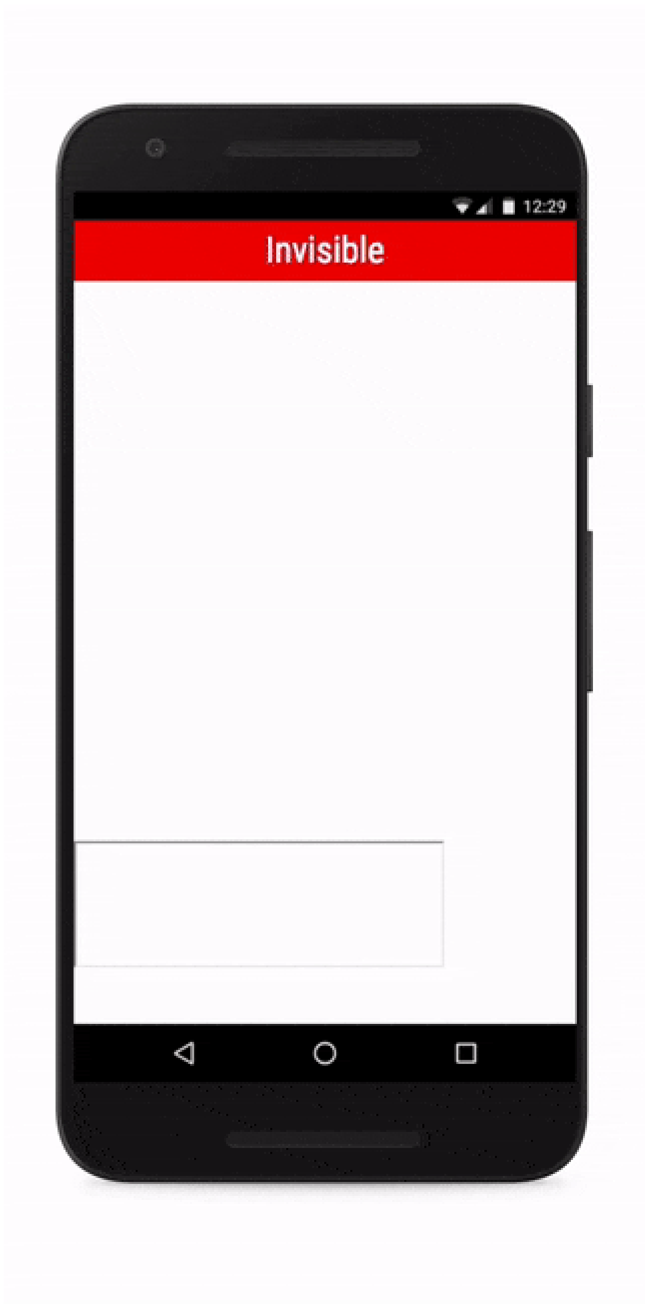
IntersectionObserver's Coming into View

By Surma

Surma is a contributor to Web**Fundamentals**

Let's say you want to track when an element in your DOM enters the visible viewport. You might want to do this so you can lazy-load images just in time or because you need to know if the user is actually looking at a certain ad banner. You can do that by hooking up the scroll event or by using a periodic timer and calling `getBoundingClientRect()` on that element. This approach, however, is painfully slow as each call to `getBoundingClientRect()` forces the browser to re-layout the entire page and will introduce considerable jank to your website. Matters get close to impossible when you know your site is being loaded inside an iframe and you want to know when the user can see an element. The Single Origin Model and the browser won't let you access any data from the web page that contains the iframe. This is a common problem for ads for example, that are frequently loaded using iframes.

Making this visibility test more efficient is what `IntersectionObserver` was designed for, and it's landed in Chrome 51 (which is, as of this writing, the beta release). `IntersectionObservers` let you know when an observed element enters or exits the browser's viewport.



