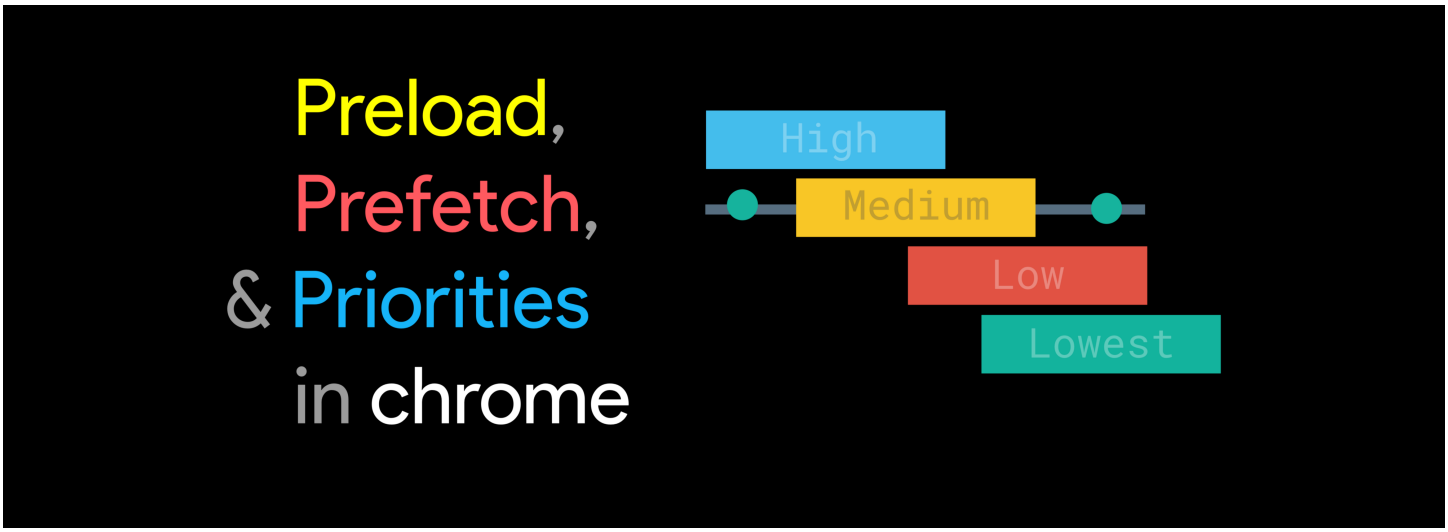


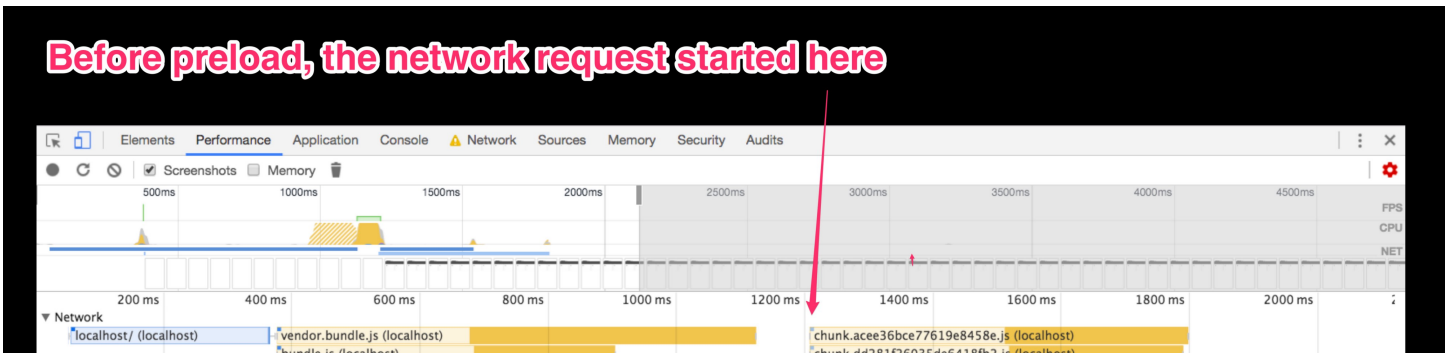
Preload, Prefetch And Priorities in Chrome

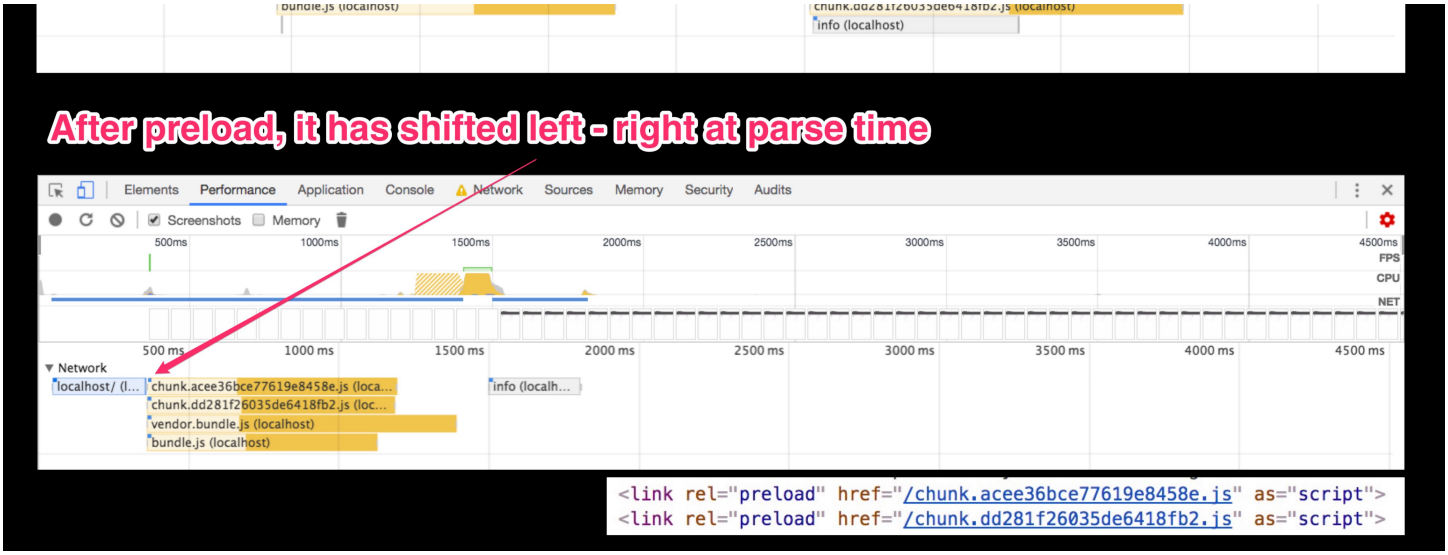
Addy Osmani
Mar 27, 2017 · 12 min read

Today we'll dive into insights from Chrome's networking stack to provide clarity on how web loading primitives (like **<link rel="preload">** & **<link rel="prefetch">**) work behind the scenes so you can be more effective with them.

As covered well in other articles, **preload is a declarative fetch, allowing you to force the browser to make a request for a resource without blocking the document's onload event.**

Prefetch is a hint to the browser that a resource might be needed, but delegates deciding whether and when loading it is a good idea or not to the browser.



