How Blink works

bit.ly/how-blink-works
Author: haraken@
Last update: 2018 Aug 14
Status: PUBLIC

Working on Blink is not easy. It's not easy for new Blink developers because there are a lot of Blink-specific concepts and coding conventions that have been introduced to implement a very fast rendering engine. It's not easy even for experienced Blink developers because Blink is huge and extremely sensitive to performance, memory and security.

This document aims at providing a **10k foot overview of "how Blink works"**, which I hope will help Blink developers get familiar with the architecture quickly:

- The document is NOT a thorough tutorial of Blink's detailed architectures and coding rules (which are likely to change and be outdated). Rather the document concisely describes Blink's fundamentals that are not likely to change in the short term and points out resources you can read if you want to learn more.
- The document does NOT explain specific features (e.g., ServiceWorkers, editing). Rather the document explains fundamental features used by a broad range of the code base (e.g., memory management, V8 APIs).

For more general information about Blink's development, see the Chromium wiki page.

What Blink does

Process / thread architecture
- Processes
- Threads
- Initialization of Blink

Directory structure
- Content public APIs and Blink public APIs
- Directory structure and dependencies
- WTF

Memory management

Task scheduling

Page, Frame, Document, DOMWindow etc
- Concepts

    OOPIF
    Detached Frame / Document

Web IDL bindings

V8 and Blink
    Isolate, Context, World
    V8 APIs
    V8 wrappers

Rendering pipeline

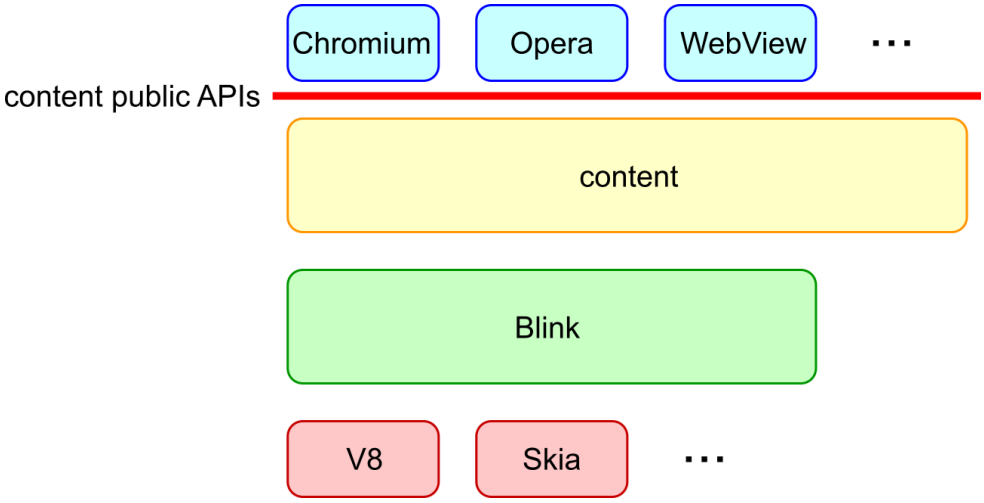
Questions?

What Blink does

Blink is a rendering engine of the web platform. Roughly speaking, Blink implements everything that renders content inside a browser tab:

- Implement the specs of the web platform (e.g., HTML standard), including DOM, CSS and Web IDL
- Embed V8 and run JavaScript
- Request resources from the underlying network stack
- Build DOM trees
- Calculate style and layout
- Embed Chrome Compositor and draw graphics

Blink is embedded by many customers such as Chromium, Android WebView and Opera via content public APIs.



From the code base perspective, "Blink" normally means //third_party/blink/. From the project perspective, "Blink" normally means projects that implement web platform features. Code that implements web platform features span //third_party/blink/, //content/renderer/, //content/browser/ and other places.

