Content Navigation API redesign

2015/7/6
clamy@chromium.org

Overview

The goal of this document is to present a new API for embedders of content/ interested in navigations. This new API defines a clearer interface for tracking and interacting with navigations, that will have the following benefits:

- simpler code for embedders of content/ that interact with navigation.
- greater dissociation between the interface for navigations and its actual implementation in content/. This is particularly important as this part of content/ is rapidly changing (in particular with Out-of-Process iFrames and PlzNavigate).

For quick reference (the tl;dr version) there is a cheat sheet showing which navigation observer methods were replaced and which were not.

Background

WebContentsObserver has several signals related to navigation. All are slightly different in their parameters, and lead to overly complex code for implementers of the API. In particular, there is no good way to keep track of a single navigation without reproducing some of the logic inside content/. This gets even more complex when realizing that there can be more than one navigation happening simultaneously in a FrameTreeNode (a class not part of the public content/ API), with one of them canceling the other.

For the record, the WebContentsObserver methods used to track navigations are:

- **WebContentsObserver::RenderFrameDeleted** / **WebContentsObserver::RenderViewDeleted** (to check for the cancellation of a cross-site navigation following a faster commit of a same-site navigation)

- **WebContentsObserver::AboutToNavigateRenderFrame** (deprecated but still used by DevTools for example)
- **WebContentsObserver::DidStartNavigationToPendingEntry** (only for browser-initiated navigations)
- **WebContentsObserver::DidStartProvisionalLoadForFrame**
- **WebContentsObserver::DidCommitProvisionalLoadForFrame**
- **WebContentsObserver::DidFailProvisionalLoad**
- **WebContentsObserver::DidNavigateMainFrame**
- **WebContentsObserver::DidNavigateAnyFrame**
- **WebContentsObserver::DidStartLoading**
- **WebContentsObserver::DidStopLoading**
- **WebContentsObserver::DidFinishLoad**
- **WebContentsObserver::DidFailLoad**
- **WebContentsObserver::NavigationEntryCommitted**

At the same time, a certain number of Chrome components are interested in network events happening during navigations. This can be done by implementing some WebContentsObserver methods:

- **WebContentsObserver::DidGetResourceResponseStart**
- **WebContentsObserver::DidGetRedirectForResourceRequest**

(note that some sub-systems in Chrome still use the obsolete notification system for that purpose).
Another option, especially if the component wants to act on the request, is to implement a **ResourceThrottle** and install it as a ResourceHandler in **ResourceDispatcherHost**. The Throttle can then pause the request at various points in time (**WillSendRequest**, **WillRedirectRequest**, **WillProcessResponse**). The Throttle then messages the component on the UI thread, where the navigation is identified by the RenderViewHost routing ID/RenderProcessHost child ID pair.

With the development of Out-of-Process IFrames and of browser-side navigation (aka PlzNavigate) this has a certain number of issues:

- the whole content/loader stack works with RenderViewHost ids instead of RenderFrameHost ids, which is a problem in OOPIF. It is also a problem with PlzNavigate, since requests made to the ResourceDispatcherHost cannot have a RFH id + child id. The only kind of ID available would be a FrameTreeNode id.
- in order to track navigations, several components (captive portal detector, WebNavigation API initially, etc.) have come to mimic what was once done in content by tracking the pending_render_view_host. This doesn't work with PlzNavigate.

Detailed design

We propose to expose a new API to embedders of content that will allow them to explicitly track a single navigation from start to finish, as well as interact with the request in a similar way the ResourceThrottles currently do on the IO thread. This will be achieved by exposing a new object, the **NavigationHandle** (or **NavigationInstance**: exact naming still TBD), along with a new interface: **NavigationThrottle**. The NavigationHandle would be passed as an argument to several new WebContentsObserver methods, that should eventually replace most of the current WebContentsObserver methods tied to navigation. In particular, all navigation-related methods tied to a RenderFrameHost should go away. Ideally, only DidStartLoading and DidStopLoading, which aggregates navigations on per-WebContents basis should remain.