How to create an IntersectionObserver

The API is rather small, and best described using an example:

```
var io = new IntersectionObserver(
  entries => {
    console.log(entries);
  },
  {
    /* Using default options. Details below */
  }
);
```

```
// Start observing an element
io.observe(element);

// Stop observing an element
// io.unobserve(element);

// Disable entire IntersectionObserver
// io.disconnect();
```

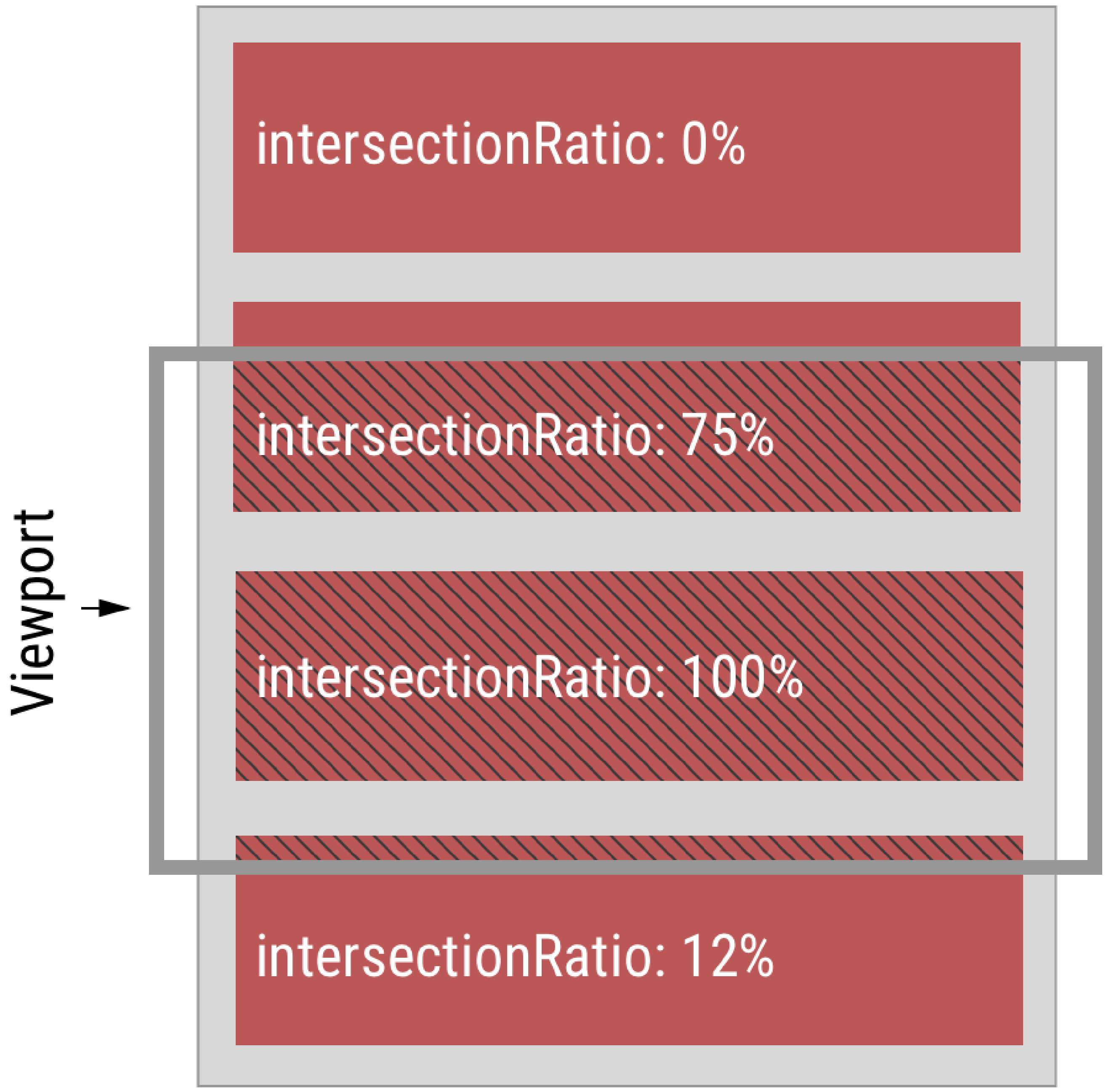
Using the default options for **`IntersectionObserver`**, your callback will be called both when the element comes partially into view and when it completely leaves the viewport.

