Les proxies ES pour le fun et la gloire

Une présentation de Christophe Porteneuve à BDX I/O 2022

whoami

```
const christophe = {
  family: { wife: 'Élodie', sons: ['Maxence', 'Elliott'] },
  city: 'Paris, FR',
  company: 'Delicious Insights',
  trainings: ['Web Apps Modernes', 'Node.js', 'ES Total'],
  jsSince: 1995,
  claimsToFame: [
    'Prototype.js',
    'script.aculo.us',
    'Bien Développer pour le Web 2.0',
    'NodeSchool Paris',
    'Paris Web',
    'dotJS'
  ]
}
```

« Proxy »…

Les proxies ES nous permettent de **redéfinir la sémantique de certains aspects clés du langage**.

C'est de la **métaprogrammation**, comme les méthodes `Object.*` et les **symboles prédéfinis**.

Ça n'altère pas l'objet d'origine : **ça l'enrobe**.

```
const proxy = new Proxy(origObject, handler)
```

En gros tout l'AOP : réactivité / *data binding*, RBAC, monitoring, logs, chronométrage, délégation…

```
const chris = { age: 41.91170431211499 }
const proxy = new Proxy(christophe, {
  set(target, prop, value, recipient) {
    if (prop === 'age' && (typeof value !== 'number' || value < 0)) {
      throw new Error(`Invalid age: ${prop}.  Must be a non-negative number.`)
    }
    Reflect.set(target, prop, value, recipient)
  }
})
```

Vocabulaire

Trappe (*trap*)

Une **fonction** au nom prédéfini qui intercepte une **interaction de langage** pour la remplacer ou la personnaliser. Nous verrons qu'elle peut **déléguer** au comportement d'origine grâce à l'API `Reflect`.

Gestionnaire (*handler*)

Un **objet constitué de *traps***. Généralement mono-sujet, il implémente juste les trappes nécessaires à son besoin. Par exemple, les indices négatifs sur les tableaux ne nécessitent que `get` et `set`.

Proxy

Un objet qui **en enrobe un autre** et intercepte tout ou partie des interactions de langage potentielles avec cet objet. Celles-ci sont déduites des méthodes du gestionnaire passé lors de la création du proxy.

Trappes disponibles (1/2)

Trappe	Intercepte…
`get`	Lecture de propriété
`set`	Écriture de propriété
`has`	Opérateur `in` (test d'existence de propriété)
`ownKeys`	`Object.keys()`, `Object.getOwnPropertyNames()` et `Object.getOwnPropertySymbols()`
`getOwnProperrtyDescriptor`	`Object.getOwnPropertyDescriptor()` / `…Descriptors()`
`defineProperty`	`Object.defineProperty()`

Trappes disponibles (2/2)

Trappe	Intercepte…
`deleteProperty`	Opérateur `delete` (retrait de propriété)
`isExtensible`, `preventExtensions`	`Object.isExtensible()`, `Object.preventExtensions()`
`getPrototypeOf`, `setPrototypeOf`	`Object.getPrototypeOf()`, `Object.setPrototypeOf()`
`apply`	Appel de la fonction
`construct`	Utilisation de la fonction comme constructeur (opérateur `new`)

Accéder au comportement d'origine

L'espace de noms `Reflect` fournit des méthodes pour chaque trappe, à signature identique.

On a parfois l'impression d'une **duplication** des méthodes de `Object`, mais en fait il y a des **différences subtiles** (ex. pas de transtypage, renvoi de booléens au lieu de levées d'exceptions).

D'une façon général c'est **plus léger** que les méthodes d'`Object`.

Ça s'approche pas mal de ce que la spec d'ES appelle des ***internal slots***, tels que `[[Call]]`.

J'utilise **presque toujours** l'API `Reflect` dans mes trappes, même s'il peut sembler plus facile de recourir à `in`, `delete` ou des propriétés en accès direct. Ainsi, je m'assure de n'oublier aucun cas à la marge.

