# 第2章： Bitcoin Script 基础

## Contents



> **参考**: `code/chapter02/`
> **最后更新**: 2025-12-06

---

Bitcoin 的真正创新不仅在于数字签名或去中心化共识，更在于其可编程货币系统。每笔 Bitcoin 交易本质上都是一个定义花费条件的计算机程序。本章探讨使 Taproot 成为可能的基础概念：UTXO 模型和 Bitcoin Script。

```
# 本章环境：bitcoinutils（一次性加载，后续代码块复用）
from bitcoinutils.setup import setup
from bitcoinutils.utils import to_satoshis
from bitcoinutils.transactions import Transaction, TxInput, TxOutput
from bitcoinutils.keys import P2wpkhAddress, P2pkhAddress, PrivateKey
from bitcoinutils.script import Script
```

## 2.1 UTXO 模型：数字现金，而非数字银行

在深入脚本之前，必须理解 Bitcoin 如何表示价值。与维护账户余额的传统银行系统不同，Bitcoin 使用**未花费交易输出（Unspent Transaction Output，UTXO）**模型——一个更像物理现金而非数字银行账户的系统。

# 现金与银行：心智模型

**传统银行（账户模型）**：

- 账户显示余额：$500
- 花费 $350 只需从余额中扣除
- 结果：账户余额更新为 $150
- 无需处理"找零"

**Bitcoin UTXO 模型（现金模型）**：

- 没有"$500 余额"
- 而是拥有具体的"钞票"：一张 $200 和三张 $100
- 要花费 $350，必须提供价值 $400 的钞票（$200 + $100 + $100）
- 收到 $50 找零作为新的"钞票"
- 结果：现在有一张 $100 和一张 $50

这种类似现金的行为是 Bitcoin 设计和安全模型的基础。

# UTXO 模型实践

让我们追踪 Alice 和 Bob 之间的简单交易：

**初始状态**：

- Alice 拥有一个 10 BTC 的 UTXO
- Bob 没有 bitcoin

**Alice 向 Bob 发送 7 BTC**：

1. **交易输入**：Alice 的 10 BTC UTXO（必须完全消费）
2. **交易输出**：
   - 7 BTC 给 Bob（新 UTXO）
   - 3 BTC 找零给 Alice（新 UTXO）
3. **结果**：原始的 10 BTC UTXO 被销毁，创建两个新 UTXO

**UTXO 标识**：每个 UTXO 由 `transaction_id:output_index` 唯一标识

- Bob 的 UTXO：`TX123:0`（7 BTC）
- Alice 的找零：`TX123:1`（3 BTC）

## UTXO 关键属性

**完全消费**：UTXO 必须完全花费——不能部分花费。

**原子创建**：交易要么完全成功（所有输入被消费，所有输出被创建），要么完全失败。

**找零处理**：输入和输出金额之间的任何差异都成为交易费用，除非明确作为找零返回。

**并行处理**：由于每个 UTXO 只能花费一次，多个交易可以并行验证，无需复杂的状态管理。

# 2.2 Bitcoin Script 与 P2PKH 基础

## Bitcoin Script：可编程花费条件

每个 UTXO 不仅包含金额——它还包含一个**锁定脚本（locking script）**（ScriptPubKey），定义可以花费它的条件。要花费一个 UTXO，必须提供一个**解锁脚本（unlocking script）**（ScriptSig），满足这些条件。

## 脚本架构

```
Unlocking Script (ScriptSig) + Locking Script (ScriptPubKey) → Valid/Invalid
```

**锁定脚本（ScriptPubKey）**：

- 附加到每个 UTXO 输出
- 定义花费条件
- 示例："只有能够为公钥 X 提供有效签名的人才能花费"

**解锁脚本（ScriptSig）**：

- 在花费 UTXO 时提供

- 包含满足锁定脚本所需的数据
- 示例："这是我的签名和公钥"

**验证过程**：

- 组合解锁和锁定脚本
- 作为单个程序执行
- 如果最终结果为 TRUE，UTXO 可以被花费

# 基于栈的执行

Bitcoin Script 使用基于栈的执行模型，类似于 Forth 或 PostScript 等编程语言。操作操作一个后进先出（Last-In-First-Out，LIFO）栈：

