8th February 2020

Event delegation

Capturing and bubbling allow us to implement one of most powerful event handling patterns called *event delegation*.

The idea is that if we have a lot of elements handled in a similar way, then instead of assigning a handler to each of them – we put a single handler on their common ancestor.

In the handler we get `event.target`, see where the event actually happened and handle it.

Let's see an example – the Ba-Gua diagram reflecting the ancient Chinese philosophy.

Here it is:

***Bagua* Chart: Direction, Element, Color, Meaning**

Northwest Metal Silver Elders	**North** Water Blue Change	**Northeast** Earth Yellow Direction
West Metal Gold Youth	**Center** All Purple Harmony	**East** Wood Blue Future
Southwest Earth Brown Tranquility	**South** Fire Orange Fame	**Southeast** Wood Green Romance

The HTML is like this:

```
<table>
  <tr>
    <th colspan="3"><em>Bagua</em> Chart: Direction, Element, Color, Meaning<
  </tr>
  <tr>
    <td class="nw"><strong>Northwest</strong><br>Metal<br>Silver<br>Elders</t
    <td class="n">...</td>
    <td class="ne">...</td>
  </tr>
  <tr>...2 more lines of this kind...</tr>
```

```
  <tr>...2 more lines of this kind...</tr>
</table>
```

The table has 9 cells, but there could be 99 or 9999, doesn't matter.

Our task is to highlight a cell `<td>` on click.

Instead of assign an `onclick` handler to each `<td>` (can be many) – we'll setup the "catch-all" handler on `<table>` element.

It will use `event.target` to get the clicked element and highlight it.

The code:

```
let selectedTd;

table.onclick = function(event) {
  let target = event.target; // where was the click?

  if (target.tagName != 'TD') return; // not on TD? Then we're not interested

  highlight(target); // highlight it
};

function highlight(td) {
  if (selectedTd) { // remove the existing highlight if any
    selectedTd.classList.remove('highlight');
  }
  selectedTd = td;
  selectedTd.classList.add('highlight'); // highlight the new td
}
```

Such a code doesn't care how many cells there are in the table. We can add/remove `<td>` dynamically at any time and the highlighting will still work.

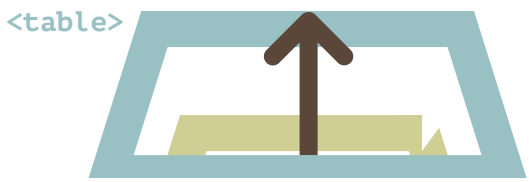
Still, there's a drawback.

The click may occur not on the `<td>`, but inside it.

In our case if we take a look inside the HTML, we can see nested tags inside `<td>`, like `<strong>`:

```
<td>
  <strong>Northwest</strong>
  ...
</td>
```

Naturally, if a click happens on that `<strong>` then it becomes the value of `event.target`.



In the handler `table.onclick` we should take such `event.target` and find out whether the click was inside `<td>` or not.

Here's the improved code:

```
table.onclick = function(event) {
  let td = event.target.closest('td'); // (1)

  if (!td) return; // (2)

  if (!table.contains(td)) return; // (3)

  highlight(td); // (4)
};
```

Explanations:

1. The method `elem.closest(selector)` returns the nearest ancestor that matches the selector. In our case we look for `<td>` on the way up from the source element.
2. If `event.target` is not inside any `<td>`, then the call returns `null`, and we don't have to do anything.
3. In case of nested tables, `event.target` may be a `<td>` lying outside of the current table. So we check if that's actually *our table's* `<td>`.
4. And, if it's so, then highlight it.

As the result, we have a fast, efficient highlighting code, that doesn't care about the total number of `<td>` in the table.

Delegation example: actions in markup

There are other uses for event delegation.

Let's say, we want to make a menu with buttons "Save", "Load", "Search" and so on. And there's an object with methods `save`, `load`, `search` ... How to match them?

