第6章：构建真实 Taproot 合约

Contents

> **参考**: `examples/ch06_single_leaf_contract.py`
> **最后更新**: 2025-12-06

为什么 Script Path 改变一切

如果你能在 Bitcoin 上构建一个看起来像简单支付的合约——直到它揭示一个解锁资金的谜题，会怎样?

想象一下：你想创建一个数字悬赏，任何解决谜题的人都能获得奖励，但如果没有人解决，你想收回资金。或者你在销售数字商品，买家在支付后自动获得解锁密钥，但你保留退款控制权。

传统 Bitcoin 脚本要么完全暴露所有条件（损害隐私），要么需要复杂的多重签名设置（低效）。更糟糕的是，即使是简单的单签名支付也可以通过区块链上的交易模式轻松识别。

Taproot 的 Script Path 使这一切变得优雅：复杂的有条件合约在未花费时看起来与普通支付完全相同，只在需要时揭示必要的执行路径。

基于前几章建立的 Taproot 理论基础，我们现在深入探讨 Script Path 支出模式。本章通过实际的有条件支付场景演示如何在 Taproot 的脚本树中实现双路径授权：支持基于秘密的有条件支付和直接密钥持有者控制。

让我们从一个具体的业务场景开始，理解 Taproot Script Path 的实际价值：Alice 想创建一个具有以下特征的有条件支付合约：

- **条件路径**：任何知道秘密词"helloworld"的人都可以认领资金

- **替代路径**：Alice 作为资金所有者，可以随时使用她的私钥收回资金
- **隐私要求**：未花费时，外部观察者无法区分简单支付和复杂合约

这种双路径设计在实际应用中非常常见：

应用场景	描述
数字商品销售	买家在支付后获得解锁密钥；卖家保留退款权限
悬赏任务	谜题解决者认领奖励；未使用的悬赏可由发布者收回
有条件托管	在特定条件下自动释放；否则所有者可以收回
教育激励	学生在正确答案时认领奖励；教师保留管理控制权

在我们的场景中，Alice 的 Taproot 地址包含两个花费路径：

Key Path（协作路径）：

- Alice 使用她的调整后私钥直接签名
- 64 字节 Schnorr 签名，最大效率
- 完全隐私，无脚本信息泄露

Script Path（脚本路径）：

- 使用哈希锁脚本：`OP_SHA256 <hash> OP_EQUALVERIFY OP_TRUE`
- 任何知道原像"helloworld"的人都可以花费
- 需要揭示脚本内容，但不暴露 Key Path 的存在

在深入代码实现之前，我们需要理解 Taproot 的核心开发模式：**Commit-Reveal**。这种模式将成为我们所有后续 Taproot 应用的基础开发模型。

Commit-Reveal 模式概述

Commit 阶段：

- 构建包含多个花费条件的脚本树
- 将脚本树承诺到 Taproot 地址
- 生成"中间地址"或"托管地址"以锁定资金

- **外部方无法知道**具体花费条件

Reveal 阶段：

- 选择一个花费条件进行解锁
- Key Path：使用密钥直接花费（最大隐私）
- Script Path：揭示并执行特定脚本分支
- **只暴露实际使用的条件**，其他条件永远保持私有

这种模式的力量在于：在 Commit 阶段，所有不同复杂度的合约看起来完全相同；在 Reveal 阶段，只需要暴露实际使用的分支。

让我们通过简单的哈希锁案例学习 Taproot 单叶脚本的完整实现流程。基于前面介绍的 Alice 有条件支付场景，我们将实现：

- **哈希锁脚本**：验证秘密词"helloworld"的哈希值
- **单叶结构**：最简单的脚本树，仅包含一个脚本分支
- **双路径支出**：Key Path（Alice 的直接控制）+ Script Path（有条件支出）

Tagged Hash（BIP340）：为不同目的添加标签前缀，防止哈希碰撞。

阶段 1：Commit 阶段

关键技术点分析：

脚本：`a8`=OP_SHA256，`20`=PUSH32，`936a185c...07af`=SHA256("helloworld")，`88`=EQUALVERIFY，`51`=TRUE。

Commit 关键代码：

```
hash_hex = hashlib.sha256(b"helloworld").hexdigest()
tr_script = Script(['OP_SHA256', hash_hex, 'OP_EQUALVERIFY', 'OP_TRUE'])
tree = [[tr_script]]
taproot_address = alice_pub.get_taproot_address(tree)
```