If you *need* to observe multiple elements, it is both possible and advised to observe multiple elements using the *same* **`IntersectionObserver`** instance by calling **`observe()`** multiple times.

An **`entries`** parameter is passed to your callback which is an array of **`IntersectionObserverEntry`** objects. Each such object contains updated intersection data for one of your observed elements.

```
▿[IntersectionObserverEntry]
    time: 3893.92
  ▿rootBounds: ClientRect
      bottom: 920
      height: 1024
      left: 0
      right: 1024
      top: 0
      width: 920
  ▿boundingClientRect: ClientRect
    // ...
  ▿intersectionRect: ClientRect
    // ...
    intersectionRatio: 0.54
  ▿target: div#observee
    // ...
```

`rootBounds` is the result of calling **`getBoundingClientRect()`** on the root element, which is the viewport by default. **`boundingClientRect`** is the result of **`getBoundingClientRect()`** called on the observed element. **`intersectionRect`** is the intersection of these two rectangles and effectively tells you *which part* of the observed element is visible. **`intersectionRatio`** is closely related, and tells you *how much* of the element is visible. With this info at your disposal, you are now able to implement features like just-in-time loading of assets before they become visible on screen. Efficiently.



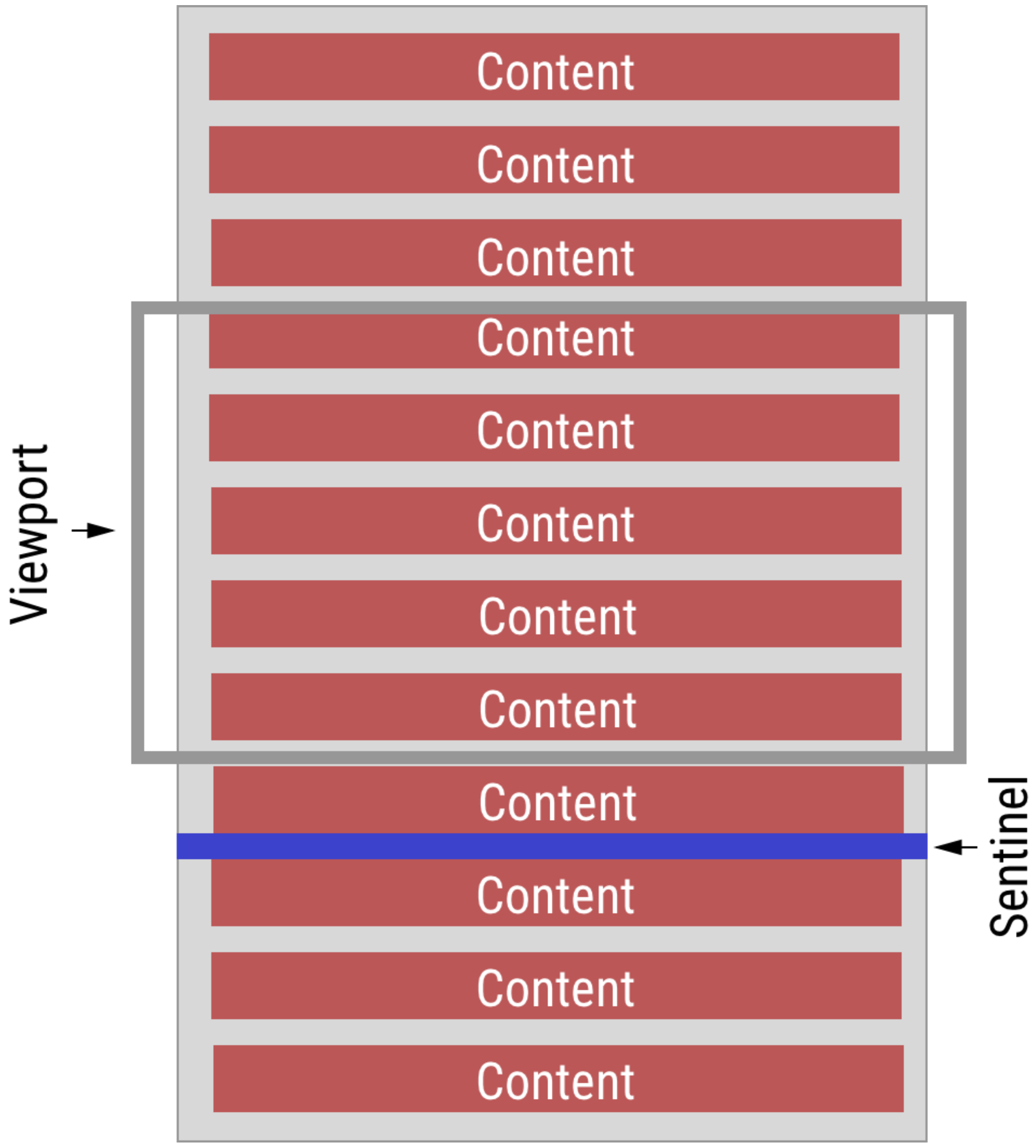
`IntersectionObservers` deliver their data asynchronously, and your callback code will run in the main thread. Additionally, the spec actually says that IntersectionObserver implementations should use `requestIdleCallback()`. This means that the call to your provided callback is low priority and will be made by the browser during idle time. This is a conscious design decision.

