JavaScript - спец-объекты

Set, Map, WeekSet, WeekMap, поиск по ключу и кэширование

Поиск нужного элемента

Доступ к свойству объекта по ключу

Доступ к свойству объекта по ключу (как с использованием синтаксиса точки, так и квадратных скобок) обычно имеет временную сложность O(1). Это означает, что операция выполняется за постоянное время, независимо от размера объекта.

Обычный поиск (например, метод find в массиве)

Когда вы ищете элемент в массиве с использованием методов, таких как find, временная сложность этой операции в худшем случае составляет O(n), где n — количество элементов в массиве. Это означает, что время выполнения пропорционально размеру массива, так как в худшем случае вам может потребоваться пройтись по всем элементам массива, чтобы найти нужный элемент или убедиться, что его нет.

Маппинг на основе объекта (хэш-таблица)

Хэш-таблица (или хеш-таблица, хэш-словарь) — это структура данных, которая обеспечивает очень быструю вставку, удаление и поиск элементов. Основная идея заключается в использовании хэш-функции для преобразования ключа в индекс массива, в котором хранится значение.

1. **Хэш-функция:**
   Функция, которая принимает ключ и возвращает индекс массива.
   Хорошая хэш-функция минимизирует вероятность коллизий (ситуаций, когда разные ключи дают одинаковый индекс).
2. **Коллизии:**
   Случаи, когда два разных ключа получают одинаковый индекс. Существует несколько методов разрешения коллизий:
   a. Метод цепочек (chaining): Каждый элемент массива является списком, и все ключи, хэш которых дали одинаковый индекс, добавляются в этот список.
   b. Метод открытой адресации: При коллизии выполняется поиск другой свободной ячейки массива по определенному алгоритму (например, линейное пробирование, квадратичное пробирование, двойное хэширование).
3. **Время доступа:**
   В среднем время вставки, удаления и поиска элемента в хэш-таблице составляет $O(1)$ — константное время. Однако в худшем случае (при неэффективной хэш-функции или плохом разрешении коллизий) время доступа может быть $O(n)$

Хэш функция

Хэш-функция (или хэш-алгоритм) — это функция, которая принимает входные данные переменной длины и возвращает фиксированный набор битов (хэш-значение, хэш-код или просто хэш). Основная цель хэш-функции заключается в том, чтобы эффективно отображать данные произвольной длины в данные фиксированной длины.

Основные характеристики хэш-функций:

- Фиксированная длина вывода: Хэш-функция всегда возвращает хэш-значение фиксированной длины, независимо от размера входных данных.
- Равномерное распределение: Хорошая хэш-функция должна обеспечивать равномерное распределение хэш-значений по всем возможным входным данным.
- Быстродействие: Хэш-функция должна быть вычислительно эффективной, чтобы обеспечивать быструю генерацию хэш-значений для больших объемов данных.
- Сопротивление коллизиям: Хэш-функция должна минимизировать вероятность возникновения коллизий, когда двум разным входным данным соответствует одно и то же хэш-значение. Однако при этом необходимо учитывать, что полное избежание коллизий невозможно, особенно для хэш-функций с фиксированным размером вывода, работающих с данными переменной длины.

```
// Пример использования
const ht = new HashTable();

ht.set("ключ1", "значение1");
ht.set("ключ2", "значение2");

console.log(ht.get("ключ1")); // вывод: значение1
console.log(ht.get("ключ2")); // вывод: значение2

ht.remove("ключ1");
console.log(ht.get("ключ1")); // вывод: undefined
```

```
class HashTable {
    constructor(size = 50) {
        this.table = new Array(size);
        this.size = size;
    }

    // Хэш-функция
    _hash(key) {
        let hash = 0;
        for (let i = 0; i < key.length; i++) {
            hash += key.charCodeAt(i);
        }
        return hash % this.size;
    }

    // Вставка элемента
    set(key, value) {
        const index = this._hash(key);
        if (!this.table[index]) {
            this.table[index] = [];
        }
        this.table[index].push([key, value]);
    }

    // Поиск элемента
    get(key) {
        const index = this._hash(key);
        const bucket = this.table[index];
        if (bucket) {
            for (let i = 0; i < bucket.length; i++) {
                if (bucket[i][0] === key) {
                    return bucket[i][1];
                }
            }
        }
        return undefined;
    }

    // Удаление элемента
    remove(key) {
        const index = this._hash(key);
        const bucket = this.table[index];
        if (bucket) {
            for (let i = 0; i < bucket.length; i++) {
                if (bucket[i][0] === key) {
                    bucket.splice(i, 1);
                    return true;
                }
            }
        }
        return false;
    }
}
```

