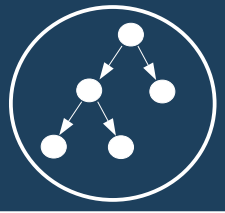


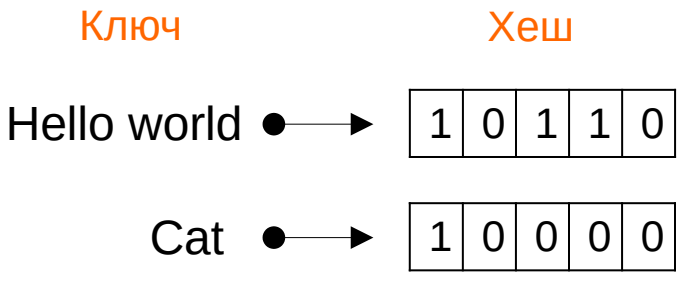
Data Structures and Algorithms

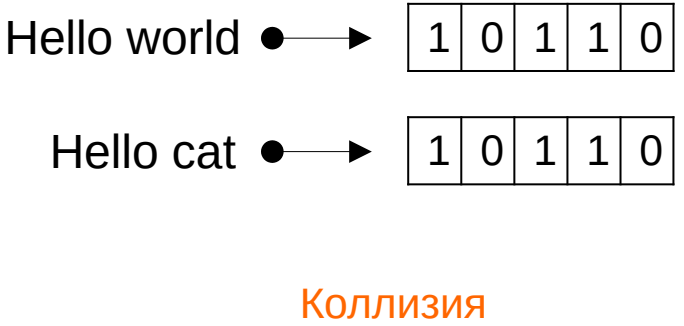
Хеш-функция

Хеш-функция

Хеш-функция (функция свертки) — функция, генерирующая на основе входных данных произвольной длины, битовую строку фиксированной длины. Процесс генерации называют хешированием. Входные данные иногда называют «ключом», «сообщением». Результирующая битовая строка называется «хешем», «хеш-кодом», «хеш-суммой», «сводкой сообщения».

В общем случае множество входящих данных больше множества хешей, это приводит к тому, что для различных входных данных генерируется один и тот же хеш-код. Такая ситуация называется коллизией.





Требования выдвигаемые к хеш-функции

В большинстве случаев к хеш-функции выдвигается несколько обязательных требований:

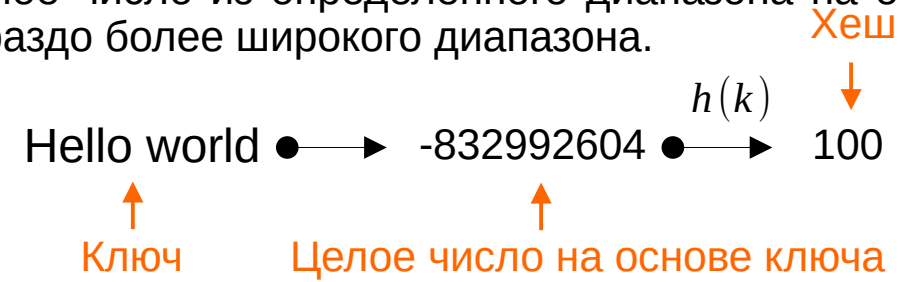
1) Детерминированность используемого алгоритма. Следствием из этого является идентичность хешей при идентичности входных данных.
2) Функция должна работать только с входными данными и не изменять их.

«Хорошая» хеш-функция должна обладать следующими желательными (но не обязательными) свойствами:

- Быстрое вычисление. Хеш-функция должна быстро генерировать код на основе данных.
- Большая вычислительная сложность обратного вычисления. Должно быть сложно восстановить вид входящих данных по их хеш-коду (криптографическая стойкость)
- Малое количество коллизий
- Равномерное распределение входных данных на множество хеш-кодов

Часто используемый подход к хешированию

Часто используемым подходом к хешированию является представление ключа как целочисленного значения. Если это проблематично, то ищут способ генерация на основе ключа целочисленного значения. В качестве битовой строки результата используется двоичное представление целого числа из определенного диапазона. Таким образом хеш-функция генерирует целое число из определенного диапазона на основе целого числа (созданного на основе ключа) из гораздо более широкого диапазона.



Таким образом хеширование сводиться к двум задачам. Задача 1 — получение на основе ключа целочисленного значения. Задача 2 — получение с помощью хеш-функции целого числа из меньшего диапазона.

Пример генерации целого числа на основе ключа

Предположим что ключами являются строки содержащие символы английского языка, цифры и базовые типографические символы. Как создать число на ее основе? Одним из способов является представление строки как числа в системе счисления основание которой равно количеству символов. Возьмем к примеру первые 256 символов ASCII кодировки. В таком случае символ это цифра (код символа), а основание этой системе счисления равно количеству символов 256.

