README.md

Cancelable Async Flows (CAF)

build passing | npm package 15.0.1 | coverage 100% | modules ESM+UMD+CJS | license MIT

CAF (/ˈkahf/) is a wrapper for `function*` generators that treats them like `async function`s, but with support for external cancellation via tokens. In this way, you can express flows of synchronous-looking asynchronous logic that are still cancelable (**C**ancelable **A**sync **F**lows).

Also included is `CAG(..)`, for alternately wrapping `function*` generators to emulate ES2018 async-generators (`async function *`).

Environment Support

This library uses ES2018 features. If you need to support environments prior to ES2018, transpile it first (with Babel, etc).

At A Glance

CAF (**C**ancelable **A**sync **F**lows) wraps a `function*` generator so it looks and behaves like an `async function`, but that can be externally canceled using a cancellation token:

```js
var token = new CAF.cancelToken();

// wrap a generator to make it look like a normal async
// function that when called, returns a promise.
var main = CAF( function *main(signal,url){
    var resp = yield ajax( url );

    // want to be able to cancel so we never get here?!?
    console.log( resp );
    return resp;
} );

// run the wrapped async-looking function, listen to its
// returned promise
main( token.signal, "http://some.tld/other" )
.then( onResponse, onCancelOrError );

// only wait 5 seconds for the ajax request!
setTimeout( function onElapsed(){
    token.abort( "Request took too long!" );
}, 5000 );
```

Create a cancellation token (via `new CAF.cancelToken()`) to pass into your wrapped `function*` generator, and then if you cancel the token, the `function*` generator will abort itself immediately, even if it's presently waiting on a promise to resolve.

The generator receives the cancellation token's `signal`, so from inside it you can call another `function*` generator via **CAF** and pass along that shared `signal`. In this way, a single cancellation signal can cascade across and cancel all the **CAF**-wrapped functions in a chain of execution:

```js
var token = new CAF.cancelToken();

var one = CAF( function *one(signal,v){
    return yield two( signal, v );
} );

var two = CAF( function *two(signal,v){
    return yield three( signal, v );
} );

var three = CAF( function *three(signal,v){
    return yield ajax( `http://some.tld/?v=${v}` );
} );

one( token.signal, 42 );

// only wait 5 seconds for the request!
setTimeout( function onElapsed(){
    token.abort( "Request took too long!" );
}, 5000 );
```

In this snippet, `one(..)` calls and waits on `two(..)`, `two(..)` calls and waits on `three(..)`, and `three(..)` calls and waits on `ajax(..)`. Because the same cancellation token is used for the 3 generators, if `token.abort()` is executed while they're all still paused, they will all immediately abort.

Note: The cancellation token has no effect on the actual `ajax(..)` call itself here, since that utility ostensibly doesn't provide cancellation capability; the Ajax request itself would still run to its completion (or error or whatever). We've only canceled the `one(..)`, `two(..)`, and `three(..)` functions that were waiting to process its response. See `AbortController(..)` and Manual Cancellation Signal Handling below for addressing this limitation.

CAG: Cancelable Async ~~Flows~~ *Generators*

ES2018 added "async generators", which is a pairing of `async function` and `function*` -- so you can use `await` and `yield` in the same function, `await` for unwrapping a promise, and `yield` for pushing a value out. An async-generator (`async function * f(..) { .. }`), like regular iterators, is designed to be sequentially iterated, but using the "async iteration" protocol.

For example, in ES2018:

```
async function *stuff(urls) {
    for (let url of urls) {
        let resp = await fetch(url);  // await a promise
        yield resp.json();  // yield a value (even a promise for a value)
    }
}

// async-iteration loop
for await (let v of stuff(assetURLs)) {
    console.log(v);
}
```

In the same way that `CAF(..)` emulates an `async..await` function with a `function*` generator, the `CAG(..)` utility emulates an async-generator with a normal `function*` generator. You can cancel an async-iteration early (even if it's currently waiting internally on a promise) with a cancellation token.

You can also synchronously force-close an async-iterator by calling the `return(..)` on the iterator. With native async-iterators, `return(..)` is not actually synchronous, but `CAG(..)` patches this to allow synchronous closing.

Instead of `yield` ing a promise the way you do with `CAF(..)`, you use a provided `pwait(..)` function with `yield`, like `yield pwait(somePromise)`. This allows a `yield ..value..` expression for pushing out a value through the iterator, as opposed to `yield pwait(..value..)` to locally wait for the promise to resolve. To emulate a `yield await ..value..` expression (common in async-generators), you use two `yield` s together: `yield yield pwait(..value..)`.