Trappes `get` et `set`

```
get(target, prop, receiver)
```

Intercepte les lectures de propriétés *(y compris inexistantes)*. Notez que c'est un préalable à *l'appel d'une méthode*, qui est une propriété comme une autre.

Par défaut : utilise l'éventuel accesseur lecteur, et renvoie `undefined` pour les propriétés absentes.

```
set(target, prop, value, receiver)
```

Intercepte les écritures de propriétés.

Par défaut : utilise l'éventuel accesseur écrivain, et si la propriété est absente, la crée à la volée (sauf si l'objet est non-extensible).

Note : `prop` est toujours de type `String` ou `Symbol` (conformément aux contraintes de noms de propriétés).

La démo LOL de `get` : tpyo

Par Mathias Bynens, ingénieur v8.

Redéfinit l'accès aux propriétés en utilisant la *distance de Levenstein* la plus faible en cas de propriété manquante 😂

```
const tpyo = require('tpyo')

const speakers = tpyo(
  ['Anaïs', 'Bérengère', 'Cécile', 'Justine', 'Manon', 'Amélie', 'Alice', 'Noémie']
)
speakers.longueur         // => 8 (et rien que pour le matin !)
speakers.flop()           // => 'Noémie' (mais pas du tout !)
speakers.splif(-4)        // => ['Justine', 'Manon', 'Amélie', 'Alice']
speakers.joint(' 😄 ')    // => 'Anaïs 😄 Bérengère 😄 Cécile'
speakers.full('of win')   // => Ben carrément (3 x 'of win')

const math = tpyo(Math)
math.skirt(9) // => 3. Ben voyons.
```

😈 Quelle est cette diablerie ?!

```
// Simplifié un poil pour tenir sur la diapo…

function tpyo(something) {
  return new Proxy(something, {
    get(target, name) {
      if (name in target) {
        return target[name]
      }

      const properties = getProperties(target)
      const closestProperty = findSimilarProperty(name, properties)
      return target[closestProperty]
    }
  })
}
```

Une démo `get` utile : client API à la volée

Tu te souviens des jours ~~heureux~~ maudits de COM, DCOM, CORBA et de la génération de proxy client ?

On peut faire beaucoup mieux désormais !

```
const api = makeRestProxy('https://jsonplaceholder.typicode.com')
await api.users()
// => [{ id: 1, name: 'Leanne Graham' … }, { id: 2, name: 'Ervin Howell', … }, …]

await api.users(1)
// => { id: 1 name: 'Leanne Graham', username: 'Bret', email: 'Sincere@april.biz', … }

// (Paie ta cohérence de ouf entre les champs factices 😄)

await api.posts(42)
// => { userId: 5, id: 42, title: 'commodi ullam…', body: 'odio fugit…' }
```

Code d'une démo `get` utile : client API à la volée

```
function makeRestProxy(baseURL) {
  return new Proxy({}, {
    get(target, prop, receiver) {
      if (!(prop in target)) {
        Reflect.defineProperty(target, prop, {
          value: makeFetchCall(baseURL, prop)
        })
      }
      return Reflect.get(target, prop, receiver)
    }
  })
}
```

```
// Considérablement simplifié pour la diapo

function makeFetchCall(baseURL, prop) {
  return async function fetch(id) {
    const path = id == null ? '' : `/${id}`
    const res = await fetch(`${baseURL}/${prop}${path}`,
      headers: {
        Accept: 'application/json',
        'Content-Type': 'application/json',
      },
    })
    return res.json()
  }
}
```

Démo `get` + `get` : indices négatifs de tableaux

Il faut dire qu'ils nous manquent (ex. Ruby), et `.at()` c'est bien mais ça ne suffit pas.

```
const names = ['Alice', 'Bob', 'Claire', 'David']
const coolNames = allowNegativeIndices(names)

coolNames[-1]
// => 'David'

coolNames[-2] = 'Clara'
names
// => ['Alice', 'Bob', 'Clara', 'David']
```