Process / thread architecture

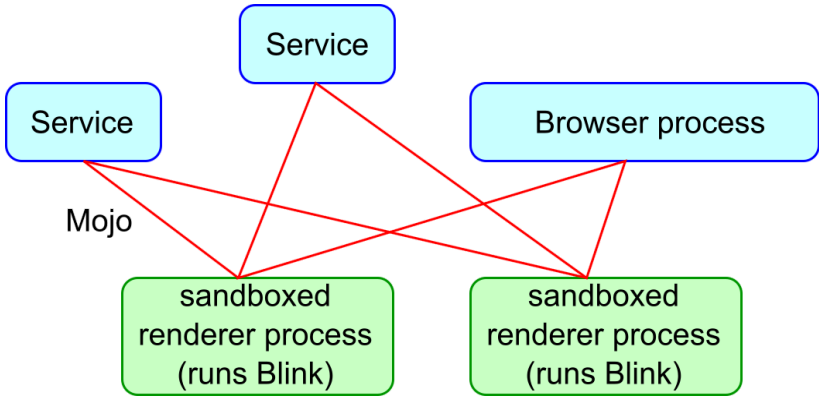
Processes

Chromium has a multi-process architecture. Chromium has one browser process and N sandboxed renderer processes. Blink runs in a renderer process.

How many renderer processes are created? For security reasons, it is important to isolate memory address regions between cross-site documents (this is called Site Isolation). Conceptually each renderer process should be dedicated to at most one site. Realistically, however, it's sometimes too heavy to limit each renderer process to a single site when users open too many tabs or the device does not have enough RAM. Then a renderer process may be shared by multiple iframes or tabs loaded from different sites. This means that iframes in one tab may be hosted by different renderer processes and that iframes in different tabs may be hosted by the same renderer process. **There is no 1:1 mapping between renderer processes, iframes and tabs**.

Given that a renderer process runs in a sandbox, Blink needs to ask the browser process to dispatch system calls (e.g., file access, play audio) and access user profile data (e.g., cookie, passwords). This browser-renderer process communication is realized by Mojo. (Note: In the past we were using Chromium IPC and a bunch of places are still using it. However, it's deprecated and uses Mojo under the hood.) On the Chromium side, Servicification is ongoing

and abstracting the browser process as a set of "service"s. From the Blink perspective, Blink can just use Mojo to interact with the services and the browser process.



If you want to learn more:

- Multi-process Architecture
- Mojo programming in Blink: platform/mojo/MojoProgrammingInBlink.md

Threads

How many threads are created in a renderer process?

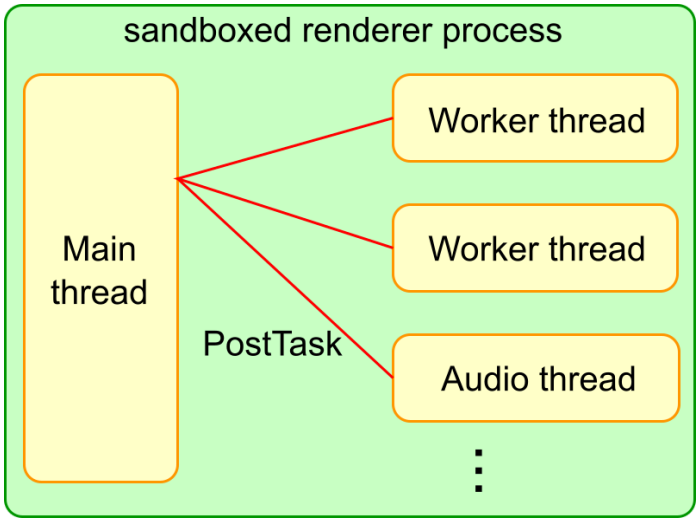
Blink has one main thread, N worker threads and a couple of internal threads.

Almost all important things happen on the main thread. All JavaScript (except workers), DOM, CSS, style and layout calculations run on the main thread. Blink is highly optimized to maximize the performance of the main thread, assuming the mostly single-threaded architecture.

Blink may create multiple worker threads to run Web Workers, ServiceWorker and Worklets.

Blink and V8 may create a couple of internal threads to handle webaudio, database, GC etc.

For cross-thread communications, you have to use message passing using PostTask APIs. Shared memory programming is discouraged except for a couple of places that really need to use it for performance reasons. This is why you don't see many MutexLocks in the Blink code base.



If you want to learn more:

- Thread programming in Blink: platform/wtf/ThreadProgrammingInBlink.md
- Workers: core/workers/README.md

Initialization & finalization of Blink

Blink is initialized by BlinkInitializer::Initialize(). This method must be called before executing any Blink code.

On the other hand, Blink is never finalized; i.e., the renderer process is forcibly exited without being cleaned up. One reason is performance. The other reason is in general it's really hard to clean up everything in the renderer process in a gracefully ordered manner (and it's not worth the effort).