For example:

```
// NOTE: this is CAG(..), not to be confused with CAF(..)
var stuff = CAG(function *stuff({ signal, pwait },urls){
    for (let url of urls) {
        let resp = yield pwait(fetch(url,{ signal }));  // await a promise
        yield resp.json();  // yield a value (even a promise for a value)
    }
});

var timeout = CAF.timeout(5000,"That's enough results!");
var it = stuff(timeout,assetURLs);

cancelBtn.addEventListener("click",() => it.return("Stop!"),false);

// async-iteration loop
for await (let v of it) {
    console.log(v);
}
```

In this snippet, the `stuff(..)` async-iteration can either be canceled if the 5-second timeout expires before iteration is complete, or the click of the cancel button can force-close the iterator early. The difference between them is that token cancellation would result in an exception bubbling out (to the consuming loop), whereas calling `return(..)` will simply cleanly close the iterator (and halt the loop) with no exception.

Background/Motivation

Generally speaking, an `async function` and a `function*` generator (driven with a generator-runner) look very similar. For that reason, most people just prefer the `async function` form since it's a little nicer syntax and doesn't require a library for the runner.

However, there are limitations to `async function` s that come from having the syntax and engine make implicit assumptions that otherwise would have been handled by a `function*` generator runner.

One unfortunate limitation is that an `async function` cannot be externally canceled once it starts running. If you want to be able to cancel it, you have to intrusively modify its definition to have it consult an external value source -- like a boolean or promise -- at each line that you care about being a potential cancellation point. This is ugly and error-prone.

`function*` generators by contrast can be aborted at any time, using the iterator's `return(..)` method and/or by just not resuming the generator iterator instance with `next()`. But the downside of using `function*` generators is either needing a runner utility or the repetitive boilerplate of handling the iterator manually.

CAF provides a useful compromise: a `function*` generator that can be called like a normal `async function`, but which supports a cancellation token.

The `CAF(..)` utility wraps a `function*` generator with a normal promise-returing function, just as if it was an `async function`. Other than minor syntactic aesthetics, the major observable difference is that a **CAF**-wrapped function must be provided a cancellation token's `signal` as its first argument, with any other arguments passed subsequent, as desired.

By contrast, the `CAG(..)` utility wraps a `function*` generator as an ES2018 async-generator (`async function *`) that respects the native async-iteration protocol. Instead of `await`, you use `yield pwait(..)` in these emulated async-generators.

Overview

In the following snippet, the two functions are essentially equivalent; `one(..)` is an actual `async function`, whereas `two(..)` is a wrapper around a generator, but will behave like an async function in that it also returns a promise:

```
async function one(v) {
    await delay( 100 );
    return v * 2;
}

var two = CAF( function *two(signal,v){
    yield delay( 100 );
    return v * 2;
} );
```

Both `one(..)` and `two(..)` can be called directly with argument(s), and both return a promise for their completion:

```
one( 21 )
.then( console.log, console.error );   // 42

var token = new CAF.cancelToken();

two( token.signal, 21 )
.then( console.log, console.error );   // 42
```

If `token.abort(..)` is executed while `two(..)` is still running, the `signal` 's promise will be rejected. If you pass a cancellation reason (any value, but typically a string) to `token.abort(..)` , that will be the promise rejection reason:

```
two( token, 21 )
.then( console.log, console.error );    // Took too long!

token.abort( "Took too long!" );
```

Delays & Timeouts

One of the most common use-cases for cancellation of an async task is because too much time passes and a timeout threshold is passed.

As shown earlier, you can implement that sort of logic with a `cancelToken()` instance and a manual call to the environment's `setTimeout(..)` . However, there are some subtle but important downsides to doing this kind of thing manually. These downsides are harder to spot in the browser, but are more obvious in Node.js

Consider this code:

```
function delay(ms) {
    return new Promise( function c(res){
        setTimeout( res, ms );
    } );
}

var token = new CAF.cancelToken();

var main = CAF( function *main(signal,ms){
    yield delay( ms );
    console.log( "All done!" );
} );

main( token.signal, 100 );

// only wait 5 seconds for the request!
delay( 5000 ).then( function onElapsed(){
    token.abort( "Request took too long!" );
} );
```

The `main(..)` function delays for `100` ms and then completes. But there's no logic that clears the timeout set from `delay( 5000 )` , so it will continue to hold pending until that amount of time expires.

