光を超えるための
フロントエンドアーキテクチャ

@MIZCHI / HTML5 CONFERENCE

ABOUT

- @mizchi / 竹馬光太郎
- フリーランスエンジニア

- フロントエンド専門
  - React / Node.js / SPA / PWA

お品書き

- 16ms以上の世界
  - Cache Aware な設計
- 16ms未満の世界
  - マイクロチューニング
  - 設計パターン

今日話すこと

- フロントエンドのキャッシュ設計
- マイクロチューニング指針
- リッチクライアントのパフォーマンス

今日話さないこと

- フレームワーク依存の話
  - React / Vue / Angular / etc
- サーバーでの各種実行コスト
  - データベースのクエリコストなど

大前提

この宇宙では

光が遅すぎる！

光が遅すぎる

- 国内: 20ms~
- アメリカ西海岸: 120ms~
- ヨーロッパ: 260ms~

ハードウェアの性能

- 一般的なディスプレイ: 60~144Hz
- VRゴーグル: 90~ Hz
- ヒトの知覚: 70~100Hz ※

フリッカー融合頻度: https://ja.wikipedia.org/wiki/フリッカー

フロントエンド技術の目的

- 小さいスコープ: リッチクライアント技術により、(部分的に) 60fps を達成する
- 大きなスコープ: 通信を含めて、その遅延・ペイロードを可能な限り抑える

16MS 以上の世界

遅延を分類する

- ~16ms: リアルタイム
- 50~100ms: フィードバックを意識
- 300~ms: やや遅いと感じる
- 1000~ms: 遅い、不満

RAILモデルでパフォーマンスを計測する - https://developers.google.com/web/fundamentals/performance/rail?hl=ja

DEV

search

Koutaro Chikuba
@mizchi

MY READING LIST (EMPTY)

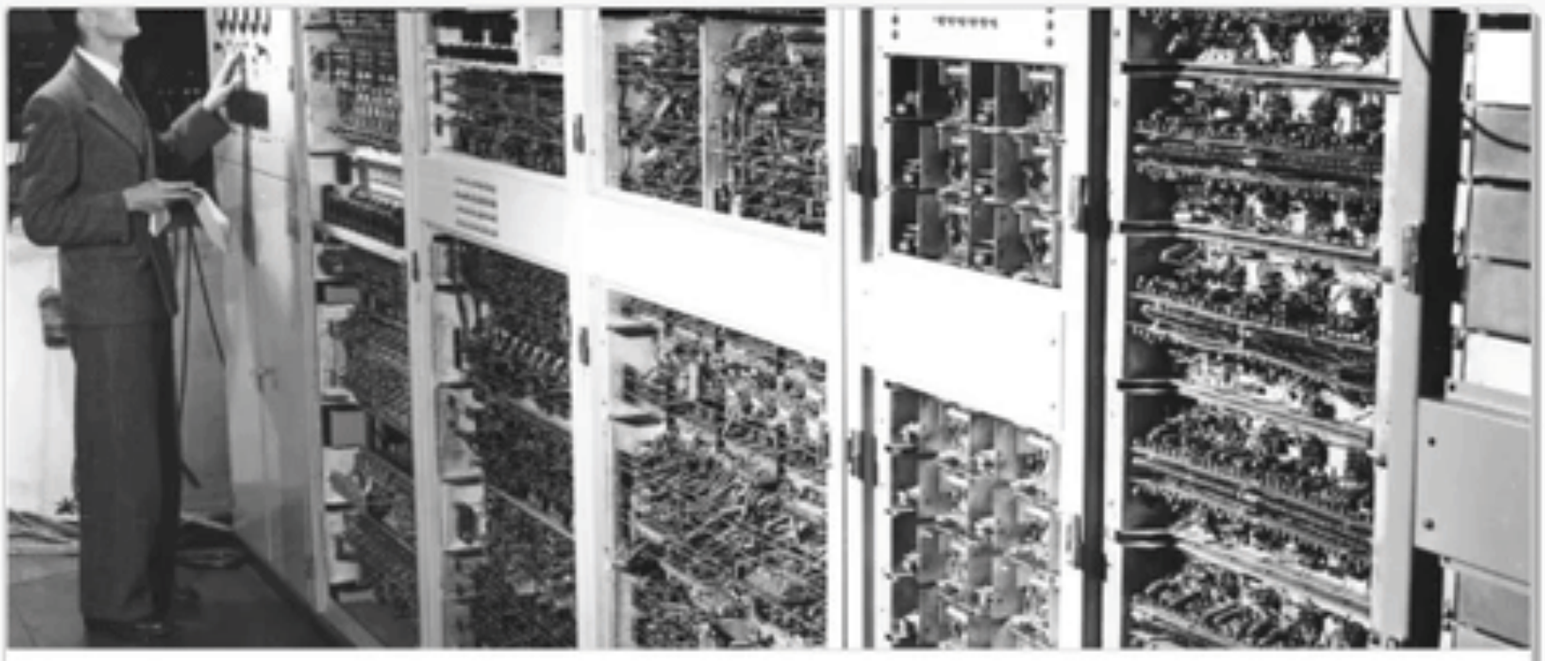
Saved Posts | Comment Activity

my tags

Follow tags to improve your feed

#javascript
#discuss
#webdev
#beginners
#react + FOLLOW
#career
#productivity
#python
#node
#opensource
#showdev

FEED | WEEK MONTH YEAR INFINITY | LATEST



How latency numbers changes from 1990 to 2020.

Sahil Rajput • Nov 23
#latency #computing

37 10 SAVE

The DEV Team

DEV.TO が教えてくれる(身も蓋もない)事実

- 構築済みHTMLをCDN Edgeで返すと速い
- 裏で先読みしておくと速い
- (この際、サーバーアーキテクチャは正直何でもいい。到達しない)

