问题三

请选择一个关于区块头的正确描述。

（1）前面一个区块的区块头的哈希值包含在后面一个区块的区块头中。
（2）区块头中包含所有交易。
（3）区块头的数据容量为 8 字节。

第8章 地址

在区块链中，地址用来识别特定的信息。地址是由公开密钥生成的，而公开密钥是利用椭圆曲线加密由私密密钥生成的。下面我们来一步一步地了解地址生成的过程。

8.1 地址生成的过程

本节主要介绍地址是如何生成的。

8.1.1 什么是地址

在发送和接收像比特币这样的虚拟货币时会使用地址。地址就是为了识别交易的另一方而传送给多人的信息。

由于地址是对多人可见的，因此它的可读性很高，但是，它又是与个人相关的具有高度隐私性的信息。为了满足这看似矛盾的公开性和保密性条件，在生成地址之前应用了多种加密技术。

8.1.2 地址生成概述

如果从源头开始追溯地址会追溯到私密密钥。从私密密钥到地址生成的过程如图 8.1 所示。

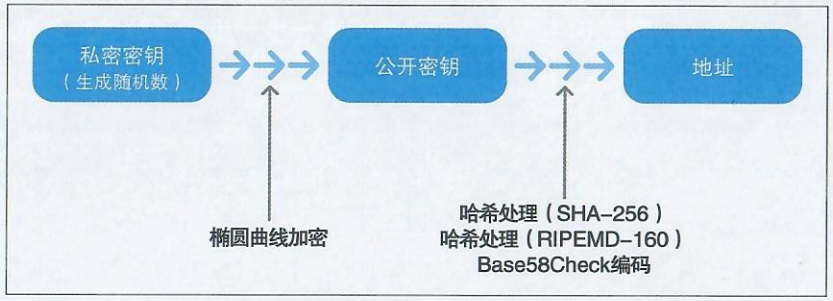

图 8.1 从私密密钥到地址生成的过程

首先，以随机数生成私密密钥。之后，通过称为椭圆曲线加密的密码技术生成一个公开密钥，然后将其两次进行哈希处理，并通过 Base58Check 编码来生成地址。

8.2 私密密钥的生成

本节介绍私密密钥的生成过程。

8.2.1 私密密钥的种子是“随机数”

私密密钥本质上是一个随机数。随机数是随机生成的数字。以比特币的区块链为例，私密密钥是 1 到 2256 之间的整数，每一次都是随机生成。

随机数是否真的是随机数而不是具有一定规则的数字，这是非常重要的一点。同时，也是很难判断的一点。例如，如果在区块链中使用的随机数是基于某种规律性的，则某些抱有恶意的第三方将能够根据这种规律推测出私密密钥。在区块链中，私密密钥代表某种资产本身的所有权，是否能够被推测起着决定成败的作用。因此，随机数生成器必须要利用先进的技术以确保其随机性，诸如 Python 等程序设计语言提供了用于生成随机数的库。

8.2.2 生成一个私密密钥

使用清单 8.1 所示的代码生成一个随机数并获取私密密钥。这里，我们将使用 Python 的标准模块 os 和 binascii。因此，有必要在开始时导入并调用这两个库。

清单 8.1 私密密钥的生成

输入

```
import os
import binascii

private_key = os.urandom(32)

print(private_key)
print(binascii.hexlify(private_key))
```

输出

```
b'"vi\xb8\xf5\x10\x7f0F\xc54\xfd\xca>g\xd3\xd8\xe7\xac➡
\xb9\x16\xb0M\xe9\xde\x8eiN\x1d\xca"\xf9'
b'227669b8f5107f3046c534fdca3e67d3d8e7acb916b04de9de8e➡
694e1dca60f9'
```

通过清单 8.1 所示的 os.urandom（32）部分产生一个 32 字节的随机数，再通过 binascii.hexlify() 将二进制数据转换为十六进制。进行这个转换是为了使其输出后更易于理解。

