Onion Soup: Loaders

http://bit.ly/loader-onion-soup
Tracking Issue: crbug.com/860403

Authors: Makoto Shimazu <shimazu@chromium.org>, Minggang Wang <minggang.wang@intel.com>
Status: **public.** implementation has started by minggang.wang@intel.com. Still needs investigation.
Last update: July 20, 2020
Created at: Apr 18, 2019

TL;DR

Now we have network.mojom.URLLoader/URLLoaderFactory as interfaces for network requests. We can replace WebURLLoader/WebURLLoaderFactory with them. However, WebURLLoader has non-trivial differences from network.mojom.URLLoader, which are implemented in WebURLLoaderImpl and ResourceDispatcher. This doc is to list the differences and discuss how to implement them.

Benefits from the onion soup

- Unblocking other onion soup
  - Service workers
  - content/renderer/fetcher/resource_fetcher
  - A part of appcache code
  - … (needs investigation)
- Eliminating thin wrapper layer (and this will help to reduce code to convert types.)
  - WebURLLoader/WebURLLoaderClient
  - WebURLRequest/WebURLResponse
- Modernizing the current implementation
  - Unify duplicated interception code: RequestPeer and URLLoaderThrottle

Potential issues on moving things in Blink

content::RequestExtraData

This is a struct used mainly at WillSendRequest() **[not sure, need to check]**, which is adding some extra info used in content. I need to figure out where it's used.

ServiceWorkerNetworkProviderForFrame::WillSendRequest() sets SetFetchWindowId(), but it's not in the ExtraData. good.
Needs some more investigation.

URLLoaderThrottle

ThrottlingURLLoader tries to interrupt the request/response if URLLoaderThrottle exists. We need to move them into blink.

Status in June 2020: blink::Platform::CreateURLLoaderThrottle is already created. Still need to migrate implementations from content to blink.

Here is the list of throttles which could be created in the renderer process as of May 8, 2019:

- MimeSniffingThrottle (**both**)
- RequestBlockerThrottle (renderer only)
- Throttles created in URLLoaderThrottleProviderImpl::CreateThrottle()
    - data_reduction_proxy::DataReductionProxyURLLoaderThrottle (**both**)
    - prerender::PrerenderURLLoaderThrottle (**both**)
    - GoogleURLLoaderThrottle (**both**)
    - safe_browsing::RendererURLLoaderThrottle (renderer only)
    - extensions::MimeHandlerViewContainerBase::PluginResourceThrottle (renderer only)
    - extensions::ExtensionURLLoaderThrottle (renderer only)
    - MergeSessionLoaderThrottle (renderer only, CrOS only)

Other browser-process-only URLLoaderThrottles (these are not related to this project. Just for reference):

- AwURLLoaderThrottle (android webview only)
- ProtocolHandlerThrottle
- PluginResponseInterceptorURLLoaderThrottle
- signin::URLLoaderThrottle
- safe_browsing::BrowserURLLoaderThrottle

In addition to that, there was a discussion about how to eliminate the typemapping:
Discussion on CL comments
Non-Blink Mojo types in Blink for Loader's Onion Soup

Now we concluded that we can use non-Blink Mojo types in Blink so that we can expose the network requests and responses to the outside of Blink without extra type conversions.

Sync Load

WebURLLoader has two types of loads - sync and async.
LoadSynchronously is implemented by creating another background thread task runner with a waitable event. We need to move the logic into blink.

Note that blink::ResourceRequest cannot be passed across threads. We might need to use network::ResourceRequest as a cross-threadable intermediate object.

RequestPeer & ResourceDispatcher

We are currently have four request peers:

- ExtensionLocalizationPeer
- SyncLoadContext
- RequestPeerImpl
- SinkPeer

ExtensionLocalizationPeer replaces magic keywords in the response body with specific sentences in the dictionary. We can implement it as a URLLoaderThrottle.
SyncLoadContext needs to be implemented in some way for loading onion soup, but maybe we don't have to do it by using RequestPeer.
RequestPeerImpl doesn't do anything. We can just remove it.

SinkPeer is just consuming the body. We can implement it in some way even if there is no RequestPeer.

Defer

Requests can be deferred for, for example, showing an alert.
It's now implemented by URLLoaderClientImpl, but we need to implement this in some way.

ResourceLoadStats

Let's move the function calls from ResourceDispatcher to RenderFrameImpl and call it from Blink through WebLocalFrameClientImpl.

