Mojo C++ System API

This document is a subset of the Mojo documentation.

Contents

Overview

The Mojo C++ System API provides a convenient set of helper classes and functions for working with Mojo primitives. Unlike the low-level C API (upon which this is built) this library takes advantage of C++ language features and common STL and `//base` types to provide a slightly more idiomatic interface to the Mojo system layer, making it generally easier to use.

This document provides a brief guide to API usage with example code snippets. For a detailed API references please consult the headers in //mojo/public/cpp/system.

Note that all API symbols referenced in this document are implicitly in the top-level `mojo` namespace.

Scoped, Typed Handles

All types of Mojo handles in the C API are simply opaque, integral `MojoHandle` values. The C++ API has more strongly typed wrappers defined for different handle types: `MessagePipeHandle`,

`SharedBufferHandle`, `DataPipeConsumerHandle`, `DataPipeProducerHandle`, `TrapHandle`, and `InvitationHandle`.

Each of these also has a corresponding, move-only, scoped type for safer usage: `ScopedMessagePipeHandle`, `ScopedSharedBufferHandle`, and so on. When a scoped handle type is destroyed, its handle is automatically closed via `MojoClose`. When working with raw handles you should **always** prefer to use one of the scoped types for ownership.

Similar to `std::unique_ptr`, scoped handle types expose a `get()` method to get at the underlying unscoped handle type as well as the `->` operator to dereference the scoper and make calls directly on the underlying handle type.

Message Pipes

There are two ways to create a new message pipe using the C++ API. You may construct a `MessagePipe` object:

```
mojo::MessagePipe pipe;

// NOTE: Because pipes are bi-directional there is no implicit semantic
// difference between |handle0| or |handle1| here. They're just two ends of a
// pipe. The choice to treat one as a "client" and one as a "server" is entirely
// the API user's decision.
mojo::ScopedMessagePipeHandle client = std::move(pipe.handle0);
mojo::ScopedMessagePipeHandle server = std::move(pipe.handle1);
```

or you may call `CreateMessagePipe`:

```
mojo::ScopedMessagePipeHandle client;
mojo::ScopedMessagePipeHandle server;
mojo::CreateMessagePipe(nullptr, &client, &server);
```

There are also some helper functions for constructing message objects and reading/writing them on pipes using the library's more strongly-typed C++ handles:

```
mojo::ScopedMessageHandle message;
mojo::AllocMessage(6, nullptr, 0, MOJO_ALLOC_MESSAGE_FLAG_NONE, &message);

void *buffer;
mojo::GetMessageBuffer(message.get(), &buffer);

const std::string kMessage = "hello";
std::copy(kMessage.begin(), kMessage.end(), static_cast<char*>(buffer));

mojo::WriteMessageNew(client.get(), std::move(message),
                      MOJO_WRITE_MESSAGE_FLAG_NONE);
```

```
// Some time later...

mojo::ScopedMessageHandle received_message;
uint32_t num_bytes;
mojo::ReadMessageNew(server.get(), &received_message, &num_bytes, nullptr,
                     nullptr, MOJO_READ_MESSAGE_FLAG_NONE);
```

See message_pipe.h for detailed C++ message pipe API documentation.

Data Pipes

Similar to Message Pipes, the C++ library has some simple helpers for more strongly-typed data pipe usage:

```
mojo::DataPipe pipe;
mojo::ScopedDataPipeProducerHandle producer = std::move(pipe.producer_handle);
mojo::ScopedDataPipeConsumerHandle consumer = std::move(pipe.consumer_handle);

// Or alternatively:
mojo::ScopedDataPipeProducerHandle producer;
mojo::ScopedDataPipeConsumerHandle consumer;
mojo::CreateDataPipe(null, &producer, &consumer);
```

C++ helpers which correspond directly to the Data Pipe C API for immediate and two-phase I/O are provided as well. For example:

```
uint32_t num_bytes = 7;
producer.WriteData("hihihi", &num_bytes, MOJO_WRITE_DATA_FLAG_NONE);

// Some time later...

char buffer[64];
uint32_t num_bytes = 64;
consumer.ReadData(buffer, &num_bytes, MOJO_READ_DATA_FLAG_NONE);
```

See data_pipe.h for detailed C++ data pipe API documentation.

