Rendering Performance

#perfmatters

ケノドン ブノア
@oldergod


fast is a feature
ORBO

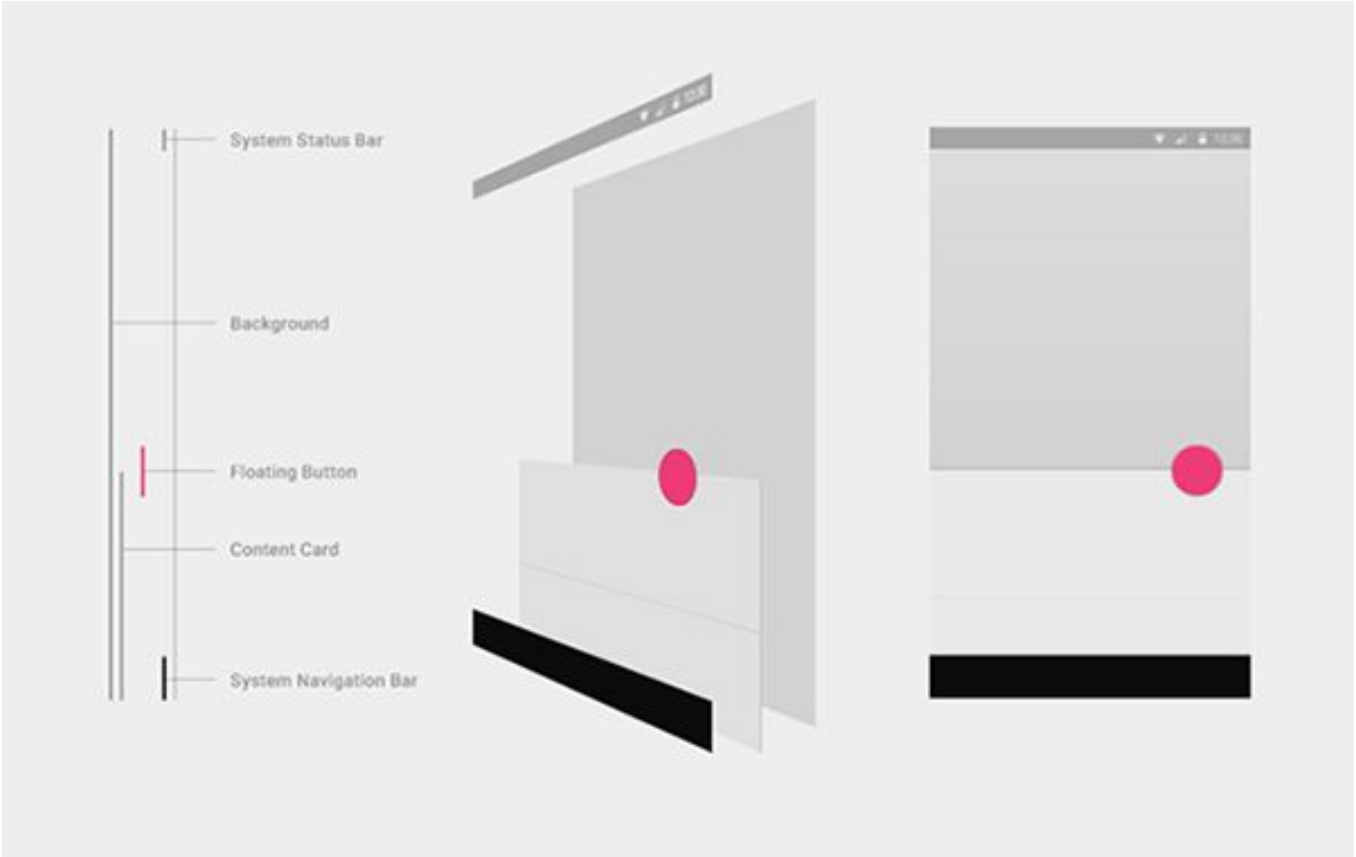
60fps 閾値

- 主のデバイスは1秒に画面を60回リフレッシュしている
  - 1 frame = 16.6ms（1000ms ÷ 60）
- アニメーション中にフレームを作る処理が16msを超えたらフレームレートが低下し、画面はガタツキが発生する

