

# 导读

区块链是什么？

区块链属于一种去中心化的记录技术。参与到系统上的节点，可能不属于同一组织、彼此无需信任；区块链数据由所有节点共同维护，每个参与维护节点都能复制获得一份完整记录的拷贝。

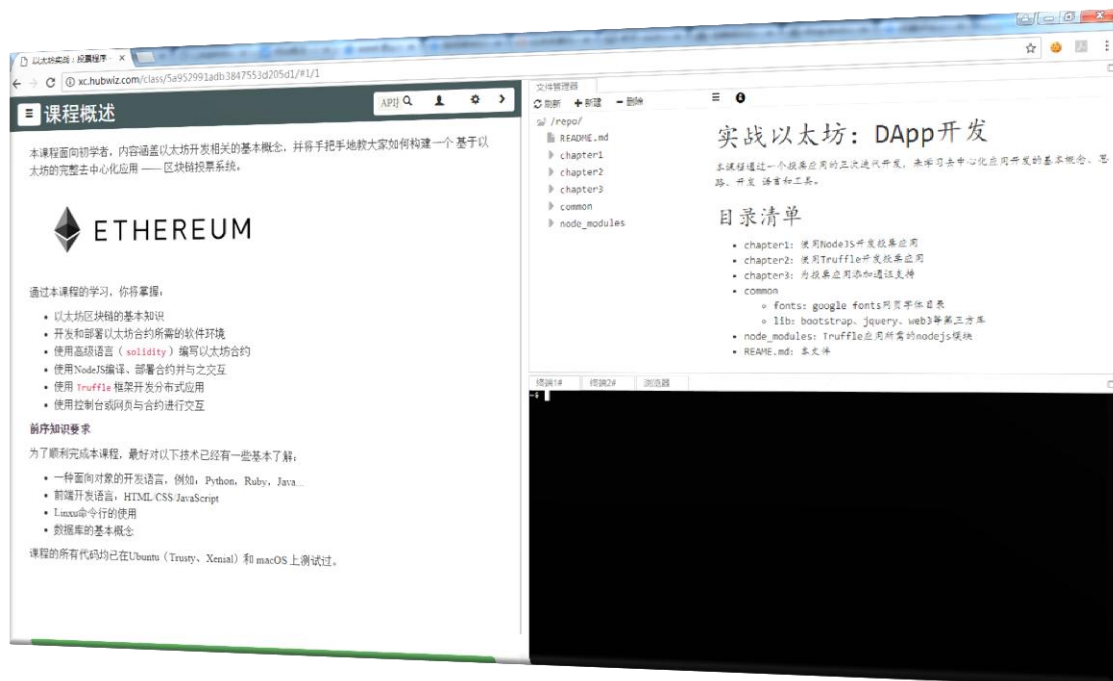
本电子书英文原文由 Nicolas Dorier 和 Bill Strait 联合编写，最早发布于 gitbook ( <https://www.gitbook.com/@programmingblockchain> )，中文版翻译内容由网友郭胜基提供，由汇智网 ( <http://www.hubwiz.com> ) 编目整理，是目前网上流传最广的区块链资料之一。

但由于区块链本身（以及周边生态）的发展非常快，一些实践性内容已经落后于现状。因此编者建议本电子书的读者，在阅读时应注意吸收核心理念思想，而不要过分关注书中的实践操作环节。

为了弥补这一遗憾，汇智网推出了在线交互式以太坊 DApp 实战开发课程，以去中心化投票应用( Voting DApp )为课程项目，通过三次迭代开发过程的详细讲解与在线实践，并且将区块链的理念与去中心化思想贯穿于课程实践过程中，为希望快速入门区块链开发的开发者提供了一个高效的学习与价值提升途径。读者可以通过以下链接访问《以太坊 DApp 开发实战入门》在线教程：

<http://xc.hubwiz.com/course/5a952991adb3847553d205d1?affid=csbcp7878>

教程预置了开发环境。进入教程后，可以在每一个知识点立刻进行同步实践，而不必在开发环境的搭建上浪费时间：



汇智网带来的是一种全新的交互式学习方式，可以极大提高学习编程的效率和学习效果：



汇智网课程内容已经覆盖以下的编程技术：

Node.js、MongoDB、JavaScript、C、C#、PHP、Python、Angularjs、Ionic、React、UML、redis、

mySQL、Nginx、CSS、HTML、Flask、Gulp、Mocha、Git、Meteor、Canvas、zebra、Typescript、Material Design Lite、ECMAScript、Elasticsearch、Mongoose、jQuery、d3.js、django、cheerio、SVG、phoneGap、Bootstrap、jQueryMobile、Saas、YAML、Vue.js、webpack、Firebird，jQuery EasyUI，ruby，asp.net，c++，Express ，Spark.....

# 一、简介

## 1.1 前言

Ayn Rand 在 Fountain Head 与我有共鸣的一篇文章

GAIL WYNAND 是享誉世界富有魅力的木偶大师，而 HOARK HOWARD 是建筑设计行业的领军人物，他们在一起言谈甚欢。GAIL 发现与 HOARK 一起的时候，有种奇妙的放松的感觉，不知道为什么，他于是就问对方。

WYNAND 问：

“HOWARD，你曾经恋爱过吗？”

ROARK 转面直视着他，不假思索地回答：

“我现在就在谈恋爱。”

“但是当走过一幢建筑的时候，你觉得还有什么比它更棒的？”

“棒得多，GAIL”

“我正在思考人们说的一句话，世上无快乐。看看他们多么艰辛地寻找生命中快乐的人。看看他们是如何孜孜以求的。为什么生命止于平淡呢？想想可以通过什么权利，任何人都可以因任何原因人存在于世上，而他自己的快乐除外？人人都需要它，而且人人身上的每一部分都需要它。但是人们从来没有找到它。我在思考为什么。有人喝了酒然后说不知道生命的意义。这是我特别看不起的一部分人。寻找更高级意义或者‘普世目标’的人，不知道为了什么而活着的人，嘟囔着必须需找自我的人。你可以在周围听到尽是他们。那些好像是我们时代的标准陈词滥调。每一本书你打开，都在口沫横飞地自我忏悔。好像忏悔是一件高贵的事情。我却认为它是最令人羞耻的事情。”

“看，GAIL。” ROARK 起身，伸手折断了一棵树上的粗大树枝，双手紧握，一只手在一边；然后，他的手腕和指关节将树枝压弯，慢慢地变成一个拱形。“现在我可以把它做成任何我想要的东西：一张弓，一支矛、一根手杖、一条扶手。那就是生命的意义。”

“你的力量？”

“你的工作。”他把树枝放到一边。“世间提供给你材料，你从中制造东西...”

我想区块链就像是那根树枝。对于局外人而言，感觉像是令人厌烦而毫无用处的数字集合。对于程序员和企业家而言，那就是绝佳的原材料，可以实现我们的梦想。我们赋予

它意义和目标。

你需要先要了解木材的特性，然后才能从一节树枝制造一张弓、一支矛或者一根手杖，同样地，你也需要知道如何编程实现区块链。我希望，你以自己的技能和智慧，将会发现可以在多大程度上利用这些看上去一无是处的数字集合。

我还是需要提醒一下你：学习区块链就好像在黑客帝国中吞下红药片。你将会发现已经无法自己，想尽快辞职转而全职专攻区块链。

本书将带你从区块链的基本应用走向高级应用。它不会教你如何使用 API（比如比特币核心提供的 RPC API），但是它将教你如何编写这样的 API。

拾遗：中本聪曾经把比特币描述成“令人厌烦的灰色。”

面向 API 编程可以帮你快速实现应用，但是开发者的创造性三就被限于 API 了。通过完整理解区块链，开发者将拥有能力充分挖掘开发区块链的潜力。

## 1.2 为什么是区块链编程而不是比特币编程？

区块链是金子，比特币是珠宝。

我们不想把比特币比作金币，更愿意说它是珠宝。因为金子的第一个杀手级应用就是珠宝。金币来得更晚一点。

你不要被蒙了，认为比特币是有瑕疵的，而区块链是珍贵的。如果金子是珍贵的，你会把金项链扔掉吗？区块链是因为比特币而建立并发展起来的。随着区块链价值的增加，更多的比特币也将会应用于区块链，从而增加了对比特币的需求。

你的应用是否使用“比特币作为一种货币”的特性取决于你自己的决策。

区块链是原材料，比特币是燃料。每次人们认为这种燃料也可以作为交易的媒介时，比特币作为货币的特性就出现了。相对于交易价值，你可以让区块链做得更多。你甚至不需要认为它是货币。我们将在本书中向你展示如果使用作为货币的比特币，但那不是全部。

## 1.3 为什么是 C#？

.NET 框架在公司很受欢迎。相信对于创业公司和爱好者而言也是理想的工具。

.NET 框架开发的代码可以在 IOS、Android、Windows 平板/手机、桌面、服务器以及嵌入式设备中运行。

从编译器到运行时内核全部都是开源的

微软创业企业扶植计划允许创业公司使用所有微软工具，包括免费使用价值 150 美元/月的 Azure 云服务。

VisualStudio community2013 是一个专业级别的 IDE，你可以作为爱好者免费使用

C#与 Java 和 C++关系密切。而且，对于已经了解 C 语言的开发人员来说也很容易阅读。

本书作者之一 Nicolas Dorier 为.NET 创建了最受欢迎的比特币框架，名叫 NBitcoin。你可以查看这个网址：<https://github.com/NicolasDorier/NBitcoin>

本书作者拥有超过 15 年的 C#开发综合经验。我们可以将它用于任何项目，无论是出于爱好还是以盈利为目的。

## 1.4 预备条件

### 1.4.1 技能

- 你应该熟悉面向对象和函数式编程
- 如果对 C#有基本的了解当然更好，不过我们认为，对于 Java 或者其它类 C 语言开发人员，本书代码也是清晰易读的。
- 不要求有专业数学知识。我们不涉及超过最基本要求的加密知识，最基本要求仅限于创建一个安全服务。
- 你不必对比特币有深入了解。但是我们推荐阅读的作为额外加分项。

### 1.4.2 工具

- Visual Studio 2013-你可以在 Google Bing 上搜索“Visual studio 2013 community”，然后免费下载使用。
- 比特币核心代码-你需要在开始前进行同步。

拾遗：你可以咨询微软 cortana 或者谷歌 Now 了解比特币的汇率

## 1.5 本书众筹

如果继续为你贡献内容，我们需要购买披萨、咖啡和寿司。我们需要获得足够多的币。并且

如果没有你的反馈，我们将变得漫不经心，以至于无法完成整本书籍的编写工作。

因此，我们启动了下列实验，希望你感兴趣。也许有一天你会将它变成完整的商业模型。

我们创建了以下内容（不需要担心，我们将在后面看到这里每一行的含义）

地址：1KF8kUVHK42XzgcmJF4Lxz4wcl5WDL97PB

签 名 :

H1jiXPzun3rXi0N9v9R5fAWrfEae9WPmIL5DJBj1eTStSvpKdRR8Io6/uT9tGH/3OnzG6ym5yytuWoA9  
ahkC3dQ=

信息：Nicolas Dorier Book Funding Address

现在我们开始编写本书。当我们肚子饿了，我们就暂停，要求得到本书一下章节的资金支持。

你将通过编程完成一个挑战的形式进行汇款，简单地使用钱包汇款将不算数。捐赠人员将通过他们的比特币地址授权获得下一章节。没有任何数字版权管理。如果你没有通过汇款取得了本书，希望你能按照本书的指引汇款。

我们将进入下一章节的详细内容。不要掉以轻心，你需要通过编程学会如何使用它。

你可以在这个网址上得到更多介绍：

<http://blockchainprogramming.azurewebsites.net/>

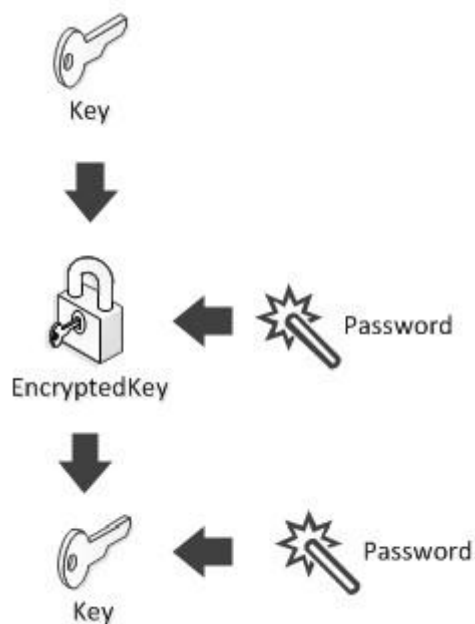
## 1.6 补充阅读

- 以下是阅读本书的辅助材料：
- Mastering Bitcoin ， Andreas M. Antonopoulos 编著
- Nicolas 关于 CodeProject 的文章：<http://www.codeproject.com/Members/NicolasDorier>
- 开发人员手册，网址：<https://bitcoin.org/en/developer-guide>

## 1.7 图标

大部分图标都有一致的形状，它们有箭头辅助阅读，表明产生的目标内容。

举例来说，下列图标应理解为“秘钥 + 密码 = 加密秘钥。加密秘钥 + 密码 = 秘钥。”



虽然代码很好，但是有时候一张图胜过千言万语。不用担心，我也会把代码写下来的。

## 1.8 许可: CC (ASA 3U)

如前面“本书众筹”部分描述的一样，我们将向资金支持者的比特币地址分发本书。

一旦获得本书，你就可以免费分享和修改，具体参照署名-相同方式共享 3.0 未本地化版（CC BY-SA 3.0）。

在免费获得本书后，如果在提醒时给点小费，我们将不胜感激。

加密货币拥趸可能说：股权证明和工作量证明是最好的感情表达方式，其它都是浮云。



**You are free to:**

**Share** — copy and redistribute the material in any medium or format

**Adapt** — remix, transform, and build upon the material

for any purpose, even commercially.

The licensor cannot revoke these freedoms as long as you follow the license terms.

**Under the following terms:**

**Attribution** — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.



**ShareAlike** — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

**No additional restrictions** — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

**Notices:**

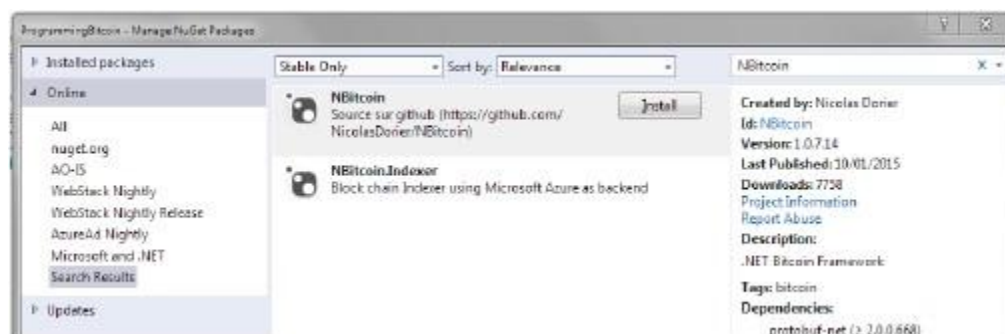
You do not have to comply with the license for elements of the material in the public domain or where your use is permitted by an applicable exception or limitation.

No warranties are given. The license may not give you all of the permissions necessary for your intended use. For example, other rights such as publicity, privacy, or moral rights may limit how you use the material.

## 1.9 项目设置

在开始前，我们想告诉你项目的设置要求。

1. 打开 Visual Studio，创建一个新的 ConsoleApplication。命名为“ProgrammingBlockchain”。
2. 在 SolutionExplorer 中右键打开“Reference”，选择“Manage NuGet Packages...”
3. 搜索“NBitcoin”并安装。注意：下图仅供参考。实际版本和发布日期因你阅读本书的时间而不同。



4.在 Solution Explorer 中右键打开“ProgrammingBlockchain”，选择“Add”，然后“NewFolder”，命名文件夹为“Chapters”。

5.右键打开“Chapters”并选择“Add”然后“New Class”。命名这个类为“Chapter1”。在本书的其它章节也如此这般做法。

6.打开“Program.cs”，并添加以下代码。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using System.Text;
using ProgrammingBlockchain.Chapters;

namespace ProgrammingBlockchain
{
    class Program
    {
        static void Main(string[] args)
        {
            //Select the chapter here.
            var chapter = new Chapter1();

            //call the lesson here.
            chapter.Lesson4();

            //this will hold the window open for you to read the output.
            Console.WriteLine("\n\n\nPress enter to continue.");
            Console.ReadLine();
        }
    }
}
```

7.注意“using ProgrammingBlockchain.Chapters;”被添加到正在使用的区块。

8.这时候，Visual Studio 在抱怨“chapter.Lesson1()”不存在。继续阅读本书，我们将创建它。



## 二、比特币传输

### 2.1 比特币地址

你的比特币地址是用来接收别人付款的。你也许知道，钱包软件使用私钥来付款。

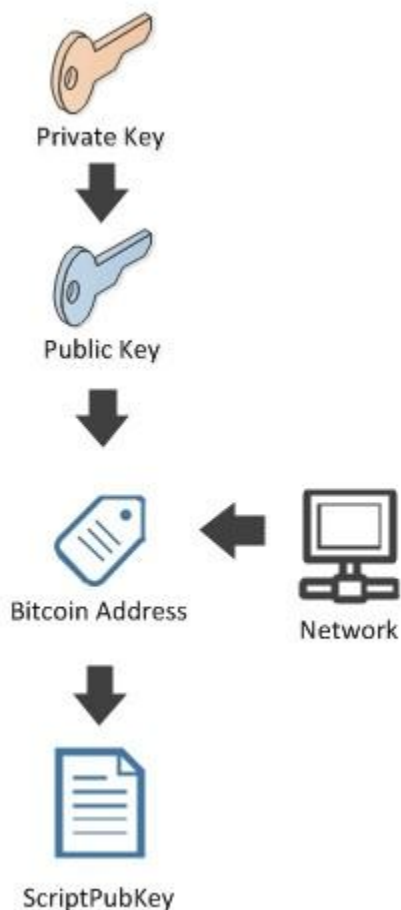
一个比特币地址由两部分组成，一部分是公钥哈希值经过 **Base58check** 编码的组合，另一部分是关于这个网络地址的信息。**Base58check** 编码有一些很精巧的特性，比如用于避免拼写错误的验证码，避免模糊的字符比如“0”和“O”。

拾遗：**TestNet** 是用于开发目的的比特币网络，在这个网上的比特币没有任何价值。**MainNet** 才是人人知道的比特币网络

你也许不知道，就区块链而言，还谈不上比特币地址。内部来说，比特币协议使用 **ScriptPubKey** 验证比特币的接收动作。**ScriptPubKey** 是一段简短的脚本，用于解释在什么情况下才能声明比特币的所有权。随着本书深入分析，我们将考察 **ScriptPubKey** 指令的类型。**ScriptPubKey** 也许包含哈希计算过的公钥，这个公钥允许支付比特币。

拾遗：在 **MainNet** 上进行比特币编程时犯的错误印象更加深刻

下图说明了公钥、私钥、比特币地址和 **ScriptPubKey** 的关系。



现在我们可以用代码向你演示它们的关系了。打开 `Chapter1.cs`，在顶部添加“using NBitcoin;”然后编写下面的方法。

```
public void Lesson1()
{
    Key key = new Key(); //generates a new private key.
    PubKey pubKey = key.PubKey; //gets the matching public key.
    Console.WriteLine("Public Key: {0}", pubKey);
    KeyId hash = pubKey.Hash; //gets a hash of the public key.
    Console.WriteLine("Hashed public key: {0}", hash);
    BitcoinAddress address = pubKey.GetAddress(Network.Main); //retrieves
the
    bitcoin address.
    Console.WriteLine("Address: {0}", address);
    Script scriptPubKeyFromAddress = address.ScriptPubKey;
    Console.WriteLine("ScriptPubKey from address: {0}",
scriptPubKeyFromAddress);
    Script scriptPubKeyFromHash = hash.ScriptPubKey;
    Console.WriteLine("ScriptPubKey from hash: {0}", scriptPubKeyFromHash);
}
```

