Mojo Bindings Conversion Cheatsheet

rockot@google.com
Last update: 30 May 2019

Overview

This covers how to transition Chromium code from old Mojo bindings types to new ones. The old and new types are generally functionally equivalent, with minor cosmetic differences.

Conversion often requires modifying both mojom and C++ sources. As of this writing Java and JS code remains unaffected and continues to use older terminology. That's WIP.

Example CLs

Here are some example CLs which do type conversions. Note that while these are all focused on converting individual services defined in the tree, in some cases it might be reasonable to write a CL which just converts a handful of source files in src/content or src/chrome, or in some src/components subdirectory.

- Chrome OS IME service conversion: https://chromium-review.googlesource.com/c/chromium/src/+/1632131
- file and filesystem service conversion: https://chromium-review.googlesource.com/c/chromium/src/+/1632895
- proxy_resolver service conversion: https://chromium-review.googlesource.com/c/chromium/src/+/1636722

Tracking Bug

Conversion CLs can be attached to bug 95Other types were migrated as well:5171.

Aliases

As a general note about type aliases, mojom-generated C++ code includes generated type aliases for the old bindings types. For example, if a mojom defines an interface Foo, there will be a C++ FooRequest type alias mapping to the mojo::InterfaceRequest<Foo> type.

We have explicitly chosen *not* to define similar aliases for the new bindings types, so don't bother looking around for them. Instead you will now always write out e.g., mojo::PendingReceiver<Foo>.

Terminology

Some terminology has changed. In general:

- An "InterfacePtr" or often verbally "interface pointer" is now a "remote".
- An "InterfacePtrInfo" is now a "pending remote".
- A "binding" is now a "receiver".

- An "interface request" is now a "pending receiver."
- A "connection error handler" is now a "disconnect handler". Invocation happens at the same time for the same reasons, this is just a rename to better reflect the actual meaning of the callback.

Requesting Interface Implementations

Old & Busted

If a mojom defines some interface FooFactory that can request an implementation of a Foo:

```
interface FooFactory {
  GetFoo(Foo& request);
};
```

Today this emits a C++ signature like `GetFoo(mojom::FooRequest request)` and an implementation might look something like:

```
class FooFactoryImpl : public mojom::FooFactory {
 public:
  explicit FooFactoryImpl(mojom::FooFactoryRequest request)
      : binding_(this, std::move(request)) {}

  // mojom::FooFactory:
  void GetFoo(mojom::FooRequest request) override {
    foo_bindings_.AddBinding(this, std::move(request));
  }

 private:
  mojo::Binding<mojom::FooFactory> factory_binding_;
  mojo::BindingSet<mojom::Foo> foo_bindings_;
};
```

New & Shiny

We replace Binding(etc) types with Receiver types, and InterfaceRequest with PendingReceiver. All of these conversions are very straightforward. Instead of the above, now do this:

```
// mojom
interface FooFactory {
  GetFoo(pending_receiver<Foo> receiver);
};

// C++
class FooFactoryImpl : public mojom::FooFactory {
 public:
  explicit FooFactoryImpl(
      mojo::PendingReceiver<mojom::FooFactory> receiver)
```

```
      : factory_receiver_(this, std::move(receiver)) {}

  // mojom::FooFactory:
  void GetFoo(mojo::PendingReceiver<mojom::Foo> receiver) override {
    foo_receivers_.Add(this, std::move(receiver));
  }

 private:
  mojo::Receiver<mojom::FooFactory> factory_receiver_;
  mojo::ReceiverSet<mojom::Foo> foo_receivers_;
};
```

Sending Callable Interface Endpoints

Old & Busted

If a mojom defines some interfaces:

```
interface Foo {
  Poke();
};

interface Bar {
  PokeFoo(Foo foo);
};
```

Today this emits a C++ signature like `PokeFoo(mojom::FooPtr foo)` and an implementation might do something like:

```
void BarImpl::PokeFoo(mojom::FooPtr foo) {
  foo->Poke();
}
```

New & Shiny

We replace InterfacePtr with Remote. Slightly less straightforward though is that we no longer send the equivalent type over mojom -- instead of sending an InterfacePtr, you send a PendingRemote, which is equivalent to the old InterfacePtrInfo (an InterfacePtr precursor). Instead of the above, now do:

```
interface Foo {
  Poke();
};

interface Bar {
  PokeFoo(pending_remote<Foo> foo);
};
```

And the implementation would look like:

```
void BarImpl::PokeFoo(mojo::PendingRemote<mojom::Foo> pending_foo) {
  mojo::Remote<mojom::Foo> foo(std::move(pending_foo));
  foo->Poke();

  // Or if you had a member mojo::Remote<mojom::Foo> foo_, you might
  // instead write:
  //
  //   foo_.Bind(std::move(pending_foo));
  //   foo_->Poke();
  //
  // Or if you wanted to be supremely lazy when sending a one-off
  // message to the remote Foo, you could write:
  //
  //   mojo::Remote<mojom::Foo>(std::move(pending_foo))->Poke();
}
```

This does mean occasionally an extra line or two of code compared to the old types, but passing PendingRemote around instead of Remote also means more efficiency in many cases; hence the change.

Associated Interface Types

Associated interface types (both AssociatedInterfaceRequest and AssociatedInterfacePtr) are affected similarly to the above two sections.

