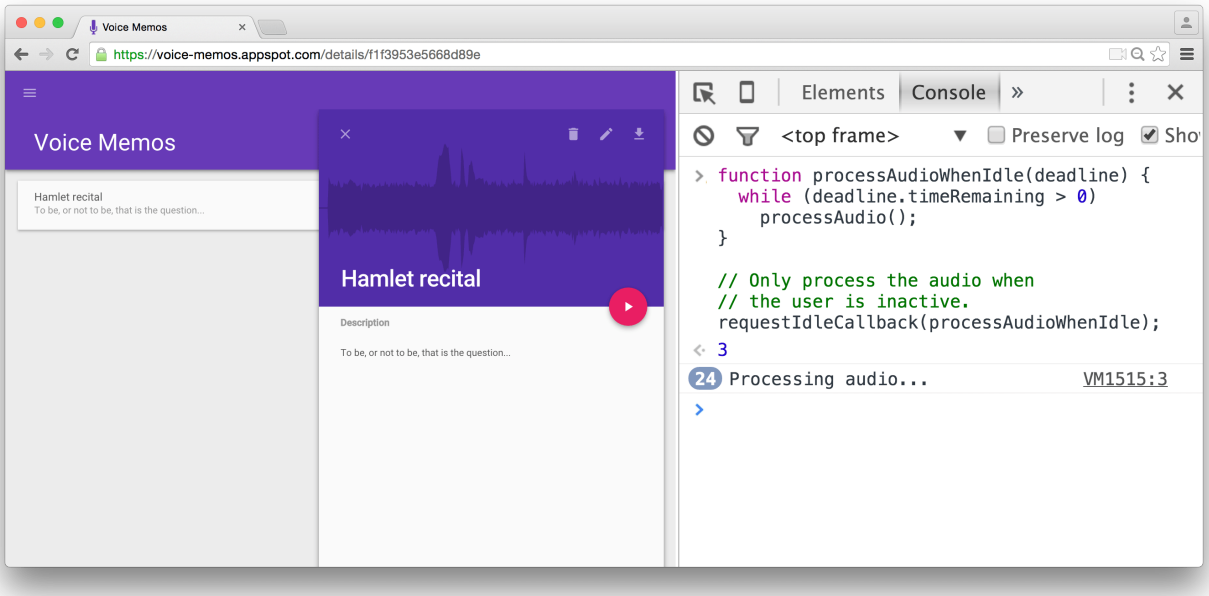
Using requestIdleCallback

By Paul Lewis

Paul is a Design and Perf Advocate

Many sites and apps have a lot of scripts to execute. Your JavaScript often needs to be run as soon as possible, but at the same time you don't want it to get in the user's way. If you send analytics data when the user is scrolling the page, or you append elements to the DOM while they happen to be tapping on the button, your web app can become unresponsive, resulting in a poor user experience.



The good news is that there's now an API that can help: **`requestIdleCallback`**. In the same way that adopting **`requestAnimationFrame`** allowed us to schedule animations properly and maximize our chances of hitting 60fps, **`requestIdleCallback`** will schedule work when there is free time at the end of a frame, or when the user is inactive. This means that there's an opportunity to do your work without getting in the user's way. It's available as of Chrome 47, so you can give it a whirl today by using Chrome Canary! It is an *experimental feature*, and the spec is still in flux, so things could change in the future.

Why should I use requestIdleCallback?

Scheduling non-essential work yourself is very difficult to do. It's impossible to figure out exactly how much frame time remains because after **`requestAnimationFrame`** callbacks execute there are style calculations, layout, paint, and other browser internals that need to run. A home-rolled solution can't account for any of those. In order to be sure that a user *isn't* interacting in some way you would also need to attach listeners to every kind of interaction event (**`scroll`**, **`touch`**, **`click`**), even if you don't need them for functionality, *just* so that you can be absolutely sure that the user isn't interacting. The browser, on the other hand, knows exactly how much time is available at the end of the frame, and if the user is interacting, and so through **`requestIdleCallback`** we gain an API that allows us to make use of any spare time in the most efficient way possible.