Current flow of network requests

Async resource loading

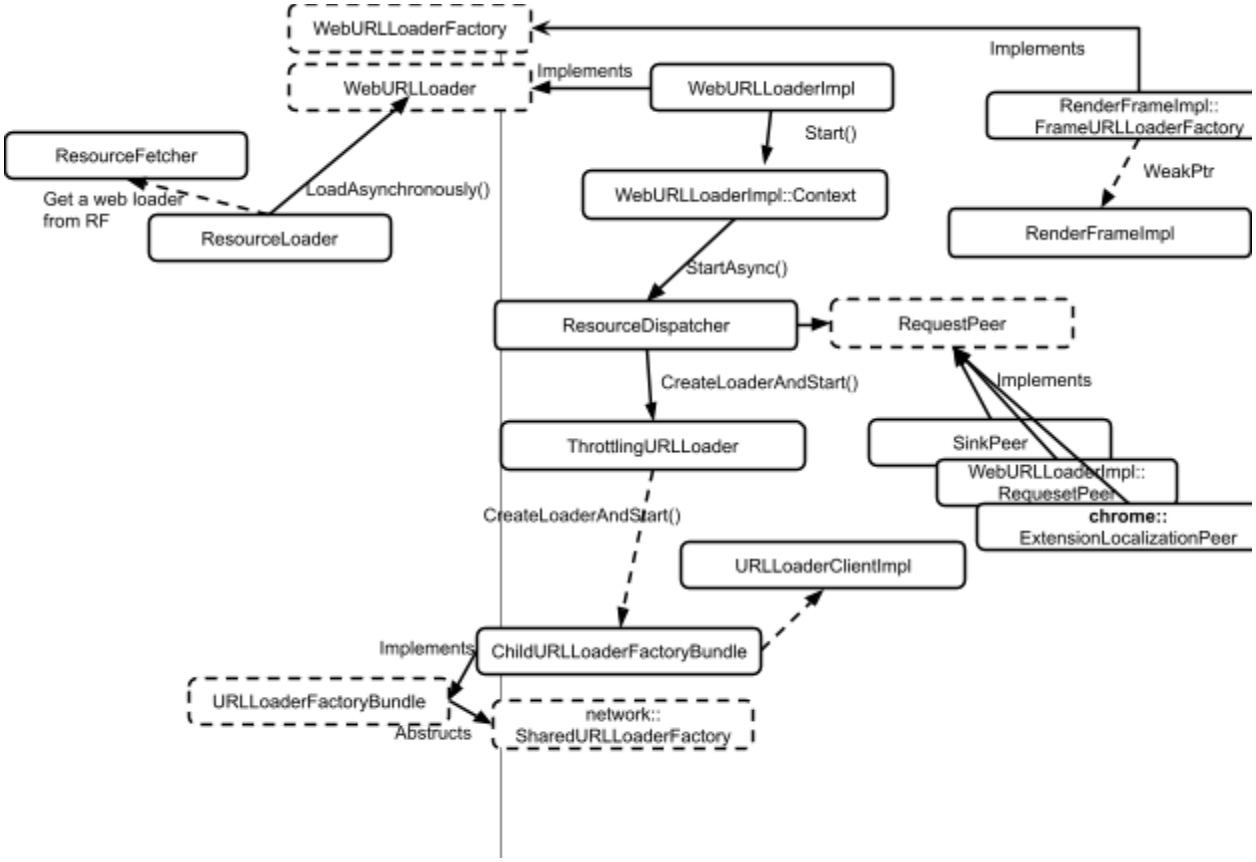
blink::ResourceFetcher -> blink::ResourceLoader
--(blink::WebURLLoader)--> content::WebURLLoaderImpl::LoadAsynchronosuly()
-> content::ResourceDispatcher::StartAsync()
-> ThrottlingURLLoader::CreateLoaderAndStart()
-> network::SharedURLLoaderFactory (owned by WebURLLoaderImpl)

Design overview

How is the current implementation?

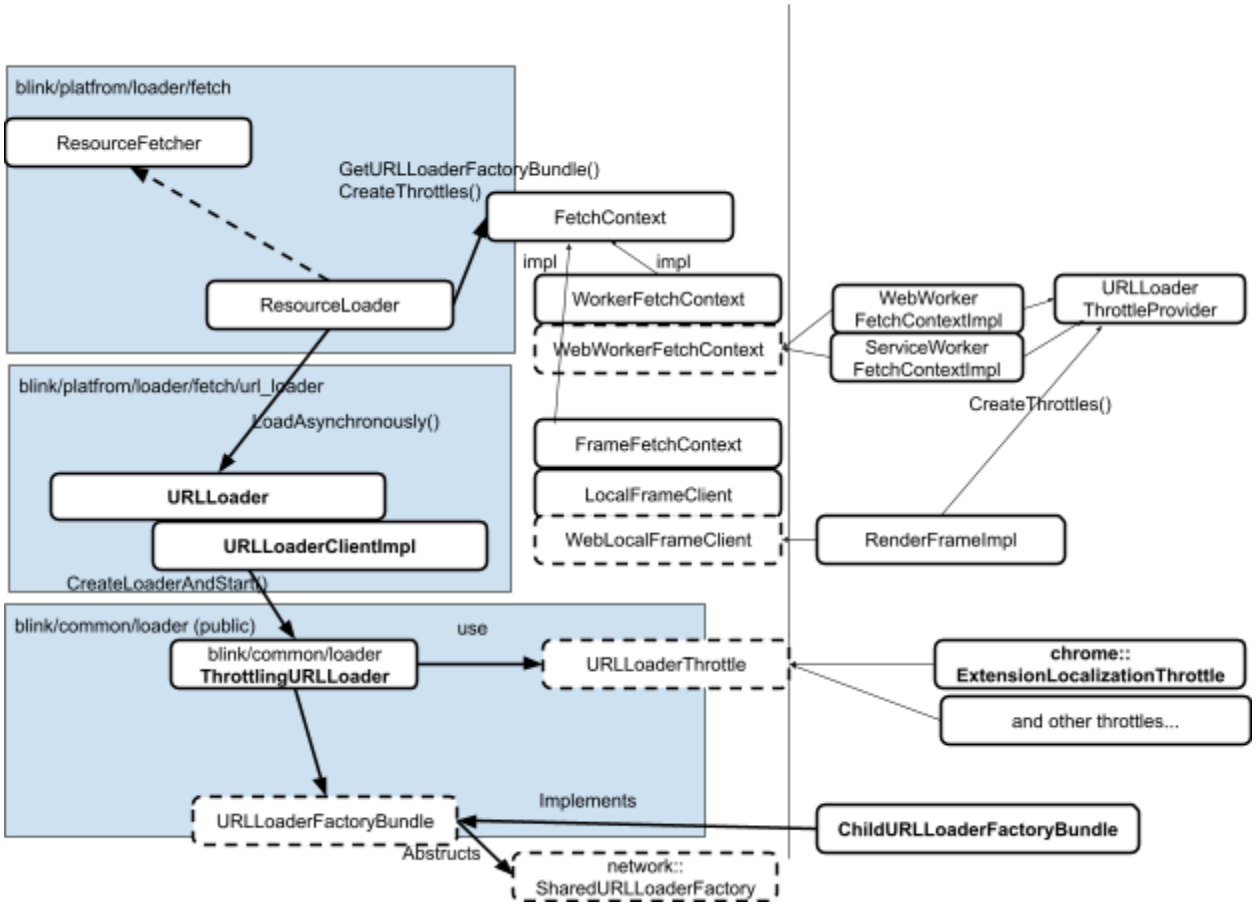
Basically, blink::ResourceLoader is an equivalent to WebURLLoader. We can move logics on WebURLLoaderImpl to blink::ResourceLoader.

