Programmation événementielle du DOM avec JavaScript

Alexandre Niveau

GREYC — Université de Caen

En partie adapté des cours de Jean-Marc Lecarpentier et Hervé Le Crosnier

Événements

- En permanence, chaque action de l'internaute produit des événements
- On peut utiliser un événement déclencheur pour exécuter du code

  ➤ ajout d'un *capteur d'événement* (*event listener*) sur un élément

  ➤ le capteur surveille les événements qui se produisent sur l'élément, et déclenche une fonction *callback* quand les conditions sont remplies

Événements dans le DOM

- Les événements produits par les actions de l'internaute sont modélisés dans le DOM par des objets `Event` [https://developer.mozilla.org/en-US/docs/Web/API/Event]
- Pour ajouter un capteur de clics sur un élément `elem` :

```
elem.addEventListener('click', maFonction);
```

- `addEventListener` prend deux arguments (ainsi qu'un troisième optionnel, voir plus loin) :
  1. une chaîne contenant le type d'événement à capter (voir plus loin)
  2. une fonction : le *callback* à exécuter lorsque l'événement se produit

Déroulement de la captation d'un événement

- Ce qui se passe par exemple pour un clic sur un élément muni d'un capteur de clics :
  1. L'internaute clique sur l'élément
  2. Le clic génère un objet `Event`
  3. Le capteur se charge d'appeler sa fonction *callback*, *en lui passant l'objet*

Event en paramètre

Exemple d'ajout d'un capteur d'événement

```
<div id=toto>Je suis la division toto.</div>

<script> "use strict";

    function clicSurToto(ev) {
        alert("On a cliqué sur toto !");
        console.log(ev);
    }

    let toto = document.getElementById("toto");
    toto.addEventListener("click", clicSurToto);

</script>
```

Résultat [demo/eventListener.html]

Types d'événements

- Événements souris : `click`, `mouseenter` et `mouseleave` (il existe aussi `mouseover` et `mouseout`, qui sont moins pratiques — voir plus loin), `mouseup` et `mousedown`
- Événements clavier : `keydown` et `keyup` (`keypress` existe aussi mais est obsolète)
- Évenements globaux : `load`, `unload`, `scroll`, `resize`...
- Événements liés aux formulaires : `focus`, `change`, `blur`, `submit`, `input`...

Normalisation : recommandation du W3C [http://www.w3.org/TR/DOM-Level-3-Events/]

Objet Event

- Lors de l'appel du callback, le paramètre sera l'objet Event correspondant à l'événement
- L'objet Event donne de nombreuses informations par ses attributs :
  - `type` : le type d'événement (demo) [demo/eventType.html]
  - `target` : l'élément qui a déclenché l'événement
  - `currentTarget` : l'élément sur lequel le listener est attaché
  - `clientX`, `clientY`, etc. pour un `MouseEvent`
  - `key`, `code`, etc. pour un `KeyboardEvent`
- Il dispose de méthodes très utiles :
  - `preventDefault()` : empêche l'exécution de l'action par défaut de l'événement (démo [demo/preventDefault.html] , détails sur MDN [https://developer.mozilla.org/en-US/docs/Web/API/Event/preventDefault])
  - `stopPropagation()` : arrête la propagation de l'événement (démo [demo/stopPropagation.html])

Flux de l'événement (1)

- Une action de l'internaute affecte un élément, mais également ses parents
- Par exemple, un clic sur un span dans un div affecte le span et le div : il génère un événement sur deux éléments
- Lequel a lieu en premier ? On peut considérer que c'est le parent qui a reçu le clic et l'a transmis à son enfant, ou l'inverse

Flux de l'événement (2)

- Explications du W3C [http://www.w3.org/TR/DOM-Level-3-Events/#event-flow]
  - Événement « à la capture » : a lieu d'abord pour les parents
  - Événement « à la remontée » (*bubbling*) : a lieu d'abord pour les enfants
- Par défaut, `addEventListener` détecte les événements à la remontée, mais elle a un troisième paramètre optionnel

  ➤ booléen qui permet de détecter seulement les événements à la capture (si vrai) ou à la remontée (si faux)

Démo [demo/eventCaptureBubbling.html]

Autres façons de capter un événement

- Il y a deux autres manières d'associer un capteur d'événement à un nœud de l'arbre DOM
  - dans le HTML, en mettant du code dans les attributs `onmouseover`, `onload`, `onclick`, etc.
  - dans le JS, en mettant une fonction dans les attributs HTML susmentionnés
- Je donne quelques explications dessus, mais il ne faudra pas les utiliser dans le cadre de ce cours !

