La gestion des événements en JavaScript (source AlsaCréations)

Introduction

JavaScript est un langage événementiel : le développeur a un contrôle limité sur le flux d'exécution du code, qui est déterminé principalement par les interactions avec l'environnement (activation d'un lien, mouvement de la souris, chargement du contenu du document, …).

La gestion des événements est donc un sujet essentiel dans le cadre de ce langage. Elle reste pourtant assez mal comprise, en partie à cause des lacunes et erreurs d'implémentation des différents navigateurs.

Cet article présente les trois grandes familles d'interfaces qui sont aujourd'hui à notre disposition :

- Le DOM niveau 0, standard *de facto* hérité de Netscape ; il s'agit de l'interface la plus largement supportée mais aussi la moins puissante.
- Le modèle d'événement DOM niveau 2 [en] (voir également la traduction française), spécifié par le W3C mais malheureusement non supporté par Internet Explorer (du moins pour l'instant, c'est-à-dire jusqu'à la version 8 au moins).
- L'interface spécifique à Internet Explorer, qui fournit un sous-ensemble des fonctionnalités du DOM niveau 2, mais avec des dénominations différentes.

Les concepts présentés dans la suite de l'article sont communs à ces trois familles, sauf indication contraire.

L'objet `Event`

Un événement est un changement d'état de l'environnement qui peut être intercepté par le code JavaScript. Dans certains cas, l'action implicite (par défaut) correspondante est alors annulable : il est par exemple possible d'empêcher que l'activation d'un lien entraîne la navigation vers l'URL associée.

Ce changement d'état peut être provoqué par l'utilisateur (pression d'une touche, ...), par le document (chargement d'une image, ...), ou même par le développeur.

Au niveau du code JavaScript, un objet `Event` est mis à disposition pour décrire l'événement. Cet objet, qui va se propager dans l'arbre DOM selon le flux d'événement présenté dans la suite de cet article, contient des données dont certaines sont spécifiques au type d'événement (par exemple, le code de la touche pressée pour un événement de type `keydown`). Nous nous intéresserons ici aux données communes à tous les types d'événements, et seulement aux plus utilisées.

Ces données sont disponibles sous la forme de propriétés (ou méthodes) de l'objet `Event` :

Propriété target

Il s'agit de la *cible* de l'événement. C'est le nœud de l'arbre DOM concerné par le changement d'état associé à l'événement. Dans le cas d'un événement de souris par exemple, c'est l'élément le plus profond de l'arbre au-dessus duquel se trouve la souris.

Cette propriété est pour l'instant disponible au sein d'Internet Explorer sous un autre nom : *`srcElement`*. On utilise donc en général le code suivant pour y accéder :

```
// event est l'objet Event
var target = event.target || event.srcElement;
```

Propriété type

Comme son nom l'indique, c'est le type d'événement (`"focus"`, `"load"`, ...).

Méthode stopPropagation

Cette méthode permet d'arrêter la propagation de l'événement dans l'arbre DOM après le nœud sur lequel il se trouve.

Son équivalent sous Internet Explorer est la propriété *`cancelBubble`*, ce qui peut être esquivé grâce au code suivant :

```
// event est l'objet Event
if (event.stopPropagation) { // test si la méthode
                             //   existe
  event.stopPropagation();
}
event.cancelBubble = true;
```

On utilise en général cette méthode en parallèle avec *`preventDefault`*.

Méthode preventDefault

Pour les types d'événements qui l'autorisent, il est possible grâce à cette méthode d'annuler l'action implicite correspondante. Par exemple, l'action implicite associée à un événement de type *`submit`* est l'envoi au serveur du formulaire concerné.

Pour un gestionnaire d'événement DOM-0 (présenté à la fin de cet article), on se contente en général du code « `return false;` », car, bien que non défini dans la spécification, il est beaucoup plus largement supporté.

Sinon, il est une fois encore nécessaire de prendre en compte le cas Internet Explorer et sa propriété *`returnValue`* :

```
// event est l'objet Event
if (event.preventDefault) { // test de l'existence
  event.preventDefault();
```

```
        }
        event.returnValue = false;
```

Propriété currentTarget

Il s'agit du nœud sur lequel l'événement se trouve actuellement dans le cadre du flux d'événement. Cette propriété assez utile n'est malheureusement pas supportée par Internet Explorer.

