МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Кафедра Обчислювальної техніки

Пояснювальна записка

до курсової роботи з дисципліни

“Основи програмування - 2.  
Програмування структурованих даних”

на тему

“Транзакціонні обєкти: enhanced event emitter”

|  |  |
| --- | --- |
| Керівник: Шемсединов Т. Г.  Допущений до захисту  2018-06-01 \_\_\_\_\_\_\_\_\_\_\_\_\_\_  підпис | Виконав: Іващук В. А.  Студент гр. ІП-73, ФІОТ 1 курс  № ІП-7311  2018-06-01 \_\_\_\_\_\_\_\_\_\_\_\_\_\_  підпис |
| Перевірив: Шемсединов Т.Г.  Захистив з оцінкою \_\_\_\_\_\_\_\_\_\_  2018-06-01 \_\_\_\_\_\_\_\_\_\_\_\_\_\_  підпис |  |

Київ 2018

Зміст

|  |  |
| --- | --- |
| Вступ | 2 |
| Розділ 1. Стан проблеми і пошук рішень | 3 |
| 1.1. Постановка цілі та задач роботи | 3 |
| 1.2. Аналіз відомих методів вирішення проблеми | 5 |
|  | 7 |
| Розділ 2. Хід розробки | 9 |
| 2.1. Опис структури проекту |  |
|  |  |
| 2.2. Програмна реалізація і її особливості | 13 |
|  |  |
| 2.3. Аналіз і тестування роботи прорамного продукту | 16 |
|  |  |
|  | 18 |
| Висновки | 19 |
| Список літератури | 20 |
| Додатки | 21 |
| A. Програмний код | 21 |
| Б. Скріншоти | 30 |
| В. Диск с кодом | 31 |

# Вступ

*“EventEmitter — это универсальная абстракция для работы с событиями чере подписку (subscription: addListener, on, once) и отправку (emit)” [Тимур Шемседінов]*

Event Emitter можна перекласти як «транслятор» або «еміттер» подій. Event Emitter дозволяє частинам програми взаємодіяти між собою.

Значна частина основного API Node.js побудована на ідіоматичній асинхронній архітектурі, керованій подіями. В ній певні види об'єктів (так звані "еміттери" або “”emitters”) виділяють іменовані події, які викликають функції ("слухачі", “лісенери” або “listeners”).

Всі об'єкти, які викликають події, є екземплярами класу EventEmitter. Ці об'єкти мають метод eventEmitter.on(), який дозволяє додавати одну або декілька функцій до події, що викликаються об'єктом.

Коли об'єкт EventEmitter викликає подію, всі функції, додані до цієї конкретної події, викликаються синхронно. Будь-які значення, які повертаються викликаними лісенерами, ігноруються.

Транзакціонні об’єкти – це програмна абстракція, що дозволяє управління об’єктом за допомогою транзакцій. Це означає, що зміни, які повинні бути здійснені над обєктом можуть накопичуватись в пам’яті і в подальшому бути підтвердженими (.commit) або відхиленими (.rollback). В першому випадку зміни перенесуться на об’єкт, а в другому – об’єкт залишиться в свому початковому стані, а буфер для зберігання змін (“дельта”) буде очищений.

Цей паттерн сильно розширює можливості роботи з об’єктами. А враховуючи те, що в якості об’єкта може бути практично будь-яка структура даних, він має широке застосування і перспективи розвитку.

Доцільність використання event emitter в транзакціонних об’єктах полягає в тому, що часто в програмах під час виклику методів транзакціонного об’єкта (таких як .commit чи .rollback), є необхідність виклику низки інших функцій, причому їх склад може залежати від ходу програми.

Більше того, може виникати потреба виклику деяких функцій до метода транзакціонного об’єкта, а деяких – після. В таких випадках event emitter повинен мати різні методи для роботи з лісенерами до і після виклику метода транзакціонного об’єкта.

РОЗДІЛ 1. Стан проблеми і пошук рішень

1.1 Постановка цілі та задач роботи

В рамках даної роботи повинна бути розроблена програмна реалізація розширеного event emitter для використання в проекті транзакціонних об’єктів. Розробка повинна повністю задовольняти потреби даного проекту, а також бути ефективною в плані затрат часу.

Саме тому важливо підібрати правильну структуру проекту. Вона повинна повністю підходити для потреб транзакційних об’єктів.

Основою event emitter є структура даних, в якій зберігаються лісенери. З цією структурою працюють практично усі методи, тому вибір може мати великий вплив на продуктивність роботи програми.

