

## 上一次疑惑：

- 哈希函数加密
- 资金池-恒定乘积

**内容提纲**

**Part 1: Blockchain & Web3**

- 区块链技术原理：哈希函数、非对称加密、带哈希指针的区块链表、数据传播、共识机制
- 最早的公有链——比特币
- 最大的公有链——以太坊：智能合约、layer2
- 五花八门的链——Salona、BNB、Polygon。各大公链上的meme币（ICO乱象）
- 区块链在实体产业中的应用——数据交易、光伏发电RWA等
- 区块链在数字产业中的应用——数字藏品、元宇宙等
- Web1 → Web2 → Web3

**Part 2: Crypto Finance**

- 中心化交易所→去中心化交易所（订单簿→流动性池）
- 热钱包→冷钱包
- 传统金融产品/衍生品→链上金融产品/衍生品→DeFi
- 稳定币（商业模式、货币金融、与CBDC对比）
- 资产证券化→RWA ➔ “链上金融” & “链下金融”的交互

In [ ]:

## 两个核心概念

### 1. 哈希函数 (Hash Function)

**这是什么？** 哈希函数就像一个“数字指纹机”。根据slides，它将**任意长度的文本**（比如一句话或整本书）转换成一串**固定长度**的数字和字母（例如 256 位的字符串）。它有两个关键特性：

- **单向性 (One-way)**：从输入算出输出很容易，但想从输出反推输入几乎是不可能的。
- **确定性与雪崩效应 (Deterministic & Avalanche)**：只要输入有一点点微小的变化（比如改了一个标点），输出的结果就会发生翻天覆地的变化。

**在区块链中用在哪？起什么作用？**

- **工作量证明 (PoW) 挖矿**：这是比特币等公链的核心。矿工需要不断尝试不同的随机数，使得区块数据的哈希值满足特定的条件（比如小于某个特定值）。这就像是在解一道极难的数学题，保证了记账权的公平分配和网络的安全性。
- **数据结构与完整性**：区块链利用 **Merkle Tree**（默克尔树）这种数据结构来组织交易。它通过层层哈希，把大量交易浓缩成一个“根哈希”。如果任何一笔交易被篡改，根哈希就会完全改变，从而被网络识别出来。

### 2. 非对称加密 (Asymmetric Encryption)

**这是什么？** 与我们常用的只有一个密码的“对称加密”不同，非对称加密有一对密钥：**公钥 (Public Key)** 和 **私钥 (Private Key)**。

- **公钥**: 像你的邮箱地址，是公开的。
- **私钥**: 像你的邮箱密码，必须严格保密。

**在区块链中用在哪？起什么作用？**

- **身份验证与数字签名**: 这是最主要的用途。当你发起一笔交易（比如转账给朋友）时，你用自己的**私钥**对交易进行“签名”。网络中的其他人（矿工和节点）使用你的**公钥**来验证这个签名。
- **作用**: 这证明了这笔钱确实是你发的，而且内容没有被篡改，同时你不需要暴露你的私钥，保证了资产安全。

## 哈希函数

### 哈希函数的定义

#### 1. 通用定义

哈希函数  $H$  将任意长度的消息  $M$  映射为固定长度的摘要  $h$ :

$$h = H(M)$$

其中：

- $M \in \{0, 1\}^*$  (任意长度的二进制串)
- $h \in \{0, 1\}^n$  (固定长度  $n$  的二进制串，例如 SHA-256 中  $n = 256$ )

#### 2. SHA-256 核心逻辑函数 (Core Logic Functions)

SHA-256 并非依赖传统的代数运算，而是依赖**位运算 (Bitwise Operations)** 的组合来打乱数据。核心函数如下：

**选择函数 (Choice) 和 多数函数 (Majority):**

$$Ch(x, y, z) = (x \wedge y) \oplus (\neg x \wedge z)$$

$$Maj(x, y, z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$$

**压缩变换中的  $\Sigma$  函数:**

$$\Sigma_0(x) = ROTR^2(x) \oplus ROTR^{13}(x) \oplus ROTR^{22}(x)$$

$$\Sigma_1(x) = ROTR^6(x) \oplus ROTR^{11}(x) \oplus ROTR^{25}(x)$$

**消息扩展中的  $\sigma$  函数:**

$$\sigma_0(x) = ROTR^7(x) \oplus ROTR^{18}(x) \oplus SHR^3(x)$$

$$\sigma_1(x) = ROTR^{17}(x) \oplus ROTR^{19}(x) \oplus SHR^{10}(x)$$

## 符号说明:

- $\wedge$ : 按位与 (AND)
- $\oplus$ : 按位异或 (XOR)
- $\neg$ : 按位取反 (NOT)
- $ROTR^n(x)$ : 循环右移  $n$  位
- $SHR^n(x)$ : 右移  $n$  位

## 哈希函数示例代码1——确定性与雪崩效应

```
In [3]: # 哈希函数示例代码1——确定性与雪崩效应
import hashlib

def get_hash(text):
    # 使用 SHA-256 算法, 这也是比特币使用的哈希算法
    return hashlib.sha256(text.encode('utf-8')).hexdigest()

# 场景模拟: 比较两个极度相似的字符串
text1 = "I love Digital Finance"
text2 = "I love Digital Finance." # 注意: 仅仅多了一个句号

hash1 = get_hash(text1)
hash2 = get_hash(text2)

print(f"输入1: {text1}\n哈希值1: {hash1}\n")
print(f"输入2: {text2}\n哈希值2: {hash2}\n")

输入1: I love Digital Finance
哈希值1: 14f962560929788a1c2a1f8c4f6b17892b0ea0750330872361104c2bce0cf2d6

输入2: I love Digital Finance.
哈希值2: 35a5651a81240dc7f4b8e1acd4d859605a97ecbe38c74e0b3ef0fb6be2378f0
```

## 哈希函数示例代码1——简单的位差异比较

```
In [2]: # 哈希函数示例代码2——简单的位差异比较
# 我们可以计算两个哈希值在二进制层面有多少位是不同的
bin1 = bin(int(hash1, 16))[2:].zfill(256)
bin2 = bin(int(hash2, 16))[2:].zfill(256)
diff_count = sum(c1 != c2 for c1, c2 in zip(bin1, bin2))
print(f"两个哈希值共有 {diff_count} 个二进制位不同 (总共256位)")

两个哈希值共有 126 个二进制位不同 (总共256位)
```

这部分代码是用来直观展示雪崩效应 (Avalanche Effect) 的，即计算两个哈希值在 **最底层 (二进制)** 到底有多少位不同。

我们逐行拆解这行代码: `bin(int(hash1, 16))[2:].zfill(256)`

1. `hash1`: 这是一个 **十六进制 (Hex)** 字符串, 比如 "a1b2..."。SHA-256 的输出长度是 64 个十六进制字符。
2. `int(hash1, 16)`: 计算机不认识 "a" 或 "b", 所以我们告诉它: "把这个 16 进制的字符串, 转换成一个巨大的整数 (Integer)。"
3. `bin(...)`: 把这个大整数转换成 **二进制 (Binary)** 字符串, 形式是 "0b10101..."。

- 知识点：1个十六进制数 = 4个二进制位（比如 Hex F = Binary 1111）。所以64个Hex字符  $\times 4 = 256$  位 (Bits)。这就是“256”的由来。
4. [2:] : `bin()` 生成的字符串前面带有 "0b" (表示 binary)，这两位我们要去掉，只留数据。
5. `.zfill(256)` : 这是关键！有时候生成的二进制不足 256 位（比如开头是 0），这个函数会在前面补零，强行补齐到 256 位长度，方便后面一位位对齐比较。

后两行逻辑：

```
# zip函数把两个长长的 0101 串像拉链一样对齐，每次各取一位 (c1, c2)
# c1 != c2 判断这一位是否不同（不同记为True/1，相同记为False/0）
diff_count = sum(c1 != c2 for c1, c2 in zip(bin1, bin2))
```

结论：这就是在计算 汉明距离 (Hamming Distance)。对于 SHA-256，只要输入变一点点，输出的 256 位里通常会有 50% (约 128 位) 发生翻转。这就是雪崩效应。

## 哈希函数示例代码2——单向性

“单向性”意味着：给你  $H(x)$ ，你算不出  $x$ 。怎么演示这个特性？最好的方法是演示\*\*“暴力破解”(Brute Force) \*\*。

在 Jupyter Notebook 中，你可以写一个代码：假设我们捡到了一个哈希值（比如这是某个人的银行卡密码的哈希），我们想知道密码是多少。我们别无他法，只能从 000000 开始一个个试。

```
In [6]: get_hash("8888")
Out[6]: '2926a2731f4b312c08982cacf8061eb14bf65c1a87cc5d70e864e079c6220731'
```

```
In [7]: import hashlib
import time

def get_hash(text):
    return hashlib.sha256(text.encode('utf-8')).hexdigest()

# 1. 假设这是我们在区块链上看到的一个哈希值（其实它是“8888”的哈希）
# target_hash = "050f5a0e4d69324d673199c4c795f7e754593f6e80b2d692095925e0e1a14878"
target_hash = "2926a2731f4b312c08982cacf8061eb14bf65c1a87cc5d70e864e079c6220731"

print(f"🎯 目标哈希值: {target_hash}")
print("🧙 开始尝试暴力破解（假设是4位数字密码）...")

start_time = time.time()
found = False

# 2. 模拟黑客，只能一个个猜
for i in range(10000):
    guess = f"{i:04d}" # 生成 '0000', '0001' ... '9999'
    guess_hash = get_hash(guess)

    if guess_hash == target_hash:
        print(f"\n✅ 找到了！原始密码是: {guess}")
        print(f"⌚ 耗时: {time.time() - start_time:.4f} 秒")
        found = True
        break
```

```
if not found:  
    print("✖ 未找到匹配的密码")
```

🎯 目标哈希值: 2926a2731f4b312c08982cacf8061eb14bf65c1a87cc5d70e864e079c6220731  
⌚ 开始尝试暴力破解 (假设是4位数字密码) ...