Il est à noter que, contrairement aux données précédentes, celle-ci change au cours de la progression de l'événement dans l'arbre DOM.

Le flux d'événement

Une fois l'objet `Event` créé, il se propage dans l'arbre DOM selon un flux bien précis déterminé par sa cible :

- Phase de *capture* : l'événement se propage de la racine du document (incluse) à la cible (exclue).
- L'événement atteint la *cible*.
- Phase de *bouillonnement* (*bubbling*) : l'événement se propage dans le sens inverse : de la cible (exclue) à la racine du document (incluse).

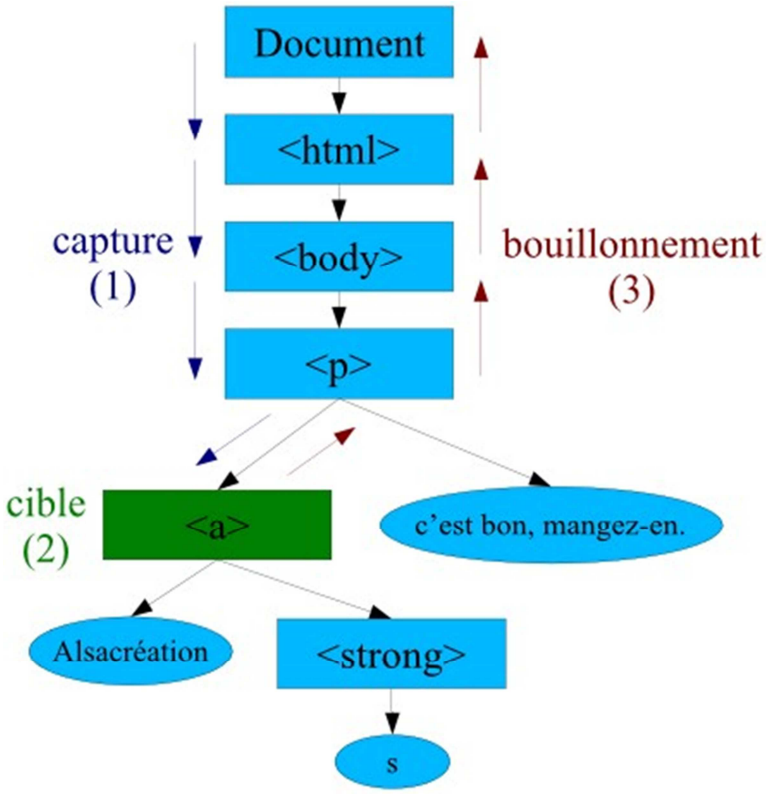
Par exemple, pour la page XHTML suivante :

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns="http://www.w3.org/1999/xhtml"
                        xml:lang="fr" lang="fr">
<head>
  <meta http-equiv="Content-Type"
          content="text/html; charset=UTF-8" />
  <title>Publicité</title>
</head>
<body>
  <p>
    <a href="http://www.alsacreations.com/">
      Alsacréation<strong>s</strong></a>
    c'est bon, mangez-en.
  </p>
</body>
</html>
```

Si l'utilisateur active le lien, voici les différentes étapes qui s'enchaînent :

- Un événement de type `click` est créé. La cible est déterminée ; il peut s'agir de a comme de `strong` (dans le cas où l'utilisateur utilise une souris et a cliqué sur la zone correspondant à la lettre « s »). Supposons que la cible est l'élément a.
- **Phase 1 (*capture*) :** l'événement se propage du nœud `Document` (inclus) au nœud `a` (exclu).
- **Phase 2 (*cible*) :** l'événement atteint le nœud `a`.
- **Phase 3 (*bouillonnement*) :** l'événement se propage du nœud `a` (exclu) au nœud *`document`* (inclus).



Cependant, la phase de bouillonnement est optionnelle. Son existence est déterminée par le type d'événement : par exemple, les événements de type *`load`* ne bouillonnent pas.

De plus, comme expliqué dans la partie précédente, le flux d'événement peut être interrompu dans le code JavaScript grâce à la méthode *`stopPropagation`*.

Les gestionnaires d'événement

Pour pouvoir intercepter un événement pendant sa propagation dans l'arbre DOM, il faut utiliser un gestionnaire d'événement.

En JavaScript, il s'agit d'une fonction que l'on attache à un nœud, en précisant :

- un type d'événement,
- une phase : capture, ou bouillonnement (voir la partie sur le flux d'événement), sachant que l'on inclut la cible dans la phase de bouillonnement et pas dans la phase de capture.

Cependant, Internet Explorer ne supporte pour l'instant que la phase de bouillonnement (à part dans certains cas que nous ne traiterons pas ici), ce qui rend la phase de capture très peu utilisée. De plus, certains autres navigateurs (notamment Firefox) incluent par erreur la cible dans la phase de capture.

Un gestionnaire d'événement est appelé lorsqu'un événement atteint le nœud auquel il a été attaché si le type de l'événement et la phase actuelle du flux d'événement correspondent à ceux précisés lors de son ajout.

Les gestionnaires d'événement DOM-2

Le modèle d'événement DOM niveau 2 définit deux méthodes, `addEventListener` et `removeEventListener`, qui permettent d'attacher ou de détacher un gestionnaire d'événement d'un nœud :

