

Содержание

Об авторе	15
Об изображении на обложке	15
Предисловие	17
Введение	19
Для кого написана эта книга	19
Что нужно знать	20
Как организована эта книга	20
Подготовка	21
Ответы к упражнениям	25
Соглашения, принятые в этой книге	25
Примеры исходного кода	26
Благодарности	27
Ждем ваших отзывов!	30
Глава 1. Конечные поля	31
Высшая математика	31
Определение конечного поля	32
Определение конечных множеств	33
Построение конечного поля на языке Python	34
Упражнение 1	36
Арифметика по модулю	36
Арифметика по модулю на языке Python	38
Сложение и вычитание конечных полей	39
Упражнение 2	40
Программная реализация операций сложения и вычитания на языке Python	40
Упражнение 3	41
Операции умножения и возведения в степень в конечном поле	41

Упражнение 4	43
Упражнение 5	43
Программная реализация операции умножения на языке Python	43
Упражнение 6	44
Программная реализация операции возведения в степень на языке Python	44
Упражнение 7	45
Операция деления в конечном поле	45
Упражнение 8	48
Упражнение 9	48
Переопределение операции возведения в степень	48
Заключение	49
Глава 2. Эллиптические кривые	51
Определение эллиптических кривых	51
Программная реализация эллиптических кривых на языке Python	57
Упражнение 1	58
Упражнение 2	58
Сложение точек	58
Математические основы сложения точек	62
Программная реализация операции сложения точек	64
Упражнение 3	66
Сложение точек, когда $x_1 \neq x_2$	66
Упражнение 4	68
Программная реализация операции сложения точек, когда $x_1 \neq x_2$	68
Упражнение 5	68
Сложение точек, когда $P_1 = P_2$	68
Упражнение 6	70
Программная реализация операции сложения точек, когда $P_1 = P_2$	70
Упражнение 7	71
Программная реализация операции сложения точек в еще одном исключительном случае	71
Заключение	72
Глава 3. Криптография по эллиптическим кривым	73
Эллиптические кривые над вещественными числами	73
Эллиптические кривые над конечными полями	75
Упражнение 1	76

Программная реализация эллиптических кривых над конечными полями	76
Сложение точек над конечными полями	78
Программная реализация операции сложения точек над конечными полями	79
Упражнение 2	80
Упражнение 3	80
Скалярное умножение для эллиптических кривых	80
Упражнение 4	82
Еще раз о скалярном умножении	83
Математические группы	84
Тождественность	85
Замкнутость	85
Обратимость	86
Коммутативность	87
Ассоциативность	87
Упражнение 5	87
Программная реализация скалярного умножения	88
Определение кривой для биткойна	91
Работа с кривой secp256k1	93
Криптография с открытым ключом	95
Подписание и верификация	95
Надписание цели	96
Подробнее о верификации	99
Верификация подписи	101
Упражнение 6	102
Программная реализация верификации подписей	102
Подробнее о подписании	103
Создание подписи	103
Упражнение 7	105
Программная реализация подписания сообщений	105
Заключение	108
Глава 4. Сериализация	109
Несжатый формат SEC	109
Упражнение 1	111
Сжатый формат SEC	111
Упражнение 2	116

Подписи в формате DER	116
Упражнение 3	118
Кодировка Base58	118
Передача открытого ключа	119
Упражнение 4	121
Формат адреса	121
Упражнение 5	122
Формат WIF	123
Упражнение 6	124
Еще раз о прямом и обратном порядке следования байтов	124
Упражнение 7	125
Упражнение 8	125
Упражнение 9	125
Заключение	125
Глава 5. Транзакции	127
Составляющие транзакции	127
Версия	130
Упражнение 1	131
Вводы	131
Синтаксический анализ сценариев	137
Упражнение 2	137
Выводы	137
Упражнение 3	139
Время блокировки	140
Упражнение 4	141
Упражнение 5	141
Программная реализация транзакций	142
Плата за транзакцию	143
Расчет платы за транзакцию	145
Упражнение 6	145
Заключение	145
Глава 6. Язык Script	147
Внутренний механизм Script	147
Принцип действия языка Script	149
Примеры операций	150
Программная реализация операций по их кодам	151

