Inside look at modern web browser (part 4)

Published on Friday, September 21, 2018 • Updated on Saturday, January 12, 2019

Mariko Kosaka
Mariko is a drawsplainer

Table of contents

Input is coming to the Compositor

This is the last of the 4 part blog series looking inside of Chrome; investigating how it handles our code to display a website. In the previous post, we looked at the rendering process and learned about the compositor. In this post, we'll look at how compositor is enabling smooth interaction when user input comes in.

Input events from the browser's point of view

When you hear "input events" you might only think of a typing in textbox or mouse click, but from the browser's point of view, input means any gesture from the user. Mouse wheel scroll is an input event and touch or mouse over is also an input event.

When user gesture like touch on a screen occurs, the browser process is the one that receives the gesture at first. However, the browser process is only aware of where that gesture occurred since content inside of a tab is handled by the renderer process. So the browser process sends the event type (like `touchstart`) and its coordinates to the renderer process. Renderer process handles the event appropriately by finding the event target and running event listeners that are attached.

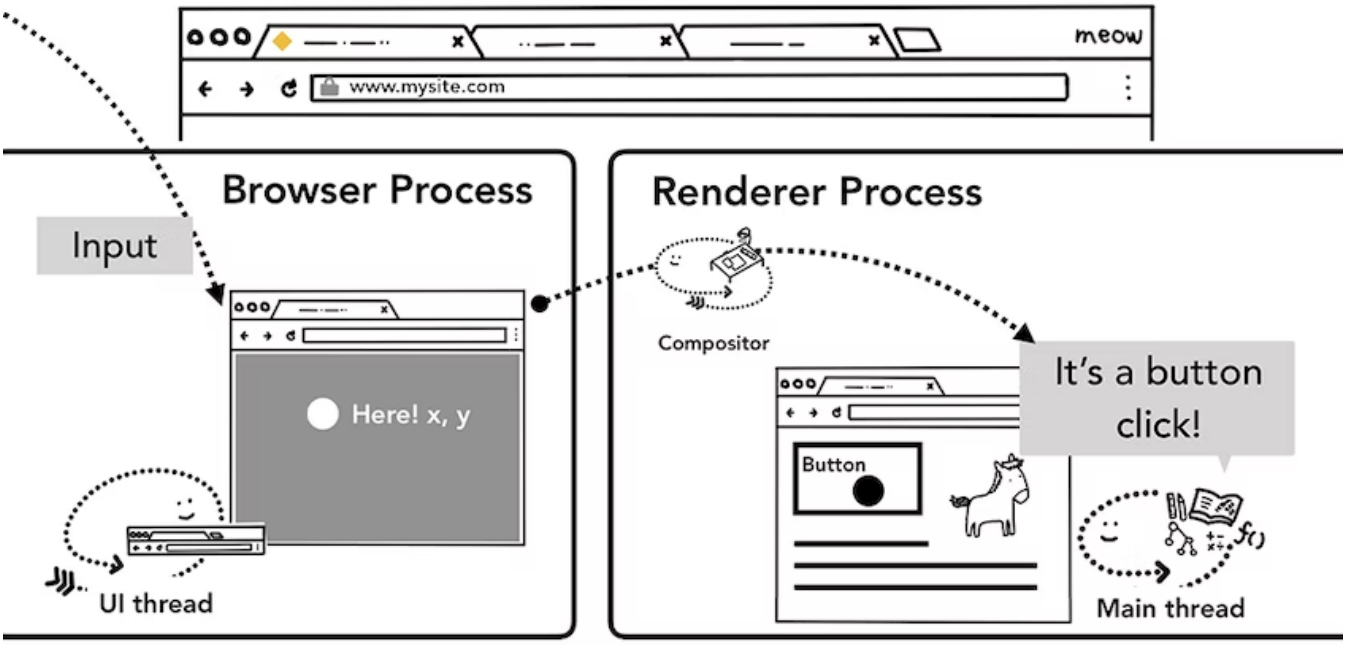

Figure 1: Input event routed through the browser process to the renderer process

Compositor receives input events

Figure 2: Viewport hovering over page layers

In the previous post, we looked at how the compositor could handle scroll smoothly by compositing rasterized layers. If no input event listeners are attached to the page,

Compositor thread can create a new composite frame completely independent of the main thread. But what if some event listeners were attached to the page? How would the compositor thread find out if the event needs to be handled?

