APR 21ST, 2014

Document Loading and DOM Lifecycle Events

Document loading and load events can be a bit confusing for newcomers. Multiple names for the same basic things, incompletely documented, ambiguously explained and those ever-present browser inconsistencies don't exactly help. I want to try and remedy this by providing a rundown of document loading lifecycle events.

Document start

The document begins to load; HTML is tokenized and parsed. Scripts are *usually* downloaded, parsed and executed as they are encountered, and hold up the parsing of the document, unless they have the `async` or `defer` properties. Async and deferred scripts are given to another thread to handle whilst the 'main' thread goes through the HTML and all other scripts in source order. Newer browsers also perform speculative parsing, racing ahead in the HTML to see if there are any resources it can start downloading whilst the current script is being obtained and run.

Once the HTML is parsed the DOM has been built, the browser waits for all deferred scripts to finish downloading, then executes them in their source order.

As they'll be executing against a document structure that isn't ready, it's best for any scripts running in this phase to just perform data setup and attach callbacks to events that will fire when the document is more complete, either window.onLoad or, more commonly, **DOMContentLoaded**.

DOMContentLoaded, aka '(jQuery ready()'

Once the HTML is parsed, the document tree is built, and all non-async scripts have been executed (*including* scripts with `defer` [note 1]), the browser will typically fire the first lifecycle event, **DOMContentLoaded**. This can happen before any resources (images, iframes, stylesheets, fonts, frames) are loaded. Most developers know this event through jQuery's `ready()`, which lets you attach callbacks to be executed once the document is ready to read and manipulate.

DOMContentLoaded is a relatively recent standard, however, and as you might expect, legacy IE doesn't implement it. Internet Explorer 5-8 use `document.readyState='interactive'` and fire `readyStateChange`. The timing isn't specified, though, and can sometimes fire late, so jQuery uses a trick where it constantly attempts to scroll the document until that action no longer returns an exception – at this point, we know IE's DOM is stable. You can check this out in the appendix below, showing how `ready()` does its stuff.

There also seems to be a bug in Firefox currently where DOMContentLoaded can fire **before** all scripts with `defer` have been run. Hopefully it should be fixed soon.

When the event fires, its callback gets executed as you'd expect. If a lot of JS is attached, this can take a few hundred milliseconds, but threads downloading async scripts and images will continue doing their thing whilst this happens (though JavaScript won't be run until the callback finishes). If you're interested in timings, WebPageTest exposes it with the `domContentLoadedEventStart` and `domContentLoadedEventEnd` metrics.

Resources loaded in this phase will fire their own `load` events as they pop into place.

window.onLoad, aka 'body.onLoad'

Once all resources are downloaded (including async scripts), we fire **window.onLoad** [note 2]. This is the granddaddy of load events, and was implemented as far back as the Netscape days. It's implemented pretty consistently across browsers, and is a fairly common metric for "loadedness" (because the UI's responsive, the images are loaded, and the browser busy indicator usually switches off). Note that 'all resources' doesn't include AJAX requests, and that dynamically added resources don't trigger re-firings of the onLoad event.

Window.onLoad and body.onLoad are aliases for the same event, and will overwrite each other. There seems to be some misinformation circulating that body.onLoad is an alias for DOMContentLoaded, but **this is incorrect**.

If a dependent resource does not load (e.g. an image request returns a 404), this will not stop the event firing.

Summary

- If you just need the DOM to be in a legal state for your JS to manipulate, you can target DOMContentLoaded rather than window.onLoad. jQuery's "ready" makes it quite easy to bind a callback to this event, or some suitable alternative. If you don't want to use jQuery, you can implement a fallback that uses readyStateChange.
- DOMContentLoaded will not wait for async scripts, so stuff attached to this event shouldn't have any async dependencies. You should be able to rely on dependencies from scripts with 'defer', Firefox's current bugs notwithstanding.
- If you have an async dependency, wait for window.onLoad
- If you're interested in the loading of a particular element (including a script or stylesheet), you can listen to load events on those elements alone.

Notes

1. There is apparently a bug in Firefox where DOMContentLoaded can fire **before** all scripts with `defer` have been run.
2. Caveat: Unless we're IE6-9, in which case it turns out we might fire the onLoad event before async scripts are downloaded. Typical.

Appendix: jQuery's cross-browser 'ready' implementation

jQuery core

```
// Catch cases where $(document).ready() is called after the
// browser event has already occurred.
    if ( document.readyState === "complete" ) {
        return jQuery.ready();
    }

    // Mozilla, Opera and webkit nightlies currently support this event
    if ( document.addEventListener ) {
        // Use the handy event callback
        document.addEventListener( "DOMContentLoaded", DOMContentLoaded, false );

        // A fallback to window.onload, that will always work
        window.addEventListener( "load", jQuery.ready, false );

// If IE event model is used
    } else if ( document.attachEvent ) {
        // ensure firing before onload,
        // maybe late but safe also for iframes
        document.attachEvent("onreadystatechange", function(){
            if ( document.readyState === "complete" ) {
                document.detachEvent( "onreadystatechange", arguments.callee );
                jQuery.ready();
            }
        });

        // If IE and not an iframe
        // continually check to see if the document is ready
        if ( document.documentElement.doScroll && window == window.top ) (function(){
            if ( jQuery.isReady ) return;

            try {
                // If IE is used, use the trick by Diego Perini
                // http://javascript.nwbox.com/IEContentLoaded/
                document.documentElement.doScroll("left");
            } catch( error ) {
                setTimeout( arguments.callee, 0 );
                return;
            }

            // and execute any waiting functions
            jQuery.ready();
        })();
    }
```

Posted by Jimmy Breck-McKye • Apr 21st, 2014 • DOM, JavaScript, events, front-end

« Disabling Firefox Safe Mode

JavaScript curry »

Comments

Recent Posts

GitHub Repos

octopress-parse
I knocked up some utils to parse Octopress post stats

tzkl
Reactive microlibrary providing Knockout-style observables and computeds for 401 bytes

react-playground-bar-charts
Toy repository for learning React.js

hands-free-website
Use speech synthesis and recognition APIs to build a hands-free website

premock
Store up commands for a function that doesn't yet exist, and run them when it does

raphael-paragraph
Add paragraphs of wrapped text in RaphaelJs

react-synthesizer
A simple WebAudio API synthesizer with a basic React UI

snakeGame
W.I.P.

dynamicText
Simple lib for dynamic text utils in JavaScript

blogStyle
Old wordpress blog for breckmckye.com

ffvii_damage_calc
Final Fantasy VII Damage Calculator

@breckmckye on GitHub