Developers Summit 2020

Webパフォーマンスガチ勢が本当に使っている技術

Pagespeed Insights 100点を獲得する高速化技術

プライム・ストラテジー株式会社
代表取締役　中村けん牛

Prime Strategy

本日お話すること

- 自己紹介
- モバイルファースト、5G時代の高速化とは？
- Web高速化の仕組み
- バックエンドの高速化
- ネットワークの高速化
- レンダリングの高速化
- 高速化の自動化

～PageSpeed Insights 100点満点を獲得するWebパフォーマンスガチ勢が本当に使っている技術を解説いたします。

資料ダウンロード

本日は原理を中心にお話しますが、項目も多いため、あらかじめ資料をダウンロードいただき、詳細を確認しながら聞いていただけるとより理解を深めていただけると思います。

資料ダウンロードは

https://www.prime-strategy.co.jp/resource/pdf/DevelopersSummit2020.pdf

@kusanagi_saya
https://twitter.com/kusanagi_saya

論より証拠

論より証拠～デモンストレーション

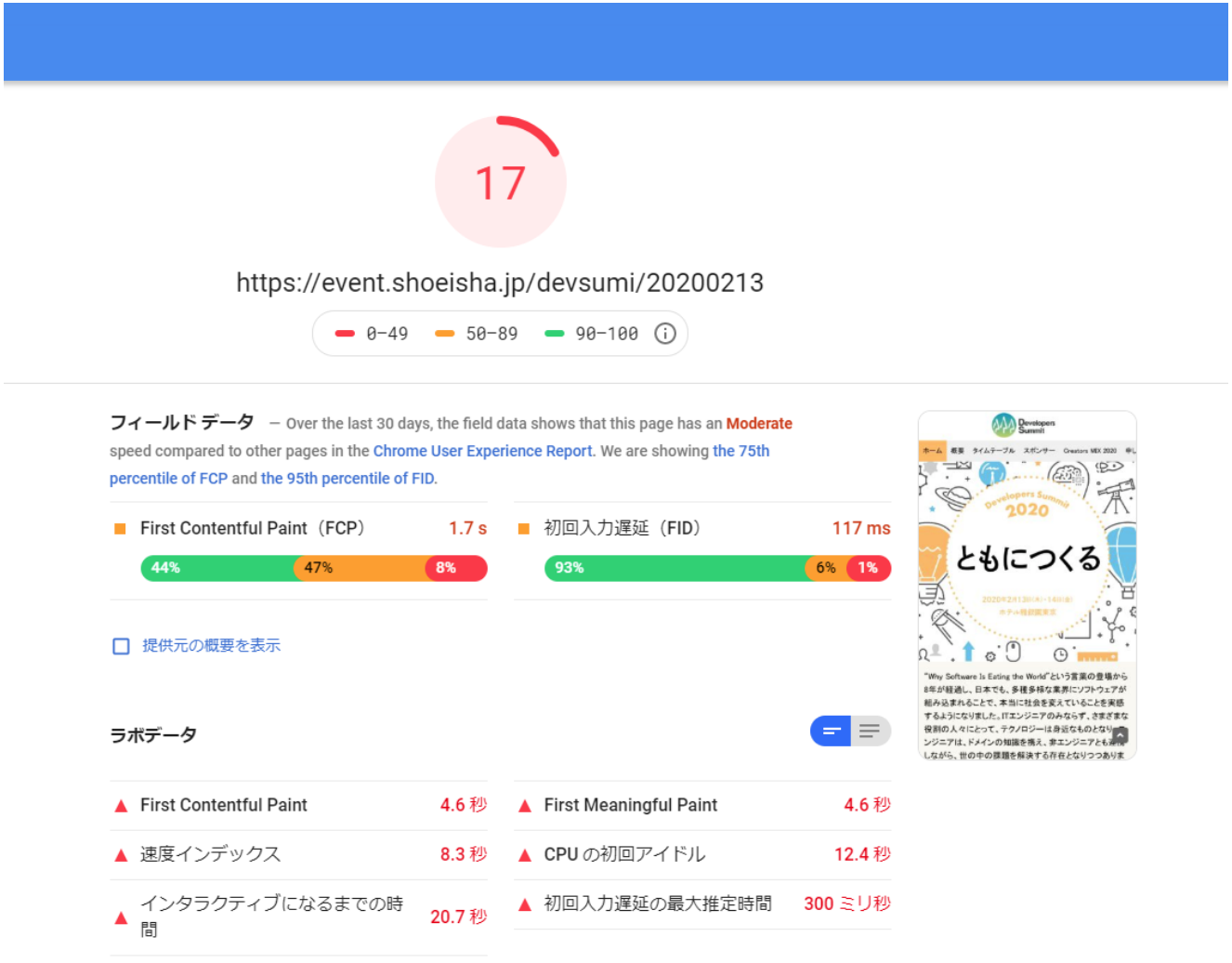
https://event.shoeisha.jp/devsumi/20200213