Пример реального использования хэш-таблицы

Реализация поиска на основе этапа и статуса:

```
const myArray = [
  {stage: 'LEAD', status: 'CREATED', value: 'test'},
  {stage: 'LEAD', status: 'APPROVED', value: 'test2'},
  {stage: 'WAREHOUSE', status: 'CREATED', value: 'test3'},
  {stage: 'WAREHOUSE', status: 'APPROVED', value: 'test4'},
  {stage: 'SALE', status: 'CREATED', value: 'test5'},
  {stage: 'SALE', status: 'SOLED', value: 'test6'},
];

function getValueByStageAndStatus (stage, status) {
  const reqObj = myArray.find(({stage: elStage, status: elStatus})=>{
    return elStage === stage && elStatus === status
  })
  return reqObj.value || 'not-found'
}

const res = getValueByStageAndStatus('LEAD','CREATED');

console.log(res)
```

```
function arrayToObject(arr) {
  const obj = {};
  arr.forEach(item => {
    if (!obj[item.stage]) {
      obj[item.stage] = {};
    }
    obj[item.stage][item.status] = item.value;
  });
  return obj;
}

const myObjectMap = arrayToObject(myArray);

function getValueByStageAndStatusFromMap(stage, status) {
  return myObjectMap?.[stage]?.[status] || 'not-found'
}

const resFromMap = getValueByStageAndStatusFromMap('SALE', 'CREATED')

console.log(resFromMap)
```

Map и Set

Map и Set предоставляют эффективные способы хранения и управления коллекциями данных. Map полезен для хранения пар ключ-значение, где ключи могут быть любого типа. Set полезен для хранения уникальных значений и предотвращения дублирования. Эти структуры данных обеспечивают более гибкие и производительные способы работы с данными по сравнению с традиционными объектами и массивами.

- MAP - коллекция ключ значение (похож на объект)
  - Хранение данных в формате ключ-значение с произвольным типом ключа.
  - Быстрый доступ, добавление и удаление пар ключ-значение.
- SET - коллекция значений (похож на массив)
  - Хранение уникальных значений.
  - Проверка наличия элемента в коллекции.
  - Быстрое добавление и удаление элементов.

Map

Map – это коллекция ключ/значение, как и Object. Но основное отличие в том, что Map позволяет использовать ключи любого типа.

Методы и свойства:

- new Map() – создает коллекцию.
- map.set(key, value) – записывает по ключу key значение value.
- map.get(key) – возвращает значение по ключу или undefined, если ключ key отсутствует.
- map.has(key) – возвращает true, если ключ key присутствует в коллекции, иначе false.
- map.delete(key) – удаляет элемент (пару «ключ/значение») по ключу key.
- map.clear() – очищает коллекцию от всех элементов.
- map.size – возвращает текущее количество элементов.

```
let map = new Map();

map.set("1", "str1");    // строка в качестве ключа
map.set(1, "num1");      // цифра как ключ
map.set(true, "bool1");  // булево значение как ключ

// помните, обычный объект Object приводит ключи к строкам?
// Map сохраняет тип ключей, так что в этом случае сохранится 2 разных значения:
alert(map.get(1)); // "num1"
alert(map.get("1")); // "str1"

alert(map.size); // 3
```

Map может использовать объекты в качестве ключей.

