最近の Web パフォーマンス改善について知っておきたいコト

2017/09/24　HTML5 Conference 2017

@ahomu - Ayumu Sato

どうもこんにちは
@ahomuです

@ahomu

- 佐藤 歩（さとう あゆむ）
- 名古屋と渋谷を往復中の Web ディベロッパー
- 最近の関心は #webperf と #a11y
- CyberAgent, Inc.
  - 技術本部 Web Initiative Center / メディア統括 技術戦略室

パフォーマンスは重要？

53% of visits are likely to be **abandoned** if pages take **longer than 3 seconds** to load.

The need for mobile speed: How mobile latency impacts publisher revenue
https://www.thinkwithgoogle.com/intl/en-154/insights-inspiration/research-data/need-mobile-speed-how-mobile-latency-impacts-publisher-revenue/

AUG 10
2017

Webパフォーマンスとプロダクト KPIの相関を可視化する話

WRITER: 1000CH

エンジニア

B! ブックマーク

KPI 云々は弊社のブログをご覧ください

https://developers.cyberagent.co.jp/blog/archives/9540/ written by **1000ch**

技術本部 Web Initiative Center で Web プロダクトの品質改善や組織の技術推進に取り組んでいます、 @1000ch です。その活動のひとつに、直帰率や PV/Session といったプロダクトの KPI を上げることを目的とした Web パフォーマンスの向上があるので、今日はその取り組みについて書きます。

旧来の Web パフォーマンスの評価指標

これまでは `DOMContentLoaded` イベントや `load` イベントの値が Web ページのパフォーマンスの良し悪しとして扱われることが多かったと思います。しかしアーキテクチャの複雑化の一途をたどる Web ページのパフォーマンスを測る上では、もはや適切な指標とは言えません。

CATEGORY
エンジニア
クリエイター
ディレクター
テクニカルクリエイター
LINKS
Technical Creator Hub
執筆者一覧 →

アジェンダ

- パフォーマンスに関する基礎的な考え方のおさらい
  - パフォーマンスの基本的な「考え方、捉え方」
  - パフォーマンスとクライアント環境の「多様性」
- 近年、知っておきたいパフォーマンス関連のはなし
  - ページロード - コンテンツを早く提供する最適化
  - ランタイム - 滑らかに動く UI と快適な操作感

本日の注意

- デモとかライブコーディングとかはありません
- 情報量優先でスライドを淡々とめくっていきます
- 新しい技術も紹介しますが対応ブラウザは各自で調べて一喜一憂しましょう

パフォーマンスの
基本的な「考え方、捉え方」

観点は大きく分けて2つ

ページロード

ランタイム

ページロード 

- **ナビゲーションの開始からページが表示されるまでの速度**
  - クリティカルレンダリングパスの最適化
  - 配信リソース(画像など)の数とファイルサイズの最適化
  - HTML 返却(≒サーバーサイド処理)の高速化 etc...

RENDERING

Notes

Start Render

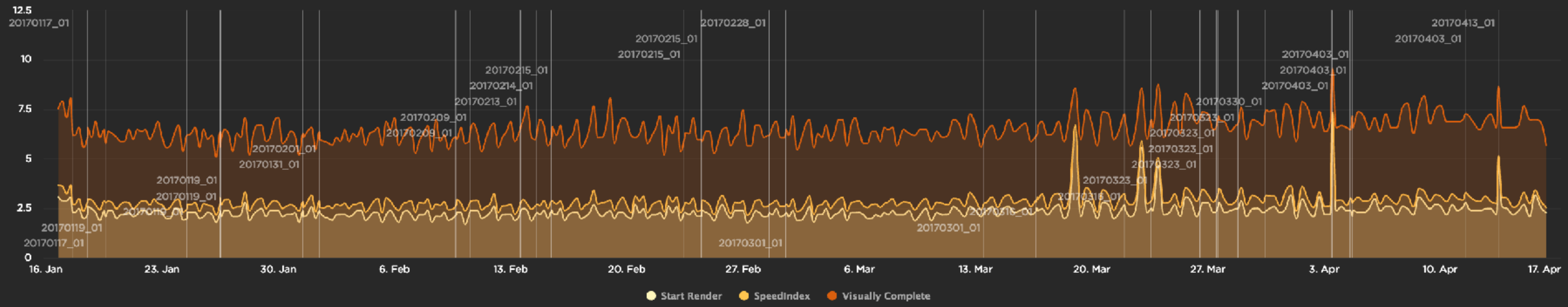
2.29s

SpeedIndex

2.8s

Visually Complete

6.41s



PAGE LOAD TIMES

Notes

Backend

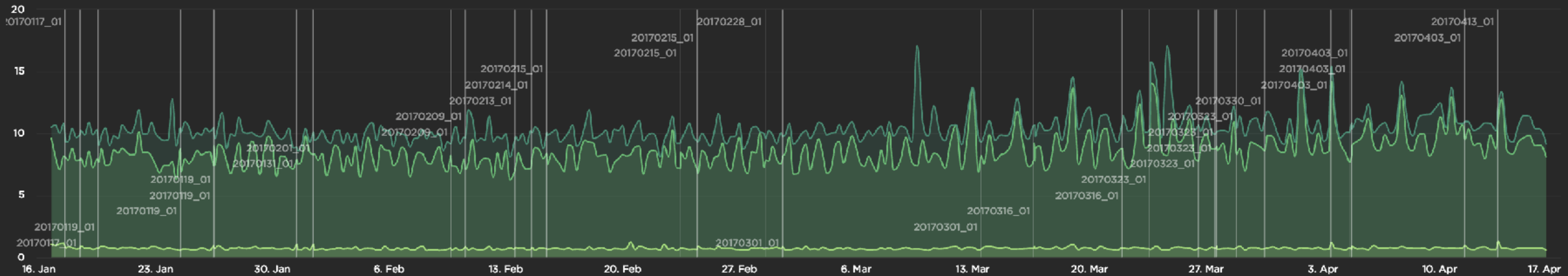
0.71s

Page Load

8.38s

Fully Loaded

10.12s



SPEEDINDEX

FRESH!

