Promise helpers

Yutaka Hirano <yhirano@chromium.org>

Note: This is a public document.

Objective

This document describes the design of Promise helpers in Blink.

Background

Promise is a ECMAScript6 standard library. It is used by many Web APIs having asynchronous operations, e.g. WebCrypto and ServiceWorker.

Preliminary: V8 context handling

As Promise is implemented with JavaScript in V8, we need to enter a valid v8 context (i.e. v8::Context) to use Promise operations. blink::ScriptState represents v8 context and you can enter / exit the context with blink::ScriptState::Scope.

```
ScriptState* scriptState = …;
ASSERT(!scriptState->isolate()->InContext());
{
    ScriptState::Scope scope(scriptState);
     // Here we have entered a v8 context |scriptState->context()|.
    ASSERT(scriptState->isolate()->InContext());
}
ASSERT(!scriptState->isolate()->InContext());
```

There are a few principles.

- When called from JavaScript, you are already in the appropriate v8 context. You should not enter another v8 context unless you truly need to.
- Otherwise, you need to enter the appropriate v8 context to use Promise operations. Typically, you can save RefPtr<ScriptState> into your class and use it when it is needed. Adding [CallWith=ScriptState] to an IDL function gives you the ScriptState when the function is called.
- In spite of the above description, some functions (e.g. ScriptPromiseResolver::resolve) enters a v8 context automatically. You don’t have to enter a v8 context to use them.

ScriptPromise

ScriptPromise represents a Promise object (i.e. v8::Promise). Because a Promise object keeps track of attached functions, holding ScriptPromise in a class leads to a memory leak. You can take or pass a ScriptPromise as a parameter or a return value, but you must not hold it as a member.
The IDL code generator converts ScriptPromise and v8::Handle<v8::Promise> automatically when “Promise” type is specified in an IDL file.
ScriptPromise::then corresponds to Promise.prototype.then. You can attach arbitrary functions to the ScriptPromise.

ScriptFunction

ScriptFunction is a helper class that enables you to define a JavaScript function easily. You can define a class inheriting ScriptFunction and override *call* method. When you call *bindToV8Function*, the result v8 Handle holds keeps the function object alive.
There is a restriction: calling bindToV8Function twice leads to a problem. We recommend you not to expose the object out of the class to avoid accidents.
Here is an example of ScriptFunction subclass.

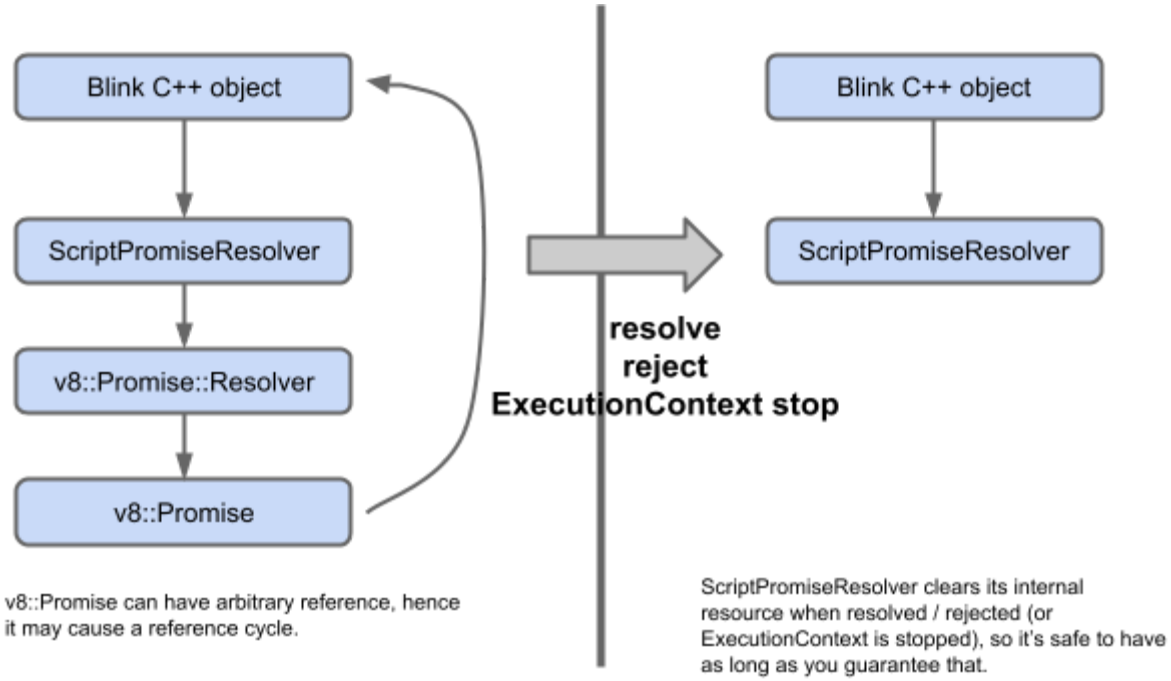
```
class AddOneFunction : public ScriptFunction {
public:
    static v8::Handle<v8::Function> createFunction(ScriptState* scriptState)
    {
        AddOneFunction* self = new AddOneFunction(scriptState);
        return self->bindToV8Function();
    }

private:
    explicit AddOneFunction(ScriptState* scriptState) : ScriptFunction(scriptState) { }

    virtual ScriptValue call(ScriptValue value) OVERRIDE
    {
        int intValue = value.v8Value().As<v8::Integer>()->Value();
        return ScriptValue(scriptState(), v8::Integer::New(scriptState()->isolate(), intValue + 1));
    }
};
```