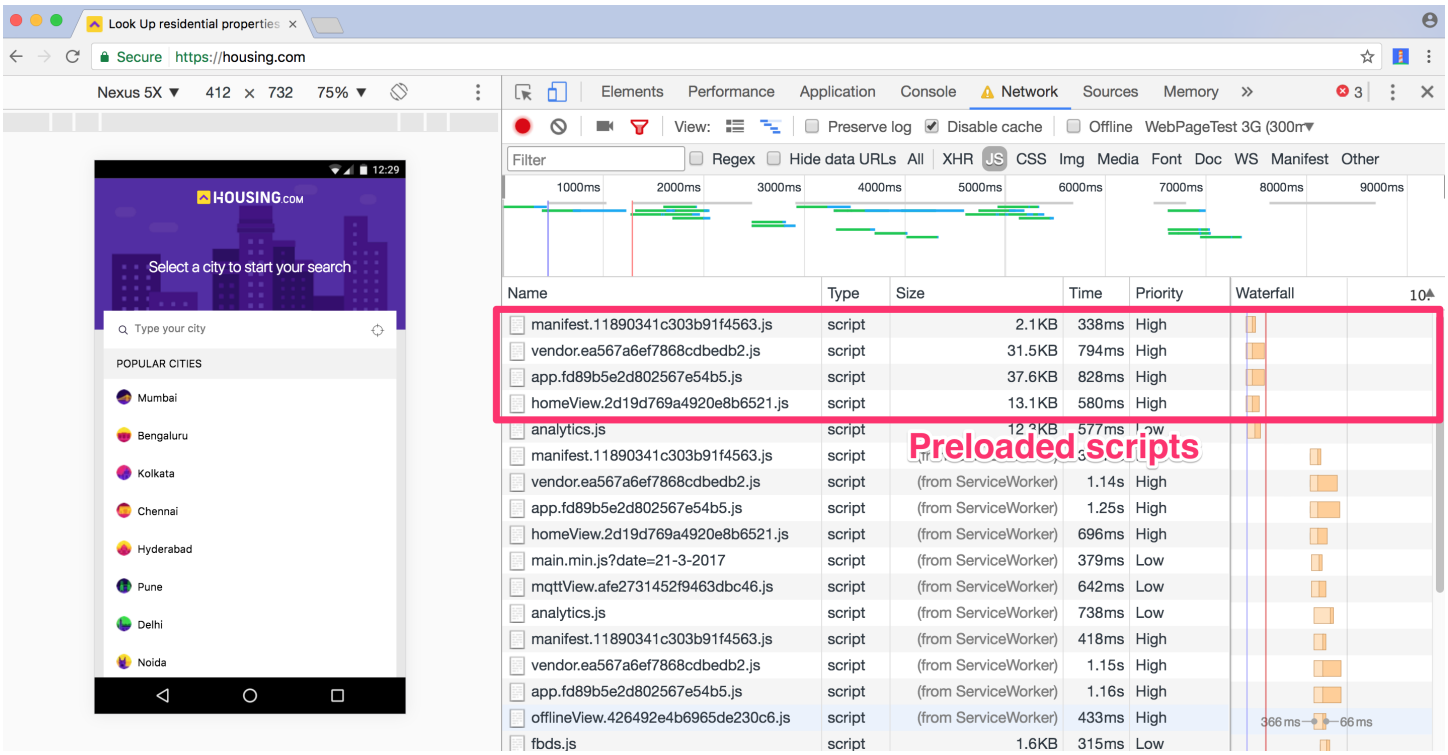


Preload can decouple the load event from script parse time. If you haven't used it before, read 'Preload: What is it Good For?' by Yoav Weiss

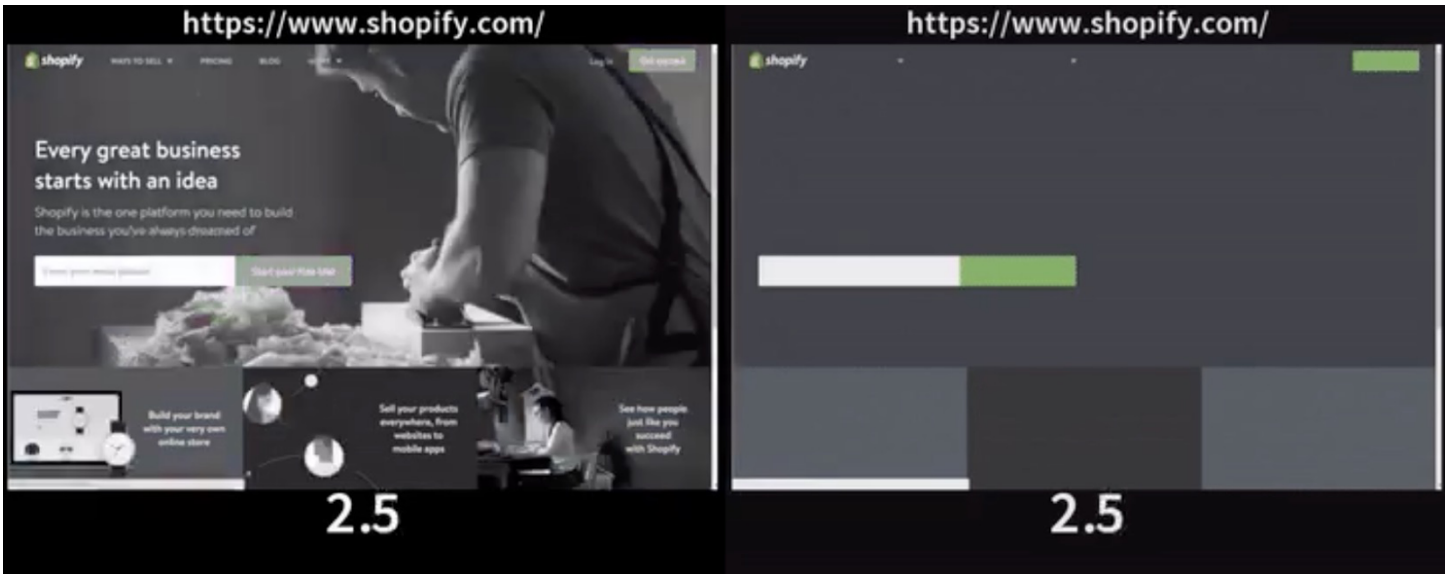
Preload success stories in production sites

Before we dive into the details, here's a quick summary of some positive impact to loading metrics that have been observed using preload in the last year:

Housing.com saw a **~10% improvement in Time to Interactive** when they switched to preloading key late-discovered scripts for their Progressive Web App:



Shopify's switch to preloading Web Fonts saw a **50% (1.2 second) improvement in time-to-text-paint** on Chrome desktop (cable). This removed their flash-of-invisible text completely.