Of course, the `token.abort(..)` call at that point is moot, and is thus silently ignored. But the problem is the timer still running, which keeps a Node.js process alive even if the rest of the program has completed. The symptoms of this would be running a Node.js program from the command line and observing it "hang" for a bit at the end instead of exiting right away. Try the above code to see this in action.

There's two complications that make avoiding this downside tricky:

1. The `delay(..)` helper shown, which is a promisified version of `setTimeout(..)`, is basically what you can produce by using Node.js's `util.promisify(..)` against `setTimeout(..)`. However, that timer itself is not cancelable. You can't access the timer handle (return value from `setTimeout(..)`) to call `clearTimeout(..)` on it. So, you can't stop the timer early even if you wanted to.

2. If instead you set up your own timer externally, you need to keep track of the timer's handle so you can call `clearTimeout(..)` if the async task completes successfully before the timeout expires. This is manual and error-prone, as it's far too easy to forget.

Instead of inventing solutions to these problems, **CAF** provides two utilities for managing cancelable delays and timeout cancellations: `CAF.delay(..)` and `CAF.timeout(..)`.

`CAF.delay(..)`

What we need is a promisified `setTimeout(..)`, like `delay(..)` we saw in the previous section, but that can still be canceled. `CAF.delay(..)` provides us such functionality:

```
var discardTimeout = new CAF.cancelToken();

// a promise that waits 5 seconds
CAF.delay( discardTimeout.signal, 5000 )
.then(
    function onElapsed(msg){
        // msg: "delayed: 5000"
    },
    function onInterrupted(reason){
        // reason: "delay (5000) interrupted"
    }
);
```

As you can see, `CAF.delay(..)` receives a cancellation token signal to cancel the timeout early when needed. If you need to cancel the timeout early, abort the cancellation token:

```
discardTimeout.abort();     // cancel the `CAF.delay()` timeout
```

The promise returned from `CAF.delay(..)` is fulfilled if the full time amount elapses, with a message such as `"delayed: 5000"`. But if the timeout is aborted via the cancellation token, the promise is rejected with a reason like `"delay (5000) interrupted"`.

Passing the cancellation token to `CAF.delay(..)` is optional; if omitted, `CAF.delay(..)` works just like a regular promisified `setTimeout(..)`:

```
// promise that waits 200 ms
CAF.delay( 200 )
.then( function onElapsed(){
    console.log( "Some time went by!" );
} );
```

`CAF.timeout(..)`

While `CAF.delay(..)` provides a cancelable timeout promise, it's still overly manual to connect the dots between a **CAF**-wrapped function and the timeout-abort process. **CAF** provides `CAF.timeout(..)` to streamline this common use-case:

```js
var timeoutToken = CAF.timeout( 5000, "Took too long!" );

var main = CAF( function *main(signal,ms){
    yield CAF.delay( signal, ms );
    console.log( "All done!" );
} );

main( timeoutToken, 100 );   // NOTE: pass the whole token, not just the .signal !!
```

`CAF.timeout(..)` creates an instance of `cancellationToken(..)` that's set to `abort()` after the specified amount of time, optionally using the cancellation reason you provide.

Note that you should pass the full `timeoutToken` token to the **CAF**-wrapped function (`main(..)`), instead of just passing `timeoutToken.signal`. By doing so, **CAF** wires the token and the **CAF**-wrapped function together, so that each one stops the other, whichever one happens first. No more hanging timeouts!

Also note that `main(..)` still receives just the `signal` as its first argument, which is suitable to pass along to other cancelable async functions, such as `CAF.delay(..)` as shown.

`timeoutToken` is a regular cancellation token as created by `CAF.cancelToken()`. As such, you can call `abort(..)` on it directly, if necessary. You can also access `timeoutToken.signal` to access its signal, and `timeoutToken.signal.pr` to access the promise that's rejected when the signal is aborted.

`finally { .. }`

`finally` clauses are often attached to a `try { .. }` block wrapped around the entire body of a function, even if there's no `catch` clause defined. The most common use of this pattern is to define some clean-up operations to be performed after the function is finished, whether that was from normal completion or an early termination (such as uncaught exceptions, or cancellation).