Вимоги транзакціонних об’єктів до event emitter включають в себе максимально повний функціонал для роботи з подіями. Це в першу чергу наявність великого переліку методів, що охоплюють всі можливі потреби, а також це поділ методів для виконання як перед викликом методу транзакції так і після нього.

Отже, задачами даної роботи є:

- визначення оптимальної структури проекту;

- вибір оптимальної структури даних для зберігання лісенерів;

- підбір методів, необхідних для роботи event emitter;

- реалізація вище згаданого в програмному коді

1.2 Аналіз методів реалізації проекту і вибір оптимального

Для того, щоб для кожної транзакції можна було створити свій event emitter, для нього повинен бути створений клас. Тоді при кожному інстанціюванні транзакціонного об’єкта буде створюватись також і екземпляр класу event emitter.

В JavaScript є декілька способів реалізації аналогу класу:

- в функціональному стилі;

- в прототипному стилі;

- за допомогою конструкції class;

- через замикання.

Третій варіант являється по суті функціональною реалізацією, загорнутою в синтаксичне оформлення. Тобто від першого варіанту він відрізняється тільки, можливо, зручністю користування, і відповідно меншою продуктивністю.

Порівнюючи функціональний і прототипний стилі, слід зазначити, що, знову ж таки, перевагою функціонального стилю може бути лише зручність, а прототипний стиль впевнено виграє в продуктивності. Зумовлено це тим, що функціональний стиль записує в кожен об’єкт і властивості і методи, а

прототипний — тільки властивості. Тому прототипний стиль — швидший і економніший по пам’яті.

Але, незважаючи на це, по продуктивності і функціональний і прототипний стилі програють замиканню. За рахунок своєї простоти і швидкості замикання являється оптимальним варіантом реалізації такої структури як event emitter з розрахунку на подальше його використання в проекті транзакціонних об’єктів.

Наступним, не менш важливим завданням роботи є вибір правильної структури даних для зберігання лісенерів. Цей вибір має бути заснованим на продуктивності і швидкості, оскільки на роботі з цією структурою даних і побудований event emitter і його методи.

Найпростішим вибором міг би стати масив, або об’єкт з масивами. Тоді в кожному внутрішньому масиві зберігався би набір лісенерів, що відповідають імені масива тобто події. Але якщо є потреби в створенні нових подій під час роботи програми, то така структура не є найкращим вибором. В цьому випадку оптимальним буде використання Map. Map являє собою структуру вигляду ключ-значення і тому підходить для випадку, коли події додаються під час роботи програми, а звертання до події відбувається по імені.

А оскільки кожен метод event emitter повинен бути розділений на виконання перед викликом методу транзакцій і після нього, то Map повинно бути розділено так само. Отже структура буде мати вигляд об’єкта подій з двома Map для зберігання лісенерів.

РОЗДІЛ 2. Хід розробки

2.1 Опис структури проекту

В результаті проведеного аналізу було вирішено, що event emitter буде реалізовано на замиканні. Функція в основі замикання не приймає ніяких аргументів, оскільки при інстанціюванні event emitter вони не потрібні. Всередині замикання повинно бути два об’єкта.

Перший — це структура даних для зберігання лісенерів, а другий — це сам event emitter, який створюється в замиканні і повертається з нього. В результаті цього він має доступ до замикання, а саме до структури з подіями. Методи ж для роботи з ними зберігаються в event emitter і оголошуються всередині замикання.

Сюди повинні входитити необхідні методи для роботи з подіями, це:

- підписка — додавання лісенера на вибрану подію;

- виклик — виклик усіх лісенерів вибраної події;

- одинарна підписка — лісенер видалаяється після першого виклику;

- видалення — видалення лісенера з вибраної події;

- очистка — видалення всіх лісенерів вибраної події;

- обрахунок — повертається кількість лісенерів на події;

- лісенери — повертається список лісенерів на події;

- імена — повертається список імен подій.

Як говорилось раніше, кожен з цих методів повинен бути розділений на виконання перед та після методу транзакції. Тому і структура для зберігання лісенерів поділена таким самим чином. Також для можливості видалення одинарної підписки, повинна бути створена ще одна така структура, тільки для зберігання функцій, що видаляють лісенер після першого виклику.

2.2 Програмна реалізація і її особливості

2.3 Аналіз і тестування роботи програмного продукту

ВИСНОВКИ

Список літератури

1. Aho A. V., Sethi R., Ullman J. D. (1986). Compilers: principles, techniques, and tools. Addison- Wesley Longman Publishing Co., Inc. Boston, MA, USA.