ScriptPromiseResolver

ScriptPromiseResolver resolves / rejects the associated Promise. A ScriptPromiseResolver has the associated ScriptPromise, so having a ScriptPromiseResolver may cause cyclic references. Unlike ScriptPromise, having a ScriptPromiseResolver is allowed because it is natural to keep objects while there is possibility to be resolved or rejected. As a result, a user must call *reject* when there is no possibility to resolve the promise in the future. Calling *resolve* or *reject* releases all resources held by the resolver, so you don’t have to worry about them after that.



V8 Context handling

ScriptPromiseResolver::resolve and ScriptPromiseResolver::reject enters the v8 context on that the resolver was created. Hence it is needless to enter the v8 context manually.

```
{
    RefPtr<ScriptPromiseResolver> resolver = …;
    …
    // You don't have to enter a v8 context here.
    resolver->resolve("hello");
}
```

Note that other functions such as *ScriptPromiseResolver::promise* doesn't have such property.

ExecutionContext state

ScriptPromiseResolver stops working and releases all resources when the associated ExecutionContext stops. That means the resource leak doesn't persist beyond the document lifetime, though calling *resolve* or *reject* appropriately is much more preferred if possible. It also means that ScriptPromiseResolver is useless (i.e. *resolve* and *reject* take no effect) when the associated ExecutionContext is stopped.

Resolution / Rejection timing

When *resolve* or *reject* is called, the Promise internal state changes immediately, but the associated handlers will be executed in the next microtask execution, i.e. asynchronously. This behavior is consistent with JavaScript Promise's behavior.

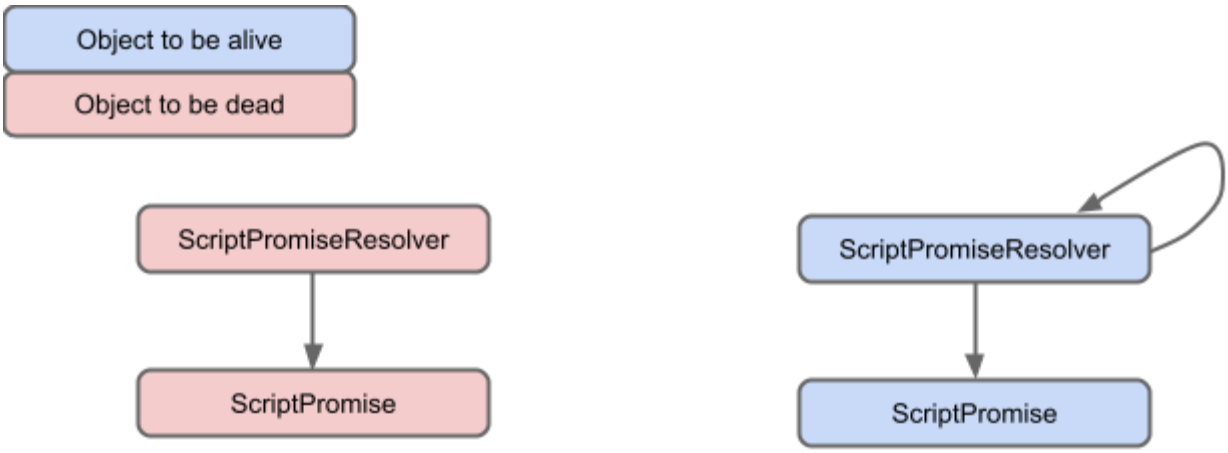
```
{
    RefPtr<ScriptPromiseResolver> resolver = …;
    // You need to enter the approriate v8 context here.
    resolver.promise().then(onFulfilled);
    …
    // You don't have to enter a v8 context here.
    resolver->resolve("hello");
    // onFulfilled is not called yet.
}
```

keepAliveWhilePending

ScriptPromiseResolver::keepAliveWhilePending is a protected method. When called, it increments the reference counter so that the instance will live without being referenced. When *resolve* or *reject* is called, or the ExecutionContext is stopped, the reference counter will be decremented.
This method is implemented for "Asynchronous Initializer"s. Some APIs such as WebMIDI provides a function that returns a Promise which will be resolved with a context object when the context object is initialized successfully.

```
partial interface Navigator {
  // The Promise will be resolved with a MIDIAccess when it is initialized.
  Promise requestMIDIAccess(optional MIDIOptions options);
};
```

In such a case, we need to create a C++ class that manages the initialization. On the other hand, there is no natural object that holds the initializer.
keepAliveWhilePending enables us to keep the initializer alive without explicit references while the promise is pending.



