WeakMap и WeakSet

Как мы знаем из главы Сборка мусора, движок JavaScript хранит значения в памяти до тех пор, пока они достижимы (то есть, эти значения могут быть использованы).

Например:

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

// объект доступен, переменная john -- это ссылка на не

// перепишем ссылку
john = null;

// объект будет удалён из памяти
```

Обычно свойства объекта, элементы массива или другой структуры данных считаются достижимыми и сохраняются в памяти до тех пор, пока эта структура данных содержится в памяти.

Например, если мы поместим объект в массив, то до тех пор, пока массив существует, объект также будет существовать в памяти, несмотря на то, что других ссылок на него нет.

Например:

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

let array = [ john ];

john = null; // перезаписываем ссылку на объект

// объект john хранится в массиве, поэтому он не будет
// мы можем взять его значение как array[0]
```

Аналогично, если мы используем объект как ключ в `Map`, то до тех пор, пока существует `Map`, также будет существовать и этот объект. Он занимает место в памяти и не может быть удалён сборщиком мусора.

Например:

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

let map = new Map();
map.set(john, "...");

john = null; // перезаписываем ссылку на объект

// объект john сохранён внутри объекта `Map`,
// он доступен через map.keys()
```

`WeakMap` – принципиально другая структура в этом аспекте. Она не предотвращает удаление объектов сборщиком мусора, когда эти объекты выступают в качестве ключей.

Давайте посмотрим, что это означает, на примерах.

WeakMap

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

```
let weakMap = new WeakMap();

let obj = {};

weakMap.set(obj, "ok"); // работает (объект в качестве

// нельзя использовать строку в качестве ключа
weakMap.set("test", "Whoops"); // Ошибка, потому что "t
```

Теперь, если мы используем объект в качестве ключа и если больше нет ссылок на этот объект, то он будет удалён из памяти (и из объекта `WeakMap`) автоматически.

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

let weakMap = new WeakMap();
weakMap.set(john, "...");

john = null; // перезаписываем ссылку на объект

// объект john удалён из памяти!
```

Сравните это поведение с поведением обычного `Map`, пример которого был приведён ранее. Теперь `john` существует только как ключ в `WeakMap` и может быть удалён оттуда автоматически.

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

В `WeakMap` присутствуют только следующие методы:

- `weakMap.get(key)`
- `weakMap.set(key, value)`
- `weakMap.delete(key)`
- `weakMap.has(key)`

К чему такие ограничения? Из-за особенностей технической реализации. Если объект станет недостижим (как объект `john` в примере выше), то он будет автоматически удалён сборщиком мусора. Но нет информации, *в какой момент произойдёт эта очистка*.

Решение о том, когда делать сборку мусора, принимает движок JavaScript. Он может посчитать необходимым как удалить объект прямо сейчас, так и отложить эту операцию, чтобы удалить большее количество объектов за раз позже. Так что технически количество элементов в коллекции `WeakMap` неизвестно. Движок может произвести очистку сразу или потом, или сделать это частично. По этой причине методы для доступа ко всем сразу ключам/значениям недоступны.

Но для чего же нам нужна такая структура данных?

Пример: дополнительные данные

В основном, `WeakMap` используется в качестве *дополнительного хранилища данных*.

Если мы работаем с объектом, который «принадлежит» другому коду, может быть даже сторонней библиотеке, и хотим сохранить у себя какие-то данные для него, которые должны существовать лишь пока существует этот объект, то `WeakMap` – как раз то, что нужно.

Мы кладём эти данные в `WeakMap`, используя объект как ключ, и когда сборщик мусора удалит объекты из памяти, ассоциированные с ними данные тоже автоматически исчезнут.

```
weakMap.set(john, "секретные документы");
```

```
2  // если john умрёт, "секретные документы" будут автомат
```


Давайте рассмотрим один пример.

