RenderFrameHost vs FrameTreeNode swap

altimin@chromium.org
21 Aug, 2020

Background

With multiple page architecture, we are considering how to best support a number of cases ranging from existing and in-progress to potential, which include pending deletion frames, back-forward cache, portals, prerendering, fenced frames and guest views. This document focuses on the cases which need to make existing pages active: back-forward cache, portals and prerendering.

The most important goal of the design is to get //content/public API surface as simple as possible. Dozens (or even hundreds) of features are implemented on top of //content API. The degree familiarity of developers of said features with //content API varies, as does the test coverage, so a simpler //content API surface is critical for both overall Chromium stability and engineering velocity.

The second most important goal is to minimise complexity in the navigation code. Apart from it being already complex and having to deal with multiple edge cases, finer implementation details and abstractions of the navigation stack invariably leak via //content/public API, which closely connects this goal with the previously stated one.

Alternatives

There are two high-level alternative ways to implement an existing page (re)activation: RenderFrameHost swap (where an existing RenderFrameHost is transferred into the target FrameTreeNode) or FrameTreeNode swap (where an existing FrameTreeNode is transferred into the target WebContents).

Back-forward cache

Back-forward cache uses RenderFrameHost swapping — for storing the page in back-forward cache the normal navigation path is used until the navigation commits, when the old page together with all views and proxies is moved into back-forward cache instead of being deleted. For restoring the page from back-forward cache, a bfcached page is injected into NavigationRequest during creation and overall the normal path is followed (with the exception of synchronously emulating DidCommit for bfcache restore instead of waiting for a reply from the renderer).

Overall that works well and the bfcache-specific complexity is overall small:

- beforeunload event is dispatched as normal both for the navigation storing the page in the back-forward cache and the navigation restoring the page from back-forward cache
- Code updating per-tab state (e.g. theme colour) is shared for “restore page from bfcache” and “navigate away to a new page”.
- NavigationThrottles are dispatched for back-forward cache restore (see below for whether its desirable)

The main unfortunate consequences are:

- Back-forward cache relies on browsing instance swap (among other things, to be able to collect render view hosts and proxies). It doesn’t seem to be possible to support pages with openers without significant additional complexity.
- NavigationThrottle API is tied to the network request processing stages (WillStartRequest / WillRedirectRequest / etc). Back-forward cache restore doesn’t make a real network request, but pretends to make a fake one to dispatch navigation throttles. (also see “additional considerations” section).
- bfcache is main-frame only, which makes main frame navigations more different from subframe navigations.

Prerendering

For prerendering, we want to be able to load a new page in advance so when a user clicks a link, we can navigate to it instantly.

The main question here is: should this page be loaded in the same FrameTreeNode or a new one?

In order to load a page, we need to commit one or more navigations (for main frame and possibly for subframes). At the moment there is a strong assumption [1, 2, 3] that the inactive (and non-current) frames can’t navigate and current_frame_host() is often used by the navigation code (examples: 1, 2, 3, 4).

There are already problems (1073269 and many more) with inappropriate frame_tree_node()->current_frame_host() accesses from inactive frames, but current_frame_host() calls from navigation flow are safe due to aforementioned checks and the fact that the navigation will affect the current_frame_host() after it commits. However, if we can have two RenderFrameHosts in the same FrameTreeNode both of which can navigate, this assumption will no longer be true.

altimin@ doesn’t believe that there is a feasible path forward, as trying to support two-navigating-RFHs-in-FTN will both require a significant amount of work (dealing with all FTN::current_frame_host calls from navigation flow, dealing with navigation request ownership) and leave us in a fragile state (some places will need FTN::current_frame_host, so it has to stay, while most of the places will need to avoid and there is no easy way to distinguish between these two cases. Given that the navigation code, origin calculation and process model decisions is security-critical, adding a potential for a lot of subtle bugs doesn't sound like a good idea).

The next question is that if we create a separate FTN to load the prerendered content, how should the activation look like? There are two alternatives again: a) move the new RFH from the prerender FTN into the main FTN and b) swap the old main FTN with the prerendered FTN. Both of the approaches are feasible, with a) being closer the existing navigation code and b) requiring a smaller initial implementation cost.