2.9s

#2 38% slower

YouTube

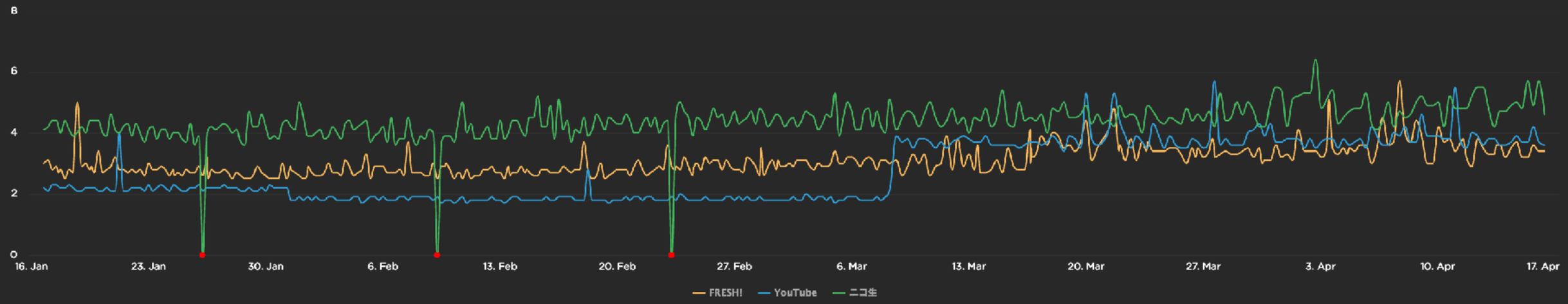
2.1s

#1

ニコ生

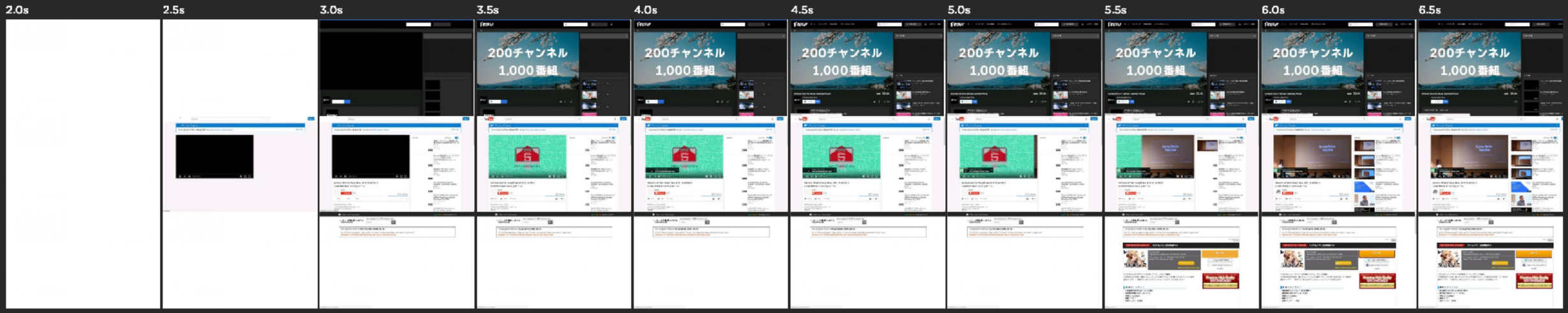
4.4s

#3 110% slower



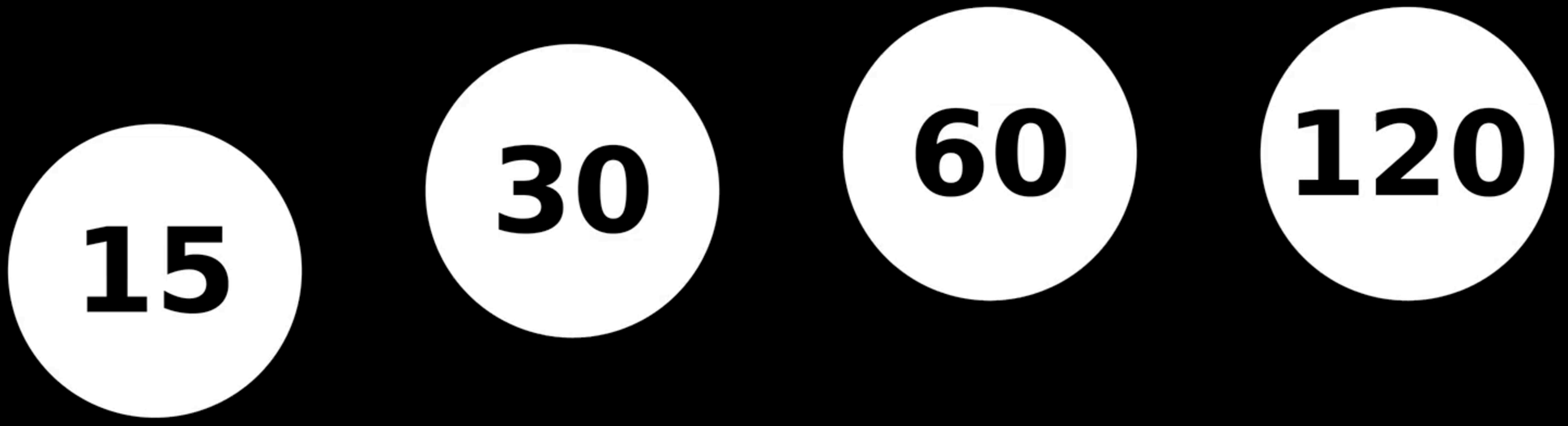
FILMSTRIP

OPTIONS

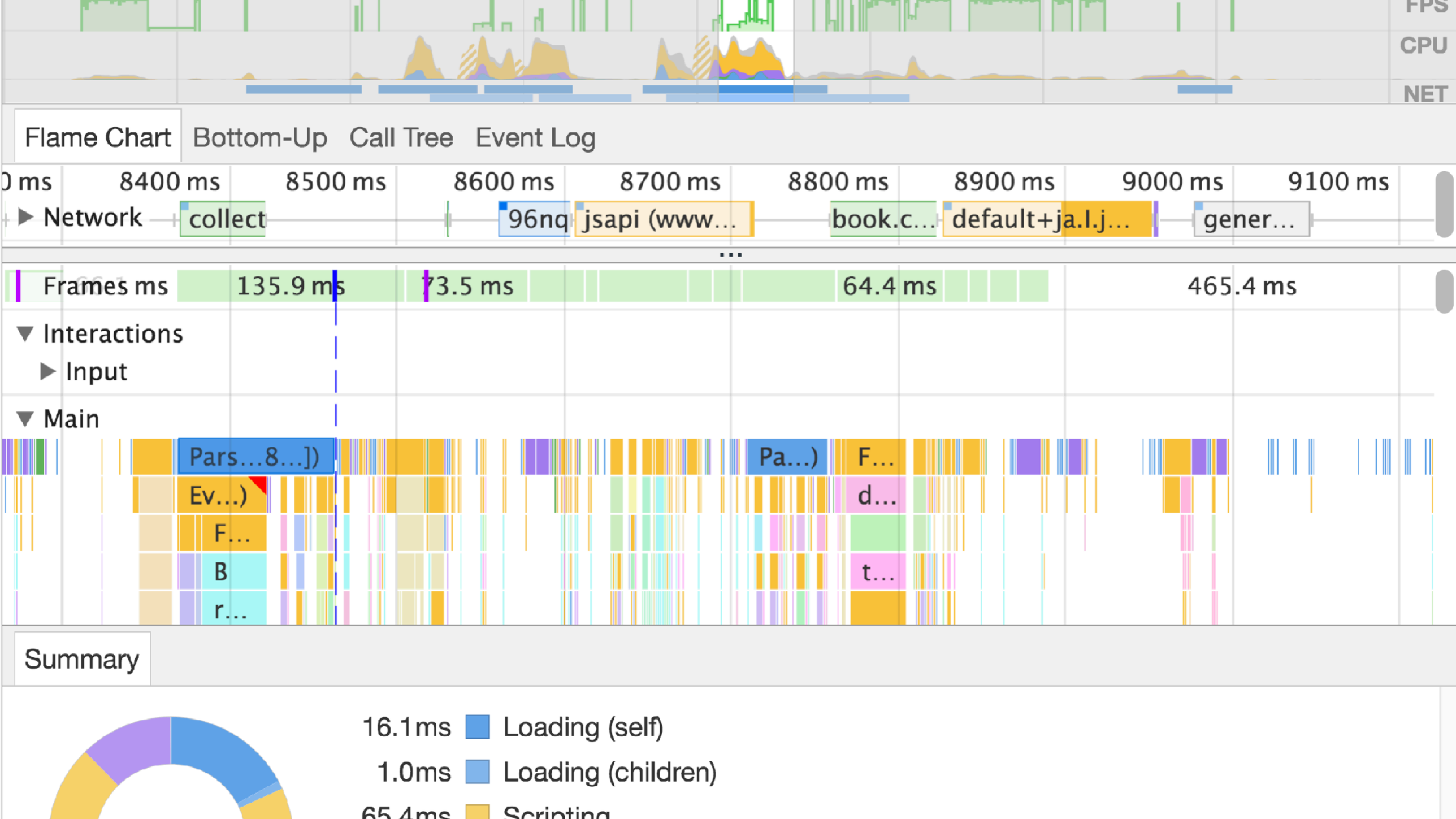


ランタイム 🔁

- **ページが表示されたあとの UI の応答速度**
  - FPS (1秒あたりの画面の更新回数) の最大化 or 反応の高速化
  - グラフィック的なレンダリング負荷の軽量化
  - 画面更新をさまたげるスクリプト処理の除去 etc...



via. https://www.youtube.com/watch?v=pfiHFqnPLZ4


FPS
CPU
NET
Flame Chart
Bottom-Up
Call Tree
Event Log
0 ms
8400 ms
8500 ms
8600 ms
8700 ms
8800 ms
8900 ms
9000 ms
9100 ms
Network
collect
96nq
jsapi (www…
book.c…
default+ja.I.j…
gener…
Frames ms
135.9 ms
73.5 ms
64.4 ms
465.4 ms
Interactions
Input
Main
Pars…8…])
Pa…)
F…
Ev…)
d…
F…
B
t…
r…
Summary
16.1ms
Loading (self)
1.0ms
Loading (children)
65.4ms
Scripting

パフォーマンスと
クライアント環境の「多様性」

ユーザーA

2年前のスマホ
地下鉄で移動中
SNSタイムラインから

ユーザーB

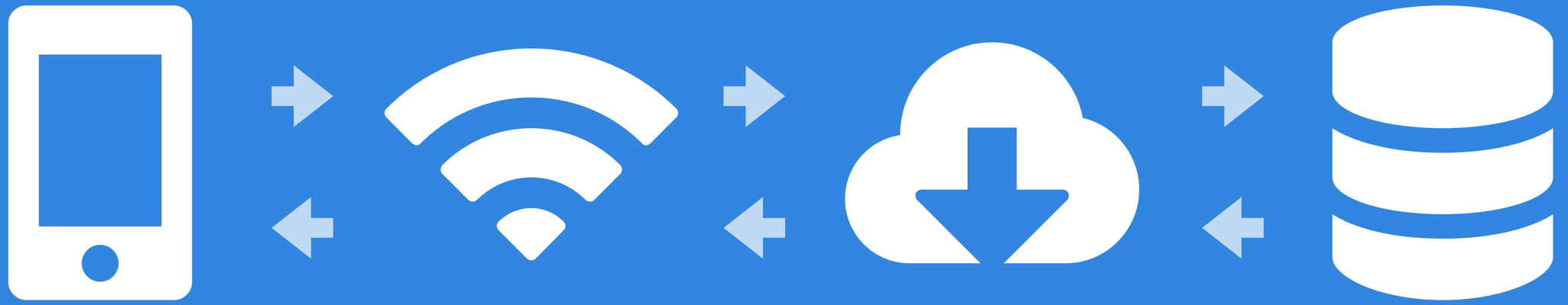
最新スマホ
地方に帰省中
ブックマークから

ユーザーC

1年前のスマホ
飲食店の絶望的Wifi
調べ物で検索から

クライアント環境の変数 (ほんの一部)

- デバイス（処理性能、スクリーンサイズ、操作方法）
- ブラウザ、OS（実行性能、ユーザーが追加した拡張機能）
- ネットワーク（オフィス、自宅、公衆Wifi、モバイル）
- ユーザー（年齢や個人差による認知特性の差異）



Client

スペックによる処理遅延

Connection

接続のレイテンシー

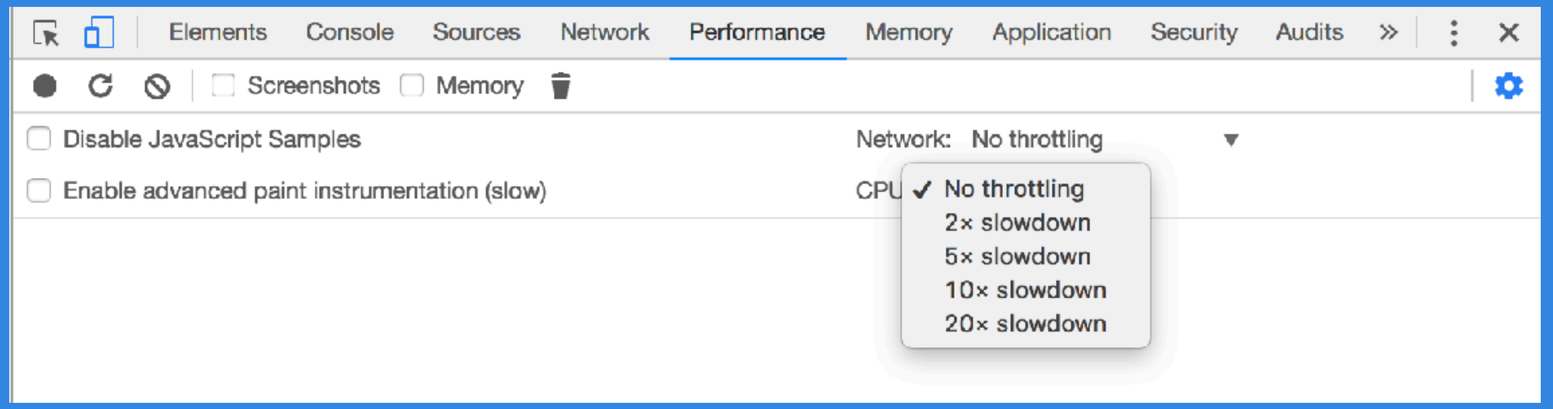
Network

機器間のオーバーヘッド

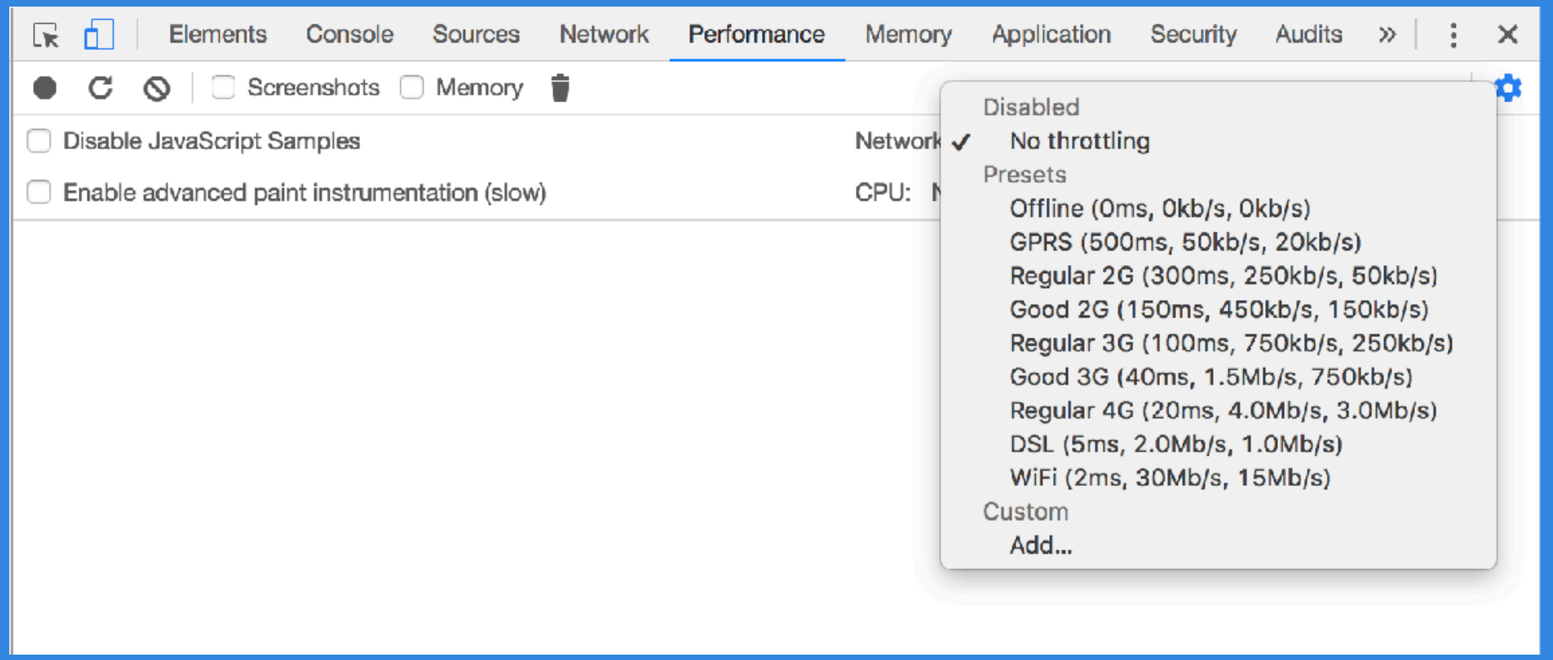
Server

データベースなどの遅延

CPU のスロットリング（処理性能のエミュレート）



ネットワークのスロットリング（帯域性能のエミュレート）



あらゆる多様性の中に
遅い、使いづらい、使えない が潜む

どれくらいなら適切な応答速度と言えるのか？

ランタイム

UI のレスポンス
100ms

アニメーション1f
10ms (16ms)