通过执行这段代码，可以生成如清单 8.1 中输出部分的私密密钥。另外，由于每次生成的随机数不同，所以每次执行的输出结果都会有所不同。

8.3 公开密钥的生成

本节主要介绍公开密钥生成的过程。虽然使用了诸如椭圆曲线加密之类的密码技术，但是由于该技术是不断发展的技术，因此在本节的后半部分主要介绍密码技术的发展过程。如果有兴趣，读者可以结合起来一起学习。随着对椭圆曲线的理解的深入，对公开密钥的理解也将不断深入。

8.3.1 生成一个公开密钥

根据私密密钥来生成一个公开密钥。直接自己创建一个对于生成公开密钥所必不可少的椭圆曲线密码是非常困难的，同时还需要考虑到安全性的问题，因此，不推荐这种做法。可以通过使用第三方库 ecdsa 来实现椭圆曲线加密。如清单 8.2 所示显示了从私密密钥生成公开密钥的代码。

●［终端］

```
$ pip install ecdsa
```

清单 8.2　公开密钥的生成

输入

```
import os
import ecdsa
import binascii

private_key = os.urandom(32)
public_key = ecdsa.SigningKey.from_string(private_key, ➡
curve=ecdsa.SECP256k1).verifying_key.to_string()

print(binascii.hexlify(private_key))
print(binascii.hexlify(public_key))
```

输出

```
b'9a8486a2494c5c62ac6df097f6beb7c453ece6ccedbd25d28b6b➡
ed19abe9e024'
b'd5c37c48d5cd6f52a21cffb95a263affcccd6638b975eafc4e60➡
6f7cf7b1463ec741d07e989f624ed70b53a5fb33b2a0e89bd5617f➡
c95e673173feff2c41e3da'
```

如 8.2.2 小节所述，通过生成一个随机数来生成私密密钥 private_key。通过将结果传递给如清单 8.3 所示的代码，来利用椭圆曲线加密技术。在 .from_string 的两个参数中，第一个参数是代入私密密钥，第二个参数是代入椭圆曲线。

清单 8.3　椭圆曲线加密技术的使用

```
ecdsa.SigningKey.from_string(private_key,curve=ecdsa.SECP256➡
k1).verifying_key.to_string()
```

以本书的开发环境为例，如清单 8.2 所示代码的输出结果如上述输出部分所示。在程序的开始处生成随机数，并生成私密密钥和公开密钥，由于随机数不同，因此每次执行的输出结果也会发生变化。

8.3.2（发展篇）椭圆曲线

与私密密钥不同，**公开密钥**是对许多人开放的信息。但是，如果可以利用这一任何人都可以看到的公开密钥能够逆向计算私密密钥的话，那么就完全无

法称为加密技术。因此，我们使用一种被称为“**椭圆曲线密码学**”的非常先进的密码技术来生成公开密钥。椭圆曲线加密技术具有像哈希函数一样的单向性（不可逆性），具有无法逆向运算的特性。

椭圆曲线加密技术使用下面的椭圆曲线公式。

$$y^2 = x^3 + ax + b$$

该公式的图会根据 a 和 b 的值的变化而变化，在比特币的区块链的情况下，使用下述公式，其中 $a=0$ 和 $b=7$。而该公式则定义为 secp256k1。

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

该公式的图形如图 8.2 所示。

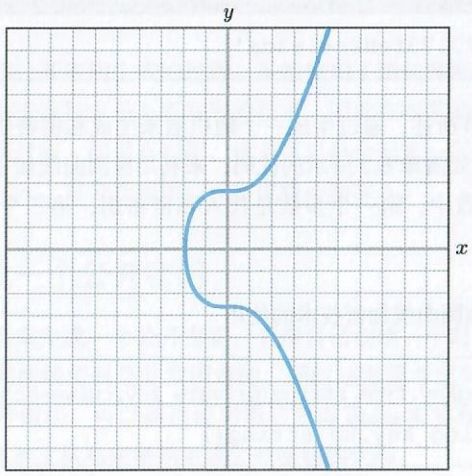

图 8.2　$y^2 = x^3 + 7$ 的图

8.3.3（发展篇）椭圆曲线加密

椭圆曲线加密所使用的公式表示如下。

$$y^2 = x^3 + 7 \bmod p$$

在此，mod 是表示余数运算的，称为**求余（模）**。例如，7 mod 3 表示 1 为 7 除以 3 的余数。因此，上面的等式表明，右侧 x^3+7 除以 p 的余数等于左侧 y^2。