⇒ **ジャンク**（Jank）と呼ぶ

  - 多くなるとUXにマイナスな影響がでる

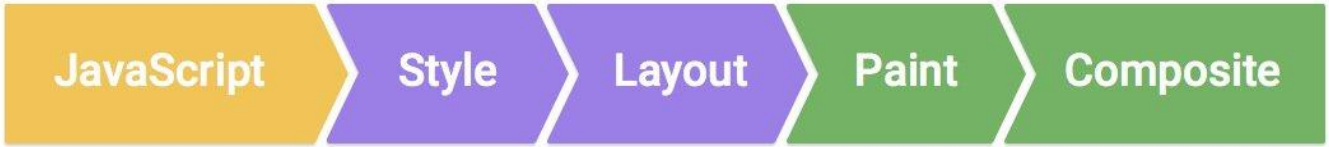
Layer って何？


System Status Bar
Background
Floating Button
Content Card
System Navigation Bar

Pixel Pipeline

- Javascript：視覚的に変化を起こす処理（javascriptに限らず）
- Style：CSSルールがどの要素に一致するかを、各要素の最終的なスタイルを算出する
- Layout（reflow）：Styleによって各要素は、どのくらいのスペースが必要か、画面内の位置、などを計算し始める
- Paint：レイヤーのピクセルを書き込む処理
- Composite：全レイヤーを組み合わせ、正しくレンダリングする