```
}
```

```
Public Key: 02fc2e5ab52c89f3bfbbbd702254ab9fc7134173e5682b69bf740c3d0bf6bfc4ce
Hashed public key: 1b2da6ee52ac5cd5e96d2964f12a0241851f8d2a
Address: 13Uhw9BmdaXbnjDXiEd4HU4yesj7kKjxCo
ScriptPubKey from address: OP_DUP OP_HASH160 1b2da6ee52ac5cd5e96d2964f12a0241851f8d2a
OP_EQUALVERIFY OP_CHECKSIG
ScriptPubKey from hash: OP_DUP OP_HASH160 1b2da6ee52ac5cd5e96d2964f12a0241851f8d2a
OP_EQUALVERIFY OP_CHECKSIG
```

按 **F5** 检查输出。你刚刚学到了如何创建一个私钥，对应的公钥、公钥哈希、比特币地址和 **ScriptPubKey**。

我们还没有深入细节，注意 **ScriptPubKey** 看上去跟比特币地址没有关系，但是它的确显示了公钥的哈希值。注意我们为何能从比特币地址产生 **ScriptPubKey**？这一步就是所有比特币客户端做的事情，它把人机交互友好的比特币地址翻译成区块链可读的地址。

比特币地址由一个网络识别码和公钥哈希组成。学习到这些，就可以由 **ScriptPubKey** 和网络识别码产生比特币地址，如下面代码所示：

```
public void Lesson2()
{
    Script scriptPubKey = new Script("OP_DUP OP_HASH160
        1b2da6ee52ac5cd5e96d2964f12a0241851f8d2a OP_EQUALVERIFY
OP_CHECKSIG");
    BitcoinAddress address =
scriptPubKey.GetDestinationAddress(Network.Main);
    Console.WriteLine("Bitcoin Address: {0}", address);
}
```

比特币地址：13Uhw9BmdaXbnjDXiEd4HU4yesj7kKjxCo

也可以从 **ScriptPubKey** 取回哈希值，产生一个比特币地址，就像我们在 **Lesson1()** 里面展示的那样。

```
public void Lesson3()
{
    Script scriptPubKey = new Script("OP_DUP OP_HASH160
        1b2da6ee52ac5cd5e96d2964f12a0241851f8d2a OP_EQUALVERIFY
OP_CHECKSIG");
    KeyId hash = (KeyId)scriptPubKey.GetDestination();
    Console.WriteLine("Public Key Hash: {0}", hash);
}
```

```
BitcoinAddress address = new BitcoinAddress(hash, Network.Main);  
Console.WriteLine("Bitcoin Address: {0}", address);  
}
```

公钥哈希: 1b2da6ee52ac5cd5e96d2964f12a0241851f8d2a

比特币地址: 13Uhw9BmdaXbnjDXiEd4HU4yesj7kKjxCo

拾遗: 公钥哈希值的产生过程使, 先将公钥进行 SHA256 哈希计算, 结果再进行 RIPEMD160 哈希计算, 按高位优先记录最终结果。函数看起来是这样的:

RIPEMD160(SHA256(pubkey))

那么现在你应该理解私钥、公钥、公钥哈希、比特币地址和 ScriptPubKey 的关系了。

私钥通常用 Base58Check 编码表示, 叫做比特币密码(也叫钱包导入格式, 简称 WIF), 就像比特币地址那样。

本书的其余部分, 你将使用自己产生的一个地址。



注意很容易就可以从比特币密码产生私钥。特别记住, 从比特币地址产生公钥是不可能的, 因为比特币地址含有公钥哈希但不是公钥本身。

```
public void Lesson4()  
{  
    Key key = new Key();  
    BitcoinSecret secret = key.GetBitcoinSecret(Network.Main);  
    Console.WriteLine("Bitcoin Secret: {0}", secret);  
}
```

比特币密码: KyVVPaNYFWgSCwkvhMG3TruG1rUQ5o7J3fX7k8w7EepQuUQACfwE

复制你得到的比特币密码, 在 Program.cs 的 main 方法中添加如下代码, 代替提供给你的密码。

```
BitcoinSecret paymentSecret = new  
BitcoinSecret("KyVVPaNYFWgSCwkvhMG3TruG1rUQ5o7J3fX7k8w7EepQuUQACfwE");
```

*练习: 注意你自己产生的私钥将用于本身的其余部分, 包括它的地址。*

在本书的其余部分, 我将自己的私钥存储在变量 BitcoinSecret paymentSecret 中。

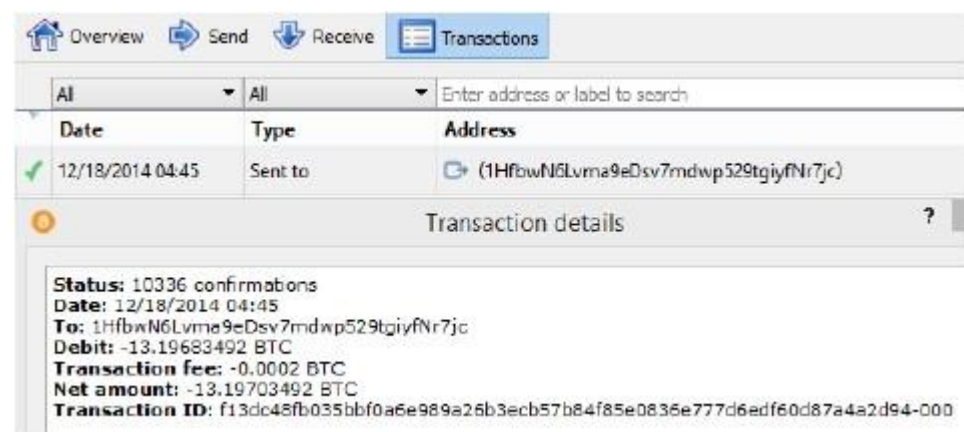
*练习: 取得 paymentSecret 的比特币地址, 存储到 paymentAddress, 在 Bitcoin Core 上发送一些币到上面。比如 0.01 比特币, 感到顺手的时候可以增加一些。*

## 2.2 交易

在开始前, 记住要创建一个新章节的类。

交易是比特币的传输。一个交易可能没有接受者, 或者有好几个。发送者也一样。在区块链上, 发送者和接受者总是被抽象为 ScriptPubKey, 就像我们在第 10 章演示的一样。假设你已经学习完了第 4 课以及相关的练习, 我们继续向前。如果你没有完成练习, 继续前请向先前你生成的地址汇款。

如果你使用 Bitcoin Core, 你的交易标签页将如下显示交易信息:



现在我们关注一下交易 ID。这里它就是:

f13dc48fb035bbf0a6e989a26b3ecb57b84f85e0836e777d6edf60d87a4a2d94

交易 ID 由 SHA256 定义。

不要使用交易 ID 处理未经确认的交易。在正式确认前交易 ID 是可以被更改的, 叫做“交易可塑性”



你可以在类似 **Blockchain.info** 这样网站上检查交易，但是作为一个开发人员，你将需要一个易于查询分析的服务。截止本文时，我们发现 **Blockr.io** 是一个不错的服务。

如果你上

<http://btc.blockr.io/api/v1/tx/raw/f13dc48fb035bbf0a6e989a26b3ecb57b84f85e0836e777d6edf60d87a4a2d94>

看看，就可以看到交易的原始字节。

```
{"status": "success", "data": {"tx": {"hex": "01000000165fcbfb990932ebacd4ef16d202b65077526071b7e9297b4987f9170ac917dbf000000006a473044022069b6b0f1a8d453bdb89e3ad475232b8e01d2851e7b53acab3f830f40e80b3b5102203c049867975360020293c735d48b4a2dda003aa781c1d8ccd2c7af290dcd11de012102e3538427350039e67ea99e935cefb740badf3d09ebc301b0bc9d1bb0301a3417fffffffff02302d89000000000001976a9145b1d720daf0e95e37d0eaedd282b6ed9a40bab7188ac40420f0000000001976a91471049fd47ba2107db70d53b127cae4ff0a37b4ab88ac00000000", "txid": "4ebf7f7ca0a5dafd10b9bd74d8cb93a6eb0831bcb637fec8e8aa
```

NBitcoin 查询 blockr，为你分析信息，所以你就不需要手工来做了。

```
public void Lesson1()
{
    var blockr = new BlockrTransactionRepository();
    Transaction transaction =
        blockr.Get("4ebf7f7ca0a5dafd10b9bd74d8cb93a6eb0831bcb637fec8e8aabf842f1c2688");
    Console.WriteLine(transaction.ToString());
}

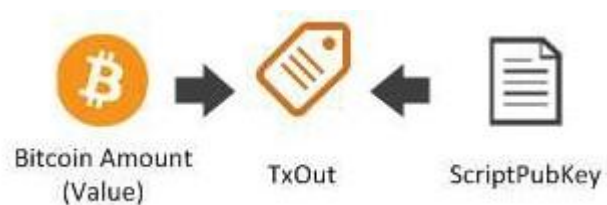
"hash":
"4ebf7f7ca0a5dafd10b9bd74d8cb93a6eb0831bcb637fec8e8aabf842f1c2688",
"ver": 1,
"vin_sz": 1,
"vout_sz": 2,
"lock_time": 0,
"size": 225,
"in": [
{
    "prev_out": {
        "hash":
"bf7d91ac70917f98b497927e1b07267507652b206df14ecdba2e9390b9bffc65",
        "n": 0
    },
    "scriptSig":
"3044022069b6b0f1a8d453bdb89e3ad475232b8e01d2851e7b53acab3f830f40e80b3b5102203c049867975360020293c735d48b4a2dda003aa781c1d8ccd2c7af290dcd11de0102e3538427350039e67ea99e935cefb740badf3d09ebc301b0bc9d1bb0301a3417"
}
],
```

```
"out": [  
  {  
    "value": "0.08990000",  
    "scriptPubKey": "OP_DUP OP_HASH160  
5b1d720daf0e95e37d0eaedd282b6ed9a40bab71  
OP_EQUALVERIFY OP_CHECKSIG"  
  },  
  {  
    "value": "0.01000000",  
    "scriptPubKey": "OP_DUP OP_HASH160  
71049fd47ba2107db70d53b127cae4ff0a37b4ab  
OP_EQUALVERIFY OP_CHECKSIG"  
  }  
]
```

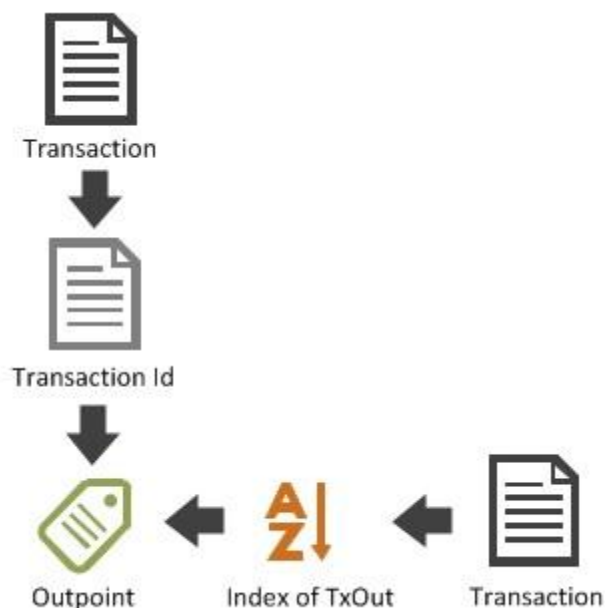
现在相关部分包括 in 和 out。你可以看到 0.0899 比特币被发送给 ScriptPubKey，0.01 比特币被发送到另外一个地方。（练习：验证 ScriptPubKey 中的公钥是否与你付款地址相关联的一样）

如果你注意一下 in，你就发现 prev\_out（先前的 out）被引用了。每一个 in 都告诉你先前的哪一个 out 被使用了，这样就可以为交易付款。术语 TxOut 和 Output 对 Out 是匿名的。

总的来说，TxOut 代表一定数量的比特币和一个 ScriptPubKey。（接收端）



每一个 out 都有一个由交易 ID 和索引定义的地址，叫做 Outpoint。比如，在我的这次交易中 0.01 比特币的 out 对应的 Outpoint 就是(71049fd47ba2107db70d53b127cae4ff0a37b4ab, 1).



现在让我们看看交易中的 in（也叫 TxIn、Inputs）



TxIn 包含正在花出去的 prev\_out 的出口、以及同时被叫做“所有权证明”的 ScriptSig。在我的例子里，prev\_out 的出口是(7def8a69a7a2c14948f3c4b9033b7b30f230308b, 0)

我们代换第一课代码中的交易 ID，就可以检查交易相关的信息。我们可以继续按此方法追踪这个交易 ID 到比特币的 coinbase，也就是他们被挖出来时所在的区块。

在我们的例子里，prev\_out 总使用量为 0.1BTC。在这次交易中，0.0899BTC 和 0.01BTC 都被发出去了。意味着还有 0.0001BTC 没有下落。发送和收到之间的差额被称之为交易费用或者挖矿费用。矿工将指定的交易包含在区块里面，因此需要收费。

## 1.3 区块链

你也许注意到，当我们证明发出去的 TxOut 所有权时，并没有证明 TxOut 的实际存在。这就是区块链的主要作用一显身手的地方：

区块链是所有交易的数据库，记录了第一个比特币交易以来的所有信息，第一个区块又叫做

创始块。区块链在全世界范围内被复制。如果你使用比特币核心，你的电脑上将拥有所有区块链。一旦在区块链上发生交易，那就不能否认它的发生。

矿工只有一个目标，就是在区块链中插入一条交易记录。一组新的交易被添加后，一个区块就被全网广播了。网络上的其它节点确认这个新的区块遵守先前设定的比特币协议。

创建一个区块成本很高。如果一个矿工企图增加一个无效的交易，其它节点将不会认可这个区块，矿工在创建这个区块时花费的投资将打水漂。

一旦矿工提交了一个有效的区块，里面所有的交易都将被确认。当这个发生时，所有矿工都必须放下手中的工作，开始新的交易。一个区块被确认后，它就被写进区块链了。随着后面继续加入新的区块，它被撤销的概率也就越来越低。

在历史上我们第一次拥有的这样一个数据库，它不能被轻易地改写，不再需要信任和审查机制，而且是大范围分布式的。如果我们把比特币当作是一种货币，那区块链就是一套账本。区块链是一个数据库，你给里面的数据赋予含义。你很快就可以发现，一次比特币交易包含的信息远远超过比特币的传输。一次比特币交易在数据库中就是一行记录，永远不可能擦除。作为用户，你可以通过两种不同的方法验证一次交易是否已经记录在区块链里面。

检查整个区块链，它在写作本书时有好几个 GB 大小

寻求一颗 merkel 树，大概有几 K。我们将介绍 merkel 树的更多内容，它与简单支付验证有关联。

## 1.4 区块链不仅仅是比特币

有意思的是，同样的语句，有两组不同人的解读。一组人认为比特币是货币，他们相信比特币的价值将牛气冲天。另一组人并不认为比特币作为货币会取得成功，他们试图解释比特币如此有趣的原因。

但是有一件事情是我们都认同的：一个不可更改的数据库，它不会被审查、干预和擦写，而且在全世界都有副本，它将对其它行业产生巨大的深远影响。

公证员在法庭上记录案件事实，他们就可以将相关文件档案永久地存储在区块链里面。审计将变得自动化而且更有说服力，那时产权归属的存储和交割都在区块链上完成。所有资金周转情况可以公开证明他们的清偿能力。自动化交易脚本可以让交易自动发生，不需要人工干预或者中心权威机构的授权。

本书的其余部分，我们将探讨一些基本原理，这些基本原理支撑了以上技术的运转，以及更

多其它内容。从支付一个比特币开始吧。

## 1.5 支付比特币

你已经了解什么是比特币地址、ScriptPubKey、私钥和矿工了，可以手动完成第一个交易了。创建一个新类 Chapter14 以及一个方法 Lesson1。在你阅读本章节的时候，你需要按照书中所述逐行添加代码创建方法，这个方法将为本书留一个 Twitter 消息风格的反馈。

我们先看看这个交易，它包含了你需要支付的 TxOut，就好像我们在第 11 章做的那样。

```
var blockr = new BlockrTransactionRepository();
Transaction fundingTransaction =
blockr.Get("0b948b0674a3dbd229b2a0b436e0fce8aa84e6de28b088c610d110c2bf54a
cb4");
```

在我们的例子里，我们需要使用第二次 output:

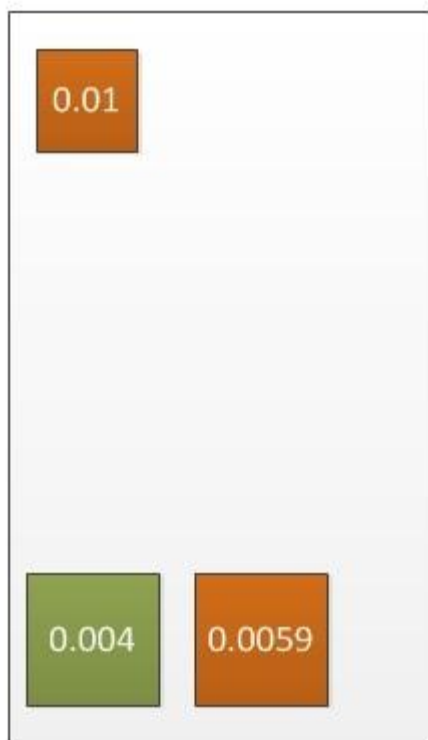
```
"out": [
  {
    "value": "0.08990000",
    "scriptPubKey": "OP_DUP OP_HASH160 5b1d720daf0e95e37d0eaedd282b6ed9a40bab71
OP_EQUALVERIFY OP_CHECKSIG"
  },
  {
    "value": "0.01000000",
    "scriptPubKey": "OP_DUP OP_HASH160 71049fd47ba2107db70d53b127cae4ff0a37b4ab
OP_EQUALVERIFY OP_CHECKSIG"
  }
]
```

为了完成支付，你需要在交易里面提到这个 output。可以如下创建一个交易：

```
Transaction payment = new Transaction();
payment.Inputs.Add(new TxIn()
{
    PrevOut = new OutPoint(fundingTransaction.GetHash(), 1)
});
```

现在我们注意一下 Output。你需要发送 0.004BTC，而你支付了 0.04BTC，所以需要找给你 0.006BTC。同时也需要给矿工一些费用，这样他们好把这次交易添加到他们的下一个区块里面去。

因此，你将拿回来 0.0059BTC。



本书的捐献地址：1KF8kUVHK42XzgcmJF4Lxz4wcl5WDL97PB

```
var programmingBlockchain =  
new BitcoinAddress("1KF8kUVHK42XzgcmJF4Lxz4wcl5WDL97PB");  
payment.Outputs.Add(new TxOut()  
{  
    Value = Money.Coins(0.004m),  
    ScriptPubKey = programmingBlockchain.ScriptPubKey  
});  
payment.Outputs.Add(new TxOut()  
{  
    Value = Money.Coins(0.0059m),  
    ScriptPubKey = paymentAddress.ScriptPubKey  
});
```

现在可以添加你的反馈了。必须少于 40 字节，不然它就崩溃了。

```
//Feedback !  
var message = "Thanks ! :)";  
var bytes = Encoding.UTF8.GetBytes(message);  
payment.Outputs.Add(new TxOut()  
{  
    Value = Money.Zero,  
    ScriptPubKey = TxNullDataTemplate.Instance.GenerateScriptPubKey(bytes)  
});  
Console.WriteLine(payment);
```