Shared Buffers

A new shared buffer can be allocated like so:

```
mojo::ScopedSharedBufferHandle buffer =
    mojo::SharedBufferHandle::Create(4096);
```

This new handle can be cloned arbitrarily many times by using the underlying handle's `Clone` method:

```
mojo::ScopedSharedBufferHandle another_handle = buffer->Clone();
mojo::ScopedSharedBufferHandle read_only_handle =
    buffer->Clone(mojo::SharedBufferHandle::AccessMode::READ_ONLY);
```

And finally the library also provides a scoper for mapping the shared buffer's memory:

```
mojo::ScopedSharedBufferMapping mapping = buffer->Map(64);
static_cast<int*>(mapping.get()) = 42;

mojo::ScopedSharedBufferMapping another_mapping = buffer->MapAtOffset(64, 4);
static_cast<int*>(mapping.get()) = 43;
```

When `mapping` and `another_mapping` are destroyed, they automatically unmap their respective memory regions.

See buffer.h for detailed C++ shared buffer API documentation.

Native Platform Handles (File Descriptors, Windows Handles, *etc.*)

The C++ library provides several helpers for wrapping system handle types. These are specifically useful when working with a few `//base` types, namely `base::PlatformFile`, `base::SharedMemoryHandle` (deprecated), and various strongly-typed shared memory region types like `base::ReadOnlySharedMemoryRegion`. See platform_handle.h for detailed C++ platform handle API documentation.

Signals & Traps

For an introduction to the concepts of handle signals and traps, check out the C API's documentation on Signals & Traps.

Querying Signals

Any C++ handle type's last known signaling state can be queried by calling the `QuerySignalsState` method on the handle:

```
mojo::MessagePipe message_pipe;
mojo::DataPipe data_pipe;
mojo::HandleSignalsState a = message_pipe.handle0->QuerySignalsState();
mojo::HandleSignalsState b = data_pipe.consumer->QuerySignalsState();
```

The `HandleSignalsState` is a thin wrapper interface around the C API's `MojoHandleSignalsState` structure with convenient accessors for testing the signal bitmasks. Whereas when using the C API you might write:

```
struct MojoHandleSignalsState state;
MojoQueryHandleSignalsState(handle0, &state);
if (state.satisfied_signals & MOJO_HANDLE_SIGNAL_READABLE) {
  // ...
}
```

the C++ API equivalent would be:

```
if (message_pipe.handle0->QuerySignalsState().readable()) {
  // ...
}
```

Watching Handles

The `mojo::SimpleWatcher` class serves as a convenient helper for using the low-level traps API to watch a handle for signaling state changes. A `SimpleWatcher` is bound to a single sequence and always dispatches its notifications on a `base::SequencedTaskRunner`.

`SimpleWatcher` has two possible modes of operation, selected at construction time by the `mojo::SimpleWatcher::ArmingPolicy` enum:

- `MANUAL` mode requires the user to manually call `Arm` and/or `ArmOrNotify` before any notifications will fire regarding the state of the watched handle. Every time the notification callback is run, the `SimpleWatcher` must be rearmed again before the next one can fire. See Arming a Trap and the documentation in `SimpleWatcher`'s header.
- `AUTOMATIC` mode ensures that the `SimpleWatcher` always either is armed or has a pending notification task queued for execution.

`AUTOMATIC` mode is more convenient but can result in redundant notification tasks, especially if the provided callback does not make a strong effort to return the watched handle to an uninteresting signaling state (by *e.g.*, reading all its available messages when notified of readability.)