Let's take a look at it in a little more detail and see how we can make use of it.

Checking for requestIdleCallback

It's early days for **`requestIdleCallback`**, so before using it you should check that it's available for use:

```
if ('requestIdleCallback' in window) {
  // Use requestIdleCallback to schedule work.
} else {
  // Do what you'd do today.
}
```

You can also shim its behavior, which requires falling back to **`setTimeout`**:

```
window.requestIdleCallback =
  window.requestIdleCallback ||
  function (cb) {
    var start = Date.now();
    return setTimeout(function () {
      cb({
        didTimeout: false,
        timeRemaining: function () {
          return Math.max(0, 50 - (Date.now() - start));
        }
      });
    }, 1);
  }

window.cancelIdleCallback =
  window.cancelIdleCallback ||
  function (id) {
```

```
    clearTimeout(id);
  }
```

Using `setTimeout` isn't great because it doesn't know about idle time like `requestIdleCallback` does, but since you would call your function directly if `requestIdleCallback` wasn't available, you are no worse off shimming in this way. With the shim, should `requestIdleCallback` be available, your calls will be silently redirected, which is great.

For now, though, let's assume that it exists.

Using requestIdleCallback

Calling `requestIdleCallback` is very similar to `requestAnimationFrame` in that it takes a callback function as its first parameter:

```
requestIdleCallback(myNonEssentialWork);
```

When `myNonEssentialWork` is called, it will be given a `deadline` object which contains a function which returns a number indicating how much time remains for your work:

```
function myNonEssentialWork (deadline) {
  while (deadline.timeRemaining() > 0)
    doWorkIfNeeded();
}
```

The `timeRemaining` function can be called to get the latest value. When `timeRemaining()` returns zero you can schedule another `requestIdleCallback` if you still have more work to do:

```
function myNonEssentialWork (deadline) {
  while (deadline.timeRemaining() > 0 && tasks.length > 0)
    doWorkIfNeeded();

  if (tasks.length > 0)
    requestIdleCallback(myNonEssentialWork);
}
```

Guaranteeing your function is called

What do you do if things are really busy? You might be concerned that your callback may never be called. Well, although **`requestIdleCallback`** resembles **`requestAnimationFrame`**, it also differs in that it takes an optional second parameter: an options object with **a timeout** property. This timeout, if set, gives the browser a time in milliseconds by which it must execute the callback:

```
// Wait at most two seconds before processing events.
requestIdleCallback(processPendingAnalyticsEvents, { timeout: 2000 });
```

If your callback is executed because of the timeout firing you'll notice two things:

- **`timeRemaining()`** will return zero.
- The **`didTimeout`** property of the **`deadline`** object will be true.

If you see that the **`didTimeout`** is true, you will most likely just want to run the work and be done with it:

```
function myNonEssentialWork (deadline) {

  // Use any remaining time, or, if timed out, just run through the tasks.
  while ((deadline.timeRemaining() > 0 || deadline.didTimeout) &&
         tasks.length > 0)
    doWorkIfNeeded();

  if (tasks.length > 0)
    requestIdleCallback(myNonEssentialWork);
}
```

Because of the potential disruption this timeout can cause to your users (the work could cause your app to become unresponsive or janky) be cautious with setting this parameter. Where you can, let the browser decide when to call the callback.