```

{
  "hash":
"258ed68ac5a813fe95a6366d94701314f59af1446dda2360cf6f8e505e3fd1b6",
  "ver": 1,
  "vin_sz": 1,
  "vout_sz": 3,
  "lock_time": 0,
  "size": 166,
  "in": [
    {
      "prev_out": {
        "hash":
"4ebf7f7ca0a5dafd10b9bd74d8cb93a6eb0831bcb637fec8e8aabbf842f1c2688",
        "n": 1
      },
      "scriptSig": ""
    }
  ],
  "out": [
    {
      "value": "0.00400000",
      "scriptPubKey": "OP_DUP OP_HASH160
c81e8e7b7ffca043b088a992795b15887c961592
OP_EQUALVERIFY OP_CHECKSIG"
    },
    {
      "value": "0.00590000",
      "scriptPubKey": "OP_DUP OP_HASH160
71049fd47ba2107db70d53b127cae4ff0a37b4ab
OP_EQUALVERIFY OP_CHECKSIG"
    },
    {
      "value": "0.00000000",
      "scriptPubKey": "OP_RETURN
42696c6c20537472616974206973207570646174696e672073637265656e73686f74732e"
    }
  ]
}

```

现在我们就完成创建交易了，我们需要对它签名。换句话说，你需要证明拥有在 input 里面提到的 TxOut。

签名比较复杂，细节可以参考：

[https://en.bitcoin.it/w/images/en/7/70/Bitcoin\\_OpCheckSig\\_InDetail.png](https://en.bitcoin.it/w/images/en/7/70/Bitcoin_OpCheckSig_InDetail.png)。

但是我们可以简单一些。

首先在 `scriptSig` 中插入 `ScriptPubKey`。

`ScriptPubKey` 无非就是 `paymentAddress.ScriptPubKey`，很简单。

```
payment.Inputs[0].ScriptSig = paymentAddress.ScriptPubKey;
//also OK:
//payment.Inputs[0].ScriptSig =
//fundingTransaction.Outputs[1].ScriptPubKey;
payment.Sign(paymentSecret, false);
Console.WriteLine(payment);
```

```
"in": [
  {
    "prev_out": {
      "hash": "4ebf7f7ca0a5dafd10b9bd74d8cb93a6eb0831bcb637fec8e8aabf842f1c2688",
      "n": 1
    },
    "scriptSig":
    "3045022100d4d8d4e1e3205399e0ab899e95bd6c7b65a094c9b86c4b020872428864a5c63502206f9
d0b3084e4a7895de520069a939347909471ca118a43723fd8734cd8e8bcac01
03e0b917b43bb877ccb68c73c82165db6d01174f00972626c5bea62e64d57dea1a"
  }
]
```

恭喜，你可以在第一个交易上完成签名了。你的交易已经准备好了。剩下的就是发布到网上去让矿工们可以看得到。

保证比特币核心正在运行，然后：

```
using (var node = Node.ConnectToLocal(Network.Main)) //Connect to the node
{
    node.VersionHandshake(); //Say hello
    //Advertise your transaction (send just the hash)
    node.SendMessage(new InvPayload(InventoryType.MSG_TX,
payment.GetHash()));
    //Send it
    node.SendMessage(new TxPayload(payment));
    Thread.Sleep(500); //Wait a bit
}
```

`using` 代码段将负责断开与节点的链接。以上就是全部！





你可以直接连到比特币网络

但是，我建议你连接到自己的可信节点（更快更便捷）

## 1.6 作为真实性验证方法的所有权证明

在开始下一章节前，我授予你一个秘钥，用于向我付款。

还记得我说的吗：

**地址:** 1KF8kUVHK42XzgcmJF4Lxz4wcL5WDL97PB

**签名:**

H1jiXPzun3rXi0N9v9R5fAWrfEae9WPmIL5DJBj1eTStSvpKdRR8Io6/uT9tGH/3OnzG6ym5yytuWoA9  
ahkC3dQ=

**消息:** Nicolas Dorier Book Funding Address

这些构成了我拥有这本书私钥的证明。

你可以通过下面的代码验证正确性：

```
var address = new BitcoinAddress("1KF8kUVHK42XzgcmJF4Lxz4wcL5WDL97PB");  
var msg = "Nicolas Dorier Book Funding Address";  
var sig =  
"H1jiXPzun3rXi0N9v9R5fAWrfEae9WPmIL5DJBj1eTStSvpKdRR8Io6/uT9tGH/3OnzG6ym5  
yytuWoA9a  
hkC3dQ=";  
Console.WriteLine(address.VerifyMessage(msg, sig));
```

True

这些就是我将授予你的。

因此如果我给你这个任务：

**消息:** “Prove me you are 1LUtd66PcpPx64GERqufPygYEWBQR2PUN6”

你可以使用私钥按照以下方法证明：

```
msg = "Prove me you are 1LUtd66PcpPx64GERqufPygYEWBQR2PUN6";  
sig = paymentSecret.PrivateKey.SignMessage(msg);  
Console.WriteLine(paymentSecret.GetAddress().VerifyMessage(msg, sig));
```

True

你将被提醒在以下网址证明自己的 ID: <http://blockchainprogramming.azurewebsites.net/>

## 三、关键的存储和数字生成机制

### 3.1 足够随机了吗？

当你调用 `new Key()` 时，实际上是在用 PRNG（伪随机数生成器）来产生私钥。在 windows 平台上，就是用系统提供的 `RNGCryptoServiceProvider`。

在安卓上，我使用 `SecureRandom`，实际上你可以自己实现 `RandomUtils.Random` 然后使用它。

在 IOS 上，我还没用实现，你需要自己实现 `IRandom`。

对一台电脑来说，随机是很难的。但是最大的问题是，几乎不可能知道一串数字是否真的是随机的。

如果有个恶意软件修改了你的 PRNG（当然可以预测你将产生的随机数），当你发现时已经太迟了。

这就意味着跨平台的 PRNG 实现（就好像使用电脑时钟加上 CPU 速度）是危险的，但是当你发现时已经太迟了。

出于效率的考虑，大多数 PRNG 工作方法都是一样的：选择一个随机数，称为种子，然后每次你需要的时候，一个可预测的规则就会为你产生下一个数字。

种子的随机程度由熵的测量来定义，但是熵的大小同时取决于观察者。

假设你从时钟时间产生一个种子，并且你的时钟分辨率可能是 1ms。（实际上是更多，大约 15ms）

如果黑客知道上星期你产生了 key，那么你的种子有  $1000 \times 60 \times 60 \times 24 \times 7 = 604800000$  种可能性。

对这个黑客来说，熵就是  $\text{LOG}(604800000;2) = 29.17 \text{ bits}$ 。

在我的家用电脑上遍历这样的数字用不了 2 秒钟，我们把这种遍历叫“暴力攻击”。

然而，如果你使用时钟时间 + 进程 ID 来产生种子，并且假设有 1024 个不同的进程 ID。

那么，现在黑客就需要遍历  $604800000 \times 1024$  种可能性，大概需要耗时 2000 秒。

然后，如果再加上我打开电脑的时间，就算黑客知道我是在今天打开的话，那也有 86400000 种可能性。

这样的话，黑客需要遍历  $604800000 \times 1024 \times 86400000 = 5.35088 \times 10^{19}$  种可能性。

可是，注意如果黑客攻击了我的电脑，他就可以获得最后的信息片段，减少数字的可能性，从而减少熵。

熵按照  $\text{LOG}(\text{possibilities}; 2)$  来测量，所以  $\text{LOG}(5,35088\text{E}+19; 2) = 65 \text{ bits}$ 。

足够了吗？也许，如果你面临的黑客不知道可能世界里的更多信息的话。

但是既然公钥的哈希是  $20\text{bytes} = 160\text{bits}$ ，比所有地址空间要小，所以你可以做得更好。

注意：增加熵的难度是线性递增的，攻击熵的难度是指数递增的。

一种可以快速产生熵的有趣方法就是人类的干预。（比如移动鼠标）

如果你不完全信任 PRNG 平台（并不奇怪），你可以给 NBitcoin 使用的 PRNG 输出增加熵。

```
RandomUtils.AddEntropy("hello");
RandomUtils.AddEntropy(new byte[] { 1, 2, 3 });
var nsaProofKey = new Key();
```

调用 `AddEntropy(data)` 时 NBitcoin 做的事情如下：

```
additionalEntropy = SHA(SHA(data) ^ additionalEntropy)
```

当你产生一个新数字的时候：

```
result = SHA(PRNG()) ^ additionalEntropy)
```

### 3.1.1 主要的派生函数

然而，数字的可能性并不是最重要的。最重要的是黑客成功打开你密钥的时间。这样 KDF 就来了。

KDF，Key Derivation Function 的简称，是一种取得更加坚固密钥的方法，即便你的熵比较低。

想象一下，你需要产生一个种子，并且黑客知道它存在 10,000,000 种可能性。这样的种子一般情况下很容就被破解了。

但是如果你可以让遍历工作慢一些呢？

一个 KDF 就是一个旨在耗费计算资源的哈希函数，下面是一个例子：

```
var derived = SCrypt.BitcoinComputeDerivedKey("hello", new byte[] { 1, 2, 3 });
RandomUtils.AddEntropy(derived);
```

即使你面临的黑客知道熵源就是 5 个字母，他仍然需要运行 `Scrypt` 以检测可能性，在我的电脑上这需要耗费 5 秒钟。

最后：不信任 PRNG 并不是固执己见，你可以通过增加熵和使用 KDF 来降低攻击的破坏性。

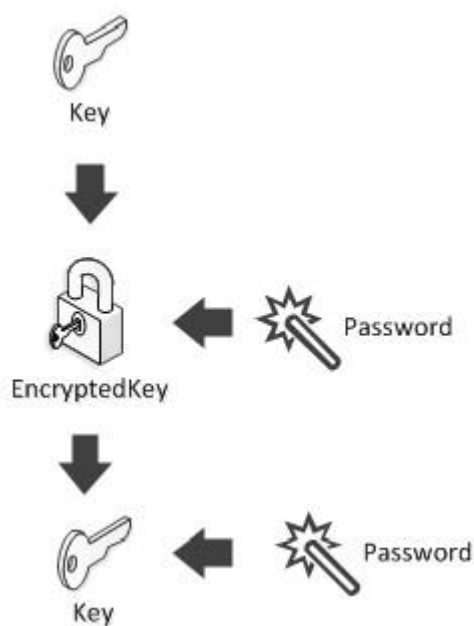
时刻记在心头，黑客可以收集你或者你的系统信息来降低熵。

如果你使用时间戳作为熵源，他就会知道你是上周生成秘钥的，并且你只在上午 9 点到下午 6 点之间使用电脑，这样就可以降低熵。

## 3.2 秘钥加密

在上一节中，我快速过了一下一个特殊的 KDF，名叫 Scrypt。按我说，KDF 的目标就是让暴力破解代价昂贵。

所以，你就不会奇怪，现在已经有一个标准，通过一个 KDF 使用密码对你的私钥进行加密。它就是 BIP38。



```
var key = new Key();
BitcoinSecret wif = key.GetBitcoinSecret(Network.Main);
Console.WriteLine(wif);
BitcoinEncryptedSecret encrypted = wif.Encrypt("secret");
Console.WriteLine(encrypted);
wif = encrypted.GetSecret("secret");
Console.WriteLine(wif);
```

```
L136rry4qogVuBBdcD2WkfsGP5SrdCxDxpcPMx4YtwYErq9mbG7W  
6PYQCwhP9uPh7ESZsV8sUIdBsroZdXuaJmzfYTS6MkEMRdT3pRJbfvu7Ts  
L136rry4qogVuBBdcD2WkfsGP5SrdCxDxpcPMx4YtwYErq9mbG7W
```

这种加密在两种情况下使用：

- 你不信任存储提供商（他们可能受到攻击）
- 你为别人存储秘钥（你不想知道他的秘钥）

如果你的服务器负责秘钥解密，请一定小心，黑客可能使用 DDOS 攻击你的服务器，迫使它解密出很多秘钥。

如果可能的话，让最终用户来解密。

## 3.3 秘钥的生成

### 3.3.1 像过去的美好日子

首先，为什么要生成好几个秘钥呢？

主要原因是为了私密性。既然你可以看到所有地址的余额，最好还是在每次交易时使用新地址。

然而，实际上，你可以为每个合约生成秘钥。因为这是识别你的付款人的一种简单的方法，而且不会泄露太多的隐私。

你可以生成秘钥，就好像刚开始的时候一样：

```
var key = new Key();
```

然而，那样有两个问题：

当你生成一个新的秘钥后，你钱包的所有备份就都过期了。

你不能把地址产生的过程委托给一个不可信的人。

如果你正在开发网络钱包，为你的用户生成秘钥，那么当一个用户被黑时，他立马就会怀疑你。

### 3.2.2 BIP38(part 2)

我们已经看到 BIP38 在秘钥加密方面的内容了，然而这个 BIP 实际上是一篇文章里的两个思想。

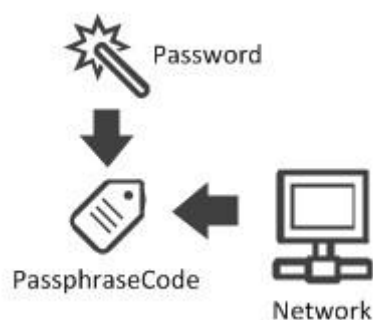
这个 BIP 的第二部分告诉你为何可以将密钥和地址的生成委托给不可信第三方。这就解决了前面的担忧。

这个方法是为密钥生成器生成一个 `PassphraseCode`。通过 `PassphraseCode`，他就可以为你生成加密后的密钥，并且不需要知道你的密码和私钥。

`PassphraseCode` 可以按 WIF 格式提供给你的密钥生成器。

Tip: 在 NBitcoin，所有以 Bitcoin 前缀开头的就是 Base58（WIF）格式数据

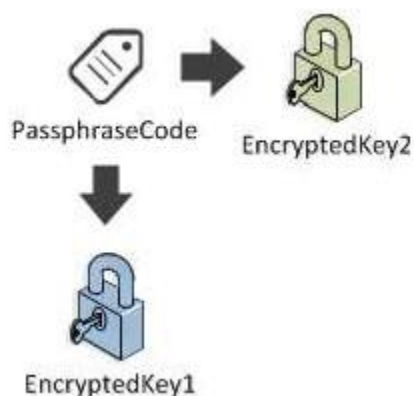
所以，作为一个想要委托生成密钥的用户，首先你需要生成 `PassphraseCode`。



```
BitcoinPassphraseCode passphraseCode = new BitcoinPassphraseCode("my  
secret",  
Network.Main, null);
```

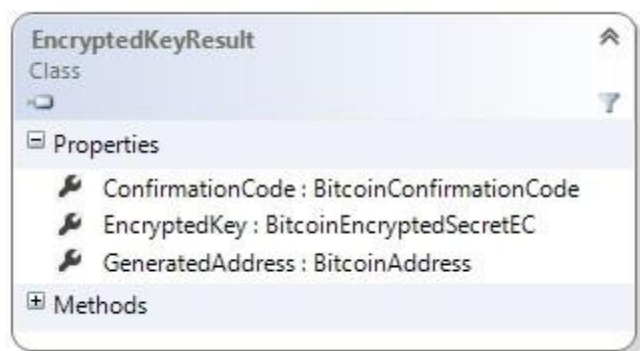
然后你可以将 `PassphraseCode` 提供给第三方密钥生成器。

密钥生成器将为你生成新的加密密钥。



```
EncryptedKeyResult encryptedKey1 =  
passphraseCode.GenerateEncryptedSecret();
```

`EncryptedKeyResult` 信息量丰富：



首先：生成的比特币地址，然后是 **EncryptedKey**，就像我们在密钥加密部分看到的那样，最后仍然很重要是 **ConfirmationCode**，这样第三方机构就可以证明生成的密钥和地址与你的密码是相对应的。

```
EncryptedKeyResult encryptedKey1 =
passphraseCode.GenerateEncryptedSecret();
Console.WriteLine(encryptedKey1.GeneratedAddress);
Console.WriteLine(encryptedKey1.EncryptedKey);
Console.WriteLine(encryptedKey1.ConfirmationCode);
```

```
1PjHUAuSnZjLRoJrC9HnhsGj4RdDCoTFUm
6PnUmv2YPEnb6NtzqQmE9o6M5w8xAeod94VyzhhGPe9qaT63Rxzk9VLUmS
cfm38VUVzCzhvJGAX22WtNdtmNwsCBHNj6Bx82QfW7NEUHRMBANaxDZxPCdXzjFouQXQLiskvF
```

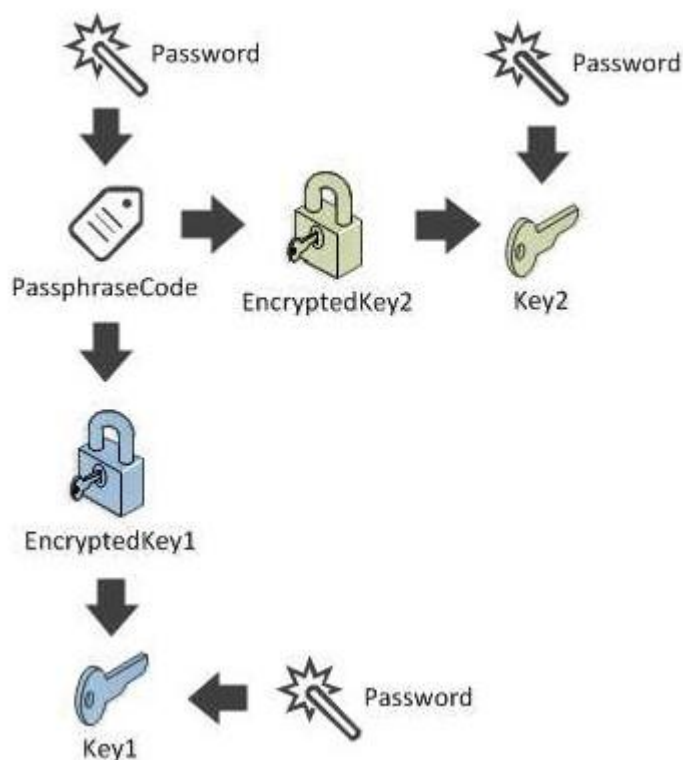
作为所有者，你一旦收到这些信息，就需要检查一下，确保密钥生成器没有使用 **ConfirmationCode.Check** 来骗人，然后你就根据密码取得了私钥。

```
Console.WriteLine(confirmationCode.Check("my secret", generatedAddress));
BitcoinSecret privateKey = encryptedKey.GetSecret("my secret");
Console.WriteLine(privateKey.GetAddress() == generatedAddress);
Console.WriteLine(privateKey);
```

```
True
True
L58tYn6FcC6Ra6iqTMggCaZupCgbRMdfvFjNURKj5D9zgq6g685Z
```

所以，我们看到了，第三方机构可以为你生成加密密钥，无需知道你的密钥和私钥。





然而，仍然存在一个问题：

- 当你生成一个密钥后，你钱包的所有备份都过期失效了。BIP32，或者叫分层确定钱包（HD 钱包）推荐了另一种解决方案，得到了更广泛的支持。

### 3.2.3 HD 钱包(BIP 32)

谨记我们需要解决的问题：

- 避免备份过期失效
- 委托不可信第三方生成密钥/地址

一个“确定的”钱包可以解决我们备份的问题。通过这样的钱包，你只需要保存种子。从这个种子，就可以多次生成同样系列的私钥。

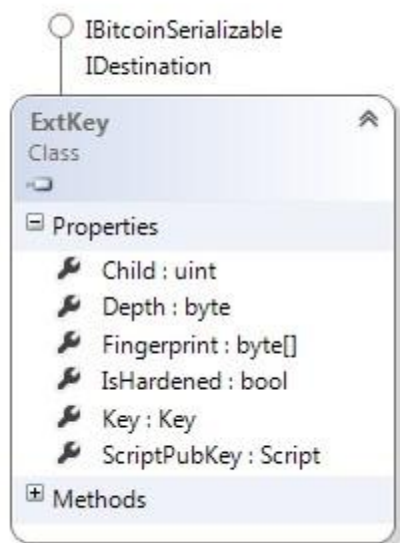
这就是“确定的”含义。你可以看到，从主密钥出发，我可以生成新的密钥：

```
ExtKey masterKey = new ExtKey();
Console.WriteLine("Master key : " + masterKey.ToString(Network.Main));
for (int i = 0 ; i < 5 ; i++)
{
    ExtKey key = masterKey.Derive((uint)i);
    Console.WriteLine("Key " + i + " : " + key.ToString(Network.Main));
}
```