✓ 找到了! 原始密码是: 8888  
⌚ 耗时: 0.0411 秒

**引导思考:** “对于4位数字，电脑瞬间就能猜出来。但比特币的私钥是**256位**的随机数。如果用同样的办法去猜，要把全宇宙的原子都变成计算机，算到宇宙毁灭也算不出来。这就是**单向性的保证**。”

## 强者恒强？PoW与PoS

### 1. PoW (工作量证明 - 比特币)

- **机制:** 算力 (Hash Rate) 就像“抽奖券”。算力越多，每秒能尝试的哈希次数越多，抽中大奖 (记账权+出块奖励) 的概率就越大。
- **是强者恒强吗？是的，有明显的规模效应。**
  - 买1万台矿机比买1台矿机，单台成本更低 (批发价)。
  - 建在大电站旁边的矿场，电费比家庭用电便宜得多。
  - **结果:** 散户基本退出，算力集中在几个大矿池手中。

### 2. PoS (权益证明 - 以太坊)

- **机制:** 钱 (Stake) 就像“股份”。你质押的币越多，被系统选中去记账的概率就越大。
- **是强者恒强吗？也是，甚至更直接。**
  - 在PoW中，你还需要不断买矿机、付电费 (有持续成本)，一旦技术迭代 (如芯片升级)，老强者可能被淘汰。
  - 在PoS中，富人直接用钱生钱 (复利效应)。只要他不卖币，他的份额就会越来越大，地位很难被撼动。这就是所谓的“马太效应”。

一个比喻：

- **PoW** 像是举重比赛，肌肉（算力）越大越容易赢，但你需要每天吃大量的肉（电费）来维持肌肉。
- **PoS** 像是公司分红，持股（代币）越多分得越多，且不需要再投入太多成本，容易导致阶级固化。

## 针对“强者恒强”的应对机制

区块链世界确实在努力通过算法设计来遏制中心化趋势，这里有几个经典的机制：

- **饱和度机制 (Saturation):** 像Cardano和Polkadot这样的PoS链设计了“奖池上限”。一旦某个质押池的资金规模超过临界值，它的收益率就会**断崖式下跌**。这迫使大户把资金分散到更多的小型节点中，人为制造“去中心化”。
- **二次方投票/融资 (Quadratic Voting/Funding):** 在治理或捐赠时，你的权重不是线性增长的（不是1块钱 = 1票），而是成本呈二次方增加。想投第1票花1元，第10票可能要花100元。这就抑制了巨鲸的垄断能力，放大了普通人的声音。

- **随机数抽签 (VRF):** 像 Algorand，它不看谁钱多，而是通过“可验证随机函数”像买彩票一样随机选出记账人。钱多只是中奖概率稍高，但无法绝对垄断。

## 非对称加密篇：椭圆曲线密码学 (ECC)

在区块链（比特币、以太坊）中，我们并不使用传统的 RSA 算法（基于大数分解），而是使用更高效的 **椭圆曲线密码学 (ECC)**，具体曲线名为 **secp256k1**。

### 数学原理：在一个奇妙的几何世界里跳跃

要理解 ECC，我们需要把思维从普通的算术转换到几何图形上。

#### 1. 曲线方程 比特币使用的曲线方程非常简洁：

$$y^2 = x^3 + 7$$

这条曲线在坐标系上大致长这样（关于 X 轴对称）：

#### 2. 核心运算：点加法 (Point Addition) 在这个曲线上，我们定义了一种特殊的“加法”：

- $P + Q = R$ : 连接曲线上  $P$  和  $Q$  两点画一条直线，它会与曲线相交于第三点，将这个交点沿 X 轴对称翻转，得到点  $R$ 。
- $P + P = 2P$  (倍点运算): 如果  $P$  和  $Q$  重合，就画切线。

#### 3. 公钥与私钥的关系

- **私钥 ( $k$ ):** 本质上只是一个随机选取的**巨大的整数** (256位)。
- **公钥 ( $P$ ):** 是曲线上的一个**坐标点**  $(x, y)$ 。
- **计算公式:**

$$P = k \times G$$

这里  $G$  是全网公认的一个起始点 (Generator Point)。这个公式的意思是：**在曲线上对  $G$  点进行  $k$  次“点加法”运算。**

#### 关键的单向性 (The Trapdoor):

- **正向计算 (私钥  $\rightarrow$  公钥):** 即使  $k$  很大（比如  $10^{77}$ ），利用二进制展开法 (Double-and-Add)，计算机能在毫秒级算出  $P$ 。
- **逆向计算 (公钥  $\rightarrow$  私钥):** 给你点  $P$  和点  $G$ ，问你是多少次加法得到的？这被称为 **离散对数问题 (Discrete Logarithm Problem)**。以目前的算力，暴力破解需要几十亿年。

```
In [7]: # 曲线方程示意图——连续型
import numpy as np
import matplotlib.pyplot as plt

def plot_elliptic_curve():
    # 1. 定义 x 的范围
    # 也就是  $x^3 + 7 \geq 0$ , 即  $x \geq -1.913$ 
    x = np.linspace(-2.5, 3, 500)

    # 2. 创建网格以绘制隐函数  $F(x, y) = y^2 - x^3 - 7$ 
    y = np.linspace(-6, 6, 500)
```

```

X, Y = np.meshgrid(x, y)

# 定义方程:  $y^2 - x^3 - 7 = 0$ 
F = Y**2 - X**3 - 7

# 3. 设置绘图风格
plt.figure(figsize=(4, 3))
plt.title('Elliptic Curve:  $y^2 = x^3 + 7$  (Bitcoin uses secp256k1)', fontsize=1)
plt.xlabel('x')
plt.ylabel('y')

# 绘制等高线, 只显示 F=0 的部分, 即曲线本身
plt.contour(X, Y, F, levels=[0], colors='blue', linewidths=2)

# 添加坐标轴线
plt.axhline(0, color='black', linewidth=0.5, linestyle='--') # X轴
plt.axvline(0, color='black', linewidth=0.5, linestyle='--') # Y轴

# 添加网格
plt.grid(True, linestyle=':', alpha=0.6)

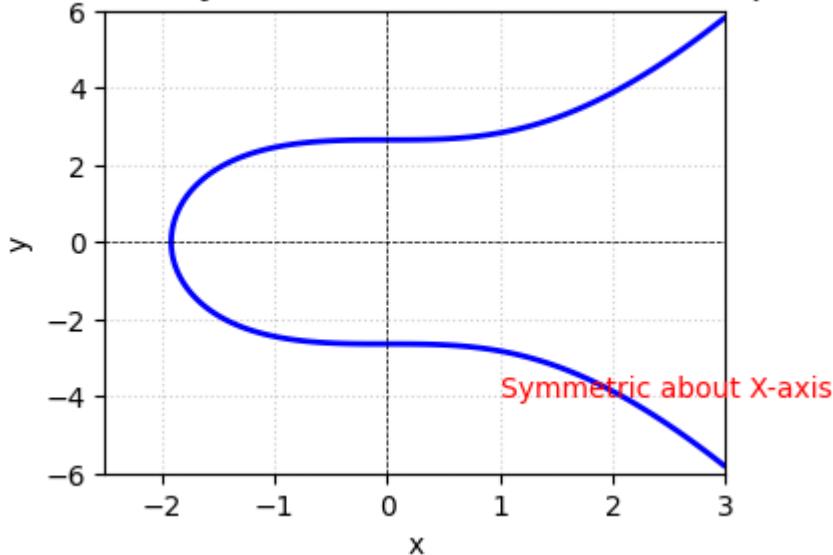
# 标记关于 X 轴对称的特性
plt.text(1, -4, "Symmetric about X-axis", fontsize=10, color='red')

plt.show()

# 运行绘图函数
plot_elliptic_curve()

```

**Elliptic Curve:  $y^2 = x^3 + 7$  (Bitcoin uses secp256k1)**



```

In [16]: # 曲线方程示意图——真实: 点阵
import matplotlib.pyplot as plt

def plot_discrete_ecc(p=59):
    """
    绘制有限域  $F_p$  上的椭圆曲线  $y^2 \equiv x^3 + 7 \pmod{p}$ 
    """
    points_x = []
    points_y = []

    # 1. 遍历所有可能的 x (0 到 p-1)
    for x in range(p):
        # 计算方程右边:  $(x^3 + 7) \% p$ 
        rhs = (x**3 + 7) % p

```

```

# 2. 遍历所有可能的 y, 寻找满足  $y^2 \% p == \text{rhs}$  的点
# (在实际大数计算中, 会使用数论算法如 Tonelli-Shanks 来求解, 而不是暴力遍历)
for y in range(p):
    if (y**2) % p == rhs:
        points_x.append(x)
        points_y.append(y)

# 3. 绘图
plt.figure(figsize=(5, 5))
# plt.scatter(points_x, points_y, color='red', s=50, label=f'${y^2 \equiv x^3 + 7}$')
# plt.scatter(points_x, points_y, color='red', s=50, label=f'${y^2 \equiv x^3 + 7}$')

# 设置图表属性
plt.title(f'Discrete Elliptic Curve over Finite Field $F_{\{{p}\}}$', fontsize=14)
plt.xlabel('x')
plt.ylabel('y')
plt.grid(True, linestyle=':', alpha=0.5)
plt.legend()

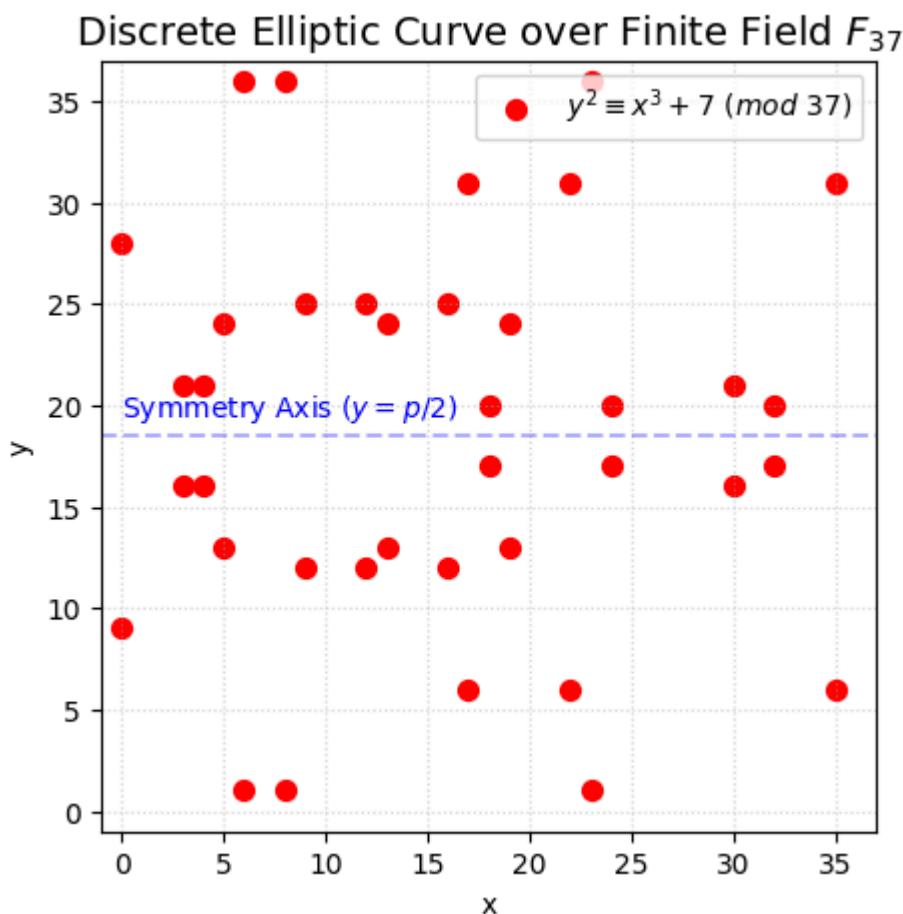
# 标出对称轴 ( $y = p/2$ )
plt.axhline(p/2, color='blue', linestyle='--', alpha=0.3)
plt.text(0, p/2 + 1, "Symmetry Axis ($y=p/2$)", color='blue', fontsize=10)

plt.xlim(-1, p)
plt.ylim(-1, p)
plt.show()

print(f"在素数 p={p} 的域上, 共有 {len(points_x)} 个整数点符合方程。")

