Creating an awaitable lock for WinJS and JavaScript Promises

devblogs.microsoft.com/oldnewthing/20180803-00

August 3, 2018

Raymond Chen

Last time, we created an awaitable lock for C++ PPL tasks. Let's do the same thing for WinJS Promises (because I needed one of those too).

```
var AwaitableLock = WinJS.Class.define(function AwaitableLock() {
  this._locked = false;
  this._p = null;
  this._c = null;
}, {
  waitAsync: function waitAsync() {
    if (!this._locked) {
      // Lock is available. Acquire it.
      this._locked = true;
      return WinJS.Promise.wrap();
    }

    // Lock is not available.
    var self = this;
    if (!this._p) {
      // Create a promise that completes when the lock is released.
      this._p = new WinJS.Promise(function (c) { self._c = c; });
    }

    // Return a promise that waits for the lock to be released
    // and then retries the wait.
    return this._p.then(function () { return self.waitAsync(); });
  },

  release: function release() {
    this._locked = false;
    var c = this._c;
    this._p = null;
    this._c = null;

    // Complete any promise that was waiting for release.
    c && c();
  }
});
```

Since JavaScript is single-threaded, we don't have to worry about concurrency, so we don't need a mutex to guard our state variables. If the lock is available, then acquire it and return a completed promise.

Otherwise, we have to create a promise. The `WinJS.Promise` constructor takes a function which in turn receives a few parameters, although we use only `c` here. The `c` parameter is itself a function that our code can call to complete the promise. All we do is save that callback in the `_c` property so that we can complete the promise later. To avoid having to remember multiple completion callbacks, we create this promise once and cache it in the `_p` property.

We then chain a continuation on that newly-created promise that restarts the `waitAsync`, and return the resulting promise.

When the lock is released, we clear the saved completion and promise (thereby resetting the lock object to its unlocked state), and if there is a completion function, we call it. This completes the promise we created in `waitAsync`, which means that each of them will race to `waitAsync` and retry the acquisition.

There is a subtlety here: We reset the lock object completely before calling the completion. The completion is going to attempt to re-acquire the lock object, and we don't want the reentrant call to see a lock object in an inconsistent state. To avoid that, we capture the completion callback into a local `c`, and then after the lock object has been reset, we call the callback if necessary.

We can use ES6 function arrow notation to simplify this code, but if we're going to do that, we may as well go all the way and use ES6 native promises.

```
var AwaitableLock = WinJS.Class.define(function AwaitableLock() {
  this._locked = false;
  this._p = null;
  this._c = null;
}, {
  waitAsync() {
    if (!this._locked) {
      // Lock is available. Acquire it.
      this._locked = true;
      return Promise.resolve();
    }

    // Lock is not available.
    if (!this._p) {
      // Create a promise that completes when the lock is released.
      this._p = new Promise(c => this._c = c);
    }

    return this._p.then(() => this.waitAsync());
  },

  release() {
    this._locked = false;
    var c = this._c;
    this._p = null;
    this._c = null;

    // Complete any promise that was waiting for release.
    c && c();
  }
});
```

The ES6 Promise method for creating an already-completed promise is called `resolve`, as opposed to `wrap`, which is what `WinJS.Promise` calls it. The Promise constructor is the same, so we didn't have to make any changes there aside from arrowizing the functions. Arrowizing is convenient because it preserves `this` inside the lambda, which saves us the trouble of having to create a separate `self` variable.

And since we're abandoning WinJS, we may as well go all the way and define the class using ES6's native class declaration syntax.

```
class AwaitableLock {
  constructor() {
    this._locked = false;
    this._p = null;
    this._c = null;
  }

  waitAsync() {
    if (!this._locked) {
      // Lock is available. Acquire it.
      this._locked = true;
      return Promise.resolve();
    }

    // Lock is not available.
    if (!this._p) {
      // Create a promise that completes when the lock is released.
      this._p = new Promise(c => this._c = c);
    }

    return this._p.then(() => this.waitAsync());
  }

  release() {
    this._locked = false;
    var c = this._c;
    this._p = null;
    this._c = null;

    // Complete any promise that was waiting for release.
    c && c();
  }
}
```

Exercise: Why couldn't we arrowize the constructor?

Exercise 2: Why did we set the member variables dynamically in the constructor instead of defining them statically on the prototype?

Exercise 3: Why couldn't I have precalculated the `.then` of the promise?

```
// Code in italics is wrong - but why?
AwaitableLock.prototype.waitAsync = function waitAsync() {
  if (!this._locked) {
    // Lock is available. Acquire it.
    this._locked = true;
    return Promise.resolve();
  }

  // Lock is not available.
  if (!this._p) {
    // Create a promise that completes when the lock is released
    // and then retries the wait.
    this._p = new Promise(c => this._c = c).then(() => this.waitAsync());
  }

  return this._p;
}
```

Raymond Chen

Follow