アイドル中の1処理
50ms

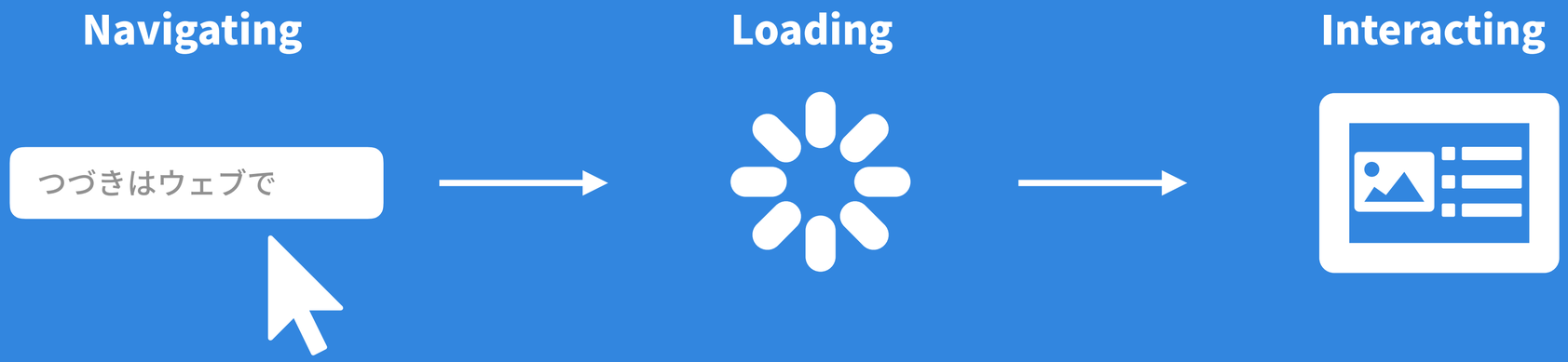
ページロード
1,000ms

via. https://developers.google.com/web/fundamentals/performance/rail

ページロード

コンテンツを早く提供する最適化

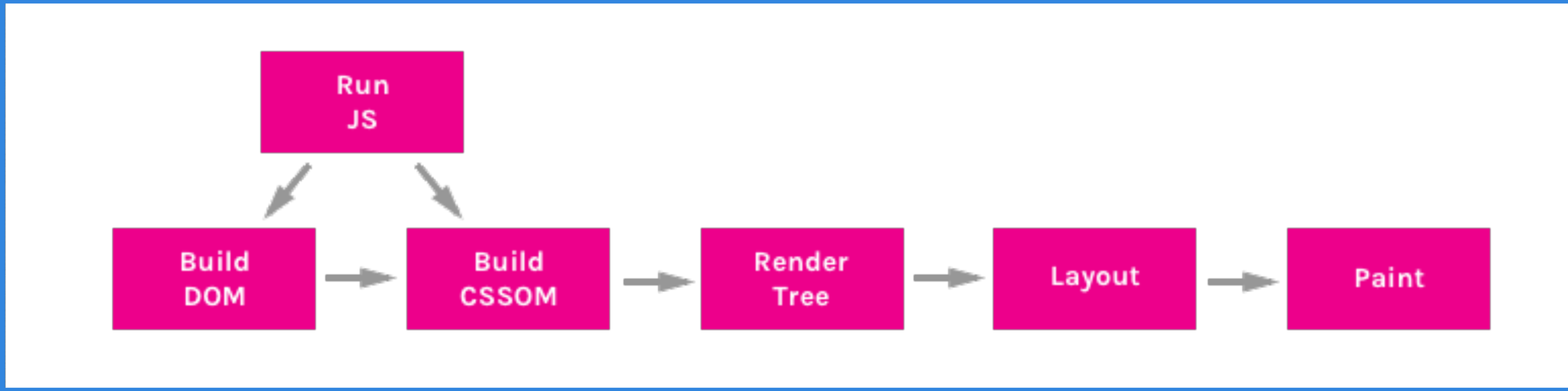
ページロードとはすなわち



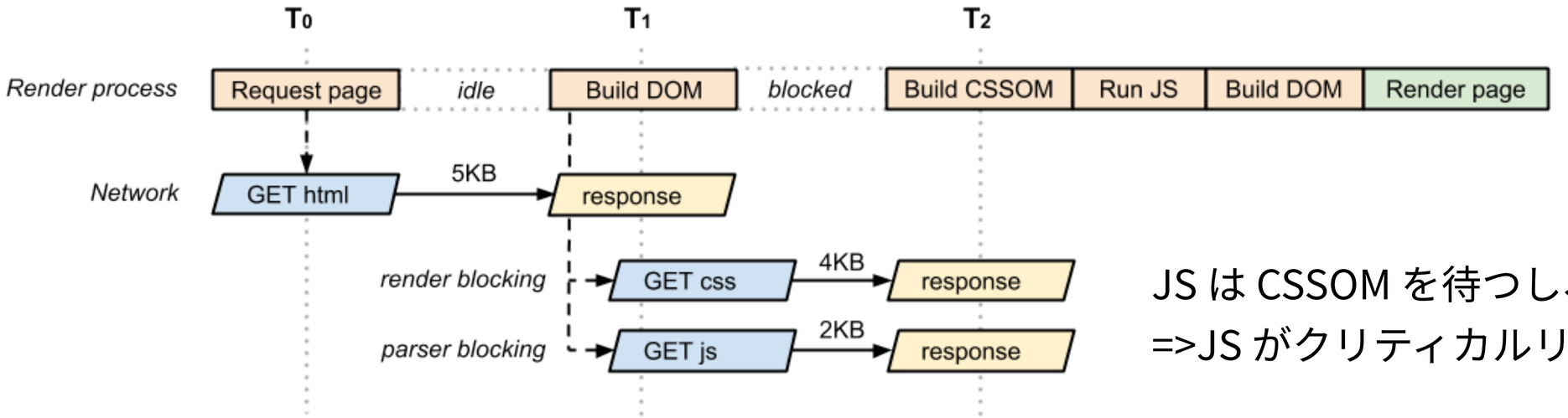
- ナビゲーションを開始して
- リソースをダウンロードのち評価して
- Web ページを実行する（ユーザーが利用可能になる）こと

いわゆるクリティカルレンダリングパス

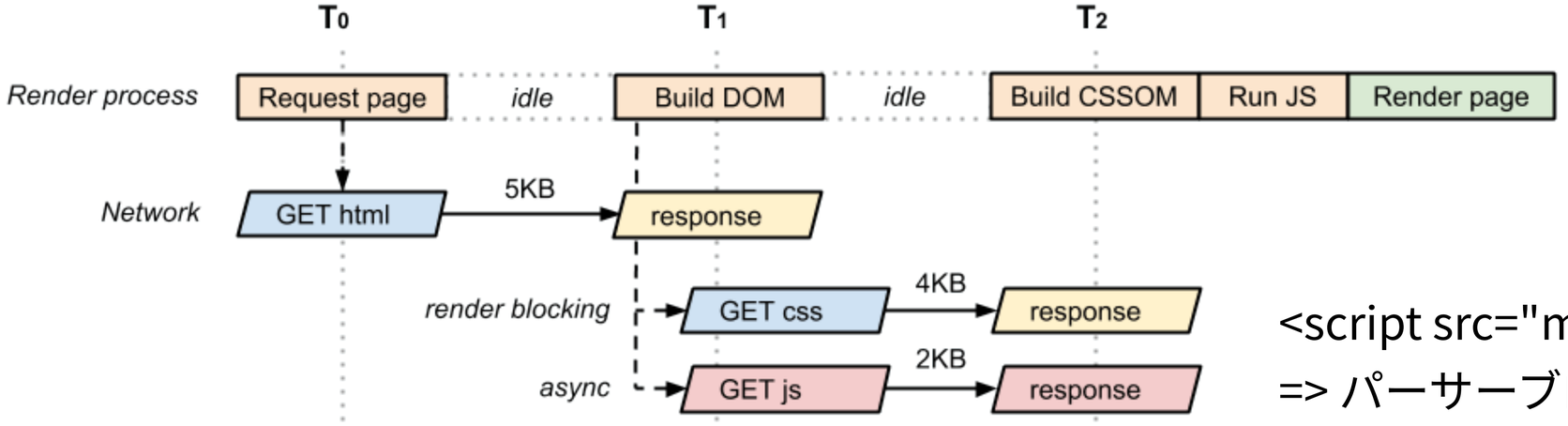
- DOM + CSSOM = Render Tree
- Download resources ... Render pixels … Transfer bitmaps
- RAIL の Load だが… 1,000ms にはそうそうおさまらない 😇



via. https://bitsofco.de/understanding-the-critical-rendering-path/



JS は CSSOM を待つし、DOM にも影響する
=>JS がクリティカルリソースになっている



<script src="main.js" async> で非同期ロード
=> パーサーブロックせずクリティカルでなくなる

via. https://developers.google.com/web/fundamentals/performance/critical-rendering-path/analyzing-crp

クリティカル パスの長さ

- HTML は DOM のルートなのでブロックせざるをえない
- CSS は CSSOM をブロックする
- JavaScript は DOM も CSSOM もブロックする
- 各種の遅延ロードはクリティカルパスの短縮にも役立つ

クリティカル リソースの数とサイズ

- 遅延できる <script> は async または defer する
- 必要の無いサードパーティスクリプトを読み込まない
- 初期表示に必要な CSS のみを埋め込む（ややハードル高い）
- 圧縮およびコードの最小化ができるものはしておく
- 日本語 Web フォントはテキスト描画を止める最終兵器