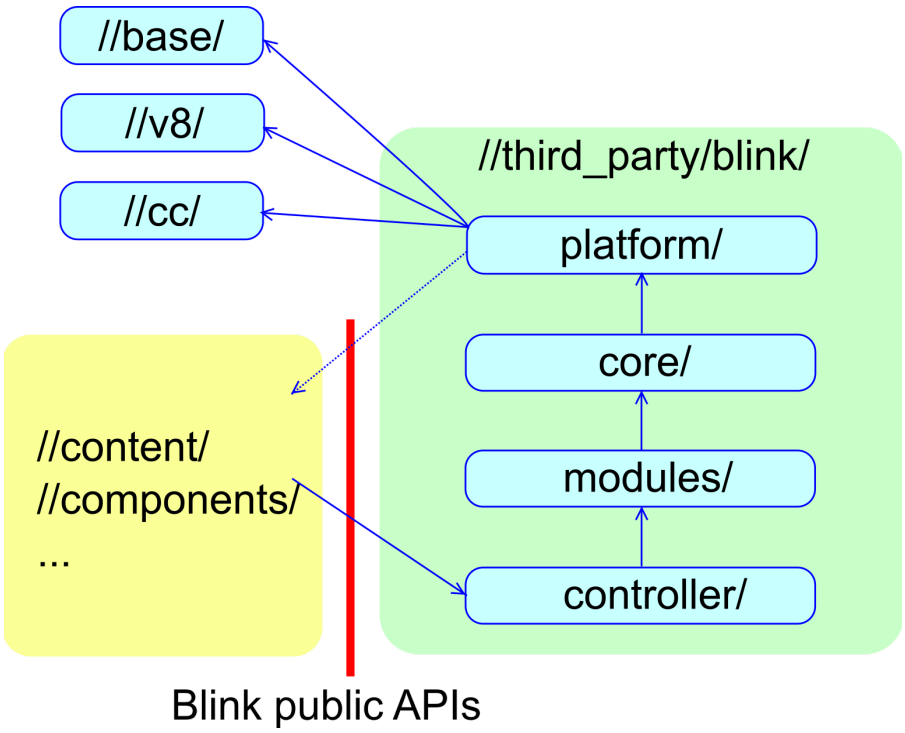
Directory structure

Content public APIs and Blink public APIs

Content public APIs are the API layer that enables embedders to embed the rendering engine. Content public APIs must be carefully maintained because they are exposed to embedders.

Blink public APIs are the API layer that exposes functionalities from //third_party/blink/ to Chromium. This API layer is just a historical artifact inherited from WebKit. In the WebKit era, Chromium and Safari shared the implementation of WebKit, so the API layer was needed to expose functionalities from WebKit to Chromium and Safari. Now that Chromium is the only embedder of //third_party/blink/, the API layer does not make sense. We're actively decreasing

# of Blink public APIs by moving web-platform code from Chromium to Blink (the project is called Onion Soup).



Directory structure and dependencies

//third_party/blink/ has the following directories. See this document for a more detailed definition of these directories:

- platform/
  - A collection of lower level features of Blink that are factored out of a monolithic core/. e.g., geometry and graphics utils.
- core/ and modules/
  - The implementation of all web-platform features defined in the specs. core/ implements features tightly coupled with DOM. modules/ implements more self-contained features. e.g. webaudio, indexeddb.
- bindings/core/ and bindings/modules/
  - Conceptually bindings/core/ is part of core/, and bindings/modules/ is part of modules/. Files that heavily use V8 APIs are put in bindings/{core,modules}.
- controller/
  - A set of high-level libraries that use core/ and modules/. e.g., devtools front-end.

Dependencies flow in the following order:

- Chromium => controller/ => modules/ and bindings/modules/ => core/ and bindings/core/ => platform/ => low-level primitives such as //base, //v8 and //cc

Blink carefully maintains the list of low-level primitives exposed to //third_party/blink/.

If you want to learn more:

- Directory structure and dependencies: blink/renderer/README.md

WTF

WTF is a "Blink-specific base" library and located at platform/wtf/. We are trying to unify coding primitives between Chromium and Blink as much as possible, so WTF should be small. This library is needed because there are a number of types, containers and macros that really need to be optimized for Blink's workload and Oilpan (Blink GC). If types are defined in WTF, Blink has to use the WTF types instead of types defined in //base or std libraries. The most popular ones are vectors, hashsets, hashmaps and strings. Blink should use WTF::Vector, WTF::HashSet, WTF::HashMap, WTF::String and WTF::AtomicString instead of std::vector, std::*set, std::*map and std::string.