Упражнение 1	152
Синтаксический анализ полей сценариев	152
Программная реализация синтаксического анализатора и сериализатора сценариев	153
Объединение полей сценариев	156
Программная реализация объединенного набора команд	156
Стандартные сценарии	157
p2рк	157
Программная реализация вычисления сценариев	161
Внутреннее представление элементов в стеке	163
Упражнение 2	165
Затруднения, связанные с p2рк	165
Разрешение затруднений средствами p2ркh	166
p2ркh	167
Построение произвольных сценариев	171
Упражнение 3	174
Польза сценариев	175
Упражнение 4	175
Пиньята для алгоритма SHA-1	175
Заключение	176
Глава 7. Создание и проверка достоверности транзакций	177
Проверка достоверности транзакций	177
Проверка расходования вводов транзакции	178
Проверка суммы вводов относительно суммы выводов транзакции	178
Проверка подписи	180
Упражнение 1	184
Упражнение 2	185
Верификация всей транзакции	185
Создание транзакции	185
Построение транзакции	186
Составление транзакции	189
Подписание транзакции	191
Упражнение 3	192
Создание собственных транзакций в сети testnet	192
Упражнение 4	193
Упражнение 5	193
Заключение	193

Глава 8. Оплата по хешу сценария	195
Простая мультиподпись	196
Программная реализация операции OP_CHECKMULTISIG	200
Упражнение 1	200
Недостатки простой мультиподписи	201
Оплата по хешу сценария	201
Программная реализация p2sh	209
Более сложные сценарии	210
Адреса	210
Упражнение 2	210
Упражнение 3	211
Верификация подписей в p2sh	211
Упражнение 4	214
Упражнение 5	214
Заключение	214
Глава 9. Блоки	215
Монетизирующие транзакции	216
Упражнение 1	217
Сценарий ScriptSig	217
Протокол VIP0034	218
Упражнение 2	219
Заголовки блоков	219
Упражнение 3	220
Упражнение 4	220
Упражнение 5	220
Версия	220
Упражнение 6	222
Упражнение 7	222
Упражнение 8	222
Предыдущий блок	223
Корень дерева Меркла	223
Отметка времени	223
Биты	224
Одноразовый номер	224
Подтверждение работы	224
Каким образом добытчик криптовалюты генерирует хеш-коды	226
Цель	226

Упражнение 9	228
Сложность	228
Упражнение 10	229
Проверка достаточности подтверждения работы	229
Упражнение 11	229
Корректировка сложности	229
Упражнение 12	231
Упражнение 13	232
Заключение	232
Глава 10. Организация сети	233
Сетевые сообщения	233
Упражнение 1	235
Упражнение 2	235
Упражнение 3	235
Синтаксический анализ полезной информации	235
Упражнение 4	237
Подтверждение подключения к сети	237
Подключение к сети	238
Упражнение 5	241
Получение заголовков блоков	241
Упражнение 6	243
Присылаемые в ответ заголовки	243
Заключение	246
Глава 11. Упрощенная проверка оплаты	247
Предпосылки	247
Дерево Меркла	248
Родительский узел дерева Меркла	249
Упражнение 1	250
Родительский уровень дерева Меркла	251
Упражнение 2	252
Корень дерева Меркла	252
Упражнение 3	253
Корень дерева Меркла в блоках	253
Упражнение 4	254
Применение дерева Меркла	254
Древовидный блок Меркла	256