Использование объектов в качестве ключей – одна из наиболее заметных и важных функций Map. Это то что невозможно для Object. Строка в качестве ключа в Object – это нормально, но мы не можем использовать другой Object в качестве ключа в Object.

```
let john = { name: "John" };

// давайте сохраним количество посещений для каждого пользователя
let visitsCountMap = new Map();

// объект john – это ключ для значения в объекте Map
visitsCountMap.set(john, 123);

alert(visitsCountMap.get(john)); // 123
```

```
let john = { name: "John" };
let ben = { name: "Ben" };

// попробуем использовать объект
let visitsCountObj = {};

// пробуем использовать объект ben в качестве ключа
visitsCountObj[ben] = 234;
// пробуем использовать объект john в качестве ключа,
// при этом объект ben будет замещён
visitsCountObj[john] = 123;

// Вот что там было записано!
alert( visitsCountObj["[object Object]"] ); // 123
```

Перебор Map

Для перебора коллекции Map есть 3 метода:

- map.keys() – возвращает итерируемый объект по ключам,
- map.values() – возвращает итерируемый объект по значениям,
- map.entries() – возвращает итерируемый объект по парам вида [ключ, значение], этот вариант используется по умолчанию в for..of.

Кроме этого, Map имеет встроенный метод forEach, схожий со встроенным методом массивов Array.

Используется порядок вставки

В отличие от обычных объектов Object, в Map перебор происходит в том же порядке, в каком происходило добавление элементов.

```
let recipeMap = new Map([
  ["огурец", 500],
  ["помидор", 350],
  ["лук",    50]
]);

// перебор по ключам (овощи)
for (let vegetable of recipeMap.keys()) {
  alert(vegetable); // огурец, помидор, лук
}

// перебор по значениям (числа)
for (let amount of recipeMap.values()) {
  alert(amount); // 500, 350, 50
}

// перебор по элементам в формате [ключ, значение]
for (let entry of recipeMap) { // то же самое, что и recipeMap.entries()
  alert(entry); // огурец,500 (и так далее)
}
```

```
// выполняем функцию для каждой пары (ключ, значение)
recipeMap.forEach((value, key, map) => {
  alert(`${key}: ${value}`); // огурец: 500 и так далее
});
```

map[key]

map[key] это не совсем правильный способ использования Map

Хотя map[key] также работает, например, мы можем установить map[key] = 2, в этом случае map рассматривался бы как обычный JavaScript объект, таким образом это ведёт ко всем соответствующим ограничениям (только строки/символьные ключи и так далее).

Поэтому нам следует использовать методы map: set, get и так далее.

Как объект Map сравнивает ключи

Чтобы сравнивать ключи, объект Map использует алгоритм SameValueZero. Это почти такое же сравнение, что и ===, с той лишь разницей, что NaN считается равным NaN. Так что NaN также может использоваться в качестве ключа.

Этот алгоритм не может быть заменён или модифицирован.

Цепочка вызовов

Каждый вызов map.set возвращает объект map, так что мы можем объединить вызовы в цепочку:

```
map.set("1", "str1")
   .set(1, "num1")
   .set(true, "bool1");
```

Object.entries: Map из Object

При создании Map мы можем указать массив (или другой итерируемый объект) с парами ключ-значение для инициализации.

Если у нас уже есть обычный объект, и мы хотели бы создать Map из него, то поможет встроенный метод Object.entries(obj), который получает объект и возвращает массив пар ключ-значение для него, как раз в этом формате.

```
// массив пар [ключ, значение]
let map = new Map([
  ['1',  'str1'],
  [1,    'num1'],
  [true, 'bool1']
]);

alert( map.get('1') ); // str1
```

```
let obj = {
  name: "John",
  age: 30
};

let map = new Map(Object.entries(obj));

alert( map.get('name') ); // John
```

Object.fromEntries: Object из Map

Есть метод Object.fromEntries, который делает противоположное: получив массив пар вида [ключ, значение], он создает из них объект.

Мы можем использовать Object.fromEntries, чтобы получить обычный объект из Map.

Вызов map.entries() возвращает итерируемый объект пар ключ/значение, как раз в нужном формате для Object.fromEntries.

```
// убрать .entries()
let obj = Object.fromEntries(map);
```

```
let prices = Object.fromEntries([
  ['banana', 1],
  ['orange', 2],
  ['meat', 4]
]);

// prices = { banana: 1, orange: 2, meat: 4 }

alert(prices.orange); // 2
```

```
let map = new Map();
map.set('banana', 1);
map.set('orange', 2);
map.set('meat', 4);

// создаём обычный объект (*)
let obj = Object.fromEntries(map.entries());

// готово!
// obj = { banana: 1, orange: 2, meat: 4 }

alert(obj.orange); // 2
```

Set

Объект Set – это особый вид коллекции: «множество» значений (без ключей), где каждое значение может появляться только один раз.

