Programare înlănțuită

Curs 8

Gabriel Danciu

D.E.C.

19 mai 2024

Prezentare Generală a Cursului

Conceptul de înlănțuire

Conceptul de înlănțuire (chaining) în programare, în special în contextul programării funcționale, subliniază importanța proiectării funcțiilor și metodelor care pot fi conectate relativ ușor între ele printr-o interfață fluentă și intuitivă. Acest stil este folosit frecvent în biblioteci JavaScript cum ar fi jQuery și Underscore/Lodash, oferind o abordare mai declarativă operațiunilor care altfel ar necesita cod mai detaliat și mai puțin lizibil. Înlănțuirea metodelor este o tehnică ce implică crearea unei serii de metode într-un obiect, fiecare dintre acestea returnând obiectul însuși. Acest lucru permite apelarea metodelor într-o secvență continuă. Acest model este deosebit de puternic în JavaScript deoarece permite construirea de expresii complexe într-un mod clar și concis.

Un exemplu de înlănțuire

Obiectul person

```
function createPerson() {
    var firstName = "";    var lastName = "";    var age
        = 0;
    return {
        setFirstName: function(fn) {
            firstName = fn;              return this;
        },
        setLastName: function(ln) {
            lastName = ln;               return this;
        },
        setAge: function(a) {
            age = a;                  return this;
        },
        toString: function() {
            return [firstName, lastName, age].join('␣');
        }
    };
```

Un exemplu de înlănțuire

Folosirea obiectului

```
var personDescription = createPerson()
    .setFirstName("Mike")
    .setLastName("Fogus")
    .setAge(108)
    .toString();
console.log(personDescription);
```

Înlănțuirea în Underscore.js

În ceea ce privește biblioteca Underscore, funcția _.*chain* este un instrument puternic care permite să înlănțuiți mai multe operațiuni Underscore împreună. Când începeți un lanț cu _.*chain*, acesta "învelește" obiectul țintă, permițându-vă să invocați mai multe metode care modifică obiectul, folosind în final _.*value*() pentru a prelua rezultatul.

chain

```
var library = [
    { title: 'JavaScript: The Good Parts', author: '
       Douglas Crockford' },
    { title: 'You 'Dont Know JS', author: 'Kyle Simpson'
       }
];
var titles = _.chain(library)
    .pluck('title')
    .sort()
    .value();
```

Tapping

O provocare comună este depanarea acestor lanțuri, în special într-o bază de coduri complexă, unde erorile pot rămâne ascunse în aceste operațiuni. Funcția _.*tap* oferită de Underscore poate fi extrem de utilă pentru a aborda această provocare. Funcția ne permite să "intervenim" într-un lanț de operațiuni pentru a inspecta stările intermediare fără a perturba fluxul de lucru. Ea primește un obiect și o funcție și aplică funcția asupra obiectului, returnând totuși obiectul original, astfel încât să nu afecteze lanțul.

one tap

```
_.tap([1, 2, 3], function(array) {
    console.log("Starea␣ăcurent:", array);
});
```

Tapping - exemplu

Iata un exemplu mai complex de folosirea a metodei "tap".

exemplu de tap

```
var TITLE_KEY = 'titel';  // Cheia scrisa gresit
_.chain(library)
    .pluck(TITLE_KEY)
    .tap(function(results) {
        console.log("ăDup␣pluck:", results);  // ă
            înregistreaz un array de valori undefined
    })
    .sort()
    .value();
```

Tapping - exemplu

Funcția *tap* primește un obiect și o funcție și aplică funcția asupra obiectului, returnând totuși obiectul original, astfel încât să nu afecteze lanțul. În acest exemplu, _.*tap* arata că rezultatele imediat după operația pluck nu sunt așa cum ne-am aștepta, indicând o problemă la începutul lanțului. Plasând .*tap* după diferite operații, putem identifica locul unde apare problema.

two taps

```
_.chain(library)
    .pluck(TITLE_KEY)
    .tap(function(results) {
        console.log("ăDup␣pluck:", results);
    })
    .sort()
    .tap(function(sorted) {
        console.log("ăDup␣sortare:", sorted);
    })
    .value();
```

Limitările _.*chain*

De menționat că, _.*chain* nu este lazy — adică toate operațiile sunt executate chiar dacă nu se finalizeața lanțul cu _.*value*(). Acest lucru poate duce la ineficiențe dacă doar o parte din operații sunt necesare pe baza unei condiții necunoscute la momentul înlănțuirii.

chain and tap

```
_.chain(library)
    .pluck('title')
    .tap(function(titles) {
        console.log("Titluri:", titles);
    })
    .sort()
    .value();
```

Definiție

LazyChain este un model de proiectare care amână executarea metodelor pe un obiect până când este nevoie explicită de rezultat. Aceasta este realizată printr-un array de "thunks" - funcții care încapsulează o porțiune de comportament pentru a fi executată mai târziu.

