ResizeObserver: it’s like document.onresize for elements

- API
- Some details
  - What is being reported?
  - When is it being reported?
  - Gotcha
- Application
- Conclusion

Before `ResizeObserver`, 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 needed the new dimensions of an element after a resize, you had to call `getBoundingClientRect()` or `getComputedStyle()`, which can cause layout thrashing if you don't take care of batching *all* your reads and *all* your writes.

This didn't even cover 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 its 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 access to the new size of the observed elements too.

API

All the APIs with the `Observer` suffix we 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 is passed an array of `ResizeObserverEntry` objects—one entry per observed element—which contains 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, a `ResizeObserverEntry` reports the content box of an element through a property called `contentRect`, which returns a `DOMRectReadOnly` object. The content box is the box in which content can be placed. It is the border box minus the padding.

A diagram of the CSS box model.

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 descendants. SVGs are an exception to the rule, where `ResizeObserver` will report the dimensions of the bounding box.

As of Chrome 84, `ResizeObserverEntry` has three new properties to provide more detailed information. Each of these properties returns a `ResizeObserverSize` object containing a `blockSize` property and an `inlineSize` property. This information is about the observered element at the time the callback is invoked.

- `borderBoxSize`
- `contentBoxSize`

- `devicePixelContentBoxSize`

All of these items return read-only arrays because in the future it's hoped that they can support elements that have multiple fragments, which occur in multi-column scenarios. For now, these arrays will only contain one element.

Platform support for these properties is limited, but Firefox already supports the first two.

When is it being reported?

The spec proscribes that `ResizeObserver` should process all resize events before paint and after layout. This makes the callback of a `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 callback to `ResizeObserver`? The answer is: you will trigger another call to the callback right away. Fortunately, `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 an 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) + 'px';
  }
});
// 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 appear. Additionally, any kind of layout change (think of a phone going from landscape to portrait or vice versa) should 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 a `ResizeObserver` can capture by definition, but calling `appendChild()` also resizes that element (unless `overflow: hidden` is set), because it needs to make space for the new elements. With this in mind, it takes very few 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, I could add more code to handle the case where the user has scrolled up manually and wants 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 its dimensions change so its children can be laid out again.

Conclusion

`ResizeObserver` is available in most major browsers. In some cases, that availability is quite recent.

There are some polyfills available but it is not possible to completely duplicate the functionality of `ResizeObserver`. Current 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.

Photo by Markus Spiske on Unsplash.

□□□ #3
□ □□□ □□□ 2026-10-31 22:14:31 UTC

 cocobook-sync □□□ 2026-11-04 17:37:57 UTC