The advantages of using a specific object instead of a list of parameters passed to WebContentsObserver methods are the following ones:

- Observers such as the WebNavigation API can explicitly track a single navigation by relying on the NavigationHandle address. Currently they do that using the RFH/provisional RFH; they will still need a way to do so in a new model. An alternative to an object for this purpose could be to provide a unique id on each call, but it will not address the two other following points.
- Other Observers are interested to know at commit time information that was given earlier, e.g. the net error code from DidFailProvisionalLoad. They usually implement their own tracking to do that. It would be simpler to have a single object explicitly implement the tracking, instead of various observers implementing it by themselves. This would lead to less duplicated code in the long term.
- Ultimately we want to not have ResourceThrottles in the network stack act on navigations. In order to provide a similar mechanism in the UI thread, it is easier to have a single object on which to register and call resume than separated calls in WebContents (each linked to a sort of navigation id).

NavigationHandle

The NavigationHandle is created when the navigation starts, and destroyed when the navigation finishes, ie when the navigation commits or is aborted. WebContentsObserver are informed of the start of a navigation in any of the frames of the WebContents by implementing a new method: **WebContentsObserver::DidStartNavigation**, which will include the newly created NavigationHandle object. For further interaction with the navigation, embedders should register a NavigationThrottle with the NavigationHandle.

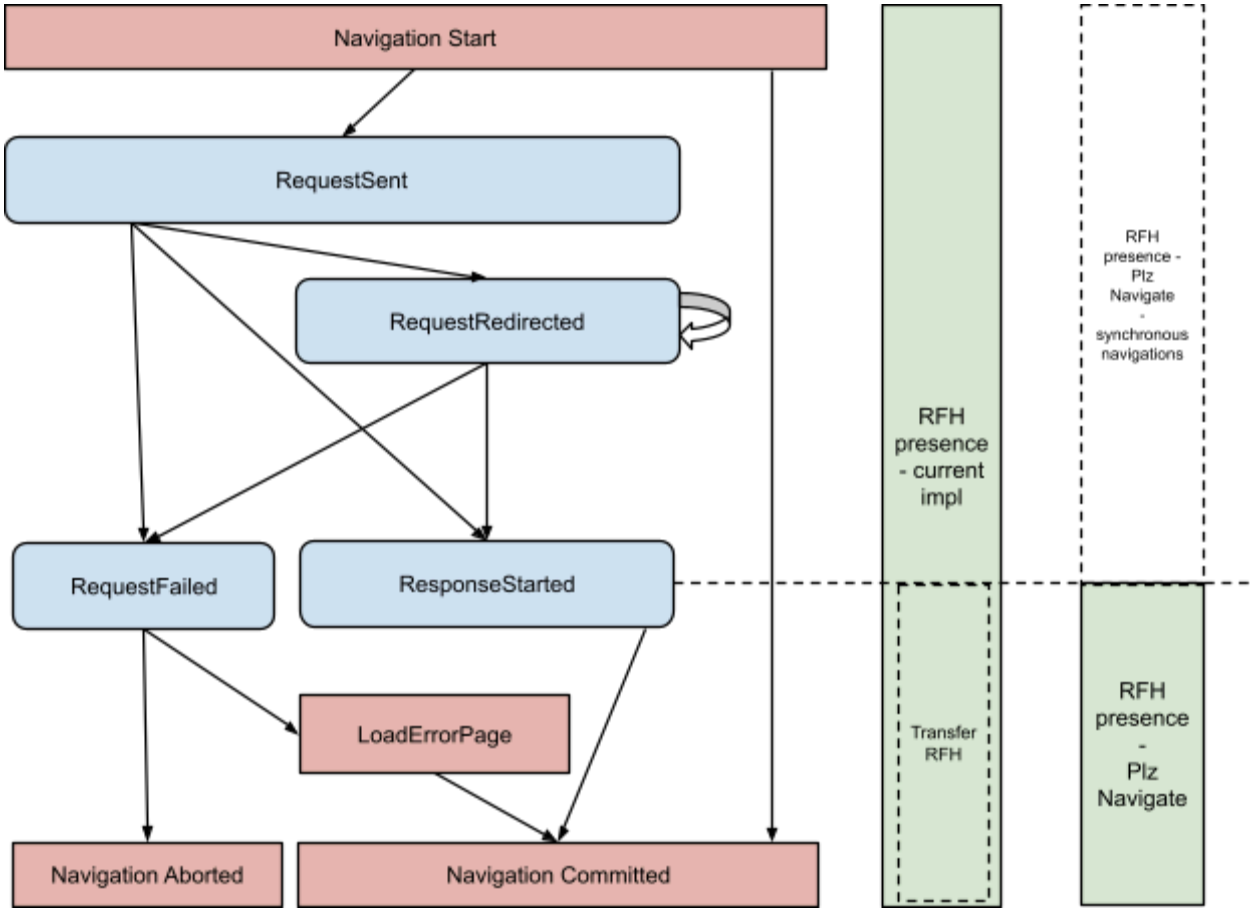
A navigation is associated with a **FrameTreeNode**, that remains an invariant during the whole navigation. Therefore the NavigationHandle should expose a const FrameTreeNode id.