JS > Style > Layout > Paint > Composite



- パフォーマンス的に何ができるか？
- どこでネックができたり、どう解決するか？
- そもそもすべてのジョブを通す必要があるか？

- JS / Style / Composite：必須

⇒ Layout、Paintについては？

  - 利用するスタイルによってスキップできる？

JS > Style > Layout > Paint > Composite

「レイアウト」プロパティの変更 ⇒

- 幅、高さ、左または上の位置などに従って、ブラウザは他のすべての要素との影響を確認する必要がある
- 影響する領域は再ペイントする必要がある

JS > Style > Paint > Composite

「ペイントのみ」プロパティの変更 ⇒

- 背景画像、テキストの色、影などの変更
- ページのレイアウトに影響を与えないものはスキップ
- ペイントは行われる

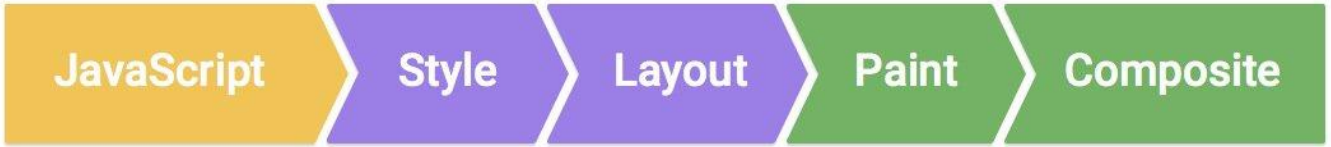
JS > Style > Composite

「コンポジットのみ」プロパティの変更 ⇒

- ブラウザは合成を行うためにジャンプする

任意の CSS プロパティを変更すると上記の 3 つのバージョンのどれが実行されるかが知りたい方 ⇒ https://csstriggers.com/

Pixel Pipelineのベストを尽くす



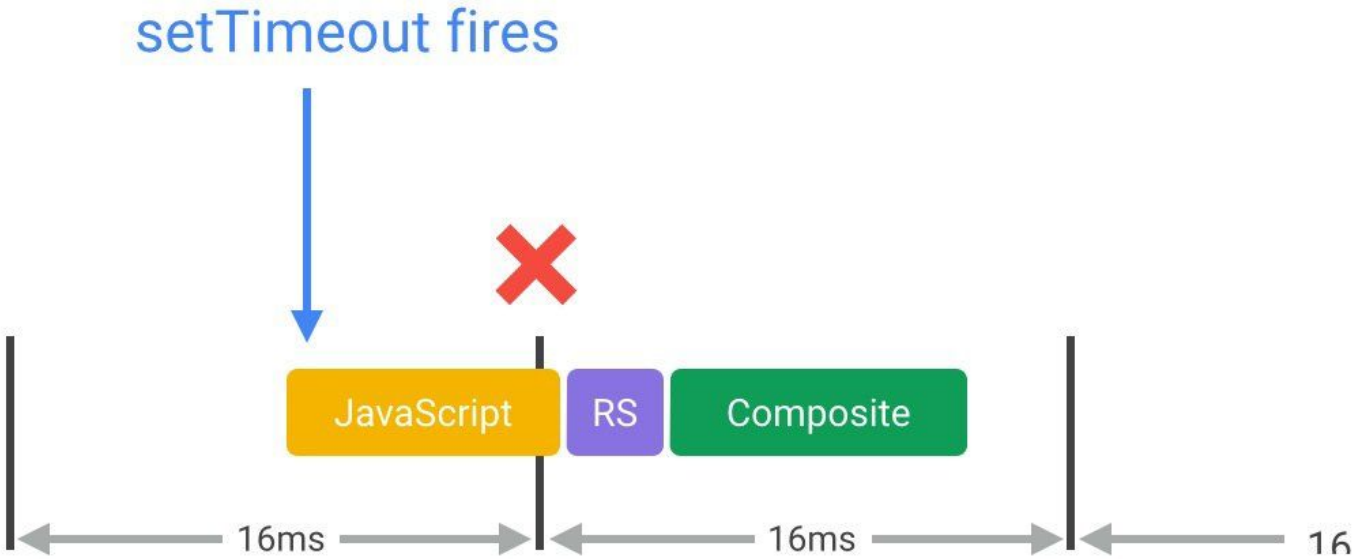
各タスクに対して何ができるかを

細かくみてみよう

JavaScript

Animation with JavaScript

- アニメーションに setTimeout、setInterval が使われる事はあるが**あかん**：いつ実行されるかわからない



Animation with JavaScript

- 60fps よりはやい setInterval は無駄
- 60fps より遅い setInterval はガタツキ
- setInterval(16.6) はブラウザに同期しないとガタツキ
- ヨーロッパは50Hzのデバイスがまだ多い

⇒ ブラウザのペースに合わせた `requestAnimationFrame` を使おう。

Window.requestAnimationFrame()

```
function step() {
  // アニメーション処理, e.g. translateX += n; 等

  // 更に次のフレームにも実行する処理を登録
  requestAnimationFrame(step);
}

// 次のフレームの頭に実行しておきたい処理を登録
requestAnimationFrame(step);
```

requestAnimationFrame() サポート

- IE10 ≧ OK
- 他のブラウザは前からOK
- Polyfillも書ける

⇒ requestAnimationFrame を使おう

Main Thread（又は UI Thread）

- Main Thread：スタイル計算、レイアウト、ペイントと一緒にJSが実行される
- そのすべてが16ms以内に終わらない ⇒ ジャンク
- JavaScript の実行中は他のタスク（UI系も含め）がブロックされる