Его основные методы это:

- new Set(iterable) – создает Set, и если в качестве аргумента был предоставлен итерируемый объект (обычно это массив), то копирует его значения в новый Set.
- set.add(value) – добавляет значение (если оно уже есть, то ничего не делает), возвращает тот же объект set.
- set.delete(value) – удаляет значение, возвращает true, если value было в множестве на момент вызова, иначе false.
- set.has(value) – возвращает true, если значение присутствует в множестве, иначе false.
- set.clear() – удаляет все имеющиеся значения.
- set.size – возвращает количество элементов в множестве.

«изюминка» SET

Основная «изюминка» – это то, что при повторных вызовах set.add() с одним и тем же значением ничего не происходит, за счёт этого как раз и получается, что каждое значение появляется один раз.

```
let set = new Set();

let john = { name: "John" };
let pete = { name: "Pete" };
let mary = { name: "Mary" };

// считаем гостей, некоторые приходят несколько раз
set.add(john);
set.add(pete);
set.add(mary);
set.add(john);
set.add(mary);

// set хранит только 3 уникальных значения
alert(set.size); // 3

for (let user of set) {
  alert(user.name); // John (потом Pete и Mary)
}
```

Перебор объекта Set

Set имеет те же встроенные методы, что и Map:

- set.values() – возвращает перебираемый объект для значений,
- set.keys() – то же самое, что и set.values(), присутствует для обратной совместимости с Map,
- set.entries() – возвращает перебираемый объект для пар вида [значение, значение], присутствует для обратной совместимости с Map.

Мы можем перебрать содержимое объекта set как с помощью метода for..of, так и используя forEach.

forEach у SET

Заметим забавную вещь. Функция в forEach у Set имеет 3 аргумента: значение value, потом снова то же самое значение valueAgain, и только потом целевой объект. Это действительно так, значение появляется в списке аргументов дважды.

Это сделано для совместимости с объектом Map, в котором колбэк forEach имеет 3 аргумента. Выглядит немного странно, но в некоторых случаях может помочь легко заменить Map на Set и наоборот.

```
let set = new Set(["апельсин", "яблоко", "банан"]);

for (let value of set) alert(value);

// то же самое с forEach:
set.forEach((value, valueAgain, set) => {
  alert(value);
});
```

WeekMap и WeekSet

WeakMap и WeakSet - это особые коллекции в JavaScript, которые работают с "слабыми" ссылками на объекты. Они имеют некоторые ограничения, но также и преимущества, особенно в контексте управления памятью и сборки мусора.

WeakMap и WeakSet полезны в ситуациях, когда нужно хранить данные временно и обеспечить автоматическое освобождение памяти, когда объекты больше не нужны. Они часто используются для хранения метаданных, связанных с объектами, например, кэширование вычислений, привязка обработчиков событий и прочие вспомогательные задачи, где важна сборка мусора для предотвращения утечек памяти.

WeekMap

WeakMap - это коллекция пар ключ-значение, где ключи обязательно должны быть объектами, а ссылки на эти объекты являются "слабыми". Это означает, что если объект, используемый в качестве ключа, больше не доступен и больше нигде не используется, он может быть автоматически удален сборщиком мусора, что предотвращает утечки памяти.

Задачи:

- Хранение данных с ключами-объектами, которые могут быть автоматически удалены сборщиком мусора, когда они становятся недоступными.
- Управление временными данными, связанными с объектами.

Методы:

- set(key, value): добавляет или обновляет элемент с заданным ключом (объектом) и значением.
- get(key): возвращает значение, соответствующее ключу.
- has(key): возвращает true, если ключ существует в коллекции.
- delete(key): удаляет элемент по ключу.

```
let weakMap = new WeakMap();
let obj = {};

weakMap.set(obj, 'Some value');

console.log(weakMap.get(obj)); // 'Some value'

// После удаления ссылки на объект, он может быть очищен сборщиком мусора
obj = null;

// В weakMap больше нет ключа,
// и его соответствующее значение будет удалено сборщиком мусора
```

Первое его отличие от Map в том, что ключи в WeakMap должны быть объектами, а не примитивными

WeakMap не поддерживает перебор и методы keys(), values(), entries(), так что нет способа взять все ключи или значения из неё.