SERVER

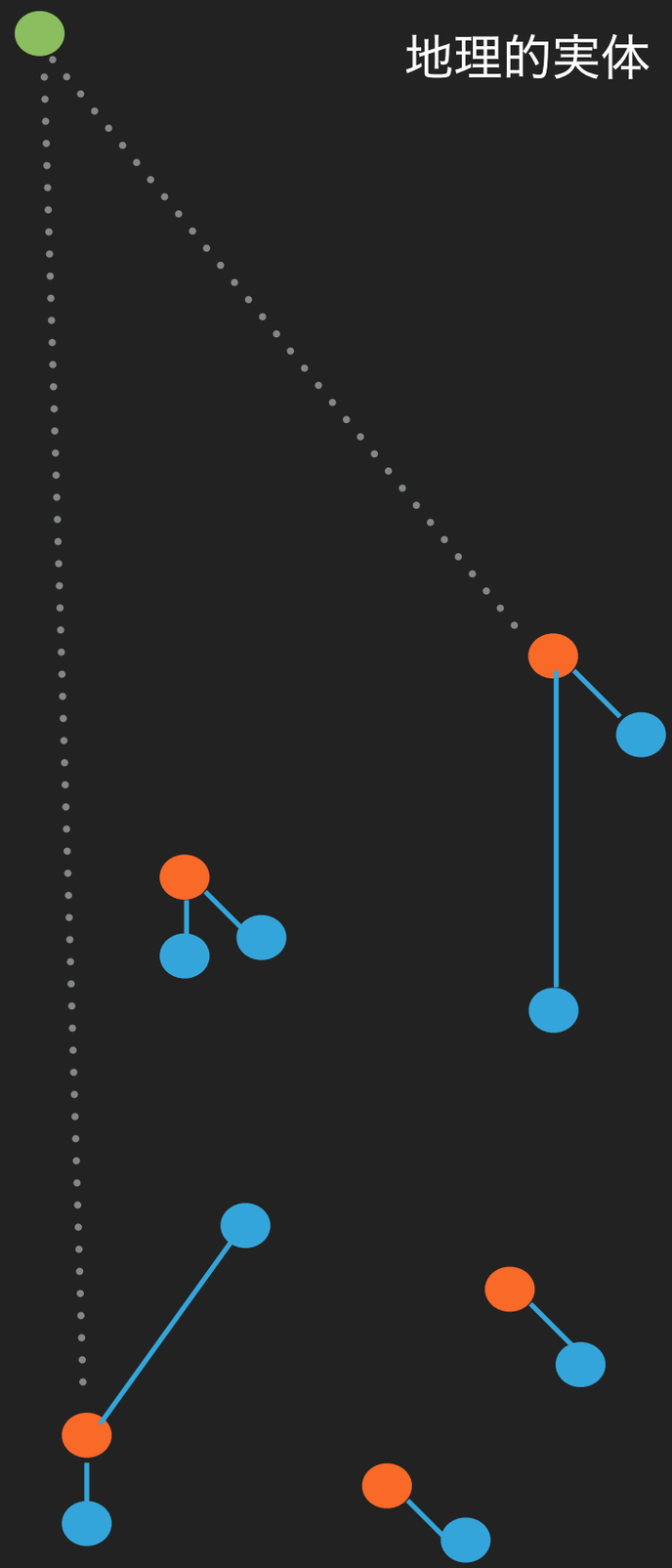
100~1000 ms

CDN

50~ms

CLIENT

地理的実体

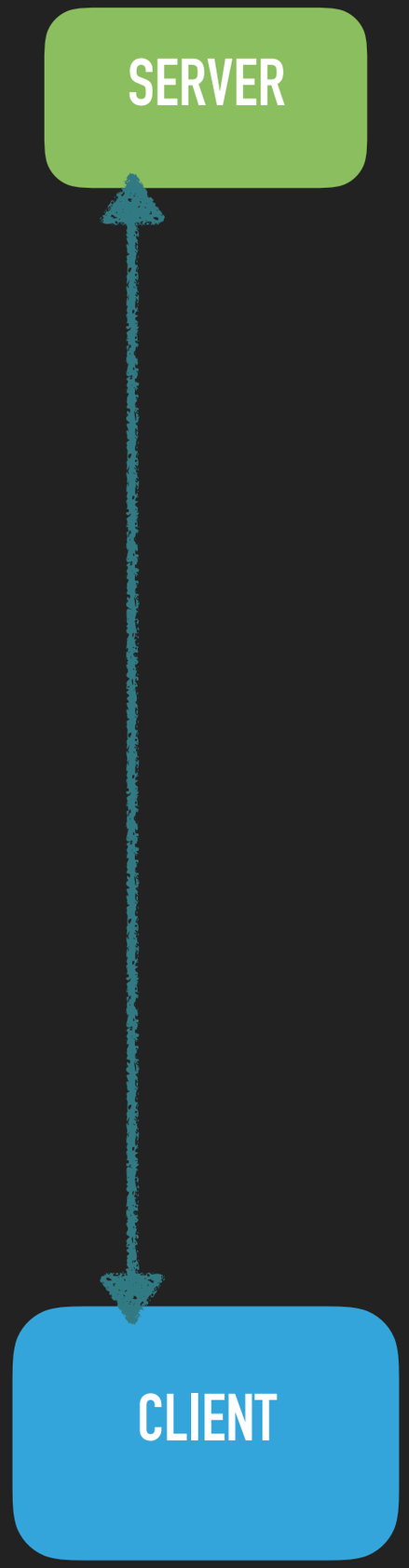


なぜ既存のアプリは遅いか

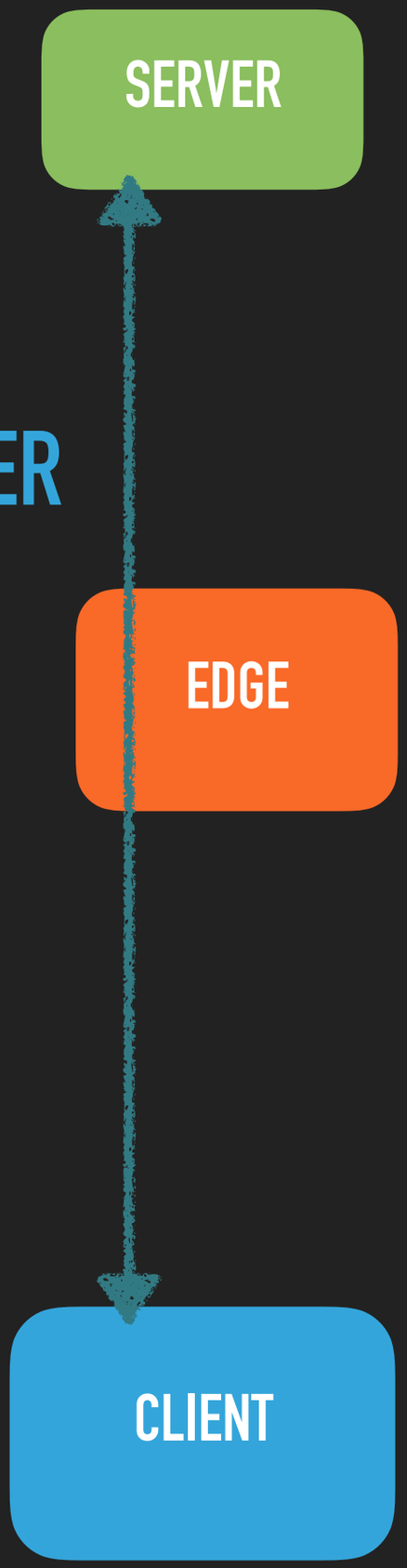
- CDN Edge は賢くない(精々 KVS+α)
- キャッシュとその破棄は本質的に難しい
- (サーバー中心の世界観だとこういう解釈にならない)