Canceling a **CAF**-wrapped `function*` generator that is paused causes it to abort right away, but if there's a pending `finally {..}` clause, it will always still have a chance to run.

```
var token = new CAF.cancelToken();

var main = CAF( function *main(signal,url){
    try {
        return yield ajax( url );
    }
    finally {
        // perform some clean-up operations
    }
} );

main( token.signal, "http://some.tld/other" )
.catch( console.log );     // 42 <-- not "Stopped!"

token.abort( "Stopped!" );
```

Moreover, a `return` of any non-`undefined` value in a pending `finally {..}` clause will override the promise rejection reason:

```
var token = new CAF.cancelToken();

var main = CAF( function *main(signal,url){
    try {
        return yield ajax( url );
    }
    finally {
        return 42;
    }
} );

main( token.signal, "http://some.tld/other" )
.catch( console.log );     // 42 <-- not "Stopped!"

token.abort( "Stopped!" );
```

Whatever value is passed to `abort(..)`, if any, is normally set as the overall promise rejection reason. But in this case, `return 42` overrides the `"Stopped!"` rejection reason.

`signal.aborted` and `signal.reason`

Standard `AbortSignal` instances have an `aborted` boolean property that's set to `true` once the signal is aborted. Recently, `AbortSignal` was extended to include a `reason` property. Prior to that change,**CAF**was manually patching `signal` with a `reason` property, but now**CAF**respects the `reason` that's built-in to `AbortSignal` instances, if the environment supports it.

To set the `reason` for an abort-signal firing, pass a value to the `AbortController` 's `abort(..)` call.

By checking the `signal.aborted` flag in a `finally` clause, you can determine whether the function was canceled, and then additionally access the `signal.reason` to determine more specific context information about why the cancellation occurred. This allows you to perform different clean-up operations depending on cancellation or normal completion:

```
var token = new CAF.cancelToken();

var main = CAF( function *main(signal,url){
    try {
        return yield ajax( url );
    }
    finally {
        if (signal.aborted) {
            console.log( `Cancellation reason: ${ signal.reason }` );
            // perform cancellation-specific clean-up operations
        }
        else {
            // perform non-cancellation clean-up operations
        }
    }
} );

main( token.signal, "http://some.tld/other" );
// Cancellation reason: Stopped!

token.abort( "Stopped!" );
```

Memory Cleanup With `discard()`

A cancellation token from**CAF**includes a `discard()` method that can be called at any time to fully unset any internal state in the token to allow proper GC (garbage collection) of any attached resources.

When you are sure you're fully done with a cancellation token, it's a good idea to call `discard()` on it, and then unset the variable holding that reference:

```
var token = new CAF.cancelToken();

// later
```

```
token.discard();
token = null;
```

Once a token has been `discard()` ed, no further calls to `abort(..)` will be effective -- they will silently be ignored.

`AbortController(..)`

`CAF.cancelToken(..)` instantiates [`AbortController` , the DOM standard] for canceling/aborting operations like `fetch(..)` calls. As such, a **CAF** cancellation token's `signal` can be passed directly to a DOM method like `fetch(..)` to control its cancellation:

```
var token = new CAF.cancelToken();

var main = CAF(function *main(signal,url) {
    var resp = yield fetch( url, { signal } );

    console.log( resp );
    return resp;
});

main( token.signal, "http://some.tld/other" )
.catch( console.log );   // "Stopped!"

token.abort( "Stopped!" );
```

`CAF.cancelToken(..)` can optionally receive an already instantiated `AbortController` , though there's rarely a reason to do so:

```
var ac = new AbortController();
var token = new CAF.cancelToken(ac);
```

Also, if you pass a raw `AbortController` instance into a **CAF**-wrapped function, it's automatically wrapped into a `CAF.cancelToken(..)` instance:

```
var main = CAF(function *main(signal,url) {
    var resp = yield fetch( url, { signal } );

    console.log( resp );
    return resp;
});

var ac = new AbortController();
main( ac, "http://some.tld/other" )
.catch( () => console.log("Stopped!") );   // "Stopped!"

ac.abort();
```

`AbortController()` Polyfill

If `AbortController` is not defined in the environment, use this polyfill to define a compatible stand-in. The polyfill is included in the `dist/` directory.