```
void addEventListener(
  in DOMString type,
  in EventListener listener,
  in boolean useCapture
);

void removeEventListener(
  in DOMString type,
  in EventListener listener,
  in boolean useCapture
);
```

- *`type`* est le type d'événement,
- *`listener`* le gestionnaire d'événement (en JavaScript, une fonction),
- et *`useCapture`* un booléen : `true` pour la phase de capture, ou `false` pour la phase de bouillonnement et la cible. On utilise quasiment toujours la valeur `false`.

Voici un exemple d'utilisation de `addEventListener` :

```
function envoiForm(event) {
  if (this.elements.adresse.value === "") {
    alert("L'adresse est vide.");
    event.preventDefault();
  }
}

var formulaire = document.getElementById("coordonnees");
formulaire.addEventListener("submit", envoiForm, false);
```

On suppose qu'il existe au moment de l'exécution de ce code un formulaire d'id « `coordonnees` » qui contient un champ dont le nom (`name`) est « `adresse` ». Lorsque ce formulaire sera envoyé, l'événement de type *`submit`* sera intercepté par le gestionnaire d'événement associé à la fonction *`envoiForm`*. Si le champ « `adresse` » est vide, un dialogue d'alerte sera affiché et l'envoi du formulaire sera annulé (*`preventDefault`*).

La fonction qui fait office de gestionnaire d'événement a accès à l'objet *`Event`* par l'intermédiaire de son premier paramètre (ici *`event`*). De plus, le nœud auquel le gestionnaire d'événement a été ajouté (c'est-à-dire *`event.currentTarget`*) est associé à *`this`* (comportement non spécifié par le modèle d'événement DOM niveau 2, mais très homogène sur les différentes implémentations), ce dont on se sert ici pour récupérer le champ « `adresse` » (*`this.elements.adresse`*).

Pour détacher un gestionnaire d'événement, il faut passer à la méthode *`removeEventListener`* les mêmes paramètres que ceux qui ont été passés à *`addEventListener`*. Par exemple :

```
formulaire.removeEventListener("submit", envoiForm, false);
```

Ces deux méthodes permettent donc de manipuler plusieurs gestionnaires d'événement de même type sur un même nœud. Comme précisé auparavant, Internet Explorer ne supporte que son propre modèle d'événement, qui met à notre disposition deux méthodes similaires.

Les gestionnaires d'événement Internet Explorer

Les méthodes *`attachEvent`* et *`detachEvent`* fonctionnent de la même façon que *`addEventListener`* et *`removeEventListener`*, à quelques différences près :

- Il n'y a pas de troisième paramètre (seules les phases de cible et de bouillonnement sont supportées).
- Le type d'événement (premier paramètre) est préfixé par `"on"` (par exemple : `"onload"`).
- Il n'y a aucun équivalent à `this` ou `event.currentTarget`, ce qui est peut être parfois très gênant (la cible n'étant pas systématiquement le même nœud que celui auquel on a attaché le gestionnaire d'événement).

En général, on utilise un code similaire à celui qui suit pour manipuler des gestionnaires d'événement à la fois sous Internet Explorer et sous les navigateurs qui supportent les gestionnaires d'événement DOM-2 :

```
function envoiForm(event) {
  var target = event.target || event.srcElement;
  if (target.elements.adresse.value === "") {
    alert("L'adresse est vide.");
    if (event.preventDefault) {
      event.preventDefault();
    }
    event.returnValue = false;
  }
}

var formulaire = document.getElementById("coordonnees");
if (formulaire.addEventListener) {
  formulaire.addEventListener("submit", envoiForm, false);
```