CACHE AWARE な設計

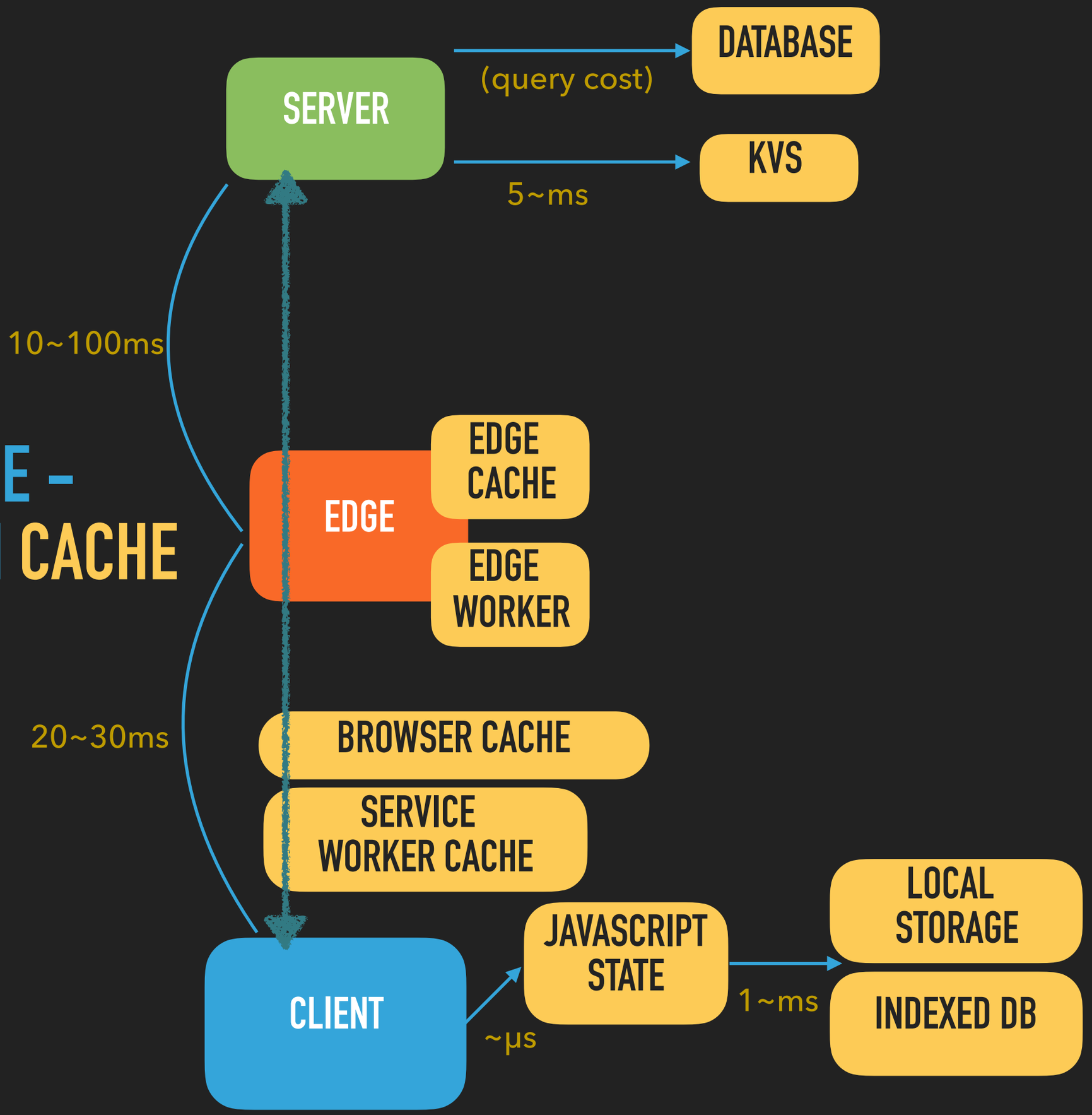
CLIENT - SERVER

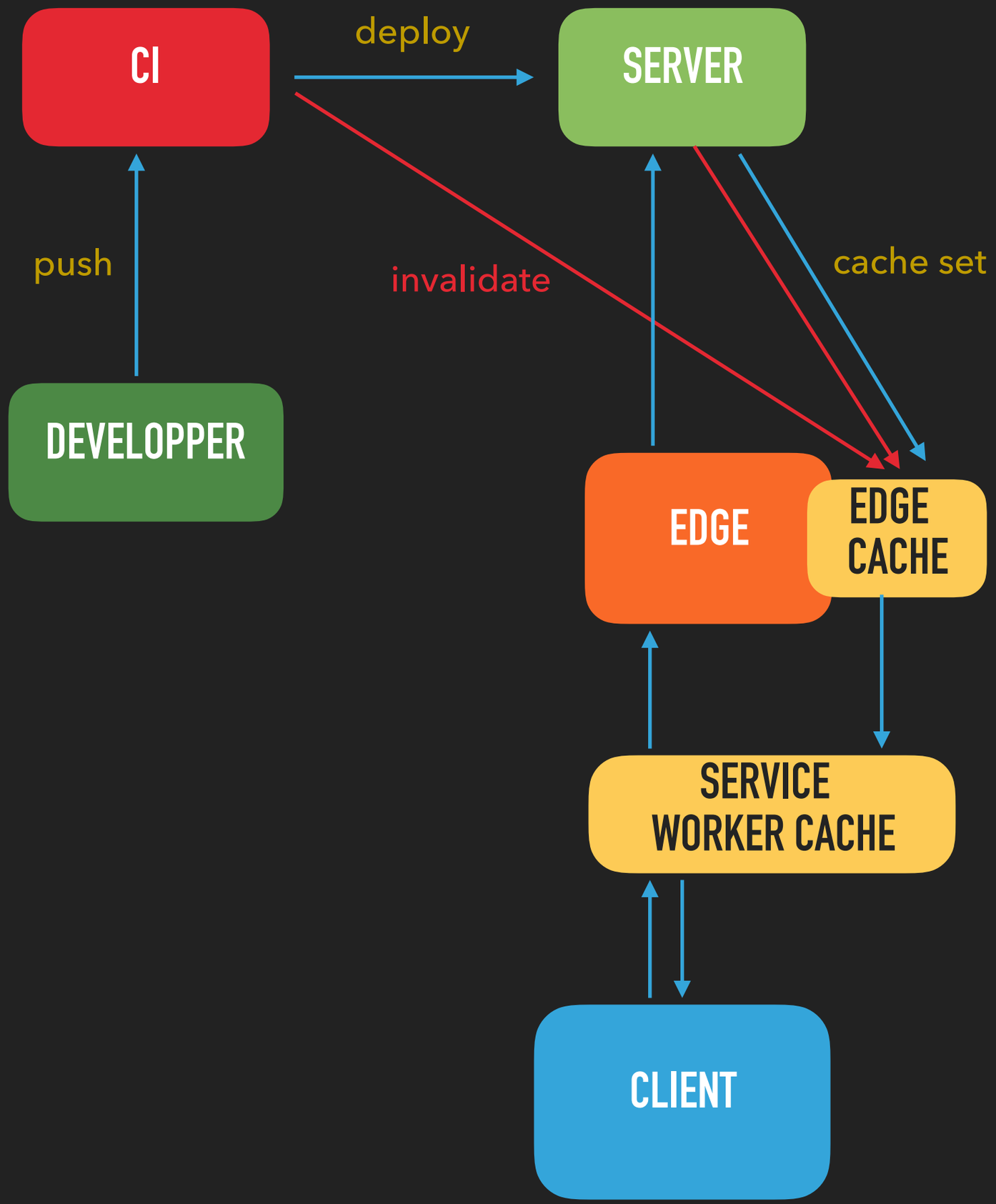


CLIENT – EDGE – SERVER



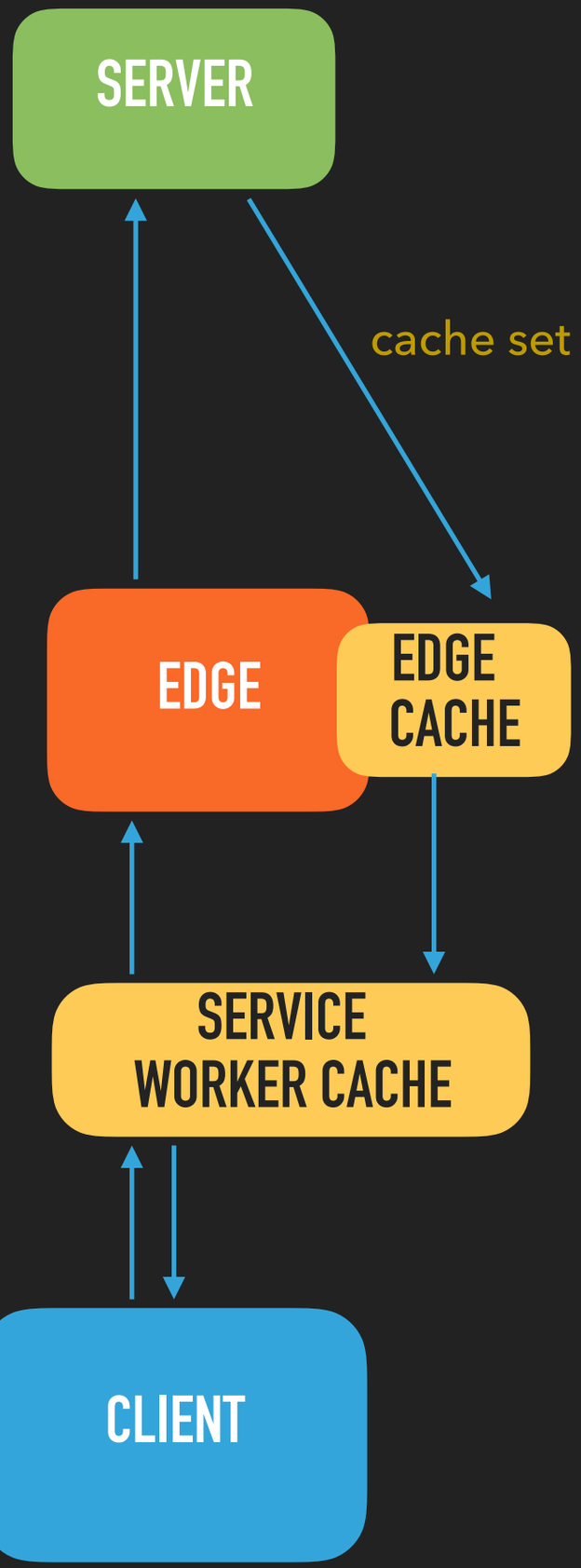
CLIENT - EDGE - SERVER WITH CACHE





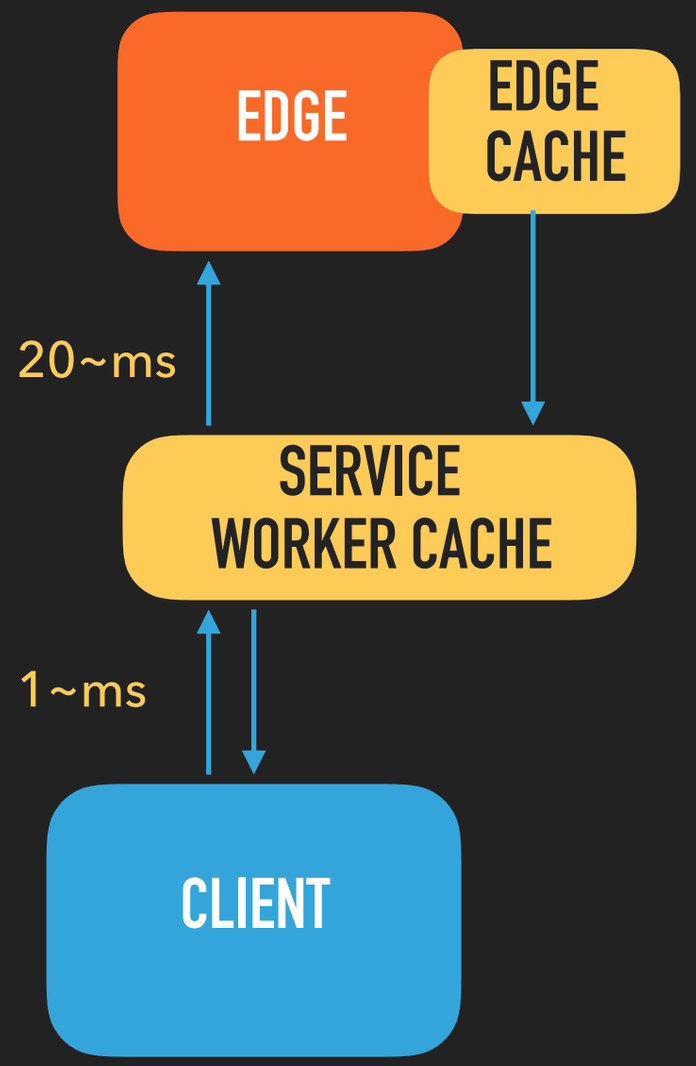
キャッシュ戦略

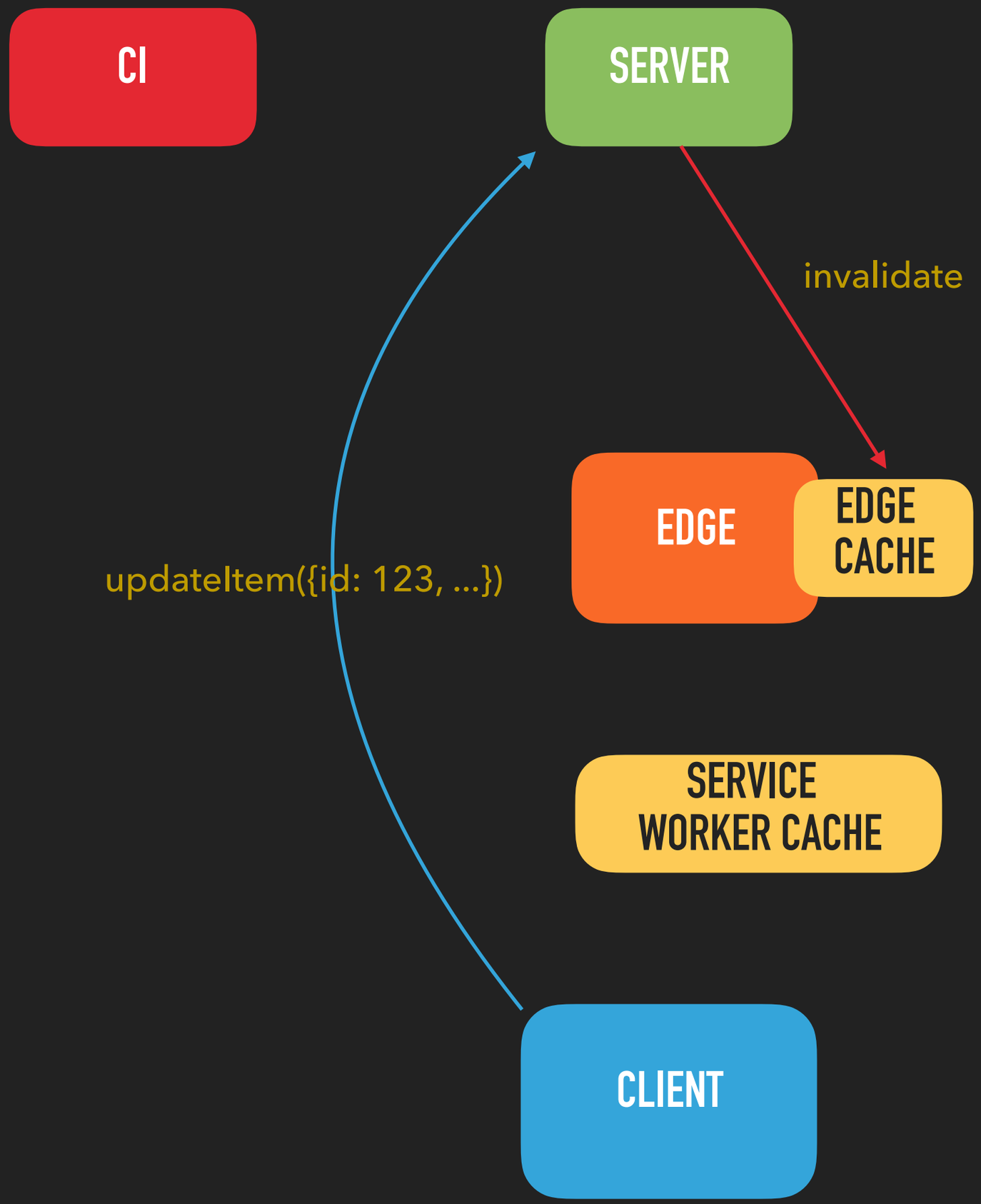
- リリースごとにキャッシュ破棄(任意)
- 更新系で依存するキャッシュを破棄
- クライアントはURLをキーに各レイヤーのキャッシュを問い合わせる
- キャッシュがなければ動的に生成



- 例: 初回
- /items/123 がほしい!
- サーバーで /items/123 の HTML を render
- Edge Cache に保存
- HTML を 返却

- 例: 二回目以降
  - /items/123 がほしい!
  - Edge Cache から HTML を返却
  - (Service Worker にキャッシュを持っていれば、それを優先)