```

# 运行代码, 你可以尝试修改 p 为其他素数, 如 17, 97 等  
plot\_discrete\_ecc(37)



在素数  $p=37$  的域上, 共有 38 个整数点符合方程。

# 非对称加密代码示例



我们可以用 Python 的 `ecdsa` 库来模拟这个过程。

下面的代码演示了三个步骤：

1. **生成钥匙对**: 看看私钥 (数字) 和公钥 (坐标点) 长什么样。
2. **签名**: 爱丽丝用私钥签署消息。
3. **验证**: 矿工用公钥验证签名。

```
In [2]: import ecdsa
import binascii

# 1. 模拟生成钥匙对
# 使用比特币/以太坊标准的曲线: SECP256k1
sk = ecdsa.SigningKey.generate(curve=ecdsa.SECP256k1)
vk = sk.verifying_key

# 获取私钥 (通常是保密的) - 本质是一个大整数
private_key_hex = binascii.hexlify(sk.to_string()).decode('utf-8')
# 获取公钥 (公开的) - 本质是曲线上的坐标 (x, y) 压缩而成
public_key_hex = binascii.hexlify(vk.to_string()).decode('utf-8')

print(f"🔑 私钥 (k, 必须保密): {private_key_hex}")
print(f"🌐 公钥 (P, 对外公开): {public_key_hex}")

🔑 私钥 (k, 必须保密): ed3c52012934373b905b4ba37747c030df326de31daa86ed26026b59679ff
9a2
🌐 公钥 (P, 对外公开): 1257e29ea6f838e88ba55448562720d281c2853a83cbe80ef73327d4574d9
85c8af915ba1b99c763c45d9ce98ac974f4b14d7409009d8e3b2f94e132b3a5b44d
```

```
In [3]: # 2. 数字签名过程 (Alice的操作)
message = b"Alice pays Bob 1 BTC"
print(f"📝 交易内容: {message}")

# Alice用自己的私钥对消息的哈希进行签名
signature = sk.sign(message)
print(f"✍ 生成的数字签名: {binascii.hexlify(signature).decode('utf-8')}")

📝 交易内容: b'Alice pays Bob 1 BTC'
✍ 生成的数字签名: 3478842cf831ab1f6491704127a864dacffb1077f6c55173d36b10bf4de835542
d7892819e487d8bf0ffa0f6d5398223a0b83f62cd26895c08686369689753ea
```

```
In [4]: # 3. 验证过程 (矿工/节点的操作)
# 矿工只拥有: Alice的公钥、交易内容、数字签名
try:
    # 验证函数: 如果返回 True 则验证通过, 否则抛出异常
    is_valid = vk.verify(signature, message)
    if is_valid:
        print("✅ 验证成功! 这笔交易确实是 Alice 发出的, 且未被篡改。")
except ecdsa.BadSignatureError:
    print("❌ 验证失败! 可能是伪造的签名或内容被篡改。")

✅ 验证成功! 这笔交易确实是 Alice 发出的, 且未被篡改。
```

```
In [5]: # 4. 模拟篡改攻击
print("👀 黑客尝试篡改交易金额...")
tampered_message = b"Alice pays Bob 100 BTC" # 改了一个数字
try:
    vk.verify(signature, tampered_message)
    print("✅ 验证成功 (这不应该发生)")

```

```
except ecdsa.BadSignatureError:  
    print(f"🛡️ 篡改被识破！签名与篡改后的消息 '{tampered_message}' 不匹配。")
```

🛡️ 黑客尝试篡改交易金额...

🛡️ 篡改被识破！签名与篡改后的消息 'b'Alice pays Bob 100 BTC'' 不匹配。

In [ ]: # todo:记账&清算

## 💡 引导思考

看完这个代码和数学原理，我想问你一个核心问题来检验理解：

在代码的第3步（验证过程）中，矿工**并没有**爱丽丝的私钥，却能确信“这笔交易是爱丽丝签名的”。

结合刚才的椭圆曲线公式  $P = k \times G$ ，你觉得**数字签名**在数学逻辑上是在向矿工证明什么？

- A. 证明爱丽丝算出了一个极其复杂的哈希值。
- B. 证明爱丽丝知道那个数  $k$ ，但不需要把  $k$  告诉任何人。
- C. 证明爱丽丝拥有比特币网络的最高权限。

答案： B

In [ ]:

# 深度解析：去中心化交易的数学与博弈机制

在理解了底层密码学之后，我们进入区块链最迷人的应用层——去中心化金融（DeFi）。与传统金融（TradFi）依赖中心化中介（如纳斯达克、纽交所）不同，DeFi 依靠代码和数学公式来维持市场的运转。

我们将从三个维度来解构这个系统：

1. **微观机制**：单个流动性池为何采用  $x \cdot y = k$ ？
2. **中观网络**：多资产环境下的图论路由。
3. **宏观博弈**：套利者如何连接 CEX 与 DEX 的孤岛。

## 1. 微观视角：自动做市商 (AMM) 的定价原理

在传统金融中，交易依赖 **订单簿 (Order Book)**：买家挂单，卖家挂单，撮合引擎在中间匹配。但在区块链上，存储和计算资源极其昂贵（Gas Fee），实时维护一个庞大的订单簿是不经济的。

于是，**自动做市商 (Automated Market Maker, AMM)** 应运而生。最经典的模型是 **恒定乘积做市商 (CPAMM)**。

### 1.1 核心公式

假设一个流动性池包含两种资产  $A$  和  $B$ ：

- $x$ : 资产  $A$  的储备量

- $y$ : 资产  $B$  的储备量

AMM 强制要求在交易前后，两者的乘积保持不变（忽略手续费）：

$$x \cdot y = k$$

## 1.2 为什么是双曲线？

这个简单的公式  $y = \frac{k}{x}$  蕴含了精妙的金融属性：

1. **无限流动性 (Infinite Liquidity)**：双曲线在第一象限没有交点。这意味着，无论你想买走多少  $x$ （只要  $< x_{total}$ ），池子总能算出你需要支付多少  $y$ 。价格会自动趋向无穷大，但永远有货。
2. **自动定价 (Automatic Pricing)**：市场的瞬时价格  $P$  其实就是曲线上某点的**切线斜率**（的绝对值）：

$$P_{A/B} = -\frac{dy}{dx} = \frac{y}{x}$$

这极其优雅：**价格完全由池子中两种资产的比例决定**。不需要外部预言机，不需要人工干预。

**思考：**这本质上是用“滑点 (Slippage)”来保护流动性提供者。买得越多，单价越贵。

## 2. 中观视角：多币种环境下的图论与路由

当一个 DEX 中存在  $m$  种货币时，理论上可以存在  $C_m^2 = \frac{m(m-1)}{2}$  个交易对。但这些交易对并不是独立的孤岛，它们构成了一个**加权有向图 (Weighted Directed Graph)**。

### 2.1 智能路由 (Smart Routing)

假设用户想把 Token A 换成 Token D。

- **直接路径**： $A \rightarrow D$  池子（可能深度不够，滑点很大）。
- **间接路径**： $A \rightarrow B \rightarrow C \rightarrow D$ （虽然路径长，但在大额交易时可能滑点更低，换出的币更多）。

DEX 的前端算法会运行类似 **Dijkstra** 或 **Bellman-Ford** 的最短路径算法，在全网寻找最优兑换路径。

$$Output_{optimal} = \max_{path \in Paths} \left( \prod_{pool \in path} (1 - fee_{pool}) \cdot ExchangeRate_{pool} \right)$$

这意味着：任意两个货币对之间的供需关系，会通过中间货币（通常是 ETH 或 USDT）传导到整个网络。

## 3. 宏观视角：套利者 (Arbitrageurs) —— 连接孤岛的桥梁

这是整个 Crypto Finance 能够有效运转的关键。我们有  $A$  个中心化交易所 (CEX, 如 Binance) 和  $B$  个去中心化交易所 (DEX, 如 Uniswap)。它们之间没有物理连接，数据库互不相通。

问：为什么比特币在 Binance 卖 60000 美元，在 Uniswap 上也大概率是 60000 美元？

答：套利 (Arbitrage)。

### 3.1 市场的自我平衡机制

AMM 是“盲目”的，它不知道外部市场的价格。如果 Binance 上 ETH 价格暴涨，而 Uniswap 还没变：

1. 价差出现：Binance Price > Uniswap Price。
2. 套利介入：机器人在 Uniswap 买入廉价 ETH（导致 Uniswap 池内 ETH 减少，价格  $P = \frac{y}{x}$  上升）。
3. 平仓获利：机器人立刻在 Binance 卖出 ETH。
4. 结果：Uniswap 的价格被“买”上去了，直到两个市场价格基本一致（价差 < 手续费 + Gas 费）。