The NavigationHandle should also expose several parameters associated with the navigation if/when they become available:

- **Original url + current URL:**
    - const GURL& GetOriginalURL()

    - const GURL& GetURL() (the current URL)
    - we may also store the chain of redirects if needed
- **RenderFrameHost** (once known - see graph below)
    - RenderFrameHost* GetRenderFrameHost()
- **FrameNavigateParams**
    - FrameNavigateParams* GetNavigateParams()
- **LoadDetails**
    - LoadCommittedDetails* GetLoadDetails()
- **Net error code in case of failure**
    - int GetNetErrorCode()
- **NavigationState** (see below)
    - NavigationHandle::State GetState()

The NavigationHandle should also keep track of the state the navigation is in, as can be seen in the following diagram:



The **NavigationState** should include:

- DID_START: this is the initial state
- WILL_SEND_REQUEST: a network request will be made. This allow the throttles to modify/cancel the request if needed.

- WILL_REDIRECT_REQUEST: the request will be redirected. This allow the throttles to modify/cancel the request if needed.
- DID_REDIRECT: the request was redirected.
- WILL_PROCESS_RESPONSE: a response was received in the network stack.
- DID_FAIL: the request in the network stack failed.
- DID_COMMIT: the navigation committed.
- DID_COMMIT_ERROR_PAGE: an error page was committed for the navigation failure.

The NavigationHandle should also have two methods used in conjunction with NavigationThrottles:

- **void Resume()** (for when the navigation was paused)
- **void Abort()** (to stop a navigation before commit)

New WebContentsObserver methods

The WebContentsObserver interface should have the following methods:

- void DidStartNavigation(NavigationHandle* navigation_handle)
- void DidRedirectNavigation(NavigationHandle* navigation_handle)
- void DidFailNavigation(NavigationHandle* navigation_handle)
- void DidCommitNavigation(NavigationHandle* navigation_handle)
- void DidFinishNavigation(NavigationHandle* navigation_handle)
-

Note that the states prefixed by WILL* are of interest to the NavigationThrottles only, therefore they should not be signaled through the WebContentsObserver interface.

NavigationThrottle

The NavigationThrottle allows implementers to pause the navigation at various points in time: network request start, redirects, and response received. Its interface should then be:

- void WillSendRequest(bool* defer)
- void WillRedirectRequest(bool* defer)
- void WillProcessResponse(bool* defer)
- NavigationHandle* navigation_handle()
- void OnNavigationHandleDestroyed()

Current navigation implementation

In the current implementation, the implementation of the NavigationHandle should be created when receiving a DidStartProvisionalLoad for a new navigation in a RenderFrameHost and should be owned by this RenderFrameHost until commit. There are a few edge cases:

- navigation failure: we will receive a DidStartProvisionalLoad marking the start of the error page load. This should not be considered as the start of a new navigation.
- transferring navigations: the NavigationHandle object should be passed from the first RenderFrameHost to the second one and the DidStartProvisionalLoad in the second renderer should not count as a new navigation.

DidFailProvisionalLoad and DidCommitProvisionalLoad will trigger state changes to DID_FAIL_REQUEST/DID_COMMIT respectively.

For the network events, we can use the NotifyRedirectOnUI and NotifyResponseUI from ResourceDispatcherHost (along with a NotifyRequestOnUI that would have to be created).