Структура дерева Меркла	258
Упражнение 5	259
Программная реализация дерева Меркла	259
Команда merkleblock	265
Упражнение 6	267
Применение битов признаков и хешей	267
Упражнение 7	272
Заключение	272
Глава 12. Фильтры Блума	273
Что такое фильтр Блума	273
Упражнение 1	276
Продвижение на шаг дальше	276
Фильтры Блума по протоколу VIP0037	277
Упражнение 2	279
Упражнение 3	280
Загрузка фильтра Блума	280
Упражнение 4	280
Получение древовидных блоков Меркла	280
Упражнение 5	281
Получение представляющей интерес транзакции	282
Упражнение 6	283
Заключение	284
Глава 13. Протокол Segwit	285
Оплата по хешу открытого ключа с отдельным заверением	285
Податливость транзакции	286
Устранение податливости транзакций	287
Транзакции по сценарию p2wpkh	287
Сценарий p2sh-p2wpkh	291
Программная реализация сценариев p2wpkh и p2sh-p2wpkh	296
Оплата по хешу сценария с отдельным заверением (p2wsh)	301
Сценарий p2sh-p2wsh	306
Программная реализация сценариев p2wsh и p2sh-p2wsh	311
Прочие усовершенствования	313
Заключение	314

Глава 14. Дополнительные вопросы и следующие шаги	315
Темы для дальнейшего изучения	315
Криптовалютные кошельки	315
Платежные каналы и протокол Lightning Network	317
Участие в разработке биткойна	317
Другие предлагаемые проекты	318
Криптовалютный кошелек в сети testnet	318
Обозреватель блоков	318
Интернет-магазин	318
Служебная библиотека	319
Трудоустройство	319
Заключение	320
Приложение. Ответы к упражнениям	321
Глава 1. Конечные поля	321
Глава 2. Эллиптические кривые	325
Глава 3. Криптография по эллиптическим кривым	327
Глава 4. Сериализация	331
Глава 5. Транзакции	335
Глава 6. Язык Script	339
Глава 7. Создание и проверка достоверности транзакций	342
Глава 8. Оплата по хешу сценария	346
Глава 9. Блоки	349
Глава 10. Организация сети	353
Глава 11. Упрощенная проверка оплаты	356
Глава 12. Фильтры Блума	358
Предметный указатель	365

Фильтры Блума

В главе 11 пояснялось, как проверять древовидный блок Меркла на достоверность. Полный узел может предоставить подтверждение включения для представляющих интерес транзакций по сетевой команде `merkleblock`. Но откуда полному узлу известно, какие именно транзакции представляют интерес?

“Тонкий” клиент мог бы сообщить полному узлу свои адреса (или открытые ключи из сценария `ScriptPubKey`). Полный узел может проверить транзакции, соответствующие этим адресам, но в то же время это может нарушить конфиденциальность “тонкого” клиента. Ведь “тонкому” клиенту вряд ли захочется раскрывать полному узлу, что он, например, обладает суммой 1000 биткойнов. Утечки конфиденциальности означают утечку информации, и в биткойне, как правило, рекомендуется при всякой возможности избегать любых утечек конфиденциальности.

Но в одном случае “тонкий” клиент может сообщить полному узлу достаточно сведений, чтобы создать *надмножество* всех представляющих интерес транзакций. Чтобы создать такое надмножество, придется воспользоваться так называемым *фильтром Блума*.

Что такое фильтр Блума

Фильтр Блума служит фильтром для всех возможных транзакций. Полные узлы выполняют транзакции через фильтр Блума и отсылают команды `merkleblock` для выполнения пропускаемых через него транзакций.

Допустим, что всего имеется 50 транзакций, а “тонкого” клиента интересует лишь одна из них. В частности, “тонкому” клиенту требуется “скрыть” транзакцию из группы, состоящей из пяти транзакций. Для этого требуется функция, составляющая 50 транзакций в 10 разных групп, чтобы полный узел смог затем отправить, условно говоря, единую группу транзакций. Такое

группирование было бы *детерминированным*, т.е. оно должно быть постоянно одинаковым. Как же этого добиться? Решение в том, чтобы получить с помощью хеш-функции детерминированное число и остаток по модулю для организации транзакций в группы.