A few examples we should consider here are:

- How beforeunload is dispatched (and maybe NavigationThrottles)
- How per-tab states including theme colour are updated
- How the previous page is handled (unload / store in bfcache).

With approach a), these things would work out of the box reusing the existing RFH swap / navigation code which handles this for new page load cases (and now — bfcache restores as well). With approach b), this will have to be reimplemented to support FTN swap specifically, doubling the complexity of this area.

The downside of the approach a) is that it will require ensuring that RFHs are not tied to a specific FTN, while b) will start (mostly) working out of the box. However, after that approach a) will work for the long tail of issues reusing common (or bfcache-specific) logic, while approach b) will require more work to support long-tail of issues and results in a higher maintenance burden, requiring many features to be have code handling both RFH swap and FTN swap.

Most of the fixes are going to be relatively simple. The two big exceptions are dealing with proxies and openers for RFH swap and unloading the old page for FTN swap.

Proxies/openers will work out of the box with FTN swap, but will require additional handling with RFH swap. The main mitigation for prerendering is that we want to disable prerendering when the page has an opener, which will allow us to reuse bfcache RVH / proxy collection logic. The longer-term plans to simplify this would be changing RVH lifetime to align it with the page lifetime.

Unloading an old page works with RFH swap out of the box, but FTN swap case presents a considerable challenge, especially in the case when the old page should be stored in a back-forward cache. Given that we want to keep RFH : FTN association constant, we will have to keep FTN in bfcache for the eventual restoration, which will require having two models (see "additional considerations" on why swapping root FTN on new page loads isn't a stable end state).

Overall altimin@ believes (especially given that disassociating RFH from FTN seems to be a generally good thing), that approach a) will result in a significantly simpler code overall and less work in the medium and long term.

Portals

The general same principles apply to portals, with a few caveats:

- As portals activation is initiated by the page, it's not quite clear whether we want to dispatch `beforeunload` or not (altimin@ believes that we do, at least for cross-origin subframes of the previous page).
- Portals get a new BrowsingInstance, so window.opener will always be null for newly created portals. However, the BrowsingInstance may have more than one window if an existing page gets adopted into a portal.
- Portals might have used window.open() API before readoption, so we will need to ensure that window.opener points to RFH rather than FTN.

Additional considerations

Why removing FrameTreeNode reference from RenderFrameHost is good

Code reaching out to FrameTreeNode from inactive frames (especially IPC handling code) is a source of long tail of hard-to-detect bugs. Removing the ability to reach out from RFH to FTN directly makes us more robust w.r.t. to race conditions around navigation, including the pending deletion document case.

In particular, the following pattern is proposed to be encouraged, which relies on FTN directly knowing which RFH is the current one (instead of RFH having to check its state) and exposing a small stateless interface to RFH.

```
class DefinitelyNotFrameTreeNode {
  void OnXUpdated();
}

class FrameTreeNode : public DefinitelyNotFrameTreeNode {}

RenderFrameHostImpl::SetX(X x) {
  x_ = x;
  delegate_->OnXUpdated();
}
```

```
FrameTreeNode::OnXUpdated() {
  if (last_x_ == current_frame_host()->GetX())
    return;
  last_x_ = current_frame_host()->GetX();
}
```

This approach scales well both to “FTN navigated to a new RFH” and “RFH has updated its value”.

Swapping FrameTreeNodes on every navigation

Q: Why couldn’t we swap FrameTreeNodes on every navigation, including new page loads and bfcache restores?
A: In this case, we will swap FrameTreeNodes on every cross-document main frame navigation, but not on subframe navigations. Also for main frames we will have to have some class (FrameTreeNodeManager) to coordinate FTN swaps, so the overall picture will look like:

- Main frame: FrameTreeNodeManager (stays the same), FTN (changes), RFH (changes)
- Subframe: FTN (stays the same), RFH (changes)

In this situation, moving all of the state that needs to be swapped to RFH and have FTN always stay the same will result in a simplification.

Do we need to dispatch NavigationThrottles for activations / bfcache restore

NavigationThrottle API is strongly tied to network request stages, but the intention for at least some navigation throttles is to control the timing when the active document changes.