这是离散对数问题的性质的一种应用。当已知某个常数 g 和质数 p 时，很容易通过公式 $y = gx \bmod p$ 求得 y；但是反过来，很难通过 y 求得 x 的值。如果进一步深入的话，将进入高级加密技术的世界，因此可以明白从 y 推导出 x 将花费大量的计算和时间，也就是说，椭圆曲线加密函数像哈希函数一样很难进行逆向计算，能够这样理解就足够了。

8.3.4（发展篇）从私密密钥生成公开密钥

本小节主要介绍如何利用椭圆曲线加密技术由私密密钥生成公开密钥。

首先，设置椭圆曲线 $y^2 = x^3 + ax + b \bmod p$ 的参数 a、b、p 和基点 $G(x,y)$。正如比特币中使用的 secp256k1 已经介绍的那样，参数 a 和 b 有如下的值。

$$a = 0x0000000000000000000000000000000000000000000000000000000000000000$$
$$b = 0x0000000000000000000000000000000000000000000000000000000000000007$$

对于比特币来说，将下述的点预先设置为基点 G。

$$Gx = 0x79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798$$
$$Gy = 0x483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8$$

另外，将以下的数值（十六进制数）设置为 p 的值。

$$p = 0xfffffffffffffffffffffffffffffffffffffffffffffffffffffffefffffc2f$$

在这里，我们将解释说明椭圆曲线上的加法。与通常的加法和乘法不同，椭圆曲线上的加法是利用椭圆曲线上的切线，如图 8.3 所示。在椭圆曲线上标识一个点 G 并为其绘制切线，由于曲线的性质，切线将始终与椭圆曲线上除切线 G 之外的其他点发生接触。与其接触的点和 x 轴的对称点为 $2G$。此属性应用在生成公开密钥上。

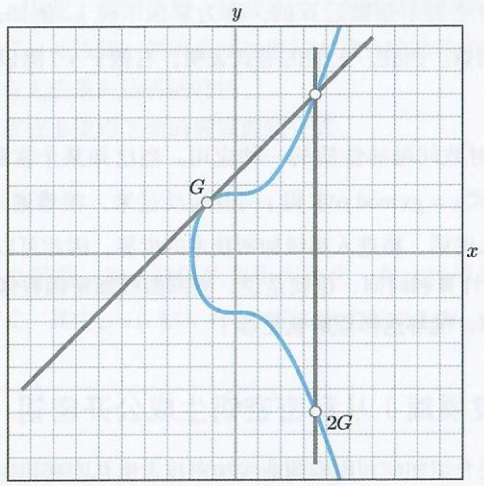

图 8.3　椭圆曲线上的加法

现在让我们详细了解生成公开密钥的细节。首先，对于刚才介绍的基点 G，直接加上 G，求得 $2G$。重复私密密钥的值的次数做如上操作。该计算的结果，即所获得的值为公开密钥。

通过多次地执行上述操作，可以获得最终的点 nG，并将其作为公开密钥。由于点 nG 需要计算大量的次数，因此需要由计算机来完成计算。

另外，作为一个重要的性质，由于使用了 mod，因此可以说试验次数越多，从终点 nG 的信息中识别出 n（即试验次数）就越困难。该性质表现出与哈希函数相同的单向性，因此解密是非常困难的。在这种情况下，想要确定试验次数 n 与想要确定私密密钥的值是同等难度，因此，该性质在加密技术中非常重要。