Фильтр Блума является структурой вычислительной техники, которую можно применить к любым данным из множества. Допустим, что имеется один элемент вроде строки “HelloWorld” (Здравствуй, мир), для которого требуется создать фильтр Блума. Для этого потребуется хеш-функция, и поэтому воспользуемся уже знакомой нам хеш-функцией hash256. Ниже показано, каким образом процесс выяснения, к какой именно группе относится данный элемент, реализуется непосредственно в коде.

```
>>> from helper import hash256
>>> bit_field_size = 10 ❶
>>> bit_field = [0] * bit_field_size
>>> h = hash256(b'hello world') ❷
>>> bit = int.from_bytes(h, 'big') % bit_field_size ❸
>>> bit_field[bit] = 1 ❹
>>> print(bit_field)
[0, 0, 0, 0, 0, 0, 0, 0, 0, 1]
```

- ❶ В переменной `bit_field` сохраняется список “групп” из 10 элементов.
- ❷ Здесь исходный элемент хешируется с помощью хеш-функции hash256.
- ❸ Здесь полученный результат интерпретируется как целочисленное значение, представленное байтами в обратном порядке их следования, а также остатка по модулю 10, чтобы определить группу, к которой относится данный элемент.
- ❹ Здесь обозначается группа, которая требуется в фильтре Блума.

Все, что было сделано в приведенном выше коде, схематически показано на рис. 12.1.

Таким образом, рассматриваемый здесь фильтр Блума состоит из следующих элементов.

1. Размер битового поля.
2. Применяемая хеш-функция, а также алгоритм преобразования в число.
3. Битовое поле, обозначающее представляющую интерес группу.

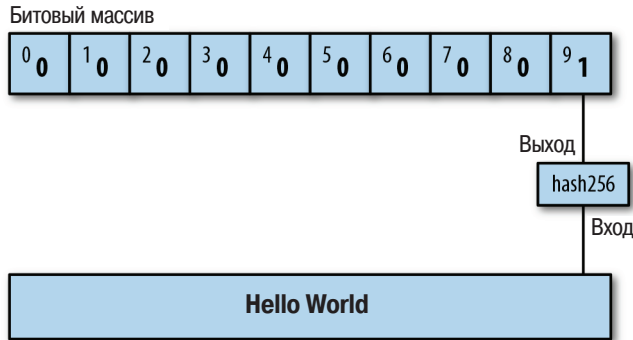


Рис. 12.1. 10-разрядный фильтр Блума с одним элементом

Такой фильтр пригоден для единственного элемента, поэтому он подходит и для представляющего интерес адреса, сценария ScriptPubKey или идентификатора транзакции. А что делать, если нас интересует не один элемент?

Выполнить второй элемент можно через тот же самый фильтр, а также установить 1 в соответствующем бите. И тогда полный узел может отправить несколько групп транзакций вместо одной группы. Итак, создадим фильтр Блума с двумя элементами (“Hello world” и “Goodbye”), воспользовавшись следующим кодом:

```
>>> from helper import hash256
>>> bit_field_size = 10
>>> bit_field = [0] * bit_field_size
>>> for item in (b'hello world', b'goodbye'):
...     h = hash256(item)
...     bit = int.from_bytes(h, 'big') % bit_field_size
...     bit_field[bit] = 1
>>> print(bit_field)
[0, 0, 1, 0, 0, 0, 0, 0, 0, 1]
```

- ❶ Здесь создается фильтр для двух элементов, хотя данную процедуру можно расширить и на большее количество элементов.

Порядок создания фильтра Блума с двумя элементами схематически показан на рис. 12.2.

Если количество всех возможных элементов равно 50, то через такой фильтр в среднем будет пропущено 10, а не 5 представляющих интерес элементов, как в том случае, если бы он был одноэлементным. Ведь в данном случае возвращаются две группы элементов вместо одной.