If you want to learn more:

- How to use WTF: platform/wtf/README.md

Memory management

As far as Blink is concerned, you need to care about three memory allocators:

- PartitionAlloc
- Oil Pan (a.k.a. Blink GC)
- malloc/free or new/delete (banned)

You can allocate an object on PartitionAlloc's heap by using USING_FAST_MALLOC():

```
class SomeObject {
  USING_FAST_MALLOC(SomeObject);
  static std::unique_ptr<SomeObject> Create() {
    return std::make_unique<SomeObject>();  // Allocated on
PartitionAlloc's heap.
  }
};
```

The lifetime of objects allocated by PartitionAlloc should be managed by scoped_refptr<> or std::unique_ptr<>. It is strongly discouraged to manage the lifetime manually. Manual `delete` is banned in Blink.

You can allocate an object on Oilpan's heap by using GarbageCollected:

```
class SomeObject : public GarbageCollected<SomeObject> {
  static SomeObject* Create() {
    return new SomeObject;  // Allocated on Oilpan's heap.
  }
};
```

The lifetime of objects allocated by Oilpan is automatically managed by garbage collection. You have to use special pointers (e.g., Member<>, Persistent<>) to hold objects on Oilpan's heap. See this API reference to get familiar with programming restrictions about Oilpan. The most important restriction is that you are not allowed to touch any other Oilpan's object in a destructor of Oilpan's object (because the destruction order is not guaranteed).

If you use neither USING_FAST_MALLOC() nor GarbageCollected, objects are allocated on system malloc's heap. This is strongly discouraged in Blink. All Blink objects should be allocated by PartitionAlloc or Oilpan, as follows:

- Use Oilpan by default.
- Use PartitionAlloc only when 1) the lifetime of the object is very clear and std::unique_ptr<> or scoped_refptr<> is enough, 2) allocating the object on Oilpan introduces a lot of complexity or 3) allocating the object on Oilpan introduces a lot of unnecessary pressure to the garbage collection runtime.

Regardless of whether you use PartitionAlloc or Oilpan, you have to be really careful not to create dangling pointers (Note: raw pointers are strongly discouraged) or memory leaks.

If you want to learn more:

- How to use PartitionAlloc: platform/wtf/allocator/Allocator.md
- How to use Oilpan: platform/heap/BlinkGCAPIReference.md
- Oilpan GC design: platform/heap/BlinkGCDesign.md

Task scheduling

To improve responsiveness of the rendering engine, tasks in Blink should be executed asynchronously whenever possible. Synchronous IPC / Mojo and any other operations that may

take several milliseconds are discouraged (although some are inevitable e.g., user's JavaScript execution).

All tasks in a renderer process should be posted to Blink Scheduler with proper task types, like this:

```
// Post a task to frame's scheduler with a task type of kNetworking
frame->GetTaskRunner(TaskType::kNetworking)->PostTask(...,
WTF::Bind(&Function));
```

Blink Scheduler maintains multiple task queues and smartly prioritizes tasks to maximize user-perceived performance. It is important to specify proper task types to let Blink Scheduler schedule the tasks correctly and smartly.

If you want to learn more:

- How to post tasks: third_party/blink/renderer/platform/scheduler/TaskSchedulingInBlink.md

Page, Frame, Document, DOMWindow etc

Concepts

Page, Frame, Document, ExecutionContext and DOMWindow are the following concepts:

- A Page corresponds to a concept of a tab (if OOPIF explained below is not enabled). Each renderer process may contain multiple tabs.
- A Frame corresponds to a concept of a frame (the main frame or an iframe). Each Page may contain one or more Frames that are arranged in a tree hierarchy.
- A DOMWindow corresponds to a `window` object in JavaScript. Each Frame has one DOMWindow.
- A Document corresponds to a `window.document` object in JavaScript. Each Frame has one Document.
- An ExecutionContext is a concept that abstracts a Document (for the main thread) and a WorkerGlobalScope (for a worker thread).

Renderer process : Page = 1 : N.

Page : Frame = 1 : M.

Frame : DOMWindow : Document (or ExecutionContext) = 1 : 1 : 1 at any point in time, but the mapping may change over time. For example, consider the following code:

```
iframe.contentWindow.location.href = "https://example.com";
```

In this case, a new DOMWindow and a new Document are created for https://example.com. However, the Frame may be reused.