- 例: キャッシュ破棄
- データベース上の Item(id: 123) を更新
- Edge Cache の / items/123 を破棄

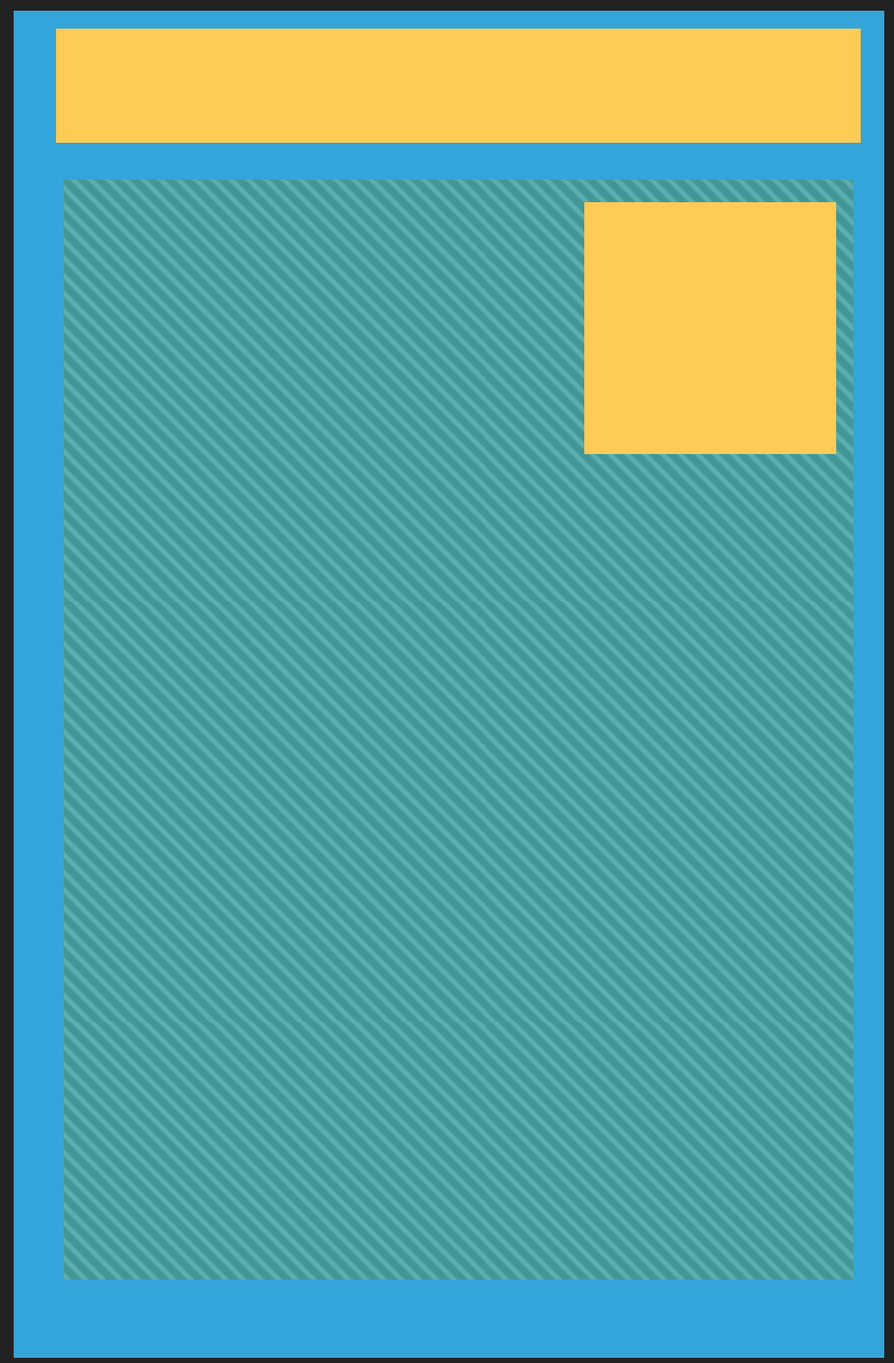
CACHE AWARE な設計(要約)

- 参照系は単純なキャッシュルール(URL)で返却
- 更新系のフックでキャッシュ破棄

設計の勘所 (1)

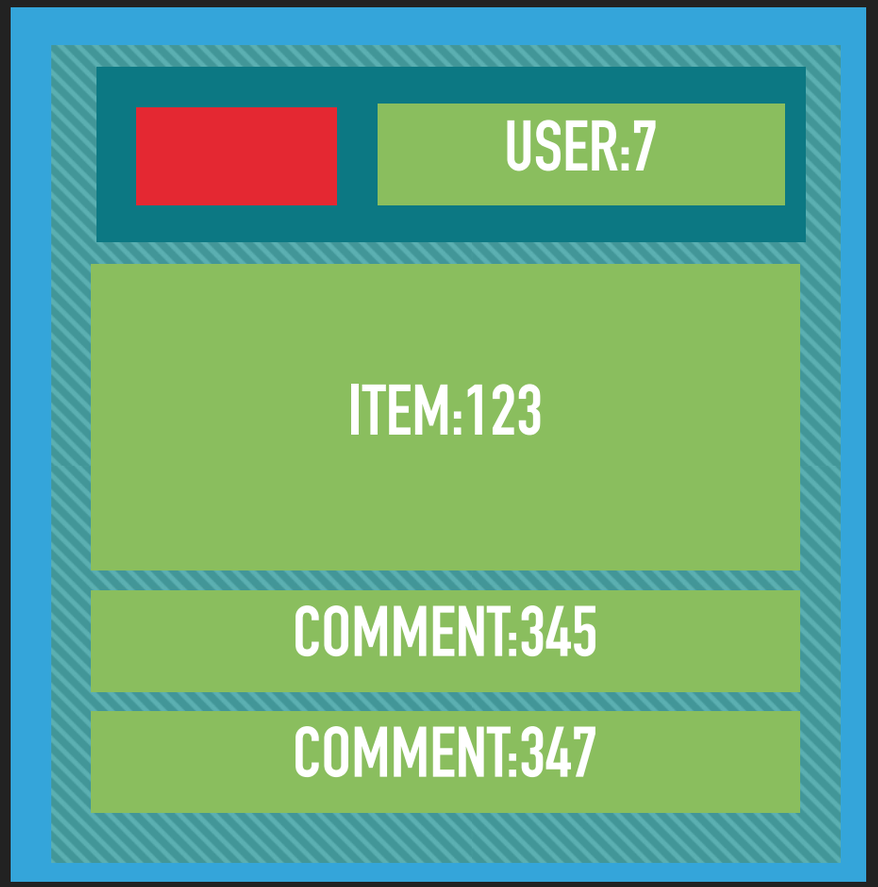
- キャッシュにセッション依存情報を使わない
- セッションに依存する

View は、初期化後に ajax で lazy に構築



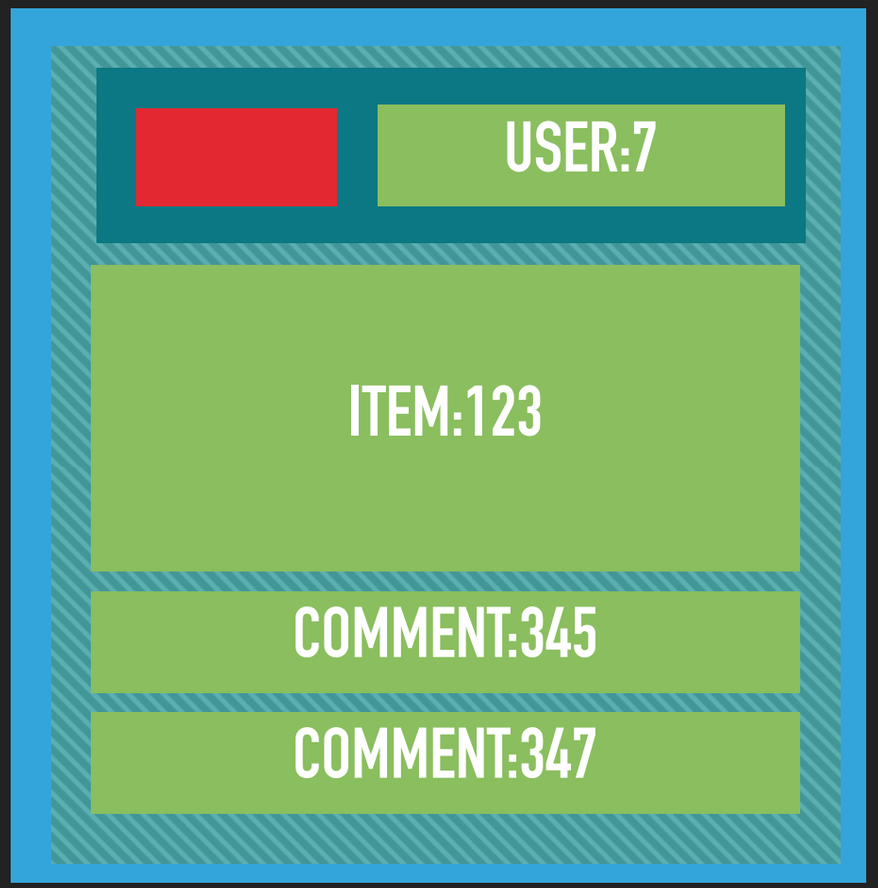
設計の勘所 (2)

- キャッシュの組成は集合で考える
- 構成要素が更新されたら破棄

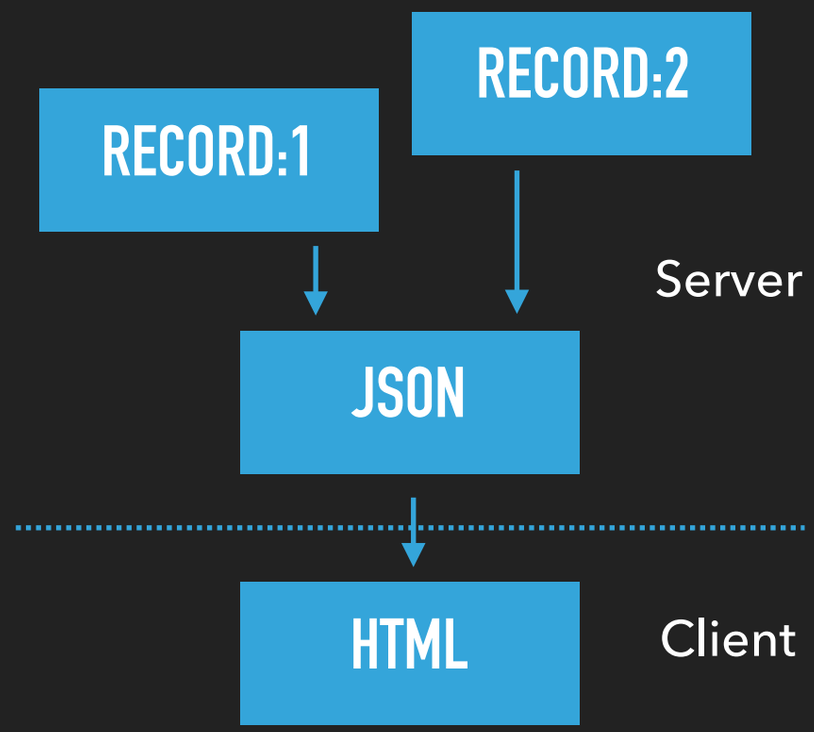


設計の勘所 (3)