The throttle part should be handled by implementing a NavigationResourceThrottle on the IO thread that would bounce back with the NavigationHandle owned by the RenderFrameHost on the UI thread. The end goal there is to have throttles for navigations be handled by the NavigationHandle, and have throttles for the subresources continue to be handled by the ResourceDispatcherHost.

PlzNavigate

The NavigationHandle should be created along with the NavigationRequest and then owned by the NavigationRequest. When a response was received/the navigation failed, the NavigationHandle should be passed to the RenderFrameHost that will commit the navigation/error page. The commit is still signaled by receiving the DidCommitProvisionalLoad from the renderer. However the DidStartProvisionalLoad/DidFailProvisionalLoad IPCs will not be taken into account for changing the state of the NavigationHandle.

For the interaction with the IO thread, we already have an interface, the NavigationURLLoader that can be used instead of a NavigationResourceThrottle.

Example code with the new interface

Below is a rewrite of the CaptivePortalTabHelper with the new interface. The current version explicitly keeps track of the pending RenderViewHost and the last seen net error code. With the new interface, this is not needed. Instead, the CaptivePortalTabHelper can implement the following WebContentsObserver methods:

```
void CaptivePortalTabHelper::DidStartNavigation(
    content::NavigationHandle* navigation_handle) {
  if (navigation_handle->IsInMainFrame()) {
    tab_reloader_->OnLoadStart(
        navigation_handle->GetURL().SchemeIsCryptographic());
  }
}

void CaptivePortalTabHelper::DidRedirectNavigation(
    content::NavigationHandle* navigation_handle) {
  if (navigation_handle->IsInMainFrame()) {
    tab_reloader_->OnRedirect(
        navigation_handle->GetURL().SchemeIsCryptographic());
```

```
  }
}

void CaptivePortalTabHelper::DidCommitNavigation(
    content::NavigationHandle* navigation_handle) {
  if (navigation_handle->IsInMainFrame()) {
    tab_reloader_->OnLoadCommitted(navigation_handle->GetNetErrorCode());
  }
}

void CaptivePortalTabHelper::DidFinishNavigation(
    content::NavigationHandle* navigation_handle) {
  if (navigation_handle->IsInMainFrame() &&
      navigation_handle->State() != NavigationHandle::DID_COMMIT) {
    tab_reloader_->OnAbort();
  }
  login_detector_->OnStoppedLoading();
}
```

An example of what a NavigationThrottle paired with a WebContentsObserver could look like:

```
public class ThrottleCreatingObserver : public WebContentsObserver {
 public:
  void OnNavigationStarted(NavigationHandle* navigation_handle) override {
    throttles_.push_back(linked_ptr<InterceptThrottle>(
        new InterceptThrottle(navigation_handle, this));
  }
  void RemoveThrottle(InterceptThrottle* throttle) {
    for (ThrottlesList::Iterator iter = throttles_.begin();
        iter != throttles_.end(); ++iter) {
      if (*iter == throttle) {
        throttles_.erase(iter);
        return;
      }
    }
  }
 private:
  typedef std::list<linked_ptr<InterceptThrottle> > ThrottlesList;
  ThrottlesList throttles_;
}

public class InterceptThrottle : public NavigationThrottle {
 public:
  InterceptThrottle(NavigationHandle* navigation_handle,
                    ThrottleCreatingObserver* observer)
    : NavigationThrottle(navigation_handle),
      throttle_creating_observer(observer) {
```

```
  }

  void WillSendRequest(bool *defer) override {
    *defer = true;
    PerformAsynchronousChecks();
  }

  void WillRedirectRequest(bool *defer) override {
    *defer = true;
    PerformAsynchronousChecks();
  }

  void OnNavigationHandleDestroyed() override {
    throttle_creating_observer_->RemoveThrottle(this);
  }

  void OnChecksPerformed(bool resume) {
    if (resume) {
      navigation_handle()->Resume();
    } else {
      navigation_handle()->Abort();
    }
  }

 private:
  void PerformAsynchronousChecks() {
    // Do interesting stuff here with the info you can get from the
    // NavigationHandle* (using navigation_handle()).
  }

  ThrottleCreatingObserver throttle_creating_observer_;
}
```