```
Master key :
xprv9s21ZrQH143K3JneCAiVkz46BsJ4jUdH8C16DccAgMVfy2yY5L8A4XqTvZqCiKXhNWFZX
dLH6VbsCs
qBFsSXahfnLajiB6ir46RxgdkNsFk
Key 0 :
xprv9tvBA4Kt8UTuEW9Fiuy1PXPWWGch1cyzd1HSAz6oQ1gcirnBrDxLt8qsis6vpNwmSVtLZ
XWgHbqff9
rVeAErb2swwzky82462r6bWZAW6Ty
Key 1 :
xprv9tvBA4Kt8UTuHyzrhkRWh9xTavFtYowhZTopNHGJSe3KomssRrQ9MTAhVWKfp4d7D8Cgm
T7TRza
uoAZXp3xwHQfxr7FpXfJKpPDUtiLdmcF
Key 2 :
xprv9tvBA4Kt8UTuLoEZPpW9fBEzC3gfTdj6QzMp8DzMbAeXgDhHSMmdnxSFHCQXycFu8FcqT
JRm2ka
mjeE8CCKzbiXyoKWZ9ihif7J5JicgaLU
Key 3 :
xprv9tvBA4Kt8UTuPwJQyxuZoFj9hcEMCoz7DAWLkz9tRMwnBDiZghWePdD7etfi9RpWEWQjK
CM8wH
vKQwQ4uiGk8XhdKybzB8n2RVuruQ97Vna
Key 4 :
xprv9tvBA4Kt8UTuQoh1dQeJTXsmmTFwCqi4RXWdjBp114rJjNtPBHjxAckQp3yeEFw7Gf4gp
nbwQTgDp
GtQgcN59E71D2V97RRDtxeJ4rVkw4E
Key 5 :
xprv9tvBA4Kt8UTuTdiEhN8iVDr5rfAPSVsCKpDia4GtEsb87eHr8yRVveRhkeLEMvo3XWL3G
jzZvncfWVK
nKLWUMNqSgdxoNm7zDzzD63dxGsm
```

你仅仅需要保存好 **masterKey**，因为由此可以多次生成同样的私钥套装。

你会发现，这些秘钥都是 **ExtKey**，不是你熟悉的秘钥。然而，先别着急，因为在里面有真实的私钥：



在 base58 格式上与 ExtKey 等价的就叫 BitcoinExtKey。

但是怎样才能解决我们的第二个难题呢：委托地址生成工作给第三方，并可能被黑（就像是支付服务器）？

技巧就是你可以让主密钥偏中性一点，这样你就有主密钥的公开版本（不含私钥）。从中性版本的角度，第三方可以生成公钥而不需要知道私钥。

```

ExtPubKey masterPubKey = masterKey.Neuter();
for (int i = 0 ; i < 5 ; i++)
{
    ExtPubKey pubkey = masterPubKey.Derive((uint)i);
    Console.WriteLine("PubKey " + i + " : " + pubkey.ToString(Network.Main));
}

```

```

PubKey 0 :
xpub67uQd5a6WCY6A7NZfi7yGoGLwXCTX5R7QQfMag8z1RMGoX1skbXAeB9JtkaTiDoeZPprGH1drvY
cviXKppXtEGSVwmmx4pAdisKv2CqoWS

PubKey 1 :
xpub67uQd5a6WCY6CUeDMBvPX6QhGMMoMMNKhEzt66hrH6sv7rxujt7igGf9AavEdLB73ZL6ZRJTRnhy
c4BTiWeXQZFu7kyjwtdG9tjRcTZunfeR

PubKey 2 :
xpub67uQd5a6WCY6Dxbqk9Jo9iopKZUqg8pU1bWXbnesppsR3Nem8y4CVFjKnzBUkSVLGK4defHzKZ3jj
AqSzGAKoV2YH4agCAEzzqKzeUaWJMW

PubKey 3 :
xpub67uQd5a6WCY6HQKya2Mwwb7bpSNB5XhWCR76kRaPxchE3Y1Y2MAiSjhRGftmeWyX8cJ3kL7LisJ
3s4hHDWvhw3DWPpEtkihPpofP3dAngh5M

PubKey 4 :
xpub67uQd5a6WCY6JddPfiPKdrR49KYEuXUwwJJ5L5rWGDDQkpPctdkrwMhXgQ2zWopsSV7buz61e5
mGSYgDisqA3D5vyvMtKYP8S3EiBn5c1u4

```

想象一下，你的支付服务器生成了 **pubkey1**，你可以通过主私钥得到对应的私钥。

```
masterKey = new ExtKey();
masterPubKey = masterKey.Neuter();

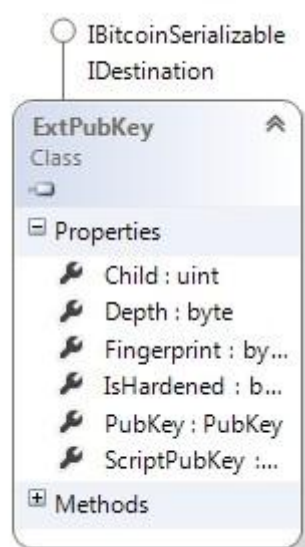
//The payment server generate pubkey1
ExtPubKey pubkey1 = masterPubKey.Derive((uint)1);

//You get the private key of pubkey1
ExtKey key1 = masterKey.Derive((uint)1);

//Check it is legit
Console.WriteLine("Generated address : " +
pubkey1.PubKey.GetAddress(Network.Main));
Console.WriteLine("Expected address : " +
key1.PrivateKey.PubKey.GetAddress(Network.Main));
```

Generated address : 1Jy8nALZNqpf4rFN9TWG2qXapZUBvquFfX  
Expected address : 1Jy8nALZNqpf4rFN9TWG2qXapZUBvquFfX

**ExtPubKey** 与 **ExtKey** 相似，除了它包含的是 **PubKey** 而非 **Key** 之外。



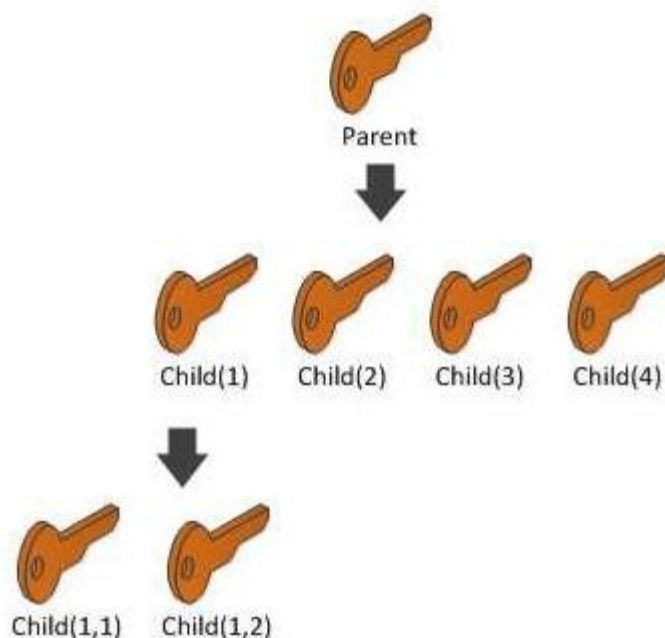
现在我们就知道了，确定密钥解决问题的过程。让我们看看分层是什么意思。

在前面的练习中，我们已经知道，通过联合 **master key + index** 我们可以生成另一个密钥。

我们称这个过程为派生，主密钥是 **parent key**,生成的密钥叫 **child key**。

然后，你也可以从子密钥中派生子密钥，这就是分层的含义。

这就是为什么从概念上来讲，你可以更一般地说：**Parent Key + KeyPath => Child Key**



在上图中，你可以通过

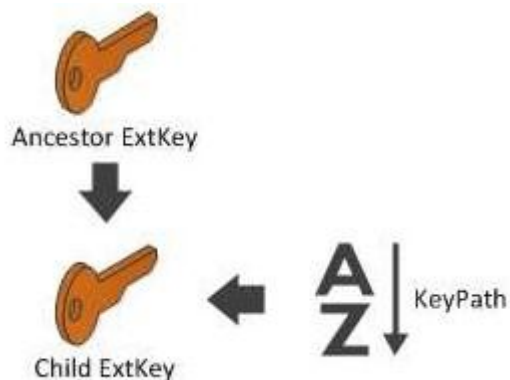
两种途径从父密钥那里派生 Child(1, 1)：

```
ExtKey parent = new ExtKey();
ExtKey child11 = parent.Derive(1).Derive(1);
```

或者

```
ExtKey parent = new ExtKey();
ExtKey child11 = parent.Derive(new KeyPath("1/1"));
```

因此，总的来说：



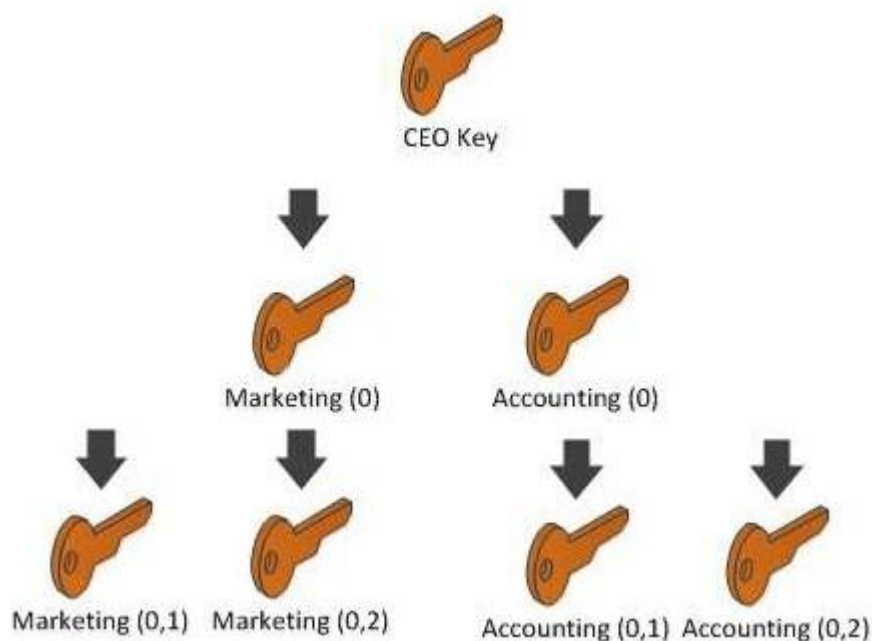
ExtPubKey 也是一样的。

为什么你需要分层秘钥？因为在多账户时可以更好地对秘钥进行分类。更多内容参考 BIP44。

它也允许在组织内部分配账户权力。

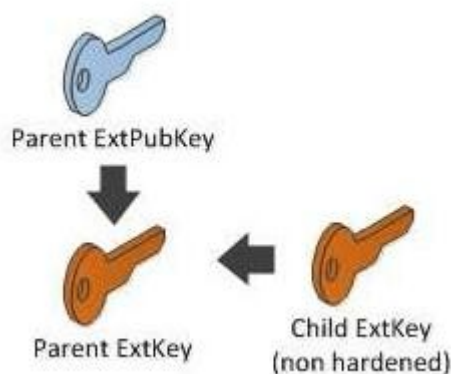
假设你是一个公司的 CEO，想要掌控所有的钱包，但是你不想要会计部门花市场部门的钱。

所以你的首选方案就是为每个部门生成一个层次结构。



然而，在这个案例中，Accounting 和 Marketing 可以恢复 CEO 的私钥。

我们把这些子秘钥定义为非固定的（non-hardened）。



```

ExtKey ceoKey = new ExtKey();
Console.WriteLine("CEO: " + ceoKey.ToString(Network.Main));
ExtKey accountingKey = ceoKey.Derive(0, hardened: false);

ExtPubKey ceoPubkey = ceoKey.Neuter();
  
```

```
//Recover ceo key with accounting private key and ceo public key
ExtKey ceoKeyRecovered = accountingKey.GetParentExtKey(ceoPubkey);
Console.WriteLine("CEO recovered: " +
ceoKeyRecovered.ToString(Network.Main));
```

```
CEO:
xprv9s21ZrQH143K2XcJU89thgkBehaMqvcj4A6JFwxPs6ZzGYHYT8dTchd87TC4NHSwvDuexuFVFpYaAt
3gztYtZyXmy2hCVyVyxumdxDBpoC

CEO recovered:
xprv9s21ZrQH143K2XcJU89thgkBehaMqvcj4A6JFwxPs6ZzGYHYT8dTchd87TC4NHSwvDuexuFVFpYaAt
3gztYtZyXmy2hCVyVyxumdxDBpoC
```

换句话说，非固定的密钥（non-hardened key）可以在分层结构中爬跃层次。

非固定的密钥（non-hardened key）应仅仅被用于对 single control 所拥有的账户进行分类。

在我们的案例中，CEO 应该创建一个固定的密钥（hardened key），这样会计部门就不能超越层次等级了。

```
ExtKey ceoKey = new ExtKey();
Console.WriteLine("CEO: " + ceoKey.ToString(Network.Main));
ExtKey accountingKey = ceoKey.Derive(0, hardened: true);

ExtPubKey ceoPubkey = ceoKey.Neuter();

ExtKey ceoKeyRecovered = accountingKey.GetParentExtKey(ceoPubkey); //Crash
你也可以通过 ExtKey.Derivate(KeyPath)创建 hardened key，在子密钥索引后使用撇号即可：
var nonHardened = new KeyPath("1/2/3");
var hardened = new KeyPath("1/2/3'");
```

我们可以设想一下，会计部门为每个客户生成一个父密钥，为每笔客户支付服务生成一个子密钥。

作为 CEO，你想要向其中一个地址上支付款项，下面就是处理过程：

```
ceoKey = new ExtKey();
string accounting = "1'";
int customerId = 5;
int paymentId = 50;
KeyPath path = new KeyPath(accounting + "/" + customerId + "/" + paymentId);
//Path : "1'/5/50"
ExtKey paymentKey = ceoKey.Derive(path);
```

### 3.2.4 HD 密钥助记代码(BIP39)

生成 HD 密钥是非常容易的。但是，如果我们想通过电话或者手写传输这些密钥，怎么样才

能一样方便呢？

类似 Trezor 这样的冷钱包，可以从一个容易写下来的句子生成 HD 秘钥。他们把这样的句子称为“种子”或者“助记符”。它也能用密码或者 PIN 保护起来。

My TREZOR / Seed recovery

Device label

My TREZOR

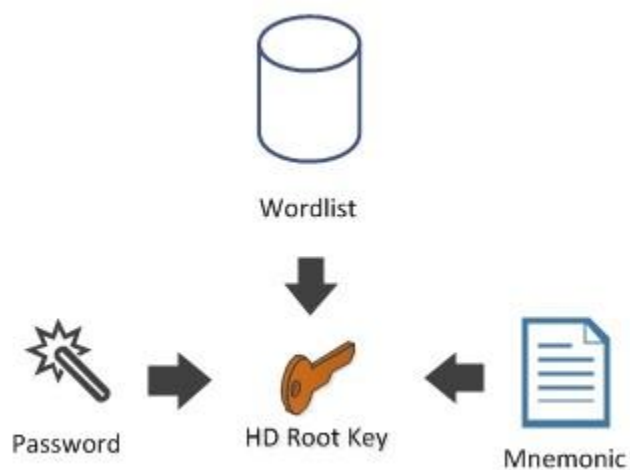
Number of words in your recovery seed

☒ 12 words
 ☐ 18 words
 ☐ 24 words

☒ Enable PIN protection
 ☐ Additional passphrase encryption

Continue Cancel

写句子的语言就叫做词汇表（Wordlist）。



```
Mnemonic mnemo = new Mnemonic(Wordlist.English, WordCount.Twelve);
ExtKey hdRoot = mnemo.DeriveExtKey("my password");
Console.WriteLine(mnemo);
```

minute put grant neglect anxiety case globe win famous correct turn link



现在，如果你有助记符和密码，就可以还原 hdRoot 秘钥。

```
mnemo = new Mnemonic("minute put grant neglect anxiety case globe win famous  
correct turn link",  
Wordlist.English);  
hdRoot = mnemo.DeriveExtKey("my password");
```

目前支持的词汇表包括：英语、日语、西班牙语、中文（简体和繁体）。

### 3.2.5 黑暗钱包

名字有点不吉利，但是它实际上没有什么黑暗的东西，它引来了没有必要的关注和麻烦。

黑暗钱包实际上解决了我们最初的两个问题：

- 避免备份过期失效
- 委托秘钥/地址生成给不可信第三方

但它有一个可以获大奖的特点。

你仅需要分享一个地址（叫 **StealthAddress**）就可以，不需要泄露其它任何私密信息。

提醒一下，如果你跟其他人分享了 **BitcoinAddress**，那么他们都可以通过区块链查询你的余额。

这是跟 **StealthAddress** 不一样的。

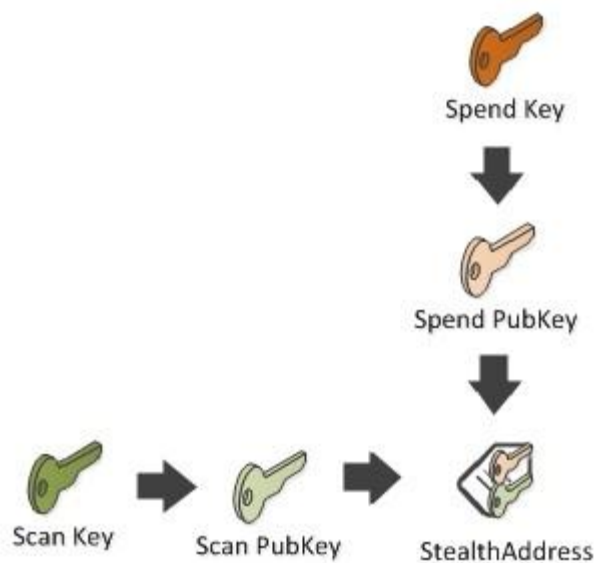
被叫做黑暗的真是很遗憾，因为它部分解决了由于比特币伪匿名而引起的隐私泄露问题。更好的名字应该是：**One Address**。

在黑暗钱包的术语里面，下面是不同的角色：

- **Scanner** 知道 **Scan Key**，有了它就可以觉察接收者的交易。
- 接收者知道 **Spend Key**，有了它就可以把他从交易中接收到的币花出去。
- 付款人知道接收者的 **StealthAddress**。

剩下的就是操作层面的细节了。

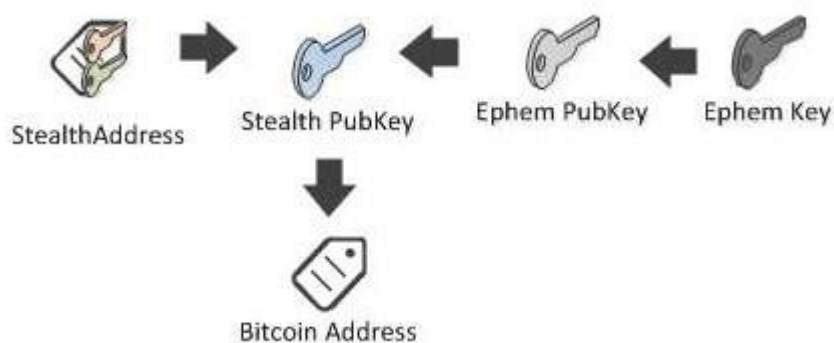
在底层，**StealthAddress** 包含一个或者几个 **Spend PubKey**，以及一个 **Scan PubKey**。