ページロードはネットワーク処理が重要？

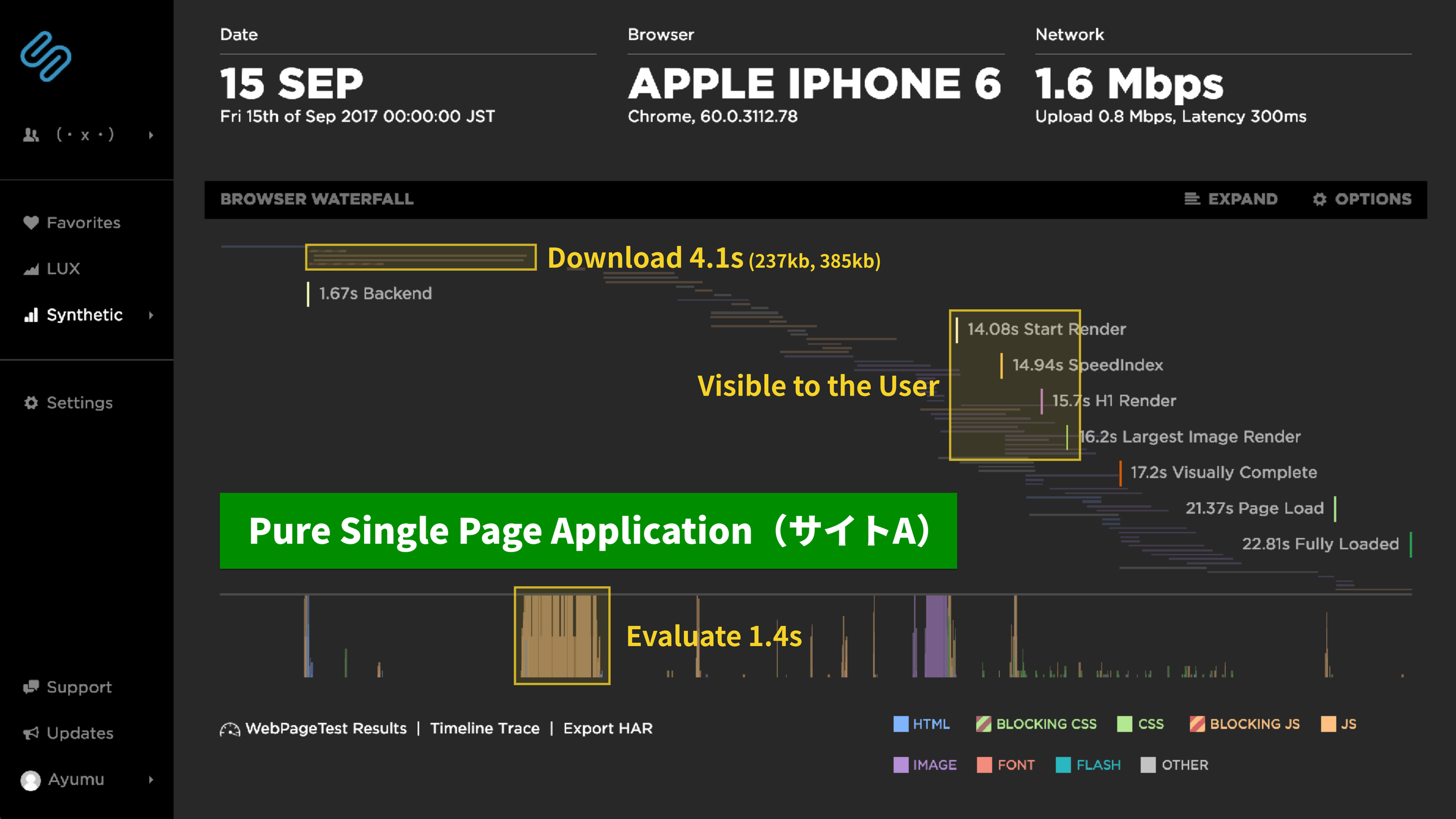
- リソースが速く届くか、遅く届くかという観点
- リソースの数と、サイズと、転送距離を短くする
- もちろん重要ではあるのだが...

ネットワークの環境と開発者の意識

- 楽観しきれない日本の通信網（下り平均 20.2 Mbps, モバイル 15.6 Mbps）
  - via. Akamai Q1 2017 State of the Internet Connectivity Report
- 開発者の意識もそれなりに高まってきている
- 適切な画像形式で最適化とか、テキストの最小化 + gzip とかみんなやるよね
- デカい画像、無駄なリソース、ふつーに考えればダメだって分かる

いま敢えて取り上げたいのはデカい JavaScript

- クライアントサイドにロジックが寄ってきている弊害
- npm などで依存パッケージの追加も容易になっている
- Web アプリはもちろんのこと、Web サイトもプラグインまみれで案外ヤバい
- そして案外、ファイルサイズを気にする人が少ない (要出典)


Date
15 SEP
Fri 15th of Sep 2017 00:00:00 JST
Browser
APPLE IPHONE 6
Chrome, 60.0.3112.78
Network
1.6 Mbps
Upload 0.8 Mbps, Latency 300ms
(• x •)
Favorites
LUX
Synthetic
Settings
Support
Updates
Ayumu
BROWSER WATERFALL
EXPAND
OPTIONS
Download 4.1s (237kb, 385kb)
1.67s Backend
Visible to the User
14.08s Start Render
14.94s SpeedIndex
15.7s H1 Render
16.2s Largest Image Render
17.2s Visually Complete
21.37s Page Load
22.81s Fully Loaded
Pure Single Page Application（サイトA）
Evaluate 1.4s
WebPageTest Results | Timeline Trace | Export HAR
HTML
BLOCKING CSS
CSS
BLOCKING JS
JS
IMAGE
FONT
FLASH
OTHER

(・ x ・)

Favorites

LUX

Synthetic

Settings

Support

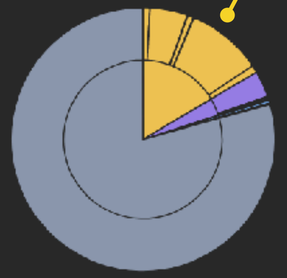
Updates

Ayumu

CPU USAGE

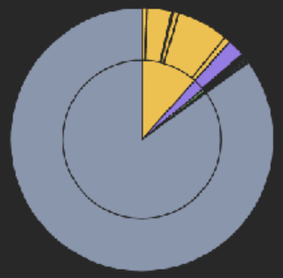
Most of CPU Usage is Scripting…

To Start Render



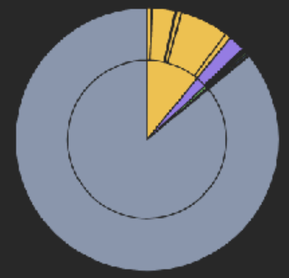
Scripting Layout Painting Loading Other Idle

To Page Load



Scripting Layout Painting Loading Other Idle

To Fully Loaded



Scripting Layout Painting Loading Other Idle

CONTENT BREAKDOWN

Pure Single Page Application（サイトA）

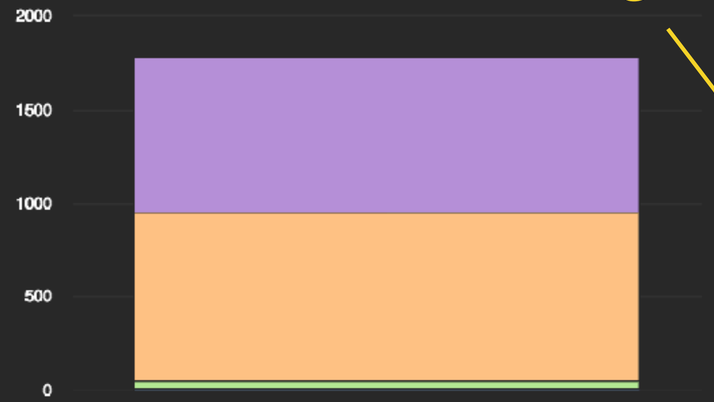
Requests

79

Size

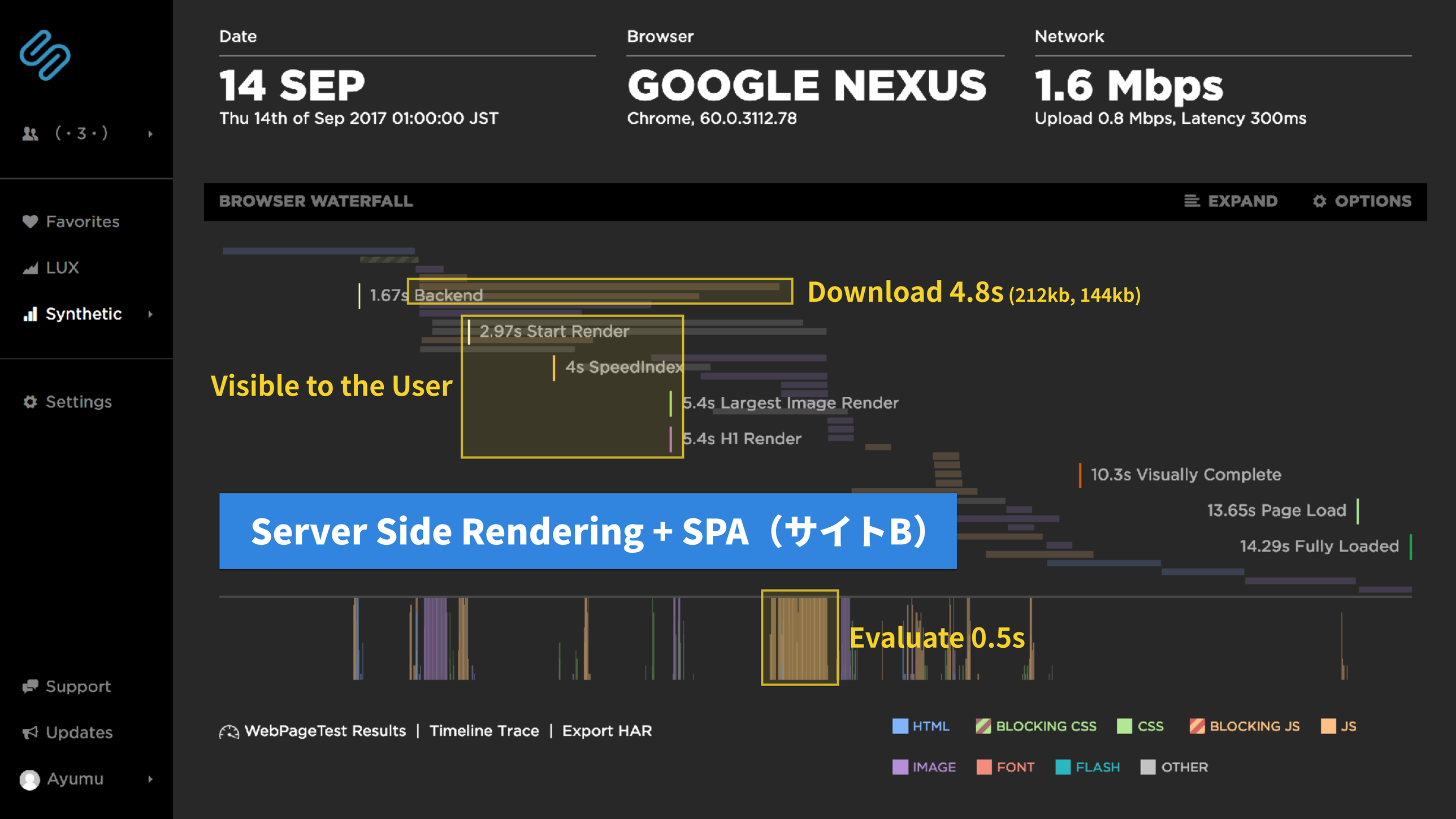
1,783 KB

JS size is larger than image size…



HTML	7	10 KB
CSS	1	40 KB
JS	18	904 KB
IMAGE	40	827 KB
FONT	0	0 KB
FLASH	0	0 KB
OTHER	13	2 KB

PERFORMANCE IMPROVEMENTS


(• 3 •)
Favorites
LUX
Synthetic
Settings
Support
Updates
Ayumu
Date
14 SEP
Thu 14th of Sep 2017 01:00:00 JST
Browser
GOOGLE NEXUS
Chrome, 60.0.3112.78
Network
1.6 Mbps
Upload 0.8 Mbps, Latency 300ms
BROWSER WATERFALL
EXPAND
OPTIONS
1.67s Backend
Download 4.8s (212kb, 144kb)
2.97s Start Render
4s SpeedIndex
Visible to the User
5.4s Largest Image Render
5.4s H1 Render
10.3s Visually Complete
13.65s Page Load
14.29s Fully Loaded
Server Side Rendering + SPA（サイトB）
Evaluate 0.5s
WebPageTest Results | Timeline Trace | Export HAR
HTML
BLOCKING CSS
CSS
BLOCKING JS
JS
IMAGE
FONT
FLASH
OTHER

(· 3 ·)

Favorites

LUX

Synthetic

Settings

Support

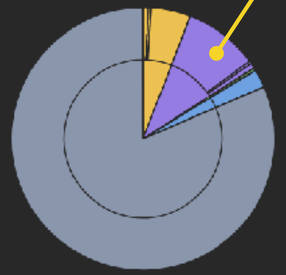
Updates

Ayumu

CPU USAGE

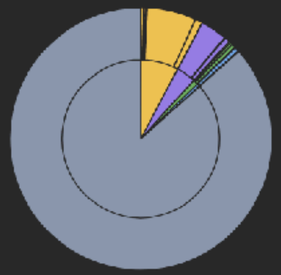
HTML is provided from SSR,
it contains a lot of layout processing