Here is the rough call graph. I'll add some notes for each part to explain the details.



What will it look like? (step 1)

As a first step, we can move a part of code which uses URLLoaderFactory/URLLoader/URLLoaderClient into Blink.



Very rough items we need to work on are as follows:

- Move ThrottlingURLLoader in blink/common to share it with blink and content.
- Make FetchContext be able to return URLLoaderFactoryBundle and URLLoaderThrottles. Get it in ResourceLoader to create a new class **blink::URLLoader**.
- We need to move things appended by the WillSendRequest() as RequestExtraData to Blink.

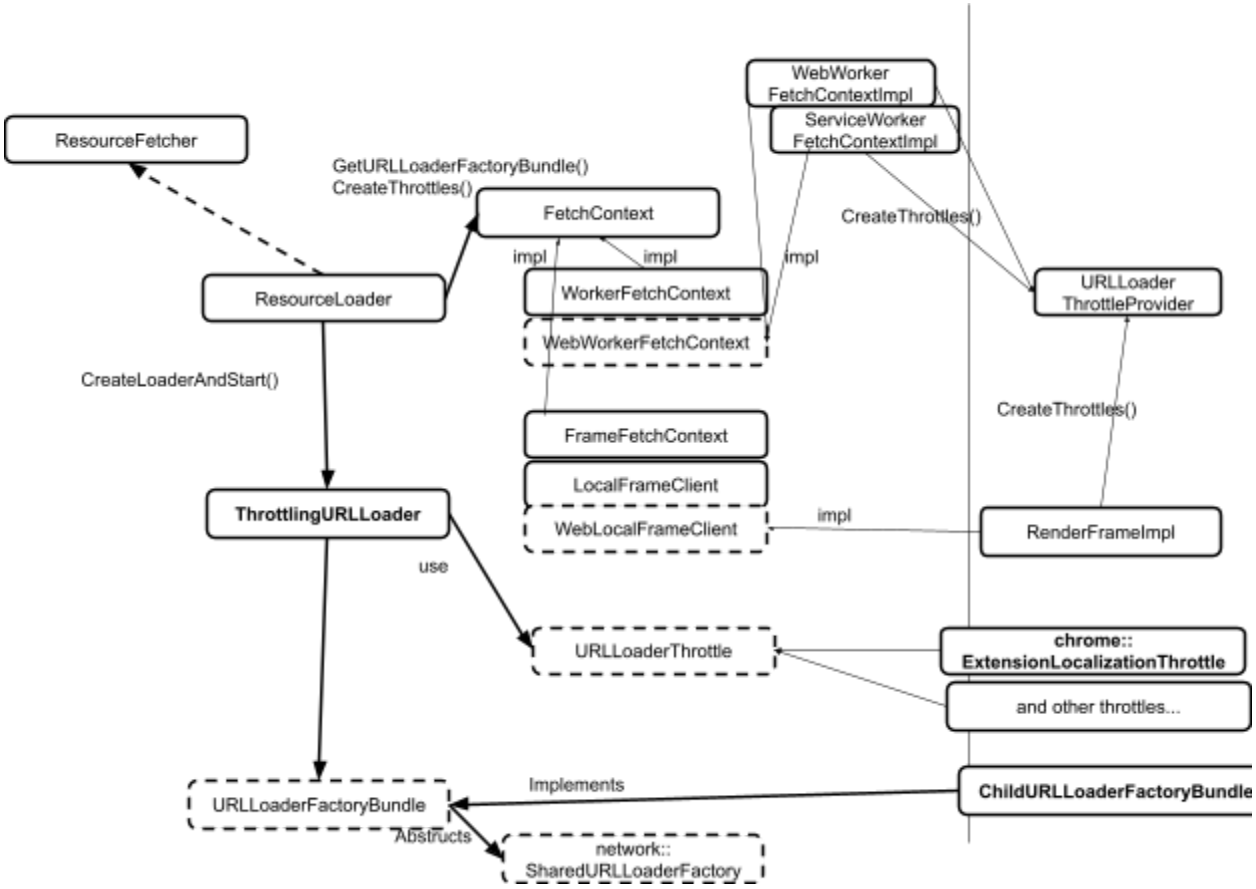
Consideration about the RequestExtraData

- blink::WebString custom_user_agent_;
  - pepper plugin add the UA.
  - RenderFrameImpl::WillSendRequest() sets User-Agent header to the custom_user_agent.
  - **Maybe at this point, it won't be a problem since it's not used in WebURLLoaderImpl.**
- std::unique_ptr<NavigationResponseOverrideParameters> navigation_response_override_;
  - only for dedicated worker and shared worker's main script.
  - **Let's think of another way to implement the worker script loading.**
- bool block_mixed_plugin_content_ = false;
  - no call sites of the setter? It doesn't seem to be used anymore.
  - https://codereview.chromium.org/2625633002 This CL seems to have added the flag.