Exemplu de lazy chains

```
function LazyChain(obj) {
    this._calls = [];
    this._target = obj;
}
LazyChain.prototype.invoke = function(methodName /*, args
    */) {
    var args = _.rest(arguments);
    this._calls.push(function(target) {
        var meth = target[methodName];
        return meth.apply(target, args);
    });
    return this;};
```

Evaluarea Lazychain

Metoda invoke adaugă un nou "thunk" la array-ul *_calls*. Fiecare thunk este o funcție care, când este executată, va aplica o metodă specificată unui obiect țintă.

Exemplu de utilizare Lazychain

```
var lazySort = new LazyChain([2,1,3]).invoke('sort');
```

Problema principală cu thunks este că ele necesită un obiect țintă pentru a fi executate corect. Apelarea unui thunk din array-ul *_calls* fără a furniza un obiect țintă va duce la erori.

Metoda force

Problema principală cu thunks este că ele necesită un obiect țintă pentru a fi executate corect. Direct apelarea unui thunk din array-ul *_calls* fără a furniza un obiect țintă va duce la erori. Pentru a rezolva această problemă, putem defini o metodă force care trece prin fiecare thunk în *_calls*, aplicându-l la obiectul țintă inițial și la orice rezultat intermediar

Exemplu de "limitare"

```
LazyChain.prototype.force = function() {
    return _.reduce(this._calls, function(target, thunk)
        {
         return thunk(target);
    }, this._target);
};
console.log(new LazyChain([2,1,3]).invoke('sort').force()
    );
```

Alt exemplu

Când adăugăm mai multe verigi în lanțul leneș (LazyChain), capacitatea de a gestiona operațiile într-un mod leneș devine și mai valoroasă. Lanțul pe care l-am exemplificat demonstrează cum diferite operații pot fi combinate fluent, respectând imutabilitatea și întârziind execuția până în momentul în care este necesară evaluarea finală.

Utilizarea lor

```
new LazyChain([2,1,3])
  .invoke('concat', [8,5,7,6])
  .invoke('sort')
  .invoke('join', '␣')
  .force();
```

Acest lanț adaugă mai multe elemente la array-ul inițial, sortează rezultatul, apoi îl convertește într-un șir de caractere, separând elementele prin spații.

Lazy tap

Adăugarea unei metode tap la LazyChain permite inspecția stării intermediare fără a perturba lanțul de operații. Acesta este un instrument puternic pentru depanare și pentru a verifica starea datelor în puncte critice ale procesării.

Exemplu

```
LazyChain.prototype.tap = function(fun) {
    this._calls.push(function(target) {
        fun(target);
        return target;
    });
    return this;
}
```

Lazy tap 2

Acest exemplu va sorta array-ul, va afișa starea intermediară prin alert (sau orice funcție trimisa lui tap), apoi va finaliza lanțul. Afișarea cu alert este doar ilustrativă și în practică poate fi înlocuită cu orice funcție de logare sau inspecție.

Exemplu

```
new LazyChain([2,1,3])
    .invoke('sort')
    .tap(alert)
    .force();
```

Exercitii

- Creați o funcție care returnează un rezultat ce se rezolvă după o perioadă specificată de timeout (indus artificial)
- Înlănțuiți multiple instanțe ale acestei funcții pentru a crea o secvență de mesaje de logare.
- Fiecare mesaj ar trebui să indice finalizarea unei sarcini (de exemplu, "Sarcina 1 completă", "Sarcina 2 completă")

Incercati ca sarcinile sa fie reale. Exemplu: sa realizati diferite inlantuiri pentru a calcula suma elementelor unui sir, sortarea lui sau orice procesare a acestuia.