If you load **CAF** in Node using its CJS format (with `require(..)`) and use the main package entry point (`require("caf")`), the polyfill is automatically loaded (in the `global` namespace). If you don't use this entry point, but instead load something more directly, like `require("caf/core")` or `require("caf/cag")` , then you need to manually load the polyfill first:

```
require("/path/to/caf/dist/abortcontroller-polyfill-only.js");

var CAF = require("caf/core");
var CAG = require("caf/cag");
```

When using the ESM format of **CAF** in Node, the polyfill is *not* loaded automatically. Node 15/16+ includes `AbortController` natively, but in prior versions of Node (12-14) while using the ESM format, you need to manually `require(..)` the polyfill (before `import` ing **CAF**) like this:

```
import { createRequire } from "module";
const require = createRequire(import.meta.url);
require("/path/to/caf/dist/abortcontroller-polyfill-only.js");

import CAF from "caf/core";
// ..
```

Be aware that if any environment needs this polyfill, utilities in that environment like `fetch(..)` won't *know* about `AbortController` so they won't recognize or respond to it. They won't break in its presence, just won't use it.

Manual Cancellation Signal Handling

Even if you aren't calling a cancellation signal-aware utility (like `fetch(..)`), you can still manually respond to the cancellation `signal` via its attached promise:

```
var token = new CAF.cancelToken();

var main = CAF( function *main(signal,url){
    // listen to the signal's promise rejection directly
    signal.pr.catch( reason => {
        // reason == "Stopped!"
    } );

    var resp = yield ajax( url );
```

```
    console.log( resp );
    return resp;
} );

main( token.signal, "http://some.tld/other" )
.catch( console.log );   // "Stopped!"

token.abort( "Stopped!" );
```

Note: The `catch(..)` handler inside of `main(..)` will still run, even though `main(..)` itself will be aborted at its waiting `yield` statement. If there was a way to manually cancel the `ajax(..)` call, that code should be placed in the `catch(..)` handler.

Even if you aren't running a **CAF**-wrapped function, you can still respond to the cancellation `signal`'s promise manually to affect flow control:

```
var token = new CAF.cancelToken();

// normal async function, not CAF-wrapped
async function main(signal,url) {
    try {
        var resp = await Promise.race( [
            ajax( url ),
            signal.pr       // listening to the cancellation
        ] );

        // this won't run if `signal.pr` rejects
        console.log( resp );
        return resp;
    }
    catch (err) {
        // err == "Stopped!"
    }
}

main( token.signal, "http://some.tld/other" )
.catch( console.log );   // "Stopped!"

token.abort( "Stopped!" );
```

Note: As discussed earlier, the `ajax(..)` call itself is not cancellation-aware, and is thus not being canceled here. But we *are* ending our waiting on the `ajax(..)` call. When `signal.pr` wins the `Promise.race(..)` race and creates an exception from its promise rejection, flow control jumps straight to the `catch (err) { .. }` clause.

Signal Combinators

You may want to combine two or more signals, similar to how you combine promises with `Promise.race(..)` and `Promise.all(..)`. **CAF** provides two corresponding helpers for this purpose:

```js
var timeout = CAF.timeout(5000,"Took too long!");
var canceled = new CAF.cancelToken();
var exit = new AbortController();

var anySignal = CAF.signalRace([
    timeout.signal,
    canceled.signal,
    exit.signal
]);

var allSignals = CAF.signalAll([
    timeout.signal,
    canceled.signal,
    exit.signal
]);

main( anySignal, "http://some.tld/other" );

// or

main( allSignals, "http://some.tld/other" );
```

`CAF.signalRace(..)` expects an array of one or more signals, and returns a new signal (`anySignal`) that will fire as soon as any of the constituent signals have fired.

`CAF.signalAll(..)` expects an array of one or more signals, and returns a new signal (`allSignals`) that will fire only once all of the constituent signals have fired.

Warning: This pattern (combining signals) has a potential downside. **CAF** typically cleans up timer-based cancel tokens to make sure resources aren't being wasted and programs aren't hanging with open timer handles. But in this pattern, `signalRace(..)` / `signalAll(..)` only receive reference(s) to the signal(s), not the cancel tokens themselves, so it cannot do the manual cleanup. In the above example, you should manually clean up the 5000ms timer by calling `timeout.abort()` if the operation finishes before that timeout has fired the cancellation.

Beware Of Token Reuse

Beware of creating a single cancellation token that is reused for separate chains of function calls. Unexpected results are likely, and they can be extremely difficult to debug.