### 3.2 结论

在 Crypto Finance 中，价格一致性不是由上帝或监管机构保证的，而是由无数贪婪的套利机器人，通过高频交易强制维持的。

这是一个完美的分布式系统：个体的逐利行为，实现了系统的全局效率。

In [ ]:

[redacted]

In [ ]:

[redacted]

## 流动性提供者——无常损失 (Impermanent Loss)

**无常损失**是指：你把资产放入流动性池 (LP) 后，最终拿回来的价值，少于如果你仅仅是把这些币放在钱包里不动 (HODL) 的价值。

核心逻辑：

1. 你提供了  $ETH$  和  $USDT$ 。
2. 当  $ETH$  价格上涨时，套利者会用  $USDT$  买走你手里的  $ETH$ （因为池子里便宜）。你手里的  $ETH$  变少了， $USDT$  变多了。你实际上是在“分批卖飞”。
3. 当  $ETH$  价格下跌时，套利者会卖给你  $ETH$  换走  $USDT$ 。你手里的  $ETH$  变多了（接盘了）， $USDT$  变少了。你实际上是在“越跌越买”。

无论涨跌，只要价格偏离了你存入时的价格，你相对于“拿着不动”都是亏的。只有价格回到原点，损失才会消失（这也是为什么叫“无常/暂时”损失）。

In [17]: # Jupyter Notebook 演示代码：计算无常损失

```
import numpy as np
import matplotlib.pyplot as plt
```

```

def calculate_impermanent_loss(price_ratio):
    """
    计算无常损失。
    price_ratio: 变动后的价格 / 初始价格 (P_new / P_old)
    公式推导结果: IL = 2 * sqrt(price_ratio) / (1 + price_ratio) - 1
    """
    return (2 * np.sqrt(price_ratio) / (1 + price_ratio)) - 1

# 1. 设置价格变化范围: 从 0.1倍 (跌90%) 到 5倍 (涨400%)
price_ratios = np.linspace(0.1, 5, 100)

# 2. 计算对应的无常损失
il_values = calculate_impermanent_loss(price_ratios)

# 3. 绘图
plt.figure(figsize=(5, 3))
plt.plot(price_ratios, il_values * 100, color='purple', linewidth=2, label='Impermanent Loss')

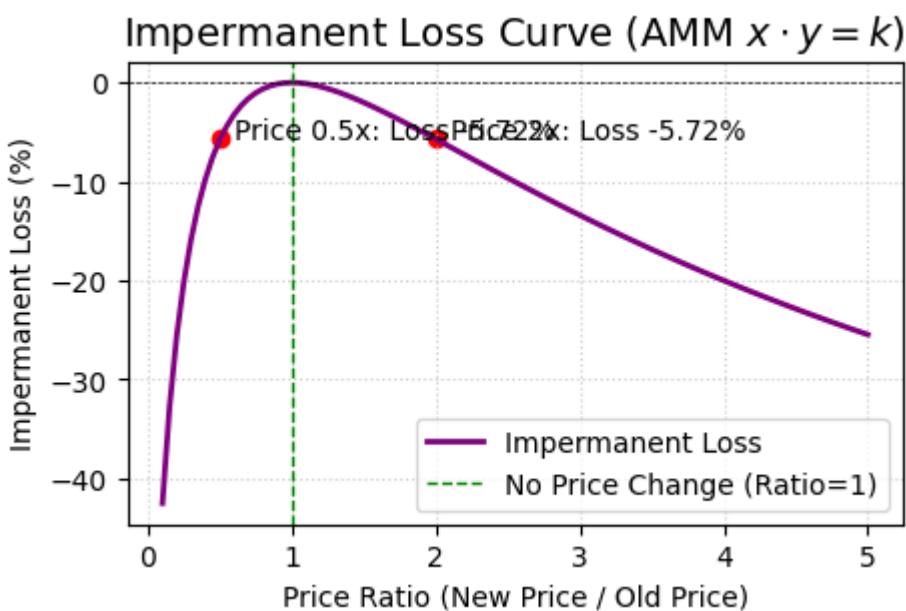
# 添加辅助线
plt.axhline(0, color='black', linewidth=0.5, linestyle='--')
plt.axvline(1, color='green', linewidth=1, linestyle='--', label='No Price Change (Ratio=1)')

# 标注关键点
# 价格翻倍 (2x)
loss_2x = calculate_impermanent_loss(2) * 100
plt.scatter([2], [loss_2x], color='red')
plt.text(2.1, loss_2x, f'Price 2x: Loss {loss_2x:.2f}%', fontsize=10)

# 价格腰斩 (0.5x)
loss_05x = calculate_impermanent_loss(0.5) * 100
plt.scatter([0.5], [loss_05x], color='red')
plt.text(0.6, loss_05x, f'Price 0.5x: Loss {loss_05x:.2f}%', fontsize=10)

plt.title('Impermanent Loss Curve (AMM $x \cdot y = k$)', fontsize=14)
plt.xlabel('Price Ratio (New Price / Old Price)')
plt.ylabel('Impermanent Loss (%)')
plt.grid(True, linestyle=':', alpha=0.6)
plt.legend()
plt.show()

```



### 观察点:

- 只要  $x$  轴不等于 1 (价格变了),  $y$  轴永远是负数。

- 价格翻倍 ( $2x$ )，你大概亏 **5.7%**。
- 价格翻 5 倍，你大概亏 **25%**。
- 这意味着：如果你非常看好一个币暴涨，最好拿着不动（**HODL**），而不是去当 **LP**。**LP** 适合横盘震荡的市场。

## 为什么要用简单的反比例函数 $y = k/x$ ? 🤔

在数学和工程上，**简单往往意味着健壮**。我们可以通过对比来证明  $x \cdot y = k$  的优越性。

### 1. 为什么不用直线 $x + y = k$ （恒定和）？

假如我们规定  $ETH + USDT = k$ 。

- **导数（价格）：**  $P = \frac{dy}{dx} = -1$ （恒定）。
- **后果：** 这意味着无论池子里有多少币，价格永远不变。
- **套利结局：** 一旦外部市场价格变动（比如 ETH 涨了），套利者会瞬间**把池子里便宜的 ETH 全部买光**。池子会彻底干涸（Drain），流动性归零。

### 2. 为什么不用更复杂的曲线？

确实有更复杂的，比如 Curve Finance 针对稳定币（USDT/USDC）设计的混合曲线（Stableswap Invariant），它在价格 1:1 附近非常平坦（像直线），在两端像双曲线。

但在以太坊这种**计算资源极其昂贵**（Gas Fee 高）的环境下，Uniswap V2 选择  $x \cdot y = k$  是因为：

1. **极简计算：** 乘法和除法是计算机最基础的指令，消耗的 Gas 最少。
2. **无限流动性 (Infinite Liquidity) 的数学证明：** 双曲线  $y = k/x$  位于第一象限，**永远没有 x 截距和 y 截距**（渐近线）。这意味着：无论你想买多少 ETH（只要不把池子买穿），在这个函数上**永远能找到一个对应的点**。虽然价格会变得无限高，但永远“有货”。这是做市商机制的根本要求。

In [19]: # 演示代码：直观对比多种曲线

```
import numpy as np
import matplotlib.pyplot as plt

def plot_amm_curves():
    x = np.linspace(0.1, 20, 400)

    # 1. 恒定乘积 (Uniswap): x * y = 100 -> y = 100 / x
    k_prod = 100
    y_prod = k_prod / x

    # 2. 恒定和 (mStable): x + y = 20 -> y = 20 - x
    k_sum = 20
    y_sum = k_sum - x
    y_sum[y_sum < 0] = np.nan # 过滤负数部分

    plt.figure(figsize=(4, 4))

    # 绘制恒定乘积 (双曲线)
    plt.plot(x, y_prod, color='blue', linewidth=2, label='Constant Product ($x \cdot dot
```

```

# 绘制恒定和 (直线)
plt.plot(x, y_sum, color='red', linestyle='--', linewidth=2, label='Constant Sum')

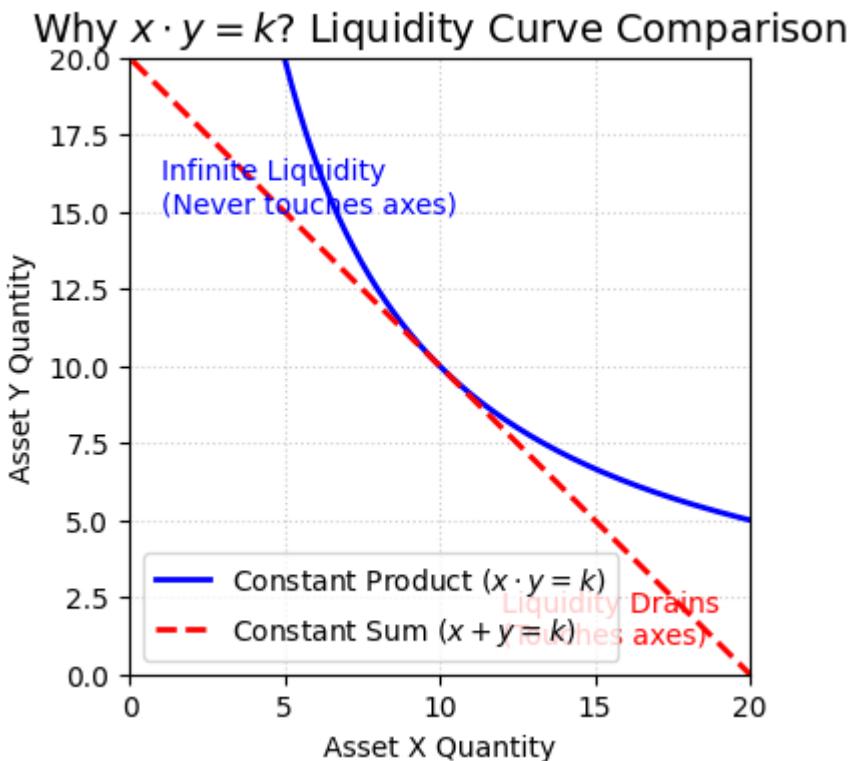
plt.title('Why $x \cdot y = k$? Liquidity Curve Comparison', fontsize=14)
plt.xlabel('Asset X Quantity')
plt.ylabel('Asset Y Quantity')
plt.xlim(0, 20)
plt.ylim(0, 20)
plt.grid(True, linestyle=':', alpha=0.6)
plt.legend()

# 标注关键差异
plt.text(1, 15, "Infinite Liquidity\n(Never touches axes)", color='blue')
plt.text(12, 1, "Liquidity Drains\n(Touches axes)", color='red')

plt.show()

plot_ammm_curves()

```



为什么 Uniswap 选择了最简单的  $y = k/x$ ?

