Mojo

Contents

Getting Started With Mojo

To get started using Mojo in Chromium, the fastest path forward will likely be to read the Mojo sections of the Intro to Mojo & Services guide.

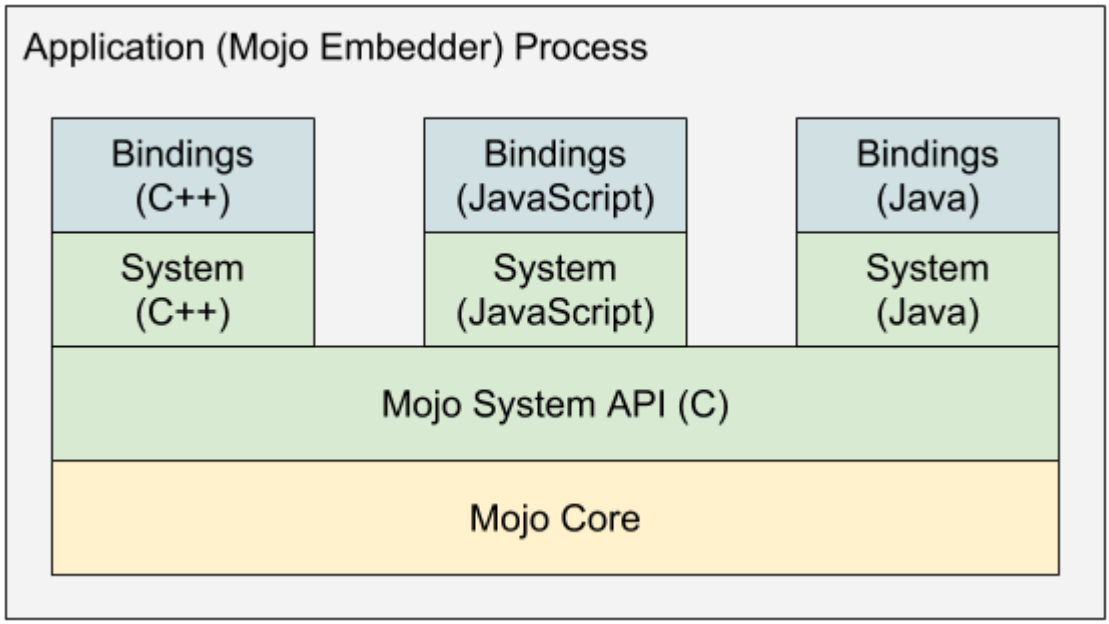
For more detailed reference material on the most commonly used features of Mojo, head directly to the bindings documentation for your language of choice or the more general mojom Interface Definition Language (IDL) documentation.

If you're looking for information on creating and/or connecting to services, you're in the wrong place! Mojo does not deal with services, it only facilitates interface definition, message passing, and other lower-level IPC primitives. Instead, you should take a look at some of the other available Mojo & Services documentation.

System Overview

Mojo is a collection of runtime libraries providing a platform-agnostic abstraction of common IPC primitives, a message IDL format, and a bindings library with code generation for multiple target languages to facilitate convenient message passing across arbitrary inter- and intra-process boundaries.

The documentation here is segmented according to the different libraries comprising Mojo. Mojo is divided into cleanly-separated layers with the basic hierarchy of subcomponents as follows:



Mojo Core

In order to use any of the more interesting high-level support libraries like the System APIs or Bindings APIs, a process must first initialize Mojo Core. This is a one-time initialization which remains active for the remainder of the process's lifetime. There are two ways to initialize Mojo Core: via the Embedder API, or through a dynamically linked library.

Embedding

Many processes to be interconnected via Mojo are **embedders**, meaning that they statically link against the `//mojo/core/embedder` target and initialize Mojo support within each process by calling `mojo::core::Init()`. See **Mojo Core Embedder API** for more details.

This is a reasonable option when you can guarantee that all interconnected process binaries are linking against precisely the same revision of Mojo Core. This includes Chromium itself as well as any developer tools and test executables built within the tree.

To support other scenarios, use dynamic linking.