2. Pratt Vaughan. Top down operator precedence. Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (1973)

3. The JSON Data Interchange Format [Електронний ресурс]. – Режим доступу: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf

4. Node.js v5.10.1 Documentation. Executing JavaScript [Електронний ресурс]. – Режим доступу https://nodejs.org/api/vm.html

5. Ecmascript 2015 language specification [Електронний ресурс]. – Режим доступу: http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf

6. Hypertext Transfer Protocol -- Http/1.1 [Електронний ресурс]. – Режим доступу: https://www.w3.org/Protocols/rfc2616/rfc2616.html

Додатки

A. Програмний код

ee.js

'use strict';

const emitter = () => {

const events = {

before: new Map(),

after: new Map()

};

const wrapped = {

before: new Map(),

after: new Map()

};

const ee = {

on: {

before: (name, f) => {

const event = events.before.get(name);

if (!event) {

events.before.set(name, [f]);

} else {

event.push(f);

events.before.set(name, event);

}

},

after: (name, f) => {

const event = events.after.get(name);

if (!event) {

events.after.set(name, [f]);

} else {

event.push(f);

events.after.set(name, event);

}

}

},

emit: {

before: (name, ...data) => {

const event = events.before.get(name);

if (event) event.forEach(f => f(...data));

},

after: (name, ...data) => {

const event = events.after.get(name);

if (event) event.forEach(f => f(...data));

}

},

once: {

before: (name, f) => {

const g = (...a) => {

ee.remove.before(name, g);

f(...a);

};

wrapped.before.set(f, g);

ee.on.before(name, g);

},

after: (name, f) => {

const g = (...a) => {

ee.remove.after(name, g);

f(...a);

};

wrapped.after.set(f, g);

ee.on.after(name, g);

}

},

remove: {

before: (name, f) => {

const event = events.before.get(name);

if (!event) return;

let i = event.indexOf(f);

if (i !== -1) {

event.splice(i, 1);

return;

}

const g = wrapped.before.get(f);

if (g) {

i = event.indexOf(g);

if (i !== -1) event.splice(i, 1);

if (!event.length) events.before.delete(name);

}

},

after: (name, f) => {

const event = events.after.get(name);

if (!event) return;

let i = event.indexOf(f);

if (i !== -1) {

event.splice(i, 1);

return;

}

const g = wrapped.after.get(f);

if (g) {

i = event.indexOf(g);

if (i !== -1) event.splice(i, 1);

if (!event.length) events.after.delete(name);

}

}

},

clear: {

before: (name) => {

if (name) events.before.delete(name);

else events.before.clear();

},

after: (name) => {

if (name) events.after.delete(name);

else events.after.clear();

}

},

count: {

before: (name) => {

const event = events.before.get(name);

return event ? event.length : 0;

},

after: (name) => {

const event = events.after.get(name);

return event ? event.length : 0;

}

},

listeners: {

before: (name) => {

const event = events.before.get(name);

return event.slice();

},

after: (name) => {

const event = events.after.get(name);

return event.slice();

}

},

names: {

before: () => [...events.before.keys()],

after: () => [...events.after.keys()]

}

};

return ee;

};

module.exports = emitter;

// Usage

const ee = emitter();

// on and emit

ee.on.after('e1', (data) => {

console.dir(data);

});

ee.emit.after('e1', { msg: 'e1 ok' });

// once

ee.once.after('e2', (data) => {

console.dir(data);

});

ee.emit.after('e2', { msg: 'e2 ok' });

ee.emit.after('e2', { msg: 'e2 not ok' });

// remove

const f3 = (data) => {

console.dir(data);

};

ee.on.before('e3', f3);

ee.remove.before('e3', f3);

ee.emit.before('e3', { msg: 'e3 not ok' });

// count

ee.on.before('e4', () => {});

ee.on.before('e4', () => {});

console.log('e4 count', ee.count.before('e4'));

// clear

ee.clear.after('e4');

ee.emit.after('e4', { msg: 'e4 not ok' });

ee.emit.after('e1', { msg: 'e1 ok' });

ee.clear.after();

ee.emit.after('e1', { msg: 'e1 not ok' });

// listeners and names

ee.on.before('e5', () => {});

ee.on.after('e5', () => {});

ee.on.after('e6', () => {});

ee.on.after('e7', () => {});

console.log('listeners.before', ee.listeners.before('e5'));

console.log('listeners.after', ee.listeners.after('e5'));

console.log('names', ee.names.after());

transaction.js

'use strict';

const emitter = require('./ee.js');

function Transaction() {}

Transaction.start = (data) => {

console.log('start transaction');

let delta = {};

const methods = {

commit: () => {

console.log('commit transaction');

Object.assign(data, delta);

delta = {};

},

rollback: () => {

console.log('rollback transaction');

delta = {};