Understanding non-fast scrollable region

Since running JavaScript is the main thread's job, when a page is composited, the compositor thread marks a region of the page that has event handlers attached as "Non-Fast Scrollable Region". By having this information, the compositor thread can make sure to send input event to the main thread if the event occurs in that region. If input event comes from outside of this region, then the compositor thread carries on compositing new frame without waiting for the main thread.

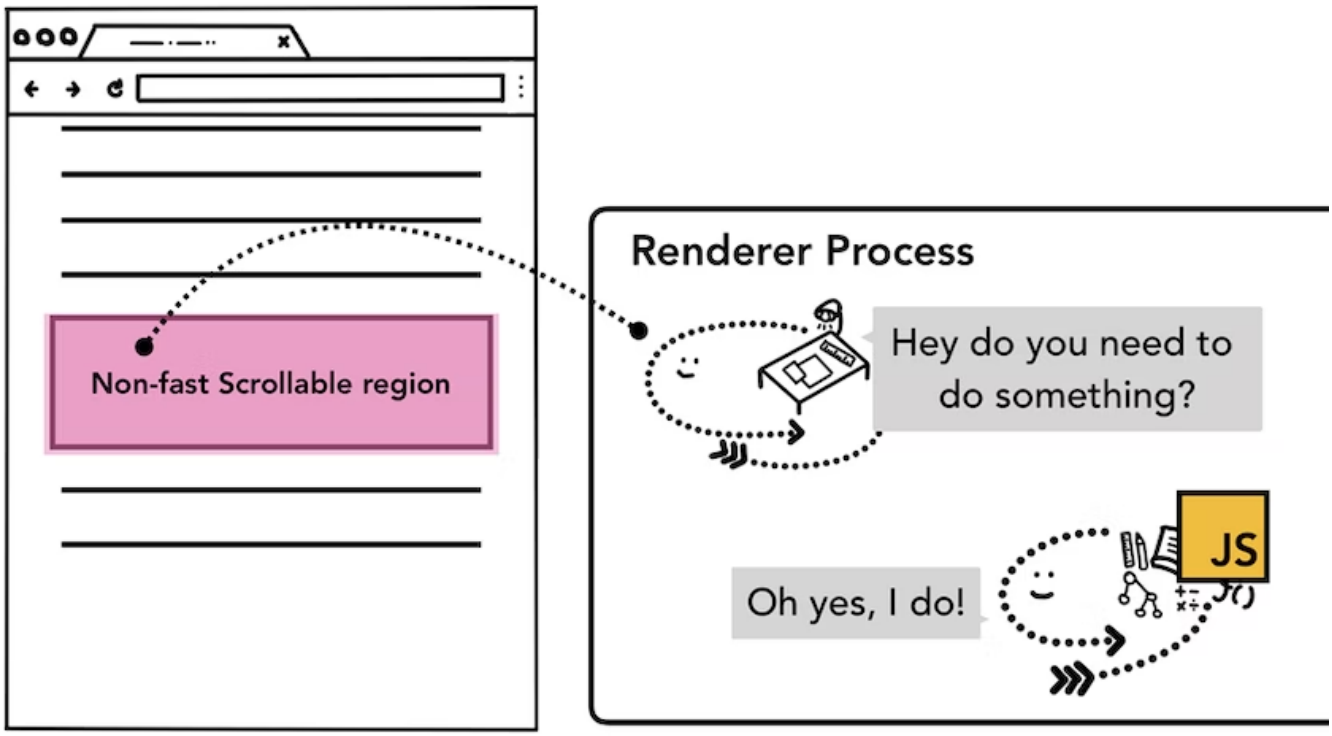

Figure 3: Diagram of described input to the non-fast scrollable region

Be aware when you write event handlers

A common event handling pattern in web development is event delegation. Since events bubble, you can attach one event handler at the topmost element and delegate tasks based on event target. You might have seen or written code like the below.

```
document.body.addEventListener('touchstart', event => {
    if (event.target === area) {
        event.preventDefault();
    }
});
```

Since you only need to write one event handler for all elements, ergonomics of this event delegation pattern are attractive. However, if you look at this code from the browser's point of view, now the entire page is marked as a non-fast scrollable region. This means even if your application doesn't care about input from certain parts of the page, the compositor thread has to communicate with the main thread and wait for it every time an input event comes in. Thus, the smooth scrolling ability of the compositor is defeated.