WeekSet

WeakSet - это коллекция, содержащая только уникальные объекты, где ссылки на эти объекты также являются "слабыми". Если объект в WeakSet становится недоступным и больше нигде не используется, он может быть удален сборщиком мусора.

Задачи:

- Хранение уникальных объектов без препятствования сбору мусора.
- Управление временными данными, связанными с объектами.

Методы:

- add(value): добавляет объект в коллекцию.
- has(value): возвращает true, если объект существует в коллекции.
- delete(value): удаляет объект из коллекции.

```
let weakSet = new WeakSet();
let obj = {};

weakSet.add(obj);

console.log(weakSet.has(obj)); // true

// После удаления ссылки на объект, он может быть очищен сборщиком мусора
obj = null;

// В weakSet больше нет объекта, и он будет удален сборщиком мусора
```

Первое его отличие от Set в том, что ключи в WeakSet должны быть объектами, а не примитивными

WeakSet не поддерживает перебор и методы keys(), values(), entries(), так что нет способа взять все ключи или значения из неё.

Различия между Map/Set и WeakMap/WeakSet

Ключи и значения:

- Map может использовать любые типы данных в качестве ключей, а WeakMap только объекты.
- Set может содержать любые типы данных, а WeakSet только объекты.

Сборка мусора:

- В WeakMap и WeakSet ссылки на объекты являются слабыми, что позволяет сборщику мусора автоматически удалять элементы, если на них больше нет ссылок.
- Map и Set сохраняют сильные ссылки на элементы, что может приводить к утечкам памяти, если объекты больше не нужны, но все еще существуют ссылки на них.

Итерация:

- Map и Set поддерживают методы итерации (forEach, keys, values, entries).
- WeakMap и WeakSet не поддерживают итерацию или методы, возвращающие список всех ключей или значений, чтобы предотвратить случайное удержание объектов.

Функция с кэшированием результата

Функция createCacheableFunction(fn):

- Принимает функцию fn, результаты которой нужно кэшировать.
- Создает новый экземпляр Map для хранения кэша.

Возвращаемая функция:

- Принимает произвольное количество аргументов (используем оператор ...args).
- Преобразует аргументы в строку с помощью JSON.stringify(args) для создания ключа кэша.
- Проверяет, есть ли уже вычисленный результат для этих аргументов в кэше с помощью cache.has(key).
- Если результат есть, возвращает его из кэша.
- Если результата нет, вызывает исходную функцию fn, сохраняет результат в кэше и затем возвращает его.

Пример использования:

- Создаем кэшируемую версию функции complexCalculation.
- При первом вызове с новыми аргументами результат вычисляется и сохраняется в кэше.
- При повторных вызовах с теми же аргументами результат берется из кэша, что позволяет избежать повторных вычислений.

```
function createCacheableFunction(fn) {
    const cache = new Map();

    return function(...args) {
        const key = JSON.stringify(args);
        if (cache.has(key)) {
            console.log('Fetching from cache:', key);
            return cache.get(key);
        }

        console.log('Calculating result for:', key);
        const result = fn(...args);
        cache.set(key, result);
        return result;
    };
}

// Пример сложной функции, результаты которой будем кэшировать
function complexCalculation(a, b) {
    // Имитация сложных вычислений
    return a + b;
}

// Создаем кэшируемую версию функции
const cachedCalculation = createCacheableFunction(complexCalculation);

// Использование кэшируемой функции
console.log(cachedCalculation(3, 5)); // Calculating result for: [3,5]
console.log(cachedCalculation(3, 5)); // Fetching from cache: [3,5]
console.log(cachedCalculation(4, 5)); // Calculating result for: [4,5]
console.log(cachedCalculation(3, 5)); // Fetching from cache: [3,5]
```

Ресурсы

Map и Set статья из учебника - ТЫК

WeakMap и WeakSet статья из учебника - ТЫК

MDN документация Map - ТЫК

MDN документация Set - ТЫК

MDN документация WeakMap - ТЫК

MDN документация WeakSet - ТЫК