Design Doc: Session History for Out-of-Process iframes

Charlie Reis, May 2014

This document outlines the changes we plan to make to Chrome's session history logic to support iframes that are rendered in a different process than their parent frame. This is part of the Out of Process iframes project within the Site Isolation effort.

---

1. Background

It is useful to understand the concepts of a frame's session history and the overall joint session history from the HTML5 spec. Each frame creates session history items as it navigates. The back/forward history that the user sees is made up of joint session history items, each of which represents the current session history item for each frame in the page. In a sense, it's a flattened view of the commits that happen in each frame.

Two interesting cases for the joint session history are subframe navigations and pruning forward history items. First, when the first navigation occurs in a subframe, it becomes part of the existing joint session history item (which we call an "auto subframe navigation"). In other words, the user can't go back to the state before the frame. However, any subsequent navigations in the subframe create new joint session history items (which we call "manual subframe navigations"), such that clicking back means to go back within the subframe.

Second, if the user goes back and then does a new navigation, this effectively forks the session history. The session history is kept as a list and not a tree, though, so the other branch is "pruned" and immediately forgotten.

---

2. Current Architecture

Today, Chrome's session history is managed in two places: NavigationController tracks the joint session history in the browser process, and HistoryController tracks the various session histories for each frame in the renderer process.

In the renderer process, HistoryController operates on joint session history items (HistoryEntries) which have HistoryItems for each frame. However, we do not keep all the HistoryEntries in the renderer process. Instead, we serialize them and send them to the browser as PageState objects stored on each NavigationEntry in NavigationController. In Chrome's current architecture, we can treat a NavigationEntry's PageState as opaque in the browser process because the HistoryItems for all of its frames will be loaded in the same renderer process. This no longer works with out-of-process iframes, where some frames may live in different processes.

Typically, the renderer process will commit a new navigation and send details to the browser process, where NavigationControllerImpl::RendererDidNavigate records it in a new NavigationEntry. The renderer will later send the serialized WebHistoryItem as a PageState object via UpdateState, where it is stored on the corresponding NavigationEntry.

For back/forward navigations or other restored history items, the browser process sends the PageState object to the renderer, where it is deserialized to create a HistoryEntry (containing one HistoryNode and WebHistoryItem per frame in the tree). Within Blink, FrameLoader starts to load the HistoryItem for the main frame, Whenever it creates a child frame, it calls out to RenderFrameImpl::historyItemForNewChildFrame to see if there is a corresponding WebHistoryItem to load in it (based on the frame's ID and name). This lets us restore multiple frames in a page as needed, one HistoryItem at a time.

Identifying History Items

Each HistoryItem in the renderer has a item sequence number (ISN) indicating its order in the frame's session history, as well as a document sequence number (DSN) indicating which history items represent same-document vs different-document navigations.

The browser process doesn't currently care about these sequence numbers, instead using a renderer-specific page ID to identify NavigationEntries. Page IDs are allocated in the renderer process on each new commit, and they can only be compared to other page IDs from the same SiteInstance and WebContents (i.e., the same RenderView). The page ID system will need to

change to support out-of-process iframes, since no one renderer process observes all NavigationEntries (e.g., a navigation in a subframe may not be visible to the top-level frame's process). We expect to allocate page IDs in the browser process instead.

One particularly complex use of page IDs is to detect a race between when the browser process requests a history navigation and when the renderer process commits a different new navigation. Consider a case where the user has gone back to page A and has an entry for page B in the forward history. The browser process could ask the renderer to go forward to page B, but while the IPC is in flight the renderer commits a new navigation to page C. When C commits, it prunes the entire forward history, making the browser process forget about page B. If the renderer now navigates to B with the forgotten page ID, the browser process won't know what to do with it when it commits. This corrupts the session history and leaves the wrong URL (C) in the address bar above the current visible page (B). Currently, RenderViewImpl tries to track which page IDs have been pruned so that it knows to ignore the request to navigate to page B after committing page C. We may have to revisit this with an alternate solution when moving page IDs to the browser process.