8.3.5 公开密钥的格式

如已经介绍的那样，公开密钥是利用椭圆曲线加密技术生成的，但是其本质是坐标。也就是说，公开密钥是由椭圆曲线密码系统上的点来定义的，并由一组 x 和 y 坐标组成。如果查看一下如清单 8.2 所示中生成的公开密钥，则可以在上半部分和下半部分将其分为 x 和 y 的坐标信息。

x 坐标：$d5c37c48d5cd6f52a21cffb95a263affcccd6638b975eafc4e606f7cf7b1463e$

y 坐标：$c741d07e989f624ed70b53a5fb33b2a0e89bd5617fc95e673173feff2c41e3da$

上述生成的公开密钥大约是私密密钥容量的两倍，考虑到整个区块链的数据容量，最好进行压缩。由此，将介绍两种类型的公开密钥，即未压缩的公开密钥和压缩的公开密钥。

未压缩的公开密钥如在 8.3.4 小节中生成的公开密钥一样，其中添加了 04 作为前缀。前缀是设置在数据开始位置的具有特定含义的字符串。根据格式，前述中生成的公开密钥如图 8.4 所示。

图 8.4　非压缩公开密钥的格式

另外，压缩的公开密钥利用公开密钥是坐标数据的特点。如果函数上的点（坐标）是 x 或 y，只要搞清楚其中一方的数据，则其他数据也将可以获得。如果公开密钥是椭圆曲线加密 $y^2=x^3+7$ 的一个点，则只需要知道 x 的信息即可计算 y 的值。

但是，因为 y 是平方的，所以每个 x 值可以对应 2 个 y 值。通过图 8.5 很容易理解这一点。可以看出 y 值位于对称轴 x 轴的上方和下方（x 值相同），如图 8.5 所示。

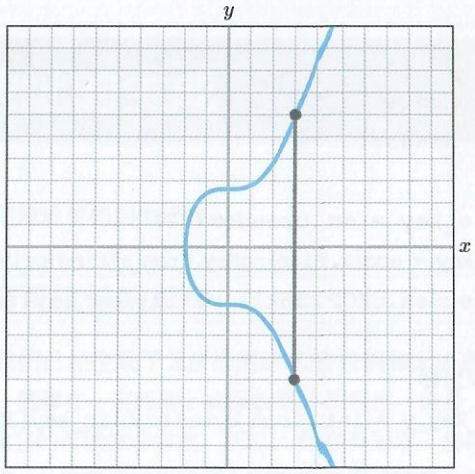

图 8.5　椭圆曲线关于 x 轴对称的图

因此，为了使一个 x 坐标确定一个 y 坐标，通过前缀进行区分。如图 8.6 所示，如果 y 的值为正数，则前缀加 02；如果 y 的值为负数，则前缀加 03。

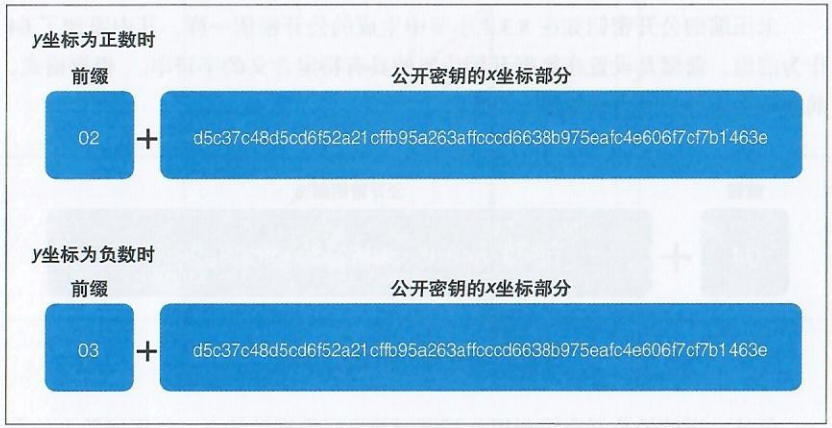

图 8.6　压缩公开密钥的格式

现在，公开密钥的信息只能包括前缀和 x 坐标信息。这使得可以将其压缩为未压缩公开密钥容量的一半大小。为了以程序方式处理此工作，可以以前缀的值为偶数或奇数进行分类，如果为偶数则为正，如果为奇数则为负，如清单 8.4 所示。