>

1. **生存本能**: 相比于直线 (恒定和), 双曲线永远不会与坐标轴相交, 这意味着池子永远不会被买光, 哪怕价格高到天上去, 市场依然运转。
2. **奥卡姆剃刀**: 在区块链上每一步计算都要烧钱 (Gas)。最简单的公式, 意味着最高的执行效率。这就是**工程美学**战胜了复杂的金融模型。”

In [ ]:

中心化金融>>去中心化金融 —— “代码逻辑”替代“中介信任”

## 1. 去中心化借贷 (Decentralized Lending)

对应 PPT：去中心化借贷 (如 Aave, Compound)

**传统模式：**银行（中介）收取存款，借给贷款人。银行需要审核贷款人的**信用分**（KYC），以防违约。

**DeFi 模式：**没有银行，只有一个**智能合约资金池**。代码不认识你是谁，也没有信用分。

**核心工作逻辑（代码机制）：****超额抵押 (Over-Collateralization)** 为了在“无信任”的环境下借钱，DeFi 引入了一个数学约束：**借出的钱必须少于抵押的钱**。

- **抵押 (Supply)**：用户 Alice 往池子里存入价值 \$150 的 ETH。
- **借贷 (Borrow)**：系统允许 Alice 借出价值 \$100 的 USDC。
- **清算 (Liquidation)**：这是代码最冷酷也最关键的部分。如果 ETH 价格大跌，导致 Alice 的抵押品价值跌破了安全线（比如跌到 \$110），智能合约会允许**清算机器人**（第三方代码）介入，强制卖掉 Alice 的 ETH 来偿还 USDC 债务，并拿走一部分作为奖励。

**给听众的类比：**这就像当铺。当铺不看你的身份证，只看你留下的金表值不值钱。

---

## 2. 理财聚合器 (Yield Aggregators)

**传统模式：**基金经理帮你选股票、债券，手动调仓，赚取管理费。

**DeFi 模式：**智能合约机器人自动在不同的 DeFi 协议之间“搬运”资金，寻找最高收益率。

**核心工作逻辑（代码机制）：****自动复利与策略路由**

- **策略 (Strategy)**：代码中写好了一套逻辑，例如：“把用户的钱存入 Aave 借贷，拿到凭证后去 Uniswap 做流动性挖矿，挖出来的币立刻卖掉换成更多本金，再存入 Aave”。
- **自动执行**：无需人工干预，代码 24 小时监控全网收益率（APY），像“扫地机器人”一样把利润扫回本金池，实现**复利 (Compound Interest)**。

---

## 3. RWA (现实世界资产)

**传统模式：**买美债、房产需要繁琐的纸质确权、中介费，且很难分割（你很难只买 10 美元的某栋大楼）。

**DeFi 模式：**将线下资产的**所有权映射**到链上的 Token。

**核心工作逻辑（代码机制）：****预言机与资产映射**

- **映射**：链下托管机构持有 1 亿美元国债，然后在链上发行 1 亿个 Token。
- **连接**：通过**预言机 (Oracle)** 将现实世界的收益（如国债利息）数据传导到链上，自动分发给 Token 持有者。



### 去中心化借贷的基本原理

在 Aave 等借贷协议中，有一个核心指标叫 **健康因子 (Health Factor,  $H_f$ )**。这个公式决定了你的资产什么时候会被代码强制没收。

公式通常是这样的：

$$H_f = \frac{\sum(\text{Collateral}_i \times \text{LiquidationThreshold}_i)}{\text{TotalDebt}}$$

- $\text{Collateral}_i$ : 你抵押的资产价值
- $\text{LiquidationThreshold}_i$ : 抵押品的折扣率（比如 ETH 是 0.8，意味着 100 块 ETH 最多算 80 块的有效抵押）
- $\text{TotalDebt}$ : 你借出的总债务

当  $H_f < 1$  时，触发清算。

In [ ]:

## 去中心化借贷分类

### 1. 超额抵押 (Over-collateralized)

- **定义**：为了解决链上“无信用分、无KYC、匿名”的问题，DeFi 借贷必须遵循 **抵押品价值 > 借款价值** 的原则。例如，抵押 \$150 的 ETH，只能借出 \$100 的 USDC。
- **核心作用**：
  - **消除违约风险**：如果借款人不还钱，或者抵押品价格下跌，智能合约会自动清算（卖掉）抵押品来偿还债务，无需追究借款人是谁。
- **细分模式**：
  - **CDP (Collateralized Debt Positions) :**
    - **中文**：抵押债仓。
    - **机制**：这种模式下，你借出的钱不是别人存进去的，而是凭空铸造出来的（通常是稳定币）。你把资产锁在保险柜（合约）里，系统给你印出对应的钞票（如 DAI）。还钱时，钞票被销毁，资产解锁。
  - **Collateralized Debt Markets (抵押借贷市场) :**
    - 这里指的是真正的“借贷”，即有人存钱（Lender），有人借钱（Borrower）。
    - **Pooled (资金池模式)**：这是目前的主流（如 Aave）。所有存款人的钱都在一个大池子里，借款人直接从池子里借。利率由算法根据池子的资金利用率（Utilization Rate）自动计算。**效率极高**。
    - **Peer-to-peer (点对点模式)**：这是早期的模式（如 ETHLend）。Alice 发单说“我想借 1 ETH，利息 5%”，Bob 看到后接单。**效率低，流动性差**，现在很少用了。

### 2. 抵押不足 (Under collateralization)

- **现状**：这是 DeFi 的圣杯，但目前很难完全去中心化实现。
- **机制**：通常需要引入“链下信用”或“白名单机制”。例如，只有通过了 KYC 的机构用户，才能在某些协议（如 Maple Finance）中进行非足额抵押借贷。

### 3. Flash Loan (闪电贷)

- **定义：无抵押 (No collateralization)。**

- **编程原理：**利用区块链交易的**原子性 (Atomicity)**。借款、使用资金（如套利）、还款，这三个步骤必须在同一个区块、同一笔交易中完成。
- **逻辑：** `if (还回来的钱 < 借出去的钱 + 利息): revert()`。
- **意义：**如果还不上，交易就会回滚，就像从未发生过一样，所以对贷方来说几乎是零风险的。

## MakerDAO 的双币机制 (MKR & DAI)

MakerDAO 是以太坊上的“去中心化央行”，它设计了一套精妙的博弈机制来维持稳定币 DAI 的价格锚定 \$1。

### 1. 两个代币的角色

- **DAI (稳定币) :**
  - 角色：产品。
  - 性质：这是用户真正想要使用的钱，通过超额抵押 ETH 等资产生成。它对标美元。
- **MKR (权益型代币/治理代币) :**
  - 角色：银行的股份 + 保险。
  - 性质：持有 MKR 的人是这个系统的“管理者”和“最后的风险承担人”。

### 2. 运作机制：MKR 如何保护 DAI？

这是一个**风险与收益对等**的设计：

- **情景 A：系统盈利 (Good Times)**
  - 当用户借出 DAI 时，需要支付**稳定费 (Stability Fee)**（相当于贷款利息）。
  - 这笔利息收入会被用来在市场上回购 **MKR 并销毁**。
  - **结果：** MKR 总量减少，价格上涨。MKR 持有者从 DAI 的广泛使用中获利（分红）。
- **情景 B：系统亏损 (Bad Times - 资不抵债)**
  - 如果抵押品（如 ETH）价格瞬间暴跌，导致资不抵债（坏账），且清算机制来不及反应。
  - 为了保证 DAI 依然值 1 美元（此时背后支撑资产不足 1 美元），系统会自动**增发 MKR 并在市场上抛售**，换取 DAI 来填补债务窟窿。
  - **结果：** MKR 总量增加，价格下跌。MKR 持有者资产被稀释，实际上是他们在为系统的坏账买单。

### 3. 总结

“MakerDAO 就像一个去中心化的银行。**DAI** 是它发行的钞票。**MKR** 是它的银行股票。

如果银行赚了钱了（收利息），就回购股票让股东赚钱；如果银行破产了（坏账），就强行增发股票卖钱还债，让股东赔钱。这就迫使 MKR 的持有者必须负责任地管理这个系统（比如设置合理的抵押率），否则他们的资产会归零。”

In [ ]:

这是一个非常深刻的问题，触及了金融科技（FinTech）最核心的变革点。

在传统金融中，“交易”（Trade）、“清算”（Clearing）和“结算”（Settlement）是三个时间上分离、由不同机构负责的步骤。而在区块链（特别是比特币）中，这三者被压缩成了一个**原子化（Atomic）**的过程。

我将为你详细拆解这两者的区别，并重点解释比特币的 **UTXO 模型** 是如何实现“转账”的。

---

## 1. 股票市场：清算与结算是分离的

在股票市场（如 A 股或美股），你点击“买入”的那一瞬间，股票并没有真正到你手里，钱也没有真正到卖家手里。

### (1) 清算 (Clearing) —— “算账”

- **时间：**交易日收盘后（T+0 晚间）。
- **核心任务：**计算净额（Netting）。
- **过程：**中央对手方（CCP，如中国证券登记结算公司）充当所有买家的卖家、所有卖家的买家。它不需要处理每一笔散户的流水，而是计算每个券商当天“一共买多少、卖多少”，最后得出一个净值。
  - **例子：**券商 A 的客户们总共买了 10 亿茅台，卖了 8 亿茅台。清算结果是：券商 A 需要支付 2 亿现金，接收等值股票。
- **目的：**降低系统风险，减少资金流转量。

### (2) 结算 (Settlement) —— “交割”

- **时间：**交易日之后（如美股 T+1，A 股 T+1）。
  - **核心任务：**资产置换。
  - **过程：**真正的“一手交钱，一手交货”。资金从银行账户划转，股票的所有权在中央存管机构（CSD）的数据库中变更。
  - **痛点：**因为有时间差（T+1），存在**对手方风险**（Counterparty Risk），即买家可能在结算前破产了。
- 