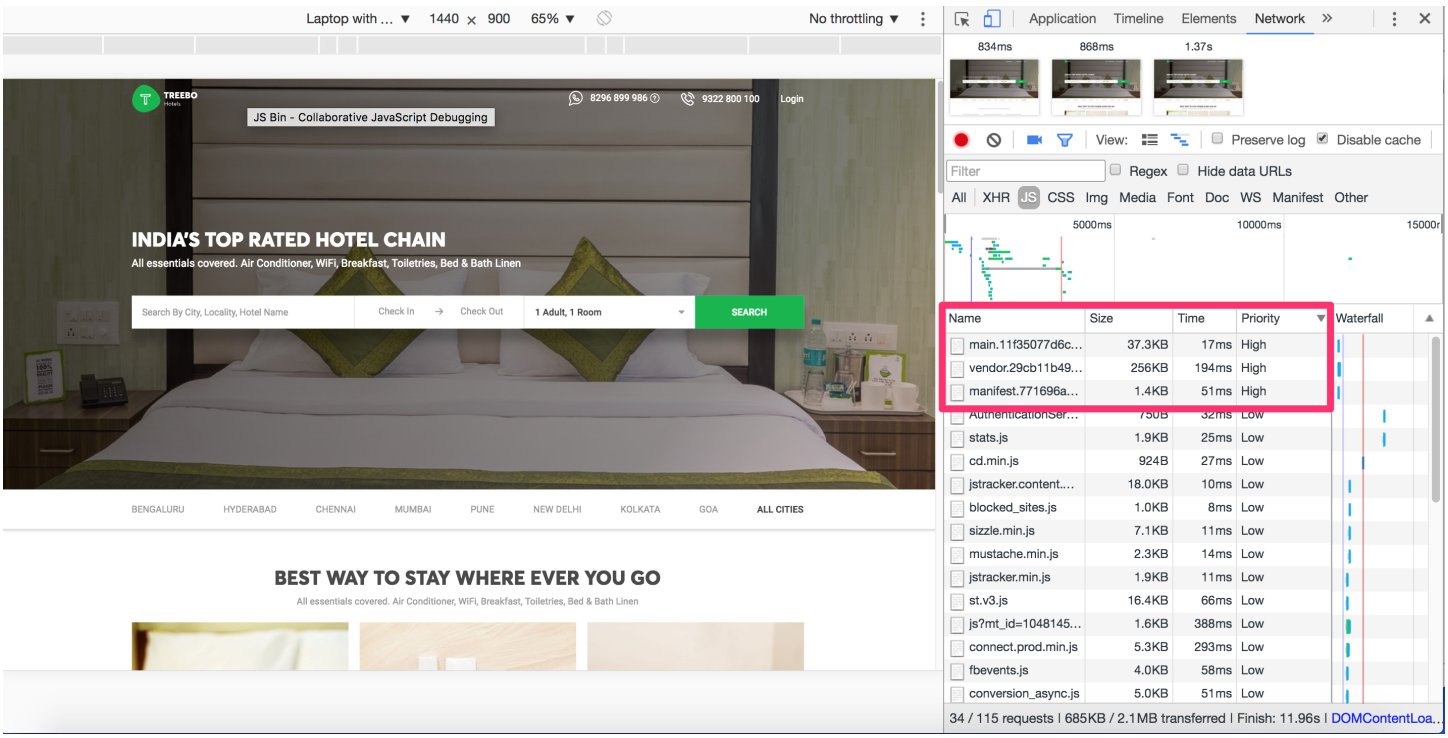


Left: with preload, Right: without (video)

```
<link rel="preload" as="font" crossorigin="crossorigin" type="font/woff2" href=
"https://cdn.shopify.com/shopify-marketing_assets/static/Brandon--light.woff2">
<link rel="preload" as="font" crossorigin="crossorigin" type="font/woff2" href=
"https://cdn.shopify.com/shopify-marketing_assets/static/Brandon--medium.woff2">
<link rel="preload" as="font" crossorigin="crossorigin" type="font/woff2" href=
"https://cdn.shopify.com/shopify-marketing_assets/static/Brandon--bold.woff2">
```

Web Font loading using <link rel="preload">

Treebo, one of India's largest hotel chains **shaved 1 second off both time to First Paint and Time to Interactive** for their desktop experience over 3G, by preloading their header image and key Webpack bundles:



Similarly, by switching to preloading their key bundles, Flipkart **shaved a great deal of main thread idle** before route chunks get evaluated on their PWA (trace from a low-end phone over 3G):

Top: without preload, Bottom: with preload

And the Chrome Data Saver team saw **time to first contentful paint improvements of 12% on average** for pages that could use preload on scripts and CSS stylesheets.

As for prefetch, it's widely used and at Google we still use it in Search results pages to prefetch critical resources that can speed up rendering destination pages.

Preload is used in production by large sites for a number of use-cases and you can find more of them later on in the article. Before that, let's dive into how the network stack actually treats preload vs prefetch.

When should you <link rel="preload"> vs <link rel="prefetch">?

Tip: Preload resources you have high-confidence will be used in the current page. Prefetch resources likely to be used for future navigations across multiple navigation boundaries.

Preload is an early fetch instruction to the browser to request a resource *needed* for a page (key scripts, Web Fonts, hero images).

Prefetch serves a slightly different use case — a future navigation by the user (e.g between views or pages) where fetched resources and requests need to persist across navigations. If Page A initiates a prefetch request for critical resources needed for Page B, the critical resource and navigation requests can be completed in parallel. If we used preload for this use case, it would be immediately cancelled on Page A's unload.

Between preload and prefetch, we get solutions for loading critical resources for the current navigation _or_ a future navigation.

What is the caching behavior for <link rel="preload"> and <link rel="prefetch">?

Chrome has four caches: the HTTP cache, memory cache, Service Worker cache & Push cache. Both preload and prefetched resources are stored in the **HTTP cache.**

When a resource is **preloaded or prefetched** is travels up from the net stack through to the HTTP cache and into the renderer's memory cache. If the resource can be cached (e.g there's a valid cache-control with valid max-age), it is stored in the HTTP cache and is available for **current and future sessions**. If the resource is **not cacheable**, it does not get stored in the HTTP cache. Instead, it goes up to the memory cache and stays there until it gets used.

How does Chrome's network prioritisation handle preload and prefetch?

Here's a break-down (courtesy of Pat Meenan) showing how different resources are prioritized in Blink as of Chrome 46 and beyond:

	Layout-blocking	Load in layout-blocking phase	Load one-at-a-time in layout-blocking phase		
Net Priority	**Highest**	**Medium**	**Low**	**Lowest**	**Idle**

Blink Priority	VeryHigh	High	Medium	Low	VeryLow
DevTools Priority	**Highest**	**High**	**Medium**	**Low**	**Lowest**
	Main Resource				
	CSS (match)				CSS (mismatch)
		Script (early** or not from preload scanner)	Script (late**)	Script (async)	
	Font	Font (preload)			
		Import			
		Image (in viewport)		Image	
				Media	
				SVG Document	
					Prefetch
		Preload*			
		XSL			
	XHR (sync)	XHR/fetch* (async)			
			Favicon		

* Preload using "as" or fetch using "type" use the priority of the type they are requesting. (e.g. preload as=style will use Highest priority). With no "as" they will behave like an XHR. ** "Early" is defined as being requested before any non-preloaded images have been requested ("late" is after). Thanks to Paul Irish for updating this table with the DevTools priorities mapping to the Net and Blink priorities.

Let's talk about this table for a moment.