},

clone: () => {

console.log('clone transaction');

const cloned = Transaction.start(data);

Object.assign(cloned.delta, delta);

return cloned;

},

delta: () => delta,

on: (name, callback) => {

const event = events[name];

if (event) event.push(callback);

}

};

Object.assign(methods, emitter());

const getKeys = () => {

const changes = Object.keys(delta);

const keys = Object.keys(data).concat(changes);

return keys.filter((x, i, a) => a.indexOf(x) === i);

};

const proxy = new Proxy(data, {

get(target, key, proxy) {

if (key === Symbol.iterator) return getKeys()[key]();

if (methods.hasOwnProperty(key)) return methods[key];

if (delta.hasOwnProperty(key)) return delta[key];

return target[key];

},

ownKeys(target) {

return getKeys();

},

getOwnPropertyDescriptor: (target, key) => (

Object.getOwnPropertyDescriptor(

delta.hasOwnProperty(key) ? delta : target, key

)

),

set(target, key, val) {

console.log('set', key, val);

if (target[key] === val) delete delta[key];

else delta[key] = val;

return true;

}

});

return proxy;

};

// Usage

const data = { name: 'Marcus Aurelius', born: 121 };

const transaction = Transaction.start(data);

console.log('data', JSON.stringify(data));

console.log('transaction', JSON.stringify(transaction));

transaction.name = 'Mao Zedong';

transaction.born = 1893;

transaction.city = 'Shaoshan';

console.log('\noutput with JSON.stringify:');

console.log('data', JSON.stringify(data));

console.log('transaction', JSON.stringify(transaction));

console.log('\noutput with console.dir:');

console.dir({ transaction });

console.log('\noutput with for-in:');

for (const key in transaction) {

console.log(key, transaction[key]);

}

transaction.commit();

console.log('data', JSON.stringify(data));

console.log('transaction', JSON.stringify(transaction));

transaction.born = 1976;

console.log('data', JSON.stringify(data));

console.log('transaction', JSON.stringify(transaction));

transaction.rollback();

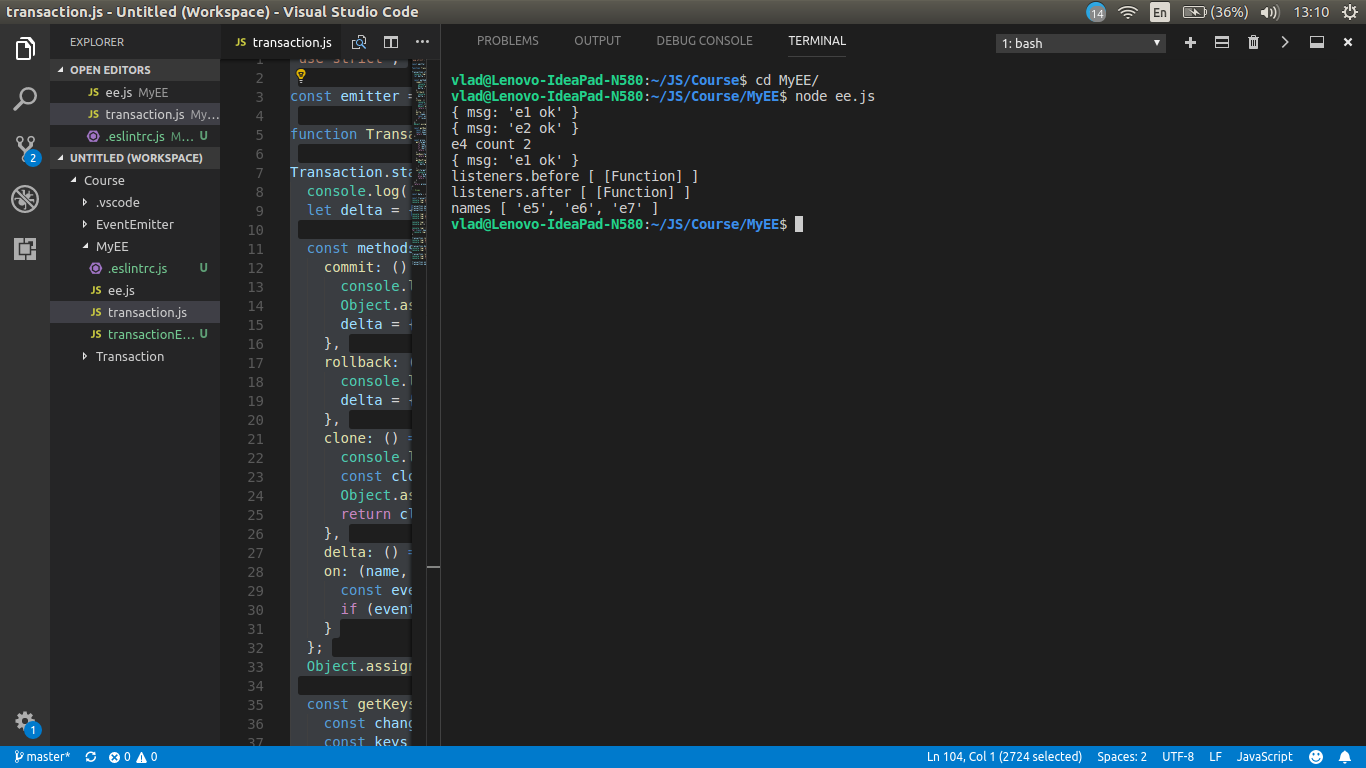
console.log('data', JSON.stringify(data));