H e l l o → $111 \cdot 256^0 + 108 \cdot 256^1 + 108 \cdot 256^2 + 101 \cdot 256^3 + 72 \cdot 256^4 = 310939249775$

4 3 2 1 0

Хеш-функция на основе деления

Одной из наиболее простых хеш-функций является хеш-функция на основе деления. Вычисление хеша сводится к вычислению остатка от деления.

$$h(K) = K \bmod M$$

K - числовое представление ключа

M - размер диапазона хешей

Для выбора *M* — существует ряд рекомендаций. Желательно, что бы M было нечетным (Если M четное, то четность хеш-кода совпадает с четностью числового значения ключа). В общем случае не рекомендуется использовать M кратным значению:

$$r^k \pm a$$

r — основание системы счисления при генерации числового представления ключа
k,a — небольшие числа

Рекомендуемым для выбора M является простое число.

Один из алгоритмов выбора M

Проводят хотя бы приблизительную оценку количества ключей и уровень коллизий между ними (сколько ключей могут давать один и тот же хеш). M — ближайшее простое число к их отношению. В ряде случаев можно использовать табулированное значение (заранее определенное) для различных диапазонов ключей.

Пример — предположим у нас есть 3500 ключей, уровень коллизии 5 ключей на один хеш. Тогда в качестве M выбираем значение 691 (близкое к значению 3500/5 = 700).

Python

Реализация алгоритма на Python

Python

Функция для генерации числа на основе строки

```
def text_to_number(text):
    radix = 256
    result = 0
    n = len(text)
    for i in range(n):
        result += ord(text[i]) * radix ** (n - 1 - i)
    return result
```

Python

Класс реализующий хеширование на основе деления

```
class HashService:

    def __init__(self, key_range, collision_level):
        self.prime_number = {10:7,20:19,50:47,100:97,250:241,500:499,1000:997,2500:2477,5000:4999,10000:9973}
        self.m = key_range // collision_level
        for k in self.prime_number.keys():
            if self.m < k:
                self.m = k
                break

    def generate_hash(self, text):
        return text_to_number(text) % self.m
```

Хеш-функция на основе умножения

Хеш-функция на основе умножения строиться следующим образом — умножаем числовое представление ключа на константу $0 < A < 1$ и выделяется дробная часть этого произведения. Полученное значение умножается на M. После этого выделяется целая часть результата.

$$h(K) = \lfloor M \cdot \{k \cdot A\} \rfloor$$

Для константы A — есть рекомендуемое значение (Дональд Кнут)

$$A = \frac{\sqrt{5}-1}{2} \approx 0.6180339887$$

В случае использования этой хеш-функции рекомендуют в качестве M выбирать одно из значений степеней двойки. Эта рекомендация основывается на двоичном представлении машинного слова и приводит к ускорению вычислений.

Представление ключа в виде полинома

Для получения на основе ключа числового представления, можно использовать части ключа как коэффициенты полинома, в качестве значения независимой переменной можно использовать небольшое простое число. Одним из популярных вариантов является выбор значения 31. Это небольшое простое число довольно оптимально для вычисления, т.к. умножение можно свести к битовому сдвигу.

$$31 \cdot n = (n \ll 5) - n$$

$$[3, 6, 1] = 3 * 31^{2} + 6 * 31^{1} + 1 * 31^{0} = 3070$$
$$\quad\ 2 \quad 1 \quad 0$$

Для вычисления полиномов такого вида отлично подходит схема Горнера.

Реализация на Java

Метод для получения целого числа на основе массива

```
public static int arrayToNumber(int[] array) {
    int result = array[0];
    for (int i = 0; i < array.length - 1; i++) {
        result = (result << 5) - result + array[i + 1];
    }
    return result;
}
```

Метод для вычисления хеш-кода используя деление

```
public static int generateHash(int[] array, int m) {
    int number = Math.abs(arrayToNumber(array));
    double a = 0.6180339887;
    return (int) (m * ((number * a) % 1));

}
```

Хеш-функция на основе таблицы переходов

Одной из простых хеш-функций является хеш-функция на основе таблицы переходов (хеширование Пирсона). Эта хеш-функция очень быстро вычисляется и проста в реализации. К недостаткам относится низкая криптографическая стойкость.

Принцип построения следующий - строиться таблица (массив) чисел в диапазоне от 0 до размера этой таблицы минус 1 расположенных случайным образом. Очень часто размер этой таблицы составляет 256 элементов (именно такой размер предложен автором). Это означает что результатом будет байтовая строка размером 1 байт (число от 0 до 255). Происходит инициализация начального значения хеша, после чего для каждого элемента ключа хеш пересчитывается. Сначала вычисляется новый индекс.

- Если рассматривать ключ как последовательность байт то используют XOR текущего значения хеша и очередного байта
- Если рассматривать ключ как последовательность целых чисел, то остаток от деления суммы текущего значения хеша и очередного числа ключа, на размер таблицы

Новым значением хеша становится элемент стоящий на полученном индексе, после чего операция циклически повторяется.