今から高速化してみます

“Why Software Is Eating the World”という言葉の登場から8年が経過し、日本でも、多種多様な業界にソフトウェアが組み込まれることで、本当に社会を変えていることを実感するようになりました。ITエンジニアのみならず、さまざまな役割の人々にとって、テクノロジーは身近なものとなり、エンジニアは、ドメインの知識を携え、非エンジニアとも連携しながら、世の中の課題を解決する存在となりつつあります。

論より証拠～デモンストレーション

https://event.shoeisha.jp/devsumi/20200213



論より証拠～デモンストレーション

100

https://event_d_shoeisha_d_jp_s_devsumi_s_20200213.o20.jp/devsumi/20200213?pst=stg

0-49　50-89　90-100

フィールド データ – Chrome ユーザー エクスペリエンス レポートには、このページの実際の速度データが十分にありません。

Origin Summary – Chrome ユーザー エクスペリエンス レポートには、この提供元の実際の速度データが十分にありません。

ラボデータ

First Contentful Paint	1.3 秒	First Meaningful Paint	1.3 秒
速度インデックス	1.5 秒	CPU の初回アイドル	1.3 秒
インタラクティブになるまでの時間	1.3 秒	初回入力遅延の最大推定時間	80 ミリ秒

高速化デモサイト（1週間ほど公開）

https://event_d_shoeisha_d_jp_s_devsumi_s_20200213.o20.jp/devsumi/20200213?pst=stg

ID　：　devsumi
Pass：　2020

Microsoft Azure

自己紹介

自己紹介

プライム・ストラテジー
代表取締役
中村 けん牛

代表書籍およびWEB掲載コラム

『詳解 WordPress』
（出版社：株式会社オライリー・ジャパン）

@IT（アットマークアイティ）
「とにかく速いWordPress」
https://www.atmarkit.co.jp/ait/series/2117/

1971年栃木県生まれ。中学1年生で電波新聞社の『マイコンBASICマガジン』にプログラムを寄稿して以来、プログラミング歴38年。早稲田大学法学部を卒業後、野村證券に入社。公認会計士第二次試験合格。2002年にプライム・ストラテジー株式会社を設立、代表取締役に就任する。執筆監訳書籍に『WordPressの教科書』シリーズ（SBクリエイティブ）、『WordPressによるWebアプリケーション開発』（オライリージャパン）、『詳解 WordPress』（オライリージャパン）など多数。

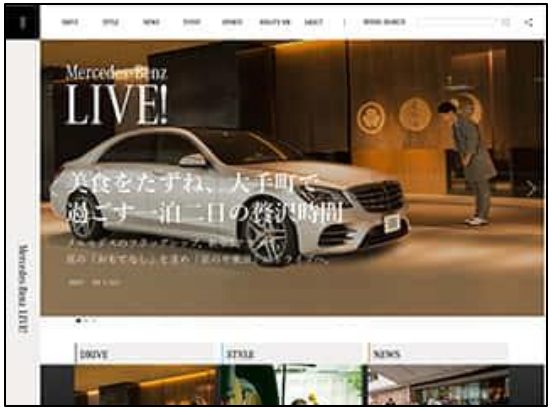
KUSANAGIスタックの開発

高速化ソリューションを開発する高速化ガチ勢

KUSANAGI Powered by Prime Strategy	高速・セキュアなサーバOS KUSANAGI https://kusanagi.tokyo/
WEXAL® Page Speed Technology	高速化エンジン WEXAL® Page Speed Technology https://www.wexal.jp/
戦略AI David	（高速化）戦略AI ONIMARU® David

実績

読売新聞社



メルセデス・ベンツ日本株式会社

株式会社ジャパンタイムズ

株式会社松屋フーズ

株式会社ファミリーマート

青山学院

（順不同・敬称略）

モバイルファースト、5G時代の高速化とは？

Webサイト高速化が求められる背景

速度とコンバージョン、収益に関する見解

ユーザーは短時間で求める情報を得ることを欲している

Google

モバイルサイトの読み込みに3秒以上かかると、

訪問者の53%が離脱します。

"Top 12 marketing insights from 2017 to carry you into 2018"(Think with Google) より引用

Google

ページの読み込み速度が1秒から5秒に増えると、

モバイルサイトの訪問者の直帰率は90%増えます。

グーグル / Find out how you stack up to new industry benchmarks for mobile page speed より引用

Webサイト高速化が求められる背景

- Google モバイルファーストインデックス（MFI）

  モバイル版のページをインデックスやランキングに使用

  https://webmaster-ja.googleblog.com/2018/03/rolling-out-mobile-first-indexing.html