## 2. 加密货币：交易即结算 (Trade is Settlement)

在比特币网络中，**没有**中央清算机构，**也没有** T+1 的概念。

- **原子性：**一旦矿工打包了你的交易，并且该区块被全网确认（例如 6 个区块确认），这笔交易就**同时也完成了清算和结算**。
  - **不可逆：**一旦“结算”完成（上链），除非回滚区块链（51% 攻击），否则无法撤销。
- 

## 3. 硬核解析：BTC 是如何“离开”和“进入”钱包的？

这是初学者最大的误区。

**核心概念：**比特币**从来没有**“离开”过任何人的钱包，也**从来没有**“进入”过别人的钱包。比特币**只存在于区块链的公共账本（UTXO 集合）上**。

**钱包 (Wallet)** 其实只是一个“钥匙包”和“浏览器”。它不存币，它只存你的私钥。

比特币的转账机制是基于 **UTXO (Unspent Transaction Output, 未花费交易输出)** 模型。这和你银行账户的“余额模型”(Balance Model) 完全不同。

## 比喻：熔化金币

想象比特币不是数字，而是一块块金属。你不能从一块 10 克的金子里“转出” 3 克。你必须把这块 10 克的金子**熔化** (Input)，重新铸造出一块 3 克的金子（给别人）和一块 7 克的金子（找零给自己）。

## 技术流程：Alice 转 1 BTC 给 Bob

假设 Alice 的钱包里显示有 1 BTC。这实际上意味着：区块链上有一笔之前的交易 (UTXO)，上面写着“拥有私钥 A 的人可以使用这 1 BTC”。

### Step 1: 构造交易 (Transaction Construction)

- **输入 (Input):** Alice 的钱包找到那个 1 BTC 的 UTXO。Alice 用她的**私钥**对这个 UTXO 进行签名，解锁它。这相当于说：“我证明我有权处理这块金子，我现在要把它熔了。”
- **输出 (Output):** Alice 构造一个新的 UTXO。这个新 UTXO 上面写着：“拥有 **Bob 公钥哈希 (即地址)** 的人可以使用这 1 BTC。”

### Step 2: 广播与验证 (Broadcasting & Validation)

- Alice 将这笔交易广播给矿工。
- 矿工验证：
  1. 签名对不对？（用 Alice 的公钥验证，证明是 Alice 操作的）
  2. 这个 UTXO 之前有没有被花过？（防止双花）

### Step 3: 上链 (On-chain Confirmation)

- 矿工把这笔交易打包进区块。
- **关键时刻：**
  - Alice 引用的那个旧 UTXO 在数据库中被标记为“已花费 (Spent)”（也就是死掉了，不能再用了）。
  - 一个新的 UTXO 被创建出来，归属权是 Bob。

### Step 4: 钱包更新 (Client Update)

- **Alice 的钱包：**扫描区块链，发现那个 1 BTC 的 UTXO 已经被标记为“已花费”，于是余额显示减少。
- **Bob 的钱包：**扫描区块链，发现了一个新的 UTXO，且脚本规定“只有 Bob 的私钥能解锁”。于是，钱包把这个 UTXO 的金额加到显示余额中。

## 总结

- **股票：**所有的变更只是中央数据库里两个数字的加减，且由两拨人（清算所、结算所）分两天完成。
- **比特币：**是一次所有权的销毁与重生。
  - “离开” = 你的私钥解锁了一个旧的 UTXO，并将其标记为废弃。

- “进入” = 链上生成了一个新的 UTXO，被加密锁住，但这把锁只有对方的私钥能开。

这个 **UTXO 模型** 非常适合用来在 Jupyter Notebook 中演示“数据结构”部分，你可以画一个 `Inputs -> Transaction -> Outputs` 的流向图，这比枯燥的代码更有说服力。

## 代码示例1——股票模式vs比特币模式

```
In [1]: import json
import uuid

print("--- 🏠 Part 1: 传统账户模型 (Account Model) ---")
print("逻辑: 类似于银行或股票清算。钱不动，只是数据库里的数字变了。\\n")

class CentralBank:
    def __init__(self):
        # 中心化账本: 只记录“谁有多少钱”
        self.ledger = {
            "Alice": 10.0,
            "Bob": 0.0
        }

    def transfer(self, sender, receiver, amount):
        if self.ledger[sender] >= amount:
            # 1. 修改发送方余额
            self.ledger[sender] -= amount
            # 2. 修改接收方余额
            self.ledger[receiver] += amount
            print(f"✓ [交易成功] {sender} -> {receiver}: {amount}")
        else:
            print("✗ 余额不足")

    def show_ledger(self):
        print(f"当前账本状态: {json.dumps(self.ledger, indent=2)}")

# --- 运行模拟 ---
bank = CentralBank()
bank.show_ledger()
bank.transfer("Alice", "Bob", 3.0) # Alice 转 3 块钱给 Bob
bank.show_ledger()

print("\n" + "="*50 + "\n")

print("--- 💰 Part 2: 比特币 UTXO 模型 (Unspent Transaction Output) ---")
print("逻辑: 没有账户余额。只有“未花费的交易输出”。")
print("转账 = 销毁旧的 UTXO (Input) -> 创建新的 UTXO (Output)\\n")

class UTXO:
    def __init__(self, owner, amount):
        self.id = str(uuid.uuid4())[:8] # 简化的交易 ID
        self.owner = owner
        self.amount = amount

    def __repr__(self):
        return f"<UTXO id={self.id} | Owner={self.owner} | Amt={self.amount}>"

class BitcoinNetwork:
    def __init__(self):
        # 这里的“账本”是一堆散乱的 UTXO 集合
```

```

self.utxo_set = []

def mint(self, owner, amount):
    """挖矿：凭空产生一个 UTXO"""
    new_coin = UTXO(owner, amount)
    self.utxo_set.append(new_coin)
    return new_coin

def get_balance(self, owner):
    """计算余额：必须遍历整个账本，把属于你的 UTXO 加起来"""
    balance = 0
    for utxo in self.utxo_set:
        if utxo.owner == owner:
            balance += utxo.amount
    return balance

def transfer(self, sender, receiver, amount, input_utxos):
    """
    发起交易
    sender: 发送者
    receiver: 接收者
    amount: 想转多少钱
    input_utxos: 打算使用的“钞票”（必须属于发送者）
    """

# 1. 计算投入的总金额
total_input = sum([u.amount for u in input_utxos])

if total_input < amount:
    print("X 资金不足以支付!")
    return

print(f"✉️ [交易开始] {sender} 想要转 {amount} BTC 给 {receiver}")
print(f"    消耗(销毁)旧 UTXO: {input_utxos}")

# 2. 【核心步骤】销毁：从全网有效 UTXO 集合中移除这些 Input
# 这就是“离开钱包”的本质：它们再也无法被使用了
for utxo in input_utxos:
    if utxo in self.utxo_set:
        self.utxo_set.remove(utxo)

# 3. 【核心步骤】重生：创建新的 UTXO
new_utxos = []

# Output 1: 给 Bob 的钱
payment = UTXO(receiver, amount)
self.utxo_set.append(payment)
new_utxos.append(payment)

# Output 2: 给 Alice 的找零 (Change)
change_amt = total_input - amount
if change_amt > 0:
    change = UTXO(sender, change_amt)
    self.utxo_set.append(change)
    new_utxos.append(change)

print(f"    生成(铸造)新 UTXO: {new_utxos}")
print("✓ 交易完成，清算与结算同步结束。")

# --- 运行模拟 ---
btc_net = BitcoinNetwork()

# 1. 初始状态：Alice 挖矿得到一个 10 BTC 的 UTXO
alice_coin = btc_net.mint("Alice", 10.0)

```

```

print(f"初始状态: Alice 余额 = {btc_net.get_balance('Alice')}")
print(f"当前全网 UTXO: {btc_net.utxo_set}\n")

# 2. 转账: Alice 转 3 BTC 给 Bob
# Alice 必须把这整块 10 BTC 的 UTXO 拿出来熔掉
btc_net.transfer("Alice", "Bob", 3.0, input_utxos=[alice_coin])

print(f"\n最终状态:")
print(f"Alice 余额 = {btc_net.get_balance('Alice')} (这是找零的 7 BTC)")
print(f"Bob 余额 = {btc_net.get_balance('Bob')} (这是收到的 3 BTC)")
print(f"当前全网 UTXO: {btc_net.utxo_set}")

```

--- 🏠 Part 1: 传统账户模型 (Account Model) ---

逻辑: 类似于银行或股票清算。钱不动, 只是数据库里的数字变了。

```

当前账本状态: {
    "Alice": 10.0,
    "Bob": 0.0
}
 [交易成功] Alice -> Bob: 3.0
当前账本状态: {
    "Alice": 7.0,
    "Bob": 3.0
}
=====
```

--- 💰 Part 2: 比特币 UTXO 模型 (Unspent Transaction Output) ---

逻辑: 没有账户余额。只有“未花费的交易输出”。

转账 = 销毁旧的 UTXO (Input) -> 创建新的 UTXO (Output)

初始状态: Alice 余额 = 10.0

当前全网 UTXO: [<UTXO id=87937988 | Owner=Alice | Amt=10.0>]

```

 [交易开始] Alice 想要转 3.0 BTC 给 Bob
消耗(销毁)旧 UTXO: [<UTXO id=87937988 | Owner=Alice | Amt=10.0>]
生成(铸造)新 UTXO: [<UTXO id=21b829c3 | Owner=Bob | Amt=3.0>, <UTXO id=e754d593 | 
Owner=Alice | Amt=7.0>]
 交易完成, 清算与结算同步结束。

```

最终状态:

Alice 余额 = 7.0 (这是找零的 7 BTC)

Bob 余额 = 3.0 (这是收到的 3 BTC)

当前全网 UTXO: [<UTXO id=21b829c3 | Owner=Bob | Amt=3.0>, <UTXO id=e754d593 | Owner=
Alice | Amt=7.0>]

## 代码示例2——UTXO（未花费交易输出）模型

In [2]: # 模块一: 定义核心数据结构 (Ledger & UTXO)

```

import hashlib
import time
import json

class UTXO:
    """
    UTXO (Unspent Transaction Output)
    想象这是一块具体的“金币”, 上面刻着面值和所有者的名字。
    """

    def __init__(self, owner, amount):
        self.owner = owner
        self.amount = amount
        # 生成一个唯一的ID (模拟交易哈希)
        unique_str = f"{owner} {amount} {time.time()}"