Пример хеш-функции на основе таблицы переходов

h = 0

C a t

2 1 0

(h + ord(t)) % 10 = (0 + 116) % 10 = 6

7	3	0	9	6	2	4	8	1	5
0	1	2	3	4	5	6	7	8	9

h = 4

Пример хеш-функции на основе таблицы переходов

h = 4

C a t

2 1 0

(h + ord(a)) % 10 = (4 + 97) % 10 = 1

7	3	0	9	6	2	4	8	1	5
0	1	2	3	4	5	6	7	8	9

h = 3

Пример хеш-функции на основе таблицы переходов

h = 3

C a t

2 1 0

(h + ord(C)) % 10 = (3 + 67) % 10 = 0

7	3	0	9	6	2	4	8	1	5
0	1	2	3	4	5	6	7	8	9

h = 7

Хеш-функции на основе таблицы переходов для многобайтового ключа

Как создать хеш из нескольких байт используя небольшую таблицу переходов? Можно получить однобайтовых хеш на основе ключа. После чего изменить предсказуемым образом копию ключа и получить еще один однобайтовый хеш. После чего операцию повторить столько раз сколько байт нужно. Объединить байтовые хеши в один многобайтовый.

Fortran

Реализация на Fortran

Класс для реализации хеширования Пирсона

```
type Pirson_hash
    integer(4)::transition_table(256)

    contains

    procedure::generate_hash
    procedure::init
    procedure::generate_four_byte_hash

end type Pirson_hash
```

Процедура инициализации и таблица переходов

```
subroutine init(this)
    class(Pirson_hash), intent(inout)::this
    this%transition_table = [98,  6, 85,150, 36, 23,112,164,135,207,169,  5, 26, 64,165,219,&
    61, 20, 68, 89,130, 63, 52,102, 24,229,132,245, 80,216,195,115,&
    90,168,156,203,177,120,  2,190,188,  7,100,185,174,243,162, 10,&
    237, 18,253,225,  8,208,172,244,255,126,101, 79,145,235,228,121,&
    123,251, 67,250,161,  0,107, 97,241,111,181, 82,249, 33, 69, 55,&
    59,153, 29,  9,213,167, 84, 93, 30, 46, 94, 75,151,114, 73,222,&
    197, 96,210, 45, 16,227,248,202, 51,152,252,125, 81,206,215,186,&
    39,158,178,187,131,136,  1, 49, 50, 17,141, 91, 47,129, 60, 99,&
    154, 35, 86,171,105, 34, 38,200,147, 58, 77,118,173,246, 76,254,&
    133,232,196,144,198,124, 53,  4,108, 74,223,234,134,230,157,139,&
    189,205,199,128,176, 19,211,236,127,192,231, 70,233, 88,146, 44,&
    183,201, 22, 83, 13,214,116,109,159, 32, 95,226,140,220, 57, 12,&
    221, 31,209,182,143, 92,149,184,148, 62,113, 65, 37, 27,106,166,&
    3, 14,204, 72, 21, 41, 56, 66, 28,193, 40,217, 25, 54,179,117,&
    238, 87,240,155,180,170,242,212,191,163, 78,218,137,194,175,110,&
    43,119,224, 71,122,142, 42,160,104, 48,247,103, 15, 11,138, 239]
end subroutine init
```

Функция вычисления однобайтового хеша

```
function generate_hash (this, text)
    character(len = *), intent(in)::text
    class(Pirson_hash), intent(in)::this
    integer(4)::generate_hash
    integer::i, new_index
    generate_hash = 0
    do i = 1, len(trim(text))
        new_index = MOD(generate_hash + iachar(text(i:i)), size(this%transition_table)) + 1
        generate_hash = this%transition_table(new_index)
    end do

end function generate_hash
```

Функция вычисления четырехбайтового хеша

```
function generate_four_byte_hash (this, text)
    character(len = *), intent(in)::text
    class(Pirson_hash), intent(in)::this
    integer::generate_four_byte_hash, hash
    integer::j, i
    character(len = :), allocatable :: temp_text
    generate_four_byte_hash = 0
    temp_text = trim(text)
    j = 1
    do i = 1, 3
        hash = this%generate_hash (temp_text)
        generate_four_byte_hash = IOR(generate_four_byte_hash, hash)
        generate_four_byte_hash = ISHFT(generate_four_byte_hash, 8)
        temp_text(j:j) = achar(iachar(temp_text(j:j)) + 1)
        j = j + 1
        if(j > len(temp_text)) then
            j = 1
        end if
    end do
    hash = this%generate_hash (temp_text)
    generate_four_byte_hash = IOR(generate_four_byte_hash, hash)
    deallocate(temp_text)
end function generate_four_byte_hash
```

Список литературы

1) Томас Кормен, Чарльз Лейзерсон, Рональд Ривест, Клиффорд Штайн // Алгоритмы: построение и анализ 3-е издание. — М.: «Вильямс», 2013. — С. 1328. ISBN 978-5-8459-1794-2
2) Дональд Кнут. «Искусство программирования, том 3. Сортировка и поиск» : Пер. с англ. - М. : ООО "И.Д. Вильямс", 2001. - 160 с. ISBN 978-5-8459-1164-3.