Mojom JS, The Sequel

rockot@

Overview

This is a quick stab at proposing a better shape for mojom JS bindings given ongoing discussions about observed pain points, as covered by dpapad@ in this doc.

Things this proposal tries to address immediately:

- JS code size, both for the support library and the stuff generated from mojom
- Usability of outbound interfaces, i.e. making calls from JS to C++
- Usability of inbound interfaces, i.e. dispatching incoming messages from C++ to JS impls

Here are things this does not try to address now:

- Associated interfaces. We currently have support for associated interfaces in JS bindings, and rebuilding it for a new bindings scheme would be a somewhat involved process. It's not immediately necessary and can be added later with no impact on new API look & feel.
- Control messages. Same -- we have support for these things now (e.g. version query and assertion, interface flushing) but they aren't critical for immediate WebUI needs and can be added as needed.
- The concern over bindings' Promise semantics being incongruent to those of chrome.send(). It is difficult to imagine a solution to this that does not require untenable changes to the mojom specification, and the relatively small payoff would not seem to justify such changes[1].

Common Code

It's still useful to have a library of common JS support code that generated mojom bindings will rely on, but we don't need to expose generic endpoint primitives like InterfacePtr or Binding. One of the most common complaints is that such primitives feel unnatural to JS developers.

Instead, the common support code will:

- Know how to serialize, deserialize, and validate all primitive mojom types -- strings, ints, handles, structs, unions, etc.
- Know how to take an arbitrary list of JS values (function arguments) or a JS Object; and serialize, deserialize, and validate it according to a declarative (JSON) method signature or struct/union specification
- Know how to stamp out an Object from a declarative (JSON) interface specification, such that the object exposes callable properties which send corresponding interface messages.

[1] It's worth noting that a majority of mojom interface methods do not bother defining a notion of "success" or "failure" and this is idiomatic design. Most failures are exceptional cases and thus don't benefit from more context, so the most common way to signal such failures is to close the interface pipe. This manifests as a Promise rejection in JS bindings.

- Know how to stamp out an EventTarget from a declarative (JSON) interface specification, such that incoming messages on an interface can be dispatched as events on that target.

This should have a significantly smaller size than the current base JS support library, whose code consists largely of defining presentable object prototypes for a large number of low-level primitives. In relegating user-facing API to a handful of simple function calls and run-time-generated Objects, we buy a lot of latitude to get scrappy on internal code size.

Generated Code

Generated JS from mojom is greatly simplified down to essentially a declarative specification of each type. Consider the mojom definitions:

```
enum BusType {
  kUsb,
  kBluetooth,
};

struct DeviceInfo {
  string guid;
  uint16 product_id;
  BusType bus_type;
 };

interface DeviceManager {
  GetAllDevices() => (array<DeviceInfo> devices);
  GetDeviceInfo(string guid) => (DeviceInfo? info);
  ResetAllDevices();
};
```

This would generate the following JS, modulo a bit of hand-waving:

```
BusType = new mojo.EnumSpec({kUsb: 0, kBluetooth: 1});

DeviceInfo = new mojo.StructSpec({
  fields: [
    { type: mojo.String, name: 'guid' },
    { type: mojo.Uint16, name: 'productId' },
    { type: BusType, name: 'busType' },
  ]
});

DeviceManager = new mojo.InterfaceSpec({
  methods: [
    {
      name: 'getAllDevices',
      params: [],
      responseParams: [{
        type: mojo.Array(DeviceInfo),
        name: 'devices'
      }],
    },
    {
      name: 'getDeviceInfo',
      params: [{ type: 'string', name: 'guid' }],
      responseParams: [{
        type: DeviceInfo,
        optional: true,
        name: 'info',
      }],
    },
    {
      name: 'resetAllDevices',
      params: [],
    },
  ]
});
```

Pretty small!

Creating Pipes & Binding Interfaces

When a user wants to access a top-level interface brokered by the browser, the JS environment is still ultimately going to rely on the native Mojo Blink IDL to make it happen. This means we will still be creating message pipes and we'll still need a way to tie a message pipe endpoint to an interface definition like the one above, as either a sender or a receiver. Ideally the pipe creation and `bindInterface` steps would be better hidden from the user, unlike today.