```
var scanKey = new Key();
var spendKey = new Key();
BitcoinStealthAddress stealthAddress
= new BitcoinStealthAddress
(
    scanKey: scanKey.PubKey,
    pubKeys: new[] { spendKey.PubKey },
    signatureCount: 1,
    bitfield: null,
    network: Network.Main);
```

付款方将用你的 **StealthAddress** 生成一个临时秘钥，叫 **Ephem Key**，然后生成一个 **Stealth 公钥**，

它是比特币地址的源头，也是支付目的地。



然后，它将 **Ephem PubKey** 打包进一个 **Stealth Metadata** 对象，嵌入到交易的 **OP\_RETURN** 中（就好像我们在第一个任务里面做的一样），它也会将输出添加到生成的比特币地址中。

(Stealth pub key 的地址)



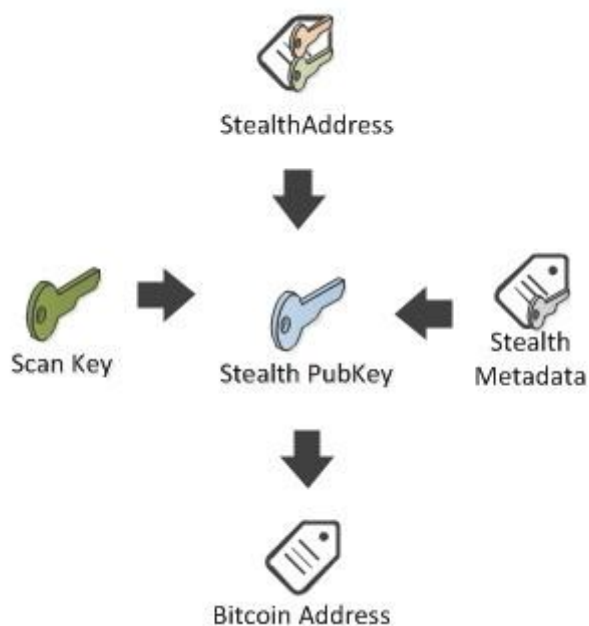
```
var ephemKey = new Key();
Transaction transaction = new Transaction();
stealthAddress.SendTo(transaction, Money.Coins(1.0m), ephemKey);
Console.WriteLine(transaction);
```

EphemKey 生成过程的实现细节你可以忽略，NBitcoin 将自动生成。

```
Transaction transaction = new Transaction();
stealthAddress.SendTo(transaction, Money.Coins(1.0m));
Console.WriteLine(transaction);
{
    ...
    "in": [],
    "out": [
        {
            "value": "0.00000000",
            "scriptPubKey": "OP_RETURN
06000000000211e76c3de929f7d8b3473f6e41fc31016d7a3e56a47e9f541b0d1cc7fa3f8
819"
        },
        {
            "value": "1.00000000",
            "scriptPubKey": "OP_DUP OP_HASH160
b4638a6b452bbfc6fdbb4e1e8c9f1952bbd18c39
OP_EQUALVERIFY OP_CHECKSIG"
        }
    ]
}
```

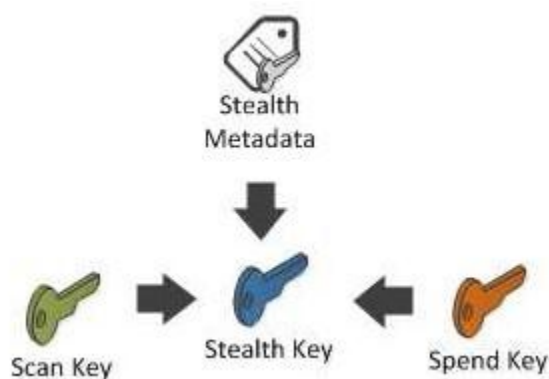
然后付款人添加输入并签名，最后把交易发送到网络上。

Scanner 知道 StealthAddress，Scan Key 可以恢复 Stealth PubKey，所以它也期待着向 BitcoinAddress 的付款。



然后 scanner 检查交易的输出是否与这些地址向对应，如果是，Scanner 向 Receiver 通知这个交易。

Receiver 可以通过它的 Spend Key 得到地址的私钥。



代码说明了 Scanner 对交易的巡查工作，至于接收者解密私钥的工作，我们将在 TransactionBuilder 部分作解释。

应当注意的是，一个 StealthAddress 可以有多个 spend pubkeys，那种情况下，地址表示一个多重签名。

黑暗钱包的一个限制就是 OP\_RETURN 的使用，我们无法轻易将任意数据嵌入到交易中，这个与传送比特币时不一样。（目前比特币规则仅允许每个交易 40 字节的 OP\_RETURN，很快将允许 80 字节）

## 四、其他类型的所有权

### 4.1 P2PK[H] (向公钥付款 [Hash])

在第二部分，我们知道了一个比特币地址就是一个公钥的哈希值。

我们也知道了，对于区块链而言，根本没有比特币地址的说法。区块链使用 `ScriptPubKey` 识别接收者，这种 `ScriptPubKey` 可以从地址生成。（反过来也一样）

```
Key key = new Key();
BitcoinAddress address = key.PubKey.GetAddress(Network.Main);
Console.WriteLine(address.ScriptPubKey);
```

```
OP_DUP OP_HASH160 c86bf818bfd2b4943c003464815a8259c0de5e59 OP_EQUALVERIFY
OP_CHECKSIG
```

然而，并不是所有的 `ScriptPubKey` 都代表一个比特币地址。

这是区块链第一个交易的例子（创世块）：

```
Console.WriteLine(Network.Main.GetGenesis().Transactions[0].ToString());
```

```
{
  ...
  "out": [
    {
      "value": "50.00000000",
      "scriptPubKey":
      "04678afdb0fe5548271967f1a67130b7105cd6a828e03909a67962e0ea1f61deb649f6bc3f4cef38c4f3
      5504e51ec112de5c384df7ba0b8d578a4c702b6bf11d5f OP_CHECKSIG"
    }
  ]
}
```

可以看到 `scriptPubKey` 的形式是不一样的：

一个比特币地址这样表示：

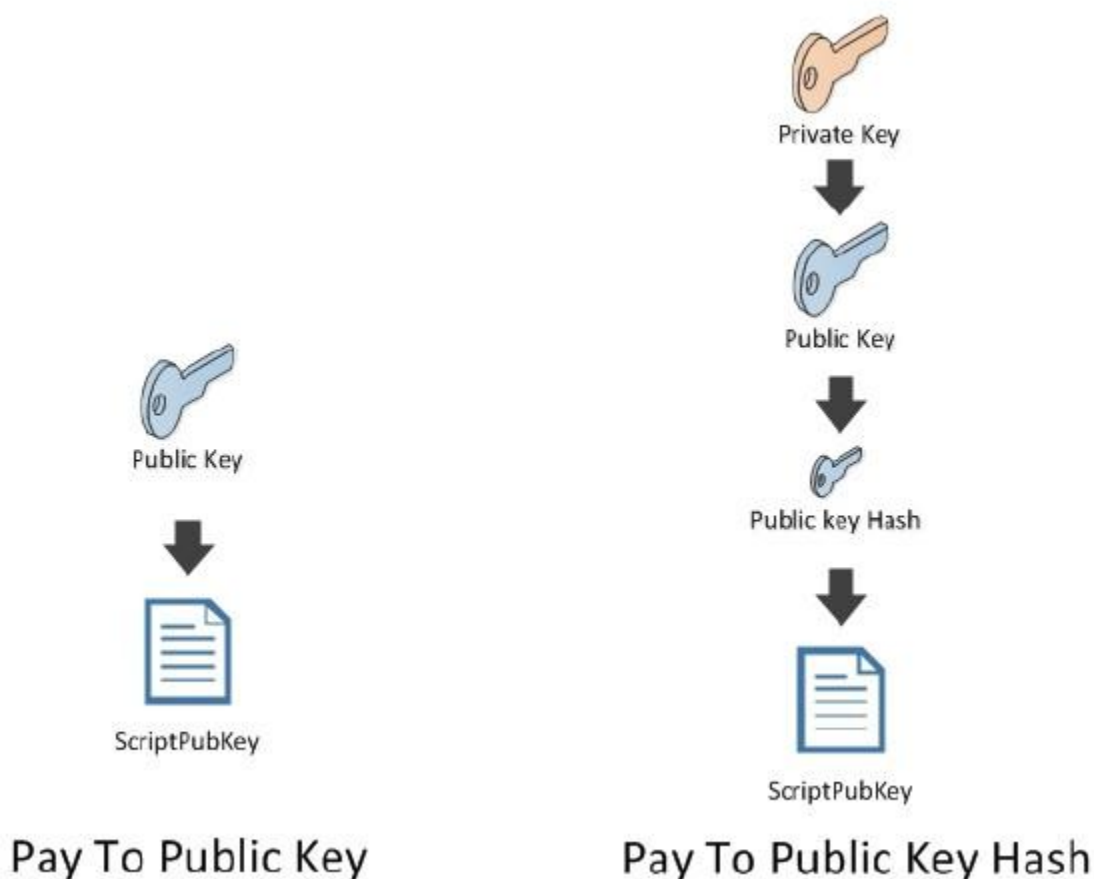
```
OP_DUP <hash> OP_EQUALVERIFY OP_CHECKSIG
```

但是这里：

```
<pubkey> OP_CHECKSIG
```

实际上，开始的时候，公钥被直接用在 `ScriptPubKey` 里面。

现在我们主要使用公钥的哈希值。



```
key = new Key();
Console.WriteLine("Pay to public key : " + key.PubKey.ScriptPubKey);
Console.WriteLine();
Console.WriteLine("Pay to public key hash : " +
key.PubKey.Hash.ScriptPubKey);
```

Pay to public key : 02fb8021bc7dedcc2f89a67e75cee81fedb8e41d6bfa6769362132544dfdf072d4  
OP\_CHECKSIG

Pay to public key hash : OP\_DUP OP\_HASH160 0ae54d4cec828b722d8727cb70f4a6b0a88207b2  
OP\_EQUALVERIFY OP\_CHECKSIG

这两种支付类型被叫做 **P2PK** (向公钥付款)、**P2PKH** (向公钥哈希付款)。

中本聪使用 **P2PKH** 而不是 **P2PK**，主要有两个原因：

- 你的公钥和私钥使用的椭圆曲线加密算法，无法抵挡改进的 **Shor** 算法的攻击，它就是用来解决在椭圆曲线上的离散逻辑问题的。简单来说，它意味着，通过一个量子计算机，理论上未来可能从一个公钥上获得一个私钥。如果只是当币被付出去时才公开公钥，这种攻击是无效的。（假设地址不会复用）
- 哈希长度很小，打印起来更容易，也很容易被嵌入到很小的空间上，比如一个 **QR** 代码。

当前，没有理由直接使用 P2PK，但是它仍然与 P2SH 联合起来使用。（后面将会看到）

## 4.2 多重签名

比特币也可以共享所有权。

你将创建一个 `ScriptPubKey` 表示 m-of-n 多重签名(multi sig)，它的意思是，如果需要支付币，m 个私钥需要对 n 个不同的公钥签名。

我们看看是如何工作的，先通过 Bob, Alice, 和 Satoshi 创建一个多重签名，如果需要支付币，就需要得到他们中的两个签名。

```
Key bob = new Key();
Key alice = new Key();
Key satoshi = new Key();

var scriptPubKey = PayToMultiSigTemplate
.Instance
.GenerateScriptPubKey(2, new[] { bob.PubKey, alice.PubKey,
satoshi.PubKey });

Console.WriteLine(scriptPubKey);
```

```
2 0282213c7172e9dff8a852b436a957c1f55aa1a947f2571585870bfb12c0c15d61
036e9f73ca6929dec6926d8e319506cc4370914cd13d300e83fd9c3dfca3970efb
0324b9185ec3db2f209b620657ce0e9a792472d89911e0ac3fc1e5b5fc2ca7683d 3
OP_CHECKMULTISIG
```

如上所述，scriptPubkey 的格式：OP\_CHECKMULTISIG

仅仅调用 `Transaction.Sign` 不能奏效，相比而言对它进行签名的过程有点复杂。

就算我们将更深入地探讨这个话题，我们还是使用 `TransactionBuilder` 来对交易签名吧。

设想一下，在一个叫 `received` 的交易里面，多重签名收到一个币。

```
Transaction received = new Transaction();
received.Outputs.Add(new TxOut(Money.Coins(1.0m), scriptPubKey));
```

Satoshi 和 Alice 一致同意为我的服务付款 1.0BTC。

这样就获得了从交易中收到的币：

```
Coin coin = received.Outputs.AsCoins().First();
```



然后，使用 TransactionBuilder, 创建 unsigned transaction。

```
BitcoinAddress nico = new Key().PubKey.GetAddress(Network.Main);
TransactionBuilder builder = new TransactionBuilder();
Transaction unsigned =
builder
    .AddCoins(coin)
    .Send(nico, Money.Coins(1.0m))
    .BuildTransaction(false);
```

交易还没有签名。这是 Alice 的签名：

```
builder = new TransactionBuilder();
Transaction aliceSigned =
builder
    .AddCoins(coin)
    .AddKeys(alice)
    .SignTransaction(unsigned);
```



然后是 Satoshi 的：

```
builder = new TransactionBuilder();
Transaction satoshiSigned =
builder
    .AddCoins(coin)
    .AddKeys(satoshi)
    .SignTransaction(unsigned);
```





现在，Satoshi 和 Alice 可以将他们的

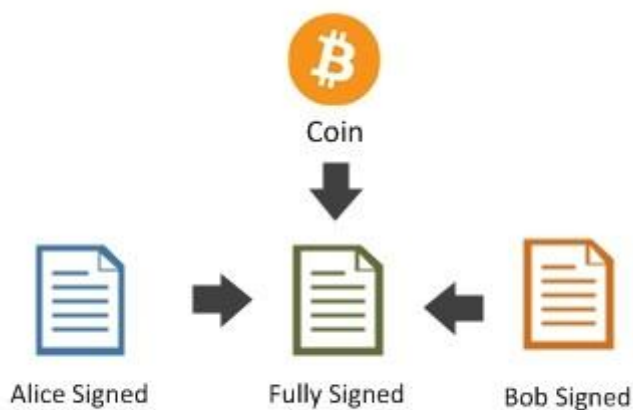
签名合并到一个交易中了。

```

builder = new TransactionBuilder();
Transaction fullySigned =
builder
    .AddCoins(coin)
    .CombineSignatures(satoshiSigned, aliceSigned);
Console.WriteLine(fullySigned);
{
    ...
    "in": [
        {
            "prev_out": {
                "hash":
"9df1e011984305b78210229a86b6ade9546dc69c4d25a6bee472ee7d62ea3c16",
                "n": 0
            },
            "scriptSig": "0
3045022100a14d47c762fe7c04b4382f736c5de0b038b8de92649987bc59bca83ea307b1a
202203e38
                dcc9b0b7f0556a5138fd316cd28639243f05f5ca1afc254b883482ddb91f01
3044022044c9f6818078887587cac126c3c2047b6e5425758e67df64e8d682dfbe373a290
2204ae7fda
                6ada9b7a11c4e362a0389b1bf90abc1f3488fe21041a4f7f14f1d856201"
            }
        ],
        "out": [
            {
                "value": "1.00000000",

```

```
    "scriptPubKey": "OP_DUP OP_HASH160
d4a0f6c5b4bcbf2f5830eabed3daa7304fb794d6
    OP_EQUALVERIFY OP_CHECKSIG"
  }
]
```



交易已经准备好了，可以发送到网络上了。

就算如上所述，比特币网络支持多重签名，但是仍然有个问题：一个对比特币没有太多概念的人而言，你如何叫他付款给 `satoshi/alice/bob` 的多重签名？因为这种 `scriptPubKey` 不能简单地使用比特币地址表示。

如果我们可以把 `scriptPubKey` 表示得跟比特币地址一样简洁的话，是不是很酷？

好吧，这是可能的，叫做 `Bitcoin Script Address`，也叫 `Pay to Script Hash`。（P2SH）

目前为止，这里你看到了 `native Pay To Multi Sig`，但是 `native P2PK` 还没有被直接用过，他们被打包进 `Pay To Script Hash`。

## 4.3P2SH ( 向脚本哈希付款)

如前所述，在代码层面多重签名可以轻而易举地运转起来，然而，在 `p2sh` 出现前，用户还没有办法像支付给比特币地址那样简单地向多重签名的 `scriptPubKey` 付款。

`P2SH` 也就是 `Pay To Script Hash`，是一种表示 `scriptPubKey` 的简便方法，无论它多么复杂，都可以像 `Bitcoin Script Address` 一样简单。

让我们看一下在前面部分中我们创建的多重签名长啥样。

```
Key bob = new Key();
Key alice = new Key();
```

```
Key satoshi = new Key();

var scriptPubKey = PayToMultiSigTemplate
    .Instance
    .GenerateScriptPubKey(2, new[] { bob.PubKey, alice.PubKey,
satoshi.PubKey });

Console.WriteLine(scriptPubKey);
```

```
2 0282213c7172e9dff8a852b436a957c1f55aa1a947f2571585870bfb12c0c15d61
036e9f73ca6929dec6926d8e319506cc4370914cd13d300e83fd9c3dfca3970efb
0324b9185ec3db2f209b620657ce0e9a792472d89911e0ac3fc1e5b5fc2ca7683d 3
OP_CHECKMULTISIG
```

很复杂吧？

那么，我们来看看作为 P2SH 支付的 scriptPubKey 是什么样子的：

```
Key bob = new Key();
Key alice = new Key();
Key satoshi = new Key();

Script redeemScript =
PayToMultiSigTemplate
    .Instance
    .GenerateScriptPubKey(2, new[] { bob.PubKey, alice.PubKey,
satoshi.PubKey });
Console.WriteLine(redeemScript.Hash.ScriptPubKey);
```

```
OP_HASH160 57b4162e00341af0ffc5d5fab468d738b3234190 OP_EQUAL
```

发现不同点了吗？p2sh 的 scriptPubKey 代表多重签名的哈希值

(redeemScript.Hash.ScriptPubKey)。

既然它是一个哈希值，你就可以轻易地将它转换为基于 base58 格式的字符串

BitcoinScriptAddress。

```
Key bob = new Key();
Key alice = new Key();
Key satoshi = new Key();

Script redeemScript =
    PayToMultiSigTemplate
        .Instance
        .GenerateScriptPubKey(2, new[] { bob.PubKey, alice.PubKey,
satoshi.PubKey });
```

```
//Console.WriteLine(redeemScript.Hash.ScriptPubKey);  
Console.WriteLine(redeemScript.Hash.GetAddress(Network.Main));
```

```
3E6RvwLNfkH6PyX3bqoVGKzrx2AqSJFhjo
```

这样的地址所有用户钱包都能识别，就算它们不理解什么是多重签名。

在 P2SH 支付中，我们提到的 Redeem Script, scriptPubKey 都是经过哈希计算的。



因为付款人只知道 RedeemScript 的哈希值，他不知道 RedeemScript，在我们的例子中甚至都不需要知道他正在给 Bob/Satoshi/Alice 的多重签名支付款项。

对这些交易进行签名跟之前的做法很像，唯一的区别就是当你为 TransactionBuilder 创建 Coin 的时候需要提供 Redeem Script。

设想一下，在一个叫 received 的交易中，多重签名 P2SH 收到一个币。

```
Script redeemScript =  
    PayToMultiSigTemplate  
        .Instance  
        .GenerateScriptPubKey(2, new[] { bob.PubKey, alice.PubKey,  
satoshi.PubKey });  
////Console.WriteLine(redeemScript.Hash.ScriptPubKey);  
//Console.WriteLine(redeemScript.Hash.GetAddress(Network.Main));  
  
Transaction received = new Transaction();  
//Pay to the script hash  
received.Outputs.Add(new TxOut(Money.Coins(1.0m), redeemScript.Hash));
```

提醒：支付款项被送到了 `redeemScript.Hash`，而不是 `redeemScript`！

然后，当 `alice/bob/satoshi` 想要将他们收到的币支付出去的时候，他们不是创建一个 `Coin` 而是创建一个 `ScriptCoin`。

```
//Give the redeemScript to the coin for Transaction construction
//and signing
ScriptCoin coin = received.Outputs.AsCoins().First()
    .ToScriptCoin(redeemScript);
```



关于交易生成的剩余代码和签名就跟前面提到的一般多重签名完全一样了。

## 4.4 灵活机动性

从比特币 0.10 开始，`RedeemScript` 可以是机动的，这意味着通过比特币的脚本语言，你可以重新对“所有权”进行定义。

比如，我们可以支付款项给任何知道我生日（用 `UTF8` 格式序列化）的人，也可以给任何知道我秘钥（`1KF8kUVHK42XzgcMJF4Lxz4wcL5WDL97PB`）的人。

脚本语言的细节超出了我们讨论的范围，你可以很容易在很多网站上找到文档，它是一个基于栈的语言，所以任何做过汇编的人都可以读懂它。

那么，我们首先创建 `RedeemScript`，