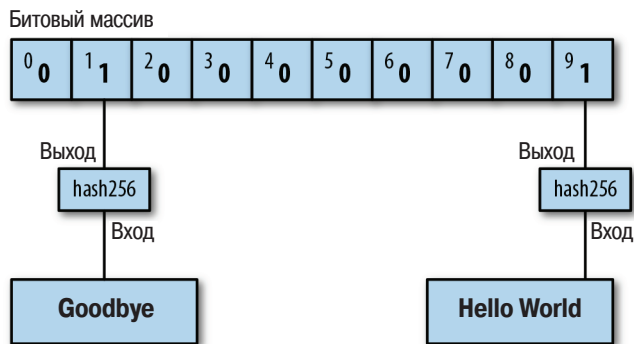


Рис. 12.2. 10-разрядный фильтр Блума с двумя элементами

Упражнение 1

Рассчитайте фильтр Блума для строк “Hello world” и “Goodbye”, применив хеш-функцию hash160 к битовому полю размером 10.

Продвижение на шаг дальше

Допустим, что пространство всех элементов составляет 1 миллион и требуется оставить размер группы равным 5. И для этого потребуется фильтр Блума длиной $1000000 / 5 = 200000$ битов. При этом каждая группа будет в среднем состоять из 5 элементов, и тогда мы получим в 5 раз больше элементов, чем нужно, т.е. лишь 20% от всего количества элементов будут представлять для нас интерес. Но передать данные объемом 200000 битов или 25000 байтов не так-то просто. Можно ли в таком случае поступить лучше?

Если воспользоваться несколькими хеш-функциями в фильтре Блума, то можно значительно сократить размер битового поля. Так, если применить 5 хеш-функций к битовому полю размером 32, то в итоге получится $32! / (27!5!) \sim 200000$ возможных комбинаций из 5 битов в этом 32-разрядном поле. А из 1 миллиона возможных элементов в среднем 5 должны иметь такую комбинацию из 5 разрядов. Таким образом, вместо 25 Кбайтов в данном случае можно передать лишь 32 бита или 4 байта!

Ниже показано, каким образом такой фильтр реализуется непосредственно в коде. Для простоты в данном примере используются то же самое 10-разрядное поле и два представляющих интерес элемента.

```
>>> from helper import hash256, hash160
>>> bit_field_size = 10
```

```
>>> bit_field = [0] * bit_field_size
>>> for item in (b'hello world', b'goodbye'):
...     for hash_function in (hash256, hash160):
...         h = hash_function(item)
...         bit = int.from_bytes(h, 'big') % bit_field_size
...         bit_field[bit] = 1
>>> print(bit_field)
[1, 1, 1, 0, 0, 0, 0, 0, 0, 1]
```

❶ Здесь циклически выполняются две разные хеш-функции (hash256 и hash160), но с тем же успехом можно сделать больше.

Порядок создания фильтра Блума с двумя элементами и хеш-функциями схематически показан на рис. 12.3.

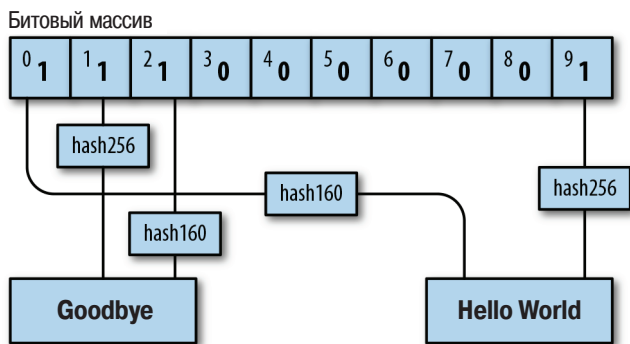


Рис. 12.3. 10-разрядный фильтр Блума с двумя элементами и хеш-функциями

Фильтр Блума можно оптимизировать, изменив количество хеш-функций и размер битового поля, чтобы получить требуемую долю ложноположительных результатов.

Фильтры Блума по протоколу VIP0037

В протоколе VIP0037 определяются фильтры Блума, применяемые при передаче данных по сети. Ниже перечислены сведения, хранящиеся в фильтре Блума.

1. Размер битового поля или количество имеющихся групп. Этот размер указывается в байтах (по 8 битов на каждый байт) и округляется по мере необходимости.
2. Количество хеш-функций.