    - **Let's remove the param.** (https://chromium-review.googlesource.com/c/chromium/src/+/1883349/ to remove the param.)
- std::vector<std::unique_ptr<blink::URLLoaderThrottle>> url_loader_throttles_;
    - **Will be moved.** FetchContext will return the throttles instead of this.
- scoped_refptr<FrameRequestBlocker> frame_request_blocker_;
    - RenderFrameImpl has a FrameRequestBlocker and it's used for the browser to control resource loading in the frame.
    - **TODO(shimazu): Investigate how to move it to Blink** (maybe adding a method like WebLocalFrameClient::GetFrameRequestBlocker()?)
- bool allow_cross_origin_auth_prompt_
    - **RendererPreference. Let's move RendererPreference and use it in Blink.**
- // Pramas in blink::WebURLRequest::ExtraData
- int render_frame_id_;
    - Used only in content/renderer to copy the info to network::ResourceRequest.
    - **Consider how to expose it in Blink.**
- bool is_main_frame_ = false;
    - Used only in content/renderer to copy the info to network::ResourceRequest.
    - **Consider how to expose it in Blink.**
- ui::PageTransition transition_type_ = ui::PAGE_TRANSITION_LINK;
    - Used only in content/renderer.
    - **Consider**
- bool is_for_no_state_prefetch_ = false;
    - set at WillSendRequest and used in WebURLRequset::GetLoadFlagsForWebURLRequest.
    - Once ContentRendererClient::IsPrefetchOnly() is available in Blink, perhaps we can eliminate it. However,
- bool originated_from_service_worker_ = false;
- bool attach_same_site_cookies_ = false;
-

What will it look like? (step 2)

After that, we can move fetch contexts.



There are three types of fetch contexts in content now.

- RenderFrameImpl (indirectly used by FrameFetchContext via WebLocalFrameClient)
- WebWorkerFetchContextImpl (for shared worker and dedicated worker)
- ServiceWorkerFetchContextImpl

Let's list how they interact with outside of Blink, and let's estimate the possibility to move the fetch contexts into Blink.

RenderFrameImpl (as FrameFetchContext)

We would be able to move some of the methods used in FrameFetchContext from content/. It would decouple FrameFetchContext and some code in content. Some of usages in WillSendRequest() in RenderFrameImpl needs more investigation.

(methods on blink::LocalFrameClient* called in blink::FrameFetchContext)

- DispatchWillSendRequest()
    - DNT header,
        - **can go to blink**
    - file path alias (???) to update URL
        - ???
    - getting requestor origin

        - ???
    - URL could be updated by the embedder
        - extensions
        - search schemes
        - **we might have different route to the embedder so we don't have to stick with RenderFrameImpl. It only needs to modify URL.**
    - setting cache mode override
        - ???
    - extra data for user agent
        - **we might be able to plumb UA by another way.**
    - extra data for NavigationResponseOverride
        - **we can eliminate response override if we need to do that.**
    - Creating settings an extra data
        - throttles
        - update Previews state
        - render_frame_id, …
        - Let's defer the listing of the members to the section above.
    - download to network cache only
        - not sure if we really need to expose it to content/. Can we move it to Blink?
    - set requestor id, existence of user gesture, empty http referrer when it shouldn't exposed.
        - ??? Still needs some investigation.
- UserAgentMetadata()
    - can be implemented by LocalFrameClientImpl.
- GetContentSettingsClient()
    - can be moved to LocalFrame and routed to LocalFrameClientImpl and eventually goes to web_frame_, which is WebLocalFrameImpl exposed to the embedder.
- CreateWebSocketHandshakeThrottle()
    - RenderFrameImpl implement it. Chrome and WebView implements CreateWebSocketHandshakeThrottleProvider. Need to keep it as is.
- DidDisplayInsecureContent()
    - IPC to the framehost.
- DidRunInsecureContent()
    - IPC to the framehost
- UserAgentOverride()
    - Returns a value in the renderer_preference or null string. If we move UAOverride to blink, we can use it in WillSendRequest.

WebWorkerFetchContextImpl

Perhaps we can remove it if we can move response override.

- Methods called by WorkerFetchContext
    - SiteForCookies()
        - a variable is kept in the instance. Probably we don't need to ask it in the content.
    - TopFrameOrigin()
        - same with site for cookeis

    - CreateWebSocketHandshakeThrottle()
        - RendererContentClient creates it and it's kept by the fetch context. We might be able to use it through blink::Platform().
    - WillSendRequest()
        - Adding a DNT header
            - not necessarily be in content.
        - RequestExtraData
            - render_frame_id => ??? Can we expose it in the renderer?
            - URLLoaderThrottles => it's RendererContentClient. It could be in blink::Platform.
        - Response override
            - Should be moved somewhere. Perhaps we can do something similar to the main resource navigation. This affects the work in WebURLLoaderImpl, so we need to move it in the first step.
            - Considering that the process of loading the main script is bound with ThreadableLoader tightly (e.g. CORS and leverage the method of loading a normal request), some potential ways of implementing this I have thought out:
                - Implement a loader in renderer/core/loader: unachievable , because the URLResponseHead doesn't have a native representative on blink variant currently.
                - Implement in the way of NavigationBodyLoader: seems plausible, but we have to add a WebXXX interface which is anti-OnionSoup, so I don't think it's a suitable one.
                - Adding a loader under blink/renderer/platform/loader/fetch/url_loader (**preferable**): as we have decided to introduce non-Blink mojom interface, we can implement something called MainScriptLoader, which has the similar behaviour of NavigationURLLoader. Meanwhile, the network::mojom::URLResponseHead and CORS utility are available there. Things need to be done (Work is ongoing, still have wpt failures):
                    - Have the functionality of WebURLLoaderImpl::PopulateURLResponse inside blink (this will be moved into URLLoaderClientImpl of blink side during finishing OnionSoup).
                    - Extract the CORS check for main script (a.k.a is_top_level_script_ in WorkerClassicScriptLoader).
        - Rewrite URL
            - It's a test-only feature. We don't have to care so much.
        - Adding Referrer
            - Not necessarily be in content.
- As a LoaderFactoryForWorker
    - WrapURLLoaderFactory()
        - Creates WebURLLoaderFactory with resource dispatcher. We can simply eliminate that.
    - GetScriptLoaderFactory(): ditto
    - GetURLLoaderFactory(): ditto

- CreateCodeCacheLoader()

ServiceWorkerFetchContextImpl

This can be moved to Blink once RendererPreference is available in Blink + render_frame_id is moved.
Needs a bit more investigation of how to remove frame_id (or make it available in Blink).