Code des indices négatifs de tableaux

```
function allowNegativeIndices(arr) {
  return new Proxy(arr, {
    get(target, prop, receiver) {
      if (prop < 0) {
        prop = target.length + Number(prop)
      }
      return Reflect.get(target, prop, receiver)
    },
    set(target, prop, value, receiver) {
      if (prop < 0) {
        prop = target.length + Number(prop)
      }
      return Reflect.set(target, prop, value, receiver)
    }
  })
}
```

Démo `get` (+ `set` ?) utile : objets défensifs

Parfois on ne veut pas `undefined` sur une propriété manquante : on veut se prendre une exception !

```
const basis = { first: 'Odile', last: 'Deray' }
const defensive = makeDefensive(basis)
defensive.first  // =>   'Odile'
defensive.middle // =>   ReferenceError: No middle property on object
```

Code des objets défensifs

```
function makeDefensive(obj) {
  return new Proxy(obj, {
    get(target, prop, receiver) {
      if (!(prop in target)) {
        throw new ReferenceError(`No ${prop} property on object`)
      }
      return Reflect.get(target, prop, receiver)
    }
  })
}
```

Un mot sur `apply` et `construct`

Pour le coup, l'objet enrobé doit être **une fonction**. Je sais, .

```
apply(target, thisArg, argumentsList)
```

- Intercepte l'appel à la fonction (opérateur `(…)`) aini que les équivalents programmatiques (`apply()` et `call()`).
- Super utile pour du *copy-on-write* qui permet d'enrober automatiquement les valeurs de retours des méthodes d'objets enrobés par un proxy.

```
construct(target, argumentsList, newTarget)
```

- Intercepte l'opérateur `new` sur une fonction. Le résultat **doit** être un objet.
- On se moque généralement de `newTarget`, sauf si on butte sur une vérification de `new.target` dans le code utilisateur.

Proxies révocables

Au lieu d'instancier un proxy par construction, on peut utiliser une *factory* qui nous permettra de révoquer à tout moment l'accès (*via* le proxy s'entend) à l'objet d'origine.

En gros, ce sont des « références périssables ».

```
const { proxy, revoke } = Proxy.revocable(target, handler)
```

Y'a clairement des super cas d'usages en termes de sécurité.

```
const { proxy, revoke } = Proxy.revocable({ first: 'John' }, {})

proxy.first // => 'John'
revoke()
proxy.first // => TypeError: Cannot perform 'get' on a proxy that has been revoked
```

Proxy révocable utile : quota d'appels

```
const fx = (...args) => args
const meteredFx = meter(fx, { max: 2 })
meteredFx('foo')        // =>   ['foo']
meteredFx('bar', 'baz') // =>   ['bar', 'baz']
meteredFx('fuu')        // =>   TypeError: Cannot perform 'apply' on a proxy that has been revoked
```

```
function meter(fx, { max }) {
  const { proxy, revoke } = Proxy.revocable(fx, {
    apply(target, thisArg, argumentsList) {
      if (--max <= 0) {
        revoke()
      }
      return Reflect.apply(target, thisArg, argumentsList)
    }
  })
  return proxy
}
```

Proxy révocable utile : TTL

```
const obj = { first: 'John' }
const moth = scheduleExpiry(obj, { ttl: 50 })

moth.first // => 'John'
setTimeout(() => console.log(moth.first), 40) // => 'John' après 40ms
setTimeout(() => console.log(moth.first), 60) // => TypeError après 60ms
```

```
function scheduleExpiry(obj, { ttl = 100 } = {}) {
  const { proxy, revoke } =  Proxy.revocable(obj, {})
  setTimeout(revoke, ttl)
  return proxy
}
```

Une dinguerie : Immer

Aide à l'immutabilité 😍 **extraordinaire** 😍 de Michel Westrate (également auteur de MobX fame). Nous permet d'écrire du **code mutatif classique** !