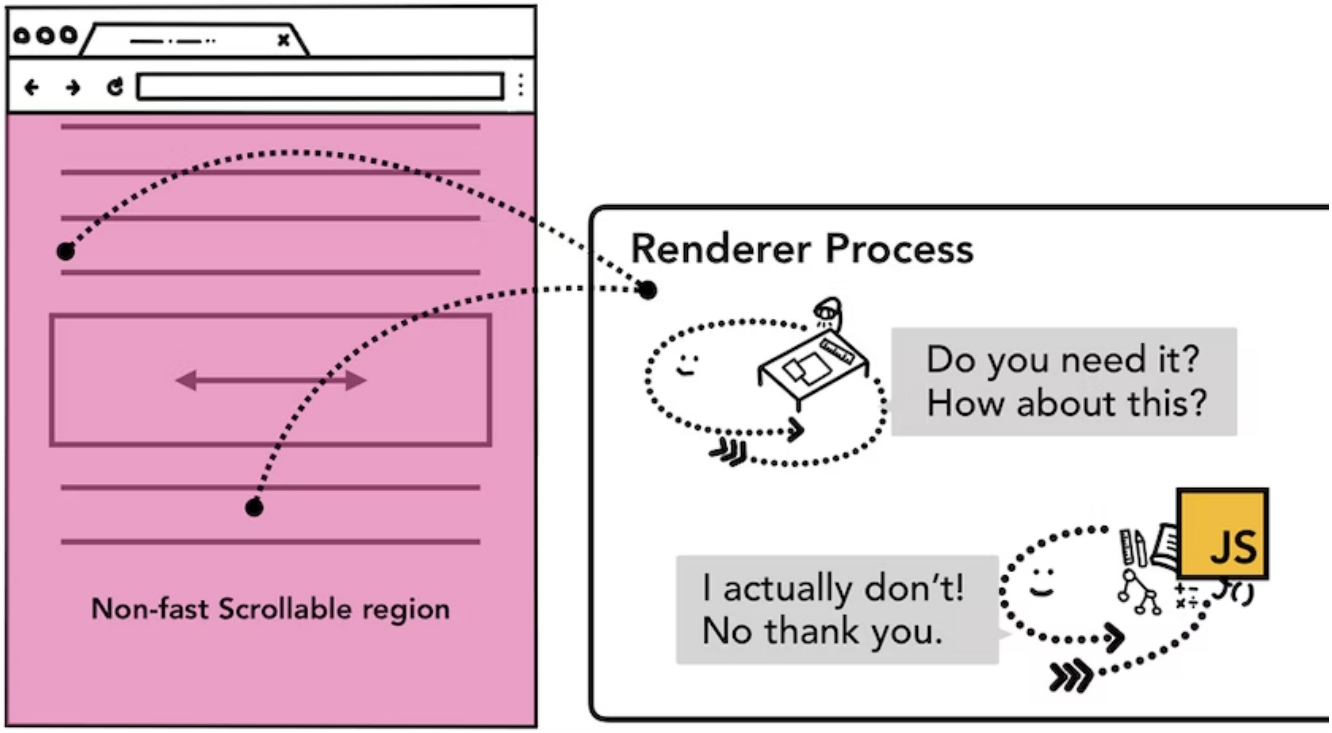

Figure 4: Diagram of described input to the non-fast scrollable region covering an entire page

In order to mitigate this from happening, you can pass `passive: true` options in your event listener. This hints to the browser that you still want to listen to the event in the main thread, but compositor can go ahead and composite new frame as well.

```
document.body.addEventListener('touchstart', event => {
    if (event.target === area) {
```

```
        event.preventDefault()
    }
 }, {passive: true});
```

Check if the event is cancelable

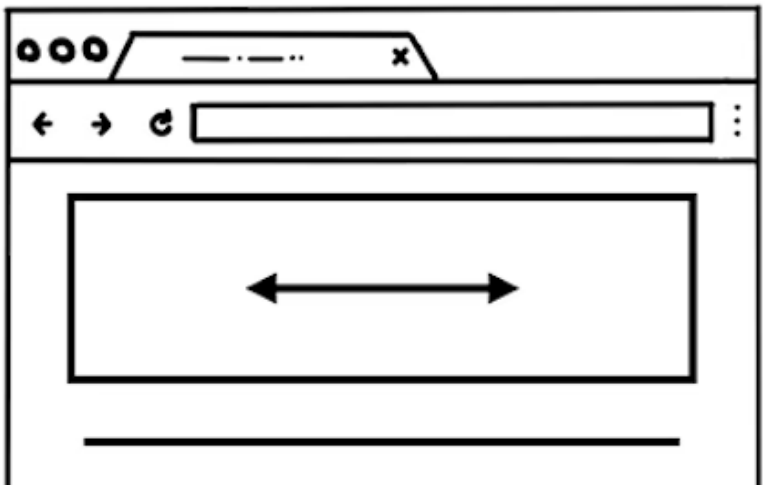

Figure 5: A web page with part of the page fixed to horizontal scroll

Imagine you have a box in a page that you want to limit scroll direction to horizontal scroll only.