清单 8.4　压缩公开密钥的生成

输入

```
import os
import ecdsa
import binascii

private_key = os.urandom(32)
public_key = ecdsa.SigningKey.from_string(private_key, ➡
curve=ecdsa.SECP256k1).verifying_key.to_string()

# 取出y坐标
public_key_y = int.from_bytes(public_key[32:], "big")

# 压缩公开密钥的生成
if public_key_y % 2 == 0:
    public_key_compressed = b"\x02" + public_key[:32]
else:
    public_key_compressed = b"\x03" + public_key[:32]

print(binascii.hexlify(public_key))
print(binascii.hexlify(public_key_compressed))
```

输出

```
b'7e863135b895112de09f9b5492d3b3a0af75ae770a6ed08627d1➡
e279d4b2bb040a018b6083f2500d25ae1904976ae8c5cc0691bf4f➡
47c82ca5b3148858deef51'
b'037e863135b895112de09f9b5492d3b3a0af75ae770a6ed08627➡
d1e279d4b2bb04'
```

如上所述，该程序输出公开密钥以及压缩公开密钥。在输出部分最下面显示的压缩公开密钥中，可以看到公开密钥的前缀 03 和前半部分的值（x 坐标）。同样地，当同一程序代码多次执行时，由于每次都会生成不同的随机数，因此前缀 02 和 03 会经常互换，并且密钥值也会发生显著的变化。

8.4　地址的生成

本节主要介绍地址是如何生成的。

8.4.1　提高可读性的思考

目前为止，我们已经生成了无法推测的私密密钥和无法逆向计算的公开密钥。在这一阶段，能够在一定程度上确保数据的机密性。但是，由于公开密钥的字符串很长，因此它的可读性非常差。所以，有必要采取一些措施，例如编码等以防止数据的误读。具体来说，为了使人们能够更容易理解，采用了哈希函数一些特殊算法，如 RIPEMD-160 可以减少字符数量、Base58Check 可以编码等。

8.4.2 Base58

Base58 是一种编码方法，可以消除容易被人误读的字符串，并且是一种在输入或写入地址时不容易出错的方法。具体来说，就是尽量不要使用以下的字母和数字。

- 数字 0
- 英文字母 O
- 英文字母的小写
- 英文字母的大写
- +（加号）
- /（反斜杠）

8.4.3 Base58Check

由公开密钥生成地址时，使用了校验和的 Base58Check。校验和是一种检查数据是否正确的方法，通过添加由原始数据计算得出的部分数据来进行校验。

通过在 Base58Check 编码中结合校验和的方法，可以发现记录的错误。这里的校验和是利用了一个 4 字节的数据，该数据是将公开密钥的哈希值以及在其开始位置附加上版本字节（在 8.4.4 小节讲述），通过 SHA-256 进行 2 次哈希处理后得到数据开始位置的 4 字节。Base58Check 编码过程如图 8.7 所示。

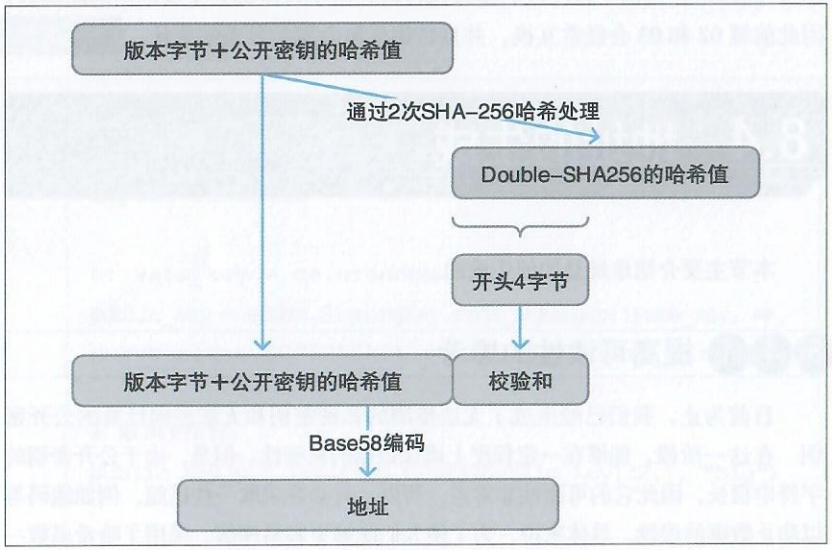

图 8.7 Base58Check 编码的过程

版本字节是十六进制数据，表示生成的地址是哪种类型的地址。版本字节又称为版本前缀（Version prefix）。版本字节通过 Base58Check 编码，转换为相同的字符。如表 8.1 所示列出了代表性的版本字节。