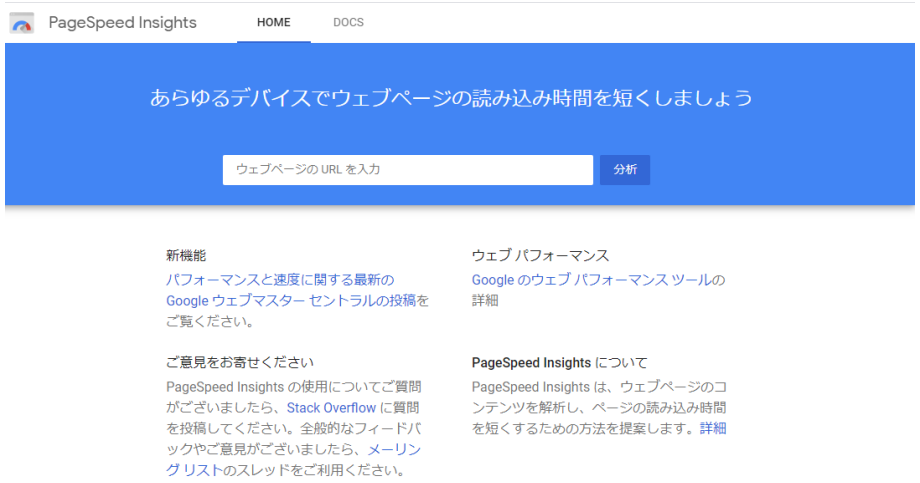
- Googleスピードアップデート

  ページの読み込み速度をモバイル検索のランキング要素に使用

  https://webmaster-ja.googleblog.com/2018/01/using-page-speed-in-mobile-search.html