Scripts get different priorities based on where they are in the document and whether they are async, defer or blocking:

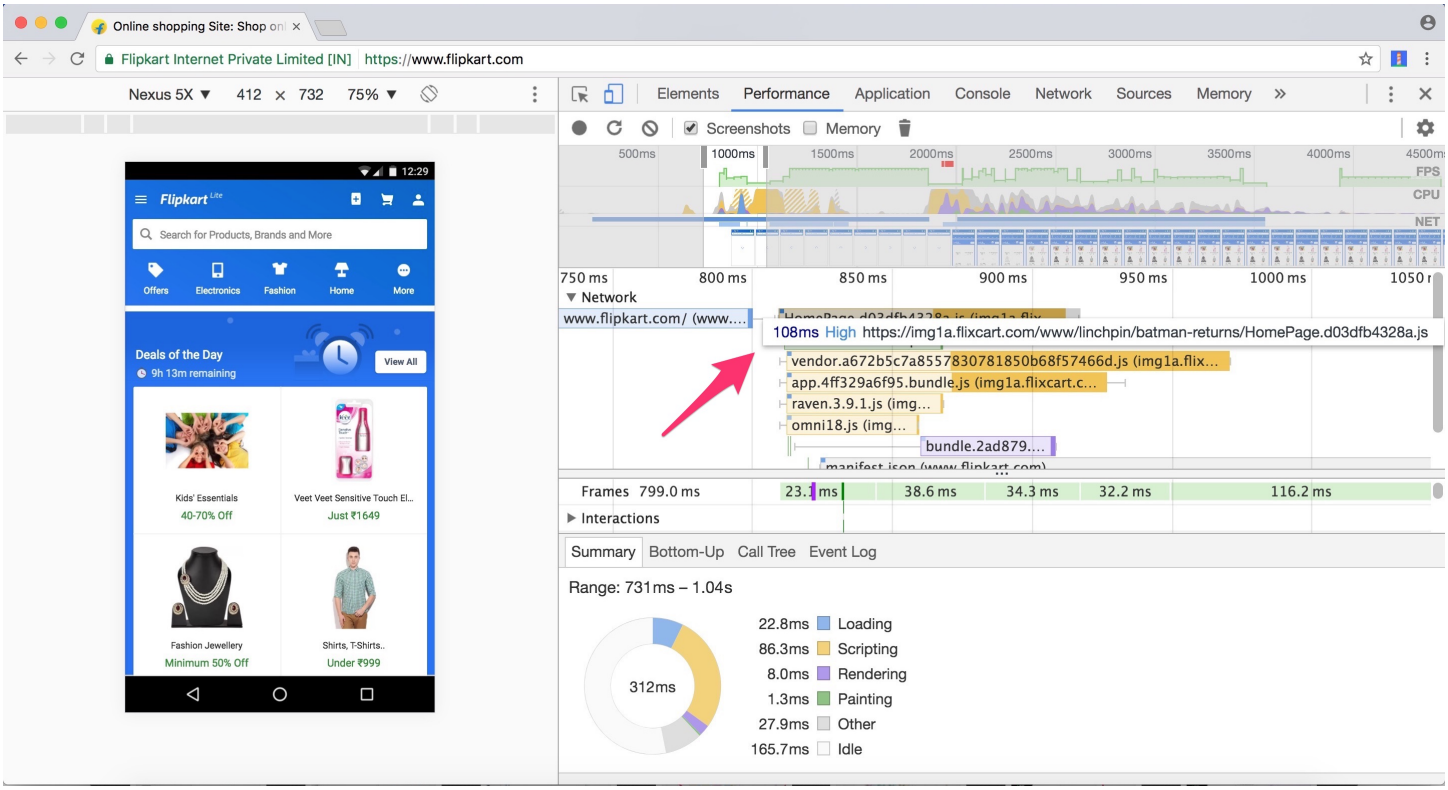
- Blocking scripts requested before the first image (an image early in the document) are Net:Medium
- Blocking scripts requested after the first image is fetched are Net:Low
- Async/defer/injected scripts (regardless of where they are in the document) are Net:Lowest

Images (that are visible and in the viewport) have a higher priority (Net:Medium) than those that are not in the viewport (Net:Lowest), so to some extent Chrome will do it's best to pseudo-lazy-load those images for you. Images start off with a lower priority and after layout is done and they are discovered to be in the viewport, will get a priority boost (but note that images already in flight when layout completes won't be reprioritized).

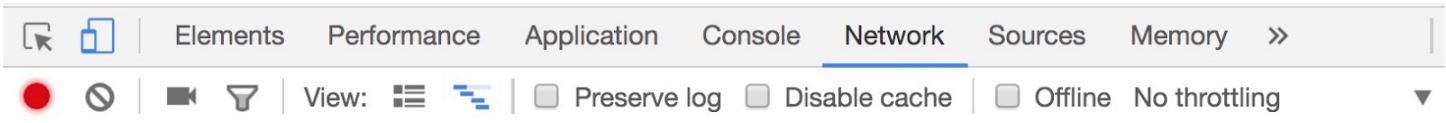
Preloaded resources using the "as" attribute will have the **same resource priority** as the **type** of resource they are requesting. For example, preload as="style" will get the highest priority while as="script" will get a low or medium priority. These resources are **also subject to the same CSP policies** (e.g script is subject to script-src).

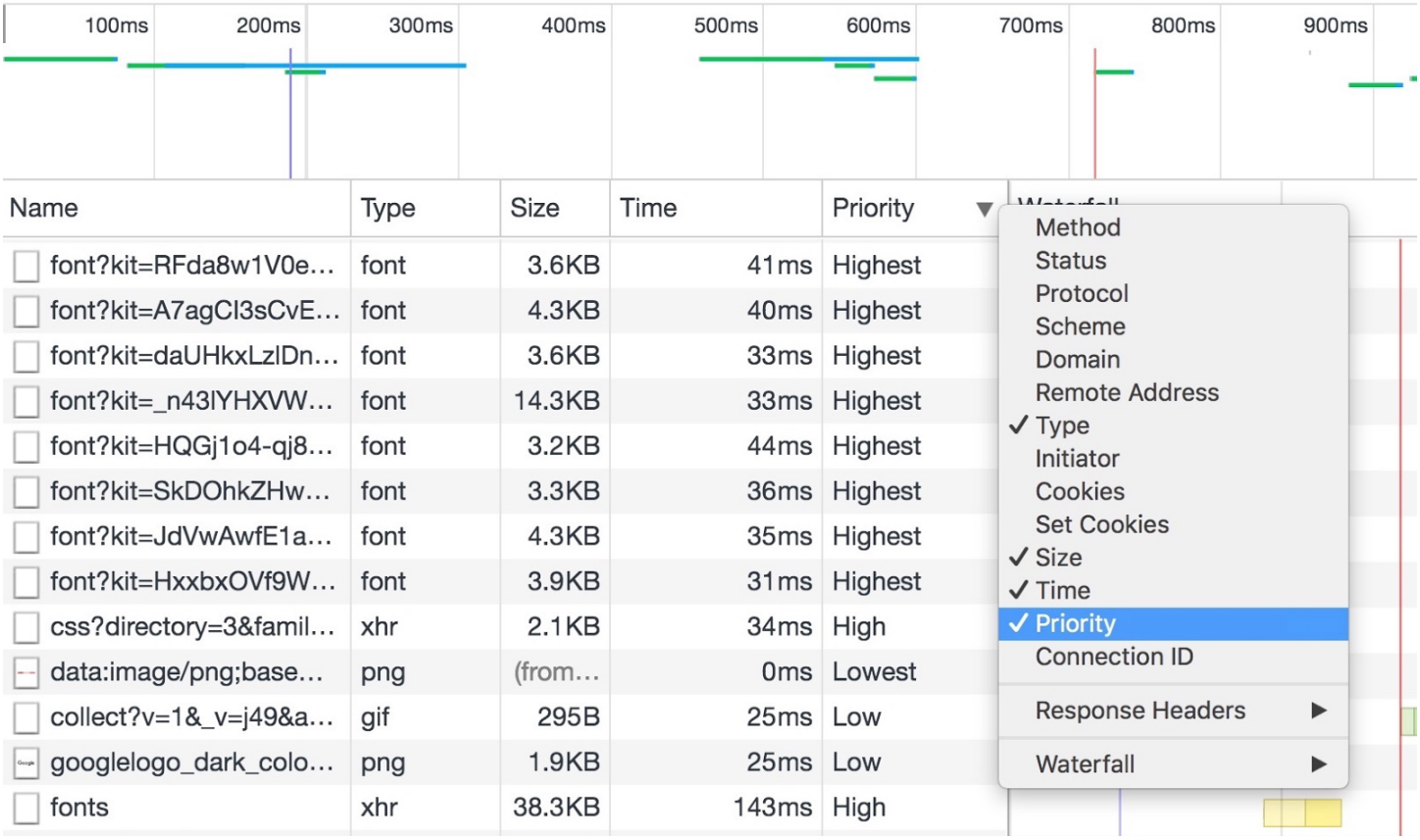
Preloaded resources without an "as" will otherwise be requested with the same priority as async XHR (so High).