Using requestIdleCallback for sending analytics data

Let's take a look using **`requestIdleCallback`** to send analytics data. In this case, we probably would want to track an event like – say – tapping on a navigation menu. However, because they normally animate onto the screen, we will want to avoid sending this event to Google Analytics immediately. We will create an array of events to send and request that they get sent at some point in the future:

```
var eventsToSend = [];
```

```
function onNavOpenClick () {

  // Animate the menu.
  menu.classList.add('open');

  // Store the event for later.
  eventsToSend.push(
    {
      category: 'button',
      action: 'click',
      label: 'nav',
      value: 'open'
    });

  schedulePendingEvents();
}
```

Now we will need to use **`requestIdleCallback`** to process any pending events:

```
function schedulePendingEvents() {

  // Only schedule the rIC if one has not already been set.
  if (isRequestIdleCallbackScheduled)
    return;

  isRequestIdleCallbackScheduled = true;

  if ('requestIdleCallback' in window) {
    // Wait at most two seconds before processing events.
    requestIdleCallback(processPendingAnalyticsEvents, { timeout: 2000 });
  } else {
    processPendingAnalyticsEvents();
  }
}
```

Here you can see I've set a timeout of 2 seconds, but this value would depend on your application. For analytics data, it makes sense that a timeout would be used to ensure data is reported in a reasonable timeframe rather than just at some point in the future.

Finally we need to write the function that **`requestIdleCallback`** will execute.

```
function processPendingAnalyticsEvents (deadline) {

  // Reset the boolean so future rICs can be set.
  isRequestIdleCallbackScheduled = false;

  // If there is no deadline, just run as long as necessary.
```

```
  // This will be the case if requestIdleCallback doesn't exist.
  if (typeof deadline === 'undefined')
    deadline = { timeRemaining: function () { return Number.MAX_VALUE } };

  // Go for as long as there is time remaining and work to do.
  while (deadline.timeRemaining() > 0 && eventsToSend.length > 0) {
    var evt = eventsToSend.pop();

    ga('send', 'event',
        evt.category,
        evt.action,
        evt.label,
        evt.value);
  }

  // Check if there are more events still to send.
  if (eventsToSend.length > 0)
    schedulePendingEvents();
}
```

For this example I assumed that if **`requestIdleCallback`** didn't exist that the analytics data should be sent immediately. In a production application, however, it would likely be better to delay the send with a timeout to ensure it doesn't conflict with any interactions and cause jank.

Using requestIdleCallback to make DOM changes

Another situation where **`requestIdleCallback`** can really help performance is when you have non-essential DOM changes to make, such as adding items to the end of an ever-growing, lazy-loaded list. Let's look at how **`requestIdleCallback`** actually fits into a typical frame.

It's possible that the browser will be too busy to run any callbacks in a given frame, so you shouldn't expect that there will be *any* free time at the end of a frame to do any more work. That makes it different to something like `setImmediate`, which *does* run per frame.

If the callback *is* fired at the end of the frame, it will be scheduled to go after the current frame has been committed, which means that style changes will have been applied, and, importantly, layout calculated. If we make DOM changes inside of the idle callback, those layout calculations will be invalidated. If there are any kind of layout reads in the next frame, e.g. `getBoundingClientRect`, `clientWidth`, etc, the browser will have to perform a Forced Synchronous Layout, which is a potential performance bottleneck.

Another reason not trigger DOM changes in the idle callback is that the time impact of changing the DOM is unpredictable, and as such we could easily go past the deadline the browser provided.

The best practice is to only make DOM changes inside of a `requestAnimationFrame` callback, since it is scheduled by the browser with that type of work in mind. That means that our code will need to use a document fragment, which can then be appended in the next `requestAnimationFrame` callback. If you are using a VDOM library, you would use `requestIdleCallback` to make changes, but you would *apply* the DOM patches in the next `requestAnimationFrame` callback, not the idle callback.

So with that in mind, let's take a look at the code:

```
function processPendingElements (deadline) {

  // If there is no deadline, just run as long as necessary.
  if (typeof deadline === 'undefined')
    deadline = { timeRemaining: function () { return Number.MAX_VALUE } };

  if (!documentFragment)
    documentFragment = document.createDocumentFragment();

  // Go for as long as there is time remaining and work to do.
  while (deadline.timeRemaining() > 0 && elementsToAdd.length > 0) {

    // Create the element.
    var elToAdd = elementsToAdd.pop();
    var el = document.createElement(elToAdd.tag);
    el.textContent = elToAdd.content;

    // Add it to the fragment.
    documentFragment.appendChild(el);

    // Don't append to the document immediately, wait for the next
```

```
    // requestAnimationFrame callback.
    scheduleVisualUpdateIfNeeded();
  }

  // Check if there are more events still to send.
  if (elementsToAdd.length > 0)
    scheduleElementCreation();
}
```