表 8.1 代表性的版本字节

版本字节	含 义	编码后
0x00	公开密钥的哈希	1
0x05	脚本哈希	3
0x08	私密密钥 WIF 形式	5
0x0488B21E	BIP32 扩展公开密钥	xpub

8.4.4 生成地址

公开密钥生成地址的过程如下。

（1）利用 SHA-256 将公开密钥进行哈希处理生成哈希值①。
（2）将哈希值①通过 RIPEMD-160 进行哈希处理生成哈希值②。
（3）将哈希值②进行 Base58Check 编码。

如清单 8.5 所示显示了根据上述过程生成地址的代码。在运行代码之前，需要先使用 pip 命令安装 Base58 库。另外，利用未压缩的公开密钥作为公开密钥的格式。

●［终端］

```
$ pip install base58
```

清单 8.5 地址的生成

输入

```
import os
import ecdsa
import hashlib
import base58
```

```
private_key = os.urandom(32)
public_key = ecdsa.SigningKey.from_string(private_key,➡
curve=ecdsa.SECP256k1).verifying_key.to_string()

prefix_and_pubkey = b"\x04" + public_key ——— ①

intermediate = hashlib.sha256(prefix_and_pubkey).digest()
ripemd160 = hashlib.new('ripemd160')
ripemd160.update(intermediate)
hash160 = ripemd160.digest() ——— ③

prefix_and_hash160 = b"\x00" + hash160 ——— ②

# 确认嵌套了hashlib.sha256!
double_hash = hashlib.sha256(hashlib.sha256(prefix_and➡
_hash160).digest()).digest()
checksum = double_hash[:4] ——— ④
pre_address = prefix_and_hash160 + checksum ——┐
                                              ⑤
address = base58.b58encode(pre_address) ——————┘
print(address.decode())
```

输出

```
1LnDk2TfxvsuJk71xLq28sAkS7YFnThPcA
```

下面就代码中的内容进行解说。

```
prefix_and_pubkey = b"\x04" + public_key
```

清单 8.5 ①的部分表示将未压缩的公开密钥的前缀 04 添加到公开密钥上。

```
prefix_and_hash160 = b"\x00" + hash160
```

清单 8.5 ②的部分表示将公开密钥哈希的版本前缀 00 和公开密钥哈希进行合并。关于这之前的代码，与公开密钥的生成的步骤是相同的。另外，通过 SHA-256 进行哈希处理，以及进一步通过 RIPEMD-160 进行哈希处理，统称为 HASH160。


```
hash160 = ripemd160.digest()
```

清单 8.5 ③的部分表示生成 HASH160。

接下来，在清单 8.5 ④中利用 SHA-256 进行两次哈希处理之后，取出前 4 个字节（清单 8.5 ④）。

```
checksum = double_hash[:4]
```

```
preaddress = prefix_and_hash160 + checksum

address = base58.b58encode(preaddress)
```

同样，在清单 8.5 ⑤中，取得的校验和并利用 Base58 对其进行编码，以此生成地址。通过执行如清单 8.5 所示的代码，可以获得如输出部分显示的结果。

在执行 Base58Check 编码的过程中，可以理解其与如图 8.7 所示的过程的对应关系。

本章习题

问题一

请选择一个关于对私密密钥生成地址的过程的描述不正确的选项。

（1）通过椭圆曲线加密由私密密钥生成公开密钥。
（2）地址是按照私密密钥→公开密钥→地址的顺序生成的。
（3）地址是私密密钥和公开密钥组合并进行哈希处理的值。
（4）通过 Base58 编码，将其转换为易于查看的字符串。