If you're interested in understanding what priority a resource was loaded with, this information is exposed in DevTools via both the Network section of Timeline/Performance:



and in the Network panel behind the "Priority" column:





What happens when a page tries to preload a resource that has already been cached in the Service Worker cache, the HTTP cache or both?

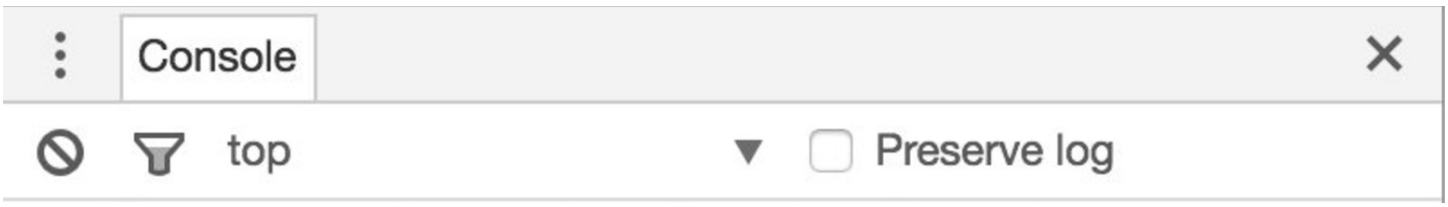
This is going to be a large “it depends” but generally, something good should almost always happen in this case — the resource won’t be refetched from the network unless it has expired from the HTTP cache or the Service Worker intentionally refetches it.

If the resource is in the HTTP cache (between the SW Cache & the network) then preload should get a cache hit from the same resource.

Are there risks with these primitives of wasting a user’s bandwidth?

With “preload” or “prefetch”, you’re running some risk of wasting a user’s bandwidth, especially if the resource is not cacheable.

Unused preloads trigger a console warning in Chrome, ~3 seconds after *onload:*



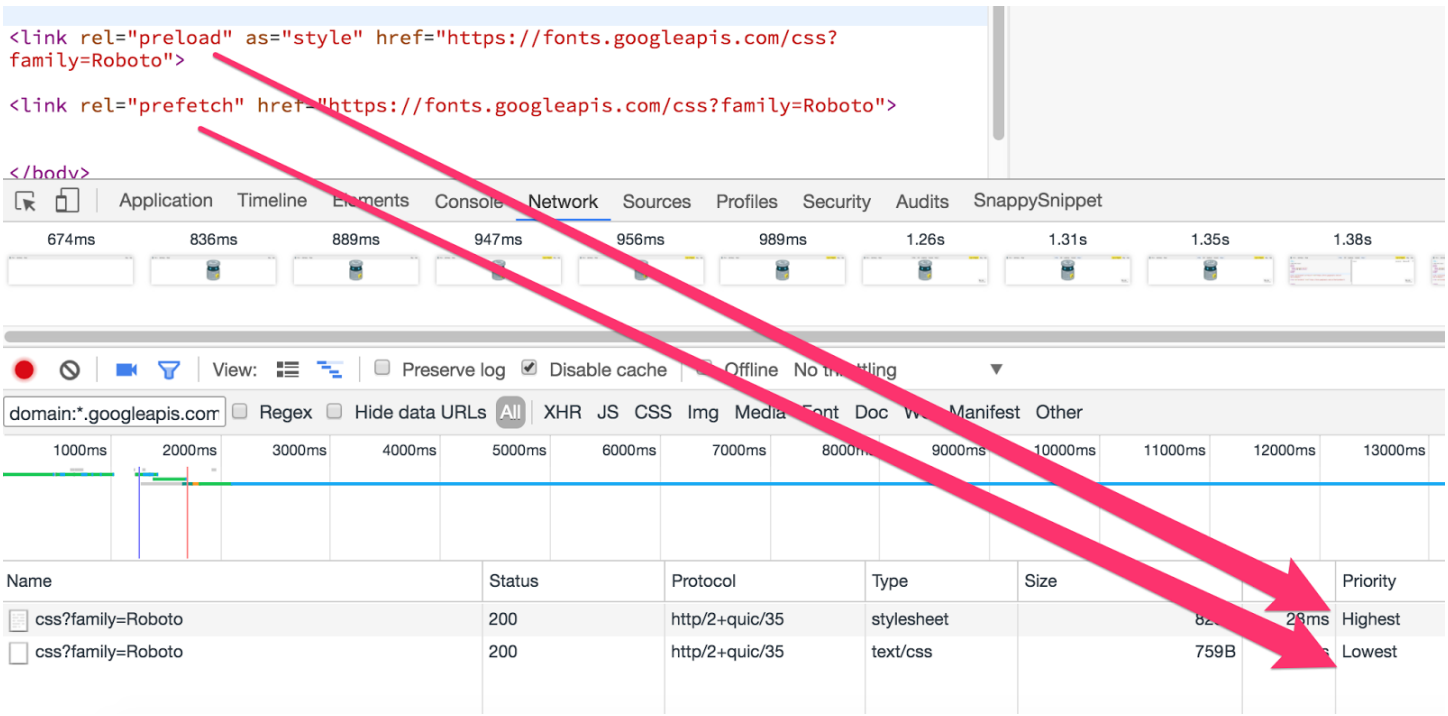
```
⚠ The resource http://localhost:3000/api/1tcvlsq/a   a:1
  was preloaded using link preload but not used within
  a few seconds from the window's load event. Please
  make sure it wasn't preloaded for nothing.
```

The reason for this warning is you're probably using preload to try warming the cache for other resources you need to improve performance but if these preloaded resources aren't being used, you're doing extra work for no reason. On mobile, this sums up to wasting a user's data plans, so be mindful of what you're preloading.

What can cause double fetches?

Preload and prefetch are blunt tools and it isn't hard to find yourself double-fetching if you aren't careful.

Don't use "prefetch" as a fallback for "preload". They're again, used for different purposes and often end up causing double fetches while this probably isn't your intention. Use preload if it's supported for warming the cache for current sessions otherwise prefetch for future sessions. Don't use one in place of the other.



Don't rely on fetch() working with "preload"… just yet. In Chrome if you try to use preload with the fetch() API you will end up triggering a double download. This doesn't currently occur with XHR and we have an open bug to try addressing it.

Supply an "as" when preloading or you'll negate any benefits!

If you don't supply a valid "as" when specifying what to preload, for example, scripts, you will end up fetching twice.

Preloaded fonts without crossorigin will double fetch! Ensure you're adding a crossorigin attribute when fetching fonts using preload otherwise they will be double downloaded. They're requested using anonymous mode CORS. This advice applies even if fonts are on the same origin as the page. This is applicable to other anonymous fetches too (e.g XHR by default).