(Note: Precisely speaking, there are some cases where a new Document is created but the DOMWindow and the Frame are reused. There are even more complex cases.)

If you want to learn more:

- core/frame/FrameLifecycle.md

Out-of-Process iframes (OOPIF)

Site Isolation makes things more secure but even more complex. :) The idea of Site Isolation is to create one renderer process per site. (A *site* is a page's registrable domain + 1 label, and its URL scheme. For example, https://mail.example.com and https://chat.example.com are in the same site, but https://noodles.com and https://pumpkins.com are not.) If a Page contains one cross-site iframe, the Page may be hosted by two renderer processes. Consider the following page:

```
<!-- https://example.com -->
<body>
<iframe src="https://example2.com"></iframe>
</body>
```

The main frame and the <iframe> may be hosted by different renderer processes. A frame local to the renderer process is represented by LocalFrame and a frame not local to the renderer process is represented by RemoteFrame.

From the perspective of the main frame, the main frame is a LocalFrame and the <iframe> is a RemoteFrame. From the perspective of the <iframe>, the main frame is a RemoteFrame and the <iframe> is a LocalFrame.

Communications between a LocalFrame and RemoteFrame (which may exist in different renderer processes) are handled via the browser process.

If you want to learn more:

- Design docs: Site isolation design docs
- How to write code with site isolation: core/frame/SiteIsolation.md

Detached Frame / Document

Frame / Document may be in a detached state. Consider the following case:

```
doc = iframe.contentDocument;
iframe.remove();  // The iframe is detached from the DOM tree.
doc.createElement("div");  // But you still can run scripts on the
detached frame.
```

The tricky fact is that you can still run scripts or DOM operations on the detached frame. Since the frame has already been detached, most DOM operations will fail and throw errors. Unfortunately, behaviors on detached frames are not really interoperable among browsers nor well-defined in the specs. Basically the expectation is that JavaScript should keep running but most DOM operations should fail with some proper exceptions, like this:

```
void someDOMOperation(...) {
  if (!script_state_->ContextIsValid()) { // The frame is already
detached
    ...;  // Set an exception etc
    return;
  }
}
```

This means that in common cases Blink needs to do a bunch of clean-up operations when the frame gets detached. You can do this by inheriting from ContextLifecycleObserver, like this:

```
class SomeObject : public GarbageCollected<SomeObject>, public
ContextLifecycleObserver {
  void ContextDestroyed() override {
    // Do clean-up operations here.
  }
  ~SomeObject() {
    // It's not a good idea to do clean-up operations here because it's
too late to do them. Also a destructor is not allowed to touch any
other objects on Oilpan's heap.
  }
};
```

Web IDL bindings

When JavaScript accesses `node.firstChild`, `Node::firstChild()` in node.h gets called. How does it work? Let's take a look at how `node.firstChild` works.

First of all, you need to define an IDL file per the spec:

```
// node.idl
interface Node : EventTarget {
  [...] readonly attribute Node? firstChild;
};
```

The syntax of Web IDL is defined in the Web IDL spec. `[...]` is called IDL extended attributes. Some of the IDL extended attributes are defined in the Web IDL spec and others are Blink-specific IDL extended attributes. Except the Blink-specific IDL extended attributes, IDL files should be written in a spec-conformant manner (i.e., just copy and paste from the spec).

Second, you need to define a C++ class for Node and implement a C++ getter for firstChild:

```
class EventTarget : public ScriptWrappable {  // All classes exposed to
JavaScript must inherit from ScriptWrappable.
  ...;
};

class Node : public EventTarget {
  DEFINE_WRAPPERTYPEINFO();  // All classes that have IDL files must
have this macro.
  Node* firstChild() const { return first_child_; }
};
```

In common cases, that's it. When you build node.idl, the IDL compiler auto-generates Blink-V8 bindings for the Node interface and Node.firstChild. The auto-generated bindings are generated in //src/out/{Debug,Release}/gen/third_party/ blink/renderer/bindings/core/v8/v8_node.h. When JavaScript calls `node.firstChild`, V8 calls V8Node::firstChildAttributeGetterCallback() in v8_node.h, then it calls Node::firstChild() which you defined in the above.