```

```

        self.id = hashlib.sha256(unique_str.encode()).hexdigest()[:8]

    def __repr__(self):
        return f"[ID:{self.id} | Owner:{self.owner} | Amt:{self.amount} BTC]"

class BlockchainLedger:
    """
    区块链账本：全网唯一的数据库
    它不记录“Alice有多少余额”，它只记录“现在有哪些金币还活着(未花费)”。
    """

    def __init__(self):
        self.all_utxos = {} # 存储所有现存的 UTXO

    def mint_utxo(self, owner, amount):
        """创世挖矿：凭空产生金币"""
        utxo = UTXO(owner, amount)
        self.all_utxos[utxo.id] = utxo
        return utxo

    def get_balance(self, owner):
        """通过扫描所有 UTXO 来计算余额（钱包的本质工作）"""
        balance = 0
        my_utxos = []
        for utxo in self.all_utxos.values():
            if utxo.owner == owner:
                balance += utxo.amount
                my_utxos.append(utxo)
        return balance, my_utxos

    def display_state(self):
        print(f"\n{'='*20} urrent全网账本状态 (UTXO Set) {'='*20}")
        if not self.all_utxos:
            print(" (空账本)")
        for utxo in self.all_utxos.values():
            print(f" {utxo}")
        print("=*65 + "\n")

# 初始化账本
ledger = BlockchainLedger()
print(" ✅ 区块链网络已启动。")

```

区块链网络已启动。

In [3]: # 模块二：交易即结算逻辑 (Transaction Logic)

```

def execute_transaction(ledger, sender, receiver, amount):
    print(f" 交易请求: {sender} 转账 {amount} BTC 给 {receiver}...")

    # 1. 【输入 Input】：钱包扫描账本，找到足够的 UTXO (凑钱)
    sender_balance, sender_utxos = ledger.get_balance(sender)

    if sender_balance < amount:
        print(" ❌ 交易失败：余额不足！")
        return

    inputs = []
    input_sum = 0
    for utxo in sender_utxos:
        inputs.append(utxo)
        input_sum += utxo.amount
        if input_sum >= amount:
            break

    print(f" 1. 收集输入 (Input): 找到 {len(inputs)} 个 UTXO 总计 {input_sum} BTC 满足")

```

```

# --- ⚡ 原子操作开始 (Atomic Operation) ---
# 在这一瞬间，清算(计算)和结算(交割)同时发生

# 2. 【销毁】：将旧的 UTXO 从账本中永久移除 (Spent)
for utxo in inputs:
    del ledger.all_utxos[utxo.id]
    print(f" 2. 销毁旧币: {utxo} ➔ 已移除 (Spent)")

# 3. 【输出 Output】：铸造新的 UTXO (结算)
# Output A: 给收款人的钱
payment_utxo = UTXO(receiver, amount)
ledger.all_utxos[payment_utxo.id] = payment_utxo
print(f" 3. 铸造新币 (支付): {payment_utxo} ➔ 归属 {receiver}")

# Output B: 给自己的找零 (Change)
change = input_sum - amount
if change > 0:
    change_utxo = UTXO(sender, change)
    ledger.all_utxos[change_utxo.id] = change_utxo
    print(f" 4. 铸造新币 (找零): {change_utxo} ➔ 归属 {sender}")

# --- ⚡ 原子操作结束 ---

print("✓ 交易上链确认 (Settlement Complete)! \n")

```

```

In [4]: # 模块三：模拟演示 (Demo)

# 1. 创世状态：给 Alice 发 10 BTC
print("--- 1. 创世 (Genesis) ---")
ledger.mint_utxo("Alice", 10)
ledger.display_state()

# 2. 转账：Alice → Bob (3 BTC)
# 这会触发 UTXO 的分裂 (10 → 3 + 7)
print("--- 2. 第一次转账 ---")
execute_transaction(ledger, "Alice", "Bob", 3)
ledger.display_state()

# 3. 再转账：Bob → Carl (2 BTC)
# 这会消耗 Bob 刚才收到的那块 3 BTC 的金币
print("--- 3. 第二次转账 ---")
execute_transaction(ledger, "Bob", "Carl", 2)
ledger.display_state()

# 4. 最终查账
print("--- 📊 最终余额查询 ---")
bal_alice, _ = ledger.get_balance("Alice")
bal_bob, _ = ledger.get_balance("Bob")
bal_carl, _ = ledger.get_balance("Carl")

print(f"Alice: {bal_alice} BTC")
print(f"Bob: {bal_bob} BTC")
print(f"Carl: {bal_carl} BTC")

```

--- 1. 创世 (Genesis) ---

===== urrent全网账本状态 (UTXO Set) =====

Bitcoin [ID:6090fd98 | Owner:Alice | Amt:10 BTC]

--- 2. 第一次转账 ---

beginTransaction 交易请求: Alice 转账 3 BTC 给 Bob...

1. 收集输入 (Input): 找到 1 个 UTXO 总计 10 BTC 准备熔化...
2. 销毁旧币: [ID:6090fd98 | Owner:Alice | Amt:10 BTC] → 已移除 (Spent)
3. 铸造新币 (支付): [ID:a1ed624d | Owner:Bob | Amt:3 BTC] → 归属 Bob
4. 铸造新币 (找零): [ID:f5adc138 | Owner:Alice | Amt:7 BTC] → 归属 Alice

交易上链确认 (Settlement Complete) !

===== urrent全网账本状态 (UTXO Set) =====

Bitcoin [ID:a1ed624d | Owner:Bob | Amt:3 BTC]

Bitcoin [ID:f5adc138 | Owner:Alice | Amt:7 BTC]

--- 3. 第二次转账 ---

beginTransaction 交易请求: Bob 转账 2 BTC 给 Carl...

1. 收集输入 (Input): 找到 1 个 UTXO 总计 3 BTC 准备熔化...
2. 销毁旧币: [ID:a1ed624d | Owner:Bob | Amt:3 BTC] → 已移除 (Spent)
3. 铸造新币 (支付): [ID:7c469cd0 | Owner:Carl | Amt:2 BTC] → 归属 Carl
4. 铸造新币 (找零): [ID:8f374a8a | Owner:Bob | Amt:1 BTC] → 归属 Bob

交易上链确认 (Settlement Complete) !

===== urrent全网账本状态 (UTXO Set) =====

Bitcoin [ID:f5adc138 | Owner:Alice | Amt:7 BTC]

Bitcoin [ID:7c469cd0 | Owner:Carl | Amt:2 BTC]

Bitcoin [ID:8f374a8a | Owner:Bob | Amt:1 BTC]

--- 📈 最终余额查询 ---

Alice: 7 BTC

Bob: 1 BTC

Carl: 2 BTC

In [ ]:

## 区块链的数据存储

在理解了 UTXO 代码实现后，我们需要澄清两个关于区块链网络架构最常见的误解：

1. 存储逻辑：数据到底存哪里？每个区块都包含所有账户余额吗？
2. 激励机制：谁在维护这些数据？如果不发钱，为什么还有人运行节点？

### 1. 数据架构：区块 (Block) vs. 状态 (State)

很多初学者认为区块链的每一个区块都存储了全网所有人的余额 (UTXO Set)。这是错误的。如果这样做，区块大小会呈指数级膨胀，网络带宽将瞬间瘫痪。

### 核心区别

- **区块 (Block) = 增量日志 (Incremental Logs/Journal)**
  - **存储内容**: 只记录过去 10 分钟内发生[动作 \(Action\)](#)。
  - **本质**: 它是只追加 (Append-only) 的历史证据, 存储在硬盘上。
  - **类比**: 银行的流水单 (Alice 转给 Bob 3 元)。
- **UTXO 集合 (Chainstate) = 当前状态 (Current State)**
  - **存储内容**: 记录当前这一刻, 全网所有[未被花费的资金 \(Snapshot\)](#)。
  - **本质**: 它是节点通过重放所有历史区块计算出来的结果, 通常驻留在**内存 (RAM)** 或高速缓存数据库 (如 LevelDB) 中。
  - **类比**: 银行的[实时数据库](#) (Alice 余额 7 元, Bob 余额 3 元)。

## 状态更新流程 (State Transition)

当全节点收到一个新的区块 (Block  $N$ ) 时, 它执行以下算法:

1. **Load (加载)**: 从内存中读取当前的 UTXO 集合。
2. **Validate (验证)**: 检查区块中每一笔交易的输入 (Input), 确认引用的 UTXO 是否存在且未被花费。
3. **Update (更新)**:
  - **销毁**: 从集合中移除被花费的旧 UTXO。
  - **铸造**: 向集合中添加新生成的 UTXO。
4. **Save (保存)**: 内存状态更新为  $State_N$ , 准备验证下一个区块。

结论: 区块只负责“传达指令”, 全节点负责“维护状态”。

---

## 2. 博弈论视角: 全节点的激励机制

在比特币/以太坊 Layer 1 协议中, **仅仅运行全节点 (Full Node)** 是没有代币奖励的。只有贡献算力 (PoW) 或质押资金 (PoS) 并成功打包区块的“矿工/验证者”才能获得奖励。

问题: 既然没钱拿, 为什么全球还有成千上万个全节点在免费打工 (倒贴存储和带宽) ?