Resources with an integrity attribute can't reuse preloaded resources (for now) and can also cause double fetches. The `integrity` attribute for link elements has not yet been implemented and there's an open spec issue about it. This means the presence of any integrity metadata will currently discard preloaded resources. In the wild, it can also result in duplicate requests where you have to make a trade-off between security and performance.

Finally, although it won't cause double fetches, this is generally good advice:

Don't try preloading absolutely everything! Instead, select specific late discovered resources that you want to load earlier and use preload to tell the browser about them.

Should I just preload all the assets that my page requests in the head? Is there a recommended limit like "only preload ~6 things"?

This is a good example of **Tools, not rules.** How much you preload may well factor in how much network contention you're going to have with other resources also being loaded on your page, your user's available bandwidth and other network conditions.

Preload resources that are likely to be discovered late in your page, but are otherwise important to fetch as early as possible. With scripts, preloading your key bundles is good as it separates fetching from execution in a way that just using say, <script async> wouldn't as it blocks the window's onload event. You can preload images, styles, fonts, media. Most things — what's important is that you're in better control of early-fetching what you as a page author knows is definitely needed by your page sooner rather than later.

Does prefetch have any magical properties you should be aware of? Well, yes.

In Chrome, if a user navigates away from a page while prefetch requests for other pages are still in flight, these requests will not get terminated.

Furthermore, prefetch requests are maintained in the unspecified net-stack cache for at least 5 minutes regardless of the cachability of the resource.

I'm using a custom "preload" implementation written in JS. How does this differ from rel="preload" or Preload headers?

Preload decouples fetching a resource from JS processing and execution. As such, preloads declared in markup are optimized in Chrome by the preload scanner. This means that in many cases, the preload will be fetched (with the indicated priority) before the HTML parser has even reached the tag. This makes it a lot more powerful than a custom preload implementation.

Wait. Shouldn't we be using HTTP/2 Server Push instead of Preload?

Use Push when you know the precise loading order for resources and have a service worker to intercept requests that would cause cached resources to be pushed again. Use preload to move the start download time of an asset closer to the initial request — it's useful for both first and third-party resources.

Again, this is going to be an "it depends". Let's imagine we're working on a cart for the Google Play store. For a given request to play.google.com/cart:

Using Preload to load key modules for the page requires the browser to wait for the play.google.com/cart payload in order for the preload scanner to detect dependencies, but after this contains sufficient information to saturate a network pipe with requests for the site's assets. This might not be the most optimal at cold-boot but is very cache and bandwidth friendly for subsequent requests.

Using H/2 Server Push, we can saturate the network pipe right away on the request for play.google.com/cart but can waste bandwidth if the resources being pushed are

already in the HTTP or Service Worker cache. There are always going to be trade-offs for these two approaches.

Although Push is invaluable, it doesn't enable all the same use-cases as Preload does.

Preload has the benefit of decoupling download from execution. Thanks to support for document onload events you can control scripting if, how and when a resource gets applied. This can be powerful for say, fetching JS bundles and executing them in idle blocks or fetching CSS and applying them at the right point in time.

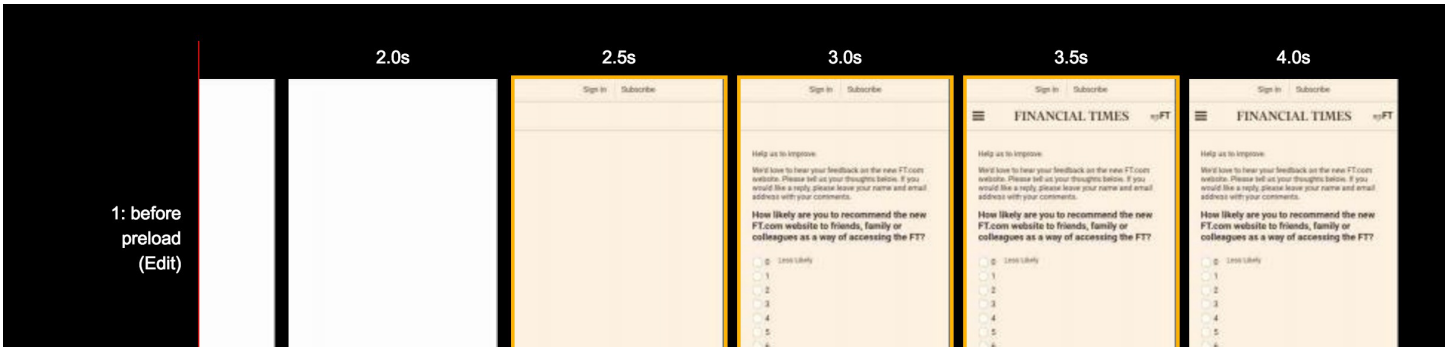
Push can't be used by third-party hosted content. By sending resources down immediately, it also effectively short-circuits the browser's own resource prioritization logic. In cases where you know exactly what you're doing, this can yield performance wins, but in cases where you don't you could actually harm performance significantly.

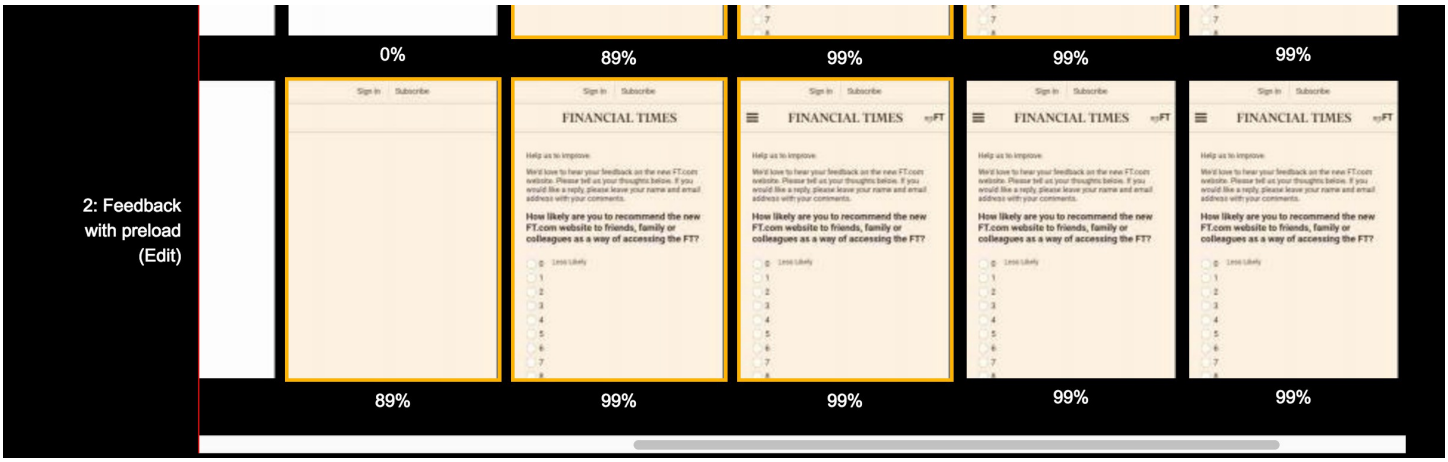
What is the Link preload header? How does it compare to the preload link tag? And how does it relate to HTTP/2 Server Push?