The first idea may be to assign a separate handler to each button. But there's a more elegant solution. We can add a handler for the whole menu and `data-action` attributes for buttons that has the method to call:

```
<button data-action="save">Click to Save</button>
```

The handler reads the attribute and executes the method. Take a look at the working example:

```
<div id="menu">
  <button data-action="save">Save</button>
  <button data-action="load">Load</button>
```

```
  <button data-action="search">Search</button>
</div>

<script>
  class Menu {
    constructor(elem) {
      this._elem = elem;
      elem.onclick = this.onClick.bind(this); // (*)
    }

    save() {
      alert('saving');
    }

    load() {
      alert('loading');
    }

    search() {
      alert('searching');
    }

    onClick(event) {
      let action = event.target.dataset.action;
      if (action) {
        this[action]();
      }
    };
  }

  new Menu(menu);
</script>
```

Save Load Search

Please note that `this.onClick` is bound to `this` in `(*)`. That's important, because otherwise `this` inside it would reference the DOM element (`elem`), not the `Menu` object, and `this[action]` would not be what we need.

So, what advantages does delegation give us here?

- We don't need to write the code to assign a handler to each button. Just make a method and put it in the markup.
- The HTML structure is flexible, we can add/remove buttons at any time.

We could also use classes `.action-save`, `.action-load`, but an attribute `data-action` is better semantically. And we can use it in CSS rules too.

The "behavior" pattern

We can also use event delegation to add “behaviors” to elements *declaratively*, with special attributes and classes.

The pattern has two parts:

1. We add a custom attribute to an element that describes its behavior.
2. A document-wide handler tracks events, and if an event happens on an attributed element – performs the action.

Behavior: Counter

For instance, here the attribute `data-counter` adds a behavior: “increase value on click” to buttons:

```
Counter: <input type="button" value="1" data-counter>
One more counter: <input type="button" value="2" data-counter>

<script>
  document.addEventListener('click', function(event) {

    if (event.target.dataset.counter != undefined) { // if the attribute exis
      event.target.value++;
    }

  });
</script>
```

Counter: 1 One more counter: 2

If we click a button – its value is increased. Not buttons, but the general approach is important here.

There can be as many attributes with `data-counter` as we want. We can add new ones to HTML at any moment. Using the event delegation we “extended” HTML, added an attribute that describes a new behavior.

For document-level handlers – always `addEventListener`

When we assign an event handler to the `document` object, we should always use `addEventListener`, not `document.on<event>`, because the latter will cause conflicts: new handlers overwrite old ones.

For real projects it’s normal that there are many handlers on `document` set by different parts of the code.

Behavior: Toggler

One more example of behavior. A click on an element with the attribute `data-toggle-id` will show/hide the element with the given `id`:

```
<button data-toggle-id="subscribe-mail">
  Show the subscription form
</button>

<form id="subscribe-mail" hidden>
```

```
  Your mail: <input type="email">
</form>

<script>
  document.addEventListener('click', function(event) {
    let id = event.target.dataset.toggleId;
    if (!id) return;

    let elem = document.getElementById(id);

    elem.hidden = !elem.hidden;
  });
</script>
```

Show the subscription form

Let's note once again what we did. Now, to add toggling functionality to an element – there's no need to know JavaScript, just use the attribute `data-toggle-id`.

That may become really convenient – no need to write JavaScript for every such element. Just use the behavior. The document-level handler makes it work for any element of the page.

We can combine multiple behaviors on a single element as well.

The "behavior" pattern can be an alternative to mini-fragments of JavaScript.

Summary

Event delegation is really cool! It's one of the most helpful patterns for DOM events.

It's often used to add the same handling for many similar elements, but not only for that.

The algorithm:

1. Put a single handler on the container.
2. In the handler – check the source element `event.target`.
3. If the event happened inside an element that interests us, then handle the event.

Benefits:

- Simplifies initialization and saves memory: no need to add many handlers.
- Less code: when adding or removing elements, no need to add/remove handlers.
- DOM modifications: we can mass add/remove elements with `innerHTML` and the like.

The delegation has its limitations of course:

- First, the event must be bubbling. Some events do not bubble. Also, low-level handlers should not use `event.stopPropagation()`.
- Second, the delegation may add CPU load, because the container-level handler reacts on events in any place of the container, no matter whether they interest us or not. But usually the load is negligible, so we don't take it into account.