Cloning State in NavigationEntries

One design flaw in NavigationEntry is that it clones all the state from the previous NavigationEntry when a subframe navigates. Consider a NavigationEntry with a main frame page A and a subframe page B. If the subframe navigates to page C, we create a new NavigationEntry with a serialized HistoryItem for A and one for C. Now suppose a replaceState operation occurs in the main frame, which modifies the URL in the NavigationEntry for A. Because the two NavigationEntries clone the state for A, the replaceState's effect on the URL will be forgotten if we go back to the previous NavigationEntry, contrary to the spec. We plan to fix this while moving the session history logic to the browser process, by sharing FrameNavigationEntries across NavigationEntries rather than cloning their state. (See http://crbug.com/373041.)

---

3. Proposed Architecture

Conceptually, most of the work done by HistoryController to track the session histories will move to NavigationController in the browser process. The renderer process will only be responsible for committing and restoring history items for individual frames.

NavigationController will continue to have a list of **NavigationEntries** to represent the joint session history. Each NavigationEntry will now contain a tree of pointers to **FrameNavigationEntries**, representing session history items from each frame, possibly from multiple renderer processes. (This replaces HistoryEntry and its tree of HistoryNodes from the renderer process.) NavigationController will also keep per-frame lists of FrameNavigationEntries, indexed by FrameTreeNode ID. This allows us to share a

FrameNavigationEntry among multiple NavigationEntries, fixing the replaceState bug due to Blink's cloning behavior.

New navigations will be reported per frame, causing us to create a FrameNavigationEntry and associate it with a new or existing NavigationEntry as appropriate (e.g., auto vs manual subframe navigations). The FrameNavigationEntry will keep track of any frame-specific details, such as URL, SiteInstance, referrer, and the serialized WebHistoryItem (in a **FrameState** object, rather than a PageState object). It will also use the renderer-allocated item sequence number (ISN) to represent its location in its session history list. The NavigationEntry, meanwhile, will use a browser-allocated page ID to represent its location in the joint session history list, and it will keep track of page-level details (e.g., title, overscroll screenshot).

Navigations to existing history items will start by sending affected frames' FrameState objects to the corresponding renderer process (based on SiteInstance). (This will usually start with the main frame, but it could be the roots of multiple subtrees instead, as in the example slides linked below.) If additional subframes are created while loading these FrameStates, RenderFrameImpl::historyItemForNewChildFrame will asynchronously ask the browser how to handle it. The browser process will either (1) send back a FrameState object for the new subframe, (2) tell the renderer process to create a RemoteFrame + RenderFrameProxy instead, since the frame will be rendered in a different process, or (3) tell the renderer to load the URL from scratch, since no record of the given frame was found in a FrameNavigationEntry (e.g., the page may have changed since the last visit).

Example:
This set of slides walks through how FrameNavigationEntries should be created, updated, and pruned for an example set of navigations.

---

4. Implementation-Level Changes

This section outlines some of the expected changes we will need to make, organized by class and by task. Section 6 tries to break these down into a rough plan for implementation.

Class-level changes

- **NavigationController**:
  Continues to have a list of NavigationEntries. Now also has a map of FrameTreeNode ID to a list of FrameNavigationEntries, representing each frame's session history. Each frame's session history list is kept alive until all of its FrameNavigationEntries are gone. NavigationController will also gain much of the current logic from HistoryController for determining which frames need to navigate when going back/forward to an existing history item.

- **NavigationEntry**:
  Now has a tree of pointers to shared FrameNavigationEntry objects. It is important for there to be a separate TreeNode class from FrameNavigationEntry, since FrameNavigationEntries can be shared across NavigationEntries and may have different children at different times. Any frame-specific state (e.g., URL, SiteInstance, PageState, etc) from NavigationEntry should move to FrameNavigationEntry. Identified by page ID, which is now browser-allocated and WebContents-specific.
  We can still have last committed, pending, and transient NavigationEntries.
- **FrameNavigationEntry**:
  New class that keeps track of frame-specific state (e.g., URL, SiteInstance, FrameState). Identified by a renderer-specific item sequence number (ISN).
- **FrameState**:
  PageState will be replaced with FrameState, which is a serialized WebHistoryItem for a single frame.
- **HistoryController**:
  Will only need to operate on a single frame. Much of the remaining logic for recursively determining which frames to navigate can be moved or adapted for NavigationController in the browser process.
- **HistoryEntry**:
  Goes away, replaced by NavigationEntry.
- **HistoryNode**:
  Goes away, replaced by NavigationEntry::TreeNode.
- **HistoryItem**:
  Stays around and gets serialized into FrameState objects.