Using `passive: true` option in your pointer event means that the page scroll can be smooth, but vertical scroll might have started by the time you want to `preventDefault` in order to limit scroll direction. You can check against this by using `event.cancelable` method.

```
document.body.addEventListener('pointermove', event => {
    if (event.cancelable) {
        event.preventDefault(); // block the native scroll
        /*
        *  do what you want the application to do here
        */
    }
}, {passive: true});
```

Alternatively, you may use CSS rule like `touch-action` to completely eliminate the event handler.

```
#area {
  touch-action: pan-x;
}
```

Finding the event target

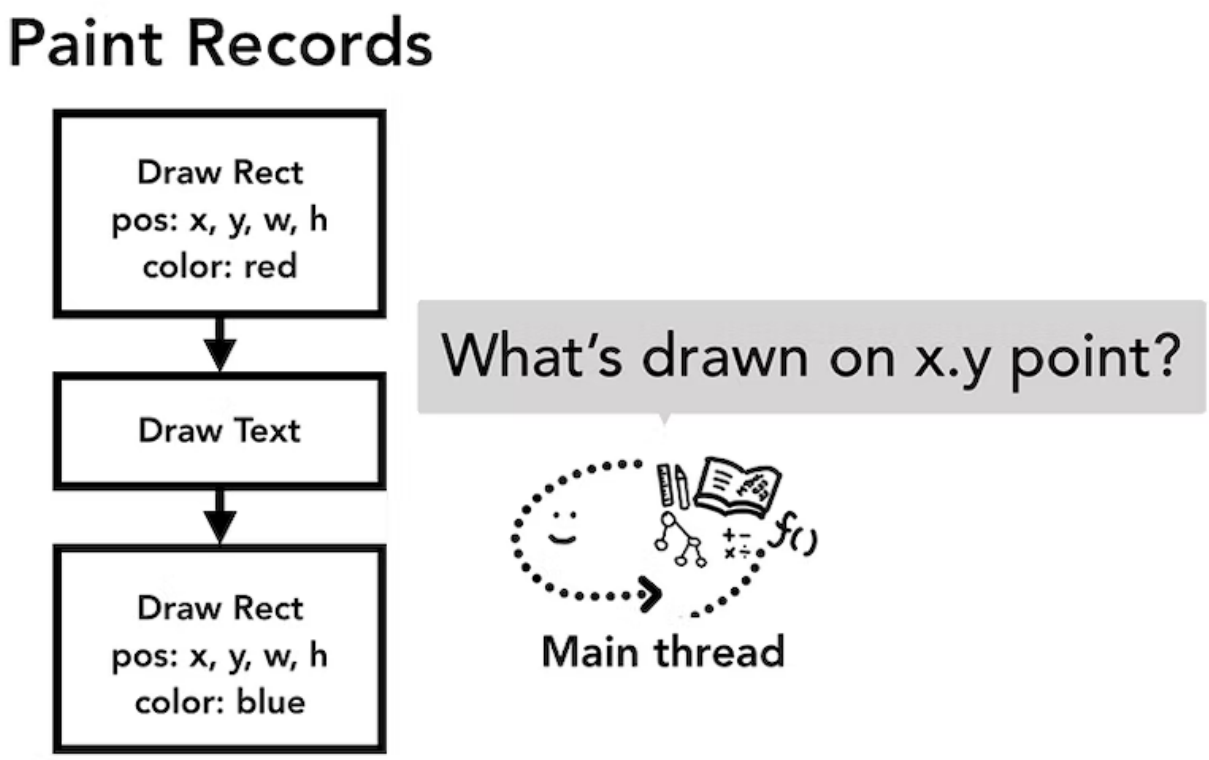

Figure 6: The main thread looking at the paint records asking what's drawn on x.y point

When the compositor thread sends an input event to the main thread, the first thing to run is a hit test to find the event target. Hit test uses paint records data that was generated in the rendering process to find out what is underneath the point coordinates in which the event occurred.