Tasks

Hide messages with delegation

importance: 5

There's a list of messages with removal buttons `[x]`. Make the buttons work.

Like this:

Horse [x]

The horse is one of two extant subspecies of Equus ferus. It is an odd-toed ungulate mammal belonging to the taxonomic family Equidae. The horse has evolved over the past 45 to 55 million years from a small multi-toed creature, Eohippus, into the large, single-toed animal of today.

Donkey [x]

The donkey or ass (Equus africanus asinus) is a domesticated member of the horse family, Equidae. The wild ancestor of the donkey is the African wild ass, E. africanus. The donkey has been used as a working animal for at least 5000 years.

Cat [x]

The domestic cat (Latin: Felis catus) is a small, typically furry, carnivorous mammal. They are often called house cats when kept as indoor pets or simply cats when there is no need to distinguish them from other felids and felines. Cats are often valued by humans for companionship and for their ability to hunt vermin.

P.S. Should be only one event listener on the container, use event delegation.

Open a sandbox for the task.

solution

Tree menu

importance: 5

Create a tree that shows/hides node children on click:

- Animals
  - Mammals
    - Cows
    - Donkeys
    - Dogs
    - Tigers
  - Other
    - Snakes
    - Birds
    - Lizards
- Fishes
  - Aquarium
    - Guppy
    - Angelfish
  - Sea
    - Sea trout

Requirements:

- Only one event handler (use delegation)
- A click outside the node title (on an empty space) should not do anything.

Open a sandbox for the task.

solution

Sortable table

importance: 4

Make the table sortable: clicks on `<th>` elements should sort it by corresponding column.

Each `<th>` has the type in the attribute, like this:

```
<table id="grid">
  <thead>
    <tr>
      <th data-type="number">Age</th>
      <th data-type="string">Name</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>5</td>
      <td>John</td>
    </tr>
```

```
    <tr>
      <td>10</td>
      <td>Ann</td>
    </tr>
    ...
  </tbody>
</table>
```

In the example above the first column has numbers, and the second one – strings. The sorting function should handle sort according to the type.

Only `"string"` and `"number"` types should be supported.

The working example:

Age	Name
5	John
2	Pete
12	Ann
9	Eugene
1	Ilya

P.S. The table can be big, with any number of rows and columns.

Open a sandbox for the task.

solution

Tooltip behavior

importance: 5

Create JS-code for the tooltip behavior.

When a mouse comes over an element with `data-tooltip`, the tooltip should appear over it, and when it's gone then hide.

An example of annotated HTML:

```
<button data-tooltip="the tooltip is longer than the element">Short button</b
<button data-tooltip="HTML<br>tooltip">One more button</button>
```

Should work like this:

LaLaLa LaLaLa LaLaLa LaLaLa LaLaLa LaLaLa LaLaLa LaLaLa LaLaLa

LaLaLa LaLaLa LaLaLa LaLaLa LaLaLa LaLaLa LaLaLa LaLaLa LaLaLa

Short button One more button

Scroll the page to make buttons appear on the top, check if the tooltips show up correctly.

In this task we assume that all elements with `data-tooltip` have only text inside. No nested tags (yet).

Details:

- The distance between the element and the tooltip should be `5px` .
- The tooltip should be centered relative to the element, if possible.
- The tooltip should not cross window edges. Normally it should be above the element, but if the element is at the page top and there's no space for the tooltip, then below it.
- The tooltip content is given in the `data-tooltip` attribute. It can be arbitrary HTML.

You'll need two events here:

- `mouseover` triggers when a pointer comes over an element.
- `mouseout` triggers when a pointer leaves an element.

Please use event delegation: set up two handlers on `document` to track all "overs" and "outs" from elements with `data-tooltip` and manage tooltips from there.

After the behavior is implemented, even people unfamiliar with JavaScript can add annotated elements.

P.S. Only one tooltip may show up at a time.

Open a sandbox for the task.

solution

Previous lesson Next lesson

Share

Tutorial map

Comments

- If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.
- If you can't understand something in the article – please elaborate.
- To insert a few words of code, use the `<code>` tag, for several lines – use `<pre>`, for more than 10 lines – use a sandbox (plnkr, JSBin, codepen…)