这是区块链博弈论中精彩的“**隐性激励 (Intrinsic Incentives)**”:

### (1) 去信任化 (Trustlessness) —— "Don't Trust, Verify"

- **轻钱包用户**: 必须信任第三方服务器告诉你的余额是对的。如果服务器作恶 (如欺骗你收到了钱), 你无法察觉。
- **全节点用户**: 拥有**财务主权**。你的节点亲自验证每一笔交易的密码学签名。如果不跑节点, 你只是在使用别人的银行服务; 跑了节点, **你自己就是银行**。

### (2) 隐私保护 (Privacy)

- 使用第三方节点 (如 MetaMask 默认节点) 会暴露你的**IP 地址** 和 **查询的钱包地址**, 服务商可以据此推断你的身份。
- 运行全节点, 查询只在本地数据库进行, 无需向公网广播你的关注点。

### (3) 业务基础设施 (Infrastructure Cost)

- **交易所/钱包商**: 必须维护全节点以确保充值提现的零延迟和绝对准确。这是开展业务的固定成本 (Operating Expense)。
- **矿工**: 必须是全节点。如果不知道最新的 UTXO 集合，矿工就无法打包合法交易，挖出的区块会被全网拒绝，导致算力浪费。

## (4) 网络的“免疫系统”

虽然协议不发工资，但全节点构成了网络的防御防线。即使有 51% 的矿工联合起来试图修改规则（例如凭空印 1 亿个比特币），全网的诚实节点会直接拒绝并丢弃这些非法区块。

总结：

- **协议层奖励**: 写入权（矿工/出块者）。
- **应用层奖励**: 读取权（安全、隐私、业务能力）。

In [ ]:

In [ ]:

## 关于以太坊网络上的智能合约

这是一个非常深刻的问题！很多初学者甚至资深的开发者容易误解这一点。

你的直觉并没有错：如果区块链真的要“时时刻刻监控历史区块”来触发代码，那么随着区块高度增加，计算量会呈指数级爆炸，网络早就崩溃了。

简短的回答是：**不，智能合约不需要监控历史。它们是“被动”的，只针对“当前状态”进行计算。**

为了给你的听众解释清楚，我们需要引入一个计算机科学中的核心概念：“**状态机**”(State Machine)。

---

### 1. 核心误区纠正：智能合约不是“后台进程”

在传统的服务器开发中，你可能会写一个 Cron Job (定时任务) 或者一个 Daemon (守护进程)，让它在后台 while(true) 循环监控数据。

**但在以太坊 (EVM) 里，没有“后台进程”！**

- **智能合约是“惰性”的**：它们就像躺在硬盘里的代码文件，平时是死的。
- **触发机制**：只有当有人（或者另一个合约）发起一笔\*\*交易（Transaction）\*\*去“戳”它一下时，它才会醒过来，执行一段代码，改变一下数据，然后立刻又“死”过去。

所以，以太坊网络不需要监控历史，它只需要处理**当下这一刻涌进来的交易请求**。

---

### 2. 模型对比：比特币 vs. 以太坊

**比特币：分布式账本 (Distributed Ledger)**

- **核心逻辑**: UTXO (未花费的输出)。
- **关注点: 动作 (Action)**。
- **数据结构**: 它像一本流水账。为了知道你现在有多少钱, 钱包软件需要扫描历史, 把你所有收到的钱 (且没花掉的) 加起来。
- **记录内容**: Alice 给 Bob 转了 1 BTC。

## 以太坊: 分布式状态机 (Distributed State Machine)

- **核心逻辑**: 账户模型 (Account Model) + 世界状态 (World State)。
- **关注点: 结果 (Result)**。
- **数据结构**: 它像一个巨大的数据库 (Merkle Patricia Trie)。每个区块生成后, 全网都会更新并保存一份最新的“快照”。
- **记录内容**:
  1. Alice 的余额 = 10 ETH。
  2. 合约 A 的变量 `count` = 50。
- **执行逻辑**: 当智能合约运行时, 它不需要去翻历史区块 (比如去查 3 年前谁调用了它), 它只需要读取当前的“世界状态”里的变量值。

数学表达 (给数学背景的同学):

$$\sigma_{t+1} \equiv \Upsilon(\sigma_t, T)$$

- $\sigma_t$ : 当前的世界状态 (S\_current)。
- $T$ : 新的交易 (Transaction)。
- $\Upsilon$ : 以太坊虚拟机 (EVM) 的计算函数。
- $\sigma_{t+1}$ : 计算后的新状态 (S\_new)。

结论: 计算只依赖于上一秒的状态和现在的交易, 与历史无关。

---

## 3. 代码演示: 智能合约的一生

为了方便放入 Jupyter Notebook, 我们可以用 Python 类来模拟这个过程。这能让程序员秒懂“状态”和“被动触发”的关系。

见下一个框

---

## 4. 进阶疑惑: 那怎么实现“定期自动扣款”?

如果合约是“被动”的, 那我怎么实现“每个月自动给房东转账”或者“每隔10秒计算一次利息”呢?

这就是 **Keeper** (守门人/外部触发器) 的作用。

- **区块链内部**: 没有定时器, 没有 `setInterval`。
- **解决方案**: 必须有一个链下的机器人 (比如你自己写的一个 Python 脚本, 或者 Chainlink Keepers 服务), 在链下看着时间。
- **流程**:
  1. 链下脚本发现: “哎呀, 今天是1号了。”
  2. 链下脚本发起一笔交易, 调用智能合约里的 `payRent()` 函数。

3. 智能合约被唤醒，检查：“哦， 确实距离上次付钱超过30天了。”

4. 执行转账。

### 一句话总结给听众：

“以太坊不是一台一直在后台跑程序的服务器，它更像是一个**巨大的、全球共享的计算器**。只有当你按下按键（发送交易）时，它才会计算一次，显示结果（更新状态），然后等待下一次按键。”

```
In [5]: print("--- 🚀 以太坊智能合约模拟器 ---")

class SimpleStorageContract:
    def __init__(self):
        # 这就是“状态” (State)
        # 它被保存在区块链最新的区块里，而不是历史记录里
        self.stored_data = 0
        self.owner = "0xAlice"

    def set_value(self, sender, new_value):
        """
        这是一个写入函数（需要发交易触发，消耗 Gas）
        """
        print(f"⚡ [EVM] 收到交易: {sender} 调用了 set_value({new_value})")

        # 1. 逻辑检查
        if new_value < 0:
            print("❌ 错误：值不能为负数，交易回滚！")
            return

        # 2. 修改状态
        print(f"📝 [EVM] 将 stored_data 从 {self.stored_data} 更新为 {new_value}")
        self.stored_data = new_value
        # 此时，新的状态被写入了最新的区块状态树中

    def get_value(self):
        """
        这是一个读取函数（免费，直接读取节点内存中的状态）
        """
        return self.stored_data

# --- 模拟运行 ---

# 1. 部署合约（状态初始化）
contract = SimpleStorageContract()
print(f"当前状态: {contract.stored_data}\n")

# 2. 区块 1001: 没有任何人理这个合约
# 合约处于“休眠”状态，不消耗任何算力
print("... (时间流逝，没有交易，合约在睡觉) ... \n")

# 3. 区块 1005: Bob 发起交易
# 合约被唤醒，执行 set_value，然后再次休眠
contract.set_value("0xBob", 88)
print(f"当前状态: {contract.stored_data}\n")

# 4. 再次查询
# 不需要回溯历史，直接读内存里的 self.stored_data
print(f"查询结果: {contract.get_value()}"")
```

--- 🚗 以太坊智能合约模拟器 ---  
当前状态: 0

... (时间流逝, 没有交易, 合约在睡觉) ...

⚡ [EVM] 收到交易: 0xBob 调用了 set\_value(88)  
📝 [EVM] 将 stored\_data 从 0 更新为 88  
当前状态: 88

查询结果: 88

In [ ]:

In [ ]:

## 加密金融小结 备用

### 1. AMM 智能合约 (自动做市商)

- ⓘ 基本信息
  - 这是去中心化交易所 (DEX) 的核心引擎, 展示了算法如何替代传统交易所的“订单簿”模式。
  - 核心公式:  $x \cdot y = k$  (恒定乘积)。
- 💰 金融原理
  - 自动定价: 不需要人工挂单, 价格由池子中两种资产的 **比例** 自动决定 ( $Price = y/x$ )。
  - 流动性博弈: 利用套利者的贪婪来维持价格与外部市场一致。
- 🔑 核心要点 (风险提示)
  - 无常损失 (**Impermanent Loss**): 当币价剧烈波动时, 做市商 (LP) 手里的资产会被套利者“低买高卖”, 导致相对于持币不动的损失。
  - 滑点 (**Slippage**): 资金池越小, 大额交易造成的某些价格偏移就越大。
  - 抢跑 (**Front-running**): 因为交易在内存池 (Mempool) 可见, 机器人可以在你买入前瞬间先买入, 抬高价格收割你 (三明治攻击)。

### 2. 去中心化借贷 (Decentralized Lending)

- ⓘ 基本信息
  - 展示了链上借贷与银行借贷的本质区别。
  - 核心模式分为: **超额抵押** (主流) 和 **无抵押/不足抵押** (创新/高风险)。
- 💰 金融原理
  - 代码替代信用: 因为没有 KYC (身份认证) 和信用分, 必须遵循 “**抵押物价值 > 借出资金**” 的铁律。
  - 清算机制: 一旦抵押物价格下跌触及警戒线, 智能合约会自动卖出抵押物还债, 执行严格的风险控制。
- 🔑 核心要点
  - **资金池模式 (Pooled)**: 存钱和借钱都是面对一个大资金池, 而不是点对点匹配 (P2P), 资金效率极高。

- **闪电贷 (Flash Loan)**: 一种无抵押贷款。原理是 必须在同一笔交易内完成借款和还款，否则交易回滚。这是区块链独有的“原子性”金融工具。
- 

## 3. MakerDAO (稳定币发行)

- **基本信息**
    - 以太坊上最古老、最经典的“去中心化央行”。
    - **双币机制**: DAI (稳定币, 锚定美元) 和 MKR (治理/权益代币)。
  - **金融原理**
    - **CDP (债仓)**: 类似于链上的“当铺”。用户锁入 ETH 等资产 (抵押)，系统铸造出 DAI 借给用户。
    - **货币创造**: 商业银行通过信贷创造货币，MakerDAO 通过智能合约抵押创造货币。
  - **核心要点**
    - **去中心化**: 不依赖法币托管，仅依赖链上资产 (如 ETH) 的价值支撑。
    - **风险兜底**: 如果发生系统性风险 (抵押物暴跌且来不及清算)，MKR 代币会被增发拍卖来偿还坏账，MKR 持有者是最后的风险承担人。
- 

## 4. RWA (现实世界资产)

- **基本信息**
  - 连接“币圈”与“传统金融”的桥梁。
  - 案例展示 (如 Tether Gold): 每一枚代币对应现实金库中的一盎司黄金。
- **金融原理**
  - **资产映射 (Tokenization)**: 通过法律确权和审计，将线下资产的所有权映射到链上的 Token。
  - **流动性释放**: 将难以分割、运输的实物资产，转化为可编程、可秒级转账的数字资产。
- **核心要点 (反思)**
  - **生息属性**: 黄金本身不生息，但 RWA 化后可因融入 DeFi 协议 (如借贷) 而产生收益。
  - **风险转移**: 虽然链上技术安全，但引入了链下托管方的 **对手方风险** (Counterparty Risk)。
  - **本质**: RWA 可视为链上的资产证券化 (ABS)，但具备全球流通性。

In [ ]: