Puritate si Imutabilitate

Curs 6

Gabriel Danciu

D.E.C.

17 aprilie 2024

Prezentare Generală a Cursului

Puritate in programare funcțională

Să ne imaginăm că avem nevoie de o funcție care, atunci când primește un număr, returnează un număr (pseudo)aleator mai mare decât 0 și inclusiv până la numărul respectiv. Funcția *random* din Underscore este aproape corectă, dar include implicit și valoarea zero.

o posibilă alternativă

```
function rand(n) {
return 1 + Math.floor(Math.random() * n);
}

function repeatedly(count, fun) {
    var result = [];
    for (var i = 0; i < count; i++) {
        result.push(fun());
    }
    return result;
}
```

Un exemplu de folosire a generarii aleatorie

Să luăm drept exemplu generarea unui sir de caractere de o lungime dată ca parametru

Funcția generateString

```
function randString(len) {
    var ascii = repeatedly(len, function() { return rand
        (26); });
    return ascii.map(function(n) {
        return n.toString(36);
    }).join('');
}
console.log(randString(2))
console.log(randString(20))
```

Deși randString se încadrează tehnic în definiția unei funcții de nivel înalt, există o diferență mare în modul în care randString este construită față de modul în care au fost construite funcțiile anterioare. Care?

Puritate și testare

Pentru testarea funcției randString, provocarea constă în validarea unei funcții care produce rezultate aleatorii. Totuși, puteți să vă asigurați că funcția se comportă corect verificând caracteristicile rezultatului, mai degrabă decât rezultatul în sine.

assert

```
import * as chai from 'chai';
const assert = chai.assert;
var result = randString(10);
assert.equal(result,"fixedoutput"); //eroare
var result = randString(10);
assert.match(result, /^[0-9a-z]{10}$/, 'rezultatul se
    potriveste cu formatul asteptat');
```

Separarea celor 2 concepte

Separarea funcțiilor pure de cele impure într-un cod sursă este o strategie eficientă pentru a îmbunătăți testabilitatea, mentenabilitatea și predictibilitatea software-ului.
Această abordare este deosebit de benefică în JavaScript, unde multe operațiuni comune sunt impure datorită efectelor secundare sau dependenței de starea externă.

- Generarea caracterelor aleatoare. Această parte rămâne impură deoarece depinde de generatorul de numere aleatoare.
- Construirea șirului. Această parte poate fi pură, deoarece construiește simplu un șir dintr-un set dat de caractere.

Funcțiile separate

În exemplul dat, *generateRandomCharacter* este o funcție care generează un caracter aleator între 'a' și 'z'. *partial*1 este folosit pentru a crea o nouă funcție, *composedRandomString*, care este de fapt *generateString* cu *generateRandomCharacter* ca prim argument presetat. Restul argumentelor (len în acest caz) pot fi furnizate când funcția este apelată.

Exemplu de utilizare a predicatelor

```
function generateString(charGen, len) {
    return repeatedly(len, charGen).join('');
}

function generateRandomCharacter() {
    return String.fromCharCode(97 + Math.floor(Math.
        random() * 26));
}
}
```

Evaluarea celor 2 funcții

Exemplu de utilizare

```
function partial1(func, arg1) {
    return function(...args) {
        return func(arg1, ...args);
    };
}
var composedRandomString = partial1(generateString,
    generateRandomCharacter);
console.log(composedRandomString(10));
```

Această tehnică este foarte utilă în JavaScript și în programarea funcțională în general, facilitând compunerea funcțiilor și reutilizarea codului. În contextul separării componentelor pure de cele impure, această abordare permite un control mai bun asupra comportamentului funcțiilor în teste, permițând injectarea dependențelor și simularea comportamentului într-un mod controlabil.

Puritate și transparență

Programarea cu funcții pure poate părea incredibil de limitativă. JavaScript, ca un limbaj extrem de dinamic, permite definirea și utilizarea funcțiilor fără o aderență strictă la tipurile argumentelor lor sau la valoarea de returnare. Uneori, această aderență se dovedește a fi problematică (de exemplu, $\text{true} + 1 === 2$). Foarte des, totuși, programatorii JavaScript echivalează capacitatea JavaScript-ului de a permite mutația liberă a variabilelor, obiectelor și sloturilor de array ca fiind esențială pentru dinamism.

Limitare

Factorul limitativ, și este unul cu care trebuie să trăim în JavaScript, este că funcția *nth* ar putea returna ceva ce este impur, cum ar fi un obiect, un array sau chiar o funcție impură:

Exemplu de "limitare"

```
function nth(array, index) {
    if (index < 0 || index >= array.length) {
        throw new Error("Index out of bounds");
    }
    return array[index];
}
nth([function() { console.log('blah') }], 0);
function second(a) {
    return nth(a, 1);
}
```

Evaluare apelurilor

Singura modalitate de a corecta această problemă (pe care o va duce mai departe funcția *second*) este să observi o aderență strictă la utilizarea și definirea funcțiilor pure care nu își modifică argumentele, nici nu depind de valori externe, exceptând cazurile în care astfel de efecte au fost minimizate în mod explicit.

Utilizarea lor

```
var a = nth(['a', 'b', 'c'], 1);
var b = ['a', 'b', 'c'];
var res = nth(b, 1);
result = (b ===  ['a', 'b', 'c']);
console.log(result);
res = _.isEqual(b, ['a', 'b', 'c']);
console.log(res);
```

Principiul imutabilității

Immutabilitatea este un concept important în programare, în special în limbaje ca JavaScript, unde managementul stării și a efectelor secundare poate deveni rapid complicat. Faptul că string-urile sunt imutabile în JavaScript este extrem de util pentru a preveni o serie de probleme complexe. Vom explora mai întâi conceptul de imutabilitate, exemplul dat mai jos, și apoi vom discuta despre mutabilitatea obiectelor.

Exemplu

```
var obj = {lemongrab: "Earl"};
(function(o) {
  _.extend(o, {lemongrab: "King"});
})(obj);
obj;
//=> {lemongrab: "King"}
```

Provocările gestionării Imutabilității în aplicațiile Enterprise

- Creșterea dimensiunii aplicației conduce la creșterea complexității dependențelor. Gestionarea acestor dependențe în contextul obiectelor mutabile poate deveni dificilă, făcând sistemul predispus la erori greu de urmărit și rezolvat.
- Obiectele mutabile pot introduce efecte secundare greu de prezis, complicând depanarea și testarea deoarece starea aplicației poate schimba neașteptat.
- Întreținerea și actualizarea repo-urilor de coduri considerabile, cu multe obiecte mutabile poate fi provocatoare, deoarece modificările într-o parte a sistemului pot afecta neintenționat alte părți.

Virtuțiile Imutabilității

- Obiectele imutabile nu se schimbă odată ce sunt create. Această predictibilitate înseamnă că poți pasa obiecte cu asigurarea că acestea nu vor fi modificate, simplificând raționamentul despre starea aplicației.
- Cu obiecte imutabile, fiecare schimbare de stare necesită crearea unui nou obiect. Acest model simplifică urmărirea modificărilor și gestionarea stării, în special în aplicații mari.
- Imutabilitatea complementează principiile programării funcționale, unde funcțiile sunt pure și returnează date noi în loc să modifice datele existente. Această abordare îmbunătățește claritatea și reutilizabilitatea codului.

Cum respecți acest principiu?

- a se utiliza *const*
- Immutable.js și biblioteci similare

Un copac care cade face zgomot in pădure

Întrebarea "Dacă un copac cade în pădure, face zgomot?" aplicată în programare, în special în contextul purității funcțiilor și efectelor secundare, ridică puncte interesante despre modul în care proiectăm și înțelegem codul. În contextul exemplului cu funcția *skipTake*, această interogare filozofică se concretizează în problema dată: mutațiile interne într-o funcție "fac zgomot" sau au un efect observabil dacă nu afectează comportamentul extern al funcției?

Exemplu

```
function skipTake(n, coll) {
    var ret = [];
    var sz = _.size(coll);
    for(var index = 0; index < sz; index += n) {
        ret.push(coll[index]);
    }
    return ret;
}
```

Mutabilitate Internă vs. Imutabilitate Externă

În această funcție, utilizarea internă a mutației (array-ul ret fiind construit cu Array.push) este încapsulată în interiorul funcției. Din perspectiva externă, funcția este pură:

- Funcția returnează întotdeauna același rezultat pentru aceleași intrări n și coll.
- Nu modifică array-ul dat ca parametru de intrare (coll).
- Nu depinde de sau nu modifică nicio stare externă.

