Freezed: Read Only Now. Please ask toyoshim@ to edit/comment

WebFonts: Cache-aware Timeout Adaption

Sub-document of "WebFonts: Adaptive Font Timeouts"

Created: 2015-11-27
Last modified: 2015-12-14
Status: **public (Anyone with the link can comment)**
By: Takashi Toyoshima <toyoshim@{google.com|chromium.org}>
crbug: 570205
Link to this doc: WebFonts: Cache-aware Timeout Adaption

Objective

Reduce unnecessary FOUT: "Flash of Unstyled Text" as much as possible, especially on having fonts in the cache.

Background

If WebFonts loading hits fallback timeout, texts using the font is rendered with a fallback font, then replaced with the right font when the font loading is finished. This rarely happens on fast network since 3 sec timeout is enough to load typical fonts. But situation is going to be changed once CSS font-display is shipped. The font-display defines several modes including swap, fallback, optional. These mode require rendering text with a fallback font as soon as possible. Thus unnecessary FOUT can happen when the font is on cache, but it takes a little long time, e.g. 100 msec, to load from the disk.

Overview

This is a timeout matrix that summarizes CSS Font Rendering Controls Module Level 1. The timeout count should be started when the UA attempts to use a font, or starts font loading.

attribute	fallback	swap	cache-aware
auto	adaptive	adaptive	depends
block	3 sec	∞	no
swap	0 sec	∞	probably yes

fallback	100 msec	3 sec	yes
optional	100 msec	0 sec	yes

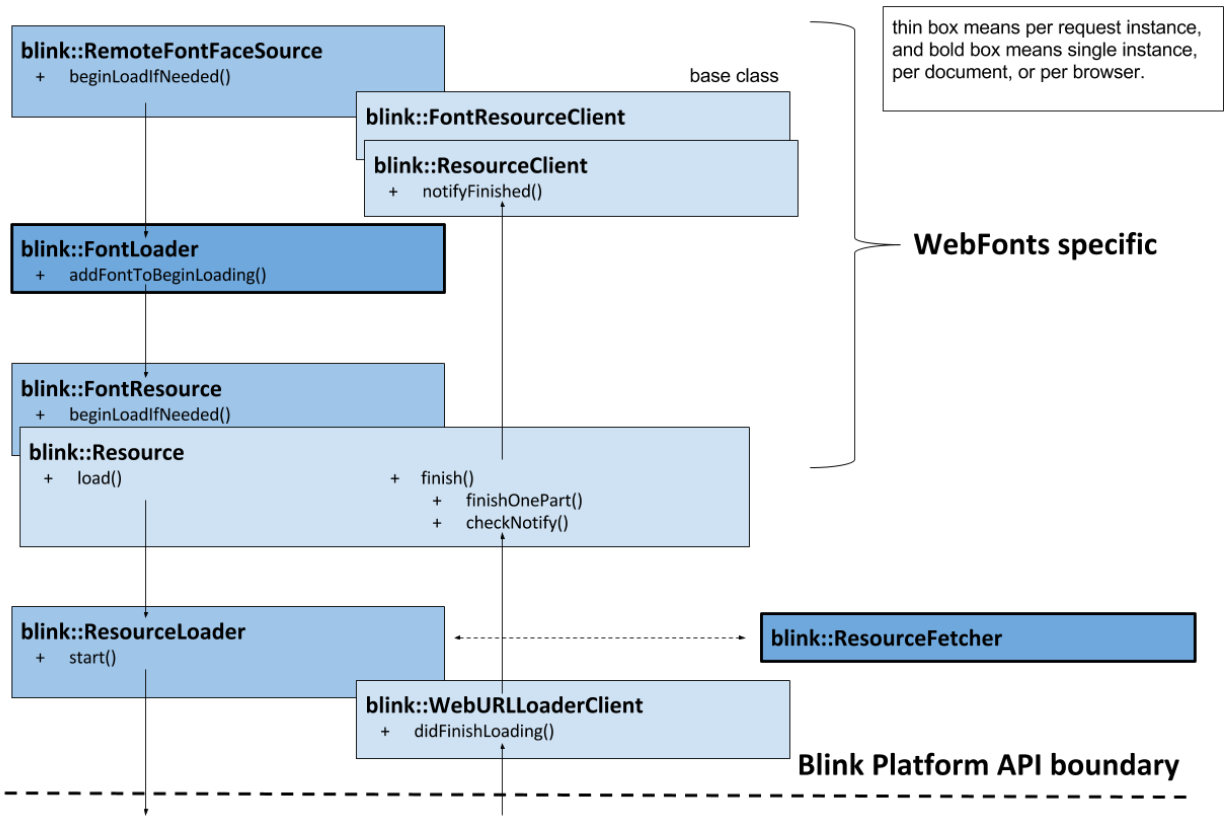
In the matrix, fallback and optional define the fallback timeout as 100 msec. The spec expects 100 msec is reasonable not to take fallback on having the font in the cache. To avoid unnecessary FOUT, it should be reasonable to wait for the font if the font exists in the cache.
Even for auto mode, the browser may want to adapt the same strategy on slow networks like 2G.
We may want to adapt the same strategy on swap mode, but it should be applied only if the cache access is fast enough. We would revisit this specific case later.