- キャッシュするのは
  HTML以外でもいい

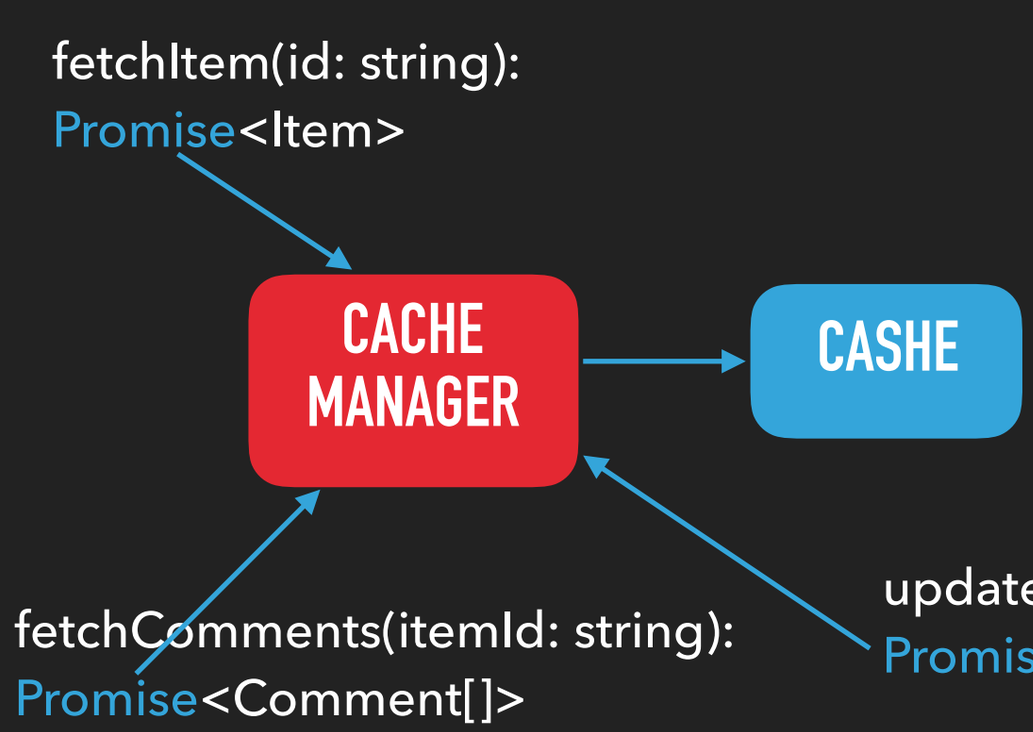


- Flux Store のJSON を
  保存など



設計の勘所 (4)

- 純粋関数と副作用を含む関数の分離 = 関数型プログラミングっぽく考える



isomorphic renderer

renderHtml(item: Item, comments: Comment[]): string

renderComponent(item: Item, comments: Comment[]): ReactNode

懸念: キャッシュ破棄が複雑すぎないか？

- Fastly: Varnish Surrogate Keys
- ORM のフックなどで管理
- ActiveRecord: after_save など
- Surrogate Keys が仕様化されてほしい 😇

複雑さへ向けて: 動的な EDGE

- AWS Lambda@Edge
- CloudFlare Workers
- Fastly Cloud Platform

Edge でJSやWASMで複雑な計算ができる = HTML を組み立てられる = SSRに便利

小規模開発者に意味があるか

- 単純に CDN = Redis, Server = MySQLと置き換えて考えてもいい
- アプリケーションが Cache Aware かどうかは、規模に関係ない

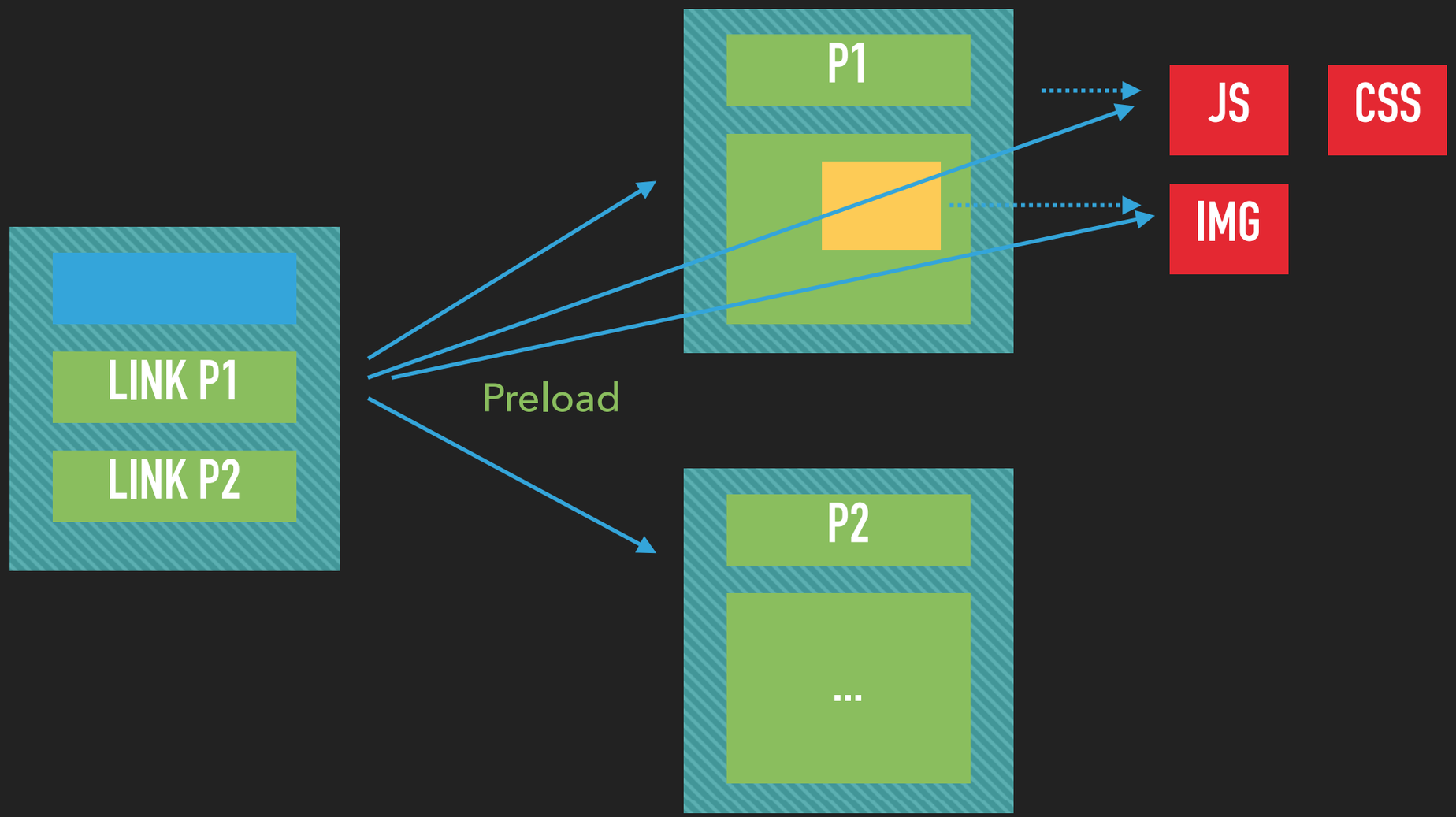
16MS以上の世界: 指針

- Lv1. ネットワークにアクセスしたら負け
- Lv2. CDNから先にアクセスしたら負け
- Lv3. DBでクエリを発行したら負け

PRELOADING

投機的キャッシュ構築

- 使うかもしれないデータを先読み



先読み関連の仕様

- https://w3c.github.io/preload/
- https://w3c.github.io/resource-hints/

- HTTP/2 Server Push
- 単純に対象を `fetch(endpoint)` 飛ばしておくだけでも効果はある

例: Next.js による preload 最適化

- ビルド時に関連アセットを収集して <link rel="preload"...> を head に挿入
- <Link to="..." prefetch> で次ページアセットを <link rel="preload" ...> に挿入

Guess.js

- https://github.com/guess-js/guess
- GA(互換)のAPIでユーザーの行動を収集
- 機械学習でユーザーの次の行動を予測
- <link rel="preload" ...> をヘッダに挿入

DEV

search

Koutaro Chikuba
@mizchi

MY READING LIST (EMPTY)

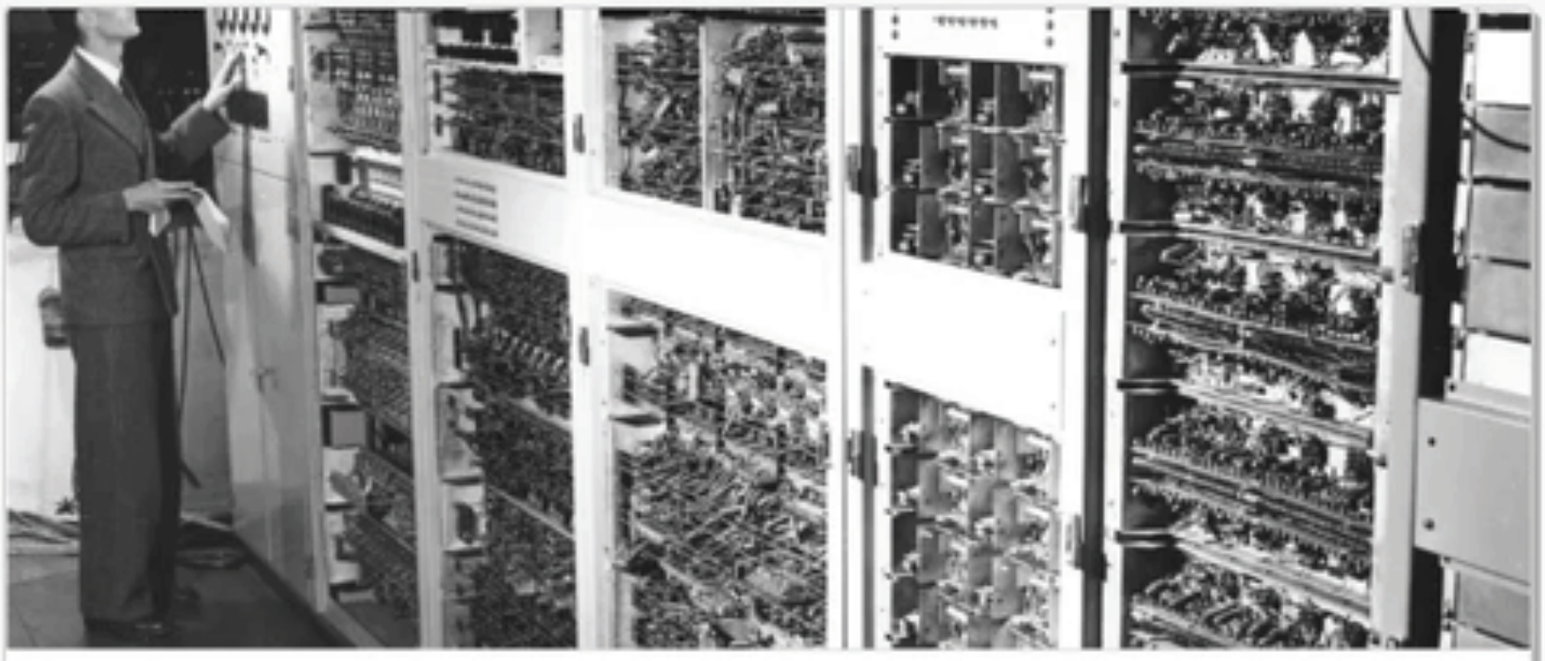
Saved Posts | Comment Activity

my tags

Follow tags to improve your feed

#javascript
#discuss
#webdev
#beginners
#react + FOLLOW
#career
#productivity
#python
#node
#opensource
#showdev

FEED | WEEK MONTH YEAR INFINITY | LATEST



How latency numbers changes from 1990 to 2020.