- WillSendRequest()
  - DNT
    - ok if renderer preference is available in Blink.
  - RequestExtraData
    - render_frame_id
    - frame_request_blocker
    - throttles
    - response override
  - g_rewrite_url
    - OK since it's only for testing.
  - enable referrers
    - ok if renderer p[reference is available in Blink.
- SetTerminateSyncLoadEvent
  - I think we can remove it
- InitializeOnWorkerThread(blink::AcceptLanguagesWatcher*) override;
  - can be moved
- GetURLLoaderFactory()
  - will be gone
- WrapURLLoaderFactory()
  - can be moved
- GetScriptLoaderFactory() override;
  - ok
- blink::mojom::ControllerServiceWorkerMode GetControllerServiceWorkerMode()
  - ok
- SiteForCookies() const override;
  - ok
- base::Optional<blink::WebSecurityOrigin> TopFrameOrigin() const override;
  - ok
- CreateWebSocketHandshakeThrottle(
  - Needs ancestor_frame_id. I'm wondering if getting a frame id in Blink might be layering violation, but technically it's possible.
- blink::WebString GetAcceptLanguages() const override;
  - ok to move if we can use renderer preferences in Blink.

What will it look like? (step 3)

We are going to eliminate WebURLRequest.
This doc captures how it's now used and considers
https://docs.google.com/document/d/1C8S2e20imwIFxEIKgvqcgDGCErOIWCTB34ABAF07QAc/edit#

Oct 2 (backlog)

Notes about WebURLRequest removal

WillSendRequest can modify the struct.

- adding DNT header
- file path alias (???) to update URL
- getting requestor origin
- URL could be updated by the embedder
- setting cache mode override
- extra data for user agent
- extra data for navigationresponseoverride
- Creating settings an extra data
    - throttles
    - update Previews state (???)
- download to network cache only =>> not sure if we really need to expose it to content/. Can we move it to Blink?
- set requestor id, existence of user gesture, empty http referrer when it shouldn't exposed.

We don't want to copy if it's performance sensitive

- media would be the one.
- resource_multibuffer_data_provider.cc
- multi_resolution_image_resource_fetcher.cc

Consensus?

WebAssociatedURLLoader can take mojom::URLRequest

- Media etc can use it
- If perf becomes a concern media can accelerate their onion soup

Observers

- resource_load_stats tasks each param as they need.
- This doesn't take URLRequest, but most of them take ResourceResponseHead
- While they're out of blink they can just use mojom one

RenderFrameImpl::WillSendRequest

- Takes URLRequest, mutable one
- At this stage (i.e. in ResourceFetcher::PrepareRequest, which happens early in the resource loading) blink still wants to use blink::ResourceRequest
- ~~One plausible idea is to pass a limited view struct (similar to WebURLRequest???)~~
    - More code should be moved into blink
- Keep blink::WebURLRequest for now, but start removing methods that are only used by WebURLLoaderImpl. Also add a clear comment to deprecate this struct, discourage new usage

RenderFrameImpl::DidStartResponse

- Takes mojom::ResourceResponseHead --> can be okay as is

Try to summarize the discussion

Design around ResourceLoader and URLLoaderThrottle

What we need are:

- Add a typemapping in Blink:
  network.mojom.URLRequest => **network::ResourceRequest**
  network.mojom.URLResponseHead => **network::ResourceResponseHead**
  Note that **eventually we'll remove the typemapping** and then going to use network::mojom::URLRequest/URLResponseHead (native Mojo structs).
- Put URLLoaderThrottle interface in **blink/public/common/loader/.**
- Create ThrottlingURLLoader in **blink/renderer/platform/loader/ (blink variant)** in addition to content/browser/loader/ (chromium variant).
- Let ThrottlingURLLoader and some new code use **non-Blink Mojo types (**network::mojom::URLLoader, URLLoaderFactory, URLLoaderClient, **not** network::mojom::**blink::**URLLoader*).

ResourceLoader would look like this.

```
ResourceLoader::RequestAsynchronously(const ResourceRequest& request) {
  std::vector<std::unique_ptr<blink::URLLoaderThrottle>> throttles =
    Context().CreateURLLoaderThrottles();
  scoped_refptr<blink::URLLoaderFactoryBundle> bundle =
    Context().GetURLLoaderFactoryBundle();
  network::mojom::URLRequest network_request =
      ConvertTo<network::mojom::URLRequest>(request);
  loader_ = blink::ThrottlingURLLoader::CreateLoaderAndStart(
      bundle, std::move(throttles), …,
      network_request,
      this /* network::mojom::URLLoaderClient* client */,
      ...);
}

ResourceLoader::OnReceiveResponse(
    network::mojom::ResourceResponseHeadPtr head) {
}
ResourceLoader::OnStartLoadingResponseBody(mojo::ScopedDataPipeConsumerHandle
pipe) {...}
...
```

`blink::URLLoaderThrottle` would be a public API.

```
// /third_party/blink/public/common/loader/url_loader_throttles.h
class URLLoaderThrottle {
  class CONTENT_EXPORT Delegate {
   public:
    virtual void CancelWithError(int error_code,
                                 base::StringPiece custom_reason = nullptr) =
0;
    virtual void Resume() = 0;
    virtual void SetPriority(net::RequestPriority priority);
    virtual void UpdateDeferredRequestHeaders(
        const net::HttpRequestHeaders& modified_request_headers);
    virtual void UpdateDeferredResponseHead(
        const network::mojom::ResourceResponseHead& new_response_head);
    virtual void PauseReadingBodyFromNet();
    virtual void ResumeReadingBodyFromNet();
    void InterceptResponse(
        network::mojom::URLLoaderPtr new_loader,
        network::mojom::URLLoaderClientRequest new_client_request,
        network::mojom::URLLoaderPtr* original_loader,
        network::mojom::URLLoaderClientRequest* original_client_request);
    virtual void RestartWithFlags(int additional_load_flags);
  };

  virtual void DetachFromCurrentSequence();
  virtual void WillStartRequest(network::mojom::ResourceRequest* request,
bool* defer);
  virtual void WillRedirectRequest(
      net::RedirectInfo* redirect_info,
      const network::mojom::ResourceResponseHead& response_head,
      bool* defer,
      std::vector<std::string>* to_be_removed_request_headers,
      net::HttpRequestHeaders* modified_request_headers);
  virtual void WillProcessResponse(  JLKJLKt
      const GURL& response_url,
      network::mojom::ResourceResponseHead* response_head,
      bool* defer)
  virtual void BeforeWillProcessResponse(
      const GURL& response_url,
      const network::mojom::ResourceResponseHead& response_head,
      bool* defer);
  virtual void WillOnCompleteWithError(
      const network::mojom::URLLoaderCompletionStatus& status,
      bool* defer);
};
```