```
BitcoinAddress address = new
BitcoinAddress("1KF8kUVHK42XzgcMJF4Lxz4wcL5WDL97PB");
var birth = Encoding.UTF8.GetBytes("18/07/1988");
var birthHash = Hashes.Hash256(birth);
Script redeemScript = new Script(
    "OP_IF "
    + "OP_HASH256 " + Op.GetPushOp(birthHash.ToBytes()) + " OP_EQUAL " +
```

```
"OP_ELSE "  
+ address.ScriptPubKey + " " +  
"OP_ENDIF");
```

这个 RedeemScript 的意思是有 2 种途径支付这些 ScriptCoin: 要么你知道可以得出 birthHash 的数据, 要么你拥有比特币地址。

假设我们发送比特币到这个 redeemScript:

```
var tx = new Transaction();  
tx.Outputs.Add(new TxOut(Money.Parse("0.0001"), redeemScript.Hash));
```

这样我们创建一个交易, 用以支付这个 output:

```
//Create spending transaction  
Transaction spending = new Transaction();  
spending.AddInput(new TxIn(new OutPoint(tx, 0)));
```

首选工作就是取得我的生日, 并且在 scriptSig 中证明:

```
////Option 1 : Spender knows my birthdate  
Op pushBirthdate = Op.GetPushOp(birth);  
Op selectIf = OpcodeType.OP_1; //go to if  
Op redeemBytes = Op.GetPushOp(redeemScript.ToBytes());  
Script scriptSig = new Script(pushBirthdate, selectIf, redeemBytes);  
spending.Inputs[0].ScriptSig = scriptSig;
```

你可以看到在 scriptSig 我推送了 OP\_1, 这样我得以进入 RedeemScript 的 OP\_IF:

因为没有模板, 创建这样的 scriptSig, 你可以参照手工创建 P2SH scriptSig。

然后你可以确认, scriptSig 证明了 scriptPubKey 的所有权:

```
//Verify the script pass  
var result = spending  
    .Inputs  
    .AsIndexedInputs()  
    .First()  
    .VerifyScript(tx.Outputs[0].ScriptPubKey);  
Console.WriteLine(result);  
//////////
```

True
------

第二种支付比特币的方法就是证明这个所有权: 1KF8kUVHK42XzgcMJF4Lxz4wcl5WDL97PB

```
////Option 2 : Spender knows my private key  
BitcoinSecret secret = new BitcoinSecret("...");  
var sig = spending.SignInput(secret, redeemScript, 0);  
var p2pkhProof = PayToPubkeyHashTemplate  
    .Instance  
    .GenerateScriptSig(sig, secret.PrivateKey.PubKey);
```

```
selectIf = OpcodeType.OP_0; //go to else
scriptSig = p2pkhProof + selectIf + redeemBytes;
spending.Inputs[0].ScriptSig = scriptSig;
```

然后所有权也被证明了。

```
//Verify the script pass
result = spending
    .Inputs
    .AsIndexedInputs()
    .First()
    .VerifyScript(tx.Outputs[0].ScriptPubKey);
Console.WriteLine(result);
//////////
```

True

## 4.5 使用 TransactionBuilder

当你对第一个 P2SH 和多重签名交易签名的时候,你就知道 **TransactionBuilder** 是如何工作的了。

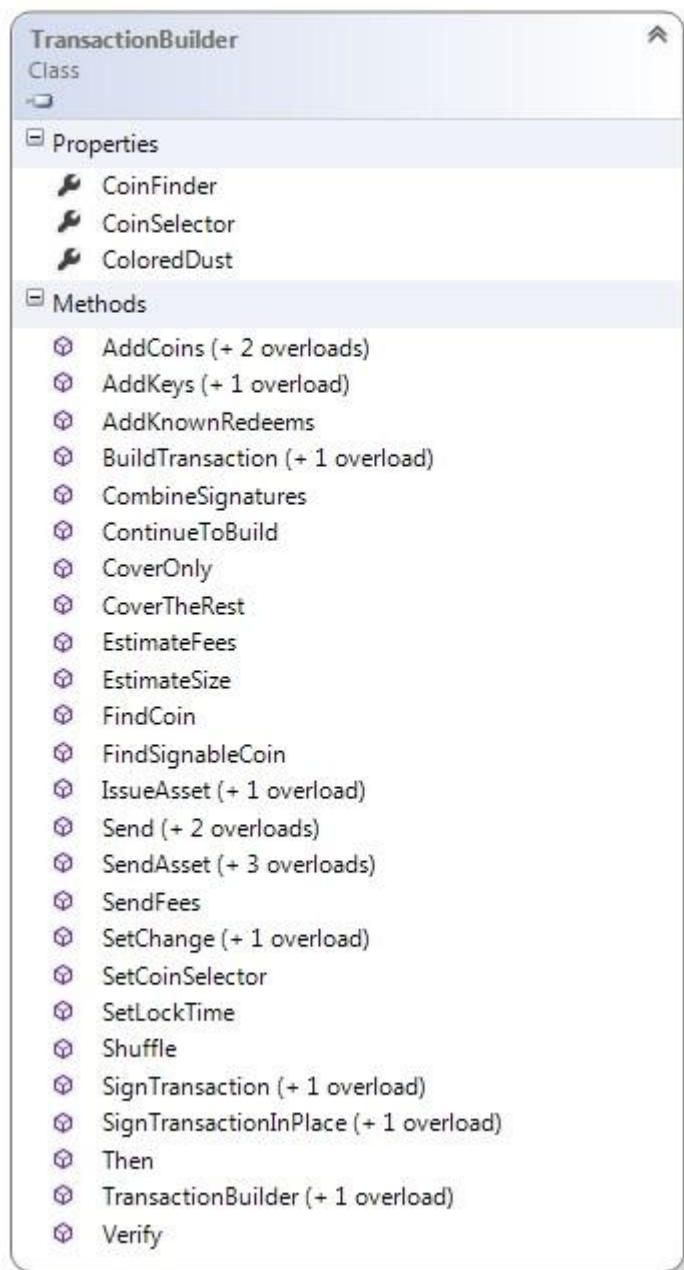
我们将看到利用它的强大力量,可以对更加复杂的交易签名。通过 **TransactionBuilder** 你可以:

- 支付任意的 P2PK, P2PKH, Multi Sig
- 在前面的 redeem script 中支付任意的 P2SH
- 支付 Stealth (黑暗钱包)
- 发行和传送颜色币(开放式资产,在下面的章节中有介绍)
- 联合部分签名的交易
- 评估未签名交易的最终大小和费用
- 验证一个交易是否被完全签名了

**TransactionBuilder** 的目标就是把 **Coin** 和 **Keys** 作为输入,返回一个经过签名的,或者部分签名的交易。



TransactionBuilder 将会弄明白应该使用什么币，签什么名。



生成器的使用包括 4 个步骤：

- 收集用于支付的货币
- 收集你拥有的秘钥
- 列出你需要向什么 scriptPubKey 发送多少钱
- 创建并对交易签名
- 可选：你把交易交给其他人，然后他签名或者继续创建它。



我们先收集一些比特币吧，然后就可以在这些比特币上创建一个虚假的交易。

假设这个交易拥有一个 P2PKH, P2PK 和关于 Bob 和 Alice 多重签名的币。

```
var bob = new Key();
var alice = new Key();
var bobAlice = PayToMultiSigTemplate
    .Instance
    .GenerateScriptPubKey(2, bob.PubKey, alice.PubKey);

Transaction init = new Transaction();
init.Outputs.Add(new TxOut(Money.Coins(1.0m), alice.PubKey));
init.Outputs.Add(new TxOut(Money.Coins(1.0m), bob.PubKey.Hash));
init.Outputs.Add(new TxOut(Money.Coins(1.0m), bobAlice));
```

现在假设他们想要将这个交易中的比特币支付给中本聪，他们首先得获得这些比特币。

```
var satoshi = new Key();
Coin[] coins = init.Outputs.AsCoins().ToArray();
```

现在假设 bob 想要支付 0.2BTC，Alice 想要支付 0.3BTC，他们一致同意以 bobAlice 名义支付 0.5BTC。

```
var builder = new TransactionBuilder();
Transaction tx = builder
    .AddCoins(bobCoin)
    .AddKeys(bob)
    .Send(satoshi, Money.Coins(0.2m))
    .SetChange(bob)
    .Then()
    .AddCoins(aliceCoin)
    .AddKeys(alice)
    .Send(satoshi, Money.Coins(0.3m))
    .SetChange(alice)
    .Then()
    .AddCoins(bobAliceCoin)
    .AddKeys(bob, alice)
    .Send(satoshi, Money.Coins(0.5m))
    .SetChange(bobAlice)
    .SendFees(Money.Coins(0.0001m))
    .BuildTransaction(sign: true);
```

你可以验证一下，它已经被完整签名了，并且已经准备好被发送到网络上了。

```
Console.WriteLine(builder.Verify(tx));
```

True

这个模型比较好的地方是它对 P2SH 的工作方法是一样的，除了你需要创建 ScriptCoin。



```

nit = new Transaction();
init.Outputs.Add(new TxOut(Money.Coins(1.0m), bobAlice.Hash));

coins = init.Outputs.AsCoins().ToArray();
ScriptCoin bobAliceScriptCoin = coins[0].ToScriptCoin(bobAlice);
    
```

然后是签名:

```

builder = new TransactionBuilder();
tx = builder
    .AddCoins(bobAliceScriptCoin)
    .AddKeys(bob, alice)
    .Send(satoshi, Money.Coins(1.0m))
    .SetChange(bobAlice.Hash)
    .BuildTransaction(true);
Console.WriteLine(builder.Verify(tx));
    
```

True

对 Stealth Coin 来说, 这基本上是一样的。除了一件事情, 如果你记得关于黑暗钱包的介绍的话, 我说你需要一个 ScanKey 用以观察 StealthCoin。



我们就像签名章节一样创建 darkAliceBob 的秘密地址:

```

Key scanKey = new Key();
    
```

```
BitcoinStealthAddress darkAliceBob =  
new BitcoinStealthAddress  
(  
    scanKey: scanKey.PubKey,  
    pubKeys: new[] { alice.PubKey, bob.PubKey },  
    signatureCount: 2,  
    bitfield: null,  
    network: Network.Main  
);
```

假设某人发送了这个交易:

```
/Someone sent to darkAliceBob  
init = new Transaction();  
darkAliceBob  
.SendTo(init, Money.Coins(1.0m));
```

scanner 将发现 StealthCoin:

```
//Get the stealth coin with the scanKey  
StealthCoin stealthCoin  
= StealthCoin.Find(init, darkAliceBob, scanKey);
```

然后发送给 bob 和 alice, 让他们签名 :

```
//Spend it  
tx = builder  
    .AddCoins(stealthCoin)  
    .AddKeys(bob, alice, scanKey)  
    .Send(satoshi, Money.Coins(1.0m))  
    .SetChange(bobAlice.Hash)  
    .BuildTransaction(true);  
Console.WriteLine(builder.Verify(tx));
```

True
------

提示: 你需要 scanKey 用以支付一个 StealthCoin

## 五、其他类型的资产

### 5.1 颜色币

在前面的章节中，我们已经看到了多种类型的所有权。

你已经看到了所有不同类型的所有权以及所有权证明，理解了比特币如何被编码产生新型的所有权。

目前为止，你看到了如何在网络上交易比特币。

然而，你可以使用比特币网络传输和交易任意类型的资产。

我们称这些资产为“颜色币”。

就区块链而言，一个比特币和一个颜色币没有区别。

一个颜色币由一个标准的 TxOut 表示，大多数时候，这个 TxOut 有一个剩余的比特币价值，叫做“尘埃”。（600 聪）

颜色币的真正价值就在于货币的发行方用什么来进行对价交易。



因为一个颜色币就是一个含有特殊含义的标准币，它遵循所有关于工作量证明的规则，以及 TransactionBuilder 认为正确的事情。你可以按照前面一样的规则传输一个颜色币。

就区块链而言，一个颜色币就是一个币，与其它币一样。

你可以用颜色币表示多种类型的资产：公司股权、债券、股票、选票。

但是无论你表示的是什么类型的资产，总是有一个可信的关系存在于资产发行方和所有者之间。

如果你拥有一些公司股份，公司可能决定不向你分红。

如果你拥有债券，银行可能在到期时不给兑现。

但是，在李嘉图合约（Ricardian Contracts）的帮助下，违反合约可以被自动检测到。一份李嘉图合约就是由发行方签名证明资产权力的合约。这种合约可以让人直接可读，也可以是结构化的（json），这样工具就可以自动证明违约情况。发行方不能更改资产附有的李嘉图合约。

区块链仅仅是金融工具的传输媒介。

富有创意的地方就是每个人都能创建并传输自己的资产而不用通过中介。传统资产传输媒介（清算所）要么是严格监管的，要么是严格保密的，反正就是不对大众开发。

开放式资产（Open Asset）是 Flavien Charlon 命名的一种协议，它描述了如何在区块链上传输和发行颜色币。

还存在其它一些协议，但是开放式资产是最简单灵活的，也是 NBitcoin 唯一支持的。

在本书的剩余部分，我不打算探讨开放式资产协议的细节，github 网页上详细介绍。

## 5.2 发行一项资产

### 5.2.1 目的

为了练习的目的，我将发行 BlockchainProgramming 币。

这些币的所有人将可以下载本书第 3 部分，只要他们把币发回给我就行。

不用担心，第三部分对任何人都是可读的，但是首先提供给 BlockchainProgramming 币的所有者。

我将发送币给支持我的人，名单在本书网站上列出来了：

<http://blockchainprogramming.azurewebsites.net/>

你每给我 0.004BTC，我就给你 1 个 BlockchainProgramming 币，如果有良好建议的话，就加 1 个。

看看我是怎么编码实现这些特性的。

### 5.2.2 发行币

在开放式资产中，AssetID 是从发行方的 ScriptPubKey 中派生的。

如果你想发行一个颜色币，你需要证明这些 ScriptPubKey 的所有权。在区块链上的唯一方法就是支付一个属于这些 ScriptPubKey 的币。

在 NBitcoin 中，为了发行颜色币而支付的币就叫做“Issuance Coin”（发行币）。

我准备从本书比特币地址上发行一项资产，本书比特币地址：

1KF8kUVHK42XzgcMJF4Lxz4wcL5WDL97PB。

看了一下我的余额，我决定使用下列比特币来发行资产。

```
{
    "transactionId":
    "eb49a599c749c82d824caf9dd69c4e359261d49bbb0b9d6dc18c59bc9214e43b",
    "index": 0,
    "value": 2000000,
    "scriptPubKey": "76a914c81e8e7b7ffca043b088a992795b15887c96159288ac",
    "redeemScript": null
}
```

下面创建 issuance coin。

```
var coin = new Coin(
    fromTxHash: new

uint256("eb49a599c749c82d824caf9dd69c4e359261d49bbb0b9d6dc18c59bc9214e43b
"),
    fromOutputIndex: 0,
    amount: Money.Satoshis(2000000),
    scriptPubKey: new

Script(Encoders.Hex.DecodeData("76a914c81e8e7b7ffca043b088a992795b15887c9
6159288ac
"))));

var issuance = new IssuanceCoin(coin);
```

现在我需要在 TransactionBuilder 的帮助下创建并签名交易。

```
var nico = BitcoinAddress.Create("15sYbVpRh6dyWycZMwPdxJWD4xbfxReeHe");
var bookKey = new BitcoinSecret("???????");
TransactionBuilder builder = new TransactionBuilder();

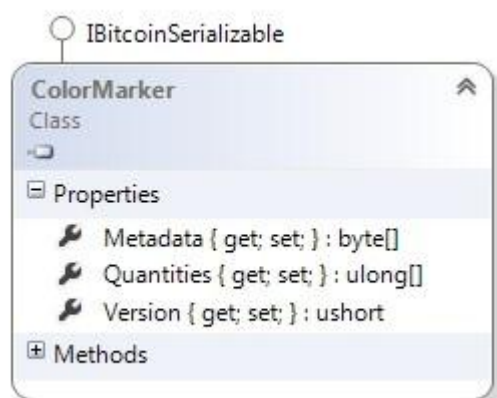
var tx = builder
    .AddKeys(bookKey)
    .AddCoins(issuance)
    .IssueAsset(nico, new Asset(issuance.AssetId, 10))
    .SendFees(Money.Coins(0.0001m))
    .SetChange(bookKey.GetAddress())
    .BuildTransaction(true);

Console.WriteLine(tx);
{
    ...
    "out": [
```

```
{
  "value": "0.00000600",
  "scriptPubKey": "OP_DUP OP_HASH160
356facdac5f5bcae995d13e667bb5864fd1e7d59
OP_EQUALVERIFY OP_CHECKSIG"
},
{
  "value": "0.01989400",
  "scriptPubKey": "OP_DUP OP_HASH160
c81e8e7b7ffca043b088a992795b15887c961592
OP_EQUALVERIFY OP_CHECKSIG"
},
{
  "value": "0.00000000",
  "scriptPubKey": "OP_RETURN 4f410100010a00"
}
]
```

你可以看到它包含一个 **OP\_RETURN** 输出，实际上，这是颜色币信息所在的地方。

这是 **OP\_RETURN** 中的数据格式。



我们的例子中，数量只有 10，这就是我发行资产的数量。

元数据是可变的数据。我们将看到可以推出一个指向“资产定义”的 url。

一个“资产定义”是描述资产是什么的文档。它是可选的，在我们的例子中不准备使用它。（后面谈到李嘉图合约时会回头讨论）

更多信息参照：[OpenAssetSpecification](#)。

交易已经准备好了，可以发送到网络上了：

```
using (var node = Node.ConnectToLocal(Network.Main)) //Connect to the node
{
```

```
node.VersionHandshake(); //Say hello
//Advertize your transaction (send just the hash)
node.SendMessage(new InvPayload(InventoryType.MSG_TX, tx.GetHash()));
//Send it
node.SendMessage(new TxPayload(tx));
Thread.Sleep(500); //Wait a bit
}
```

我的比特币钱包同时有本书的地址和“Nico”的地址。

```
État: 0/non confirmée
Date: 25/02/2015 16:51
Débit: 0.00 BTC
Frais de transaction: -0.0001 BTC
Montant net: -0.0001 BTC
ID de la transaction:
fa6db7a2e478f3a8a0d1a77456ca5c9fa593e49fd0cf65c7e349e5a4cbe58842-000
```

你会发现，比特币核心仅仅显示我支付的 0.0001BTC 费用，忽略了 600 聪，这是由于垃圾邮件预防特性导致的。

这个经典的比特币钱包就只知道颜色币。

更加糟糕的是：如果一个经典比特币钱包付出一个颜色币，它将毁掉底层资产，仅仅传送 TxOut 的比特币价值（600 聪）

为了防止用户发送颜色币到一个不支持它的钱包，开放式资产拥有自己的地址格式，这种格式仅仅颜色币钱包能理解。

```
nico = BitcoinAddress.Create("15sYbVpRh6dyWycZMwPdxJWD4xbfxReeHe");
Console.WriteLine(nico.ToColoredAddress());
```

```
akFqRqfdmAAxFPDmvQZVpcAQnQZmqrX4gcZ
```

现在，你可以对比一下开放式资产兼容的钱包，比如 Coinprism，然后看看我正确检测到的资产：



Address	akFqRqfdmAAxfPDmvQZVpcAQnQZmqrX4gcZ
Total transactions	67
Legacy address	15sYbVpRh6dyWycZMwPdxjWD4xbfxReeHe

Balance		
	Bitcoin	0.71339717 BTC
Assets		
	Unnamed colored coins	10 Units
	Asset ID: AVAVfLSb1KZf9tJzrUVpktjxKUXGxUTD4e	

我前面介绍说，AssetID 是从发行方的 ScriptPubKey 派生出来的，下面是对应的代码实现：