As with other types of links, a preload link can be specified using either an HTML tag or an HTTP header (a Link preload header). In either case, a preload link directs the browser to begin loading a resource into the memory cache, indicating that the page expects with high confidence to use the resource and doesn't want to wait for the preload scanner or the parser to discover it.

How does a preload tag and Link header differ? <link rel=preload> tags are initiated after the browser receives a document and the preload scanner discovers these tags. Preloading via headers _may_ offer a very negligible improvement, but require the document to be committed before processing happens.

When the Financial Times introduced a Link preload header to their site, they **shaved 1 second off the time it took to display the masthead image:**





Bottom: with preload, Top: without. **Comparison for Moto G4 over 3G:** Before: https://www.webpagetest.org/result/170319_Z2_GFR/, After: https://www.webpagetest.org/result/170319_R8_G4Q/

You can provide preload links in either form, but there is one important difference you should understand: as allowed by the spec, many servers initiate an HTTP/2 Server Push when they encounter a preload link in HTTP header form. The performance implications of H/2 Server Push are different from those of preloading (see below), so you should make sure you don't unintentionally trigger pushes.

You can avoid unwanted pushes by using preload link tags instead of headers, or by including the 'nopush' attribute in your headers.

How can I feature detect support for link rel=preload?

Feature detecting for <link rel="preload">can be accomplished using the following snippet:

```
const preloadSupported = () => {
  const link = document.createElement('link');
  const relList = link.relList;
  if (!relList || !relList.supports)
    return false;
  return relList.supports('preload');
};
```

The FilamentGroup also have a preload check they use as part of their async CSS loading library, loadCSS.

Can you immediately apply preloaded CSS stylesheets?

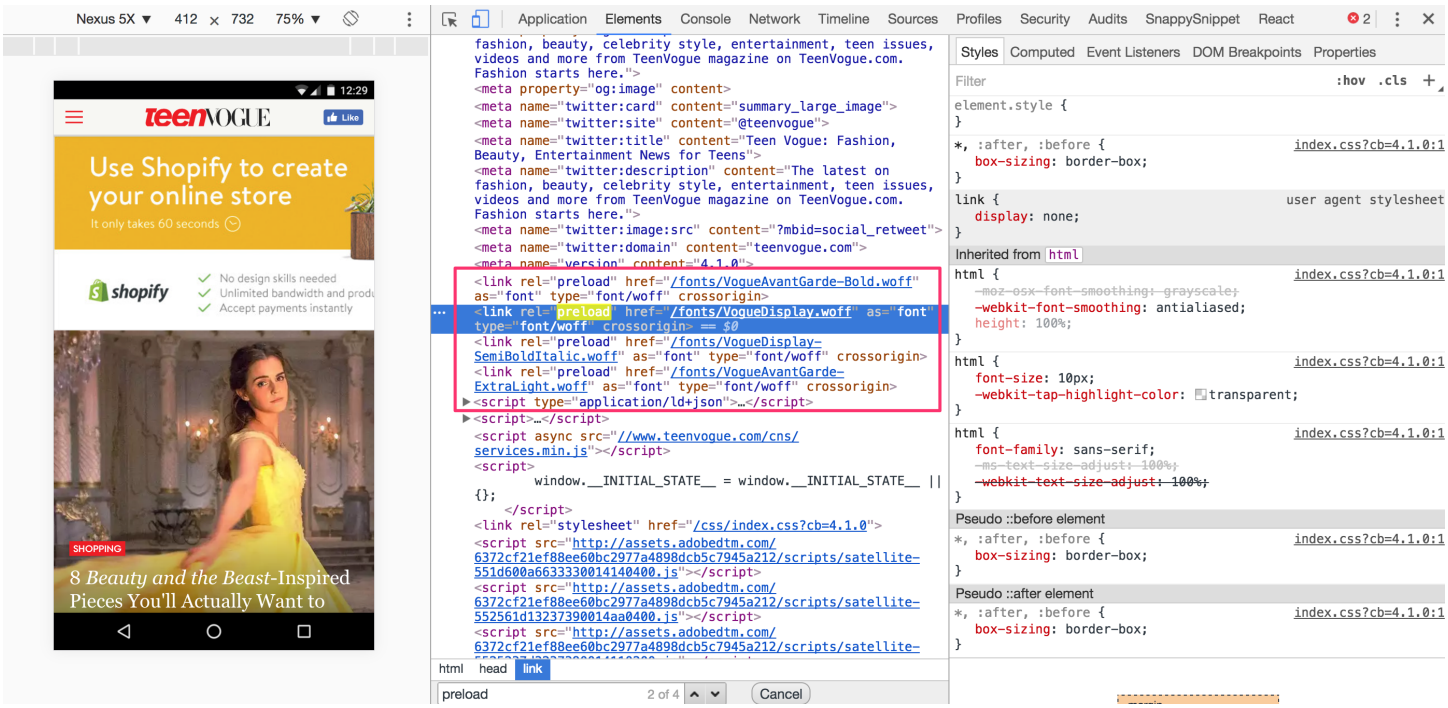
Absolutely. Preload support markup based asynchronous loading. Stylesheets loaded using <link rel=”preload”> can be immediately applied to the current document using the `onload` event as follows:

```
<link rel="preload" href="style.css" onload="this.rel=stylesheet">
```

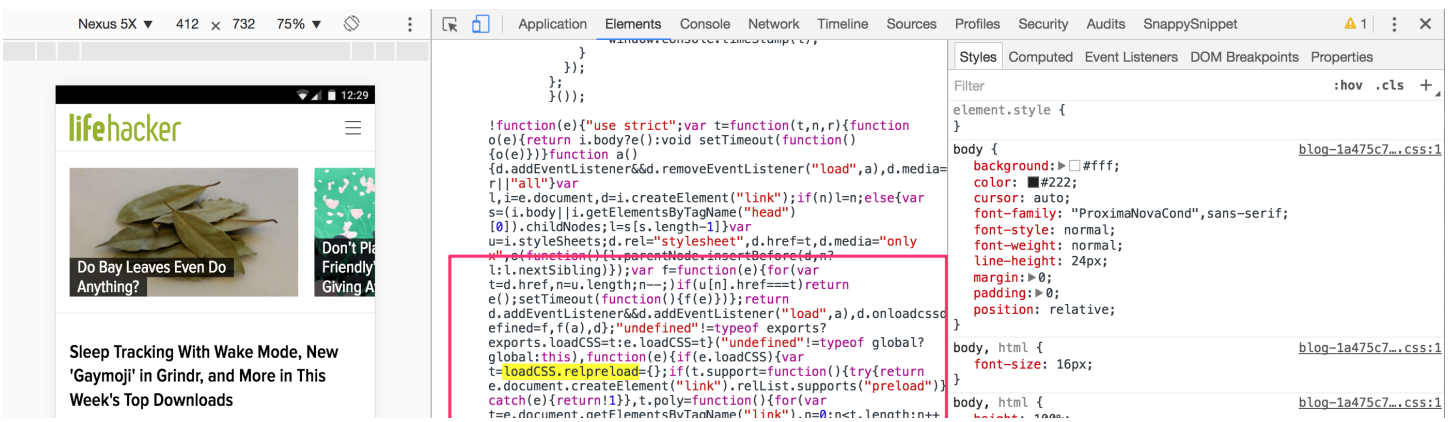
For more examples like this, see *Use Cases* in this great Yoav Weiss deck.