Minimizing event dispatches to the main thread

In the previous post, we discussed how our typical display refreshes screen 60 times a second and how we need to keep up with the cadence for smooth animation. For input, a

typical touch-screen device delivers touch event 60-120 times a second, and a typical mouse delivers events 100 times a second. Input event has higher fidelity than our screen can refresh.

If a continuous event like `touchmove` was sent to the main thread 120 times a second, then it might trigger excessive amount of hit tests and JavaScript execution compared to how slow the screen can refresh.

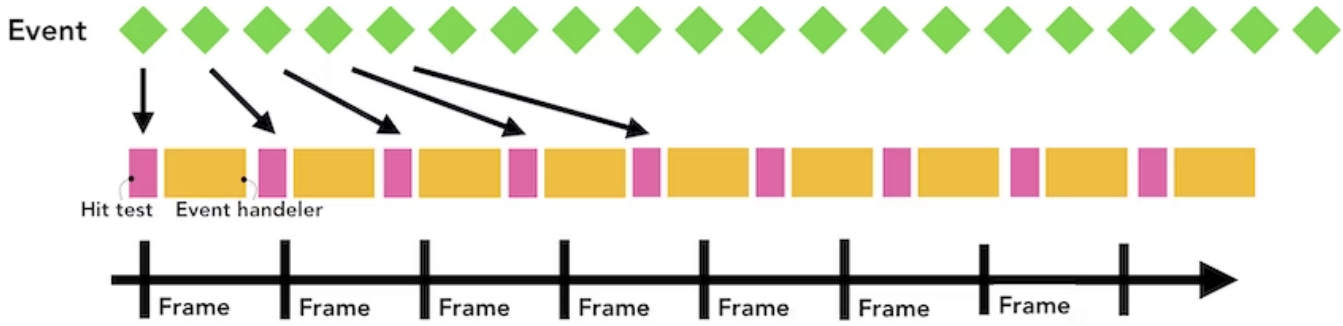

Figure 7: Events flooding the frame timeline causing page jank

To minimize excessive calls to the main thread, Chrome coalesces continuous events (such as `wheel`, `mousewheel`, `mousemove`, `pointermove`, `touchmove`) and delays dispatching until right before the next `requestAnimationFrame`.

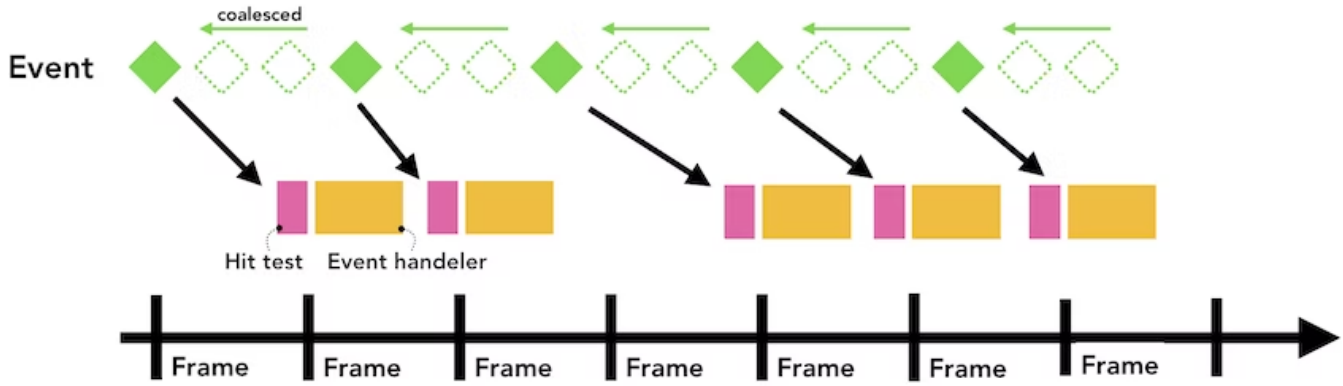

Figure 8: Same timeline as before but event being coalesced and delayed

Any discrete events like `keydown`, `keyup`, `mouseup`, `mousedown`, `touchstart`, and `touchend` are dispatched immediately.

Use `getCoalescedEvents` to get intra-frame events

For most web applications, coalesced events should be enough to provide a good user experience. However, if you are building things like drawing application and putting a path based on `touchmove` coordinates, you may lose in-between coordinates to draw a smooth line. In that case, you can use the `getCoalescedEvents` method in the pointer event to get information about those coalesced events.