Steps

I'll describe the order of the tasks.
TODO(leon): complete this.

1. Change typemapping as following this doc: https://docs.google.com/document/d/1B6SkPVvovGFJcIL4ssRyveIPypVJxZhjIJuK9_xnt-4/edit#
2. Move URLLoaderThrottle from content/public/common into blink/public/common/.
3. Convert RequestPeer to URLLoaderThrottles if possible.
4.

Appendix: Idea notes

How to add “extra” data as blink-recognizable way

One (very rough but maybe possible) idea is to add “extra internal headers” field in network::ResourceRequest. We carry the internal headers to pass through the extended info from the initiator to just before sending the socket. It allows us to tell all the info to where it needs. For example, a service worker needs to track where the request comes from, and we have “fetch_request_context_type” for that purpose. It should not be in the network::ResourceRequest, but it should be accompanied with the request. The “extra headers” concept might be matching with our needs. Maybe it’s worth having separated field for the extra internal headers in network::ResourceRequest not to expose those headers to outside of chrome.
The downside of it is that we need codes to serialize/deserialize the extra headers. It may add some amount of cost and also it loses type. Needs some discussion.

====

Backlog - Options to implement URLLoaderThrottle

There were long investigation on how to implement URLLoaderThrottle. Let me put the notes here for future reference.

Options

1. Having ThrottlingURLLoader and URLLoaderThrottle in **services/network/public/**.
    - We don’t have to change existing browser-only throttles in this option.
    - We need to allow Blink to use network::ResourceRequest and network::ResourceResponseHead.
    - Also we need to think of the mojom types. ThrottlingURLLoader and URLLoaderThrottle are using chromium-variant Mojo interfaces, but Blink is not allowed to use them. Should we have very thin wrapper class to expose them to Blink? (though that’s sad. described below.)
    - If we typemap network::ResourceRequest to blink::ResourceRequest in Blink, Blink loader code (maybe it’s blink::ResourceLoader) needs to create network::ResourceRequest from blink::ResourceRequest to use network::ThrottlingURLLoader/URLLoaderThrottle.
    - However, we still need some code to convert blink::ResourceRequest to network::ResourceRequest somehow even if we don’t typemap network::ResourceRequest.

2. Having the interface class of URLLoaderThrottle in **blink/public/common/** and **adding ThrottlingURLLoader** in blink/renderer/platform/loader/. Let content/renderer/ create throttles.
    - After typemapping, we'll use blink::ResourceRequest as a request to send to network::mojom::URLLoader in Blink, but it needs to be exposed in content/ and other components and embedder (chrome/) as a parameter of URLLoaderThrottle. We need type conversion.
    - We need to have two ThrottlingURLLoaders.
    - We don't have to modify URLLoaderThrottles. They just need to override **blink**::URLLoaderThrottle instead of **content**::URLLoaderThrottle.
    - **Currently this seems most plausible (link to section).**
3. Having two URLLoaderThrottle/ThrottlingURLLoader in content/browser/ and blink/common/.
    - We need to have the implementation of MimeSniffingThrottle and some of other throttles under components/ and chrome/ into blink. I don't think we can do this because of layering violation.
4. Having the interface class of URLLoaderThrottle as mojo interface in blink/public/mojom, and let content/renderer/ create throttles.
    - It doesn't seem to be able to do this since all method calls will be asynchronous, but if possible, we don't have to care about the types across the blink/content boundary.
    - We might have some additional cost of function calls to check the interception.

Digging into option 1

Let's think of the design based on small example code.
As the option 1, I'm imagining something like this, but that will have a few problems:

```
ResourceLoader::RequestAsynchronously(const ResourceRequest& request) {
  std::vector<std::unique_ptr<network::URLLoaderThrottle>> throttles =
    Platform()->CreateURLLoaderThrottles();
  loader_ = network::ThrottlingURLLoader::CreateLoaderAndStart(
      loader_factory_.get(), std::move(throttles), …,
      ConvertToNetworkResourceRequest(request),            // Problem A
      this /* network::mojom::URLLoaderClient* client */,  // Problem B
      ...);
}

ResourceLoader::OnReceiveResponse(
    network::mojom::blinkResourceResponseHeadPtr head) { // Problem C
}
ResourceLoader::OnStartLoadingResponseBody(mojo::ScopedDataPipeConsumerHandle
pipe) {...}
...
```

Problem A:
ConvertToNetworkResourceRequest() is sad, but we may not need to have the typemapping for network.mojom.URLRequest.

Problem B:

If we used network::ThrottlingURLLoader in Blink, we would need to pass network::mojom::URLLoaderClient, but ResourceLoader would override network::mojom**::blink::**URLLoaderClient. This seems complicated.

For example, this could solve the issue?

```
ResourceLoader::RequestAsynchronously(const ResourceRequest& request) {
  Platform()->CreateLoaderAndStart(
    static_cast<URLLoaderThinInterfaceWhichConvertsType>(this),
    request, ...);
}
```

Ah, but this looks very similar to the current state (before onion soup)....

Problem C:
We need to have these native structs as Mojo structs. Maybe it'd be just creating them. I don't think this is a big deal.