初始栈：空

```
(empty)
```

PUSH 3

```
3
```

PUSH 5

```
5
3
```

ADD 操作

```
8
```

操作过程：

从栈中弹出两个数字：5（顶部）和 3 执行加法：5 + 3 = 8 将结果 8 推回栈顶

这个简单模型在保持可预测和安全的同时，支持复杂的花费条件。

# P2PKH：基础脚本

支付到公钥哈希（Pay-to-Public-Key-Hash，P2PKH）是 Bitcoin 最基础的脚本类型，也是理解更复杂脚本（如 Taproot 中使用的脚本）的基础。

**P2PKH 锁定脚本**

```
OP_DUP OP_HASH160 <pubkey_hash> OP_EQUALVERIFY OP_CHECKSIG
```

此脚本表示："此 UTXO 可以由任何能够提供哈希到 `pubkey_hash` 的公钥以及来自相应私钥的有效签名的人花费。"

**P2PKH 解锁脚本**

```
<signature> <public_key>
```

花费者提供：

- 证明拥有私钥的数字签名
- 公钥本身（将被哈希和验证）

# 真实示例：Satoshi 向 Hal Finney

让我们检查著名的第一笔 Bitcoin 交易：中本聪（Satoshi Nakamoto）向 Hal Finney 发送 10 BTC。

**交易 ID**：`f4184fc596403b9d638783cf57adfe4c75c605f6356fbc91338530e9831e9e16`

**交易结构**：

- **输入**：Satoshi 的 coinbase UTXO（挖矿获得的 50 BTC）
- **输出**：
  - 10 BTC 给 Hal Finney
  - 40 BTC 找零给 Satoshi

**注意**：此早期交易使用 P2PK（Pay-to-Public-Key）而非 P2PKH，直接在锁定脚本中嵌入公钥。现代 Bitcoin 使用 P2PKH 以获得更好的安全性和空间效率。

# P2PKH 执行（Hal Finney 示例）

锁定：`OP_DUP OP_HASH160 <hash> OP_EQUALVERIFY OP_CHECKSIG`
解锁：`<signature> <public_key>`

流程：sig + pk 压栈 → OP_DUP → OP_HASH160 → 哈希匹配 → OP_CHECKSIG → 1 (TRUE)

```
┌──────────────────────────────────────┐
│ 02898711...8519 (public_key)         │
│ 30440220...914f01 (signature)        │
└──────────────────────────────────────┘
```

3. **OP_DUP**：复制栈顶项（公钥）：

```
┌──────────────────────────────────────┐
│ 02898711...8519 (public_key)         │
│ 02898711...8519 (public_key)         │
│ 30440220...914f01 (signature)        │
└──────────────────────────────────────┘
```

4. **OP_HASH160**：哈希栈顶项：

```
┌──────────────────────────────────────┐
│ 340cfcff...7a571 (hash160_result)    │
│ 02898711...8519 (public_key)         │
│ 30440220...914f01 (signature)        │
└──────────────────────────────────────┘
```

5. **推送预期哈希**：来自锁定脚本：

```
| 340cfcff...7a571 (expected_hash)       |
| 340cfcff...7a571 (computed_hash)       |
| 02898711...8519 (public_key)           |
| 30440220...914f01 (signature)          |
└────────────────────────────────────────┘
```

6. **OP_EQUALVERIFY**：比较栈顶两项，如果相等则移除两者：

```
| 02898711...8519 (public_key)           |
| 30440220...914f01 (signature)          |
└────────────────────────────────────────┘
```

（如果哈希不匹配，脚本失败）

7. **OP_CHECKSIG**：验证公钥和交易的签名：

```
| 1 (TRUE)                               |
└────────────────────────────────────────┘
```