Sahil Rajput • Nov 23
#latency #computing

37 10 SAVE

The DEV Team

投機的キャッシュ構築の現実(1)

- 貪欲にキャッシュを収集すると、ユーザーとCDNの帯域に負荷
- 賢い先読み範囲を、コンテキストを考慮し、職人的に記述することに...

投機的キャッシュ構築の現実(2)

- キャッシュ破棄戦略はWAFの支援が必要
  - 例: next.js の preload 挿入, guess.js
- アーキテクチャが整理されてないと実現不可

身も蓋もなく言うと...

副作用を分離した綺麗なコードを書く

=> キャッシュ破棄が予測できる😇

16MS 未満の世界

16MS未満

- 途切れず連続してると感じる更新頻度
- つまりクライアント視点だとインメモリ or バックグラウンド処理

色々なマイクロチューニング…

- 初期化
  - ネットワーク最適化(今までの話)
  - スクリプティングの削減(js bundle size)
- フレーム最適化
  - レイアウト抑制(BoundingBox再計算)
  - 高頻度処理の抑制(スクロール等)
  - 初期化遅延・オフスレッド退避

スクリプティング抑制

- 単純に使うライブラリの量を減らす
- 回線速度より、貧弱なCPU(モバイル)で効果大
- オフスレッド退避(後述)

スクリプティング抑制：手段

- ES2015 Dynamic Import
- Webpack: Code Spritting
- Webpack: Dead Code Elimination

PREPACK

- facebook/prepack
- 静的解析して定数畳み込み
- (現状、コードが大幅に増える…)

Prepack

ABOUT GETTING STARTED FAQS T

Fibonacci OPTIONS

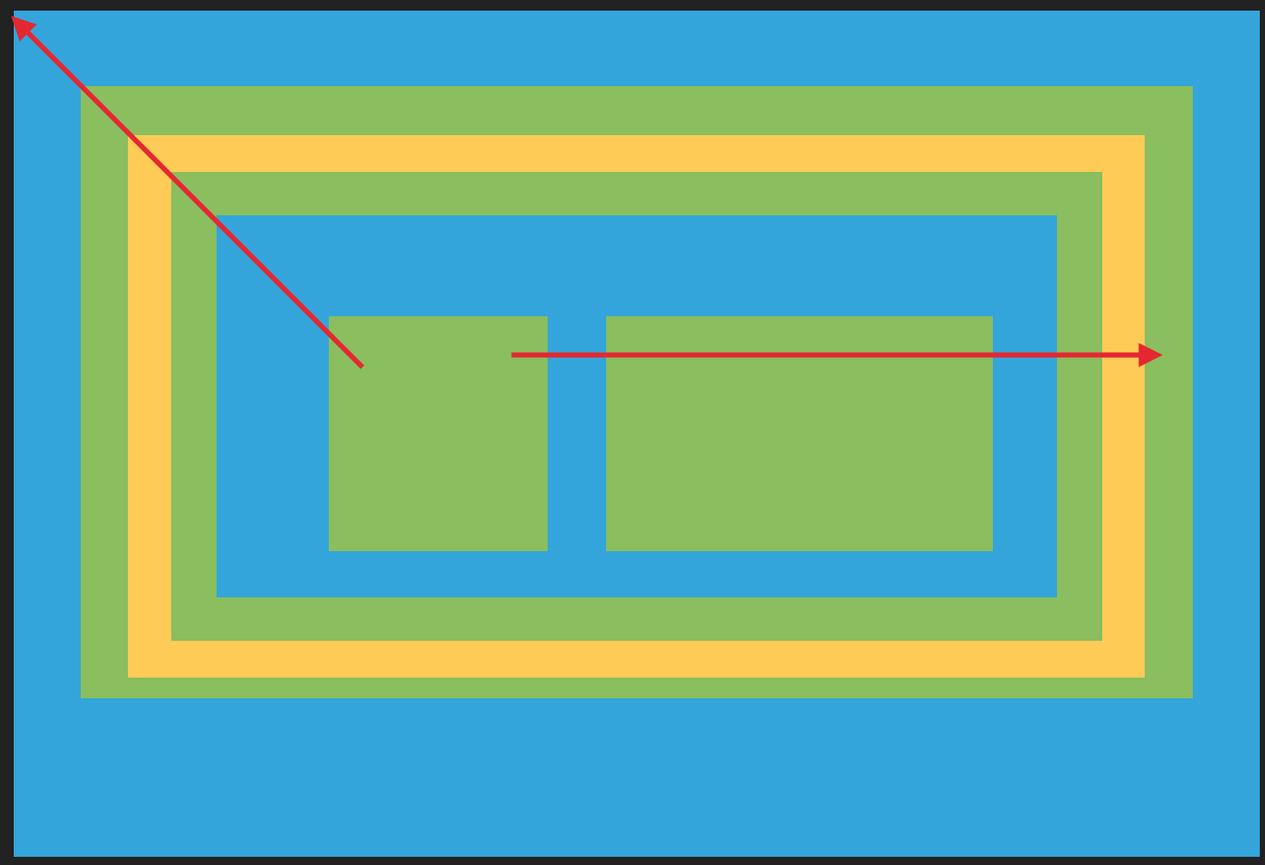
Fibonacci SAVE D

```
1  (function () {
2    function fibonacci(x) {
3      return x <= 1 ? x : fibonacci(x - 1) + fibonacci(x - 2);
4    }
5    global.x = fibonacci(10);
6  })();
```

```
1  (function () {
2    var _$0 = this;
3
4    _$0.x = 55;
5  }).call(this);
```

レイアウト抑制

- 要素幅を変更すると親方向を再計算
- 「ガタッ」や「ピクッ」という悪体験



レイアウト抑制: スケルトンスクリーン

- 領域を占めるモックで初期化/すり替え
- レイアウト再計算の抑制

https://uxplanet.org/how-to-use-animation-to-improve-ux-338819e93bdb

AMP に学ぶレイアウト抑制

- CSSは全てインライン化
  - キャッシュヒットとのトレードオフ
- 画像(amp-img)は width/height 必須 or 明示的な layout="responsive" 指定

初期化遅延/オフスレッド処理

- WebWorker でバックグラウンド処理
- スクリプティングも結果的に抑制

DEMO: MARKDOWN PREVIEW

- off thread markdown compile (22ms~)
- off thread prettier (1.2MB)

マイクロチューニングの現実

- 実際にはこれらの複合で問題が発生
- 高頻度イベント + レイアウト再計算
- フレームワーク特有の事情

低遅延環境の為の
設計パターン

設計パターン

- 楽観的UI
- クライアントファースト
- off-the-main-thread

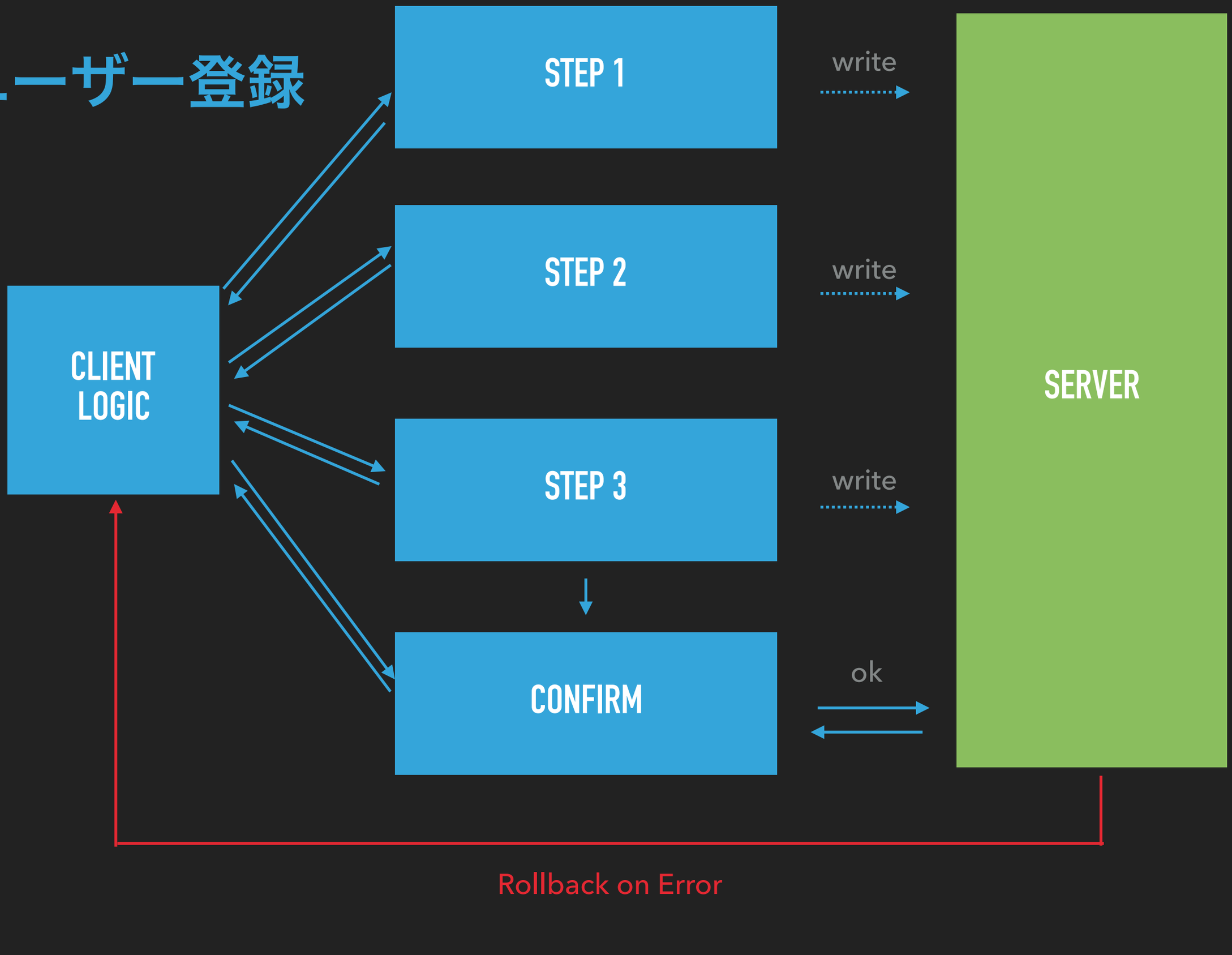
楽観的UI

楽観的UI

- 常にリクエストを成功したことにする
- 失敗時にロールバック

楽観的UI:

例: ユーザー登録



楽観的UI: いつ使えるか

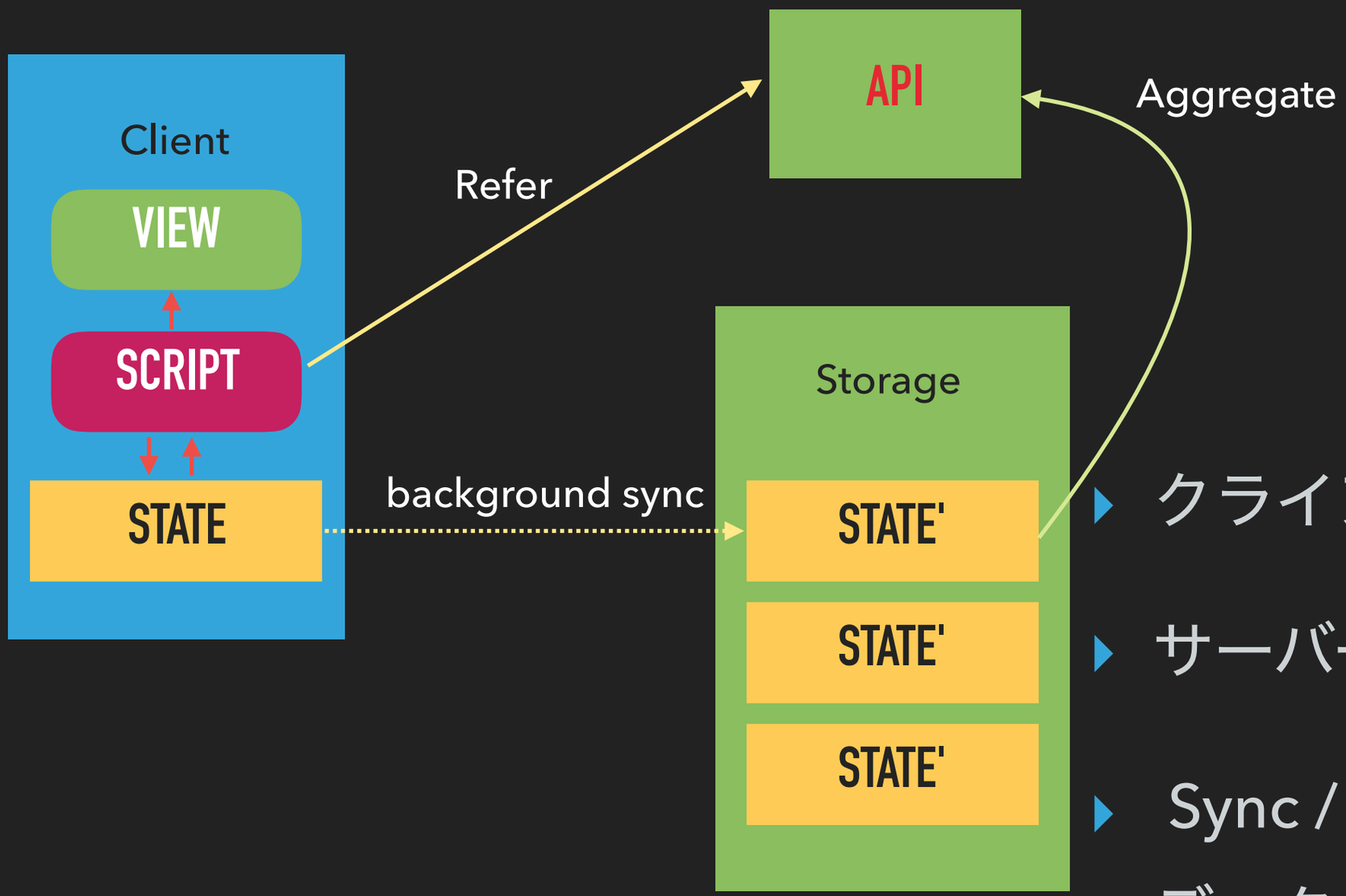
- 正常系にネットワーク上の分岐がない場合
- ログインフロー / Like Button / 記事やコメントの投稿など

クライアントファースト設計

クライアントファースト設計

- 基本的にクライアントでのデータ処理を優先する
- サーバーは単に sync されるだけ
- 別途集約系APIを用意

クライアントファースト設計



- クライアントのステートが主
- サーバーステートが従属
- Sync / API が集約する際にデータの整合性の検証

クライアントファースト設計: いつ使えるか

- 扱うデータが自己完結し、コミュニケーションが少ないもの(ゲーム・ツール)
- firebase(firestore) と相性がいい

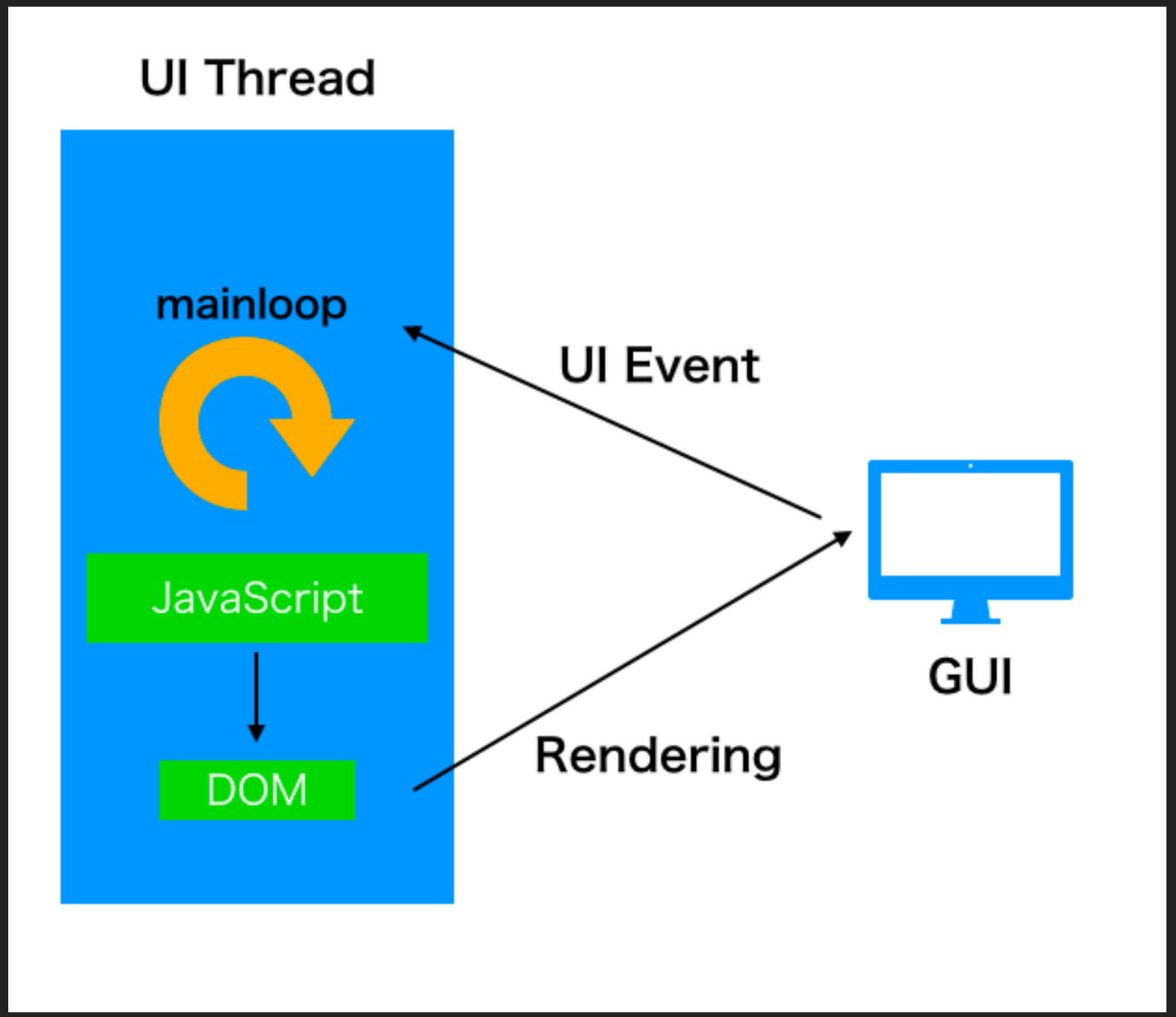
クライアントファースト設計: 問題

- チート耐性が低い
  - Web はクライアント改竄に弱い
- 真面目に突き詰めるとP2Pでの相互監視(ブロックチェーン技術)などが必要に...