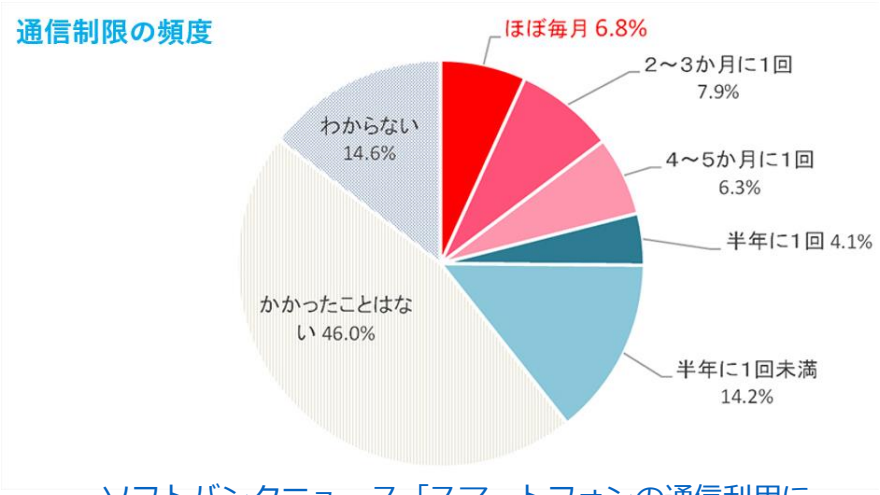
Google PageSpeed Insights

ユーザーエクスペリエンスの指標のひとつ



モバイルファースト、5G時代の高速化

5Gの時代
＝ネットワークの高速化



ソフトバンクニュース「スマートフォンの通信利用に関する実態調査」より

バックエンドとレンダリングの高速化が
より大きな差を生む時代

高速化の目的と効果

高速化の目的

UX（ ユーザーエクスペリエンス）を最大化する

→ PVの向上

→ CVRの向上

→回遊率の向上

→滞在時間の向上

また、高速化することで・・・

CPUの負荷が減る→インスタンスサイズの削減

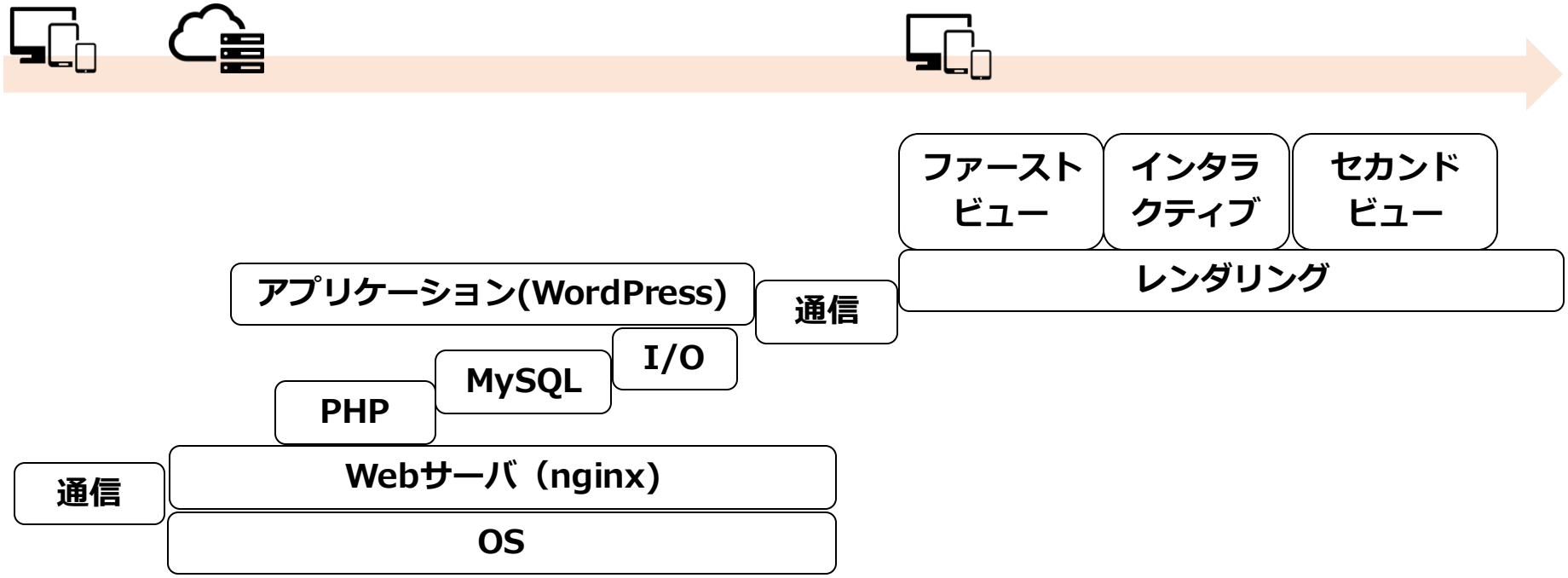
転送量が減る→ネットワーク負荷の低減

地球にも優しい

Webの高速化の仕組み