Tasks

- **Navigating to a new main frame:**
  Create a FNE for the frame with the ISN and add it to the frame's session history. Create a NE with pointers to all the current FNEs for the current FrameTreeNode IDs. The NE gets a new page ID. The renderer process simply reports a page ID of -1 since it doesn't know what the new page ID will be.
- **First new navigation in a subframe:**
  Create a FNE for the frame with the ISN and add it to the frame's session history. Add the FNE to the current NE's tree in the correct location, without touching any other FNEs in the NE's tree.
  (auto-subframe)
- **Subsequent new navigations in subframes:**
  Same as navigating to a new main frame, creating a new NE. We can perhaps distinguish between manual subframe and auto-subframe navigations based on the PageTransition, since page ID won't help here anymore (always -1 for new navigations).
  (manual-subframe)

- **New in-page navigation:**
  Generally same steps as above.
- **Navigate to an existing NE:**
  Using logic that used to be in HistoryController, walk the NE's tree looking for frames that need to be navigated. Send the FrameState to each one. If an affected frame has subframes, RenderFrameImpl::historyItemForNewChildFrame will request the FrameState for them.
  For existing entries, the browser process sends the NE's page ID to the renderer, where it is stored in NavigationState::pending_page_id() and simply echoed back to the browser on commit. The renderer doesn't need to otherwise store or use this value, but the browser needs it to distinguish between multiple NavigationEntries that might share some FrameNavigationEntries. (Note: we may be able to skip telling the renderer process about the page ID if we can rely on the pending NavigationEntry, but there's a risk that the pending entry will have changed in the meantime.)
- **Delete subframe:**
  All browsers seem to behave differently here. If you go back/forward after a frame has been deleted, it appears that the frame can either be recreated, restored if present (best effort), or kept deleted.
- **location.replace:**
  These navigations are intended to replace the current history item without pruning the forward history. This behavior becomes somewhat undefined when the back or forward items involve subframes that have been deleted by the location.replace operation. We may want to drop NavigationEntries that become meaningless after such an event.

We currently have a draft CL exploring some of these changes:
https://codereview.chromium.org/281653003
At the moment, it covers most new navigations but not navigations to existing entries.

---

5. Open Questions

There are still a set of questions that remain to be answered for the project.

- Short term:
  - How can we support existing IsActiveEntry callers, since this API is currently broken and should be removed?
  - Can we get by with no stale page ID protection by treating stale forward navigations as new navigations? Will re-ordered NavEntries be a problem (e.g., fast back/forward)?
- Medium term:
  - How should we update session restore for the new architecture?
  - Can we still get by with no NavigationEntry for an initial empty document, even when subframes commit? Seems to work for now, but this might cause

headaches (e.g., having no origin associated with the document in the browser process).
    - What should the pending entry be? NE vs FNE? Only for top-level? One per frame? Multiple allowed at once?

---

6. Implementation Plan

We can divide the work into multiple stages, some of which make the later work possible.

Stage 1: Page IDs

First, we need to move page ID allocation to the browser process, clearing out most uses of it in the renderer process.

