10th August 2019

Fetch: Abort

As we know, `fetch` returns a promise. And JavaScript generally has no concept of “aborting” a promise. So how can we abort a `fetch` ?

There’s a special built-in object for such purposes: `AbortController` , that can be used to abort not only `fetch` , but other asynchronous tasks as well.

The usage is pretty simple:

- Step 1: create a controller:

  ```
  let controller = new AbortController();
  ```

  A controller is an extremely simple object.

  - It has a single method `abort()` , and a single property `signal` .
  - When `abort()` is called:
    - `abort` event triggers on `controller.signal`
    - `controller.signal.aborted` property becomes `true` .

  All parties interested to learn about `abort()` call set listeners on `controller.signal` to track it.

  Like this (without `fetch` yet):

  ```
  let controller = new AbortController();
  let signal = controller.signal;

  // triggers when controller.abort() is called
  signal.addEventListener('abort', () => alert("abort!"));

  controller.abort(); // abort!

  alert(signal.aborted); // true
  ```

- Step 2: pass the `signal` property to `fetch` option:

  ```
  let controller = new AbortController();
  fetch(url, {
    signal: controller.signal
  });
  ```

The `fetch` method knows how to work with `AbortController`, it listens to `abort` on `signal`.

- Step 3: to abort, call `controller.abort()`:

  ```
  controller.abort();
  ```

  We're done: `fetch` gets the event from `signal` and aborts the request.

When a fetch is aborted, its promise rejects with an error `AbortError`, so we should handle it, e.g. in `try..catch`:

```
// abort in 1 second
let controller = new AbortController();
setTimeout(() => controller.abort(), 1000);

try {
  let response = await fetch('/article/fetch-abort/demo/hang', {
    signal: controller.signal
  });
} catch(err) {
  if (err.name == 'AbortError') { // handle abort()
    alert("Aborted!");
  } else {
    throw err;
  }
}
```

`AbortController` is scalable, it allows to cancel multiple fetches at once.

For instance, here we fetch many `urls` in parallel, and the controller aborts them all:

```
let urls = [...]; // a list of urls to fetch in parallel

let controller = new AbortController();

let fetchJobs = urls.map(url => fetch(url, {
  signal: controller.signal
}));

let results = await Promise.all(fetchJobs);

// if controller.abort() is called from elsewhere,
// it aborts all fetches
```

If we have our own asynchronous jobs, different from `fetch`, we can use a single `AbortController` to stop those, together with fetches.

We just need to listen to its `abort` event:

```
let urls = [...];
let controller = new AbortController();

```

```
let ourJob = new Promise((resolve, reject) => { // our task
  ...
  controller.signal.addEventListener('abort', reject);
});

let fetchJobs = urls.map(url => fetch(url, { // fetches
  signal: controller.signal
}));

// Wait for fetches and our task in parallel
let results = await Promise.all([...fetchJobs, ourJob]);

// if controller.abort() is called from elsewhere,
// it aborts all fetches and ourJob
```

So `AbortController` is not only for `fetch`, it's a universal object to abort asynchronous tasks, and `fetch` has built-in integration with it.

Previous lesson Next lesson

Share Tutorial map

Comments