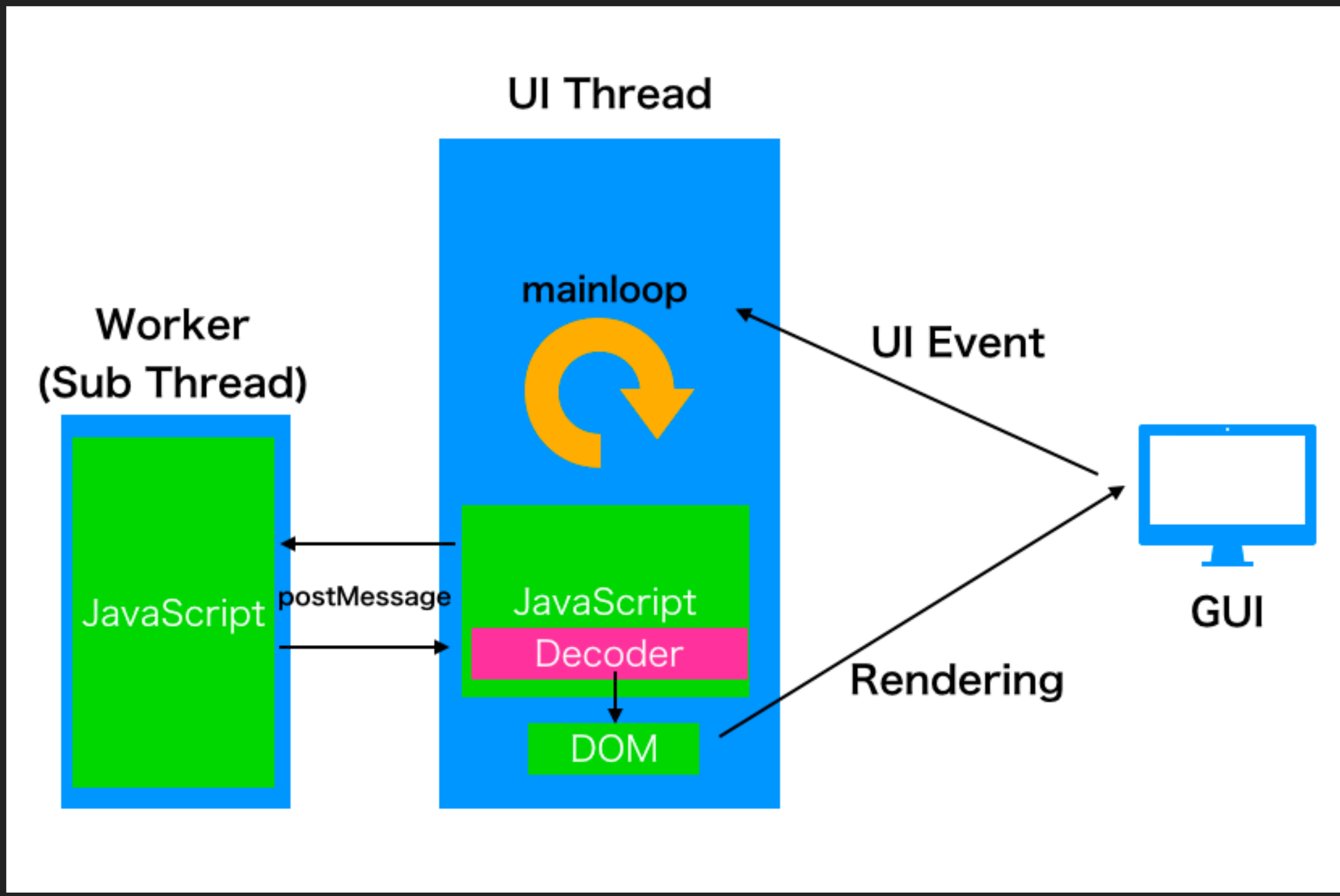
OFF THE MAIN THREAD

OFF THE MAIN THREAD

- 最近の(Chrome方面の)トレンド
- UI Thread 以外に処理をオフロード
- WebWorker を積極的に使っていく



https://mizchi.hatenablog.com/entry/2018/10/18/155448



https://mizchi.hatenablog.com/entry/2018/10/18/155448

OFF THE MAIN THREAD: 何が解決できるか

- Scripting: UI Thread が薄くなるので、JS の初期化時間を減らせる
- Universal: 自然とViewとデータを分離することに

パフォーマンス

メトリクス

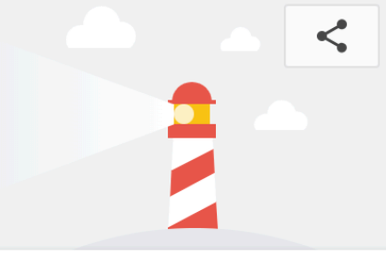
SPEED INDEX

- First Paint
- First Meaningful Paint
- First Contentful Paint
- Time to Interact

LIGHTHOUSE

- https://github.com/GoogleChrome/lighthouse
- 各種メトリクスのレポートを作成
- (PWA系を過大評価な気も...)

https://pwa.rocks/
Sep 17, 2018, 1:27 PM PDT
Emulated Nexus 5X, Simulated Fast 3G network



100	92	88	100	100
Performance	Progressive Web App	Accessibility	Best Practices	SEO

Score scale: 0-49 50-89 90-100

Performance

100

Metrics

First Contentful Paint	0.9 s	First Meaningful Paint	1.2 s
Speed Index	1.1 s	First CPU Idle	1.2 s
Time to Interactive	1.2 s	Estimated Input Latency	10 ms

Values are estimated and may vary.

Opportunities

These optimizations can speed up your page load.

	Opportunity	Estimated Savings
1	Serve images in next-gen formats	0.3 s

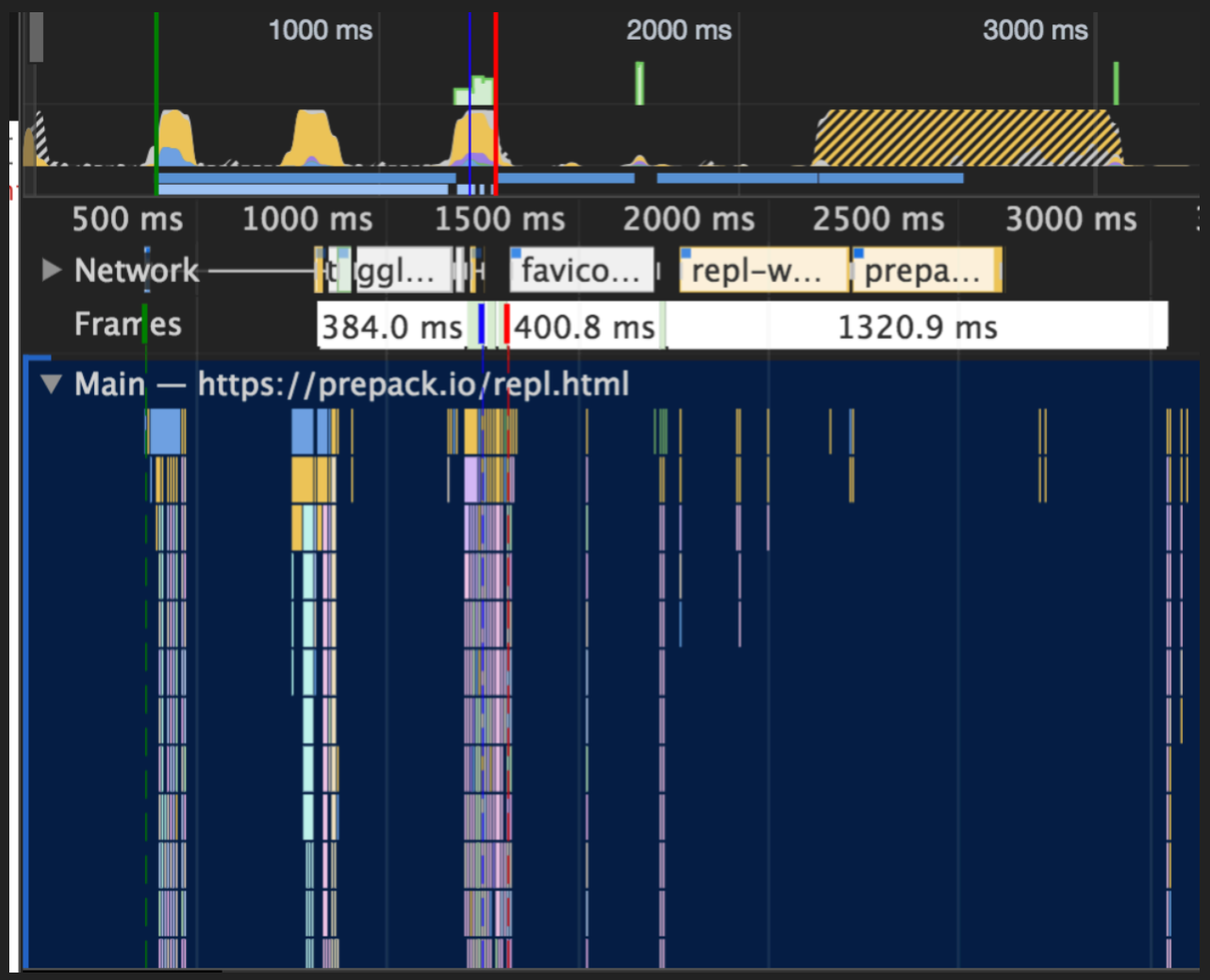
Diagnostics

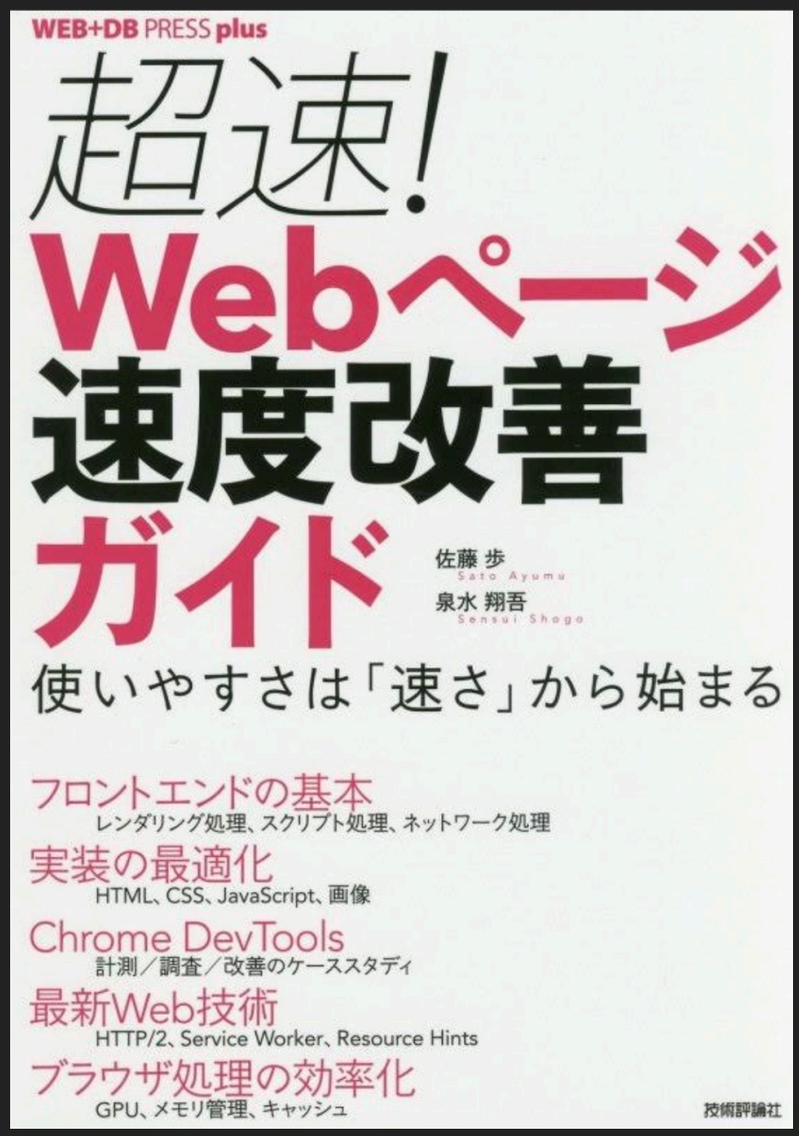
More information about the performance of your application.

1	Ensure text remains visible during webfont load	
2	Serve static assets with an efficient cache policy	57 resources found

CHROME DEVTOOLS

- とにかく触って慣れる!





最後に:

なぜ高速化するか

なぜ高速化するか

- 綺麗なアーキテクチャでないと高速化できない ≒ 高速化「できる」ことが健全なアーキテクチャの一つの証左
- 高速化そのものは結果に過ぎない

コードの綺麗さと速度

- よく誤解されるが、コードの綺麗さと速度は対立構造ではない
- アーキテクチャの生む、洗練された「余裕」こそがキャッシュ戦略のようなチューニングの余地になる

参考

- キャッシュフレンドリーなステートレスアプリケーション設計について考える https://mizchi.hatenablog.com/entry/2018/04/15/011520
- off-the-main-thread の時代 https://mizchi.hatenablog.com/entry/2018/10/02/093750

おわり