- struct URLResponseHead;
- struct URLRequestRedirectInfo;
- struct CorsErrorStatus;
- struct URLLoaderCompletionStatus;

Digging into Option 2

- We would have two ThrottlingURLLoader (in blink and in content).
- Typemaps for blink variant:
    - network.mojom.URLRequest => network::ResourceRequest
    - network.mojom.URLRequestBody => network::ResourceRequestBody
-

```
ResourceLoader::RequestAsynchronously(const ResourceRequest& request) {
  std::vector<std::unique_ptr<blink::URLLoaderThrottle>> throttles =
    blink::CreatePlatformURLLoaderThrottes();
  loader_ = blink::ThrottlingURLLoader::CreateLoaderAndStart(
      loader_factory_.get(), std::move(throttles), …,
      request, // OK
      this /* network::mojom::blink::URLLoaderClient* client */, // OK
      ...);
}

ResourceLoader::OnReceiveResponse(
    network::mojom::blink::ResourceResponseHeadPtr head) {  // OK
}
ResourceLoader::OnStartLoadingResponseBody(mojo::ScopedDataPipeConsumerHandle
pipe) {...}
...
```

This seems solving issues from option 1.
Here is an example of blink::URLLoaderThrottle which would be in blink/renderer/platform/.
This wraps WebURLLoaderThrottle to convert blink types to chromium types. Bold means there is something notable in some aspects.

```
// /third_party/blink/renderer/platform/loader/url_loader_throttle.h
class URLLoaderThrottle {
  // Not sure if this should be GC-able object.
  class CONTENT_EXPORT Delegate {
   public:
    virtual void CancelWithError(int error_code,
                                 base::StringPiece (or some equivalent in
Blink) custom_reason = nullptr) = 0;
    virtual void Resume() = 0;
    virtual void SetPriority(net::RequestPriority priority);
    virtual void UpdateDeferredRequestHeaders(
        const net::HttpRequestHeaders& modified_request_headers);
    virtual void UpdateDeferredResponseHead(
        const blink::ResourceResponseHead& new_response_head);
    virtual void PauseReadingBodyFromNet();
    virtual void ResumeReadingBodyFromNet();
    virtual void InterceptResponse(
        network::mojom::blink::URLLoaderPtr new_loader,
        network::mojom::blink::URLLoaderClientRequest new_client_request,
        network::mojom::blink::URLLoaderPtr* original_loader,
        network::mojom::blink::URLLoaderClientRequest*
original_client_request);
    virtual void RestartWithFlags(int additional_load_flags);
  };

  void DetachFromCurrentSequence();
  void WillStartRequest(blink::ResourceRequest* request, bool* defer);
  void WillRedirectRequest(
      net::RedirectInfo* redirect_info,
      const blink::ResourceResponseHead& response_head,
      bool* defer,
      std::vector<std::string>* to_be_removed_request_headers,
      net::HttpRequestHeaders* modified_request_headers);
  void WillProcessResponse(const KURL& response_url,
                           blink::ResourceResponseHead* response_head,
                           bool* defer)
  void BeforeWillProcessResponse(
      const KURL& response_url,
      const network::ResourceResponseHead& response_head,
      bool* defer);
  void WillOnCompleteWithError(
      const network::mojom::blink::URLLoaderCompletionStatus& status,
      bool* defer);
  void set_delegate(std::unique_ptr<Delegate> delegate);
  private:
    std::unique_ptr<Delegate> delegate_;
    std::unique_ptr<WebURLLoaderThrottle> instance_;
```

```
};

std::vector<blink::URLLoaderThrottles> CreatePlatformURLLoaderThrottles();
```

Then this is an example of CreatePlatformURLLoaderThrottles implementation.

```
std::vector<blink::URLLoaderThrottles>
CreatePlatformURLLoaderThrottles() {
  std::vector<std::unique_ptr<WebURLLoaderThrottle>> throttles =
      Platform()->CreateURLLoaderThrottles();
  std::vector<std::unique_ptr<URLLoaderThrottle>> out_throttles;
  for (auto& throttle : throttles) {
    out_throttles.emplace(std::make_unique<URLLoaderThrottle>(
        std::move(throttle));
  }
  return out_throttle;
}
```

and this is an impl of URLLoaderThrottle::WillStartRequest.

```
// /third_party/blink/renderer/platform/loader/url_loader_throttle.cc

void URLLoaderThrottle::WillStartRequest(blink::ResourceRequest* request,
bool* defer) {
  instance_->WillStartRequest(WrapResourceRequest(request), defer);
}
```

And the WebURLLoaderThrottle will look like this.

```
// /third_party/blink/public/common/loader/url_loader_throttles.h
class WebURLLoaderThrottle {
  class CONTENT_EXPORT Delegate {
   public:
    virtual void CancelWithError(int error_code,
                                 base::StringPiece custom_reason = nullptr) =
0;
    virtual void Resume() = 0;
    virtual void SetPriority(net::RequestPriority priority);
    virtual void UpdateDeferredRequestHeaders(
        const net::HttpRequestHeaders& modified_request_headers);
    virtual void UpdateDeferredResponseHead(
        const network::ResourceResponseHead& new_response_head);
    virtual void PauseReadingBodyFromNet();
    virtual void ResumeReadingBodyFromNet();
    virtual void InterceptResponse(
        network::mojom::URLLoaderPtr new_loader,
        network::mojom::URLLoaderClientRequest new_client_request,
        network::mojom::URLLoaderPtr* original_loader,
        network::mojom::URLLoaderClientRequest* original_client_request);
    virtual void RestartWithFlags(int additional_load_flags);
  };
```

```
  virtual void DetachFromCurrentSequence();
  virtual void WillStartRequest(network::ResourceRequest* request, bool*
defer);
  virtual void WillRedirectRequest(
      net::RedirectInfo* redirect_info,
      const network::ResourceResponseHead& response_head,
      bool* defer,
      std::vector<std::string>* to_be_removed_request_headers,
      net::HttpRequestHeaders* modified_request_headers);
  virtual void WillProcessResponse(
      const GURL& response_url,
      network::ResourceResponseHead* response_head,
      bool* defer)
  virtual void BeforeWillProcessResponse(
      const GURL& response_url,
      const network::ResourceResponseHead& response_head,
      bool* defer);
  virtual void WillOnCompleteWithError(
      const network::mojom::URLLoaderCompletionStatus& status,
      bool* defer);
};
```

