15th July 2019

Shadow DOM and events

The idea behind shadow tree is to encapsulate internal implementation details of a component.

Let's say, a click event happens inside a shadow DOM of `<user-card>` component. But scripts in the main document have no idea about the shadow DOM internals, especially if the component comes from a 3rd-party library.

So, to keep the details encapsulated, the browser *retargets* the event.

Events that happen in shadow DOM have the host element as the target, when caught outside of the component.

Here's a simple example:

```
<user-card></user-card>

<script>
customElements.define('user-card', class extends HTMLElement {
  connectedCallback() {
    this.attachShadow({mode: 'open'});
    this.shadowRoot.innerHTML = `<p>
      <button>Click me</button>
    </p>`;
    this.shadowRoot.firstElementChild.onclick =
      e => alert("Inner target: " + e.target.tagName);
  }
});

document.onclick =
  e => alert("Outer target: " + e.target.tagName);
</script>
```

Click me

If you click on the button, the messages are:

1. Inner target: `BUTTON` – internal event handler gets the correct target, the element inside shadow DOM.
2. Outer target: `USER-CARD` – document event handler gets shadow host as the target.

Event retargeting is a great thing to have, because the outer document doesn't have to know about component internals. From its point of view, the event happened on `<user-card>`.

Retargeting does not occur if the event occurs on a slotted element, that physically lives in the light DOM.

For example, if a user clicks on `<span slot="username">` in the example below, the event target is exactly this `span` element, for both shadow and light handlers:

```html
<user-card id="userCard">
  <span slot="username">John Smith</span>
</user-card>

<script>
customElements.define('user-card', class extends HTMLElement {
  connectedCallback() {
    this.attachShadow({mode: 'open'});
    this.shadowRoot.innerHTML = `<div>
      <b>Name:</b> <slot name="username"></slot>
    </div>`;

    this.shadowRoot.firstElementChild.onclick =
      e => alert("Inner target: " + e.target.tagName);
  }
});

userCard.onclick = e => alert(`Outer target: ${e.target.tagName}`);
</script>
```

Name: John Smith

If a click happens on `"John Smith"`, for both inner and outer handlers the target is `<span slot="username">`. That's an element from the light DOM, so no retargeting.

On the other hand, if the click occurs on an element originating from shadow DOM, e.g. on `<b>Name</b>`, then, as it bubbles out of the shadow DOM, its `event.target` is reset to `<user-card>`.

Bubbling, event.composedPath()

For purposes of event bubbling, flattened DOM is used.

So, if we have a slotted element, and an event occurs somewhere inside it, then it bubbles up to the `<slot>` and upwards.

The full path to the original event target, with all the shadow elements, can be obtained using `event.composedPath()`. As we can see from the name of the method, that path is taken after the composition.

In the example above, the flattened DOM is:

```html
<user-card id="userCard">
  #shadow-root
    <div>
      <b>Name:</b>
      <slot name="username">
        <span slot="username">John Smith</span>
      </slot>
    </div>
</user-card>
```

So, for a click on `<span slot="username">`, a call to `event.composedPath()` returns an array: [`span`, `slot`, `div`, `shadow-root`, `user-card`, `body`, `html`, `document`, `window`]. That's exactly the parent chain from the target element in the flattened DOM, after the composition.

Shadow tree details are only provided for `{mode:'open'}` trees

If the shadow tree was created with `{mode: 'closed'}`, then the composed path starts from the host: `user-card` and upwards.

That's the similar principle as for other methods that work with shadow DOM. Internals of closed trees are completely hidden.

event.composed

Most events successfully bubble through a shadow DOM boundary. There are few events that do not.

This is governed by the `composed` event object property. If it's `true`, then the event does cross the boundary. Otherwise, it only can be caught from inside the shadow DOM.

If you take a look at UI Events specification, most events have `composed: true`:

- `blur`, `focus`, `focusin`, `focusout`,
- `click`, `dblclick`,
- `mousedown`, `mouseup` `mousemove`, `mouseout`, `mouseover`,
- `wheel`,
- `beforeinput`, `input`, `keydown`, `keyup`.

All touch events and pointer events also have `composed: true`.

There are some events that have `composed: false` though:

- `mouseenter`, `mouseleave` (they do not bubble at all),
- `load`, `unload`, `abort`, `error`,
- `select`,
- `slotchange`.

These events can be caught only on elements within the same DOM, where the event target resides.

Custom events

When we dispatch custom events, we need to set both `bubbles` and `composed` properties to `true` for it to bubble up and out of the component.

For example, here we create `div#inner` in the shadow DOM of `div#outer` and trigger two events on it. Only the one with `composed: true` makes it outside to the document:

```
<div id="outer"></div>

<script>
outer.attachShadow({mode: 'open'});

let inner = document.createElement('div');
```

```
outer.shadowRoot.append(inner);

/*
div(id=outer)
  #shadow-dom
    div(id=inner)
*/

document.addEventListener('test', event => alert(event.detail));

inner.dispatchEvent(new CustomEvent('test', {
  bubbles: true,
  composed: true,
  detail: "composed"
}));

inner.dispatchEvent(new CustomEvent('test', {
  bubbles: true,
  composed: false,
  detail: "not composed"
}));
</script>
```

Summary

Events only cross shadow DOM boundaries if their `composed` flag is set to `true`.

Built-in events mostly have `composed: true`, as described in the relevant specifications:

- UI Events https://www.w3.org/TR/uievents.
- Touch Events https://w3c.github.io/touch-events.
- Pointer Events https://www.w3.org/TR/pointerevents.
- …And so on.

Some built-in events that have `composed: false`:

- `mouseenter`, `mouseleave` (also do not bubble),
- `load`, `unload`, `abort`, `error`,
- `select`,
- `slotchange`.

These events can be caught only on elements within the same DOM.

If we dispatch a `CustomEvent`, then we should explicitly set `composed: true`.

Please note that in case of nested components, one shadow DOM may be nested into another. In that case composed events bubble through all shadow DOM boundaries. So, if an event is intended only for the immediate enclosing component, we can also dispatch it on the shadow host and set `composed: false`. Then it's out of the component shadow DOM, but won't bubble up to higher-level DOM.

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…)