Example usage:

```
class PipeReader {
 public:
  PipeReader(mojo::ScopedMessagePipeHandle pipe)
      : pipe_(std::move(pipe)),
        watcher_(mojo::SimpleWatcher::ArmingPolicy::AUTOMATIC) {
    // NOTE: base::Unretained is safe because the callback can never be run
    // after SimpleWatcher destruction.
    watcher_.Watch(pipe_.get(), MOJO_HANDLE_SIGNAL_READABLE,
                   base::Bind(&PipeReader::OnReadable, base::Unretained(this)));
  }

  ~PipeReader() {}
```

```
 private:
  void OnReadable(MojoResult result) {
    while (result == MOJO_RESULT_OK) {
      mojo::ScopedMessageHandle message;
      uint32_t num_bytes;
      result = mojo::ReadMessageNew(pipe_.get(), &message, &num_bytes, nullptr,
                                    nullptr, MOJO_READ_MESSAGE_FLAG_NONE);
      DCHECK_EQ(result, MOJO_RESULT_OK);
      messages_.emplace_back(std::move(message));
    }
  }

  mojo::ScopedMessagePipeHandle pipe_;
  mojo::SimpleWatcher watcher_;
  std::vector<mojo::ScopedMessageHandle> messages_;
};

mojo::MessagePipe pipe;
PipeReader reader(std::move(pipe.handle0));

// Written messages will asynchronously end up in |reader.messages_|.
WriteABunchOfStuff(pipe.handle1.get());
```

Synchronous Waiting

The C++ System API defines some utilities to block a calling sequence while waiting for one or more handles to change signaling state in an interesting way. These threads combine usage of the low-level traps API with common synchronization primitives (namely `base::WaitableEvent`.)

While these API features should be used sparingly, they are sometimes necessary.

See the documentation in wait.h and wait_set.h for a more detailed API reference.

Waiting On a Single Handle

The `mojo::Wait` function simply blocks the calling sequence until a given signal mask is either partially satisfied or fully unsatisfiable on a given handle.

```
mojo::MessagePipe pipe;
mojo::WriteMessageRaw(pipe.handle0.get(), "hey", 3, nullptr, nullptr,
                      MOJO_WRITE_MESSAGE_FLAG_NONE);
MojoResult result = mojo::Wait(pipe.handle1.get(), MOJO_HANDLE_SIGNAL_READABLE);
DCHECK_EQ(result, MOJO_RESULT_OK);

// Guaranteed to succeed because we know |handle1| is readable now.
mojo::ScopedMessageHandle message;
uint32_t num_bytes;
```

```
mojo::ReadMessageNew(pipe.handle1.get(), &num_bytes, nullptr, nullptr,
                     MOJO_READ_MESSAGE_FLAG_NONE);
```

`mojo::Wait` is most typically useful in limited testing scenarios.

Waiting On Multiple Handles

`mojo::WaitMany` provides a simple API to wait on multiple handles simultaneously, returning when any handle's given signal mask is either partially satisfied or fully unsatisfiable.

```
mojo::MessagePipe a, b;
GoDoSomethingWithPipes(std:move(a.handle1), std::move(b.handle1));

mojo::MessagePipeHandle handles[2] = {a.handle0.get(), b.handle0.get()};
MojoHandleSignals signals[2] = {MOJO_HANDLE_SIGNAL_READABLE,
                                MOJO_HANDLE_SIGNAL_READABLE};
size_t ready_index;
MojoResult result = mojo::WaitMany(handles, signals, 2, &ready_index);
if (ready_index == 0) {
  // a.handle0 was ready.
} else {
  // b.handle0 was ready.
}
```

Similar to `mojo::Wait`, `mojo::WaitMany` is primarily useful in testing. When waiting on multiple handles in production code, you should almost always instead use a more efficient and more flexible `mojo::WaitSet` as described in the next section.