If you want to learn more:

- How to add Web IDL bindings: bindings/IDLCompiler.md
- How to use IDL extended attributes: bindings/IDLExtendedAttributes.md
- Spec: Web IDL spec

V8 and Blink

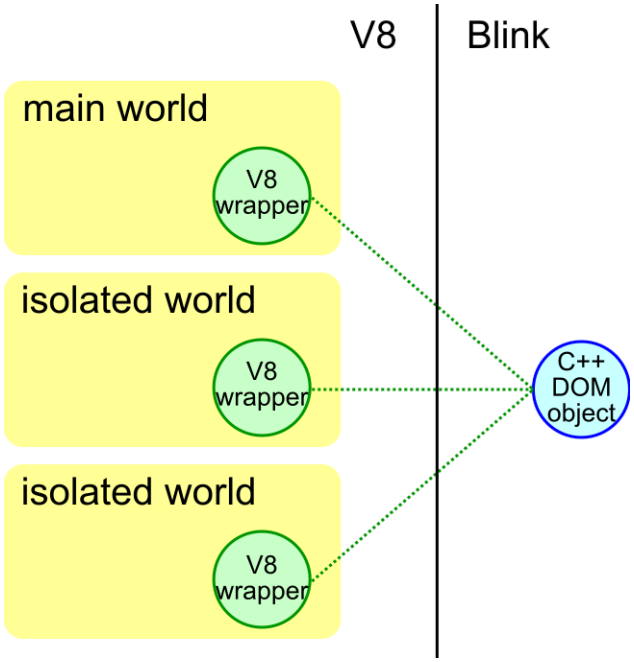
Isolate, Context, World

When you write code that touches V8 APIs, it is important to understand the concept of Isolate, Context and World. They are represented by v8::Isolate, v8::Context and DOMWrapperWorld in the code base respectively.

Isolate corresponds to a physical thread. Isolate : physical thread in Blink = 1 : 1. The main thread has its own Isolate. A worker thread has its own Isolate.

Context corresponds to a global object (In case of a Frame, it's a `window` object of the Frame). Since each frame has its own `window` object, there are multiple Contexts in a renderer process. When you call V8 APIs, you have to make sure that you're in the correct context. Otherwise, v8::Isolate::GetCurrentContext() will return a wrong context and in the worst case it will end up leaking objects and causing security issues.

World is a concept to support content scripts of Chrome extensions. Worlds do not correspond to anything in web standards. Content scripts want to share DOM with the web page, but for security reasons JavaScript objects of content scripts must be isolated from the JavaScript heap of the web page. (Also a JavaScript heap of one content script must be isolated from a JavaScript heap of another content script.) To realize the isolation, the main thread creates one main world for the web page and one isolated world for each content script. The main world and the isolated worlds can access the same C++ DOM objects but their JavaScript objects are isolated. This isolation is realized by creating multiple V8 wrappers for one C++ DOM object; i.e., one V8 wrapper per world.



What's a relationship between Context, World and Frame?

Imagine that there are N Worlds on the main thread (one main world + (N - 1) isolated worlds). Then one Frame should have N `window` objects, each of which is used for one world. Context is a concept that corresponds to a `window` object. This means that when we have M Frames and N Worlds, we have M * N Contexts (but the Contexts are created lazily).

In case of a worker, there is only one World and one global object. Thus there is only one Context.

Again, when you use V8 APIs, you should be really careful about using the correct context. Otherwise you'll end up leaking JavaScript objects between isolated worlds and causing security disasters (e.g., an extension from A.com can manipulate an extension from B.com).

If you want to learn more:

- bindings/core/v8/V8BindingDesign.md

V8 APIs

There are a lot of V8 APIs defined in //v8/include/v8.h. Since V8 APIs are low-level and hard to use correctly, platform/bindings/ provides a bunch of helper classes that wrap V8 APIs. You should consider using the helper classes as much as possible. If your code has to use V8 APIs heavily, the files should be put in bindings/{core,modules}.

V8 uses handles to point to V8 objects. The most common handle is v8::Local<>, which is used to point to V8 objects from a machine stack. v8::Local<> must be used after allocating v8::HandleScope on the machine stack. v8::Local<> should not be used outside the machine stack:

```
void function() {
  v8::HandleScope scope;
  v8::Local<v8::Object> object = ...;  // This is correct.
}

class SomeObject : public GarbageCollected<SomeObject> {
  v8::Local<v8::Object> object_;  // This is wrong.
};
```

If you want to point to V8 objects from outside the machine stack, you need to use wrapper tracing. However, you have to be really careful not to create a reference cycle with it. In general V8 APIs are hard to use. Ask blink-review-bindings@ if you're not sure about what you're doing.