レンダリングとJavaScriptの関係 - 実験

http://oldergod.github.io/ui-rendering-opt/demos/main-thread/index.html

Main Thread（又は UI Thread）

⇒ DOMアクセスが必要のない処理は **Web Worker** に移動

- ロード処理、検索、ソートとかに使える！

Web Worker

- ➕ 別スレッドで動く
- ➕ レンダリングに影響がでない
- ➕ XMLHttpRequest可能
- ➖ DOMアクセス不可能

どうしてもメインスレッドで実行したい場合はバッチ的に小さいタスクを `requestAnimationFrame` で実行するのがおすすめ

Web Worker デモ

http://oldergod.github.io/ui-rendering-opt/demos/web-workers/index.html

Web Worker デモ

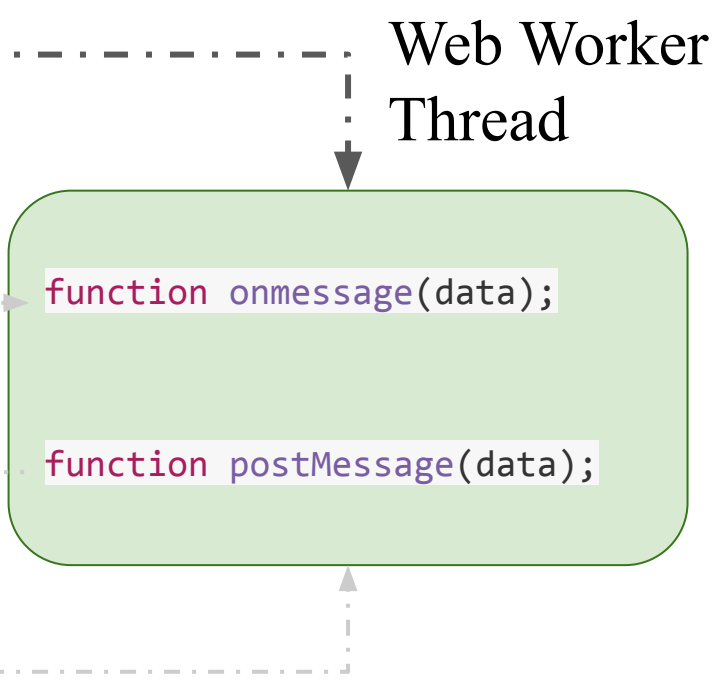
```
// worker の登録
const worker = new Worker('scripts/worker.js');

// manipulateImage にて worker にメッセージを送信
worker.postMessage({ type, imageData });

// メッセージの受信時
worker.onmessage = function(message) {
  const imageData = message.data;

  toggleButtonsAbledness();
  ctx.putImageData(imageData, 0, 0);
};

// worker の終了
worker.terminate();
```



Web Worker デモ

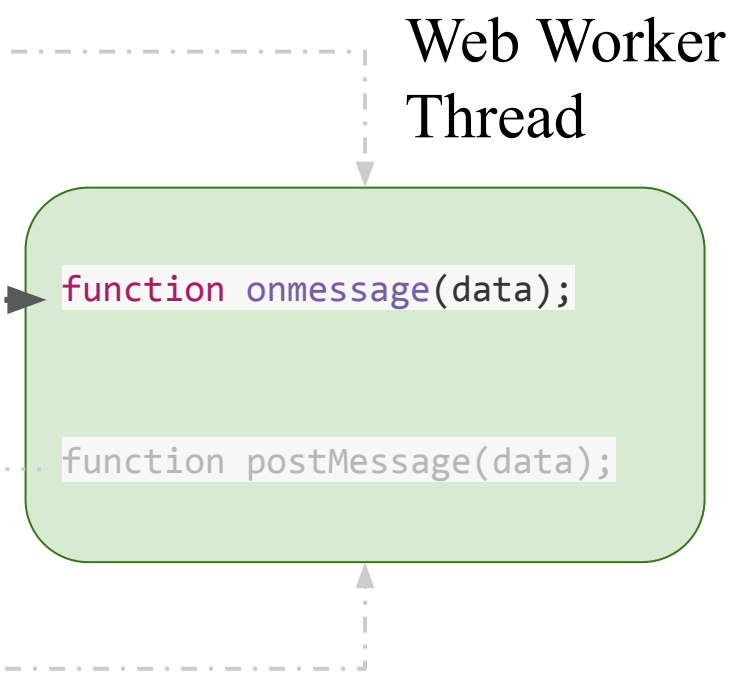
```
// worker の登録
const worker = new Worker('scripts/worker.js');

// manipulateImage にて worker にメッセージを送信
worker.postMessage({ type, imageData });

// メッセージの受信時
worker.onmessage = function(message) {
  const imageData = message.data;

  toggleButtonsAbledness();
  ctx.putImageData(imageData, 0, 0);
};

// worker の終了
worker.terminate();
```



Web Worker デモ

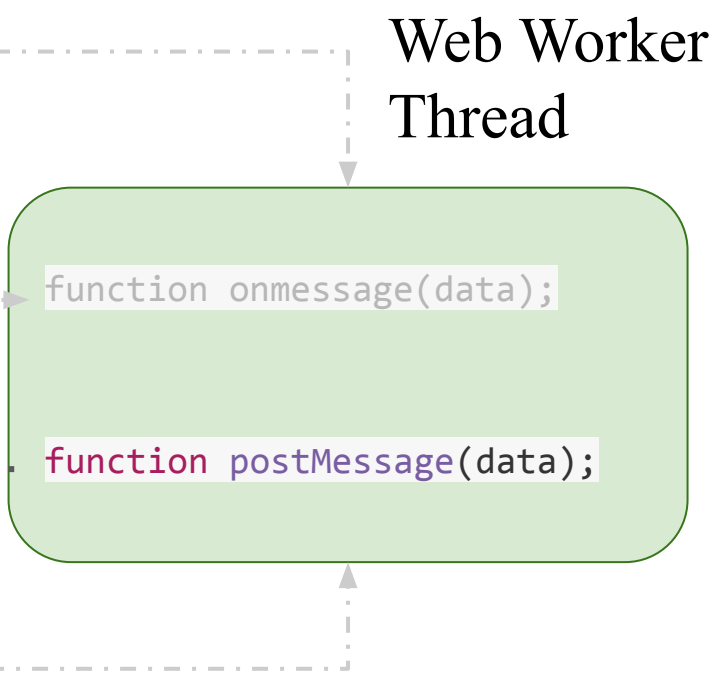
```
// worker の登録
const worker = new Worker('scripts/worker.js');

// manipulateImage にて worker にメッセージを送信
worker.postMessage({ type, imageData });

// メッセージの受信時
worker.onmessage = function(message) {
  const imageData = message.data;

  toggleButtonsAbledness();
  ctx.putImageData(imageData, 0, 0);
};

// worker の終了
worker.terminate();
```