```
var book = BitcoinAddress.Create("1KF8kUVHK42XzgcmJF4Lxz4wcL5WDL97PB");
var assetId = new AssetId(book).GetWif(Network.Main);
Console.WriteLine(assetId);
```

```
AVAVfLSb1KZf9tJzrUVpktjxKUXGxUTD4e
```

## 5.3 传输资产

现在，假设我发送给你一些 BlockchainProgramming 币。

你发送回给我吧，这样我就将第 3 部分开放给你？

你需要创建一个颜色币。

在上面的例子中，假设我想要将“nico”地址上收到的资产支付 10 个单位出去。

从这个网络服务，我就可以看到想要支付什么货币。

```
{
    "transactionId": "fa6db7a2e478f3a8a0d1a77456ca5c9fa593e49fd0cf65c7e349e5a4cbe58842",
    "index": 0,
    "value": 600,
    "scriptPubKey": "76a914356facdac5f5bcae995d13e667bb5864fd1e7d5988ac",
    "redeemScript": null,
    "assetId": "AVAVfLSb1KZf9tJzrUVpktjxKUXGxUTD4e",
    "quantity": 10
}
```

下面是实例化这些颜色币的代码：

```
var coin = new Coin(
    fromTxHash: new
uint256("fa6db7a2e478f3a8a0d1a77456ca5c9fa593e49fd0cf65c7e349e5a4cbe58842"),
    fromOutputIndex: 0,
    amount: Money.Satoshis(2000000),
    scriptPubKey: new
Script(Encoders.Hex.DecodeData("76a914356facdac5f5bcae995d13e667bb5864fd1e7d5988ac")));
BitcoinAssetId assetId = new
BitcoinAssetId("AVAVfLSb1KZf9tJzrUVpktjxKUXGxUTD4e");
ColoredCoin colored = coin.ToColoredCoin(assetId, 10);
```

后面我们将告诉你，如何使用一些网络服务或者定制化代码，可以更容易得到币。

我也需要另一种币（费用）来支付费用。

通过 `TransactionBuilder` 进行资产传输是非常容易的。

```
var book = BitcoinAddress.Create("1KF8kUVHK42XzgcmJF4Lxz4wcL5WDL97PB");
var nicoSecret = new BitcoinSecret("????????");
var nico = nicoSecret.GetAddress(); //15sYbVpRh6dyWycZMwPdxJWD4xbfxReeHe

var forFees = new Coin(
    fromTxHash: new
uint256("7f296e96ec3525511b836ace0377a9fbb723a47bdfb07c6bc3a6f2a0c23eba26"),
    fromOutputIndex: 0,
    amount: Money.Satoshis(4425000),
    scriptPubKey: new
Script(Encoders.Hex.DecodeData("76a914356facdac5f5bcae995d13e667bb5864fd1e7d5988ac")));

TransactionBuilder builder = new TransactionBuilder();
```



```
var tx = builder
    .AddKeys(nicoSecret)
    .AddCoins(colored, forFees)
    .SendAsset(book, new Asset(assetId, 10))
    .SetChange(nico)
    .SendFees(Money.Coins(0.0001m))
    .BuildTransaction(true);
Console.WriteLine(tx);
{
    ...
    "out": [
        {
            "value": "0.00000000",
            "scriptPubKey": "OP_RETURN 4f410100010a00"
        },
        {
            "value": "0.00000600",
            "scriptPubKey": "OP_DUP OP_HASH160
c81e8e7b7ffca043b088a992795b15887c961592
OP_EQUALVERIFY OP_CHECKSIG"
        },
        {
            "value": "0.04415000",
            "scriptPubKey": "OP_DUP OP_HASH160
356facdac5f5bcae995d13e667bb5864fd1e7d59
OP_EQUALVERIFY OP_CHECKSIG"
        }
    ]
}
```

基本上是成功的:

Hash	a9abbc6773c6eae9937fee382d0f8391c6f1a8b0cffb5874743e80302ce09b67
Date	Wednesday, February 25, 2015 5:22:47 PM
Fee paid	0.0001 BTC
Assets transacted	1



The transaction is not confirmed yet.

Bitcoin			
	← akFqRqfdmAaXfP...	-0.000106	> akVD1ze cnXvDrvn8Ls... 0.000006 Fees 0.0001
Unnamed colored coins		AVAVfLSb1KZF9tjzrUVpktjxKUXGxUTD4e	
	← akFqRqfdmAaXfP...	-10	> akVD1ze cnXvDrvn8Ls... 10

## 5.4 单元测试

前面我硬编码了颜色币的属性。

原因是，我只打算向你展示一下，如何从颜色币构建一个交易。

在实际中，你也可以依赖第三方的 API 来获取一个交易的颜色币或者余额。那可能不是一个好办法，因为你的程序就必须依赖于对 API 提供商的信任。

NBitcoin 允许你依赖于一个网络服务，或者提供你自己的实现来获取一个交易的颜色币。这就使得你可以灵活地对代码进行单元测试，使用别人的实现或者自己的。

我们介绍一下 2 位发行人：Silver 和 Gold。还有 3 位参与者：Bob, Alice、Satoshi。

我们创建一个虚拟的交易，支付一些比特币给 Silver，Gold 和 Satoshi。

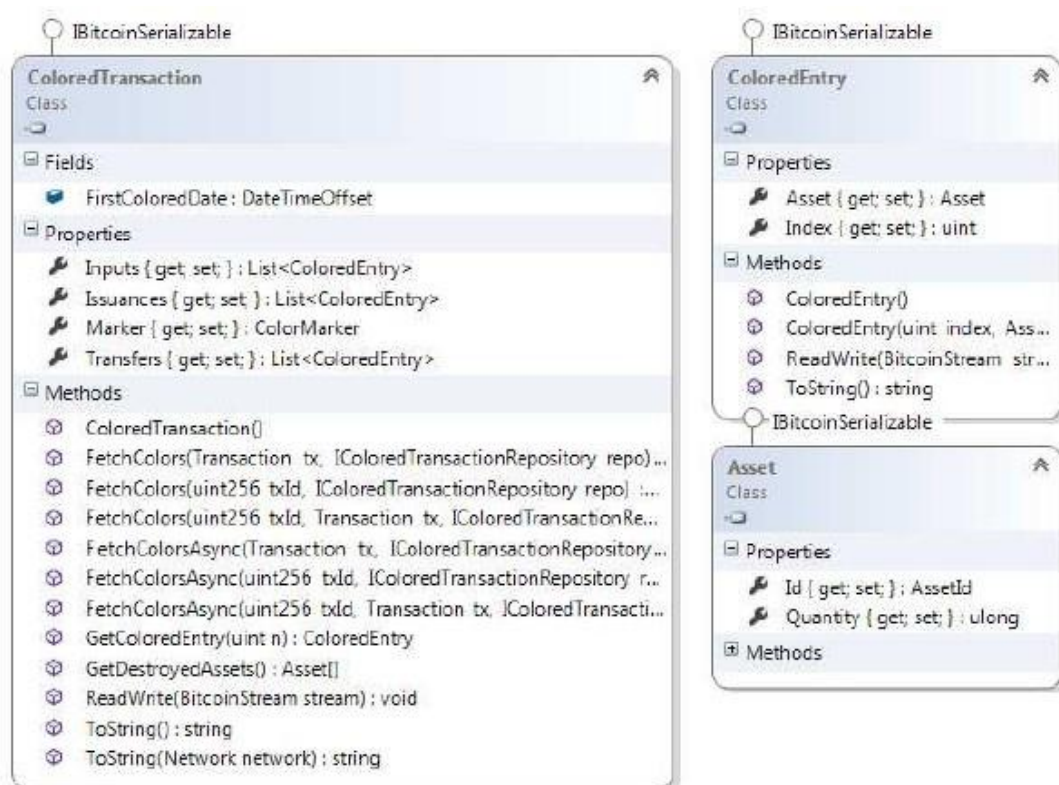
```
var gold = new Key();
var silver = new Key();
var goldId = gold.PubKey.ScriptPubKey.Hash.ToAssetId();
var silverId = silver.PubKey.ScriptPubKey.Hash.ToAssetId();

var bob = new Key();
var alice = new Key();
var satoshi = new Key();
```

```
var init = new Transaction()
{
    Outputs =
    {
        new TxOut("1.0", gold),
        new TxOut("1.0", silver),
        new TxOut("1.0", satoshi)
    }
};
```

Init 没有包含任何 Colored Coin issuance 和 Transfer。

但是设想一下，如果你需要确定一下，将会做什么？在 NBitcoin 中，传输和发行颜色币的概述使用一个类来描述，名叫 ColoredTransaction。

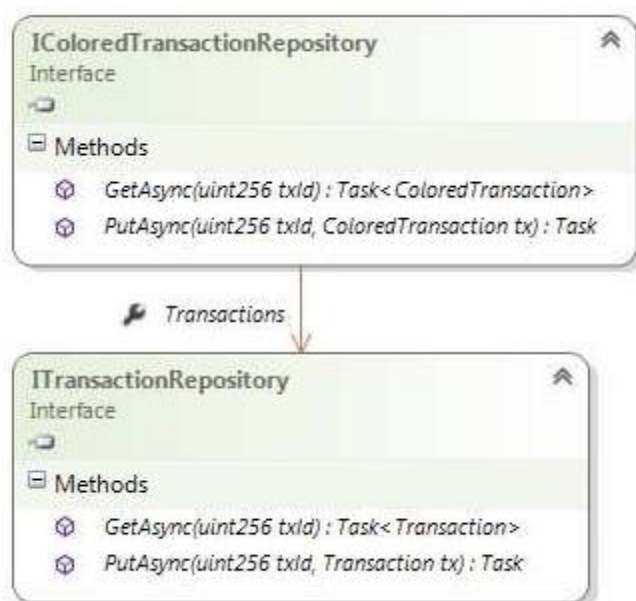


ColoredTransaction 将告诉你：

- 通过哪个 TxIn 支付哪项资产
- 哪个 TxOut 发行哪项资产
- 哪个 TxOut 传输哪项资产

让我们感兴趣的方法是 `FetchColor`，这个方法允许你从 input 的交易中提取颜色币的信息。

你会发现它依赖于一个 `IColoredTransactionRepository`。

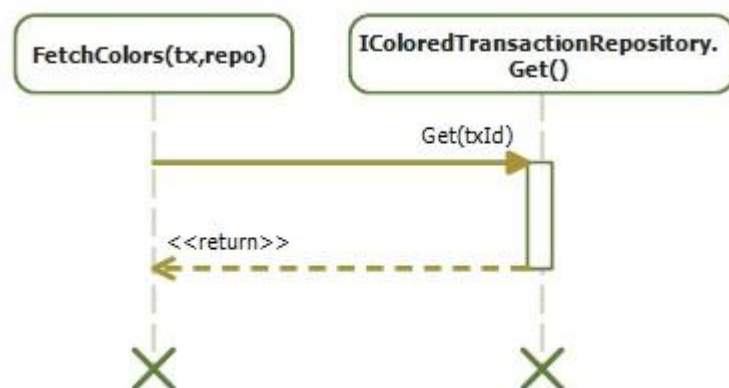


一个 **IColoredTransactionRepository** 仅仅是一个商店，它从 txid 给你 **ColoredTransaction**。然而，你会发现它依赖于 **ITransactionRepository** 将交易 ID 映射到它的交易。

**IColoredTransactionRepository** 的一个实现就是 **CoinprismColoredTransactionRepository**，颜色币操作的公共 API。

然而，你可以自己轻松地完成，下面描述了 **FetchColors** 是如何工作的。

最简单的例子就是：**IColoredTransactionRepository** 知道颜色币，在这个例子中 **FetchColors** 仅仅返回结果。



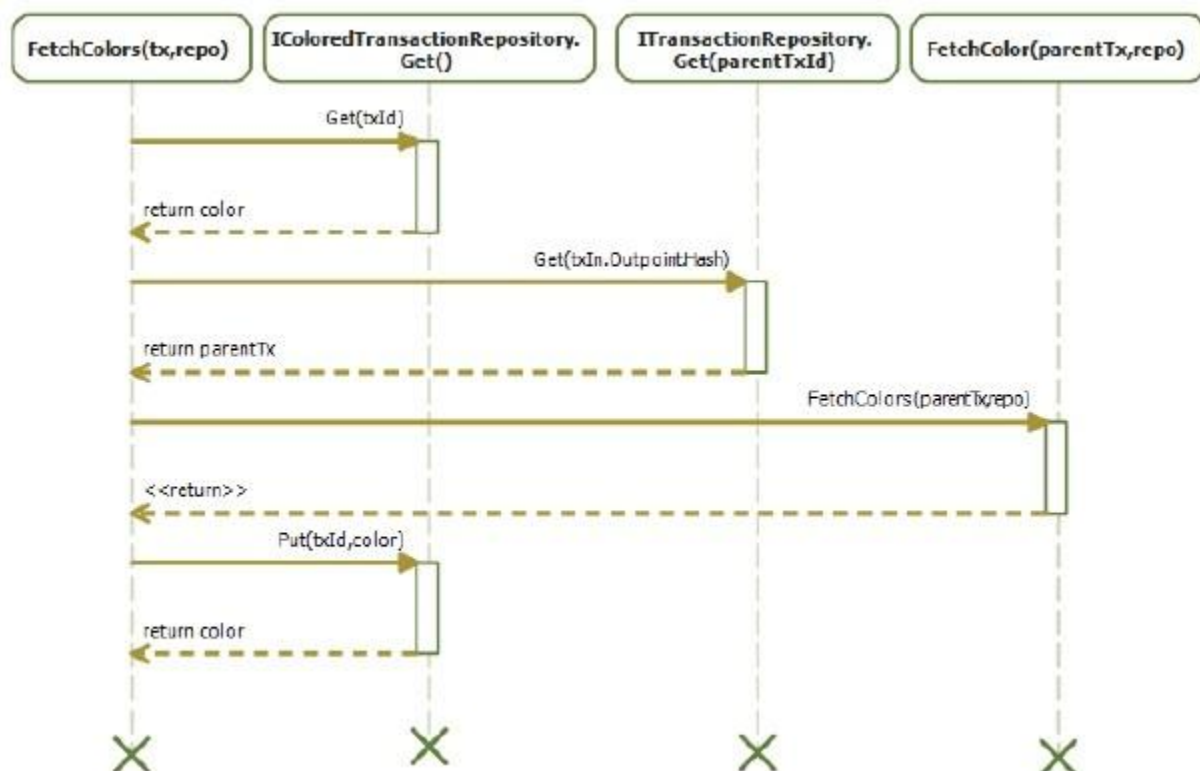
第二个例子中，**IColoredTransactionRepository** 不知道交易中关于颜色币的任何信息。

所以 **FetchColors** 需要根据公开资产详述来计算出颜色币。

但是，为了计算颜色币，**FetchColors** 需要父交易的颜色币。

所以它就从 `ITransactionRepository` 得到他们，并在他们身上调用 `FetchColors`。

一旦 `FetchColors` 递归解决了父交易颜色币，它就计算交易颜色币，并把结果保存到 `IColoredTransactionRepository` 的缓存。



按照那样做的话，未来提取交易颜色币的请求都可以很快解决了。

有些 `IColoredTransactionRepository` 是只读的（就好像 `CoinprismColoredTransactionRepository`，`Put` 操作被忽略了）

那么，回到我们的例子：

编写单元测试的技巧就是使用内存中的 `IColoredTransactionRepository`：

```
var repo = new NoSqlColoredTransactionRepository();
```

现在，我们可以将 `init` 放到交易里面了：

```
repo.Transactions.Put(init);
```

注意 `Put` 是一个扩展的方法，所以你需要添加如下代码：

```
using NBitcoin.OpenAsset;
```

放在文件顶部以便访问。

现在，你可以抽取颜色币了：

```
ColoredTransaction color = ColoredTransaction.FetchColors(init, repo);
Console.WriteLine(color);
```

```
{  
  "inputs": [],  
  "issuances": [],  
  "transfers": [],  
  "destructions": []  
}
```

结果正如预期，init 交易没有输入，发行方，传输或者颜色币的销毁。

现在，我们把发往 Silver 和 Gold 的两个币当做是发行币。

```
var issuanceCoins =  
    init  
    .Outputs  
    .AsCoins()  
    .Take(2)  
    .Select((c, i) => new IssuanceCoin(c))  
    .OfType<ICoin>()  
    .ToArray();
```

Gold 是第一个币，Silver 是第二个。

按照上面做法，你可以通过 TransactionBuilder 将 Gold 发送给中本聪，就跟我们在前面的练习中一样，将交易放到仓库中，并打印结果。

```
var sendGoldToSatoshi =  
builder  
    .AddKeys(gold)  
    .AddCoins(issuanceCoins[0])  
    .IssueAsset(satoshi, new Asset(goldId, 10))  
    .SetChange(gold)  
    .BuildTransaction(true);  
repo.Transactions.Put(sendGoldToSatoshi);  
color = ColoredTransaction.FetchColors(sendGoldToSatoshi, repo);  
Console.WriteLine(color);
```



```
{
  "inputs": [],
  "issuances": [
    {
      "index": 0,
      "asset": "ATEwaRSNeCgBjxcu7JtfypFjqQgAtLJs",
      "quantity": 10
    }
  ],
  "transfers": [],
  "destructions": []
}
```

这意味着第一个 TxOut 承担 10 个 gold。

假设中本聪想要发送 4 个 gold 给 Alice。

首先他将从交易中提取颜色币。

```
var goldCoin = ColoredCoin.Find(sendGoldToSatoshi, color).FirstOrDefault();
```

然后，如下建立一个交易：

```
builder = new TransactionBuilder();
var sendToBobAndAlice =
    builder
        .AddKeys(satoshi)
        .AddCoins(goldCoin)
        .SendAsset(alice, new Asset(goldId, 4))
        .SetChange(satoshi)
        .BuildTransaction(true);
```

除非你得到异常 `NotEnoughFundsException`。

原因是交易包含 600 聪的输入（goldCoin），1200 聪的输出。（一个 TxOut 用于发送资产给 Alice，一个用于发回找零给 Satoshi）

这就意味着你发出去 600 聪了。

你可以在 satoshi 的 init 交易中增加最后 1BTC，从而解决问题。

```
var satoshiBtc = init.Outputs.AsCoins().Last();
builder = new TransactionBuilder();
var sendToAlice =
    builder
        .AddKeys(satoshi)
        .AddCoins(goldCoin, satoshiBtc)
        .SendAsset(alice, new Asset(goldId, 4))
```

```

        .SetChange(satoshi)
        .BuildTransaction(true);
repo.Transactions.Put(sendToAlice);
color = ColoredTransaction.FetchColors(sendToAlice, repo);

Let's see the transaction and its colored part:
Console.WriteLine(sendToAlice);
Console.WriteLine(color);
{
    ...
    "in": [
        {
            "prev_out": {
                "hash":
"46117f3ef44f2dfd87e0bc3f461f48fe9e2a3a2281c9b3802e339c5895fc325e",
                "n": 0
            },
            "scriptSig":

"304502210083424305549d4bb1632e2c67736383558f3e1d7fb30ce7b5a3d7b87a53cdb3
940220687
          ea53db678b467b98a83679dec43d27e89234ce802daf14ed059e7a09557e801
03e232cda91e719075a95ede4c36ea1419efbc145afd8896f36310b76b8020d4b1"
        },
        {
            "prev_out": {
                "hash":
"aefa62270999baa0d57ddc7d2e1524dd3828e81a679adda810657581d7d6d0f6",
                "n": 2
            },
            "scriptSig":

"30440220364a30eb4c8a82cc2a79c54d0518b8ba0cf4e49c73a5bbd17fe1a5683a0dfa64
0220285e98f
          3d336f1fa26fb318be545162d6a36ce1103c8f6c547320037cb1fb8e901
03e232cda91e719075a95ede4c36ea1419efbc145afd8896f36310b76b8020d4b1"
        }
    ],
    "out": [
        {
            "value": "0.00000000",
            "scriptPubKey": "OP_RETURN 4f41010002060400"
        }
    ]
}

```