Capter un événement dans le code HTML

- Chaque élément HTML a un attribut par événement, `on`*`événement`* (`onmouseover`, `onload`, `onclick`) où il est possible de mettre du JavaScript directement
- Considéré comme une mauvaise pratique : mélange sémantique et comportement [https://en.wikipedia.org/wiki/Unobtrusive_JavaScript#Separation_of_behavior_from_markup]. **À ne pas utiliser** en général ; **interdit dans le cadre de ce cours**
- Exemple : cliquer sur cet élément déclenche une action sur l'item suivant
- Cet élément possède l'identifiant *toto*

```
<li onclick="document.getElementById('toto').style.color='green';
             document.getElementById('toto').style.fontSize='larger';">
Exemple : cliquer sur cet élément déclenche une action sur l'item suivant
```

```
</li>
<li id="toto">Cet élément possède l'identifiant toto</li>
```

Utilisation des attributs d'événement dans le JS

- Un peu plus propre : remplir les attributs `onclick` et consorts dans le script JS
- Il suffit de mettre une fonction dans l'attribut, et elle sera exécutée si l'événement se produit sur l'élément

```
<div id=toto>
  Je suis la division toto.
</div>
<script>
function clicSurToto() {
  alert("On a cliqué sur toto !");
}
toto = document.getElementById("toto");
toto.onclick = clicSurToto;
</script>
```

- Inconvénient : une seule fonction par événement, donc on ne peut pas gérer différents comportements indépendamment
- **Interdit dans le cadre de ce cours**

Avantages de `EventListener`

- Il y a quelques années, il restait intéressant d'utiliser `onclick` et consorts pour la portabilité, mais ce n'est plus le cas, `addEventListener` est supporté partout [http://caniuse.com/#feat=addeventlistener]
- Avantages de `addEventListener` :
  - on peut ajouter plusieurs listeners pour un événement sur un même élément
  - on peut retirer des listeners
  - contrôle plus fin du déclenchement de l'événement (capture ou remontée)
- Unique inconvénient : c'est un peu plus long à écrire

Événements `load` et `DOMContentLoaded`

- `load` est lancé lorsque la page est entièrement chargée, y compris toutes les dépendances (notamment les images)
- Utilisé autrefois pour retarder l'exécution du JS jusqu'à après que la page soit chargée
- Mais en général seul le chargement du DOM nous intéresse
- Événement `DOMContentLoaded` de HTML5 est plus adapté, et fonctionne partout (sauf IE < 9…)
- Permet notamment de placer dans l'en-tête du document un script interne

(*inline*) : Démo [demo/domcontentloaded.html]

- *Attention* cependant : le HTML qui suit le script ne sera parsé et donc affiché qu'après l'exécution du script, et donc après son téléchargement si nécessaire.

  ➤ avec un script externe, cette technique peut donc ralentir l'affichage de la page, potentiellement très fortement si le script est sur un autre serveur et que ce serveur rencontre des problèmes

  ➤ Pour les script inclus, utiliser l'attribut `defer` de l'élément `script` (voir cours précédent) [../js/#ou-placer-script]

[#mouseenter]
Événements `mouseover` et `mouseenter`

- `mouseover` est lancé lorsque la souris entre sur la surface d'un élément, `mouseout` quand elle en sort
- Problèmes potentiels :
  - ces événements *remontent* et sont donc reçus par tous les parents
  - entrer dans la surface d'un élément fils envoie un `mouseout` sur le parent et un `mouseover` sur le fils, et inversement
- Démo (première section) [demo/mouseover.html#demo-mouseover]
- Solution : événements `mouseenter` et `mouseleave`, qui ne remontent pas et ne sont pas lancés quand la souris rentre et sort d'un descendant
- Fonctionnent comme `:hover` en CSS
- Démo (deuxième section) [demo/mouseover.html#demo-mouseenter]

Événements clavier

- Les événements clavier ne sont pas implémentés de la même façon sur tous les navigateurs et OS, et les spécs sont encore assez instables
- La spéc actuelle ne définit que deux événements, `keydown` et `keyup`
- Ils ont chacun deux attributs intéressants :
  - `event.key` contient une chaîne qui représente le *caractère* auquel correspond la touche pressée (ou relâchée).
    Ex.: `"a"`, `"A"` (les majuscules et minuscules sont distinctes), `"é"`, `"["`, `Control`, `"ArrowLeft"`...
  - `event.code` contient une chaîne qui représente la touche physique que l'internaute a pressée (ou relâchée), indépendamment de la disposition du clavier (qwerty, azerty, bépo...).
    Ex.: `"KeyQ"`, `"Digit3"`, `"BracketLeft"`, `"ControlRight"`...

  Démo des différentes propriétés
  [demo/keyboard.html]

- Malheureusement, `event.key` n'est pas encore supporté partout [http://caniuse.com/#feat=keyboardevent-key], et `event.code` encore moins [http://caniuse.com/#feat=keyboardevent-code]

- Quand `event.key` n'est pas supporté, si on veut simplement tester l'appui sur une touche alphanumérique (sans accent), on peut utiliser `String.fromCharCode(event.which || event.keyCode)`, qui devrait marcher à peu près partout… (Quelques détails) [http://stackoverflow.com/a/41656511]

Démo event.key et event.code [demo/keyboard2.html]

- Autre possibilité : on peut utiliser un polyfill, c'est-à-dire une librairie JS qui ajoute au navigateur l'implémentation de `event.key` — celui-ci par exemple [https://github.com/cvan/keyboardevent-key-polyfill]

Spécifications et normes

- Normes du W3C sur le DOM [http://www.w3.org/TR/dom]

Références et guides

- Document Object Model [https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model] sur MDN
- Web API Interfaces [https://developer.mozilla.org/en-US/docs/Web/API] sur MDN
- Event Reference [https://developer.mozilla.org/en-US/docs/Web/Events] sur MDN