Paralele filozofice

La fel ca și copacul care cade în pădure, dacă mutațiile interne dintr-o funcție nu afectează comportamentul sau rezultatele sale externe, ele ar putea fi considerate că "nu fac zgomot". Acesta este un mod de a spune că efectele unor astfel de mutații sunt neobservabile din exteriorul funcției, menținând astfel o puritate funcțională din perspectiva utilizatorului. Avantajele acestei abordări:

- Uneori, utilizarea construcțiilor mutabile intern poate duce la optimizări de performanță.
- Buclele imperative și mutațiile locale pot face implementarea mai ușor de scris și înțeles.
- Păstrând starea mutabilă locală funcției și neexpunând-o, funcția menține o interfață pură –> încapsulare.

Folosirea recursivității

Recursivitatea este adesea folosită în programarea funcțională deoarece se potrivește bine cu principiul imutabilității. În limbaje funcționale tradiționale, unde variabilele locale sunt imutabile, recursivitatea permite "mutația" valorilor printr-un mecanism natural, fără a necesita schimbarea stării locale.

Mutație local

```
function summ(array) {
    var result = 0;
    var sz = array.length;
    for (var i = 0; i < sz; i++)
        result += array[i];
    return result;
}
```

Folosirea recursivității

Aceasta versiune nu are nicio variabilă locală mutabilă. Starea este transmisă și modificată prin argumentul funcției (seed), respectând principiile imutabilității.

Fără mutație

```
function summRec(array, seed) {
    if (_.isEmpty(array))
        return seed;
    else
        return summRec(_.rest(array), _.first(array) +
            seed);
}
```

Funcțiile recursive pure sunt mai ușor de testat deoarece nu depind de starea externă sau locală mutabilă. Fără stări partajate mutabile, funcțiile recursive pure sunt mai sigure în medii in care apare concurența.

Imutabilitate la obiecte

Observând imutabilitatea în obiectele JavaScript înseamnă adesea să adoptăm strategii pentru a preveni modificările nedorite ale stării, mai ales în contextul obiectelor definite de utilizator. În exemplul dat cu obiectul Point, folosirea unor metode care returnează instanțe noi ale obiectului în loc să modifice instanțele existente este o strategie excelentă pentru a menține imutabilitatea.

Implementarea obiectului Point

```
function Point(x, y) {
    this._x = x;
    this._y = y;
}
Point.prototype = {
    withX: function(val) {
        return new Point(val, this._y);     },
    withY: function(val) {
        return new Point(this._x, val);     }
};
```

Imutabilitate la obiecte

Metodele *withX* și *withY* sunt exemple de cum se poate gestiona starea fără a o modifica direct. Acestea creează noi instanțe ale obiectului *Point* cu valorile actualizate, păstrând instanța originală nealterată:

Utilizarea metodelor imutabile

```
var p = new Point(0, 1);
var newP = p.withX(1000);
console.log(newP); //=> {_x: 1000, _y: 1}
console.log(p); //=> {_x: 0, _y: 1}
```

Problema imutabilității în tipul Queue

La fel ca Point, obiectul Queue ar trebui să mențină imutabilitatea. Cu toate acestea, folosind referința directă a array-ului primit la construcție, orice mutație asupra seed-ului afectează instanța de Queue:

O problemă

```
function Queue(elems) {
    this._q = elems;
}

Queue.prototype = {
    enqueue: function(thing) {
        return new Queue(this._q.concat([thing]));
    }
};
var seed = [1, 2, 3];
var q = new Queue(seed);
seed.push(10000);
console.log(q); //=> {_q: [1, 2, 3, 10000]}
```

Problema imutabilității în tipul Queue

Pentru a evita această capcană, puteți folosi clonarea pentru a proteja starea internă de modificări externe:

O soluție

```
var SaferQueue = function(elems) {
    this._q = _.clone(elems); // Clonarea superficiala
        este suficienta pentru exemplul dat
}

SaferQueue.prototype = {
    enqueue: function(thing) {
        return new SaferQueue(this._q.concat([thing]));
    }
};
var seed = [1, 2, 3];
var q = new SaferQueue(seed);
seed.push(10000);
console.log(q); //=> {_q: [1, 2, 3]}
```

Exercitii

- Scrieți o funcție recursivă care calculează factorialul unui număr. Creați metode care testează pe diferite scenarii această funcție.
- Scrieți o funcție care face o clonare profundă a unui obiect pentru a evita problemele de imutabilitate. Cod parțial ajutător mai jos

```
function deepClone(obj) {
    if obj nu este obiect sau e null
        return obj;
    temp = new obj.constructor();
    for each key in obj:
        if obj.hasOwnProperty(key)
            temp[key] = deepClone(obj[key]);
    return temp
}
```

- Să se implementeze o funcție recursivă care calculează suma,produsul, concatenarea elementelor unui array. Testați și timpii de execuție.