Dynamic Linking

On some platforms, it's also possible for applications to rely on a dynamically-linked Mojo Core library (`libmojo_core.so` or `mojo_core.dll`) instead of statically linking against Mojo Core.

In order to take advantage of this mechanism, the library's binary must be present in either:

- The working directory of the application
- A directory named by the `MOJO_CORE_LIBRARY_PATH` environment variable
- A directory named explicitly by the application at runtime

Instead of calling `mojo::core::Init()` as embedders do, an application using dynamic Mojo Core instead calls `MojoInitialize()` from the C System API. This call will attempt to locate (see above) and load the Mojo Core library to support subsequent Mojo API usage within the process.

Note that the Mojo Core shared library presents a **stable C ABI** designed with both forward- and backward-compatibility in mind. Thus old applications will work with new versions of the shared library, and new applications can work with old versions of the shared library (modulo any dependency on newer features, whose absence can be gracefully detected at runtime).

C System API

Once Mojo is initialized within a process, the public **C System API** is usable on any thread for the remainder of the process's lifetime. This encapsulates Mojo Core's stable ABI and comprises the total public API surface of the Mojo Core library.

The C System library's only dependency (apart from the system libc and e.g. pthreads) is Mojo Core itself. As such, it's possible build a fully-featured multiprocess system using only Mojo Core and its exposed C API. It exposes the fundamental cross-platform capabilities to create and manipulate Mojo primitives like **message pipes**, **data pipes**, and **shared buffers**, as well as APIs to help bootstrap connections among processes.

Despite this, it's rare for applications to use the C API directly. Instead this API acts as a stable foundation upon which several higher-level and more ergonomic Mojo libraries are built.

Platform Support API

Mojo provides a small collection of abstractions around platform-specific IPC primitives to facilitate bootstrapping Mojo IPC between two processes. See the Platform API documentation for details.

Higher-Level System APIs

There is a relatively small, higher-level system API for each supported language, built upon the low-level C API. Like the C API, direct usage of these system APIs is rare compared to the bindings APIs, but it is sometimes desirable or necessary.

These APIs provide wrappers around low-level system API concepts, presenting interfaces that are more idiomatic for the target language:

- **C++ System API**
- **JavaScript System API**
- **Java System API**

Bindings APIs

The **mojom Interface Definition Language (IDL)** is used to generate interface bindings for various languages to send and receive mojom interface messages using Mojo message pipes. The generated code is supported by a language-specific bindings API:

- **C++ Bindings API**
- **JavaScript Bindings API**
- **Java Bindings API**

Note that the C++ bindings see the broadest usage in Chromium and are thus naturally the most feature-rich, including support for things like associated interfaces, synchronous calls, and type-mapping.

FAQ

Why not protobuf? Why a new thing?

There are number of potentially decent answers to this question, but the deal-breaker is that a useful IPC mechanism must support transfer of native object handles (*e.g.* file descriptors) across process boundaries. Other non-new IPC things that do support this capability (*e.g.* D-Bus) have their own substantial deficiencies.

Are message pipes expensive?

No. As an implementation detail, creating a message pipe is essentially generating two random numbers and stuffing them into a hash table, along with a few tiny heap allocations.

So really, can I create like, thousands of them?

Yes! Nobody will mind. Create millions if you like. (OK but maybe don't.)

What are the performance characteristics of Mojo?

Compared to the old IPC in Chrome, making a Mojo call is about 1/3 faster and uses 1/3 fewer context switches. The full data is available here.

Can I use in-process message pipes?

Yes, and message pipe usage is identical regardless of whether the pipe actually crosses a process boundary -- in fact the location of the other end of a pipe is intentionally obscured, in part for the sake of efficiency, and in part to discourage tight coupling of application logic to such details.

Message pipes which don't cross a process boundary are efficient: sent messages are never copied, and a write on one end will synchronously modify the message queue on the other end. When working with generated C++ bindings, for example, the net result is that a `Remote` on one thread sending a message to a `Receiver` on another thread (or even the same thread) is effectively a `PostTask` to the `Binding`'s `TaskRunner` with the added -- but often small -- costs of serialization, deserialization, validation, and some internal routing logic.

What about ___?

Please post questions to `chromium-mojo@chromium.org`! The list is quite responsive.