Webの高速化とは

ネットワーク・バックエンド・レンダリングすべてを高速化

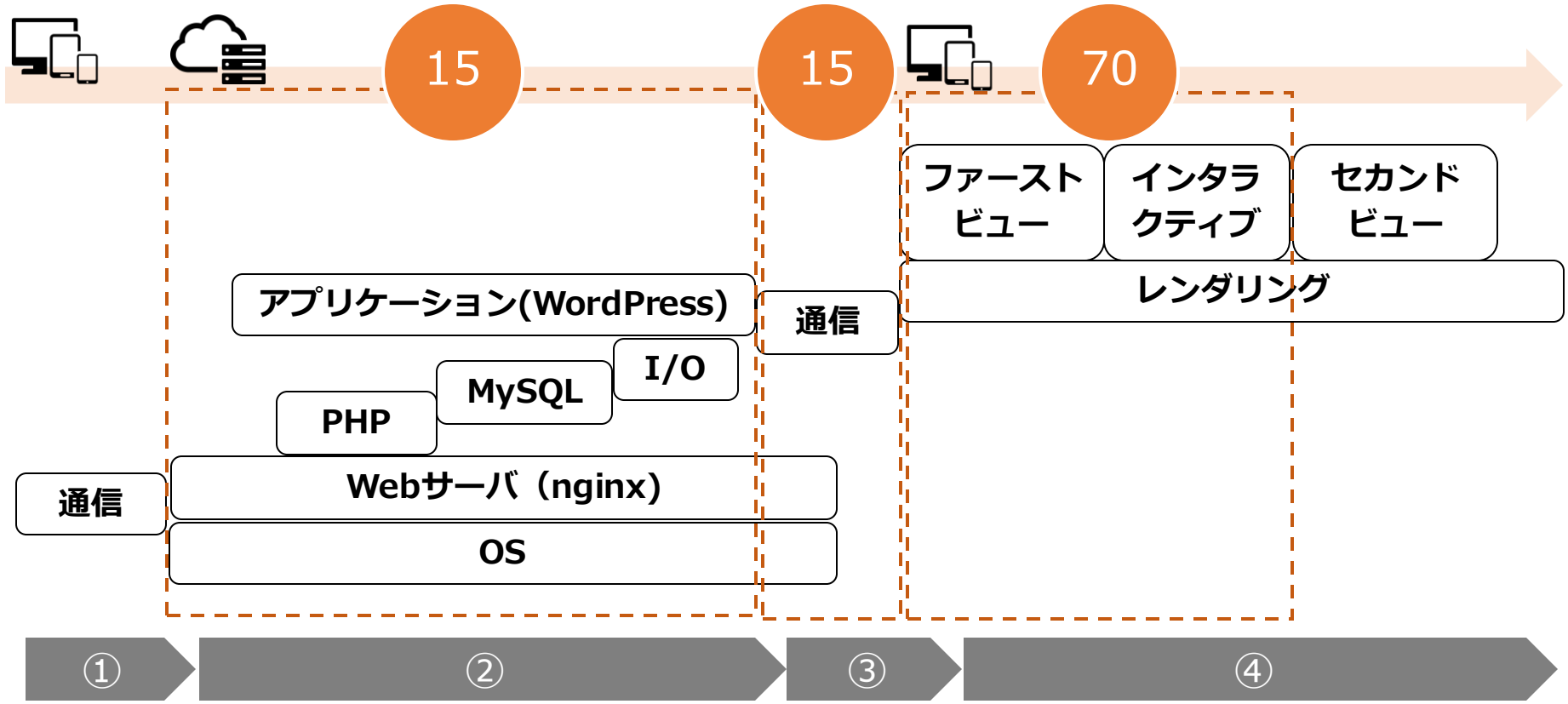


①HTTPリクエスト　②動的処理　③HTTPレスポンス　④レンダリング

nginx+PHP+MySQL構成のWordPress

Webの高速化とは

現在のGoogle PageSpeed Insightsのおよその配点



nginx+PHP+MySQL構成のWordPress

バックエンドの高速化

nginx(+lua)+PHP+MySQL構成の
WordPressアプリケーションを想定

バックエンドの高速化

バックエンド高速化の基本はキャッシュ戦略

キャッシュ戦略の要諦

C1. 一度実行した処理は再利用してCPUを動かさない

C2. メモリを効率的に利用してネットワークやストレージにアクセスさせない

C3.そもそも高速な処理系を採用する