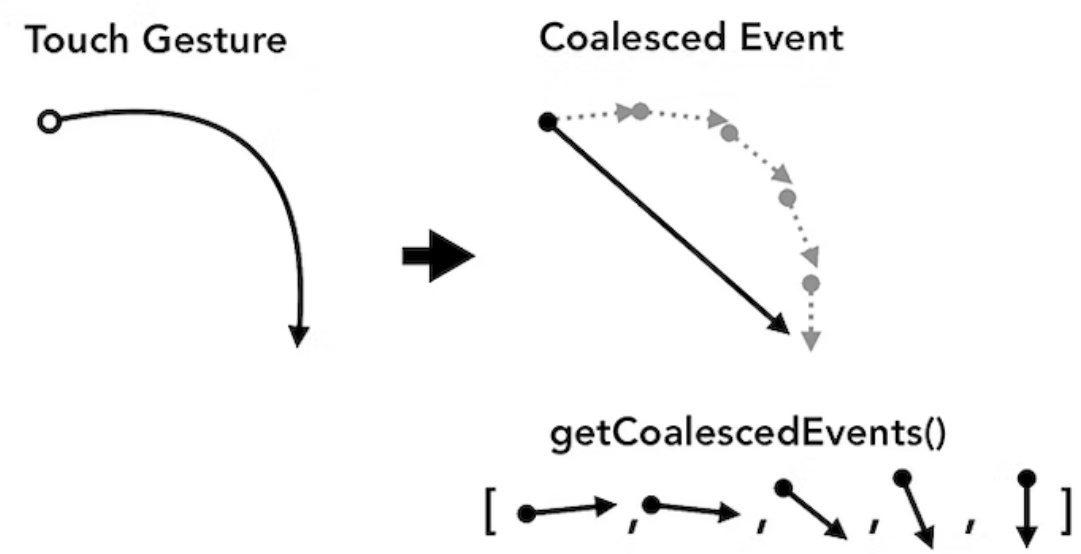

Figure 9: Smooth touch gesture path on the left, coalesced limited path on the right

```
window.addEventListener('pointermove', event => {
    const events = event.getCoalescedEvents();
    for (let event of events) {
        const x = event.pageX;
        const y = event.pageY;
        // draw a line using x and y coordinates.
    }
});
```

Next steps

In this series, we've covered inner workings of a web browser. If you have never thought about why DevTools recommends adding `{passive: true}` on your event handler or why you might write `async` attribute in your script tag, I hope this series shed some light on why a browser needs those information to provide faster and smoother web experience.

Use Lighthouse

If you want to make your code be nice to the browser but have no idea where to start, Lighthouse is a tool that runs audit of any website and gives you a report on what's being done right and what needs improvement. Reading through the list of audits also gives you an idea of what kind of things a browser cares about.

Learn how to measure performance

Performance tweaks may vary for different sites, so it is crucial that you measure the performance of your site and decide what fits the best for your site. Chrome DevTools team has few tutorials on how to measure your site's performance.

Add Feature Policy to your site

If you want to take an extra step, Feature Policy is a new web platform feature that can be a guardrail for you when you are building your project. Turning on feature policy guarantees the certain behavior of your app and prevents you from making mistakes. For example, If you want to ensure your app will never block the parsing, you can run your app on synchronous scripts policy. When `sync-script: 'none'` is enabled, parser-blocking JavaScript will be prevented from executing. This prevents any of your code from blocking the parser, and the browser doesn't need to worry about pausing the parser.

Wrap up

When I started building websites, I almost only cared about how I would write my code and what would help me be more productive. Those things are important, but we should also think about how browser takes the code we write. Modern browsers have been and continue to invest in ways to provide a better web experience for users. Being nice to the browser by organizing our code, in turn, improves your user experience. I hope you join me in the quest to be nice to the browsers!

Huge thank you to everyone who reviewed early drafts of this series, including (but not limited to): Alex Russell, Paul Irish, Meggin Kearney, Eric Bidelman, Mathias Bynens, Addy Osmani, Kinuko Yasuda, Nasko Oskov, and Charlie Reis.

Did you enjoy the this series? If you have any questions or suggestions for the future post, I'd love to hear from you in the comment section below or @kosamari on Twitter.

Last updated: Saturday, January 12, 2019 Improve article

Connect

Twitter

YouTube

GitHub

Google Developers

Chrome Firebase All products Privacy Terms

Choose language

ENGLISH (en)