ResizeObserver: It’s Like document.onresize for Elements

By Surma

Surma is a contributor to Web**Fundamentals**

After MutationObserver, PerformanceObserver and IntersectionObserver, we have another observer for your collection! `ResizeObserver` allows you to be notified when an element’s content rectangle has changed its size, and react accordingly. The spec is currently being iterated on in the WICG and *your* feedback is very much welcome.

Motivation

previously, you had to attach a listener to the document’s `resize` event to get notified of any change of the viewport’s dimensions. In the event handler, you would then have to figure out which elements have been affected by that change and call a specific routine to react appropriately. If you need the new dimensions of an element after a resize, you need to call `getBoundingClientRect` or `getComputerStyle`, which can cause layout thrashing if you don’t take care of batching *all* your reads and *all* your writes.

And then you realize that this doesn’t even cover the cases where elements change their size without the main window having been resized. For example, appending new children, setting an element’s `display` style to `none`, or similar actions can change the size of an element, its siblings or ancestors.

This is why `ResizeObserver` is a useful primitive. It reacts to changes in size of any of the observed *elements*, independent of what caused the change. It provides you access to the new size of the observed elements, too. Let’s get straight into it.

API

All the APIs with the “observer” suffix I mentioned above share a simple API design. `ResizeObserver` is no exception. You create a `ResizeObserver` object and pass a callback to the constructor. The callback will be given an array of `ResizeOberverEntries` – one entry per observed element – which contain the new dimensions for the element.

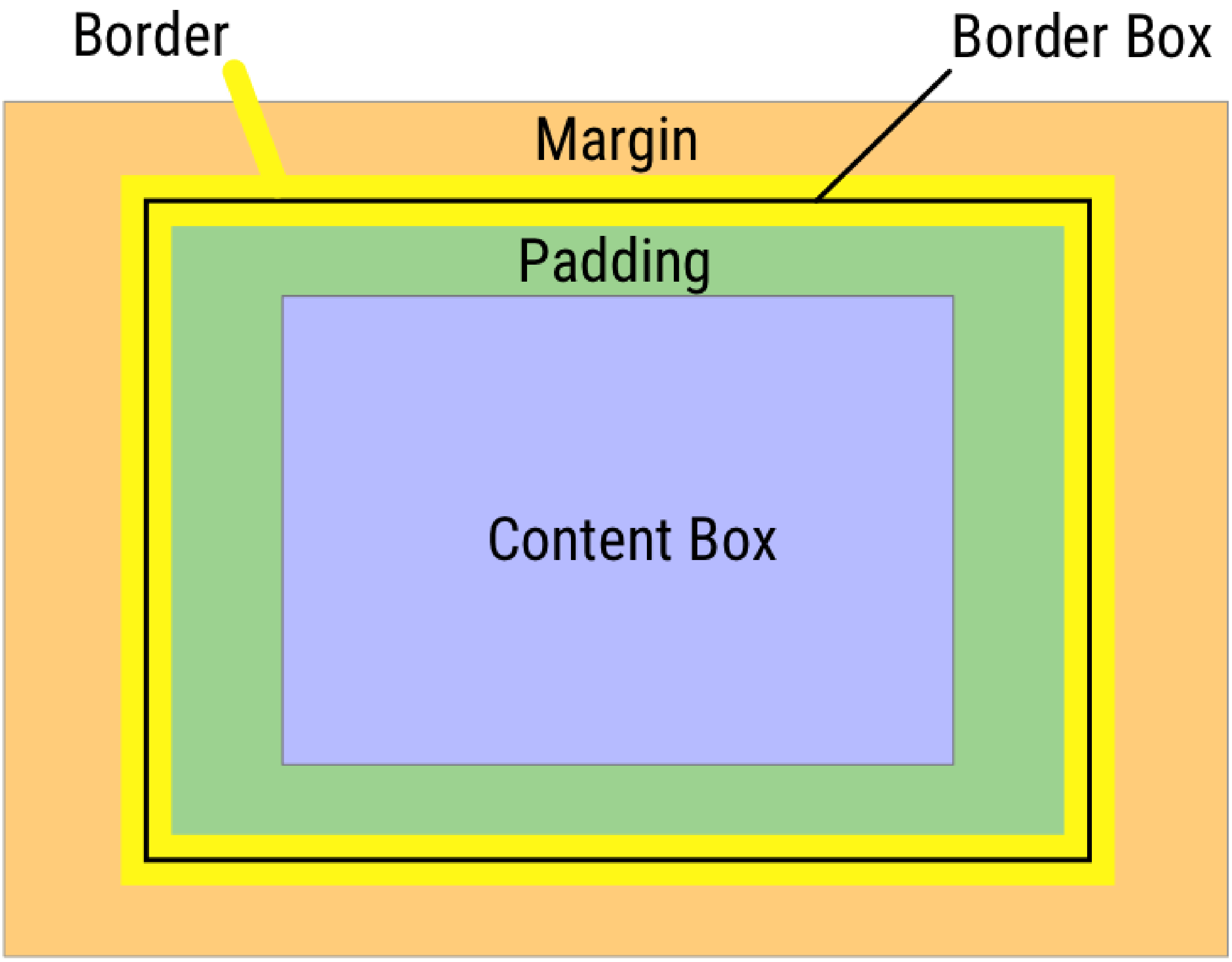
```
var ro = new ResizeObserver( entries => {
  for (let entry of entries) {
    const cr = entry.contentRect;
    console.log('Element:', entry.target);
    console.log(`Element size: ${cr.width}px x ${cr.height}px`);
    console.log(`Element padding: ${cr.top}px ; ${cr.left}px`);
  }
});

// Observe one or multiple elements
ro.observe(someElement);
```

Some details

What is being reported?