```
} else if (formulaire.attachEvent) {
  formulaire.attachEvent("onsubmit", envoiForm);
}
```

Dans cet exemple, on a pu remplacer l'utilisation de *`this`* par *`event.target`* (et *`event.srcElement`* pour Internet Explorer) car pour un événement de type *`submit`*, la cible est toujours le formulaire.

Pour écrire du code compatible à la fois pour les navigateurs qui supportent le modèle d'événement DOM niveau 2 et pour Internet Explorer, il faut donc tester l'existence des méthodes d'ajout ou de suppression de gestionnaires d'événement (*`addEventListener`*, *`attachEvent`*, ...) et des différentes propriétés de l'objet *`Event`* (*`target`*, *`srcElement`*, ...).

Cependant, dans la plupart des cas, les gestionnaires d'événements DOM-0 permettent de simplifier ce problème.

Les gestionnaires d'événement DOM-0

Il s'agit en fait d'un standard *de facto*, hérité de Netscape, mais qui est encore aujourd'hui la façon la plus fiable de gérer les événements en JavaScript.

Il ne supporte que la phase de bouillonnement (et la cible), et ne permet pas d'ajouter plusieurs gestionnaires d'événement de même type à un même nœud.

Pour ajouter un gestionnaire d'événement à un élément, il suffit de définir une fonction comme propriété de l'objet JavaScript correspondant. Par exemple :

```
function envoiForm(event) {
  if (this.elements.adresse.value === "") {
    alert("L'adresse est vide.");
    return false;
  }
}

var formulaire = document.getElementById("coordonnees");
formulaire.onsubmit = envoiForm;
```

Internet Explorer supporte dans ce cas l'accès à l'élément auquel on a ajouté le gestionnaire d'événement par l'intermédiaire de *`this`*. Par contre, il ne passe par l'objet `Event` comme paramètre de la fonction, mais l'associe à la variable globale *`event`*. Pour le récupérer sur tous les navigateurs, il faut donc écrire quelque chose de similaire à :

```
function envoiForm(event) {
  event = event || window.event;
  // event est l'objet Event
}
```

Une autre façon d'utiliser les gestionnaires d'événements DOM-0 (et sans doute la plus utilisée) est de passer par les attributs (X)HTML correspondants (à écrire totalement en minuscules en XHTML) :

```
<form id="coordonnees"
  onsubmit="alert(event.type);"
  action="traiter-coordonnees" method="post">
  [...]
```

Notez qu'il ne faut pas préfixer le contenu de l'attribut par « `javascript:` ».

Cette solution est en fait équivalente à celle de l'attribut JavaScript. Le navigateur va créer une fonction anonyme dont le corps correspondra au contenu de l'attribut HTML :

```
formulaire.onsubmit = function(event) {
  alert(event.type);
};
```

Quant à Internet Explorer, il fera de même mais sans spécifier le paramètre `event`. On pourra donc accéder à l'objet `Event` en écrivant « `event` » : pour Internet Explorer, cela fera référence à la variable globale, et pour les autres navigateurs au paramètre de la fonction anonyme.

L'utilisation des gestionnaires d'événements par l'intermédiaire des attributs HTML est en général à éviter, car elle va à l'encontre de la sépararation du comportement et de la structure.

Conclusion

Les gestionnaires d'événement DOM-0 représentent encore le moyen le plus facile de manipuler les événements en JavaScript, notamment parce qu'ils ne nécessitent pas plusieurs branches de code en ce qui concerne l'ajout ou la suppression des gestionnaires d'événement, et parce qu'ils permettent d'accéder à l'élément auquel ils ont été ajoutés au sein d'Internet Explorer (par l'intermédiaire de `this`).

De plus, ils offrent un comportement très cohérent sur les différents navigateurs.

Néanmoins, il n'est pas toujours possible de les utiliser, car ils posent un problème d'écrasement (voir l'article sur les modules JavaScript).

Il est donc important de bien connaître les avantages et inconvénients de chacune des méthodes (DOM-0 ou DOM-2/IE) pour savoir quand choisir l'une ou l'autre.

Ressources

- http://www.quirksmode.org/js/introevents.html
- http://www.w3.org/TR/DOM-Level-2-Events/
- http://www.yoyodesign.org/doc/w3c/preface.php?id=dom2-events#lex_fr_en