Examples:

- JavaScriptDialogNavigationThrottle, which wants to defer navigation while the JS dialog is pending. This should include bfcache restores and portal / prerendering activation.
- Ablation studies, adding an artificial delay to navigations.

Overall it seems that we should add a way to control the timing of non-document-load navigations via NavigationThrottle or equivalent API, but avoid tying it to network requests (even a fake one) to avoid confusion.

---

creis@'s in-progress braindump on tradeoffs:

1) Move RFH to a different FTN

- Conceptually, main frame FTN is a "page holder," and main frame RFH represents a "page." Page changes on every document swap, staying in the same FTN.
- Subframe FTN represents a position in the page's frame tree. It can't change its page or position in the frame tree.
- Move BI state for a frame (name, opener, FrameReplicationState, proxy topology, maybe proxies themselves and RVHs) away from FTN/RFHM, closer to RFH. Might be on an object specific to the BrowsingInstance, to make it clear when it's shared vs reset.
- RFH can change FTNs while taking the relevant state with it (e.g., for portals, prerendering) or swap within a given FTN (e.g., normal navigations, bfcache, pending delete, GuestView, fenced frame). Only main frame RFHs can change FTNs.
- Pages can change over time whether in the same BI (cross-document nav) or not (BI swap in same FTN, portal activation across FTNs).

Pros:
- Consistent way to swap RFHs whether the RFH's FTN changes or not.
- Allows thinking about "page swaps" as happening on every main frame document change.
- Could cause us to move BI state closer to RFH and not FTN/RFHM. (This also applies for the "Swap FTNs" approach.)
- RFHs are never bound to FTNs even for subframes (e.g., avoiding bugs for pending delete).
- Keeps the FTN ID constant (which is used in extension APIs).

Cons:
- Breaks the concept of FTN representing a position in a frame tree of a *page*. Treats main frame FTN as a placeholder for a *tab* instead.
- It's never ok to move a RFH into a different position in any page's frame tree, and never ok to move subframe RFHs between FTNs period. Would have to enforce that as a separate invariant, rather than having it naturally follow from swapping at the page level.
- Requires changing RFH (and related code) to not have a fixed FTN. (That could be beneficial for pending delete cases, though.)
- Might not be as good a fit for portal activation, which doesn't really seem as much like a navigation.

Unclear:
- What replaces the RFH in its old FTN when it gets moved? Don't want a swappedout:// situation, and FTN doesn't really support a no-RFH state.
- NavigationController and FNE depend on FTN, at least for subframes. Maybe it's flexible enough for the main frame FTN to change without breaking that, though?

2) Swap FTNs
- Conceptually, FTN represents a position in a page's frame tree. Changes each time the page changes. Page is external to FTN.
- BI state can stay on FTN and RFHM (i.e., name, opener, FrameReplicationState, proxies, RVHs). Maybe it's better to move it to a new class to allow swapping FTNs on same-BrowsingInstance navigations, though.

- Portals/prerendering cause a WebContents to swap its FTN. Normal cross-document navigations do not.
- NavigationRequest might need the ability to start in one FTN and finish in another.

Pros:

- FTN has a single conceptual meaning: a position in a given page's frame tree.
- No need to move BI state if FTN swaps only happen for bfcache, portals, prerendering (which all involve BI swaps).
- Could cause us to move BI state closer to RFH and not FTN/RFHM, if FTN swaps happen for same-BI navigations. (This also applies for the "Swap RFHs" approach.)

Cons:

- Swapping a page requires action outside of RFHM, which lives within a FTN. This could be awkward, requiring another level of swapping and/or coordination with the RFHM.
- Requires changing NavigationRequest to not have a fixed FTN. (Might not be a problem?)
- Might assume a "page swap" / FTN swap is about special cases (portals, prerender) rather than getting a new page on every cross-document navigation. That feels less general.

Unclear:

- How awkward is swapping an FTN during an RFHM operation (in terms of layering, etc)?
- Would you do a page swap on every main frame cross-document navigation, akin to a new RVH every time? Including ones in the same BrowsingInstance?