V8 Async function design doc

Daniel Ehrenberg

Strategy

Async functions can be implemented mostly by a desugaring to generators. The desugaring proposed for V8 corresponds closely to the organization of the [the async-await specification](#).

Context

Async functions are defined in https://tc39.github.io/ecmascript-asyncawait/ and are at Stage 3 at TC39. Edge, WebKit and Firefox implementations are in progress. This document discusses implementing that proposal.

The desugaring

An async function like

```
async function foo(params...) {
      let y = await x;
      return y;
}
```

The desugaring is careful to use just a single function in the generated code, rather than an initializing/wrapping function and an inner generator. The idea here is to desugar `await` to trigger the `.next()` call, and skipping the initializing yield that generators have. `foo` would desugar into:

```
function* foo(params...) {
      // without the initializing yield and creating the generator
      // as in generators--done explicitly below
      try {
            // destructure params... and then
            let asyncContext = %CreateJSGeneratorObject(foo);
            let y = %RawYield(asyncContext,
                        AsyncFunctionAwait(asyncContext, x));
            return PromiseResolve(y);
      } catch (reason) {
            return PromiseReject(reason);
      }
}
```

Generally, `await x` is desugared into `yield AsyncFunctionAwait(asyncContext, x)` and `return x` is desugared into return `PromiseResolve(x)`.

In this desugaring, `asyncContext` is represented by a generator instance. Although the "execution context" is an abstract concept in the spec, generators map to the operations that the async/await spec does on execution contexts. The `asyncContext` does not leak to user code.

Exceptions are handled through this desugaring. If an exception is thrown and not caught by the async function code itself, the outer catch block in the desugaring will return a rejected promise with the reason. This works both for initial calls and later resumption. There is no need to handle an exception at each await, as they are all handled this way.

The loop is driven by reaching the next `await`, rather than by making successive `next` calls as other browsers have implemented it. Raw yields are used rather than iterator yields in the above code.

```
function AsyncFunctionAwait(asyncContext, value) {
     return PromiseThen(PromiseResolve(value),
          AsyncFunctionAwaitedFulfilled(asyncContext),
          AsyncFunctionAwaitedRejected(asyncContext));
}

function AsyncFunctionAwaitedFulfilled(asyncContext) {
     return value =>
          %_Call(GeneratorObjectNext, asyncContext, value);
}

function AsyncFunctionAwaitedRejected(asyncContext) {
     return reason =>
          %_Call(GeneratorObjectThrow, asyncContext, reason);
}
```

Beyond the desugaring

- A new constructor, %AsyncFunction%, needs to be added, with support for calling it with string arguments like the Function constructor. The function returned by AsyncFunctionStart has to have the right name and have %AsyncFunction%.prototype as its prototype.
- Minor: ensure yield and function.sent are not available within an async function
- We have to make sure devtools can do its normal operations through async functions. This involves updating the parser to parse async functions, and ensuring that Promise integration continues to work (no particular issues are anticipated).