Fait du copy-on-write récursif à l'aide de proxies révocables récursifs sur à peu près toutes les trappes 😄

```
import produce from 'immer'

const baseState = [
  { todo: 'Apprendre React', done: true },
  { todo: 'Explorer Immer', done: false },
]

const nextState = produce(baseState, (draft) => {
  draft.push({ todo: 'Tooter à ce sujet' })
  draft[1].done = true
}) // => baseState intact, nextState correct.
```

Super leçon Egghead • *Billet d'intro (juin 2018)* • *Site officiel*

Immer et les états locaux React

Avant 😒

```
setState((prev) => ({
  ...prev,
  user: {
    ...prev.user,
    age: prev.user.age + 1,
    daysUsed: [...prev.user.daysUsed, new Date()]
  }
}))
```

Après

```
setState(produce(({ user }) => {
  user.age += 1
  user.daysUsed.push(new Date())
}))
```

Immer et les réducteurs Redux

(Au fait, Immer est pré-configuré par défaut dans Redux Toolkit)

Avant 😒

```
export function byId(state, action) {
  switch (action.type) {
    case RECEIVE_PRODUCTS:
      return {
        ...state,
        ...action.products.reduce((obj, product) => {
          obj[product.id] = product
          return obj
        }, {})
      }
    default:
      return state
  }
}
```

Après

```
export default createReducer(DEFAULT_STATE, (builder
  builder
    .addCase(RECEIVE_PRODUCTS, (draft, { products })
      for (const product of products) {
        draft[product.id] = product
      }
    })
)
```

Immer : à quoi ça ressemble sous le capot ?

```
// Extraits choisis, simplifiés pour cette diapo
export function createProxy(base, parent) {
  // …
  const {revoke, proxy} = Proxy.revocable(state, {
    get(state, prop) {
      if (prop === DRAFT_STATE) return state
      const { drafts } = state
      if (!state.modified && has(drafts, prop)) {
        return drafts[prop]
      }
      // …
      return (drafts[prop] = createProxy(value, state))
    },
    // …
  })
  // …
}
```

<Insérer avertissement obligatoire ici>

Il faut faire attention à `this` : l'enrobage par proxy l'affecte (il référence le proxy).

Les mécanismes comparant l'identité, tels que `WeakMap` (pour transpiler les champs d'instances privés par exemple), sont donc affectés.

Idem pour les **constructeurs prédéfinis** dont les méthodes utiliseraient des *internal slots* (ex. `Date#getDate()`) : ça contourne les trappes `get` / `set`.

Problème

```
const target = new Date('2022-12-02')
const proxy = new Proxy(target, {})

proxy.getDate()
// => TypeError: this is not a Date object
// (car `getDate` exploite [[NumberData]])
```

Solution de contournement

```
const target = new Date('2022-12-02')
const proxy = new Proxy(target, {
  get(target, prop, receiver) {
    const result = Reflect.get(target, prop, receiver)
    return prop === 'getDate' ? result.bind(target) : result
  }
})
proxy.getDate() // => 5 (vendredi)
```

Viens nous voir !

On est sympas.

Chez Delicious Insights, on fait des **formations qui déchirent tout**, notamment sur 100% de JS pur, React et les PWA, Node.js et Git.

(Franchement, elles envoient du bois.)

On peut aussi venir gronder ton archi / ta codebase (mais gentiment), voire réaliser tes **preuves de concept** pour toi, en mode pas jetable du tout™.

À côté de ça, tu devrais **carrément** t'abonner à notre fabuleuse chaîne YouTube, qui déborde de tutos, cours, livestreams, talks en conférences, etc. et c'est évidemment **gratuit** !

Merci

Laisse tes impressions ici ! Ça ne prend qu'un instant.

Cette présentation est sur `bit.ly/proxies-es`.

`@porteneuve@piaille.fr` / `@DelicioInsights` / YouTube

Crédits : photos de couverture par Riccardo Annandale sur Unsplash