```

    },
    {
      "value": "0.00000600",
      "scriptPubKey": "OP_DUP OP_HASH160
5bb41cd29f4e838b4b0fdcd0b95447dcf32c489d
OP_EQUALVERIFY OP_CHECKSIG"
    },
    {
      "value": "0.00000600",
      "scriptPubKey": "OP_DUP OP_HASH160
469c5243cb08c82e78a8020360a07ddb193f2aa8
OP_EQUALVERIFY OP_CHECKSIG"
    },
    {
      "value": "0.99999400",
      "scriptPubKey": "OP_DUP OP_HASH160
5bb41cd29f4e838b4b0fdcd0b95447dcf32c489d
OP_EQUALVERIFY OP_CHECKSIG"
    }
  ]
}
Colored :
{
  "inputs": [
    {
      "index": 0,
      "asset": " ATEwaRSNeCgBjxjcur7JtfypFjqQgAtLJs ",
      "quantity": 10
    }
  ],
  "issuances": [],
  "transfers": [
    {
      "index": 1,
      "asset": " ATEwaRSNeCgBjxjcur7JtfypFjqQgAtLJs ",
      "quantity": 6
    },
    {
      "index": 2,
      "asset": " ATEwaRSNeCgBjxjcur7JtfypFjqQgAtLJs ",
      "quantity": 4
    }
  ],
  "destructions": []
}

```

```
}
```

我们最终创建了一个单元测试，并且发行和传输了若干资产，其中没有依赖任何外部帮助。如果你不想依赖于第三方机构服务的话，你可以创建自己的 `IColoredTransactionRepository`。在 `NBitcoin` 测试中你可以找到更多复杂的场景，在我 `codeproject` 的文章“[Build them all](#)”中也有很多。（比如多重签名发行和颜色币互换）。

## 5.5 李嘉图合约

这部分是我在 `Coinprism` 博客上的一篇文章。在撰写本文时，`NBitcoin` 还没有关于李嘉图合约的代码。

### 5.5.1 什么是李嘉图合约

一般来说，一项资产代表在一定条件下向发行者要求赎回或者兑付的权利。

- 公司股份给予获取股息的权利，
- 债券给予到期兑付本金的权利，息票代表每期获取利息的权利，
- 投票令牌给予相关实体投票决策的权利（公司，选举）
- 可能是以上几项的混合：股份同时也是公司领导人选举的投票令牌

这些权利通常在一项合约里面列举出来，并由发行人签名。（如果需要，有一个可信第三方，比如认证机构） 一项李嘉图合约是由发行人加密签名的合约，不可能与资产脱钩。 所以合约不能否认、篡改，并且可以证明就是发行人签名的。 这种合约可以在发行人和赎回人之间保密，或者公开。 开放资产已经可以支持以上所有事项，除了改变核心协议。下面是如何改变核心协议的内容。

### 5.5.2 开放资产里面的李嘉图合约

这是李嘉图合约的正式定义：

1. 一项由发行人提供给持有人的合约，
2. 服务于持有人拥有、发行人管理的有价值权利，
3. 人工可读（就像是纸质合约），
4. 程序可读（就像数据库那样可以解析），

5. 数字签名,
6. 带有秘钥和服务信息
7. 附有唯一的安全识别标志。

资产 ID 由 OpenAsset 按下列方式具体定义:

```
AssetId = Hash160(ScriptPubKey)
```

把 ScriptPubKey 转换成 P2SH 形式:

```
ScriptPubKey = OP_HASH160 Hash(RedeemScript) OP_EQUAL
```

这里:

```
RedeemScript = HASH160(RicardianContract) OP_DROP IssuerScript
```

对于一个简单的发行人而言, IssuerScript 表示一个典型的 P2PKH, 如果发行人需要多个同意意见, 它表示多重签名。(举例来说: issuer + notary )

需要注意的是, 从比特币 0.10 开始, IssuerScript 是机动灵活的, 可以是任何事情。

李嘉图是机动灵活的, 并且可以保持私密性。任何持有合约的人都可以通过 ScriptPubKey 哈希值证明, 合约是适用于这笔资产的。

在资产定义协议下, 这些李嘉图合约在客户钱包面前是可发现和可验证的。

假设我们正在向候选人 A、B、C 发行投票令牌。

增加开放资产标记的内容, 加入下列资产定义网址 : `u=http://issuer.com/contract`

在网页 <http://issuer.com/contract> 中, 创建下列资产定义文件:

```
{
  "IssuerScript" : IssuerScript,
  "name" : "MyAsset",
  "contract_url" : "http://issuer.com/readableContract",
  "contract_hash" : "DKDKocezifefiouOIUOIUOlufiez980980",
  "Type" : "Vote",
  "Candidates" : ["A","B","C"],
  "Validity" : "10 jan 2015"
}
```

现在我们可以定义李嘉图合约了:

```
RicardianContract = AssetDefinitionFile
```

这就结束了我们在 OA 中的李嘉图合约实现

### 5.5.3 检查列表

由发行人提供给持有人的合约

合约由发行人管理，不可更改，发行人每次发行一项新资产就签名一次。

服务于持有人拥有、发行人管理的有价值权利

在这个例子中的权利就是候选人 A、B、C 的选举权利，可以在 2015 年 1 月 10 日以前赎回。

人工可读（就像是纸质合约）

人工可读的合约可在 `contract_url` 看到，但是 JSON 可能已经足够了。

程序可读（就像数据库那样可以解析）

选举的细节在 `AssetDefinitionFile` 里面，是 JSON 格式的，合约的真实性由 `IssuerScript` 通过软件验证，哈希值在 `ScriptPubKey` 里面。

数字签名

当发行人发行资产时，`ScriptPubKey` 就被签名，并且合约的哈希值以及合约本身也被签名。

带有秘钥和服务信息

`IssuerScript` 包含在合约里面

附有唯一的安全识别标志。

`AssetId` 由 `Hash(ScriptPubKey)` 定义，并且是唯一、不能更改的。

### 5.5.4 它有什么作用？

没有李嘉图合约，恶意的发行人就很容易更改或者否认资产定义文件。

李嘉图合约保证了不可否认性，让合约不可更改，因此它在赎回人和发行人之间建立起了仲裁机制。

同时，由于资产定义文件不可更改，就可以把它保存在赎回人自己的存储上，避免了恶意发行人断开合约访问。

## 5.6 流动的民主

### 5.6.1 概览

这部分纯粹是颜色币一项应用的概念性练习。设想这样一个公司，投资者委员会经过投票作出决策。

- 一些投资者不知道主题内容，所以他们想委托其他一些人作出决策
- 投资者的数量非常之大
- 作为 CEO，你想拥有为公司融资而销售投票权的能力
- 作为 CEO，你想在作出决定后，可以投出一票

颜色币如何可以透明地组织这样的选举？

开始前，我们探讨一下区块链上投票的一些背景：

- 没有人知道投票人的真实 ID，
- 矿工可以审查（就算它是可证明的，而且不是出于自利）
- 虽然没有人知道投票人的真实 ID，在几轮投票后通过对投票人进行行为分析可以识别出他的 ID

这几点是否关联取决于投票组织者的决策。我们先大概看看它的实现。

### 5.6.2 发行投票权

一切从公司创始人开始（我们叫他老板），他想销售公司的决策权给一些投资者。决策权采取颜色币的形式，在本练习中，我们称之为“权力币”。

我们以紫色表示它：



假设 3 个人感兴趣，Satoshi、Alice 和 Bob。（是的，又是他们 3 个）

老板决定按照 0.1BTC/权力币的价格向他们每个人出售。

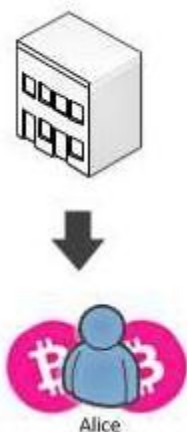
我们开始向权力币的地址发送一些货币，包括 Satoshi、Alice 和 Bob。

```
var powerCoin = new Key();
```

```
var alice = new Key();
var bob = new Key();
var satoshi = new Key();
var init = new Transaction()
{
    Outputs =
    {
        new TxOut(Money.Coins(1.0m), powerCoin),
        new TxOut(Money.Coins(1.0m), alice),
        new TxOut(Money.Coins(1.0m), bob),
        new TxOut(Money.Coins(1.0m), satoshi),
    }
};

var repo = new NoSqlColoredTransactionRepository();
repo.Transactions.Put(init);
```

设想 Alice 购买了 2 个权力币，创建交易如下：



```
var issuance = GetCoins(init, powerCoin)
    .Select(c => new IssuanceCoin(c))
    .ToArray();
var builder = new TransactionBuilder();
var toAlice =
builder
    .AddCoins(issuance)
    .AddKeys(powerCoin)
    .IssueAsset(alice, new Asset(powerCoin, 2))
    .SetChange(powerCoin)
    .Then()
    .AddCoins(GetCoins(init, alice))
    .AddKeys(alice)
    .Send(alice, Money.Coins(0.2m))
```



```

        .SetChange(alice)
        .BuildTransaction(true);
repo.Transactions.Put(toAlice);

```

总的来说，权力币向 Alice 发行了 2 个币，并把零钱发送给自己。类似，Alice 发送 0.2BTC 给权力币，并把零钱发送给她自己。

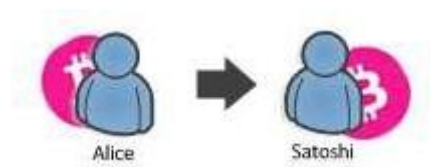
GetCoins 函数如下：

```

private IEnumerable<Coin> GetCoins(Transaction tx, Key owner)
{
    return tx.Outputs.AsCoins().Where(c => c.ScriptPubKey ==
owner.ScriptPubKey);
}

```

由于一些原因，Alice 可能想要出售一些投票权给 Satoshi。



```

builder = new TransactionBuilder();
var toSatoshi =
builder
    .AddCoins(ColoredCoin.Find(toAlice, repo))
    .AddCoins(GetCoins(init, alice))
    .AddKeys(alice)
    .SendAsset(satoshi, new Asset(powerCoin, 1))
    .SetChange(alice)
    .Then()
    .AddCoins(GetCoins(init, satoshi))
    .AddKeys(satoshi)
    .Send(alice, Money.Coins(0.1m))
    .SetChange(satoshi)
    .BuildTransaction(true);
repo.Transactions.Put(toSatoshi);

```

你可以注意到我正从 init 交易中双花 Alice 的币。

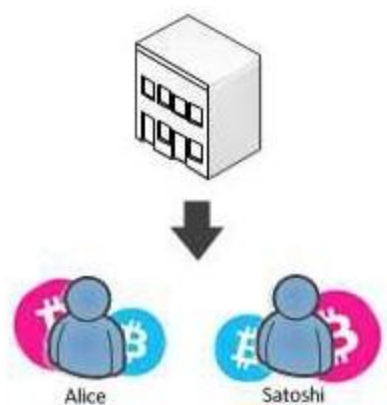
这种事情在区块链上是不被允许的。然而，我们还没有看到如何从区块链上轻松地取回未花出去的币，所以我们在本练习中暂且设想并没有双花。

现在 Alice 和 Satoshi 都有投票权了，看看老板如何举行一次投票选举。

### 5.6.3 投票选举

通过询问区块链，老板任何时候都可以知道 `ScriptPubKeys`，它拥有权力币。

因此他按投票权力比例发送投票币给这些所有者，在我们的例子中，给 Alice 和 Satoshi 各一个投票币。



首先，我需要为 `votingCoin` 创建一些资金池。

```
var votingCoin = new Key();
var init2 = new Transaction()
{
    Outputs =
    {
        new TxOut(Money.Coins(1.0m), votingCoin),
    }
};
repo.Transactions.Put(init2);
```

然后，就是发行投票币。

```
issuance = GetCoins(init2, votingCoin).Select(c => new
IssuanceCoin(c)).ToArray();

builder = new TransactionBuilder();
var toVoters =
    builder
        .AddCoins(issuance)
        .AddKeys(votingCoin)
        .IssueAsset(alice, new Asset(votingCoin, 1))
        .IssueAsset(satoshi, new Asset(votingCoin, 1))
        .SetChange(votingCoin)
        .BuildTransaction(true);
repo.Transactions.Put(toVoters);
```

## 5.6.4 投票代理

问题是投票牵涉到一些商业上的金融事项，而 Alice 比较关心市场方面的情况。

她的策略就是把投票币委托给她认为对金融事项有比较准确判断的人。她选择将投票委托给 Bob。



```
var aliceVotingCoin = ColoredCoin.Find(toVoters,repo)
    .Where(c=>c.ScriptPubKey == alice.ScriptPubKey)
    .ToArray();
builder = new TransactionBuilder();
var toBob =
    builder
        .AddCoins(aliceVotingCoin)
        .AddKeys(alice)
        .SendAsset(bob, new Asset(votingCoin, 1))
        .BuildTransaction(true);
repo.Transactions.Put(toBob);
```

你会注意到这里没有 `SetChange`，原因是输入的颜色币被全部花掉了，所以没有剩下的供返回。

## 5.6.5 投票

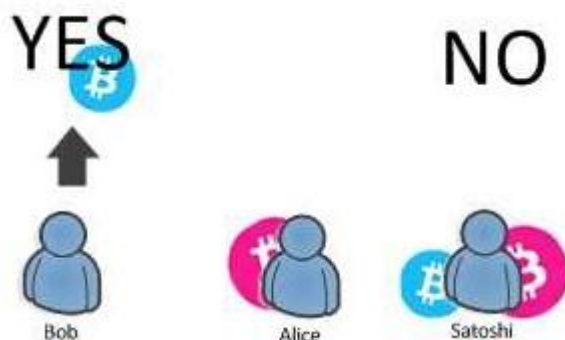
设想 Satoshi 太忙了决定不去投票。现在 Bob 必须表明自己的观点。投票议题是公司是否应该向银行申请一笔贷款用于投资新生产机器。

老板在公司网站上说明：

发送你的币到 1HZwkjkeaoZfTSaJxDw6aKkxp45agDiEzN 表示同意，

到 1F3sAm6ZtwLAUnj7d38pGFxtP3RVEvtsbV 表示不同意。

Bob 认为公司应该申请这笔贷款：



```
builder = new TransactionBuilder();
var vote =
    builder
        .AddCoins(bobVotingCoin)
        .AddKeys(bob)
        .SendAsset(BitcoinAddress.Create("1HZwkjkeaoZfTSaJxDw6aKkxp45agDiEzN"
),
            new Asset(votingCoin, 1))
        .BuildTransaction(true);
```

现在老板可以计算出投票结果了，1 票赞同，0 票反对，结果是同意。因此他申请了贷款。  
每一个参与者都可以自己计算出结果。

## 5.6.6 替代方案：使用李嘉图合约

在签名的练习中，我们假设老板在区块链之外声明投票的形式，也就是在公司网站上。

这样很好，但是 Bob 需要知道网址在哪里。

另一种解决方案就是把投票形式直接发布在区块链上，在资产定义文件里面，这样一些软件就可以自动地获得并展现给 Bob。

唯一需要更改的代码段就是发行投票币给投票人员。

```
issuance = GetCoins(init2, votingCoin).Select(c => new
IssuanceCoin(c)).ToArray();
issuance[0].DefinitionUrl = new Uri("http://boss.com/vote01.json");
builder = new TransactionBuilder();
var toVoters =
    builder
        .AddCoins(issuance)
        .AddKeys(votingCoin)
        .IssueAsset(alice, new Asset(votingCoin, 1))
        .IssueAsset(satoshi, new Asset(votingCoin, 1))
        .SetChange(votingCoin)
```

```
.BuildTransaction(true);  
repo.Transactions.Put(toVoters);
```

本例中，Bob 可以看到，在发行他的投票币期间，一份资产定义文件被发布了，这份文件是 JSON 格式文档，部分格式定义在开放资产中。

格式定义可以被扩展到一些信息，比如：

- 投票期限
- 每位候选人的投票地址
- 人工可读的描述

然而，设想有一位黑客想要在投票中作弊。他总可以篡改 json 文档（中间人攻击、物理访问 boss.com 或者访问 Bob 的机器），由此 Bob 就被欺骗了，然后投票给错误的候选人。

通过签名将资产定义文件转换为李嘉图合约，将使得任何改变 Bob 的软件都可以实时监测到。（参照在资产定义协议中的真实性证明）

## 5.7 烧钱和声誉证明

问题很简单：在 P2P 市场里面，如果执法成本太高，参与者如何最大限度降低被骗的概率呢？

OpenBazaar 貌似是第一个试图使用烧钱证明作为声誉评判机制的。

对那个问题有好几种解决方案，但是这里我们就讲讲烧钱证明。

设想你自己在中世纪，生活在一个小山村里面，陪伴你的有一些本地商人。一天，一个行销商走进你的村子，卖给你一些商品，并且价格与本地相比低得让人不敢相信。

然而，行销商因为劣质商品欺骗顾客而臭名昭著，但是与本地商人相比，他们付出的代价相当低。

本地商人投资一个实惠的商铺、做广告和营造声誉。不满意的顾客可以很容易毁掉他们。但是行销商人没有本地商铺，临时的声誉不足以让他们不欺骗人们。

在互联网上，创建标识是如此地廉价，以至于所有商人潜在都是中世纪的行销商。

市场供应商的解决方案就是收集市场所有参与方的真实标识，这样执法才有可能。

如果你在 Amazon 上被 Ebay 骗了，你的开户行很可能会返还款项，因为他们有方法通过 Amazon 和 Ebay 找到窃贼。

在一个使用比特币的纯 P2P 市场里面，我们没有那些设施。如果你被骗，那就损失金钱了。

那么顾客怎么能信任一个行销商呢？

答复就是：检查一下他给自己的声誉投资了多少

作为一个专业的销售商，你想提高顾客对你的信任。你以部分财富为代价建立声誉，让每位顾客都可以看得到。这就是“投资于你的声誉”的定义。

设想你为自己的声誉烧掉 50BTC。有个顾客需要你身上购买 2BTC 的物品。他有很好的理由相信你不会欺骗他，因为你对声誉的投资比欺骗他得到的更多。

通过欺骗他获利就变得经济上不划算。

技术细节必然会随时间而变化，下面是烧钱证明的例子：

```
var alice = new Key();

//Giving some money to alice
var init = new Transaction()
{
    Outputs =
    {
        new TxOut(Money.Coins(1.0m), alice),
    }
};

var coin = init.Outputs.AsCoins().First();

//Burning the coin
var burn = new Transaction();
burn.Inputs.Add(new TxIn(coin.Outpoint)
{
    ScriptSig = coin.ScriptPubKey
}); //Spend the previous coin

var message = "Burnt for \"Alice Bakery\"";
var opReturn = TxNullDataTemplate
.Instance
.GenerateScriptPubKey(Encoding.UTF8.GetBytes(message));
burn.Outputs.Add(new TxOut(Money.Coins(1.0m), opReturn));
burn.Sign(alice, false);

Console.WriteLine(burn);
{
    ...
    "in": [
    {
```

```
    "prev_out": {
      "hash":
"0767b76406dbaa95cc12d8196196a9e476c81dd328a07b30954d8de256aa1e9f",
      "n": 0
    },
    "scriptSig":

"304402202c6897714c69b3f794e730e94dd0110c4b15461e221324b5a78316f97c4dffab
0220742c81
1d62e853dea433e97a4c0ca44e96a0358c9ef950387354fbc24b8964fb01
03fedc2f6458fef30c56cafd71c72a73a9ebfb2125299d8dc6447fdd12ee55a52c"
  }
],
  "out": [
    {
      "value": "1.00000000",
      "scriptPubKey": "OP_RETURN
4275726e7420666f722022416c6963652042616b65727922"
    }
  ]
}
```

在区块链上，这个交易是不可否认的证明，证明 Alice 向她的面包店投资了。

ScriptPubKey OP\_RETURN 为 4275726e7420666f722022416c6963652042616b65727922 币再也没有办法花掉了，所以这些币就永远丢失了。

## 5.8 存在性证明

1. 比特币发展的挑战
2. 如何证明一个币存在于区块链上
3. 如何证明一个颜色币存在于区块链上
4. 断开与第三方 API 的信任关系
5. 防止延展性攻击
6. 保护你的私钥