Design

There are two strategies to realize cache-aware loading, and there are pros and cons for each. I will implement both strategies because the one fits ‘fallback’ and the other fits ‘optional’.

Plan A: Load Only Cached Font, and Try Loading Later

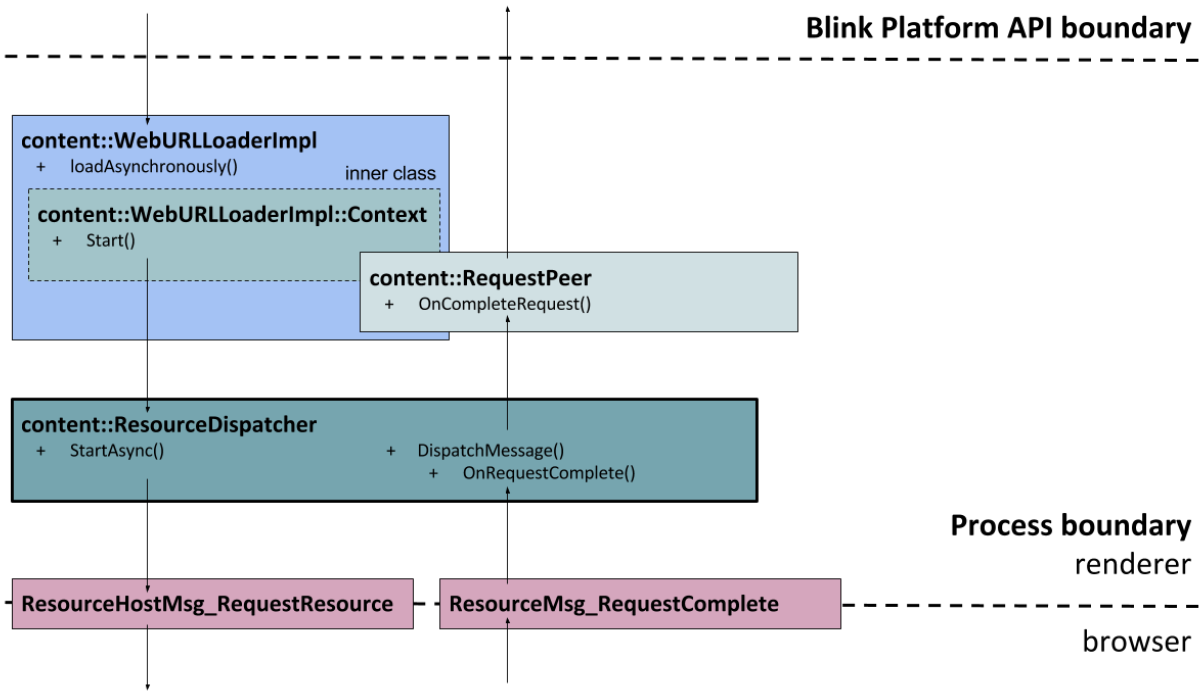
Now, blink::WebURLLoader can accept a flag blink::WebURLRequerst::ReturnCacheDataDontLoad that allows a client to load data only from caches. We can use the flag to realize the first load. Then, try another loading after ‘onload’ event is fired for the next navigation.
To use the flag from WebFonts, I will add a flag to blink::ResourceLoaderOptions. Following diagram shows WebFonts loading call and callback chains.