Acquiring Top-Level Interfaces

For the simple and common case of acquiring a top-level interface, the caller can write something like:

```
let deviceManagerProxy = DeviceManager.getProxy();
```

The generated `getProxy` method creates a new message pipe, sending one end to the browser[2] as an interface request. It returns the other end as a proxy, stamped out according to the interface spec in DeviceManager. This proxy object includes callable properties `'getAllDevices'`, `'getDeviceInfo'`, and `'resetAllDevices'`, which are each wrappers around a single shared serialization function (provided by the support library) which takes the corresponding generated method spec as an input.

Exposing Interfaces from JS

Many services incorporate client interfaces into their design as the most natural way to push messages back to their clients. A webby way to model receipt of these messages is as Event dispatches. Let's assume more mojom interface definitions:

```
interface DeviceObserver {
  OnDeviceAdded(DeviceInfo info);
  OnDeviceRemoved(DeviceInfo info);
};

interface DeviceMonitor {
  AddObserver(DeviceObserver observer);
};
```

In order to support an implementation of the DeviceObserver interface within JS, the caller could write something like:

```
let observer = new DeviceObserver;
observer.onDeviceAdded.addListener(request => {
  console.log('Added new device %s.', request.info.guid);
});
observer.onDeviceRemoved.addListener(/* etc */);
```

Straightforward and feels like web. Note that this is incomplete because creating a receiver is pointless without creating a proxy to give to the receiver's corresponding sender.

We still need a way to pass a DeviceObserver proxy over a DeviceManager API call.

Passing Endpoints

One of the fundamental features of mojom interface design is the ability to freely pass around interface endpoints over other interface methods. This is the basis for countless useful patterns, such as the DeviceManager/DeviceObserver relationship above. It's extremely common for

[2] This means to the WebUI's frame host, which may then route the interface request anywhere else in the system, including to other services.

service APIs to define multiple logical sub-interfaces which are acquired from higher-level interfaces, and it's imperative that JS bindings support this if we want to reuse the same IPC surface from JS.

There are really two distinct cases: sending interface requests (i.e. endpoints that will receive messages) and sending interface proxies (i.e. endpoints that can be used to send messages). At a low level they're identical, but it will probably help to treat them separately for JS bindings.

Sending Requests

The request case is interesting because all we have so far is the generated `Foo.getProxy()` as described above, and this can only send requests to the browser for further routing with no additional context. For a concrete example of where more context is required, consider these interfaces:

```
interface DeviceManager {
  GetDevice(string guid, Device& request);
};

interface Device {
  Explode();
};
```

A caller could not ask the browser for a Device interface directly, because the browser (or whatever the ultimate target service is) has insufficient context to know how to bind it. The caller wants a Device interface which controls a specific device identified by a specific GUID. Such context is established by calling a method on a higher-level interface -- in this case, DeviceManager -- which carries additional arguments along with the interface request itself.

In JS, at least naively this could look something like:

```
let deviceManagerProxy = DeviceManager.getProxy();
let deviceProxy = new DeviceProxy;
deviceManagerProxy.getDevice(kAwesomeGuid, deviceProxy.createRequest());
deviceProxy.explode();
```

NOTE: The exact shape of this API is open for discussion. It's mostly here to clarify the necessity of having an object which represents an unbound interface request, however transient, because we need to be able to pass such objects as arguments to arbitrary interface calls.

Sending Proxies

This is also an interesting problem, with a legitimate use case (the DeviceObserver thing) outlined further above. In this case, you want to create an event target which receives (e.g.)

DeviceObserver events, and you want to send the corresponding DeviceObserver proxy elsewhere to be used to make calls. The most straightforward approach would again look something like:

```
let deviceMonitorProxy = DeviceMonitor.getProxy();
let observer = new DeviceObserver;
deviceMonitor.addObserver(observer.createProxy());

observer.addEventListener('ondeviceadded', (event) => { ... });
```

Open Questions

Closure

It's not immediately clear how this proposal would affect integration with Closure and related tooling. Specifically this is because all interface endpoints (i.e., the mojom things used directly by WebUI code) would be dynamically stamped-out objects with no statically-analyzable type information.

A trade-off would be to add a little more codegen back to the mix so that the callable interface prototypes are still generated with proper annotation, but their implementation would follow the suggestions outlined in this doc, i.e., using generic shared serialization code with each method's declarative spec as input.

Other Concerns?

Eh?