バックエンドの高速化

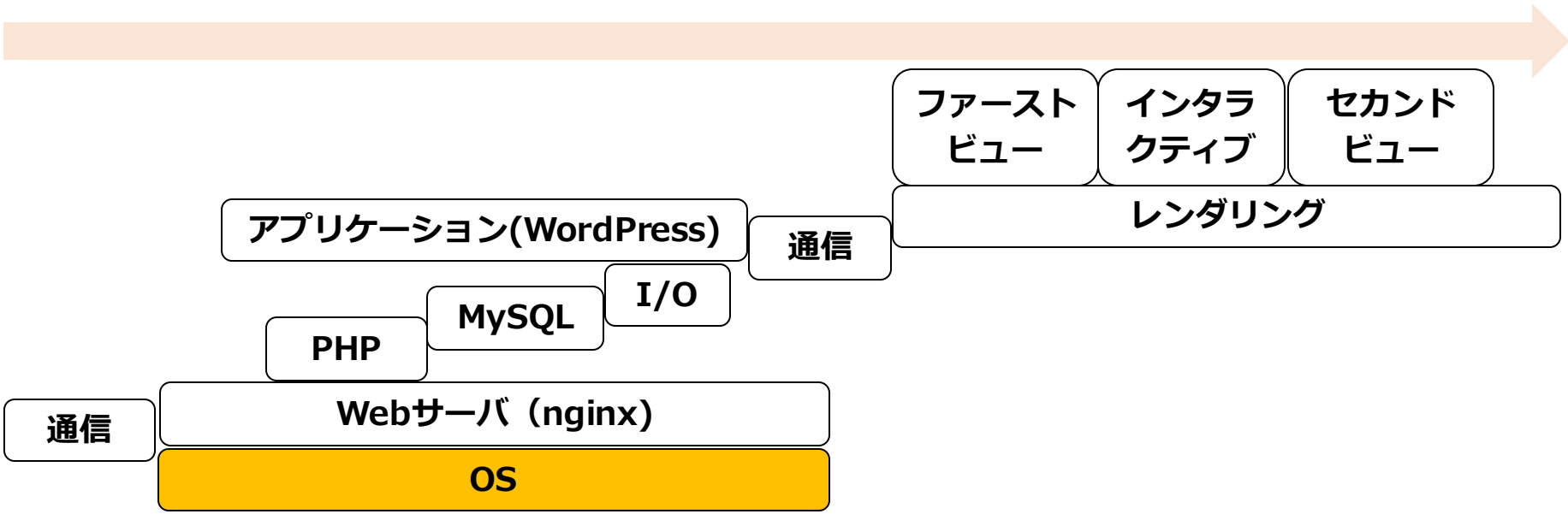
キャッシュ戦略の類型

A. 最新のデータによるアウトプットが保証されるもの
B. 過去の情報を再利用するもの

1. 直接アプリケーションの修正を要しないもの
2. 直接アプリケーションの修正を要するもの

インスタンスとOS

- 周波数の高いものを選択
- ディスク（ページ）キャッシュを利用

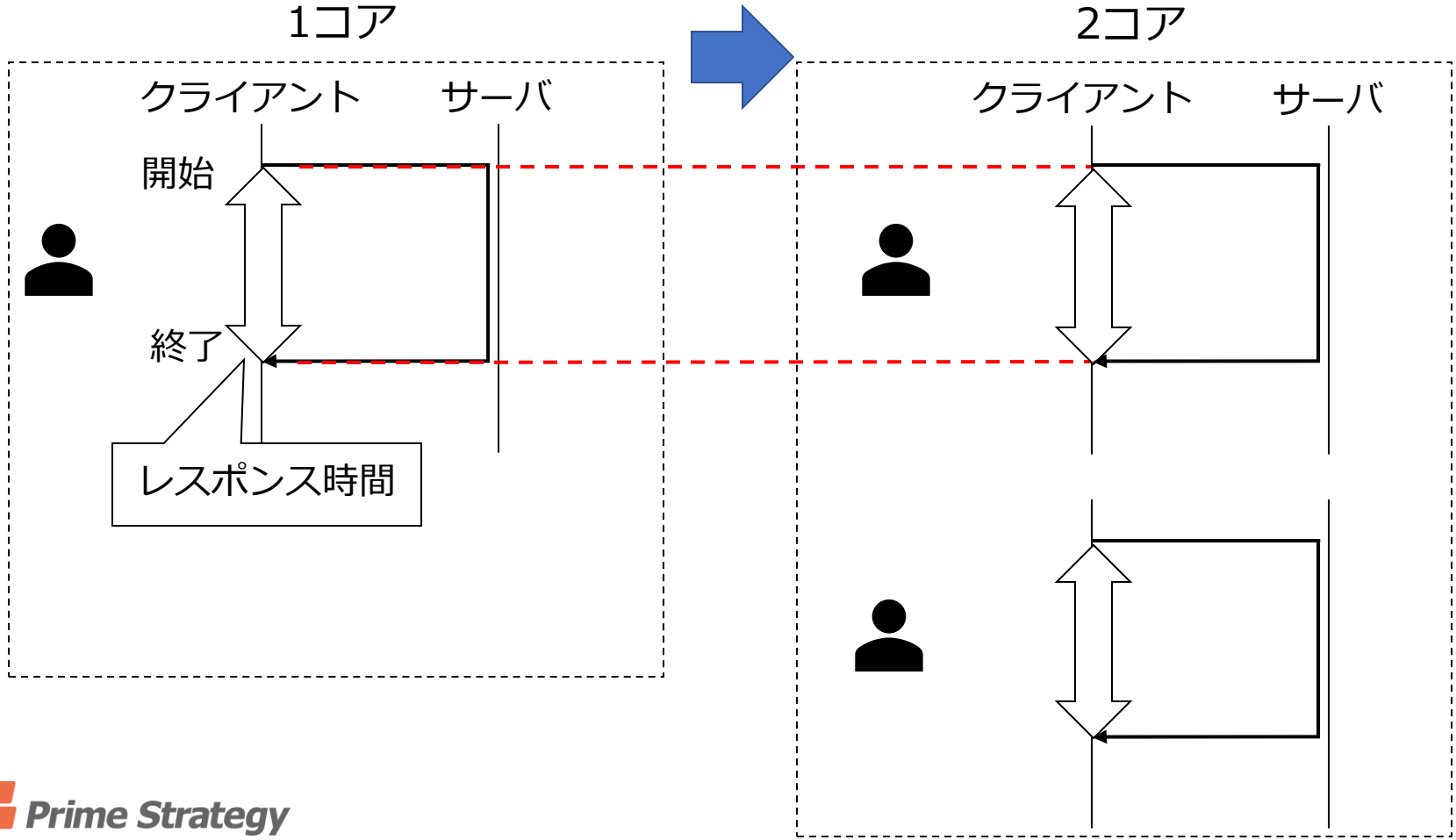


インスタンス

A1/C3

CPUのコア数より周波数の高いものを選択する

コア数を増やしてもレスポンス時間は変わらない

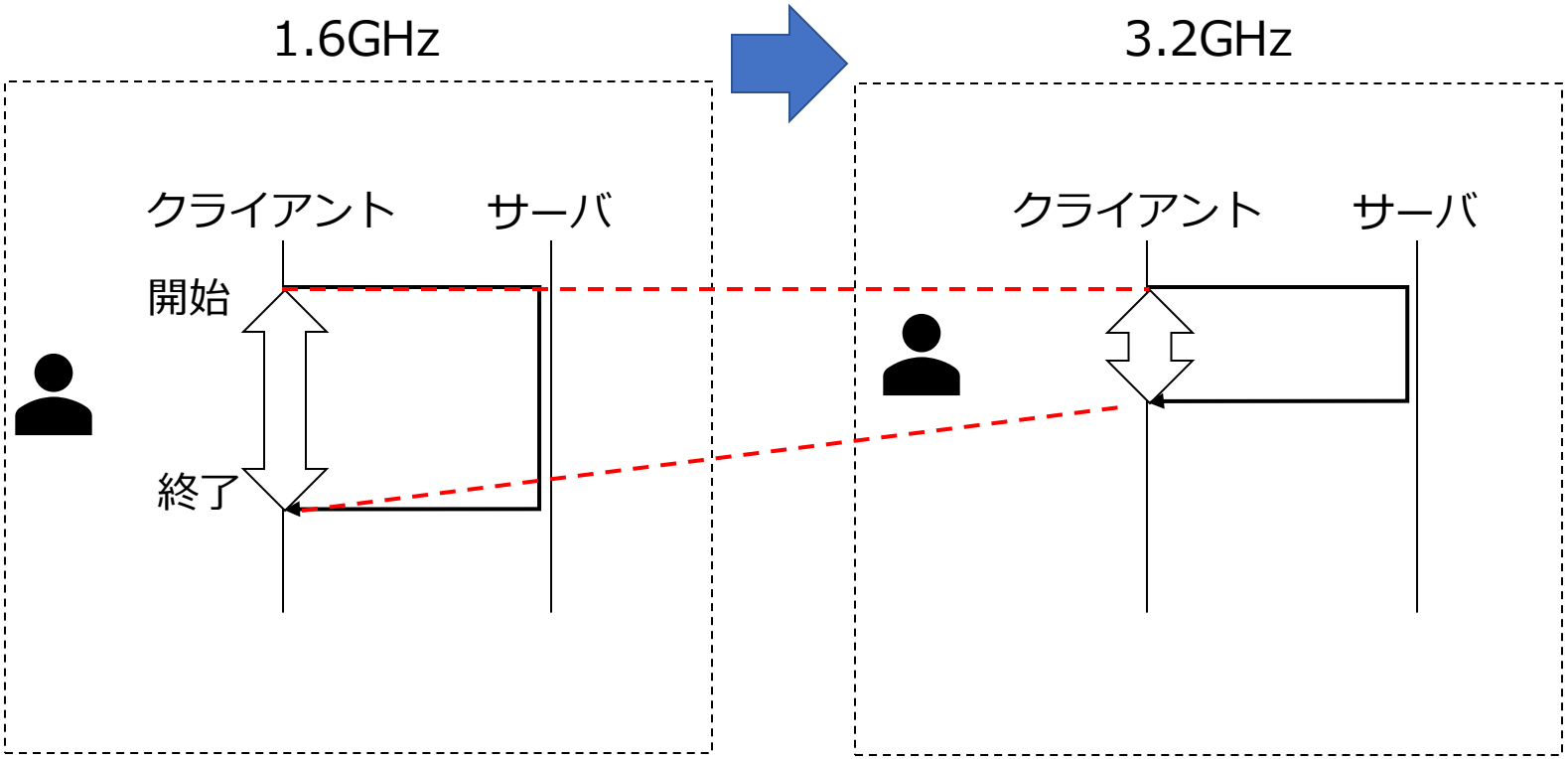


インスタンス

A1/C3

CPUのコア数より周波数の高いものを選択する