Usual Resolver: Without reference, it is deleted and so its internal state is.

With KeepAliveWhilePending: As it is self-referencing, the resolver is alive until it is resolved, rejected or the ExecutionContext is stopped.

Here is an example of asynchronous initializer.

```
class MIDIAccessInitializer : public ScriptPromiseResolver, public MIDIAccessorClient {
public:
    ...
    static ScriptPromise start(ScriptState* scriptState, const MIDIOptions& options)
    {
        RefPtr<MIDIAccessInitializer> p =
            adoptRef(new MIDIAccessInitializer(scriptState, options));
        p->keepAliveWhilePending();
        p->suspendIfNeeded();
        return p->start();
    }
    virtual ~MIDIAccessInitializer();

    // MIDIAccessorClient
    virtual void didStartSession(...) override;
    ...
private:
    MIDIAccessInitializer();
    ScriptPromise start();
    ...
};
```

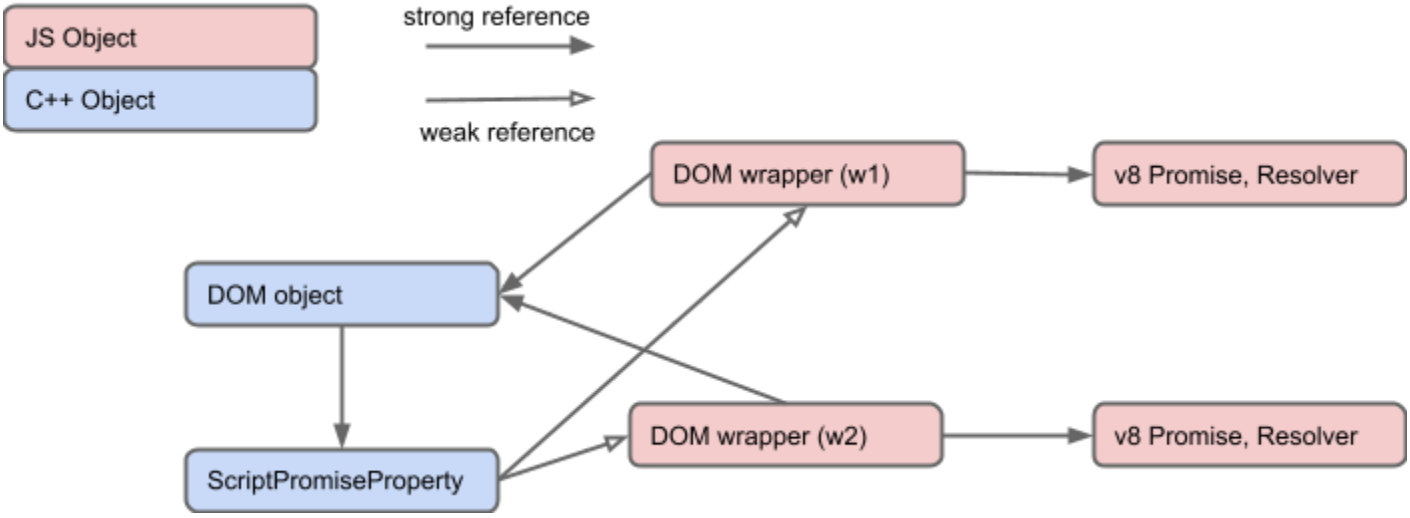
This class doesn't expose an instance reference, but the instance is kept alive until *resolve* or *reject* is called, or the ExecutionContext is stopped. *resolve* or *reject* is called in *didStartSession*. That way, we can implement "asynchronous initializer"s with this feature. Note that this is complex and you shouldn't use it without understanding the mechanism.

ScriptPromiseProperty

ScriptPromiseProperty represents a property of type Promise held in a DOM object. As said before, ScriptPromise should not be held in an object as a member, for two reasons.

1. ScriptPromise has a strong reference to the v8 Promise object and a v8 Promise object can have a (strong) reference for arbitrary object. That may cause a circular reference leading to resource leaks.
2. A DOM object can be shared among multiple worlds. Returning the same Promise to multiple worlds leads to an object leak between worlds which is a problem in terms of security.

ScriptPromiseProperty solves these problems.