Предположим, у нас есть код, который ведёт учёт посещений для пользователей. Информация хранится в коллекции `Map` : объект, представляющий пользователя, является ключом, а количество визитов – значением. Когда пользователь нас покидает (его объект удаляется сборщиком мусора), то больше нет смысла хранить соответствующий счётчик посещений.

Вот пример реализации счётчика посещений с использованием `Map` :

```
1  // 🐚 visitsCount.js
2  let visitsCountMap = new Map(); // map: пользователь =>
3
4  // увеличиваем счётчик
5  function countUser(user) {
6    let count = visitsCountMap.get(user) || 0;
7    visitsCountMap.set(user, count + 1);
8  }
```

А вот другая часть кода, возможно, в другом файле, которая использует `countUser` :

```
1  // 🐚 main.js
2  let john = { name: "John" };
3
4  countUser(john); //ведём подсчёт посещений
5
6  // пользователь покинул нас
7  john = null;
```

Теперь объект `john` должен быть удалён сборщиком мусора, но он продолжает оставаться в памяти, так как является ключом в `visitsCountMap` .

Нам нужно очищать `visitsCountMap` при удалении объекта пользователя, иначе коллекция будет бесконечно расти. Подобная очистка может быть неудобна в реализации при сложной архитектуре приложения.

Проблемы можно избежать, если использовать `WeakMap` :

```
1  // 🐚 visitsCount.js
2  let visitsCountMap = new WeakMap(); // map: пользовател
3
4  // увеличиваем счётчик
5  function countUser(user) {
6    let count = visitsCountMap.get(user) || 0;
7    visitsCountMap.set(user, count + 1);
8  }
```

Теперь нет необходимости вручную очищать `visitsCountMap` . После того, как объект `john` стал недостижим другими способами, кроме как через `WeakMap` , он удаляется из памяти вместе с информацией по такому ключу из `WeakMap` .

Применение для кеширования

Другая частая сфера применения – это кеширование, когда результат вызова функции должен где-то запоминаться («кешироваться») для того, чтобы дальнейшие её вызовы на том же объекте могли просто брать уже готовый результат, повторно используя его.

Для хранения результатов мы можем использовать `Map` , вот так:

```
1  // 🐚 cache.js
```

```
let cache = new Map();

// вычисляем и запоминаем результат
function process(obj) {
  if (!cache.has(obj)) {
    let result = /* тут какие-то вычисления результата ,

    cache.set(obj, result);
  }

  return cache.get(obj);
}

// Теперь используем process() в другом файле:

// 🐬 main.js
let obj = {/* допустим, у нас есть какой-то объект */};

let result1 = process(obj); // вычислен результат

// ...позже, из другого места в коде...
let result2 = process(obj); // ранее вычисленный результ

// ...позже, когда объект больше не нужен:
obj = null;

alert(cache.size); // 1 (Упс! Объект всё ещё в кеше, за
```

Многократные вызовы `process(obj)` с тем же самым объектом в качестве аргумента ведут к тому, что результат вычисляется только в первый раз, а затем последующие вызовы берут его из кеша. Недостатком является то, что необходимо вручную очищать `cache` от ставших ненужными объектов.

Но если мы будем использовать `WeakMap` вместо `Map`, то эта проблема исчезнет: закешированные результаты будут автоматически удалены из памяти сборщиком мусора.

```
// 🐬 cache.js
let cache = new WeakMap();

// вычисляем и запоминаем результат
function process(obj) {
  if (!cache.has(obj)) {
    let result = /* вычисляем результат для объекта */ 

    cache.set(obj, result);
  }

  return cache.get(obj);
}

// 🐬 main.js
let obj = {/* какой-то объект */};

let result1 = process(obj);
let result2 = process(obj);

// ...позже, когда объект больше не нужен:
obj = null;

// Нет возможности получить cache.size, так как это Wea
// но он равен 0 или скоро будет равен 0
// Когда сборщик мусора удаляет obj, связанные с ним да
```

WeakSet

Коллекция `WeakSet` ведёт себя похоже:

- Она аналогична `Set`, но мы можем добавлять в `WeakSet` только объекты (не примитивные значения).
- Объект присутствует в множестве только до тех пор, пока доступен где-то ещё.
- Как и `Set`, она поддерживает `add`, `has` и `delete`, но не `size`, `keys()` и не является перебираемой.