To Start Render



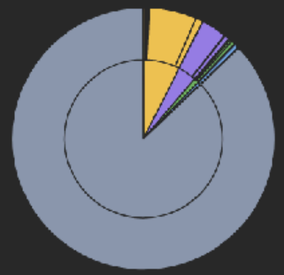
Scripting Layout Painting Loading Other
Idle

To Page Load



Scripting Layout Painting Loading Other
Idle

To Fully Loaded



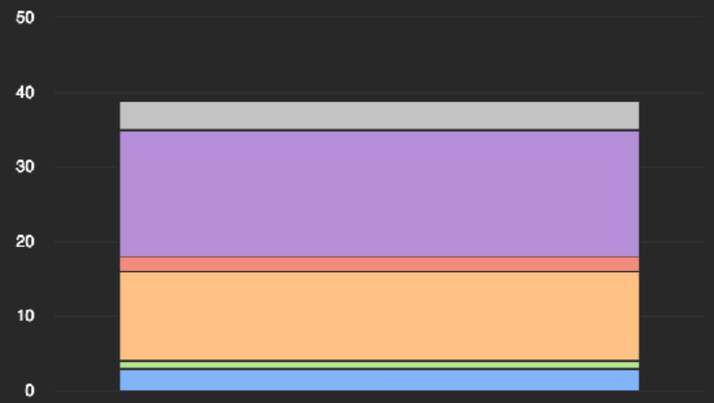
Scripting Layout Painting Loading Other
Idle

Server Side Rendering + SPA（サイトB）

CONTENT BREAKDOWN

Requests

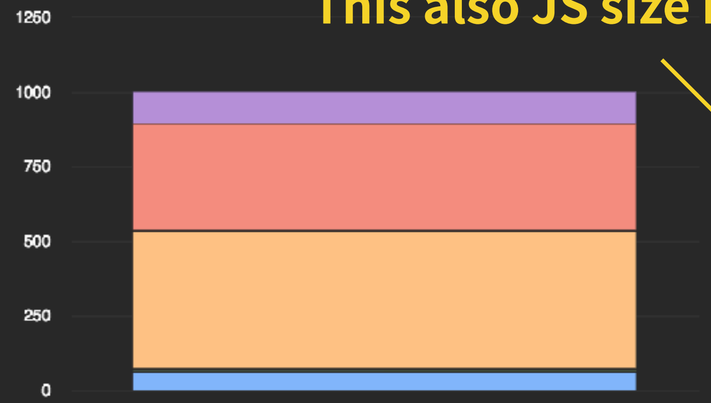
39



Size

1,004 KB

This also JS size is larger than image size



HTML	3	65 KB
CSS	1	9 KB
JS	12	462 KB
IMAGE	17	108 KB
FONT	2	359 KB
FLASH	0	0 KB
OTHER	4	1 KB

PERFORMANCE IMPROVEMENTS

JavaScript の初期化コストこわい

- 100kb の画像より 100kb の JS のほうが殺傷力をもつ
  - ダウンロードもパースも実行も大変なコスト
- ライブラリ選定時はファイルサイズも気をつけないと容易に破綻する
- webpack の code splitting みたいなのは必要な機能
- SSR も表示が速いだけで操作可能になるまでには時間がかかる

※ minify 適用後、gzip 適用前

lodash: 190 KB (12.5%)

react-dom: 183 KB (12.0%)

rx: 154 KB (10.1%)

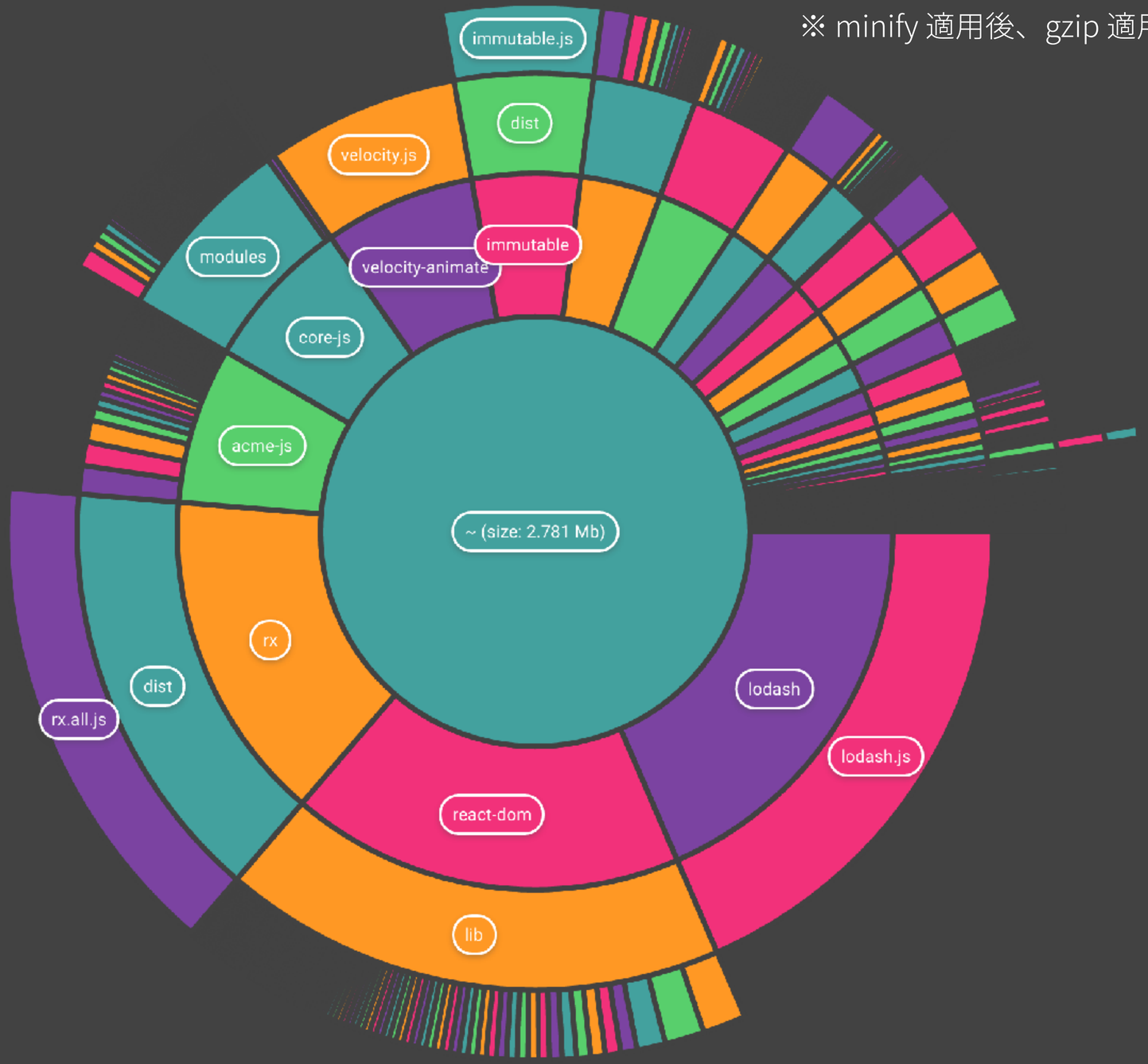
acme-js: 72KB (4.78%)

core-js: 70 KB (4.69%)

velocity-animate: 70KB (4.61%)

immutable: 49 KB (3.30%)

react: 37 KB (2.46%)



Generated by https://alexkuz.github.io/webpack-chart

Addy Osmani Follow

Senior Staff Engineer at Google working with the Chrome team • Creator of TodoMVC, Yeoman, Mat...

Feb 10 · 12 min read

https://medium.com/reloading/javascript-start-up-performance-69200f43b201

ページロードに関わるトピック

HTTP/2 多くの方がご存じでしょうが...

- 多重化による並列性の向上、ヘッダの HPACK 圧縮、サーバープッシュ etc...
- ただ入れるだけでは HTTP/2 の恩恵はより大きい問題によって誤差になりうる
- ドメインシャーディング不要とか concat 不要とか言われている
  - HTTP/1 でもデカい concat は NG、更新性があるならキャッシュヒット率も下がる

zopfli, brotli どっちもスイスのパンの名前らしい🍞

- **zopfli** は deflate 互換で、いくらか圧縮率が高い
  - つまり Accept-Encoding: gzip なクライアント相手なら使える
  - 通常の gzip よりも時間をかけて丁寧に圧縮するスタイル → gzip_static も検討すると良
- **brotli** は新規のアルゴリズムで、zopfli より更に高圧縮
  - Accept-Encoding: br を返せるのは Edge, Firefox, Chrome、もうすぐ Safari も

ES Modules 主要ブラウザで有効になるまであとちょっと？

- import mod from './mod.js' をブラウザ上でも解決可能になる
  - Import のパス指定のとおり JavaScript がロードされるようになる
- HTTP/2 と抱き合わせで Concat 不要論の完成に必要な最後のパーツ
- Polymer すら HTML Imports に別れを告げて ES Modules と握手した

ES modules 対応ブラウザ向けのブートストラップファイル

```
<script type="module" src="main.js"></script>
```

ES modules 未対応ブラウザ向けのバンドルファイル

```
<script nomodule src="main-bundled.js"></script>
```

See also 💡 Node.jsのES Modulesサポートの現状確認と備え - teppeis blog
http://teppeis.hatenablog.com/entry/2017/08/es-modules-in-nodejs

Preload, Resource Hints なんとなく似てるけど別仕様

- **Preload** は現在のページのために指定リソースを優先してロードする
- **Resource Hints** は以降のページのために投機的処理をする
  - dns-prefetch: DNS 解決の事前処理
  - preconnect: TCP 接続の事前処理
  - prefetch: リソース取得の事前処理
  - prerender: ページのレンダリングの事前処理(!!)

CSS 評価前に Preload で Web フォントの先行ロード

Web フォント待ちで発生するテキストのレンダリングブロックを軽減

```
<link rel="preload" href="font.woff" as="font" crossorigin>

<link rel="stylesheet" href="main.css">
```

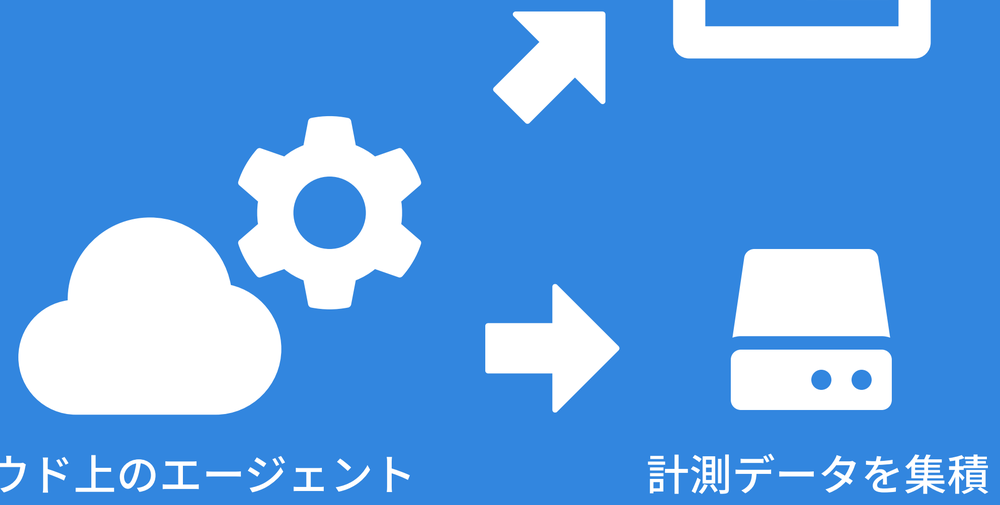
Resource Hints 各位

```
<link rel="dns-prefetch" href="//cdn.example.com">

<link rel="preconnect" href="https://api.example.com">

<link rel="prefetch" href="/libs.js" as="script">

<link rel="prerender" href="/next-page.html">
```