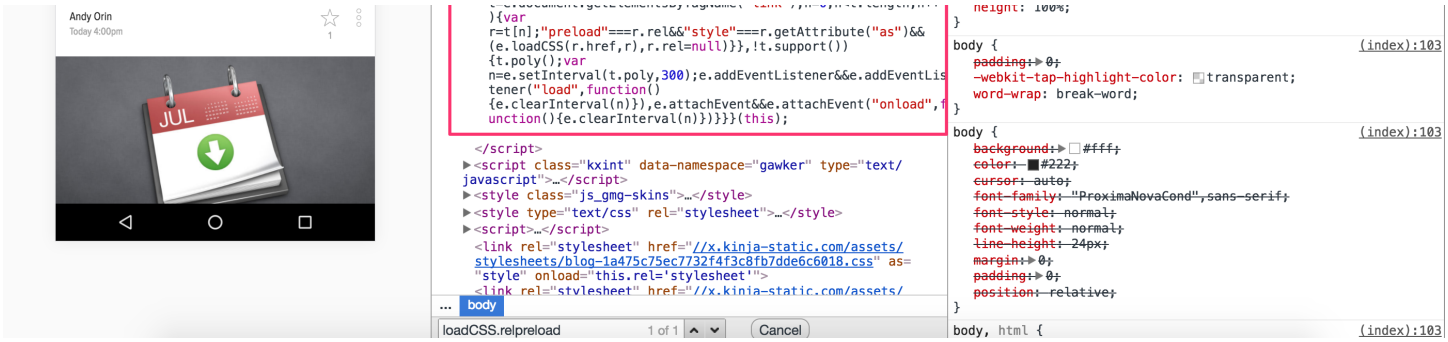
What else is Preload being used for in the wild?

According to the HTTPArchive, most sites using <link rel=”preload”> use it to preload Web Fonts, including Teen Vogue and as mentioned earlier, Shopify:

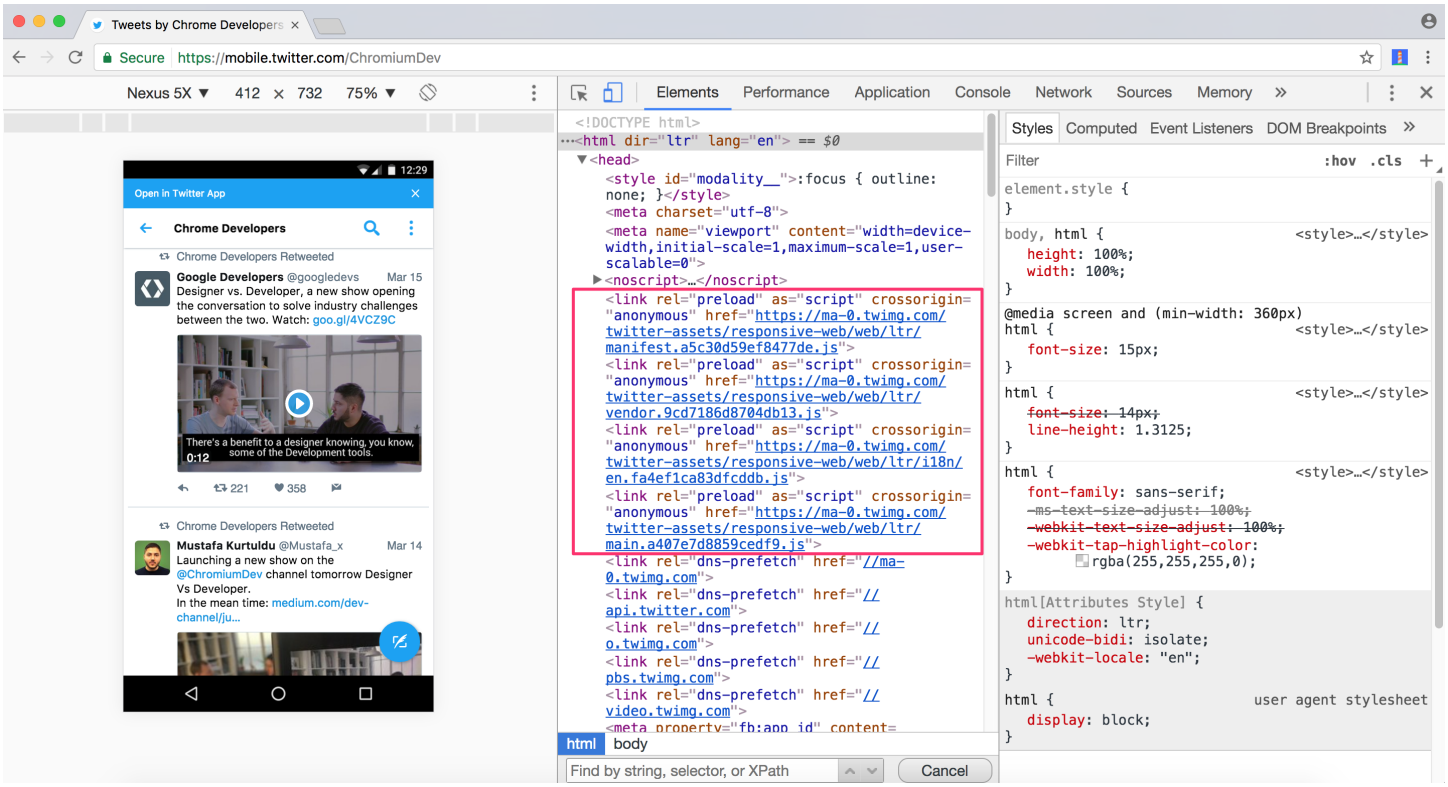


While other popular sites like LifeHacker and JCPenny use it to asynchronously load CSS (via the FilamentGroup's loadCSS):





And then there are a growing breed of Progressive Web Apps (like Twitter.com mobile, Flipkart and Housing) using it to preload scripts that are needed for the current navigation using patterns like PRPL:



The basic idea there is to maintain artifacts at high-granularity (as opposed to monolithic bundles) so any facet of the app can on demand load it's dependencies or preload those that are likely to be needed next to warm up the cache.

What is the current browser support for Preload and Prefetch?

<link rel="preload"> is available to ~50% of the global population according to CanIUse and is implemented in the Safari Tech Preview. <link rel="prefetch"> is available to 71% of global users.

Further insights you may find helpful:

- Yoav Weiss landed a recent change in Chrome that avoids preload contending with CSS & blocking scripts.
- He also recently split the ability to preload media into three distinct types: video, audio and track.
- Domenic Denicola is exploring a spec change to add support for preloading ES6 Modules.
- Yoav also recently shipped support for Link header support for “prefetch” allowing easier additional of resource hints needed for the next navigation.

Further reading on these loading primitives:

- Preload — what is it good for? — Yoav Weiss
- A <link rel=”preload”> study by the Chrome Data Saver team
- Planning for performance — Sam Saccone
- Webpack plugin for auto-wiring up <link rel=”preload”>
- What is preload, prefetch and preconnect? — KeyCDN
- Web Fonts preloaded by Zach Leat
- HTTP Caching: cache-control by Ilya Grigorik

With thanks to @ShopifyEng, @AdityaPunjani from Flipkart, @HousingEngg, @adgad and @wheresrhys at the FT and @__lakshya from Treebo for sharing their before/after preload stats.

With many thanks for their technical reviews & suggestions: Ilya Grigorik, Gray Norton, Yoav Weiss, Pat Meenan, Kenji Baheux, Surma, Sam Saccone, Charles Harrison, Paul Irish, Matt Gaunt, Dru Knox, Scott Jehl.