Waiting On Handles and Events Simultaneously

Typically when waiting on one or more handles to signal, the set of handles and conditions being waited upon do not change much between consecutive blocking waits. It's also often useful to be able to interrupt the blocking operation as efficiently as possible.

`mojo::WaitSet` is designed with these conditions in mind. A `WaitSet` maintains a persistent set of (not-owned) Mojo handles and `base::WaitableEvent`s, which may be explicitly added to or removed from the set at any time.

The `WaitSet` may be waited upon repeatedly, each time blocking the calling sequence until either one of the handles attains an interesting signaling state or one of the events is signaled. For example let's suppose we want to wait up to 5 seconds for either one of two handles to become readable:

```
base::WaitableEvent timeout_event(
    base::WaitableEvent::ResetPolicy::MANUAL,
    base::WaitableEvent::InitialState::NOT_SIGNALED);
mojo::MessagePipe a, b;

GoDoStuffWithPipes(std::move(a.handle1), std::move(b.handle1));
```

```
mojo::WaitSet wait_set;
wait_set.AddHandle(a.handle0.get(), MOJO_HANDLE_SIGNAL_READABLE);
wait_set.AddHandle(b.handle0.get(), MOJO_HANDLE_SIGNAL_READABLE);
wait_set.AddEvent(&timeout_event);

// Ensure the Wait() lasts no more than 5 seconds.
bg_thread->task_runner()->PostDelayedTask(
    FROM_HERE,
    base::Bind([](base::WaitableEvent* e) { e->Signal(); }, &timeout_event);
    base::TimeDelta::FromSeconds(5));

base::WaitableEvent* ready_event = nullptr;
size_t num_ready_handles = 1;
mojo::Handle ready_handle;
MojoResult ready_result;
wait_set.Wait(&ready_event, &num_ready_handles, &ready_handle, &ready_result);

// The apex of thread-safety.
bg_thread->Stop();

if (ready_event) {
  // The event signaled...
}

if (num_ready_handles > 0) {
  // At least one of the handles signaled...
  // NOTE: This and the above condition are not mutually exclusive. If handle
  // signaling races with timeout, both things might be true.
}
```

Invitations

Invitations are the means by which two processes can have Mojo IPC bootstrapped between them. An invitation must be transmitted over some platform-specific IPC primitive (*e.g.* a Windows named pipe or UNIX domain socket), and the public platform support library provides some lightweight, cross-platform abstractions for those primitives.

For any two processes looking to be connected, one must send an `OutgoingInvitation` and the other must accept an `IncomingInvitation`. The sender can attach named message pipe handles to the `OutgoingInvitation`, and the receiver can extract them from its `IncomingInvitation`.

Basic usage might look something like this in the case where one process is responsible for launching the other.

```
#include "base/command_line.h"
#include "base/process/launch.h"
#include "mojo/public/cpp/platform/platform_channel.h"
```

```
#include "mojo/public/cpp/system/invitation.h"
#include "mojo/public/cpp/system/message_pipe.h"

mojo::ScopedMessagePipeHandle LaunchAndConnectSomething() {
  // Under the hood, this is essentially always an OS pipe (domain socket pair,
  // Windows named pipe, Fuchsia channel, etc).
  mojo::PlatformChannel channel;

  mojo::OutgoingInvitation invitation;

  // Attach a message pipe to be extracted by the receiver. The other end of the
  // pipe is returned for us to use locally.
  mojo::ScopedMessagePipeHandle pipe =
      invitation->AttachMessagePipe("arbitrary pipe name");

  base::LaunchOptions options;
  base::CommandLine command_line("some_executable")
  channel.PrepareToPassRemoteEndpoint(&options, &command_line);
  base::Process child_process = base::LaunchProcess(command_line, options);
  channel.RemoteProcessLaunchAttempted();

  OutgoingInvitation::Send(std::move(invitation), child_process.Handle(),
                           channel.TakeLocalEndpoint());
  return pipe;
}
```