Scrolling divs

I am not a big fan of scrolling inside an element, but I am not here to judge, and neither are `IntersectionObservers`. The `options` object takes a `root` option that lets you define an alternative to the viewport as your root. It is important to keep in mind that `root` needs to be an ancestor of all the observed elements.

Intersect all the things!

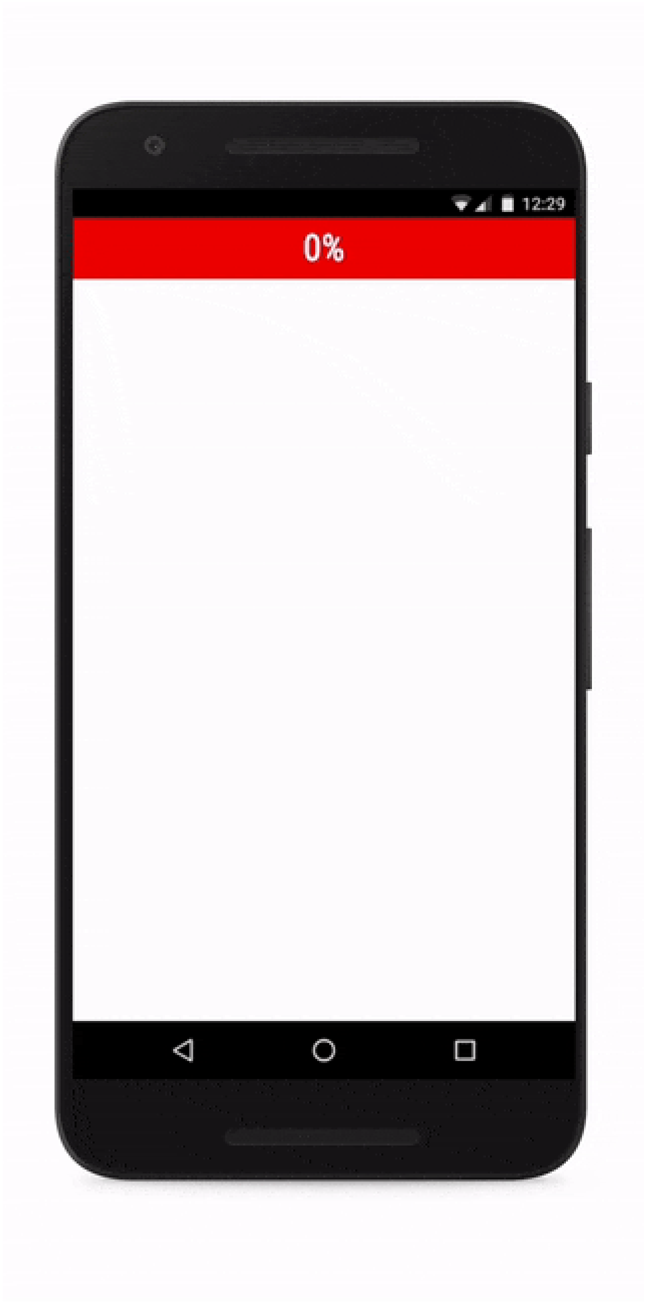
No! Bad developer! That's not mindful usage of your user's CPU cycles. Let's think about an infinite scroller as an example: In that scenario, it is definitely advisable to add sentinels to the DOM and observe (and recycle!) those. You should add a sentinel close to the last item in the infinite scroller. When that sentinel comes into view, you can use the callback to load data, create the next items, attach them to the DOM and reposition the sentinel accordingly. If you properly recycle the sentinel, no additional call to `observe()` is needed. The `IntersectionObserver` keeps working.