3. “Настройка”, способная немного изменить фильтр Блума, если он встретит слишком много элементов.
4. Битовое поле, получаемое в результате выполнения фильтра Блума над представляющими интерес элементами.

Несмотря на возможность определить немало хеш-функций (sha512, кессак384, ripemd160, blake256 и т.д.), на практике применяется единственная хеш-функция, но с разными начальными случайными значениями. Благодаря этому упрощается реализация данного фильтра.

Мы будем пользоваться здесь хеш-функцией, называемой *murmur3*. В отличие от хеш-функции sha256, хеш-функция *murmur3* не является криптографически безопасной, но действует намного быстрее. И хотя для решения задачи фильтрации и получения детерминированного, равномерно распределенного значения по модулю криптографическая защита не требуется, оно только выиграет от повышения быстродействия, а потому хеш-функция *murmur3* больше подходит для решения этой задачи. Начальное случайное значение для хеш-функции *murmur3* определяется по следующей формуле:

```
i*0xfba4c795 + tweak
```

Здесь **fba4c795** — это константа для фильтров Блума, применяемых в биткойне, **i** равно **0** для первой хеш-функции, **1** — для второй хеш-функции, **2** — для третьей хеш-функции и так далее, а **tweak** — доля энтропии, которая может быть введена, если не удовлетворяют результаты одной настройки. Хеш-функции и размер битового поля служат для вычисления содержимого битового поля, которое затем передается, как показано ниже.

```
>>> from helper import murmur3 ①
>>> from bloomfilter import BIP37_CONSTANT ②
>>> field_size = 2
>>> num_functions = 2
>>> tweak = 42
>>> bit_field_size = field_size * 8
>>> bit_field = [0] * bit_field_size
>>> for phrase in (b'hello world', b'goodbye'): ③
...     for i in range(num_functions): ④
...         seed = i * BIP37_CONSTANT + tweak ⑤
...         h = murmur3(phrase, seed=seed) ⑥
...         bit = h % bit_field_size
...         bit_field[bit] = 1
>>> print(bit_field)
[0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0]
```

- ❶ Хеш-функция murmur3 реализуется в исходном файле `helper.py` только на языке Python.
- ❷ Константа `BIP37_CONSTANT` содержит число `fba4c795`, указанное в протоколе BIP0037.
- ❸ Здесь циклически перебираются представляющие интерес элементы.
- ❹ Здесь применяются две хеш-функции.
- ❺ Здесь реализуется формула для определения начального случайного значения.
- ❻ Хеш-функция `murmur3` возвращает случайное число, и поэтому его не нужно преобразовывать в целое число.

В четырех из шестнадцати битов (т.е. двух байтов) реализованного выше фильтра Блума установлена 1, и поэтому вероятность прохождения произвольного элемента через этот фильтр равна $1/4 \times 1/4 = 1/16$. Так, если количество всех элементов равно 160, клиент получит в среднем 10 элементов, 2 из которых будут представлять интерес.

А теперь можно приступить к определению класса `BloomFilter`, реализующего фильтр Блума следующим образом:

```
class BloomFilter:
    def __init__(self, size, function_count, tweak):
        self.size = size
        self.bit_field = [0] * (size * 8)
        self.function_count = function_count
        self.tweak = tweak
```

Упражнение 2

Если задан фильтр Блума с параметрами `size=10`, `function_count=5`, `tweak=99`, то какие байты устанавливаются после ввода приведенных ниже элементов? (*Подсказка*: воспользуйтесь функцией `bit_field_to_bytes()` из исходного файла `helper.py` для преобразования содержимого битового поля в байты.)

- `b'Hello World'`
- `b'Goodbye!'`

Упражнение 3

Напишите метод `add()` для класса `BloomFilter`, чтобы реализовать в нем ввод элементов в фильтр Блума.