Generally, `ResizeObserver` reports the content rectangle of an element. The content rectangle is the box in which content can be placed. It is the border box minus the padding.



It's important to note that while `ResizeObserver` *reports* both the dimensions of the `contentRect` and the padding, it only *watches* the `contentRect`. *Don't* confuse `contentRect` with the bounding box of the element. The bounding box, as reported by `getBoundingClientRect`, is the box that contains the entire element and its decendants. SVGs are an exception to the rule, where `ResizeObserver` will report the dimensions of the bounding box.

When is it being reported?

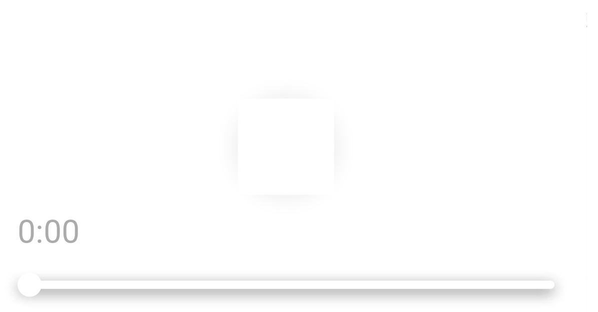
The spec prescribes that `ResizeObserver` should process all resize events before paint and after layout. This makes the callback of an `ResizeObserver` the ideal place to make changes to your page's layout. Because `ResizeObserver` processing happens between layout and paint, doing so will only invalidate layout, not paint.

Gotcha

You might be asking yourself: What happens if I change the size of an observed element inside the `ResizeObserver`'s callback? The answer is: You will trigger another call to the callback right away. However, `ResizeObserver` has a mechanism to avoid infinite callback loops and cyclic dependencies. Changes will only be processed in the same frame if the resized element is deeper in the DOM tree than the *shallowest* element processed in the previous callback. Otherwise, they'll get deferred to the next frame.

Application

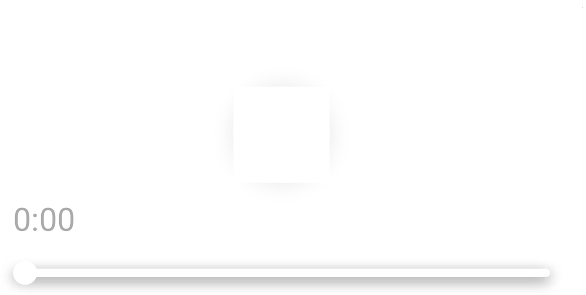
One thing that `ResizeObserver` allows you to do is to implement per-element media queries. By observing elements, you can imperatively define your design breakpoints and change the element's styles. In the following example, the second box will change its border radius according to its width.



```
const ro = new ResizeObserver(entries => {
  for (let entry of entries) {
    entry.target.style.borderRadius = Math.max(0, 250 - entry.contentRect.width)
  }
});
// Only observe the second box
ro.observe(document.querySelector('.box:nth-child(2)'));
```

Another interesting example to look at is a chat window. The problem that arises in a typical top-to-bottom conversation layout is scroll positioning. To avoid confusing the user, it is helpful if the window sticks to the bottom of the conversation, where the newest messages will appear. Additionally, any kind of layout change (think of a phone going from landscape to portrait or vice versa) should strive to achieve the same.

`ResizeObserver` allows you to write a *single* piece of code that takes care of *both* scenarios. Resizing the window is an event that `ResizeObservers` can capture by very definition, but calling `appendChild()` will also resize that element (except if `overflow: hidden` is set), because it needs to make space for the new elements. With this in mind, you can get away with a couple of lines to achieve the desired effect:



```
const ro = new ResizeObserver(entries => {
  document.scrollingElement.scrollTop =
    document.scrollingElement.scrollHeight;
});

// Observe the scrollingElement for when the window gets resized
ro.observe(document.scrollingElement);
// Observe the timeline to process new messages
ro.observe(timeline);
```

Pretty neat, huh?

From here, we could add more code to handle the case where the user has scrolled up manually and we want to the scrolling to stick to *that* message when a new message comes in.

Another use case is for any kind of custom element that is doing its own layout. Until `ResizeObserver`, there was no reliable way to get notified when your own dimensions change so you ca re-layout your own children.

Out now!

As with a lot of the observer APIs, `ResizeObserver` is not 100% polyfillable, which is why native implementations are needed. Current polyfill implementations either rely on polling or on adding sentinel elements to the DOM. The former will drain your battery on mobile by keeping the CPU busy while the latter modifies your DOM and might mess up styling and other DOM-reliant code.

`ResizeObserver` is in Chrome 55 Canary, behind the Experimental Web Platform flag. It is a small primitive that allows you to write certain effects in a much more efficient way. Try them out and let us know what you think or if you have questions.

Except as otherwise noted, the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code samples are licensed under the Apache 2.0 License. For details, see our Site Policies. Java is a registered trademark of Oracle and/or its affiliates.

Last updated July 2, 2018.