More updates, please

As mentioned earlier, the callback will be triggered a single time when the observed element comes partially into view and another time when it has left the viewport. This way `IntersectionObserver` gives you an answer to the question, “Is element X in view?”. In some use cases, however, that might not be enough.

That’s where the `threshold` option comes into play. It allows you to define an array of `intersectionRatio` thresholds. Your callback will be called every time `intersectionRatio` crosses one of these values. The default value for `threshold` is `[0]`, which explains the default behavior. If we change `threshold` to `[0, 0.25, 0.5, 0.75, 1]`, we will get notified every time an additional quarter of the element becomes visible:



Any other options?

As of now, there's only one additional option to the ones listed above. `rootMargin` allows you to specify the margins for the root, effectively allowing you to either grow or shrink the area used for intersections. These margins are specified using a CSS-style string, á la "`10px 20px 30px 40px`", specifying top, right, bottom and left margin respectively. To summarize, the `IntersectionObservers` options struct offers the following options:

```
new IntersectionObserver(entries => {/* … */}, {
  // The root to use for intersection.
  // If not provided, use the top-level document's viewport.
  root: null,
  // Same as margin, can be 1, 2, 3 or 4 components, possibly negative lengths.
  // If an explicit root element is specified, components may be percentages of t
  // root element size.  If no explicit root element is specified, using a percen
  // is an error.
  rootMargin: "0px",
  // Threshold(s) at which to trigger callback, specified as a ratio, or list of
  // ratios, of (visible area / total area) of the observed element (hence all
  // entries must be in the range [0, 1]).  Callback will be invoked when the vis
  // ratio of the observed element crosses a threshold in the list.
  threshold: [0],
});
```

iframe magic

`IntersectionObservers` were designed specifically with ads services and social network widgets in mind, which frequently use iframes and could benefit from knowing whether they are in view. If an iframe observes one of its elements, both scrolling the iframe as well as scrolling the window *containing the iframe* will trigger the callback at the appropriate times. For the latter case, however, `rootBounds` will be set to `null` to avoid leaking data across origins.

What is IntersectionObserver *Not* about?

Something to keep in mind is that `IntersectionObservers` are intentionally neither pixel perfect nor low latency. Using them to implement endeavours like scroll-dependent animations are bound to fail, as the data will be – strictly speaking – out of date by the time you'll get to use it. The explainer has more details about the original use cases for `IntersectionObserver`.

How much work can I do in the callback?

Short'n'Sweet: Spending too much time in the callback will make your app lag – all the common practices apply.

Go forth and intersect thy elements

The browser support for `IntersectionObservers` is still fairly slim, so it won't work everywhere right off the bat just yet. In the meantime, a polyfill is being worked on in the WICG's repository. Obviously, you won't get the performance benefits using that polyfill that a native implementation would give you.

You can start using `IntersectionObservers` right now in Chrome Canary! Tell us what you came up with.