Загрузка фильтра Блума

Как только “тонкий” клиент создаст фильтр Блума, ему придется уведомить полный узел об этом фильтре, чтобы полный узел смог отправлять подтверждения включения. Для этого “тонкий” клиент должен, прежде всего, установить 0 в дополнительном признаке пересылки из сообщения о версии (см. главу 10). Этим полному узлу указывается не отправлять сообщения о транзакциях, если только они не пройдут фильтр Блума или не будут запрошены специально. После установки признака пересылки “тонкий” клиент должен передать полному узлу сам фильтр Блума. Для этого и служит команда `filterload`, структура которой приведена на рис. 12.4.

```
0a4000600a080000010940050000006300000000
```

- 0a4000600a080000010940 - Битовое поле переменной длины
- 05000000 - Подсчет хешей, 4 байта в прямом порядке их следования
- 63000000 - Настройка, 4 байта в прямом порядке их следования
- 00 - Признак совпадающего элемента

Рис. 12.4. Результат синтаксического анализа команды `filterload`

Элементы фильтра Блума кодируются в байтах. Битовое поле, поля подсчета хеш-функций и настройки кодируются в сообщении о фильтре Блума. А последнее поле с признаком совпадающего элемента служит для запроса полного узла на ввод любых совпадающих транзакций в фильтр Блума.

Упражнение 4

Напишите метод `filterload()` для класса `BloomFilter`, чтобы реализовать в нем загрузку фильтра Блума.

Получение древовидных блоков Меркла

“Тонкому” клиенту потребуется еще одна команда, чтобы получить из полного узла сведения об интересующих его транзакциях, находящихся в древовидном блоке из дерева Меркла. Сведения о блоках и транзакциях передает команда `getdata`, а конкретный тип данных, которые потребуются “тонкому”

клиенту из полного узла, называется *фильтрованным блоком*. Такой блок является запросом транзакций, проходящим через фильтр Блума в форме древовидного блока Меркла. Иными словами, “тонкий” клиент может запросить древовидные блоки Меркла, в которых находятся транзакции, которые интересуют данного клиента и совпадают с критерием прохождения через фильтр Блума. Структура команды `getdata` приведена на рис. 12.5.

```
020300000030eb2540c41025690160a1014c577061596e32e426b712c7ca000  
0000000000000300000001049847939585b0652fba793661c361223446b6fc410  
89b8be00000000000000
```

- 02 - Количество элементов данных
- 03000000 - Тип элемента данных (транзакция, блок, фильтрованный блок, компактный блок), в прямом порядке следования байтов
- 30...00 - Идентификатор хеша

Рис. 12.5. Результат синтаксического анализа команды `getdata`

Количество элементов данных типа `varint` обозначает, сколько требуется таких элементов. Каждый элемент данных относится к конкретному типу. В частности, значение типа 1 относится к транзакции (см. главу 5), типа 2 — к обычному блоку (см. главу 9), типа 3 — к древовидному блоку Меркла (см. главу 11), а типа 4 — к компактному блоку (этот тип в данной книге не рассматривается).

В исходном файле `network.py` можно создать следующее сообщение:

```
class GetDataMessage:  
    command = b'getdata'  
  
    def __init__(self):  
        self.data = [] ❶  
  
    def add_data(self, data_type, identifier):  
        self.data.append((data_type, identifier)) ❷
```

- ❶ Сохранить требующиеся элементы данных.
- ❷ Ввести элементы данных в сообщение, используя метод `add_data()`.

Упражнение 5

Напишите метод `serialize()` для класса `GetDataMessage`, чтобы реализовать в нем сериализацию получаемых элементов данных.