If you want to learn more:

- How to use V8 APIs and helper classes: platform/bindings/HowToUseV8FromBlink.md

V8 wrappers

Each C++ DOM object (e.g., Node) has its corresponding V8 wrapper. Precisely speaking, each C++ DOM object has its corresponding V8 wrapper per world.

V8 wrappers have strong references to their corresponding C++ DOM objects. However, the C++ DOM objects have only weak references to the V8 wrappers. So if you want to keep V8 wrappers alive for a certain period of time, you have to do that explicitly. Otherwise, V8 wrappers will be prematurely collected and JS properties on the V8 wrappers will be lost...

```
div = document.getElementbyId("div");
child = div.firstChild;
child.foo = "bar";
child = null;
gc();  // If we don't do anything, the V8 wrapper of |firstChild| is
collected by the GC.
assert(div.firstChild.foo === "bar");  //...and this will fail.
```

If we don't do anything, `child` is collected by the GC and thus `child.foo` is lost. To keep the V8 wrapper of `div.firstChild` alive, we have to add a mechanism that "keeps the V8 wrapper of `div.firstChild` alive as long as the DOM tree which `div` belongs to is reachable from V8".

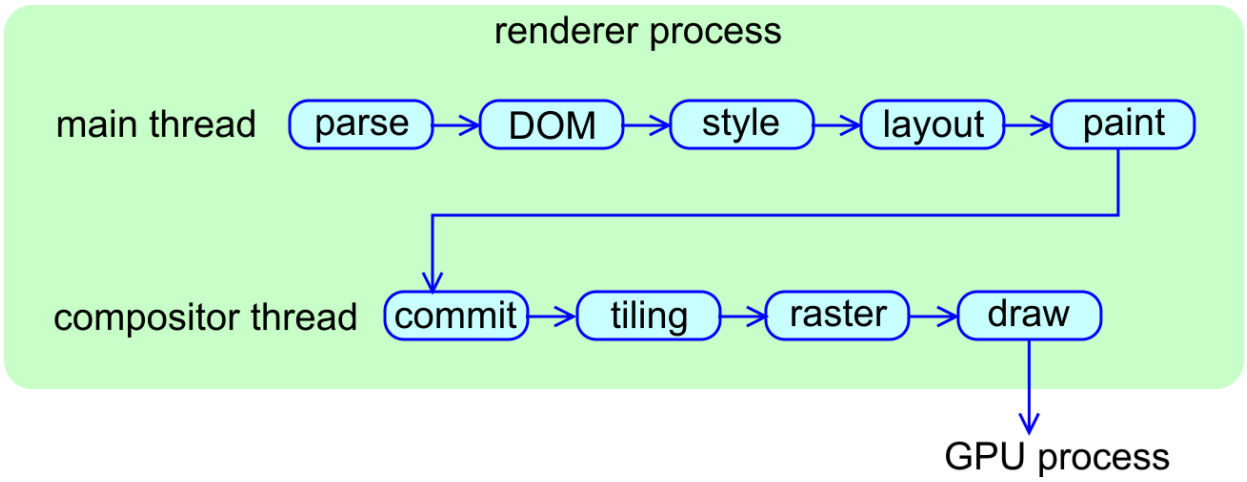
There are two ways to keep V8 wrappers alive: ActiveScriptWrappable and wrapper tracing.

If you want to learn more:

- How to manage lifetime of V8 wrappers: bindings/core/v8/V8Wrapper.md
- How to use wrapper tracing: platform/bindings/TraceWrapperReference.md

Rendering pipeline

There is a long journey from when an HTML file is delivered to Blink to when pixels are displayed on the screen. The rendering pipeline is architectured as follows.



Read this excellent deck to learn what each phase of the rendering pipeline does. (I don't think I can write a better explanation than the deck :-)

If you want to learn more:

- Overview: Life of a pixel
- DOM: core/dom/README.md
- Style: core/css/README.md
- Layout: core/layout/README.md
- Paint: core/paint/README.md
- Compositor thread: Chromium graphics

Questions?

You can ask any questions to blink-dev@chromium.org (for general questions) or platform-architecture-dev@chromium.org (for architecture-related questions). We're always happy to help! :D