地址：`tb1p53ncq9ytax924ps66z6al3wfhy6a29w8h6xfu27xem06t98zkmvsakd43h`

Key Path

见证 64 字节 Schnorr 签名。需 `tapleaf_scripts` 供 tweak 计算；`script_path=False`。

Script Path 支出

见证顺序：`[preimage, script, control_block]`。单叶控制块 33 字节（version+parity + internal_pubkey，无 Merkle 路径）。

```
cb = ControlBlock(alice_pub, tree, 0, is_odd=taproot_address.is_odd())
tx.witnesses.append(TxWitnessInput(["helloworld".encode().hex(), tr_script.to_
```

交易 68f7c8f0... 见证栈

```
[0] 68656c6c6f776f726c64 (preimage)
[1] a820936a185c...8851 (script)
[2] c150be5fc4...bb4d3 (control_block)
```

```
# 单叶 Taproot 合约实现（btcaaron）
# 参考: examples/ch06_single_leaf_contract.py

from btcaaron import Key, TapTree

alice = Key.from_wif("cRxebG1hY6vVgS9CSLNaEbEJaXkpZvc6nFeqqGT7v6gcW7MbzKNT")

# Commit 阶段：构建单叶脚本树（哈希锁 + Key Path）
program = (TapTree(internal_key=alice)
    .hashlock("helloworld", label="hash")
).build()

print("=== 单叶 Taproot 合约 ===")
print(f"地址:  {program.address}")
print(f"叶子:  {program.leaves}")
print(program.visualize())
print(f"预期地址: tb1p53ncq9ytax924ps66z6al3wfhy6a29w8h6xfu27xem06t98zkmvsakd43h

# Key Path 支出（Alice 直接收回）
tx_key = (program.keypath()
    .from_utxo("4fd83128fb2df7cd25d96fdb6ed9bea26de755f212e37c3aa017641d3d2d2c
    .to("tb1p060z97qusuxe7w6h8z0l9kam5kn76jur22ecel75wjlmnkpxtnls6vdgne", 3700
    .sign(alice)
    .build())
print(f"\nKey Path TXID: {tx_key.txid}")

# Script Path 支出（知道原像的人花费）
tx_hash = (program.spend("hash")
    .from_utxo("9e193d8c5b4ff4ad7cb13d196c2ecc210d9b0ec144bb919ac4314c12406298
    .to("tb1p060z97qusuxe7w6h8z0l9kam5kn76jur22ecel75wjlmnkpxtnls6vdgne", 4000
    .unlock(preimage="helloworld")
    .build())
print(f"Script Path TXID: {tx_hash.txid}")
```

```
=== 单叶 Taproot 合约 ===
地址:  tb1p53ncq9ytax924ps66z6al3wfhy6a29w8h6xfu27xem06t98zkmvsakd43h
叶子:  ['hash']

  Taproot
    |
[hash]

预期地址: tb1p53ncq9ytax924ps66z6al3wfhy6a29w8h6xfu27xem06t98zkmvsakd43h

Key Path TXID: dcb366c4c08fc15f0bf78d8079305a5cd29f3ec769c9afc57baccf7b536e27b5
Script Path TXID: 6602f1948b71521c5dc3010d37d766d4b229109f6190429292cf8dce25737
```

```
# 可运行：解析单叶控制块（33 字节，交易 68f7c8f0... Script Path）
cb_hex = "c150be5fc44ec580c387bf45df275aaa8b27e2d7716af31f10eeed357d126bb4d3"
cb = bytes.fromhex(cb_hex)
print(f"控制块长度: {len(cb)} 字节")
print(f"Internal pubkey: {cb[1:33].hex()[:16]}...")
```

常见错误与调试

见证顺序：✅ `[preimage, script, control_block]`。❌ 勿颠倒。

检查点：脚本一致性、控制块 `is_odd`、原像 UTF-8→hex 编码（`"helloworld"` → `68656c6c6f776f726c64`）。

接下来，让我们观察哈希锁脚本的完整栈执行过程：

执行脚本：`OP_SHA256 OP_PUSHBYTES_32 936a185c...07af OP_EQUALVERIFY OP_PUSHNUM_1`

0. 初始栈状态：加载脚本输入

```
┌──────────────────────────────────────────────────────────────┐
│ 68656c6c6f776f726c64                                         │
│ (preimage_hex: "helloworld")                                 │
└──────────────────────────────────────────────────────────────┘
```