- ~~**Remove broken uses of IsActiveEntry**~~ (371150)
  This task is necessary regardless of the session history project. IsActiveEntry tries to compare page IDs without comparing SiteInstances, which is broken because page IDs from different processes will conflict. (It's theoretically possible to keep the API if we make page IDs WebContents-specific, but we still couldn't support many of the callers who currently pass in page IDs from the renderer process.)
    - ~~SearchIPCRouter is one of the main users. I've started a discussion with kmadhusu@ about what they'll need to replace it. It sounds like a commit counter and process ID checks should be sufficient.~~ (fixed with http://crrev.com/292123015)
    - ~~The only other usage is SearchEngineTabHelper::OnPageHasOSDD (open search descriptor document). In this case, the OSDD is likely specific to a URL and not necessarily a particular visit to the page, so we could likely compare doc_url to WebContents::GetLastCommittedURL() instead.~~ (fixed with http://crrev.com/295103003)
- ~~**Remove RenderView::GetPageId**~~ (371151)
  If we plan to move page ID knowledge to the browser process, we can't expose GetPageId. It appears to mainly be used in places that pass it to IsActiveEntry in the browser process, suggesting we may be able to find alternate ways to meet the callers' needs. In many cases, watching for commit or keeping a commit counter is probably a more accurate thing to do, especially if the check cares about reloads.
    - ChromeRenderViewObserver::CapturePageInfo is doing page indexing, but it's not clear that page ID is the right distinguishing info (e.g., should the same URL be indexed twice at different page IDs, and should we skip for reloads?).
    - Translate and user scripts are some of the other callers; a full list is in the bug.
- **Verify that we know the right page ID in the browser process for any message**
  I have a CL that tries to check that the browser process knows what page ID the renderer is on during any IPC that passes it (e.g., UpdateState), and thus the renderer

does not need to report it. There are some corner cases to resolve, but this seems like it will work. Once verified, we can remove page ID from these IPC messages.
    - CL: https://codereview.chromium.org/423393008
- **Move page ID allocation to browser process** (369661)
  We can start by moving page ID allocation from RenderViewImpl to RenderViewHostImpl to minimize the impact and risk. We can later evaluate whether it should move to NavigationControllerImpl and be WebContents-specific rather than RenderView-specific. The only remaining page ID logic in RenderViewImpl should be temporarily storing NavigationState::pending_page_id() during a history navigation.
    - **Remove stale page ID detection and deal with it in browser process**
      This is probably the riskiest part. There doesn't appear to be an easy way for the renderer to detect attempts to navigate to page IDs that have been pruned from the forward history anymore. This means the browser process may see commits of "existing" history items that are no longer in the session history. Fortunately, with page ID allocation in the browser process, it may be possible to treat these as new navigations instead and simply append them to the end of the history. That risks reordering history but might offer a sane way to handle this case.

Stage 2: FrameNavigationEntries

With page IDs in the browser process, we should be able to add FrameNavigationEntries. It may be worthwhile to hide the effects of this change behind the --site-per-process flag so that we do not affect users' current session restore data until the refactor is far enough along.

- Add FNE and only create it when using --site-per-process
    - First round should work for new navigations
    - Be able to convert to/from old session restore format?
- Move HistoryController's per-frame navigation logic to NavigationController

Part of this is explored in the following draft CL:
https://codereview.chromium.org/281653003

Stage 3: Update Session Restore

This stage still needs to be planned.
(Check with sky@ and darin@ for expertise here.)

---

7. Appendix

General problem areas

There are a few areas that have proved troublesome in Chrome's multi-process navigation logic over time. These are worth mentioning here to consider whether the planned changes pose risks for triggering related problems.

- Corrupted entries or hangs with fast back/forward (89798, 93427, 416184)
  These were often the result of page ID confusion between the browser and renderer processes, especially related to page IDs (and max page IDs) from different SiteInstances. The fact that Chrome limits the session history to a maximum of 50 entries and drops earlier entries contributed to these bugs.
- URL spoofs due to races (102408)
  These used to occur when a renderer process committed a page ID that the browser process didn't recognize, and which was smaller than the max page ID. These cases now turn into renderer kills (in NavigationControllerImpl::ClassifyNavigation) rather than risking a spoof, and some metadata is stored in the URL of the crashdump to help diagnose it. We still have a low volume of these crashes.
- URL spoofs around pending entries (9682)
  Pending entries were originally only used for browser-initiated navigations, but we now create them for renderer-initiated navigations as well. This allows us to show the pending URL on slow renderer-initiated navigations in new tabs, as long as no one has injected content into the new tab. It proved to be quite tricky to do this while preventing URL spoofs.
- Confusion about in-page vs not
  We've seen multiple bugs in both Blink and the cross-process navigation logic with determining when a navigation stays in the same document or not. Exposing the document sequence number to the browser process may simplify much of this logic.
- Many corner cases around subframe navigations
  Many browsers seem to have frequent corner cases around subframe navigation that can lead to unexpected behavior (e.g., how the session history is affected by deleted subframes or location.replace).