As illustrated earlier, it's totally OK and intended that a single cancellation token `signal` be shared across all the functions in **one** chain of calls (`A -> B -> C`). But reusing the same token **across two or more chains of calls** (`A -> B -> C` ***and*** `D -> E -> F`) is asking for trouble.

Imagine a scenario where you make two separate `fetch(..)` calls, one after the other, and the second one runs too long so you cancel it via a timeout:

```js
var one = CAF( function *one(signal){
    signal.pr.catch( reason => {
        console.log( `one: ${reason}` );
    } );

    return yield fetch( "http://some.tld/", {signal} );
} );

var two = CAF( function *two(signal,v){
    signal.pr.catch( reason => {
        console.log( `two: ${reason}` );
    } );

    return yield fetch( `http://other.tld/?v=${v}`, {signal} );
} );

var token = CAF.cancelToken();

one( token.signal )
.then( function(v){
    // only wait 3 seconds for this request
    setTimeout( function(){
        token.abort( "Second response too slow." );
    }, 3000 );

    return two( token.signal, v );
} )
.then( console.log, console.error );

// one: Second response too slow.   <-- Oops!
// two: Second response too slow.
// Second response too slow.
```

When you call `token.abort(..)` to cancel the second `fetch(..)` call in `two(..)`, the `signal.pr.catch(..)` handler in `one(..)` still gets called, even though `one(..)` is already finished. That's why `"one: Second response too slow."` prints unexpectedly.

The underlying gotcha is that a cancellation token's `signal` has a single `pr` promise associated with it, and there's no way to reset a promise or "unregister" `then(..)` / `catch(..)` handlers attached to it once you don't need them anymore. So if you reuse the token, you're reusing the `pr` promise, and all registered promise handlers will be fired, even old ones you likely don't intend.

The above snippet illustrates this problem with `signal.pr.catch(..)`, but any of the other ways of listening to a promise -- such as `yield` / `await`, `Promise.all(..)` / `Promise.race(..)`, etc -- are all susceptible to the unexpected behavior.

The safe and proper approach is to always create a new cancellation token for each chain of **CAF**-wrapped function calls. For good measure, always unset any references to a token once it's no longer needed, and make sure to call `discard()` ; thus, you won't accidentally reuse the token, and the JS engine can properly GC (garbage collect) it.

Cycling Tokens

A common use case in managing async operations is when a currently pending operation needs to be canceled only because it's being replaced by a subsequent operation.

For example, imagine a button on a page that requests some remote data to display. If the user clicks the button again while a previous request is still pending, you can likely discard/cancel the previous request and start up a new fresh request in its place.

In these sorts of cases, you may find yourself "cycling" through cancellation tokens, where you store a reference to the current token, then when a new one is needed, the former token is aborted (to cancel all its chained operations) and replaced with the new token instance. This sort of logic is not too complex, but it does require keeping the token around across async operations, which unfortunately pollutes an outer scope.

This use case is common enough to warrant a standard helper shipped with this library to reduce the friction/impact of managing these cycles of tokens. **CAF** ships with `tokenCycle()` for this purpose:

```
// create a token cycle
var getReqToken = CAF.tokenCycle();

btn.addEventListener("click",function onClick(){
    // get a new cancellation token, and
    // cancel the previous token (if any)
    //
    // note: this function optionally takes a
    //   reason for aborting the previous token
    var cancelToken = getReqToken();

    requestUpdatedData(cancelToken,"my-data");
});
```

The `tokenCycle()` function creates a separate instance of the token cycle manager, so you can create as many independent cycles as your app needs. It returns a function (named `getReqToken()` in the above snippet) which, when called, will produce a new token and cancel the previous token (if one is pending). This function also **optionally** takes a single argument to use as the *reason* passed in to abort the previous token.

You can of course keep these tokens around and use them for other cancellation controls. But in that situation you likely don't need `tokenCycle()` . This helper is designed for the lightweight use case where you wouldn't otherwise need to keep the token other than to make sure the previous operation is canceled before being replaced with the new operation.

CAG: Emulating Async Generators

Where `CAF(..)` emulates a promise-returning `async function` using a generator, `CAG(..)` is provided to emulate an async-iterator returning async-generator (`async function*`).

Async iteration is similar to streams (or primitive observables), where the values are consumed asynchronously (typically using an ES2018 `for await (..)` loop):

```js
for await (let v of someAsyncGenerator()) {
    // ..
}

// or:
var it = someAsyncGenerator();
for await (let v of it) {
    // ..
}
```

For all the same reasons that `async function` s being non-cancelable is troublesome, async-generators are similarly susceptible. An async-generator can be "stuck" `await` ing internally on a promise, and the outer consuming code cannot do anything to force it to stop.