計測と評価指標

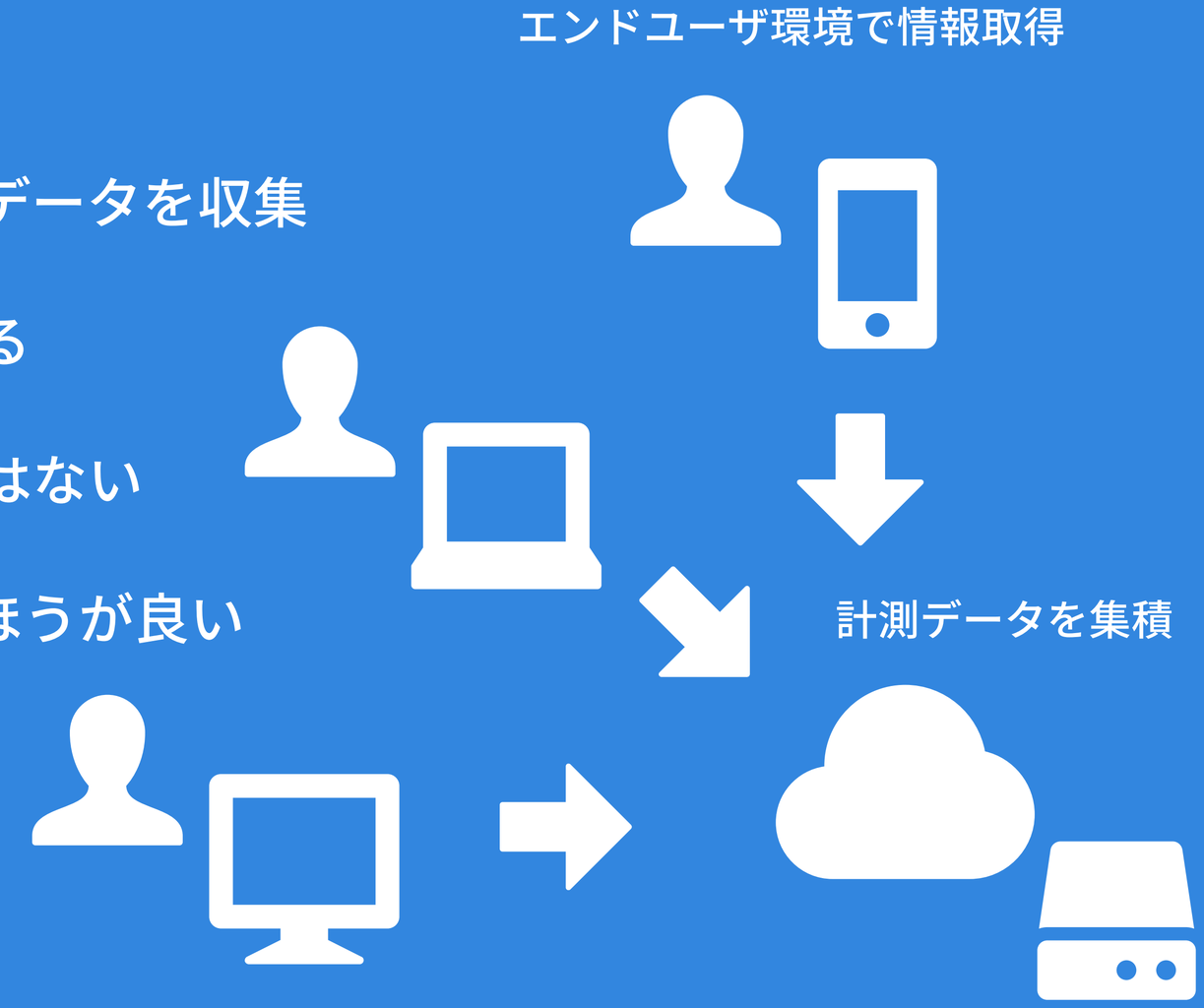
Synthetic Monitoring

- 特定環境から Web サイトにアクセスしてデータを収集
- 現実のユーザではないが速度の上下を評価しやすい
- 端末や回線など条件をコントロールできる
- 詳細な情報をとれるので開発者はありがたい



Real User Monitoring

- エンドユーザの端末上で記録したデータを収集
- 実際にユーザが体験した値をとれる
- 詳細なプロファイルを見るものではない
- KPI との関連を見るならこっちのほうが良い



Navigation Timing API, Resource Timing API

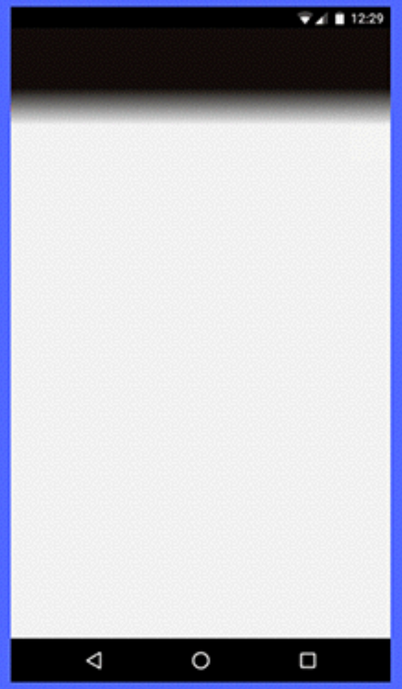
```
dumpTiming(performance.timing);
performance.getEntriesByType('resource').forEach(dumpTiming);

// ナビゲーションまたはリソースのロードにおける各フェーズの所要時間を出力
function dumpTiming(timing) {
  console.log(`
    Name:       ${timing.name || location.href}
    Unload:     ${timing.unloadEventEnd - timing.unloadEventStart}
    Redirect:   ${timing.redirectEnd - timing.redirectStart}
    DNS:        ${timing.domainLookupEnd - timing.domainLookupStart}
    TCP:        ${timing.connectEnd - timing.connectStart}
    Request:    ${timing.responseStart - timing.requestStart}
    Response:   ${timing.responseEnd - timing.responseStart}
    Processing: ${timing.loadEventStart - timing.responseEnd}
    OnLoad:     ${timing.loadEventEnd - timing.loadEventStart}`);
}
```

via. https://1000ch.net/posts/2016/performance-api.html 一部改変

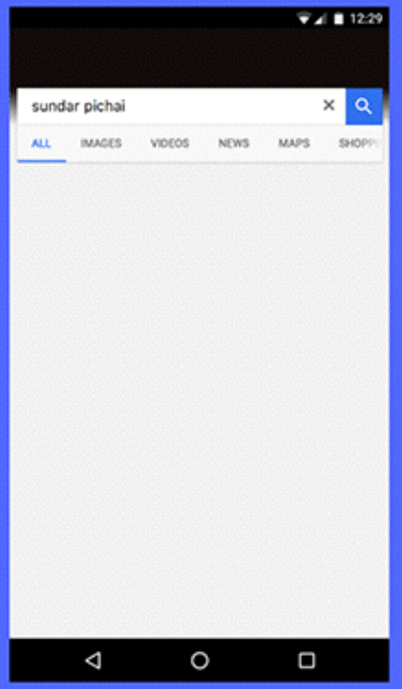
「ざっくり分かった気になれる指標」の変遷

古代 **ブラウザイベント期**	DOMContentLoaded load	ブラウザイベントで代用していた 相関性はあるが本当に見たい値ではない
近代 **ビジュアル描画の計量期**	Start Render (First Paint) Speed Index	描画処理のタイミングに関心が移った WebPagetest が流行り始める
現代 **ユーザー体験期**	First Contentful Paint First Meaningful Paint Time to Consistently Interactive	よりユーザー体験と直結する指標が登場 意味が伝わる表示になったタイミング？ 実際に利用可能になったタイミング？

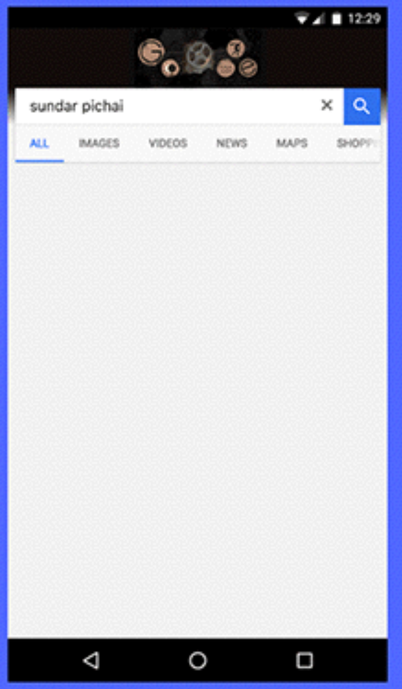
See also 💡 Web クライアントサイドのパフォーマンスメトリクス ::ハブろぐ
https://havelog.ayumusato.com/develop/performance/e744-performance_metrics.html

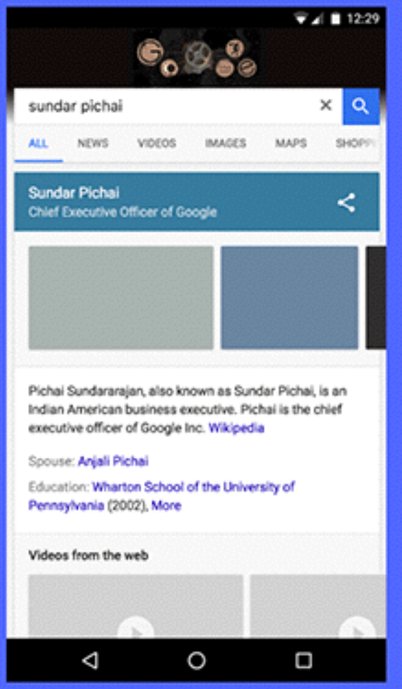


First Paint
(FP)

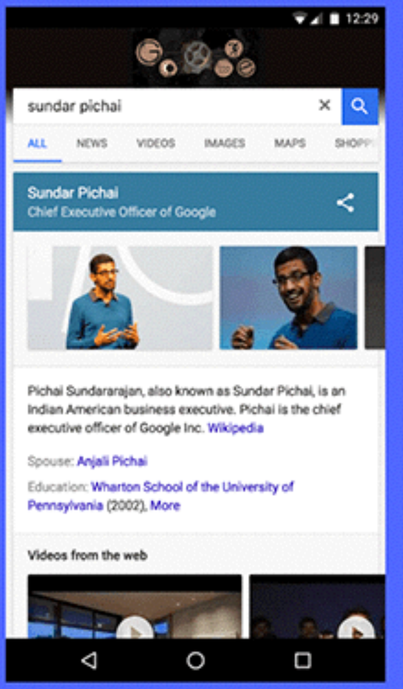


First Contentful
Paint (FCP)





First Meaningful
Paint (FMP)



Time to
Interactive (TTI)

Paint Timing API による FP と FCP の取得

```
const observer = new PerformanceObserver((list) => {
  for (const entry of list.getEntries()) {
    // `name` will be either 'first-paint' or 'first-contentful-paint'.
    const metricName = entry.name;
    const time = Math.round(entry.startTime + entry.duration);

    ga('send', 'event', {
      eventCategory: 'Performance Metrics',
      eventAction: metricName,
      eventValue: time,
      nonInteraction: true,
    });
  }
});
// Start observing paint entries.
observer.observe({entryTypes: ['paint']});
```

via. https://developers.google.com/web/updates/2017/06/user-centric-performance-metrics#tracking_fpfcp

Developer Tools - https://havelog.ayumusato.com/develop/performance/e744-performance_metrics.html

Elements | Console | Sources | Network | Performance | Memory | Application | Security | Audits | aXe

havelog.ayumusato.com 9/21/2017, 9:06:00 PM

Performance

These encapsulate your app's performance.

67

Metrics

These metrics encapsulate your app's performance across a number of dimensions.

- First meaningful paint 3,440 ms
- First Interactive (beta) 5,870 ms
- Consistently Interactive (beta) 5,870 ms
- Perceptual Speed Index: 4,914 (target: < 1,250) 56
- Estimated Input Latency: 148 ms (target: < 50 ms) 18

Opportunities

These are opportunities to speed up your application by optimizing the following resources.