Получение представляющей интерес транзакции

“Тонкий” клиент, загружающий фильтр Блума из полного узла, получит в свое распоряжение все необходимые сведения, позволяющие ему убедиться, что интересующие его транзакции действительно входят в состав конкретных блоков, как показано ниже.

```
>>> from bloomfilter import BloomFilter
>>> from helper import decode_base58
>>> from merkleblock import MerkleBlock
>>> from network import FILTERED_BLOCK_DATA_TYPE, GetHeadersMessage, \
GetDataMessage, HeadersMessage, SimpleNode
>>> from tx import Tx
>>> last_block_hex = '0000000000538d5c2246336644f9a4956551afb44ba\
47278759ec55ea912e19'
>>> address = 'mwJn1YPMq7y5F8J3LkC5Hxg9PHyZ5K4cFv'
>>> h160 = decode_base58(address)
>>> node = SimpleNode('testnet.programmingbitcoin.com', testnet=True, \
logging= False)
>>> bf = BloomFilter(size=30, function_count=5, tweak=90210) ❶
>>> bf.add(h160) ❷
>>> node.handshake()
>>> node.send(bf.filterload()) ❸
>>> start_block = bytes.fromhex(last_block_hex)
>>> getheaders = GetHeadersMessage(start_block=start_block)
>>> node.send(getheaders) ❹
>>> headers = node.wait_for(HeadersMessage)
>>> getdata = GetDataMessage() ❺
>>> for b in headers.blocks:
...     if not b.check_pow():
...         raise RuntimeError('proof of work is invalid')
...         getdata.add_data(FILTERED_BLOCK_DATA_TYPE, b.hash()) ❻
>>> node.send(getdata) ❼
>>> found = False
>>> while not found:
...     message = node.wait_for(MerkleBlock, Tx) ❽
...     if message.command == b'merkleblock':
...         if not message.is_valid(): ❾
...             raise RuntimeError('invalid merkle proof')
...         else:
...             for i, tx_out in enumerate(message.tx_outs):
...                 if tx_out.script_pubkey.address(testnet=True) \
...                     == address: ❿
...                     print('found: {}:{}'.format(message.id(), i))
...                     found = True
...                     break
```

found: e3930e1e566ca9b75d53b0eb9acb7607f547e1182d1d22bd4b661\
cfe18dcddf1:0

- 1 Создать фильтр Блума размером 30 байтов, использующий хеш-функции и настройку, особенно распространенную в 1990-е годы.
- 2 Отфильтровать по приведенному выше адресу.
- 3 Отправить команду `filterload` из созданного фильтра Блума.
- 4 Получить заголовки блоков после шестнадцатеричного идентификатора последнего блока (`last_block_hex`).
- 5 Создать сообщение для получения данных для древовидных блоков Меркла, в которых могут присутствовать представляющие интерес транзакции.
- 6 Запросить древовидный блок Меркла, чтобы убедиться в наличии в нем представляющих интерес транзакций. В большинстве блоков они, вероятнее всего, будут отсутствовать.
- 7 Запросить в сообщении для получения данных 2000 древовидных блоков Меркла после определения блока по его шестнадцатеричному идентификатору (`last_block_hex`).
- 8 Ожидать команды `merkleblock`, подтверждающей включение, а также команды `tx`, предоставляющей искомую транзакцию.
- 9 Проверить, подтверждает ли древовидный блок Меркла включение транзакции.
- 10 Найти неизрасходованные выводы транзакций UTXO по конкретному адресу (`address`) и вывести найденное на экран.

В данном случае проанализированы 2000 блоков после конкретного блока, чтобы обнаружить неизрасходованные выводы транзакций UTXO, соответствующие конкретному адресу. А поскольку все сделано безо всякой помощи обозревателя блоков, в какой-то степени это сохранило нашу конфиденциальность.

Упражнение 6

Получите идентификатор текущего блока в сети testnet, отправьте себе немного монет по сети testnet, найдите неизрасходованный вывод транзакции UTXO, соответствующий этим монетам, *не* пользуясь обозревателем блоков,

создайте транзакцию, используя данный UTXO в качестве ввода, а также перешлите сообщение `tx` по сети testnet.

Заключение

В этой главе было показано, как создать все, что требуется для подключения “тонкого” клиента по одноранговой сети, запроса и получения неизрасходованных выводов транзакции UTXO, необходимых для построения транзакции. И все это можно сделать, сохранив конфиденциальность с помощью фильтра Блума. А теперь перейдем к протоколу Segwit, определяющему новый тип транзакции, внедренный в 2017 году.