Mojo & The Naming of Things

rockot@
Feb 2019

Overview

Chromium developers are understandably confused by names like `InterfacePtr`, `InterfaceRequest`, `MakeRequest`, *etc*. This document proposes new names for things.

UPDATE: The result of this discussion is that new names have been introduced and this document provides detailed information about how to convert old code to new types.

Naming Concerns

InterfacePtr / InterfacePtrInfo

`InterfacePtr<T>` is the type held by *callers* calling into a remote implementation of an interface. Its intermediate (i.e. transferable) precursor is called `InterfacePtrInfo<T>`.

So if you define a mojom like:

```
interface Widget {
  AddObserver(WidgetObserver observer);
};
```

You'll get a C++ interface like:

```
class Widget {
  virtual void AddObserver(mojo::InterfacePtr<WidgetObserver> observer)=0;
};
```

And when a `Widget` implementation receives an `AddObserver` message, they can take ownership of the `InterfacePtr` object and use it to subsequently call methods on some remote `WidgetObserver` implementation with which the `InterfacePtr` is entangled.

There are two central problems with the name `InterfacePtr<T>`:

- It isn't a "pointer" in the sense that virtually every developer seems to anticipate. While the spirit of the name is that the object points to a remote implementation of the interface *T*, experience shows that the term "pointer" may be too misleading.
- Building on the above, the object isn't really pointing to an interface *per se*, but to an *implementation*. So even having "interface" in the name is somewhat redundant at best and misleading at worst.

The proposed replacement for this is `Remote<T>`. It's short, simple, and conveys the most important information about the object: it references a *remote* implementation of *T*. The proposed equivalent for `InterfacePtrInfo<T>` is `PendingRemote<T>`.

Finally, we can introduce new syntax in mojom IDL to pass this type of endpoint:

```
interface Widget {
  AddObserver(pending_remote<WidgetObserver> observer);
};
```

This is equivalent to saying `AddObserver(WidgetObserver observer)` today, but the generated bindings would take a `PendingRemote<WidgetObserver>` instead of a `WidgetObserverPtr`. If/when we get rid of `InterfacePtr<T>`, we would also get rid of the corresponding mojom syntax where bare interface names can be used as field/param types.

Binding / InterfaceRequest

`Binding<T>` is the type that interface *implementations* use to receive and locally dispatch incoming interface messages. `InterfaceRequest<T>` serves as its intermediate (i.e. transferable) precursor. Mojom IDL uses the syntax of "T&" to refer to an `InterfaceRequest<T>` in transit. Typical usage looks something like:

```
// mojom
interface Widget { Click(); };
interface WidgetFactory { CreateWidget(Widget& request); };

// generated C++
class WidgetFactory {
  virtual void CreateWidget(mojo::InterfaceRequest<Widget> request)=0;
};

// application C++
class WidgetFactoryImpl : public mojom::WidgetFactory {
  // ...
  void CreateWidget(mojo::InterfaceRequest<Widget> request) override {
    widgets_.emplace_back(std::move(request));
  };

 private:
  std::vector<WidgetImpl> widgets_;
};

class WidgetImpl : public mojom::Widget {
 public:
  explicit WidgetImpl(mojo::InterfaceRequest<mojom::Widget> request)
      : binding_(this, std::move(request)) {}
 private:
  mojo::Binding<mojom::Widget> binding_;
};
```

Anecdotally, the term "request" seems to mislead developers to believe there is more substance to an `InterfaceRequest` than a mere `Binding` precursor.

The working proposal offered here is to replace `Binding<T>` with `Receiver<T>`, and `InterfaceRequest<T>` with `PendingReceiver<T>`. IDL would look like:

```
interface WidgetFactory {
  CreateWidget(pending_receiver<Widget> widget);
};
```

MakeRequest

`MakeRequest` creates a new message pipe, yielding one end as the return value (an `InterfaceRequest<T>`) and the other end in an output parameter (an `InterfacePtr<T>`). This function is used in a large number of places, and an overwhelming majority of message pipes created in the system today are created by calling `MakeRequest`. For example:

```
mojom::WidgetPtr widget;
widget_factory->CreateWidget(mojo::MakeRequest(&widget));
```

This creates a new pipe, binding the *calling* side to `widget` and passing the *receiving* side to `CreateWidget`.

The name and purpose of the function has been a consistent source of confusion across multiple renames (original `Get`, then `GetProxy`, and now `MakeRequest`).

The proposed replacement for this is a series of appropriate methods on each endpoint primitive. Each method need only mutate `this` and return a single value. For example:

```
PendingReceiver<T> Remote::BindNewReceiver();
PendingRemote<T> Receiver::BindNewRemote();
PendingRemote<T> ReceiverSet::BindNewRemote();
```

The equivalent to the above example would thus be:

```
mojo::Remote<mojom::Widget> widget;
widget_factory->CreateWidget(widget.BindNewReceiver());
```

Associated

Nevermind, working on a possible solution for deleting these altogether :D In the meantime, we'll probably keep the "Associated" modifier for new endpoint types.

While We're Here: Related Concerns

Dropping Type Aliases

EDIT: *This suggestion has received near-unanimous support from developers so far, so it looks like a sure thing now.*

The proposal here is that we avoid introducing shorthand type aliases (like `FooPtr = mojo::InterfacePtr<Foo>`) for all new bindings types. While the aliases save some typing they do not necessarily improve readability, especially for people less familiar with Mojo.

Something like `mojo::Remote<T>` is approximately as verbose as `base::Optional<T>` or `std::unique_ptr<T>`, and no more common than either. Developers will probably be OK with reading and writing it.

Call Site Attribution

There is also a desire to have better attribution to IPC call sites, which would mean somehow capturing a `base::Location` (*i.e.*, `FROM_HERE`) when making Mojo interface calls. One way we could achieve this (which would also obviate the need for the `rpc()` suggestion above) would be to introduce a helper function like:

```
mojo::Call(FROM_HERE, gpu_)->EstablishGpuChannel(...);
```

An alternative suggestion to Call was `PostRemoteTask`, but this doesn't translate well for sync calls, and it may be better to have a similar calling mechanism for both sync and async. Sync calls could be made more obvious with SyncCall analog:

```
mojo::SyncCall(FROM_HERE, gpu_)->DoSomethingSync(...);
```

Note that this calling convention would also address concerns that IPC call sites are not immediately obvious to readers.

Dropping the Inner mojom Namespace?

EDIT: *Not going to do this because there's much more feedback opposing it than favoring it. Leaving the section here for posterity and any follow-up discussion.*

This is a more contentious issue, but it has been suggested by many developers that we should consider dropping the convention of using an inner mojom namespace for all symbols generated from mojom bindings. So it's the difference between writing:

```
module network.mojom;

interface URLLoaderFactory {...};
```

And having e.g. `mojo::Remote<network::mojom::URLLoaderFactory>` vs just writing:

```
module network;

interface URLLoaderFactory {...};
```

And having `mojo::Remote<network::URLLoaderFactory>`.

The original motivation for the namespace was to avoid anticipated collisions between mojom-defined symbols and other symbols (e.g. mojom::Foo and its impl Foo), but there is

some evidence that this isn't much of an issue anyway (people seem to prefer Impl suffixes regardless).

On the other hand, many developers have expressed that they find the presence of "mojom" to be a useful signal when reading code. For example, when scanning a class definition it is obvious that a batch of method overrides are part of an IPC implementation.