变。如果你的节点给出的回答和书里不一致，请相信节点和最新文档。RpcNode 的套餐和功能见网站 rpcnode.dev。

## 再见

要不要把一部分营业收入放进 DeFi，阿尔乔姆还在考虑。不过他知道该问哪些问题了。这大概是他从整个故事里带走的最重要的东西。

书里有些数字会过时，动手核实的习惯不会。祝你和你的事业一切顺利！



# 附录 A 术语表

*括号里是详细讲解该词的章节号。*

**地址**——收钱用的“信箱号”（6）。

**AML**——核查资金是否和不良记录有牵连（27）。

**API**——一个程序找另一个程序办事的窗口（11）。

**归档节点**——记得过去任意区块时的状态（10）。

**负载均衡**——把请求分给多个节点（23）。

**区块**——“账本的一页”：交易、编号、时间和上一区块的印章（5）。

**区块链**——按共同规则、由大家一起维护的公共记录清单（4）。

**验证者**——交押金并参与生成区块的参与者（8）。

**Webhook**——有事发生时，服务主动发到你地址上的请求（26）。

**Wei**——ETH 的最小单位（1）。

**WebSocket**——为接收消息而保持不关闭的连接（11）。

**Gas**——工作量单位，Ethereum 及其亲戚按它收手续费（15）。

**gRPC**——程序之间按严格格式用紧凑消息通信的方式（11）。

**双重支付**——想把同一枚币付两次（4）。

**DeFi**——由智能合约执行的金融服务（31）。

**幂等性**——重复多少次，结果都和做一次一样（29）。

**索引器**——把区块链记录归进“卡片柜”的程序（10）。

**手续费**——为区块里的位置付给网络的钱（7）。

**共识**——参与者选定同一条链所依据的规则（8）。

**钱包**——保存密钥的程序或设备（6）。

**额度**——服务用来计算请求消耗的单位（3、25）。

**L2，“二层”**——在自己这边执行转账、把记录发到 Ethereum 的网络（16）。

**清算**——抵押品不足以担保借款时，把抵押品卖掉（31）。

**请求限制（RPS）**——每秒可以问多少次（3、25）。

**挖矿**——为争取添加区块的权利而靠穷举解题（8）。

**内存池（mempool）**——等待进区块的转账队列（7）。

**跨链桥**——把价值从一个网络搬到另一个网络的方法（16）。

**Nonce**——账户交易的顺序号（15）。

**节点**——运行网络程序的电脑：保存、检查、回答（1）。

**预言机**——把外界信息（如价格）写进链里的一方（31）。

**签名**——证明转账由密钥主人授权（6）。

**确认数**——压在你交易上面的区块数，连它所在的区块一起算（5）。

**私钥**——支配资金的东西；谁都不给看（6）。

**服务商**——维护节点并提供访问的服务（1、3）。

**流动性池**——放着两种币、按公式让人兑换的合约（31）。

**链重组**——一部分最新区块被另一条分支替换（8）。

**REST**——每种问题在地址里有自己路径的提问方式（11）。

**RPC、JSON-RPC**——请另一台电脑上的程序干活的方式（11）。

**助记词**——能恢复密钥的 12 或 24 个单词（6）。

**同步**——节点追赶网络的过程（12）。

**智能合约**——写进区块链的程序（9）。

**事件（log）**——合约对发生了什么的通告（15）。

**稳定币**——尽力保持在一美元左右的代币（9）。

**质押（staking）**——用币作押金参与保护网络（8）。

**测试网**——币没有价格的演练场（22）。

**代币**——由智能合约记账的记录（9）。

**裁剪节点**——全部检查过，但删掉了旧区块（10）。

**UTXO**——Bitcoin 及其亲戚里未花费的“钱块”（14）。

**最终性**——过了这一刻，区块就不会再被替换（8）。

**哈希，“印章”**——任意数据的简短指纹（5）。

**Chain ID**——EVM 网络的编号（16）。

**EVM**——执行合约的 Ethereum 虚拟机（15）。

**端点**——程序向节点提问所用的地址（3）。

# 附录 B 一页看懂书中的网络

*信息出自第6、8章和第14 – 20章，截至 2026 年 11 月 27 日。它们会随网络升级而变化。*

## Bitcoin 和亲戚们（第14章）

| 网络 | 币 | 出块间隔，约 | 地址开头 |
|---|---|---|---|
| Bitcoin | BTC | 10 分钟 | `1`、`3`、`bc1` |
| Litecoin | LTC | 2.5 分钟 | `L`、`M`、`ltc1` |
| Dogecoin | DOGE | 1 分钟 | `D` |
| Bitcoin Cash | BCH | 10 分钟 | `bitcoincash:q…` |
| Dash | DASH | 2.5 分钟 | `X` |
| Zcash | ZEC | 75 秒 | `t1`、`zs`、`u1` |