レスポンス時間を短くするには処理速度（周波数）を上げる



A1/C2

ディスク（ページ）キャッシュを活用する

ディスクキャッシュ

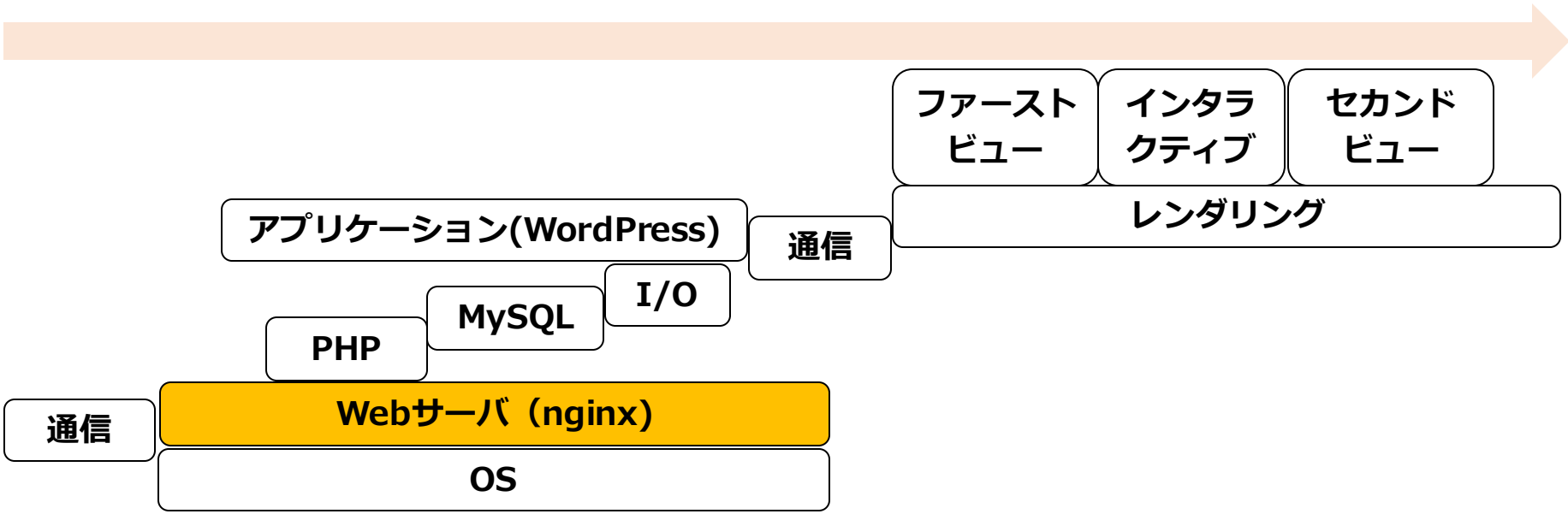
- バッファキャッシュ　ディスクブロックをキャッシュ
  →ブロックI/Oを最適化
- ページキャッシュ　ファイルのページをキャッシュ
  →ファイルI/Oを最適化


```
# free -m
              total        used        free      shared  buff/cache   available
Mem:           7802        5782         187         379        1833        1334
Swap:          2047         451        1596
```