Web Worker デモ

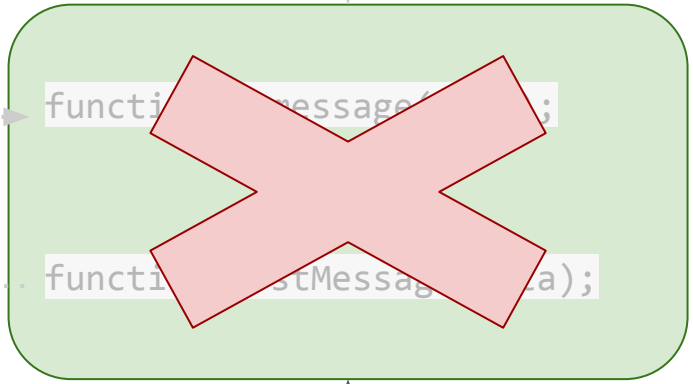
```
// worker の登録
const worker = new Worker('scripts/worker.js');

// manipulateImage にて worker にメッセージを送信
worker.postMessage({ type, imageData });

// メッセージの受信時
worker.onmessage = function(message) {
  const imageData = message.data;

  toggleButtonsAbledness();
  ctx.putImageData(imageData, 0, 0);
};

// worker の終了
worker.terminate();
```



Web Worker デモ

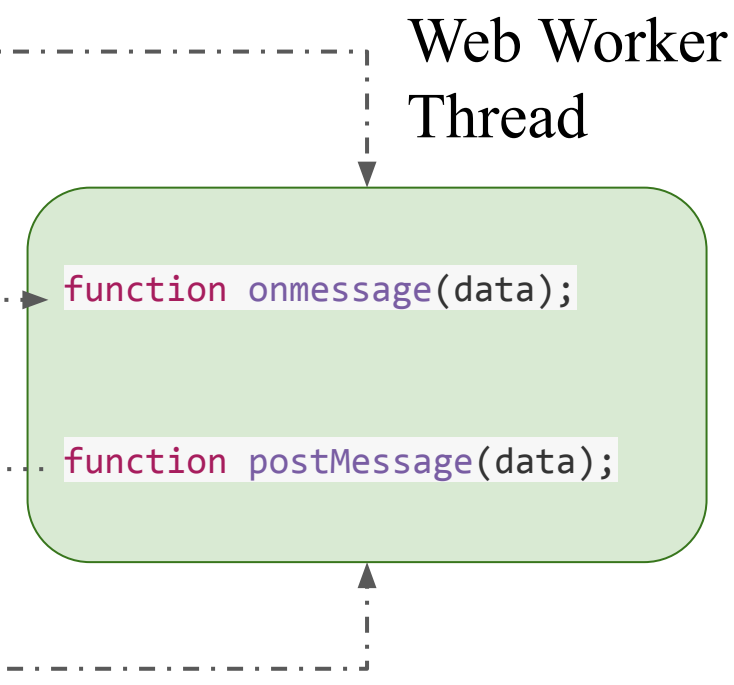
```
// worker の登録
const worker = new Worker('scripts/worker.js');

// manipulateImage にて worker にメッセージを送信
worker.postMessage({ type, imageData });

// メッセージの受信時
worker.onmessage = function(message) {
  const imageData = message.data;

  toggleButtonsAbledness();
  ctx.putImageData(imageData, 0, 0);
};

// worker の終了
worker.terminate();
```