## Ethereum 和 EVM 网络（第15、16章）

它们的地址都以 `0x` 开头。

| 网络 | Chain ID | 手续费用 | 出块间隔，约 |
|---|---|---|---|
| Ethereum | 1 | ETH | 12 秒 |
| Base | 8453 | ETH | 0.2 秒 |
| Arbitrum One | 42161 | ETH | 0.3 秒 |
| BNB Smart Chain | 56 | BNB | 0.45 秒 |
| Mantle | 5000 | MNT | 2 秒 |
| Monad | 143 | MON | 0.3 秒 |

## 其他网络（第17–20章）

| 网络 | 币 | 地址开头 | 何时成定局 |
|---|---|---|---|
| TRON | TRX | `T` | 约一分钟 |
| Solana | SOL | 没有特定开头 | 不到一秒 |
| TON | Gram | `EQ`、`UQ` | 约一秒 |
| Sui | SUI | `0x` | 不到一秒 |
| Aptos | APT | `0x` | 不到一秒 |
| XRP Ledger | XRP | `r` | 3 – 5 秒 |
| Stellar | XLM | `G` | 约 5 秒 |
| Cardano | ADA | `addr1` | 逐渐确定 |
| Polkadot Hub | DOT | `1` | 以标记为准 |
| Cosmos Hub | ATOM | `cosmos1` | 很快 |

# 附录 C 请求速查表

所有示例都能在 RpcNode 公共节点上运行：无需注册和密钥，但有频率限制。用在自己项目里时，把地址换成个人端点。

## JSON-RPC：一条命令走天下

```
curl https://public-ethereum-mainnet.rpcnode.dev \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,"method":"方法","params":[参数]}'
```

把下面表里的方法和参数填进去。换别的网络，改地址里的 `ethereum`。

## Ethereum 和 EVM 网络

| 要查什么 | 方法 | 参数 |
| --- | --- | --- |
| 最新区块号 | `eth_blockNumber` | 空 |
| 网络编号 | `eth_chainId` | 空 |
| 原生币余额 | `eth_getBalance` | `"0x地址","latest"` |
| 交易回执 | `eth_getTransactionReceipt` | `"0x哈希"` |
| Gas 价格 | `eth_gasPrice` | 空 |
| 向合约提问 | `eth_call` | `{"to":"0x…","data":"0x…"},"latest"` |

## Bitcoin 和 Solana

| 地址 | 方法 | 查什么 |
| --- | --- | --- |
| `public-bitcoin-mainnet` | `getblockcount` | 链高度 |
| `public-bitcoin-mainnet` | `getblockchaininfo` | 节点状态 |
| `public-solana-mainnet` | `getSlot` | slot 编号 |
| `public-solana-mainnet` | `getBalance` | 以 lamport 计的余额，参数是地址 |

## TRON 和 TON：问题写在地址里

```
curl https://public-tron-mainnet.rpcnode.dev/wallet/getnowblock
```

```
curl https://public-ton-mainnet.rpcnode.dev/api/v2/getMasterchainInfo
```

## 错误代码

| 代码 | 发生了什么 |
| --- | --- |
| `-32700` | 请求无法解析 |
| `-32601` | 方法不存在或不可用 |
| `-32602` | 参数无效 |
| `-32005`，HTTP 429 | 请求太频繁 |
| `-32006`，HTTP 402 | 额度不足 |

完整文档和最新套餐

**https://rpcnode.dev**

我们的 Telegram

**@rpcnode_dev**

本书作者：rpcnode.dev

全书完。

# 用大白话讲

# 拨开迷雾看区块链

## 这本书能教你什么？

顾客问：“能用加密货币
付款吗？”怎么回答，怎么知道
钱到了账，又怎样不让它丢在半路？

书里有：区块、密钥、手续费和代币，
节点和对节点的请求，Bitcoin、Ethereum、
TRON、Solana、TON 等网络。然后是
实战：RpcNode 端点、付款、
Webhook 和店铺防护。最后
坦诚聊一聊 DeFi。

“我的网站会
算购物车，
会发邮件。
它怎么才能知道
区块链上
发生了什么？”

——阿尔乔姆，
本书主人公




区块链
怎样运作

节点
和网络

网站
收款

DeFi
及其风险

## 这本书
## 有什么特别？

不需要任何
预备知识。
每章有主人公的
问题、简单的
回答、插图
和练习。
请求可以
亲手执行，
看到
真实的回答。

**rpcnode.dev**
docs.rpcnode.dev
Telegram: @rpcnode_dev
作者：rpcnode.dev