- Properly size images 1,240 ms 241 KB
- Offscreen images 680 ms 132 KB

プロダクトによって本当に必要な指標は？

- さきほどの一般的な指標だけでは知りたいことが知れないことがある
- 速く表示されるのはもちろん、速く動画が再生開始してほしい、とか
- そこで User Timing API の出番

User Timing API で任意の2点間の所要時間を計測する例

```
let reqCount = 0;
const req = new XMLHttpRequest();
req.open('GET', url, true);
performance.mark('mark_start_xhr');
req.onload = (e) => {
  performance.mark('mark_end_xhr');
  reqCnt++;
  performance.measure(
    `measure_xhr_${reqCnt}`, 'mark_start_xhr', 'mark_end_xhr'
  );
  do_something(e.responseText);
}
req.send();

const entries = performance.getEntriesByType('measure');
entries.forEach((e)=>console.log(`XHR ${e.name} took ${e.duration}ms`))
```

via. https://www.html5rocks.com/ja/tutorials/webperformance/usertiming/ 一部改変

予算の設定と定常的な計測

- ピンポイントで速くするミッションが必要なこともあるにはあるが
- 予算(基準値)を決め、それよりも遅くならないよう運用するのがベースライン
- 日常的に数字と向き合ってこそ真っ当な高速化の手が打てる
- 有料だけど SpeedCurve はお手軽で開発者が好きそうな感じ

See also 💡 Performance Budget Metrics

https://timkadlec.com/2014/11/performance-budget-metrics

ランタイム

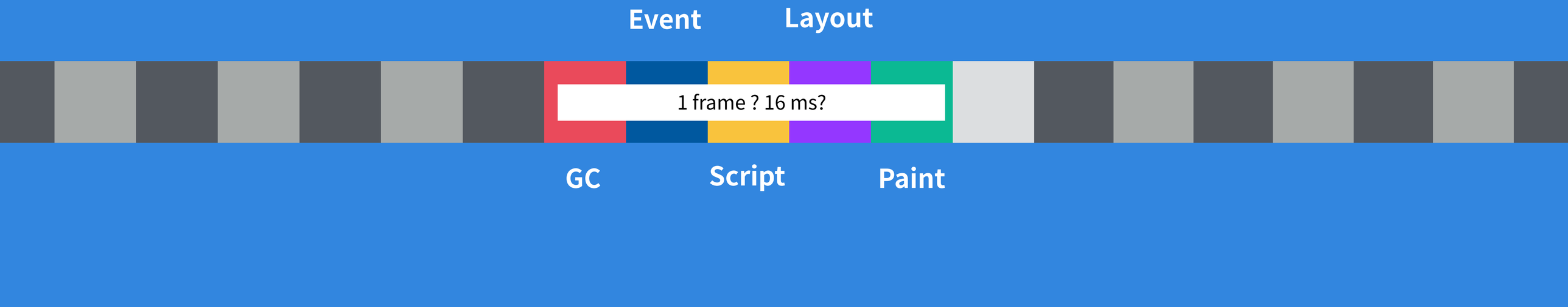
滑らかに動く UI と快適な操作感

ランタイム

- UI やアニメーションが滑らかに動くか
  - FPS、RAIL の Animation 16ms
- ユーザインプットに迅速に反応できるか
  - RAIL の Response 100ms と Idle 50ms

シングルスレッドな世界と向き合って生きる

- ブラウザはシングルスレッド、何かが長い処理をすれば他の処理は遅れる
- アニメーションのための処理が重いと、実際の画面の更新が遅れる
- 何かが処理を占有していると、ユーザーインプットへの応答が遅れる



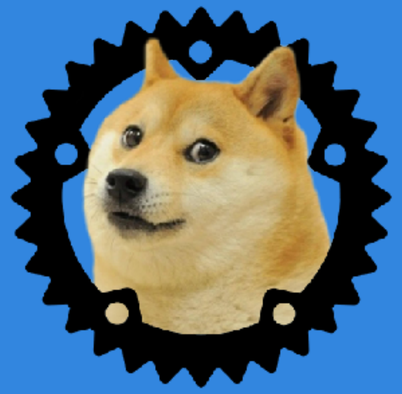
対策はいくつかある

- とにかく処理を軽く、短くする
- メインスレッドを長々と占有してしまうような処理を潰していく
  - イベントループ的に他の処理が割り込めるよう分割する (setTimeout とか)
  - 並列化できる処理は他のスレッドに逃がす（WebWorker とか）

レンダリングエンジンの開発側も色々やってる

- 特定の処理を専用スレッドに切り出して並列化させたり
  - Blink だと Raster Thread, Compositor Thread, ScriptStreamerThread とか
- レンダラのタスクを今よりも小さいチャンクに分割したり
- Mozilla の Project Quantum は名前がついていて有名な一連の取り組み

Project Quantum



- 実験場である Servo の成果を、コンポーネントとして Gecko に取り込む
- Quantum CSS (Stylo): スタイル処理とセレクタマッチの並列化
- Quantum Render: OpenGL ベースで CSS に特化したレンダラへの置き換え
- Quantum Compositor: Compositor のスレッドからプロセスへの独立
- Quantum DOM: 各タブのマルチプロセス&スレッド化とスケジュール管理
- etc...

See also 💡 MozillaはQuantumプロジェクトで過去と訣別し、未来に賭ける
http://rockridge.hatenablog.com/entry/2016/11/05/020802

ランタイムの特にレンダリング周りで
発生する一般的な問題...

Ayumu Sato
32 Presentations

Star this Talk 45 Stars
Published in Programming
Stats 8,053 Views

Edit this presentation

Share

Twitter, Facebook
Embed
Direct Link
Download PDF

Web Fron
Tips
by Ayumu
Published Novemb

なんと HTML5 Conference 2013 で発表した内容がまだ賞味期限内です

SpeakerDeck
https://speakerdeck.com/ahomu/web-frontend-rendering-performance-knowledge-and-tips

YouTube
https://www.youtube.com/watch?v=cBT1Are3jXE

ランタイムに関わるトピック

Intersection Observer スクロール同期処理の救世主

- ある要素を observe して対象が画面に出たり入ったりすると callback する
- 従来の getBoundingClientRect と clientWidth/Height のロジックから解放
- JS からレイアウト情報に参照する際のコーディングリスクを軽減できる
  - リスク = レイアウトの過剰な再計算、いわゆる Forced Layout や Layout Thrashing

従来の要素の画面内の入出検知

```
const target = document.getElementById('target');
let viewport = getViewportSize();
function getViewportSize() {
  return {
    width: document.documentElement.clientWidth,
    height: document.documentElement.clientHeight
  };
}
window.addEventListener('scroll', () => {
  const { width, height } = viewport;
  const rect = target.getBoundingClientRect();
  // 水平方向において要素の一部または全部が画面内に存在し得るか
  const isInHorizontal = rect.left > 0 && rect.left < width ||
                         rect.right > 0 && rect.right < width ||
                         rect.left < 0 && rect.right > width;
  // 垂直方向において要素の一部または全部が画面内に存在し得るか
  const isInVertical = rect.top > 0 && rect.top < height ||
                       rect.bottom > 0 && rect.bottom < height ||
                       rect.top < 0 && rect.bottom > height;
  // 要素の一部または全部が画面内に存在するか
  if (isInHorizontal && isInVertical)  console.log('要素が画面内に入りました');
  else console.log('要素が画面内から出ました');
}, false);
```

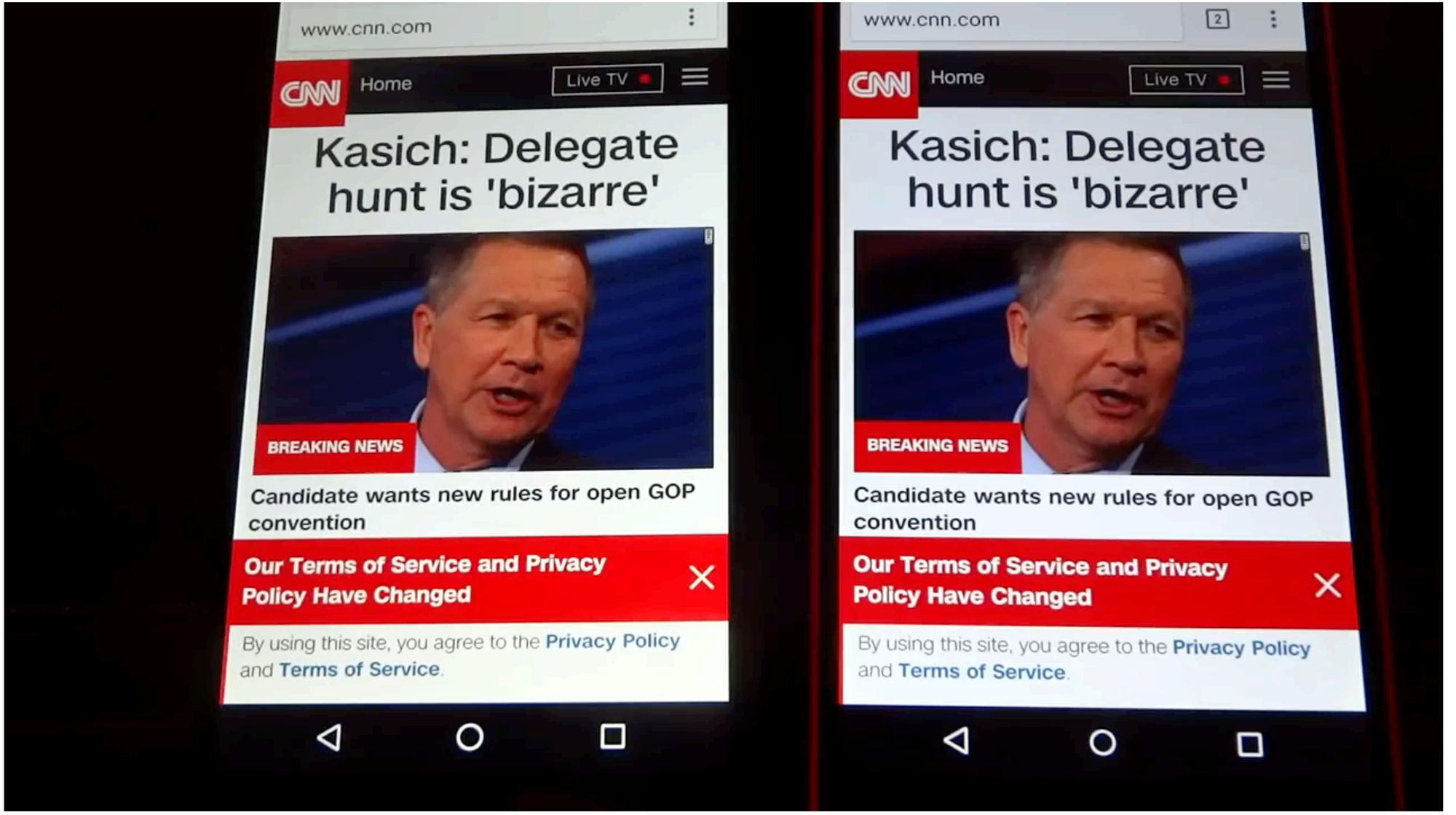
IntersectionObserver で同じことを実現する

```
const target = document.getElementById('target');
const observer = new IntersectionObserver(entries => {
  const entry = entries[0];

  // 要素が少しでも画面内に入っていれば真
  if (entry.intersectionRatio > 0) {
    console.log('要素が画面内に入りました');
  } else {
    console.log('要素が画面内から出ました');
  }
});
observer.observe(target);
```

Passive Event Listeners スクロールジャンクを軽減

- touchmove や mousewheel などはデバイスの操作と同期して発生する
- e.preventDefault() に備えて、画面更新の前にハンドラの処理待ちがある
- ☝を呼び出さないことを保証すれば、スムーズに画面更新できる

```
element.addEventListener('touchmove', handler, {
  passive: true
});
```


www.cnn.com
CNN
Home
Live TV
Kasich: Delegate hunt is 'bizarre'
BREAKING NEWS
Candidate wants new rules for open GOP convention
Our Terms of Service and Privacy Policy Have Changed
By using this site, you agree to the Privacy Policy and Terms of Service.
www.cnn.com
CNN
Home
Live TV
Kasich: Delegate hunt is 'bizarre'
BREAKING NEWS
Candidate wants new rules for open GOP convention
Our Terms of Service and Privacy Policy Have Changed
By using this site, you agree to the Privacy Policy and Terms of Service.