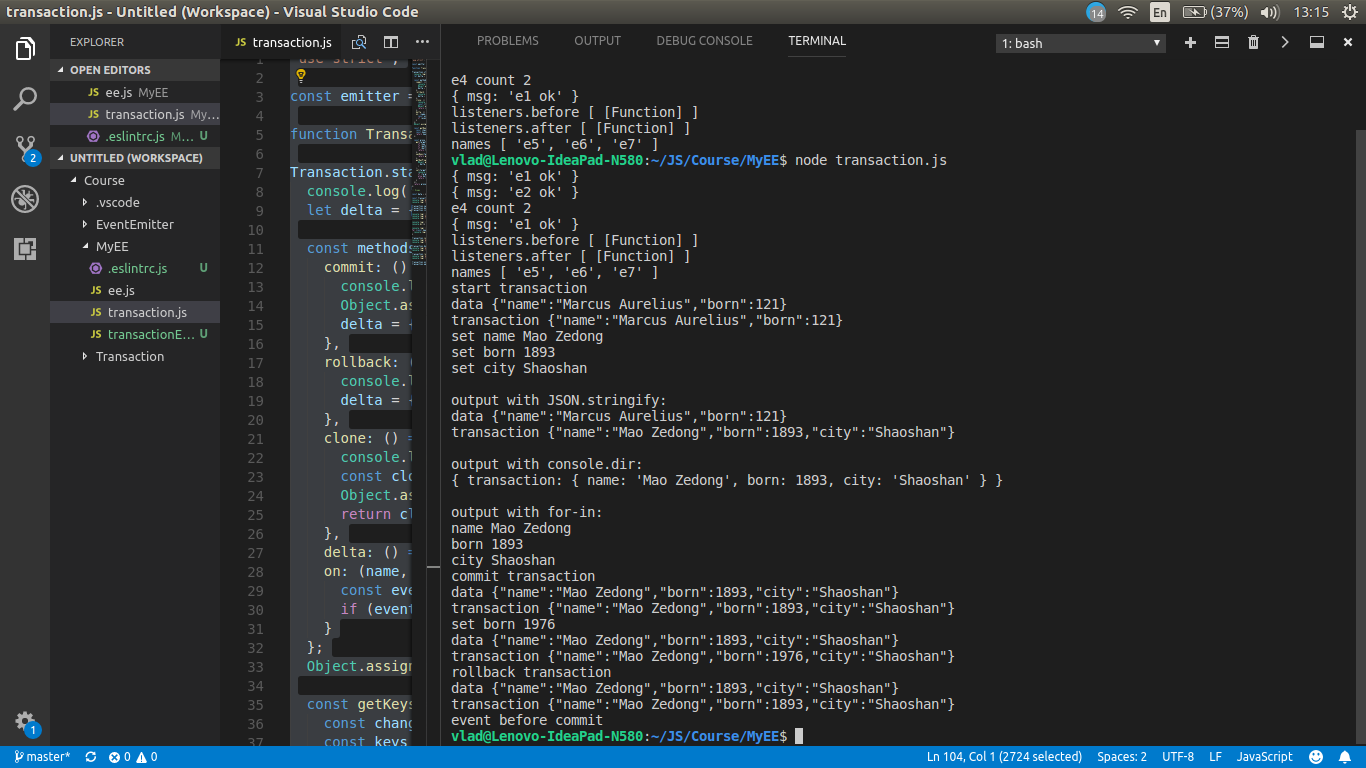
console.log('transaction', JSON.stringify(transaction));

transaction.on.before('commit', () => console.log('event before commit'));

transaction.emit.before('commit');

Б. Скріншоти





В. Диск с кодом

{ Place CD or DVD disk here }