That's why `CAG(..)` is useful:

```js
// instead of:
async function *stuff(urls) {
    for (let url of urls) {
        let resp = await fetch(url);  // await a promise
        yield resp.json();  // yield a value (even a promise for a value)
    }
}

// do this:
var stuff = CAG(function *stuff({ signal, pwait },urls){
    for (let url of urls) {
        let resp = yield pwait(fetch(url,{ signal }));  // await a promise
        yield resp.json();  // yield a value (even a promise for a value)
    }
});
```

Like `CAF(..)` , functions wrapped by `CAG(..)` expect to receive a special value in their first parameter position. Here, the object is destructured to reveal it contains both the cancellation-token `signal` (as with `CAF(..)`) and the `pwait(..)` function, which enables emulating local `await ..promise..` expressions as `yield pwait(..promise..)` .

The return from a `CAG(..)` wrapped function is an async-iterator (exactly as if a real native async-generator had been invoked). As with `CAF(..)` values, the first argument passed should always be the mandatory cancellation token (or its signal):

```
var stuff = CAG(function *stuff(..){ .. });

var timeout = CAF.timeout(5000,"Took too long.");
var it = stuff(timeout);
```

The returned async-iterator (`it` above) can be iterated manually with `it.next(..)` calls -- each returns a promise for an iterator-result -- or more preferably with an ES2018 `for await (..)` loop:

```
var timeout = CAF.timeout(5000,"Took too long.");
var it = stuff(timeout);

var { value, done } = await it.next();
// ..do that repeatedly..

// or preferably:
for await (let value of it) {
    // ..
}
```

In addition to being able to `abort(..)` the cancellation token passed into a `CAG(..)` -wrapped generator, async-iterators also can be closed forcibly by calling their `return(..)` method.

```
var timeout = CAF.timeout(5000,"Took too long.");
var it = stuff(timeout);

// later (e.g. in a timer or event handler):
it.return("all done");
// Promise<{ value: "all done", done: true }>
```

Typically, the `return(..)` call on an async-iterator (from an async-generator) will have "wait" for the attached async-generator to be "ready" to be closed -- in case an `await promise` expression is currently pending. This means you cannot actually synchronously force-close them. But since `CAG(..)` emulates async-generators with regular sync-generators, this nuance is "fixed". For consistency, `return(..)` still returns a Promise, but it's an already-resolved promise with the associated iterator-result.

`CAG(..)` -wrapped functions also follow these behaviors of `CAF(..)` -wrapped functions:

- Aborting the cancellation token results in an exception (which can be trapped by `try..catch`) propagating out from the most recent `for await (..)` (or `it.next(..)`) consumption point.

- The `reason` provided when aborting a cancellation token is (by default) set as the exception that propagates out. This can be overriden by a `return ..` statement in a `finally` clause of the wrapped generator function.

Event Streams

One of the most common use cases for async iterators (aka, streams) is to subscribe to an event source (DOM element events, Node.js event emitters, etc) and iterate the received events.

CAG provides two helpers for event stream subscription: `onEvent(..)` and `onceEvent(..)`. As the name implies, `onEvent(..)` listens for all events, whereas `onceEvent(..)` will listen only for a single event to fire (and then close the stream and unsubscribe from the event emitter).

`onEvent(..)` returns an ES2018 async iterator, but `onceEvent(..)` returns a promise that resolves (with the event value, if any) when the event fires.

```
var cancel = new CAF.cancelToken();

var DOMReady = CAG.onceEvent(cancel,document,"DOMContentLoaded",/*evtOpts=*/false);
var clicks = CAG.onEvent(cancel,myBtn,"click",/*evtOpts=*/false);

// wait for this one-time event to fire
await DOMReady;

for await (let click of clicks) {
    console.log("Button clicked!");
}
```

`onEvent(..)` event subscriptions are lazy, meaning that they don't actually attach to the emitter element until the first attempt to consume an event (via `for await..of` or a manual `next(..)` call on the async iterator). So, in the above snippet, the `clicks` event stream is not yet subscribed to any click events that happen until the `for await..of` loop starts (e.g., while waiting for the prior `DOMReady` event to fire).