1. OP_SHA256：计算栈顶元素的 SHA256 哈希

OP_SHA256 从栈顶弹出原像数据，计算其 SHA256 哈希，然后将结果推回栈：

```
┌──────────────────────────────────┐
│ 936a185c...07af (computed_hash)  │
└──────────────────────────────────┘
```

（计算过程：SHA256(“helloworld”) = 936a185c...07af）

2. PUSH 32 字节：推送预期哈希值

脚本将预设的预期哈希值推送到栈顶：

```
936a185c...07af (expected_hash)
936a185c...07af (computed_hash)
```

（栈中现在有两个相同的哈希值）

3. OP_EQUALVERIFY：验证哈希值相等

OP_EQUALVERIFY 弹出栈顶两个元素进行比较，如果相等则继续执行，否则脚本失败：

```
(empty_stack)
```

（验证成功：936a185c...07af == 936a185c...07af，两个元素都被消耗）

4. OP_TRUE：推送成功标志

最后，OP_TRUE（OP_PUSHNUM_1）将值 1 推送到栈，标记脚本执行成功：

```
01 (true_value)
```

（脚本执行成功：栈顶是非零值）

通过实际代码实现和链上数据分析，我们可以清楚地看到两种支出方法之间的差异：

Aspect	Key Path	Script Path
Witness Data	1 element (64-byte signature)	3 elements (input+script+control block)
Transaction Size	~153 bytes	~234 bytes
Privacy Level	Complete privacy, zero information leakage	Partial privacy, only exposes used script branch
Verification Complexity	Single Schnorr signature verification	Control block verification + script execution
Fee Cost	Lowest cost	Medium cost (~50% additional overhead)

这种**选择性揭示**设计使 Taproot 能够支持各种复杂的应用场景：数字商品销售、悬赏任务、有条件托管、多方合约等，同时在未使用时保持最大隐私。

与 P2SH 在花费时**所有条件都被揭示**不同，Taproot Script Path 确保只有执行的分支才会被看到。这种根本性转变重新定义了 Bitcoin 合约的隐私模型。

传统脚本局限性：

- 所有花费条件在链上可见
- 复杂合约容易被观察者识别
- 即使未使用的分支也损害隐私

Taproot 的隐私创新：

- 未使用的条件永远保持隐藏
- 复杂合约与简单支付不可区分
- 只揭示执行的逻辑，保持最大隐私

这种隐私优先的设计使 Taproot 成为 Bitcoin 下一代智能合约的基础，其中复杂性不会损害机密性。

通过 Alice 的哈希锁合约案例，我们深入理解了 Taproot 对 Bitcoin 智能合约的革命性方法：

Commit-Reveal 模式的力量

1. **Commit 阶段**：将复杂的有条件逻辑承诺到普通 Taproot 地址，生成中间地址以锁定资金
2. **Reveal 阶段**：根据实际需要选择 Key Path 或 Script Path 支出，只暴露必要信息

技术实现掌握

1. **单叶脚本树**：TapLeaf 哈希直接作为 Merkle 根，无需额外的 Merkle 计算
2. **控制块验证**：通过地址恢复密码学证明脚本合法性
3. **栈执行**：哈希锁通过哈希匹配验证实现有条件支出

开发最佳实践

1. **Tagged Hash 理解**：掌握不同目的的哈希标签，确保安全性
2. **见证数据顺序**：严格遵循 [输入参数, 脚本, 控制块] 顺序
3. **系统化调试**：使用我们的调试流程图排查常见问题
4. **代码一致性**：使用辅助函数确保 commit-reveal 阶段对齐

更大的图景

本章建立的不仅仅是哈希锁实现——它展示了 Taproot 的根本性隐私革命。通过允许复杂合约在执行之前伪装成简单支付，Taproot 在不牺牲用户隐私的情况下改变了 Bitcoin 的能力。

下一步：在下一章中，我们将探索**双叶脚本树**，学习如何在一个 Taproot 地址中组织多个不同的花费条件，引入真实的 Merkle 树计算，并体验 Taproot 脚本树架构的完整力量，用于更复杂的应用场景。