Hmm, the conversion of blink::ResourceRequest -> network::ResourceRequest in URLLoaderThrottle will be a problem.
URLLoaderThrottles are possible to mutate the parameters in network::ResourceRequest, and those changes should be reflected in blink::ResourceRequest. We can solve the issue by creating network::ResourceRequest at the beginning of blink::URLLoaderThrottle::WillStartRequest() and converting back to blink::ResourceRequest at the end of it, but we don't want to do that due to the cost. Maybe we need to think of another way.

```
void URLLoaderThrottle::WillStartRequest(blink::ResourceRequest* request,
bool* defer) {
  network::ResourceRequest network_request =
      CreateNetworkResourceRequest(request); // sad
  instance_->WillStartRequest(network_request, defer);
  request = CreateBlinkResourceRequest(network_request); // sad
}
```

Most plausible plan

So maybe it's easier to make URLLoaderThrottle understand network::ResourceRequest natively.
It means that we won't have a typemap from network::ResourceRequest -> blink::ResourceRequest.
network::mojom::blink::URLLoader will receive network::ResourceRequest.

===
TL;DR: In this option...
Typemapping in Blink: network.mojom.URLRequest/Response => **network::ResourceRequest/Response**

URLLoaderThrottle interface is in **blink/public/common/loader/.**
ThrottlingURLLoader is in **blink/renderer/platform/loader/ (blink variant) and content/browser/loader/ (chromium variant).**
===

ResourceLoader would look like this in this option.

```
ResourceLoader::RequestAsynchronously(const ResourceRequest& request) {
  std::vector<std::unique_ptr<blink::URLLoaderThrottle>> throttles =
    blink::CreatePlatformURLLoaderThrottes();
  network::ResourceRequest network_request =
      ConvertTo<network::ResourceRequest>(request);  // Problem A
  loader_ = blink::ThrottlingURLLoader::CreateLoaderAndStart(
      loader_factory_.get(), std::move(throttles), …,
      network_request,
      this /* network::mojom::blink::URLLoaderClient* client */, // OK
      ...);
}

ResourceLoader::OnReceiveResponse(
    const network::ResourceResponseHead& head) { // OK, maybe?
}
ResourceLoader::OnStartLoadingResponseBody(mojo::ScopedDataPipeConsumerHandle
pipe) {...}
...
```

This time, `blink::URLLoaderThrottle` would be a public API.

```
// /third_party/blink/public/common/loader/url_loader_throttles.h
class URLLoaderThrottle {
  class CONTENT_EXPORT Delegate {
   public:
    virtual void CancelWithError(int error_code,
                                 base::StringPiece custom_reason = nullptr) =
0;
    virtual void Resume() = 0;
    virtual void SetPriority(net::RequestPriority priority);
    virtual void UpdateDeferredRequestHeaders(
        const net::HttpRequestHeaders& modified_request_headers);
    virtual void UpdateDeferredResponseHead(
        const network::ResourceResponseHead& new_response_head);
    virtual void PauseReadingBodyFromNet();
    virtual void ResumeReadingBodyFromNet();
    void InterceptResponse(
        network::mojom::URLLoaderPtr new_loader,
        network::mojom::URLLoaderClientRequest new_client_request,
        network::mojom::URLLoaderPtr* original_loader,
        network::mojom::URLLoaderClientRequest* original_client_request);
```

```
    virtual void RestartWithFlags(int additional_load_flags);
#ifdef INSIDE_BLINK  // This is tricky.
    // blink::URLLoaderThrottle::Delegate::InterceptResponse() calls this.
    // Please see below for an example implementation.
    virtual void InterceptResponse(
        network::mojom::blink::URLLoaderPtr new_loader,
        network::mojom::blink::URLLoaderClientRequest new_client_request,
        network::mojom::blink::URLLoaderPtr* original_loader,
        network::mojom::blink::URLLoaderClientRequest*
            original_client_request);
#endif
  };

  virtual void DetachFromCurrentSequence();
  virtual void WillStartRequest(network::ResourceRequest* request, bool*
defer);
  virtual void WillRedirectRequest(
      net::RedirectInfo* redirect_info,
      const network::ResourceResponseHead& response_head,
      bool* defer,
      std::vector<std::string>* to_be_removed_request_headers,
      net::HttpRequestHeaders* modified_request_headers);
  virtual void WillProcessResponse(  JLKJLKt
      const GURL& response_url,
      network::ResourceResponseHead* response_head,
      bool* defer)
  virtual void BeforeWillProcessResponse(
      const GURL& response_url,
      const network::ResourceResponseHead& response_head,
      bool* defer);
  virtual void WillOnCompleteWithError(
      const network::mojom::URLLoaderCompletionStatus& status,
      bool* defer);
};
```

This is an example of the implementation of blink::URLLoaderThrottle::InterceptResponse().

```
// /third_party/blink/common/loader/url_loader_throttles.cc
void URLLoaderThrottle::Delegate::InterceptResponse(
        network::mojom::URLLoaderPtr new_loader,
        network::mojom::URLLoaderClientRequest new_client_request,
        network::mojom::URLLoaderPtr* original_loader,
        network::mojom::URLLoaderClientRequest* original_client_request) {
  // Convert the namespace.
  InterceptResponse(
    network::mojom::blink::URLLoaderPtrInfo(new_loader.PassHandle(),
        network::mojom::blink::URLLoaderPtrInfo::Version_),
    ...);
}
```