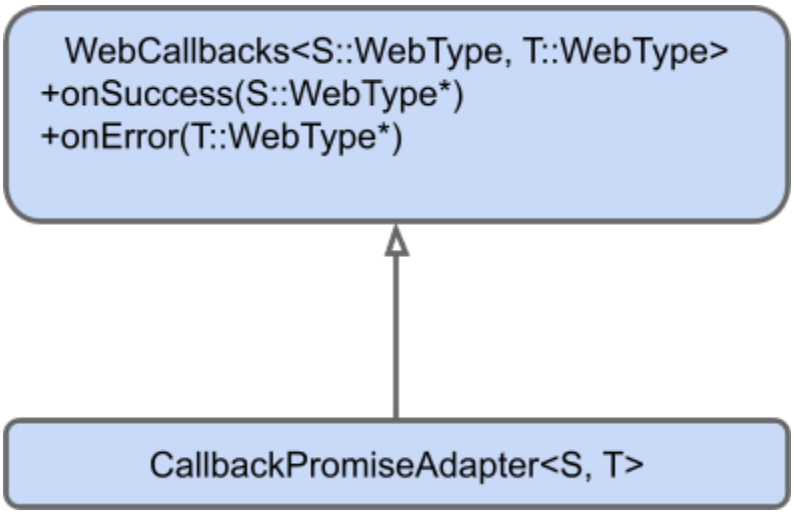


The above describes the case where two worlds share a DOM object. In such a case, one DOM wrapper for each world is created. ScriptPromiseProperty holds a set of weak references to DOM wrappers of the DOM object that holds the property. Despite the fact that v8 Promise objects can have (string) references for arbitrary objects, there are no reference cycles because DOM wrappers are referenced by weak references. In addition to that, as ScriptPromiseProperty has a set of wrappers, it can return a different Promise object for each world.
ScriptPromiseProperty has *resolve* and *reject* methods. They resolve / reject the all involved promise objects simultaneously, respectively.

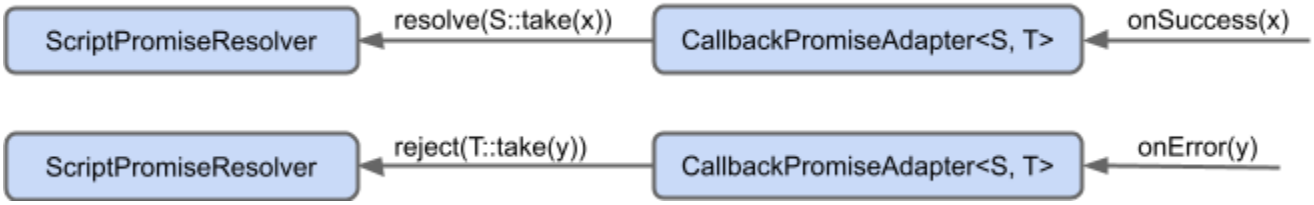
There is one caveat. As ScriptPromiseProperty doesn't hold strong references, a DOM wrapper can be garbage collected. Once it is garbage collected, *resolve* and *reject* don't take effect even if promise handlers were attached to the promise object. To prevent this, you may need to keep DOM wrappers until *resolve* or *reject* is called. ActiveDOMObject provides such functionality.

CallbackPromiseAdapter

CallbackPromiseAdapter enables you to resolve / reject a Promise from outside of Blink. CallbackPromiseAdapter is a template class that takes two types S and T. Both types should have its associated types S::WebType and T::WebType. CallbackPromiseAdapter<S, T> is a subclass of WebCallbacks<S::WebType, T::WebType>. As WebCallbacks is defined in public/platform, it's visible from content/ layer.



CallbackPromiseAdapter has a ScriptPromiseResolver. When onSuccess is called, CallbackPromiseAdapter<S, T> calls S::take and resolves Promise with its return value. If the associated ExecutionContext is stopped, it calls S::dispose instead. Note that the ownership of the argument of onSuccess is not specified - it completely depends on the caller and the callee. When onError is called, CallbackPromiseAdapter<S, T> calls T::take in a similar way.



Please read the CallbackPromiseAdapter's class comment for details.

Web IDL code generator

Promise type in WebIDL is tied to ScriptPromise in Blink.

Promise parameter

When a non-promise value is given to a parameter that is specified as Promise, the value will be automatically converted to a Promise object with *ScriptPromise::cast*. As a result, when you write an IDL function that takes a Promise parameter, the code generator accepts any value for the parameter and converts it to a Promise.

Promise return value

As specified in the Web IDL spec, functions returning Promise return a rejected Promise instead of throwing a error. This is done by the code generator. As a side effect, when you throw an exception (via ExceptionState), it will not be thrown and a rejected Promise will be returned. But simply returning a rejected Promise is preferable because it doesn't confuse readers.