The launched process can in turn accept an `IncomingInvitation`:

```
#include "base/command_line.h"
#include "base/threading/thread.h"
#include "mojo/core/embedder/embedder.h"
#include "mojo/core/embedder/scoped_ipc_support.h"
#include "mojo/public/cpp/platform/platform_channel.h"
#include "mojo/public/cpp/system/invitation.h"
#include "mojo/public/cpp/system/message_pipe.h"

int main(int argc, char** argv) {
  // Basic Mojo initialization for a new process.
  mojo::core::Init();
  base::Thread ipc_thread("ipc!");
  ipc_thread.StartWithOptions(
      base::Thread::Options(base::MessageLoop::TYPE_IO, 0));
  mojo::core::ScopedIPCSupport ipc_support(
      ipc_thread.task_runner(),
      mojo::core::ScopedIPCSupport::ShutdownPolicy::CLEAN);

  // Accept an invitation.
  mojo::IncomingInvitation invitation = mojo::IncomingInvitation::Accept(
```

```
      mojo::PlatformChannel::RecoverPassedEndpointFromCommandLine(
          *base::CommandLine::ForCurrentProcess()));
  mojo::ScopedMessagePipeHandle pipe =
      invitation->ExtractMessagePipe("arbitrary pipe name");

  // etc...
  return GoListenForMessagesAndRunForever(std::move(pipe));
}
```

Now we have IPC initialized between the two processes.

Also keep in mind that bindings interfaces are just message pipes with some semantic and syntactic sugar wrapping them, so you can use these primordial message pipe handles as mojom interfaces. For example:

```
// Process A
mojo::OutgoingInvitation invitation;
auto pipe = invitation->AttachMessagePipe("x");
mojo::Binding<foo::mojom::Bar> binding(
    &bar_impl,
    foo::mojom::BarRequest(std::move(pipe)));

// Process B
auto invitation = mojo::IncomingInvitation::Accept(...);
auto pipe = invitation->ExtractMessagePipe("x");
foo::mojom::BarPtr bar(foo::mojom::BarPtrInfo(std::move(pipe), 0));

// Will asynchronously invoke bar_impl.DoSomething() in process A.
bar->DoSomething();
```

And just to be sure, the usage here could be reversed: the invitation sender could just as well treat its pipe endpoint as a `BarPtr` while the receiver treats theirs as a `BarRequest` to be bound.

Process Networks

Accepting an invitation admits the accepting process into the sender's connected network of processes. Once this is done, it's possible for the newly admitted process to establish communication with any other process in that network via normal message pipe passing.

This does not mean that the invited process can proactively locate and connect to other processes without assistance; rather it means that Mojo handles created by the process can safely be transferred to any other process in the network over established message pipes, and similarly that Mojo handles created by any other process in the network can be safely passed to the newly admitted process.

Invitation Restrictions

A process may only belong to a single network at a time.

Additionally, once a process has joined a network, it cannot join another for the remainder of its lifetime even if it has lost the connection to its original network. This restriction will soon be lifted, but for now

developers must be mindful of it when authoring any long-running daemon process that will accept an incoming invitation.

Isolated Invitations

It is possible to have two independent networks of Mojo-connected processes; for example, a long-running system daemon which uses Mojo to talk to child processes of its own, as well as the Chrome browser process running with no common ancestor, talking to its own child processes.

In this scenario it may be desirable to have a process in one network talk to a process in the other network. Normal invitations cannot be used here since both processes already belong to a network. In this case, an **isolated** invitation can be used. These work just like regular invitations, except the sender must call `OutgoingInvitation::SendIsolated` and the receiver must call `IncomingInvitation::AcceptIsolated`.

Once a connection is established via isolated invitation, Mojo IPC can be used normally, with the exception that transitive process connections are not supported; that is, if process A sends a message pipe handle to process B via an isolated connection, process B cannot reliably send that pipe handle onward to another process in its own network. Isolated invitations therefore may only be used to facilitate direct 1:1 communication between two processes.