8. **最终检查**：如果栈顶非零，脚本成功。

## P2PKH 安全属性

**哈希原像抗性**：公钥在首次花费前保持隐藏，提供针对 ECDSA 潜在量子攻击的保护。

**签名验证**：密码学证明花费者控制与公钥哈希对应的私钥。

**交易完整性**：签名覆盖交易详情，防止签名后修改。

**重放保护**：签名特定于特定交易，不能重复使用。

# 2.3 实践实现：构建 P2PKH 交易

## 构建真实的测试网 Legacy 到 SegWit 交易

让我们逐步构建完整的 P2PKH 交易，解释每个组件，然后使用真实数据追踪脚本执行。

```
# 示例 1: 构建 P2PKH 交易
# 参考: code/chapter02/01_build_p2pkh_transaction.py

setup('testnet')
private_key = PrivateKey('cPeon9fBsW2BxwJTALj3hGzh9vm8C52Uqsce7MzXGS1iFJkPF4AT
public_key = private_key.get_public_key()
from_address = P2pkhAddress('myYHJtG3cyoRseuTwvViGHgP2efAvZkYa4')
to_address = P2wpkhAddress('tb1qckeg66a6jx3xjw5mrpmte5ujjv3cjrajtvm9r4')

txin = TxInput('34b90a15d0a9ec9ff3d7bed2536533c73278a9559391cb8c9778b7e7141806
txout = TxOutput(to_satoshis(0.00029400), to_address.to_script_pub_key())
tx = Transaction([txin], [txout])
p2pkh_script = from_address.to_script_pub_key()
signature = private_key.sign_input(tx, 0, p2pkh_script)
txin.script_sig = Script([signature, public_key.to_hex()])
signed_tx = tx.serialize()

print(f"Transaction size: {tx.get_size()} bytes")
```

```
Transaction size: 188 bytes
```

# 关键函数

`TxInput(txid, vout)` | `TxOutput(amount, script_pubkey)` | `Transaction([txin], [txout])`

`sign_input(tx, idx, script)` | `Script([sig, pk])` → script_sig

# 真实数据分析和栈执行

TXID: **bf41b47481a9d1c99af0b62bb36bc864182312f39a3e1e06c8f6304ba8e58355**

**ScriptSig**: `473044...8519`（签名 + 公钥）

**ScriptPubKey**: `76a914c5b28d6b...890fb288ac`（OP_DUP OP_HASH160 hash OP_EQUALVERIFY OP_CHECKSIG）

# P2PKH 栈执行简要

```
| sig | → | pk, sig | → OP_DUP → | pk, pk, sig | → OP_HASH160 → 哈希匹配 → OP_CH
```

## 从 P2PKH 到高级脚本

P2PKH 为理解 Bitcoin 的可编程货币系统提供了基础，但这只是开始。相同的原则——基于栈的执行、密码学验证和条件逻辑——支持更复杂的脚本，我们将在后续章节中探讨：

**P2SH（Pay-to-Script-Hash）**：

- 在保持地址简短的同时支持复杂的花费条件
- 将脚本复杂性从区块链转移到花费者
- 包装 SegWit 和多重签名方案的基础

**P2WPKH（Pay-to-Witness-Public-Key-Hash）**：

- SegWit 的 P2PKH 等价物，效率更高
- 将签名数据与交易数据分离
- 减少交易可延展性并支持 Lightning Network

**P2TR（Pay-to-Taproot）**：

- Bitcoin 脚本演化的顶点
- 支持看起来像简单支付的复杂智能合约
- 将 Schnorr 签名与 Merkle 树结合，实现最大灵活性

每次演化都保持向后兼容，同时添加新功能。理解 P2PKH 的栈执行模型至关重要，因为 Taproot 使用相同的基础方法，只是使用更复杂的密码学原语和脚本结构。

在下一章中，我们将深入探讨这些地址类型，检查它们的脚本结构，并理解每个改进如何建立在从 P2PKH 学到的经验之上。

# 章节总结

本章建立了使 Taproot 成为可能的基础概念：

**UTXO 模型**：Bitcoin 将价值表示为离散的、可花费的输出，而非账户余额。每个 UTXO 必须完全消费，创建一个类似现金的系统，支持并行交易验证并消除复杂的状态管理。

**脚本系统**：每个 UTXO 通过锁定脚本（ScriptPubKey）包含可编程的花费条件。要花费一个 UTXO，必须提供一个解锁脚本（ScriptSig），当一起执行时满足这些条件。

**栈执行**：Bitcoin Script 使用简单的基于栈的模型处理条件，操作操作一个后进先出栈以验证花费授权。

**P2PKH 实现**：基础脚本类型通过七步过程演示签名验证和公钥验证：提供签名和公钥、复制、哈希、比较和签名验证。

**实践开发**：使用 `bitcoinutils` 等工具，开发者可以构建、签名和广播 P2PKH 交易，同时理解底层的密码学操作和栈执行。

理解这些概念至关重要，因为 Taproot 建立在这些概念之上，使用相同的基于栈的执行模型，同时引入新的密码学原语和脚本结构。从简单的 P2PKH 脚本到 Taproot 复杂花费条件的旅程，说明了 Bitcoin 从基本数字现金到复杂金融应用平台的演化——同时保持了使 Bitcoin 独特的安全性和简洁性。