Web Worker サポート

- IE10 ≧ OK
- 他のブラウザは前からOK

Layout

複雑なLayout、Layout Thrashing

- Layoutのスコープは基本、documentのすべて※
  - 一つだけ弄れば、全部が再計算される
  - アイテムが多ければ多いほどコストが高まる

⇒ 可能な限りレイアウトを回避

  - Layoutを実行しないスタイルプロパティを利用する
  - DevToolsで調査して最善を尽くす

Layout Thrashing デモ

Layout Thrashing：強制的な同期レイアウトを数多く実行する事

http://oldergod.github.io/ui-rendering-opt/demos/layout-thrashing/index.html

強制的な同期レイアウトの原因

```
while (i--) {
  paragraphs[i].style.width = greenBar.offsetWidth + 'px';
}
```

強制的な同期レイアウトの原因

```
paragraphs[i + 1].style.width = greenBar.offsetWidth + 'px';

paragraphs[i].style.width = greenBar.offsetWidth + 'px';
```

強制的な同期レイアウトの原因

```
offsetWidth = greenBar.offsetWidth + 'px';

paragraphs[i + 1].style.width = offsetWidth;
```

```
offsetWidth = greenBar.offsetWidth + 'px';

paragraphs[i].style.width = offsetWidth;
```

強制的な同期レイアウトの原因

```
offsetWidth = greenBar.offsetWidth + 'px';

paragraphs[i + 1].style.width = offsetWidth;


offsetWidth = greenBar.offsetWidth + 'px';

paragraphs[i].style.width = offsetWidth;
```

強制的な同期レイアウトの原因

```
// 最後に計算された layout 情報
offsetWidth = greenBar.offsetWidth + 'px';

paragraphs[i + 1].style.width = offsetWidth;


offsetWidth = greenBar.offsetWidth + 'px';

paragraphs[i].style.width = offsetWidth;
```

強制的な同期レイアウトの原因

```
// 最後に計算された layout 情報
offsetWidth = greenBar.offsetWidth + 'px';

paragraphs[i + 1].style.width = offsetWidth;


offsetWidth = greenBar.offsetWidth + 'px';

paragraphs[i].style.width = offsetWidth;
```

強制的な同期レイアウトの原因

```
// 最後に計算された layout 情報
offsetWidth = greenBar.offsetWidth + 'px';
// style を変更 ⇒ layout が無効化
paragraphs[i + 1].style.width = offsetWidth;


offsetWidth = greenBar.offsetWidth + 'px';

paragraphs[i].style.width = offsetWidth;
```

強制的な同期レイアウトの原因

```
// 最後に計算された layout 情報
offsetWidth = greenBar.offsetWidth + 'px';
// style を変更 ⇒ layout が無効化
paragraphs[i + 1].style.width = offsetWidth;
```


```
paragraphs[i].style.width = offsetWidth;
```

強制的な同期レイアウトの原因

```
// 最後に計算された layout 情報
offsetWidth = greenBar.offsetWidth + 'px';
// style を変更 ⇒ layout が無効化
paragraphs[i + 1].style.width = offsetWidth;

// layout が無効なため、再計算
offsetWidth = greenBar.offsetWidth + 'px';

paragraphs[i].style.width = offsetWidth;
```

強制的な同期レイアウトの原因

```
// 最後に計算された layout 情報
offsetWidth = greenBar.offsetWidth + 'px';
// style を変更 ⇒ layout が無効化
paragraphs[i + 1].style.width = offsetWidth;

// layout が無効なため、再計算
offsetWidth = greenBar.offsetWidth + 'px';

paragraphs[i].style.width = offsetWidth;
```

強制的な同期レイアウトの原因

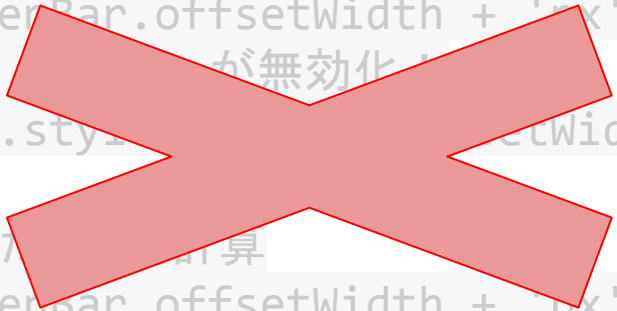
```
// 最後に計算された layout 情報
offsetWidth = greenBar.offsetWidth + 'px';
// style を変更 ⇒ layout が無効化
paragraphs[i + 1].style.width = offsetWidth;

// layout が無効なため、再計算
offsetWidth = greenBar.offsetWidth + 'px';
// style を変更 ⇒ layout が更に無効化、etc.
paragraphs[i].style.width = offsetWidth;
```