Будучи «слабой» версией оригинальной структуры данных, она тоже служит в качестве дополнительного хранилища. Но не для произвольных данных, а скорее для значений типа «да/нет». Присутствие во множестве `WeakSet` может что-то сказать нам об объекте.

Например, мы можем добавлять пользователей в `WeakSet` для учёта тех, кто посещал наш сайт:

```
let visitedSet = new WeakSet();

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

visitedSet.add(john); // John заходил к нам
visitedSet.add(pete); // потом Pete
visitedSet.add(john); // John снова

// visitedSet сейчас содержит двух пользователей

// проверим, заходил ли John?
alert(visitedSet.has(john)); // true

// проверим, заходила ли Mary?
alert(visitedSet.has(mary)); // false

john = null;

// структура данных visitedSet будет очищена автоматиче
```

Наиболее значительным ограничением `WeakMap` и `WeakSet` является то, что их нельзя перебрать или взять всё содержимое. Это может доставлять неудобства, но не мешает `WeakMap/WeakSet` выполнять их главную задачу – быть дополнительным хранилищем данных для объектов, управляемых из каких-то других мест в коде.

Итого

`WeakMap` – это `Map`-подобная коллекция, позволяющая использовать в качестве ключей только объекты, и автоматически удаляющая их вместе с соответствующими значениями, как только они становятся недостижимыми иными путями.

`WeakSet` – это `Set`-подобная коллекция, которая хранит только объекты и удаляет их, как только они становятся недостижимыми иными путями.

Обе этих структуры данных не поддерживают методы и свойства, работающие со всем содержимым сразу или возвращающие информацию о размере коллекции. Возможны только операции на отдельном элементе коллекции.

`WeakMap` и `WeakSet` используются как вспомогательные структуры данных в дополнение к «основному» месту хранения объекта. Если объект удаляется из основного хранилища и нигде не используется, кроме как в качестве ключа в `WeakMap` или в `WeakSet`, то он будет удалён автоматически.

Задачи

Хранение отметок "не прочитано"

важность: 5

Есть массив сообщений:

```
let messages = [
  {text: "Hello", from: "John"},
  {text: "How goes?", from: "John"},
  {text: "See you soon", from: "Alice"}
];
```

У вас есть к ним доступ, но управление этим массивом происходит где-то ещё. Добавляются новые сообщения и удаляются старые, и вы не знаете в какой момент это может произойти.

Имея такую вводную информацию, решите, какую структуру данных вы могли бы использовать для ответа на вопрос «было ли сообщение прочитано?». Структура должна быть подходящей, чтобы можно было однозначно сказать, было ли прочитано это сообщение для каждого объекта сообщения.

P.S. Когда сообщение удаляется из массива `messages`, оно должно также исчезать из структуры данных.

P.P.S. Нам не следует модифицировать сами объекты сообщений, добавлять туда свойства. Если сообщения принадлежат какому-то другому коду, то это может привести к плохим последствиям.

решение

Хранение времени прочтения

важность: 5

Есть массив сообщений такой же, как и в предыдущем задании.

```
let messages = [
  { text: "Hello", from: "John" },
  { text: "How goes?", from: "John" },
  { text: "See you soon", from: "Alice" }
];
```

Теперь вопрос стоит так: какую структуру данных вы бы предложили использовать для хранения информации о том, когда сообщение было прочитано?

В предыдущем задании нам нужно было сохранить только факт прочтения «да или нет». Теперь же нам нужно сохранить дату, и она должна исчезнуть из памяти при удалении «сборщиком мусора» сообщения.

P.S. Даты в JavaScript можно хранить как объекты встроенного класса `Date`, которые мы разберём позднее.

решение

Комментарии

перед тем как писать...