However, there may be cases where you want to force the event subscription to start early even before consuming its events. Use `start()` to do so:

```
var cancel = new CAF.cancelToken();

var clicks = CAG.onEvent(cancel,myBtn,"click",/*evtOpts=*/false);

// force eager listening to events
clicks.start();

// .. consume the stream later ..
```

Event streams internally buffer received events that haven't yet been consumed. This buffer grows unbounded, so responsible memory management implies always consuming events from a stream that is subscribed and actively receving events.

Once an event stream is closed (e.g., token cancellation, breaking out of a `for await..of` loop, manually calling `return(..)` on the async iterator), the underlying event is unsubscribed.

npm Package

npm package 15.0.1 modules ESM+UMD+CJS

To install this package from `npm`:

```
npm install caf
```

IMPORTANT: The **CAF** library relies on `AbortController` being present in the JS environment. If the environment does not already define `AbortController` natively, **CAF** needs a polyfill for `AbortController`. In some cases, the polyfill is automatically loaded, and in other cases it must be manually required/imported. See the linked section for more details.

As of version 12.0.0, the package is available as an ES Module (in addition to CJS/UMD), and can be imported as so:

```
// named imports
import { CAF, CAG } from "caf";

// or, default imports:
import CAF from "caf/core";
import CAG from "caf/cag";
```

Note: Starting in version 15.0.0, the (somewhat confusing) ESM specifier `"caf/caf"` (which imports **only** `CAF` as a default-import) has been deprecated and will eventually be removed. Use `"caf/core"` to default-import only the `CAF` module, or use just `"caf"` for named imports (`{ CAF, CAG }`).

Also Note: Starting in version 11.x, **CAF** was also available in ESM format, but required an ESM import specifier segment `/esm` in **CAF** `import` paths. As of version 15.0.0, this has been removed, in favor of unified import specifier paths via Node Conditional Exports. For ESM `import` statements, always use the specifier style `"caf"` or `"caf/cag"`, instead of `"caf/esm"` and `"caf/esm/cag"`, respectively.

To use **CAF** in Node via CJS format (with `require(..)`):

```
var CAF = require("caf");
var CAG = require("caf/cag");
```

Builds

build passing | npm package 15.0.1 | modules ESM+UMD+CJS

The distribution files come pre-built with the npm package distribution, so you shouldn't need to rebuild it under normal circumstances.

However, if you download this repository via Git:

1. The included build utility (`scripts/build-core.js`) builds (and minifies) the `dist/*` files.
2. To install the build and test dependencies, run `npm install` from the project root directory.
3. To manually run the build utility with npm:

   ```
   npm run build
   ```

4. To run the build utility directly without npm:

   ```
   node scripts/build-core.js
   ```

Tests

A test suite is included in this repository, as well as the npm package distribution. The default test behavior runs the test suite using the files in `src/` .

1. The tests are run with QUnit.
2. You can run the tests in a browser by opening up `tests/index.html` .
3. To run the test utility:

   ```
   npm test
   ```

   Other npm test scripts:

   - `npm run test:package` will run the test suite as if the package had just been installed via npm. This ensures `package.json` : `main` and `exports` entry points are properly configured.

- `npm run test:umd` will run the test suite against the `dist/umd/*` files instead of the `src/*` files.
- `npm run test:esm` will run the test suite against the `dist/esm/*` files instead of the `src/*` files.
- `npm run test:all` will run all four modes of the test suite.

Test Coverage

coverage 100%

If you have NYC (Istanbul) already installed on your system (requires v14.1+), you can use it to check the test coverage:

```
npm run coverage
```

Then open up `coverage/lcov-report/index.html` in a browser to view the report.

Note: The npm script `coverage:report` is only intended for use by project maintainers. It sends coverage reports to Coveralls.

License

license MIT

All code and documentation are (c) 2022 Kyle Simpson and released under the MIT License. A copy of the MIT License is also included.

Releases

36 tags

Sponsor this project

patreon.com/**getify**

https://www.paypal.com/paypalme2/getify

https://www.blockchain.com/btc/address/32R5dVrqirdcbiyvUw85y7YbPFZTd7YpnH

Learn more about GitHub Sponsors

Packages

No packages published

Used by 94

+ 86

Contributors 4

getify Kyle Simpson

DanielRuf Daniel Ruf

vkrol Veniamin Krol

chrisregnier Chris Regnier

Languages

- **JavaScript** 99.0%
- **HTML** 1.0%