Scroll jank due to touch/wheel handlers demo

View on GitHub

Instructions

The log shows the touch/wheel events which block scrolling (have cancelable=true).
Scroll the page, observing the latency associated with each event.
The 'running' animation represents the responsiveness of the JavaScript thread. Jank can be added to this thread as a regular repeating pattern ("periodic jank"), and/or directly inside of each touch/wheel handlers ("handler jank").
In Chrome 51+ and Safari 10, the touch/wheel listeners can be marked as passive to keep them from being able to block the scroll.
When working properly (eg. Chrome 49+) this demo will be able to measure both types of jank.
Safari 9+ appears to work well (though only at millisecond precision and with possible issues with NTP skew).

Scroll performance resources

- https://blog.chromium.org/2016/05/new-apis-to-help-developers-improve.html
- Jank Free
- HTML5 Rocks - Scrolling Performance
- How to Use the Timeline Tool
- Measuring Webpage Jank

Scroll jank due to touch/wheel handlers demo

http://rbyers.github.io/scroll-latency.html

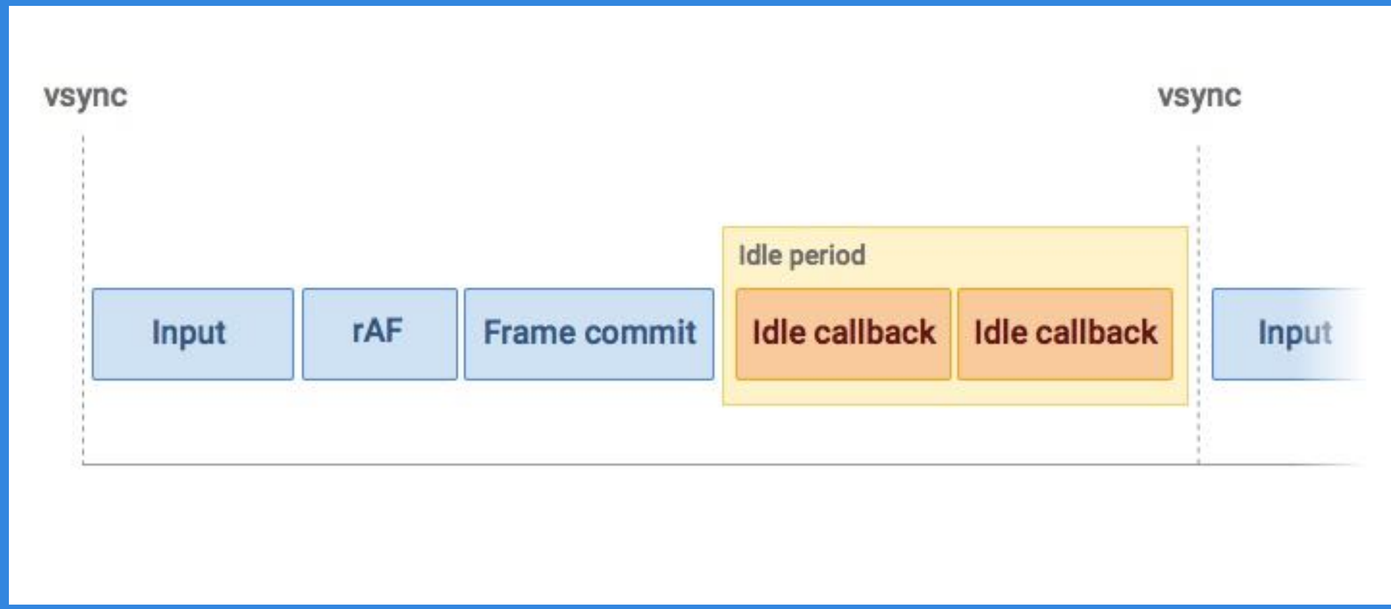
Periodic jank: 200 ms
Handler jank: 200 ms
Touch listener: touchstart
passive preventDefault
show non-blocking events

Running

wheel: 20.795ms
wheel: 21.29ms
wheel: 22.145ms
wheel: 20.945ms
wheel: 22.39ms
wheel: 22.775ms
wheel: 21.335ms

requestIdleCallback

- 処理すべきタスクがなく完全に入力待ちのとき(Idle)に発火するコールバック
- setTimeout(fn, 0) より恣意的
- フレームの更新を優先できる
- 使い道は色々ありそう



Using requestIdleCallback | Web | Google Developers
https://developers.google.com/web/updates/2015/08/using-requestidlecallback

requestIdleCallback で暇なときに最大 50ms 仕事をする

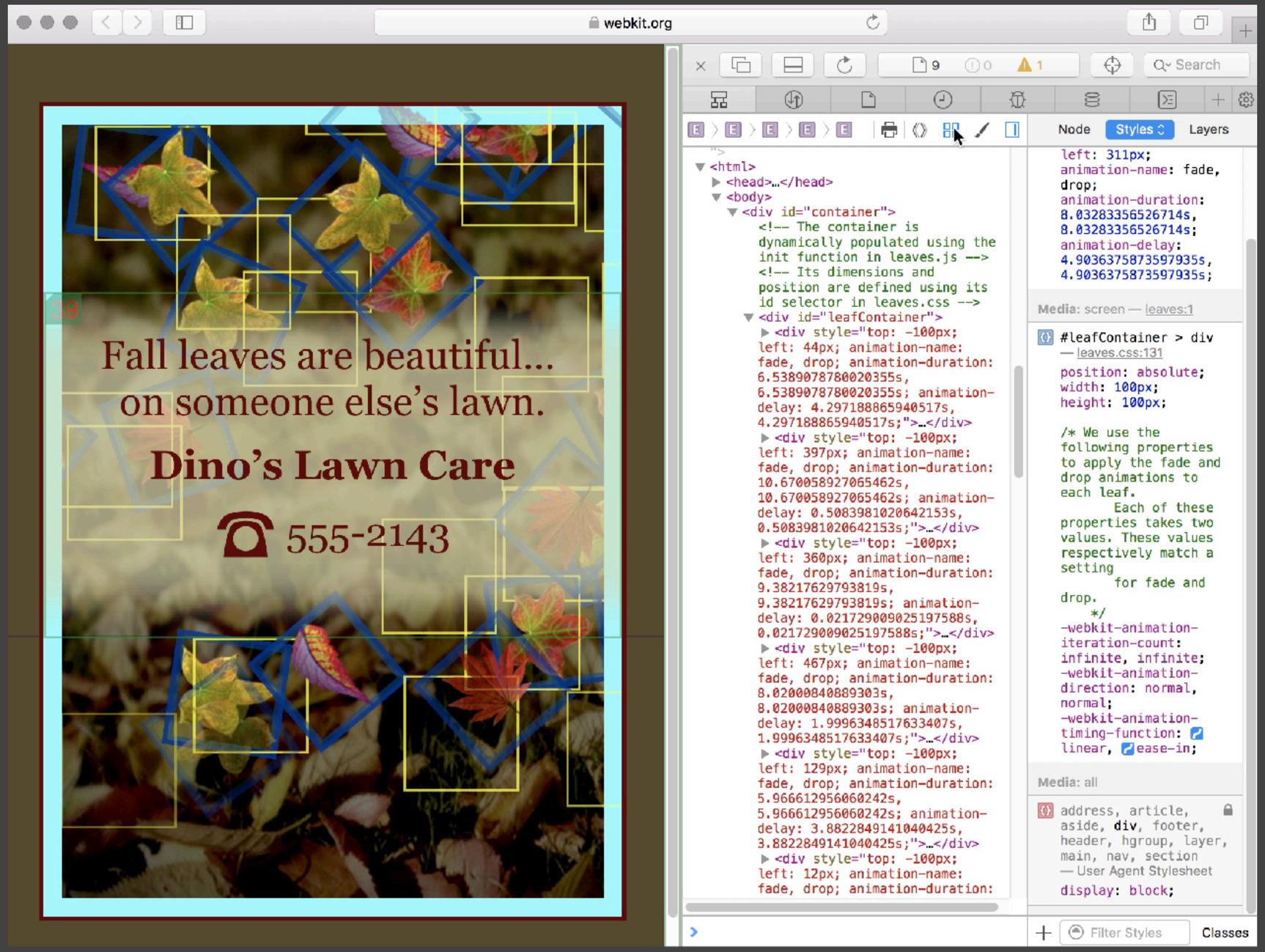
```
function myNonEssentialWork (deadline) {

  // rIC の持ち時間は 50ms であり、timeRemaining() はその残り時間を返す
  while (deadline.timeRemaining() > 0 && tasks.length > 0) {
    tasks.shift()();
  }

  if (tasks.length > 0) {
    requestIdleCallback(myNonEssentialWork);
  }
}

requestIdleCallback(myNonEssentialWork);
```

CSS Will Change 正しく使わないと副作用があるけど便利

- その要素に変更が発生することを、ブラウザに予告するプロパティ
- 乱暴にいうと `transform: translate3d(0, 0, 0)` ハックの正式版
- GPU アクセラレーションなど、ブラウザが勝手に最適化するためのマーキング

```
.element {
  transform: scale(1);
  will-change: transform;
}
.element.clicked {
  transform: scale(2);
}
```

See also 💡 CSS will-changeプロパティについて知っておくべきこと
https://dev.opera.com/articles/ja/css-will-change-property/



via. https://webkit.org/blog/324/css-animation-2/

CSS Containment 伝わらない CSS プロパティ第1位（※個人の感想）

- スタイル、レイアウト、描画の計算範囲を限定して最適化できる
- ある要素内の状態による外界への影響を封じ込めるプロパティ
  - contain: size → 指定された要素のサイズがコンテンツ、子孫要素に影響されなくなる
  - contain: layout → 指定された要素が親兄弟要素のレイアウトに影響を与えなくなる
  - contain: style → 指定された要素の外に影響を及ぼす一部のスタイルを内側で完結させる
  - contain: paint → 指定された要素の境界の外側に子孫要素がペイントされなくなる

Hello, CSS Containment!

1. contain: size
2. contain: layout
3. contain: style
4. contain: paint
5. height: 100px

I'm a Container's child!

Hello, CSS Containment Playground

https://codepen.io/ahomu/full/ZpKGrN/

This is Container

6. list item

7. list item

8. list item

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

I'm floating child. Neque porro quisquam est qui dolorem ipsum quia dolor sit amet, consectetur, adipisci velit

I'm floating child.. Neque porro quisquam est qui dolorem ipsum quia dolor sit amet, consectetur, adipisci velit

I'm floating child... Neque porro quisquam est qui dolorem ipsum quia dolor sit amet, consectetur, adipisci velit

References

9. CSS Containment Module Level 3

10. CSS Containment in Chrome 52 – Dev Channel – Medium

まとめ

言いたかったこと

- とにかく最近のデカい JavaScript には気をつけておくと良い
- ページロードの計測指標を知っておくと「なにが速いのか」が判断できて良い
- ランタイムは色々と API が充実してきてるからキャッチアップしとくと良い
- 具体的な調査や改善のケーススタディについては...

超速！Webページ速度改善ガイド（仮）

著者: @ahomu @1000ch

発売: 11月23日 (予定)

出版: 技術評論社

乞うご期待!!

Thank you 🍻

- http://aho.mu
- @ahomu
- github.com/ahomu