Here I create the element and use the `textContent` property to populate it, but chances are your element creation code would be more involved! After creating the element `scheduleVisualUpdateIfNeeded` is called, which will set up a single `requestAnimationFrame` callback that will, in turn, append the document fragment to the body:

```
function scheduleVisualUpdateIfNeeded() {

  if (isVisualUpdateScheduled)
    return;

  isVisualUpdateScheduled = true;

  requestAnimationFrame(appendDocumentFragment);
}

function appendDocumentFragment() {
  // Append the fragment and reset.
  document.body.appendChild(documentFragment);
  documentFragment = null;
}
```

All being well we will now see much less jank when appending items to the DOM. Excellent!

FAQ

- **Is there a polyfill?** Sadly not, but there is a shim if you want to have a transparent redirection to `setTimeout`. The reason this API exists is because it plugs a very real gap in the web platform. Inferring a lack of activity is difficult, but no JavaScript APIs exist to determine the amount of free time at the end of the frame, so at best you have to make guesses. APIs like `setTimeout`, `setInterval`, or `setImmediate` can be used to schedule work, but they are not timed to avoid user interaction in the way that `requestIdleCallback` is.
- **What happens if I overrun the deadline?** If `timeRemaining()` returns zero, but you opt to run for longer, you can do so without fear of the browser halting your work. However,

the browser gives you the deadline to try and ensure a smooth experience for your users, so unless there's a very good reason, you should always adhere to the deadline.

- **Is there maximum value that `timeRemaining()` will return?** Yes, it's currently 50ms. When trying to maintain a responsive application, all responses to user interactions should be kept under 100ms. Should the user interact the 50ms window should, in most cases, allow for the idle callback to complete, and for the browser to respond to the user's interactions. You may get multiple idle callbacks scheduled back-to-back (if the browser determines that there's enough time to run them).
- **Is there any kind of work I shouldn't do in a requestIdleCallback?** Ideally the work you do should be in small chunks (microtasks) that have relatively predictable characteristics. For example, changing the DOM in particular will have unpredictable execution times, since it will trigger style calculations, layout, painting, and compositing. As such you should only make DOM changes in a `requestAnimationFrame` callback as suggested above. Another thing to be wary of is resolving (or rejecting) Promises, as the callbacks will execute immediately after the idle callback has finished, even if there is no more time remaining.
- **Will I always get a `requestIdleCallback` at the end of a frame?** No, not always. The browser will schedule the callback whenever there is free time at the end of a frame, or in periods where the user is inactive. You shouldn't expect the callback to be called per frame, and if you require it to run within a given timeframe you should make use of the timeout.
- **Can I have multiple `requestIdleCallback` callbacks?** Yes, you can, very much as you can have multiple `requestAnimationFrame` callbacks. It's worth remembering, though, that if your first callback uses up the time remaining during its callback then there will be no more time left for any other callbacks. The other callbacks will then have to wait until the browser is next idle before they can be run. Depending on the work you're trying to get done, it may be better to have a single idle callback and divide the work in there. Alternatively you can make use of the timeout to ensure that no callbacks get starved for time.
- **What happens if I set a new idle callback inside of another?** The new idle callback will be scheduled to run as soon as possible, starting from the *next* frame (rather than the current one).

Idle on!

`requestIdleCallback` is an awesome way to make sure you can run your code, but without getting in the user's way. It's simple to use, and very flexible. It's still early days, though, and

the spec isn't fully settled, so any feedback you have is welcome.

Check it out in Chrome Canary, give it a spin for your projects, and let us know how you get on!

Last updated July 2, 2018.