強制的な同期レイアウトの原因

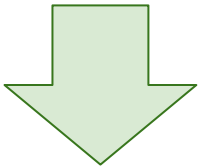
```
// 最後に計算された layout 情報
offsetWidth = greenBar.offsetWidth + 'px';
// style を変更 ⇒ layout が無効化！
paragraphs[i + 1].style.width = offsetWidth;

// layout が無効な状態で計算
offsetWidth = greenBar.offsetWidth + 'px';
// style を変更 ⇒ layout が無効化！
paragraphs[i].style.width = offsetWidth;
```



強制的な同期レイアウトの原因

```
while (i--) {
  paragraphs[i].style.width = greenBar.offsetWidth + 'px';
}
```



```
let offsetWidth = greenBar.offsetWidth + 'px';
while (i--) {
  paragraphs[i].style.width = offsetWidth;
}
```

強制的な同期レイアウトの原因

- Layout系の情報は最後のフレーム時のデータ
- Style を更新すると、Layoutの情報が無効となり、Layout再計算が必要となる

⇒ Style、Layoutの取得を一回のみ、最初にしておく事

⇒ コードを書く時にPixel Pipelineの順序を考慮しよう

Paint

ペイントをシンプルにする

- ペイントはパイプライン内のすべてのタスクのうちで最も長く実行される

ペイントの対象を絞る

- ブラウザは複数のレイヤー、必要な場合はコンポジ層だけに対してペイントを行うことができる

⇒ 再ペイントされるもの、移動するものを、他のコンポネントに影響を与えないで処理できる！

レイヤーはどう生成できる？

Composite

レイヤーの作る方法

1. 最良： will-change: transform;
2. ハック：transform: translateZ(0);
    - will-changeをサポートしていないブラウザに利用

コンポジットのみプロパティを使用

⚠️ 条件：プロパティを変更するに要素が自身コンポジタのレイヤーでなければならない！

⇒ その要素をプロモートし、レイヤーを作る

コンポジットのみプロパティを使用

- Position　transform: translate(xpx, ypx);
- Scale　transform: scale(n);
- Rotation　transform: rotate(ndeg),
- Skew　transform: skew(X|Y)(ndeg),
- Matrix　transform: matrix(3d)(...),
- Opacity　opacity: 0...1;

レイヤーの作る方法 デモ

http://oldergod.github.io/ui-rendering-opt/demos/layout-management/index.html

コンポネントのプロモート

will-change: transform; か transform: translateZ(0); を使う事で要素がプロモートされる

```
* {
  will-change: transform;        // 管理とメモリコストが
  transform: translateZ(0);      // 余計に発生するからNG
}
```

will-change サポート

- IE、Edge：アウト
- 他のブラウザは超最近でOK

Rendering Performance

まとめ : 上

- 改修する前に必ず測る事
- 問題ないものを直すのが無駄

⇒ 測ってから改修するのが第一

Rendering Performance

まとめ : 下

- Less work
- Scheduled work
- Isolated work
- Smooze work

⇒ Happy User

Fin

#perfmatters

格好良いエンジニア募集中！
http://goo.gl/9GK409

リファレンス

- グーグルによるレンダリングパフォーマンス
  - https://developers.google.com/web/fundamentals/performance/rendering/?hl=ja
- 無料ブラウザレンダリング改善コース
  - https://www.udacity.com/course/browser-rendering-optimization--ud860
- デモのレポジトリ:
  - https://github.com/oldergod/ui-rendering-opt
- CSSプロパティ詳細
  - https://csstriggers.com/
- 効率良いCSS
  - https://developer.mozilla.org/en-US/docs/Web/Guide/CSS/Writing_efficient_CSS
- BEM方針
  - https://en.bem.info
- アニメーションFLIP方法
  - https://github.com/googlechrome/flipjs