Old & Busted

```
interface Foo {
  GetBar(associated Bar& request, associated BarClient client);
};
```

This generates C++ like:

```
void GetBar(mojom::BarAssociatedRequest,
            mojom::BarClientAssociatedPtr client) = 0;
```

New & Shiny

```
interface Foo {
  GetBar(pending_associated_receiver<Bar> receiver,
         pending_associated_remote<Barclient> client);
};
```

This generates C++ like:

```
void GetBar(mojo::PendingAssociatedReceiver<mojom::Bar> receiver,
            mojo::PendingAssociatedRemote<mojom::BarClient> client) = 0;
```

Note that similarly to non-associated interfaces, we transfer the "pending" remote type only.

There are corresponding AssociatedRemote, AssociatedReceiver and AssociatedReceiverSet types which map 1:1 to the AssociatedInterfacePtr, AssociatedBinding, and AssociatedBindingSet counterparts, modulo superficial naming differences.

Creating Message Pipes

Old & Busted

A common way to create message pipes today looks like this:

```
mojom::FooPtr foo;
foo_factory->GetFoo(mojo::MakeRequest(&foo));
```

The actual message pipe creation happens inside MakeRequest(), which creates a pipe, binds one end to foo, and returns the other end as a FooRequest.

New & Shiny

Instead of the above, you'd write something like:

```
mojo::Remote<mojom::Foo> foo;
foo_factory->GetFoo(foo.BindNewPipeAndPassReceiver());
```

`mojo::Remote<T>::BindNewPipeAndPassReceiver()` returns a `mojo::PendingReceiver<T>`.

Additional Notes on Pipe Creation

It is not always the case that you want to hold onto the Remote of a new pipe while passing the receiving endpoint. Sometimes you want to do the opposite, and you can:

```
mojo::Receiver<mojom::Foo> receiver(&foo_impl);
foo_observer->AddFoo(receiver.BindNewPipeAndPassRemote());
```

Or maybe you want to send both endpoints elsewhere. In this case, you want to construct a PendingRemote first, and use InitWithNewPipeAndPassReceiver:

```
mojo::PendingRemote<mojom::Foo> remote;
foo_factory->GetFoo(remote.InitWithNewPipeAndPassReceiver());
foo_observer->AddFoo(std::move(remote));
```

Strong Bindings

MakeStrongBinding Considered Harmful

Uses of MakeStrongBinding have historically been error-prone, and developers should prefer explicit ownership of Mojo interface implementations.

There is a new mojo::UniqueReceiverSet type which can bind arbitrarily many pending receivers to uniquely-owned implementations. Each bound implementation object still has its lifetime conveniently limited by that of its corresponding receiver, with the added constraint that it's *also* limited by the lifetime of the UniqueReceiverSet itself.

UniqueReceiverSet thus provides most of the convenience of MakeStrongBinding (automatic implementation destruction on disconnect) without the unbounded lifetime issues that can easily lead to memory bugs.

MakeSelfOwnedReceiver

If you really must convert without moving away from pipe-bound ownership, there is mojo::MakeSelfOwnedReceiver, which is functionally equivalent to mojo::MakeStrongBinding.

ThreadSafeInterfacePtr

This is replaced by SharedRemote, with a slightly nicer API in that it only requires a single dereference to make calls like any other Remote, whereas ThreadSafeInterfacePtr required double-dereferencing.

Service APIs

Connecting to Services

You still use `service_manager::Connector`, but instead of calling `BindInterface()`, call `Connect()`. This method takes any type of `mojo::PendingReceiver` but is otherwise equivalent to BindInterface().

Implementing a Service

You still implement service_manager::Service, but instead of overriding `OnBindInterface`, prefer to override `OnConnect` instead. **Do not override both**.

These methods are equivalent, but the "OnConnect" naming reflects the new naming of Connector::Connect.

BinderRegistry

We also introduce a new service_manager::BinderMap type which should be preferred over service_manager::BinderRegistry. See the header for documentation.

NOTE: As it stands, BinderRegistry is already overused. If a service only has ~1-3 interface binders registered, consider writing out explicit interface name comparisons instead of using BinderMap. e.g.

```
void FooService::OnConnect(const service_manager::ConnectSourceInfo& source,
                           const std::string& interface_name,
                           mojo::ScopedMessagePipeHandle receiver_pipe) {
  if (interface_name == mojom::Bar::Name_) {
    bar_receivers_.Add(
        this, mojo::PendingReceiver<mojom::Bar>(std::move(receiver_pipe));
  } else if (interface_name == mojom::Frobinator::Name_) {
    ...
  }
```

Miscellaneous Conversion Tips

Splitting CLs

Some components may be too large for a single conversional CL to be practical. In such cases, try to find reasonable API boundaries to split the changes along. There are implicit constructors and conversion operators in place between many old and new types to ease this burden.

For example, any method parameter which takes a mojom::FooRequest (i.e. a mojo::InterfaceRequest<mojom::Foo>) can also implicitly accept a mojo::PendingReceiver<mojom::Foo>.

Stricter Assertions

The newer types make a few stricter assertions than the old ones. For example it is illegal to call Bind() on an already-bound Remote, or to call Bind() with an invalid PendingRemote. Conversely, InterfacePtr silently allows this kind of error-prone behavior.

Some conversion work may require choices to be made about, e.g., explicitly resetting the state of a Remote that was being silently trampled as an InterfacePtr.

base::Callback and base::Bind

Mojo consumer code uses callbacks heavily, and a lot of callback code is still on deprecated base::Callback and base::Bind APIs. While migrating code to new Mojo types, please consider also migrating surrounding use of these APIs to base::OnceCallback/base::RepeatingCallback and base::BindOnce/base::BindRepeating, respectively.