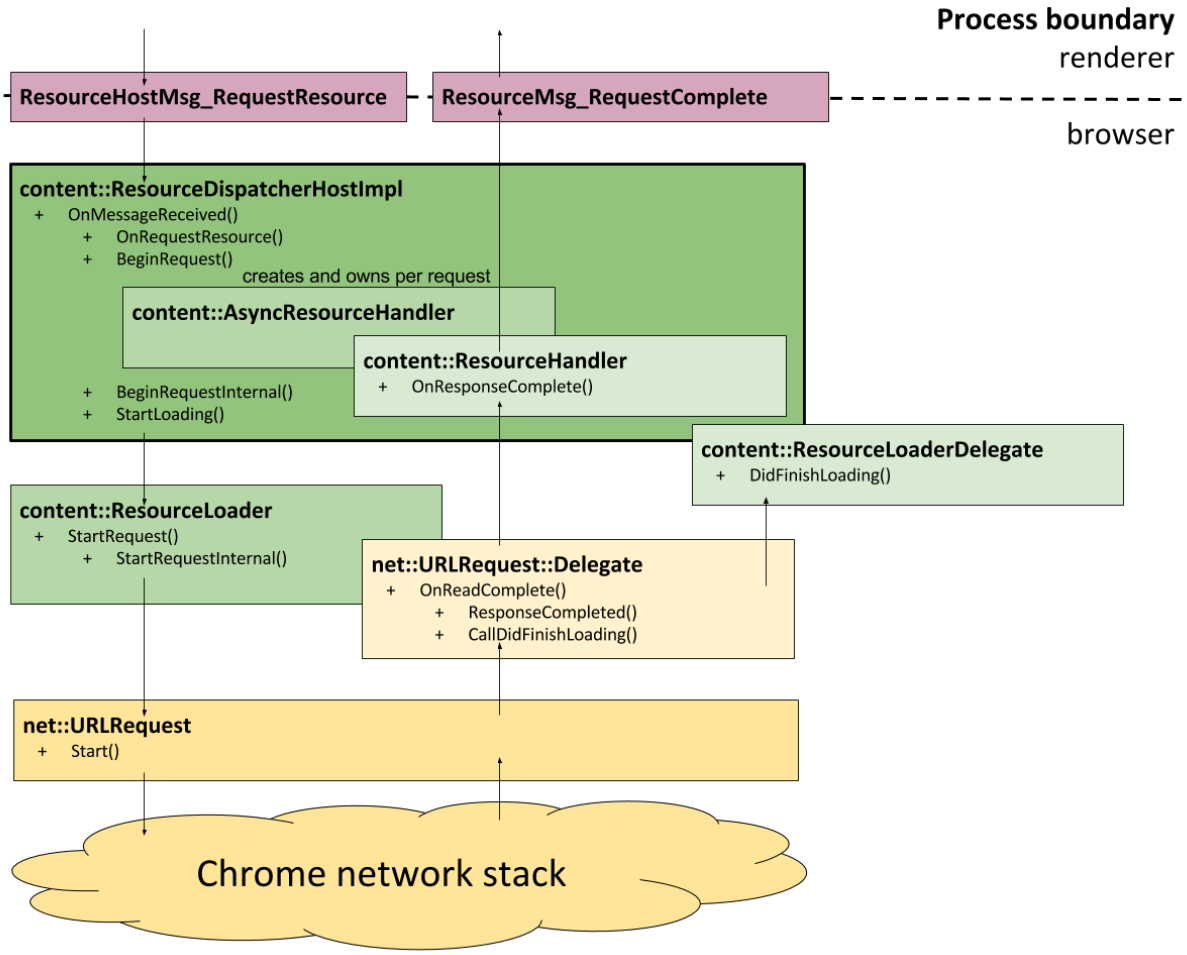


blink::RemoteFontFaceSource manages font-display properties, and will make a decision to use the flag. ResourceLoader just pretends to fail loading if the request does not hit the cache.

Plan B: Wait for Cache Results before Invoking Timeouts

Plan A is simple and only Blink side change is needed. But it does not fit the use case for 'fallback'. I want to know the cache status before handling timeouts, but want to keep loading anyway. Another option is to base on the Plan A, but to issue the second request immediately. This may work somehow, but double requests hit the cache manager, and cause an unnecessary RTT between the Blink and the browser process that hosts cache.
net::URLRequestDelegate::OnBeforeNetworkStart() is the right callback to know if the request does not hit the cache. If I plumb the callback from the Chrome net stack to the Blink, I can simply postpone timeout invocations until the callback is triggered. If the callback isn't triggered, it means the request hits the cache, and the data comes first.
Following two pictures show the call and callback chains outside Blink.


Blink Platform API boundary
content::WebURLLoaderImpl
+ loadAsynchronously()
inner class
content::WebURLLoaderImpl::Context
+ Start()
content::RequestPeer
+ OnCompleteRequest()
content::ResourceDispatcher
+ StartAsync()
+ DispatchMessage()
+ OnRequestComplete()
Process boundary
renderer
ResourceHostMsg_RequestResource
ResourceMsg_RequestComplete
browser



Now the OnBeforeNetworkStart() callback is terminated in the content::AsyncResourceHandler, and higher level clients do not have a corresponding callback. So, to plumb the callback, I should define a new IPC to propagate the event from the browser to the renderer, and add callback chains in the renderer side.

DevTools

We have a new behaviors in the Plan A, and there would be a discussion how the sequence should appear in the DevTools' Network timeline.

New Sequence

1. Request
2. Cache miss
3. Stalled until 'onload'
4. Loading through networks
5. Final results (200, 404, and so on)

TODO: Ask experts. I prefer simple solution, like just hide cache miss and stalled state in the step 2 and 3, and showing 2-4 together as a one long period of loading or waiting for scheduling. But if this will confuse developers, I should consider altenative plan.

Note: If ‘Disable cache’ is enabled from the DevTools, ReturnCacheDataDontLoad behaves as loading from the network always, as the flag is ignored.

TODO: How is this aligned with Resource Timing?

Timeline

I’d implement the Plan A for the ‘optional’ first, then implement the Plan B. That will happen in 2016 early Q1.