nginx

- FastCGI Cache
- proxy cache
- Expires
- open file cache

- Lua
- LuaJIT
- PCREJIT

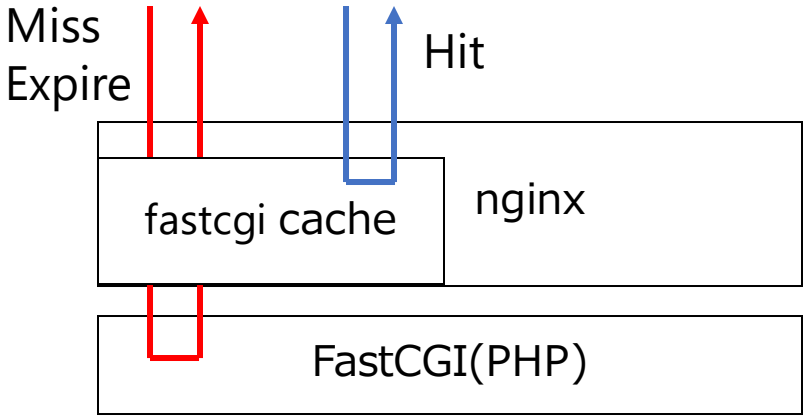


nginx

B1/C1&C2

FastCGI Cacheを利用する

nginxのFastCGIのレスポンスを直接キャッシュする




```
fastcgi_cache_path /var/cache/nginx/wordpress levels=1:2 keys_zone=wpcache:30m max_size=512M inactive=600m;
fastcgi_ignore_headers "Vary" "Cache-Control" "Expires";
```

```
fastcgi_cache_valid 200 10m;
```

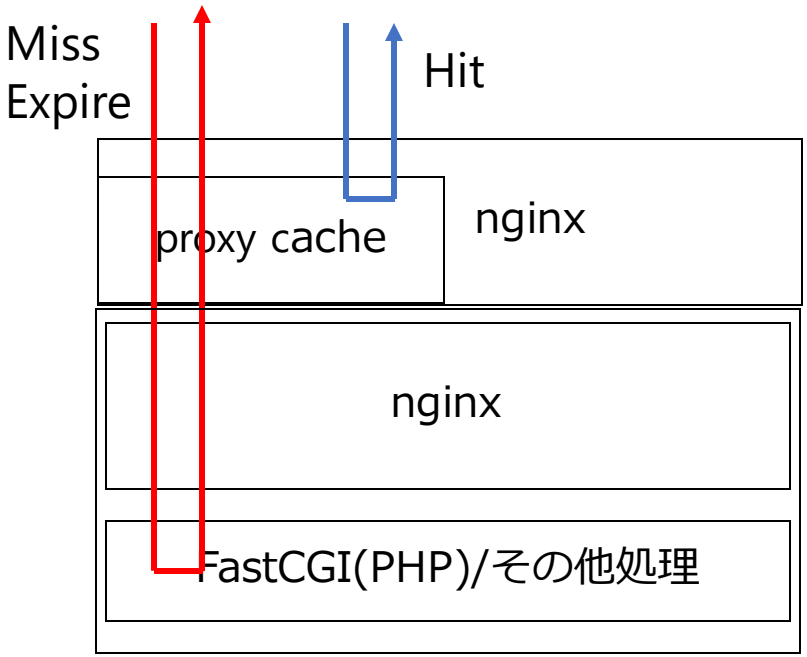
nginx

B1/C1&C2

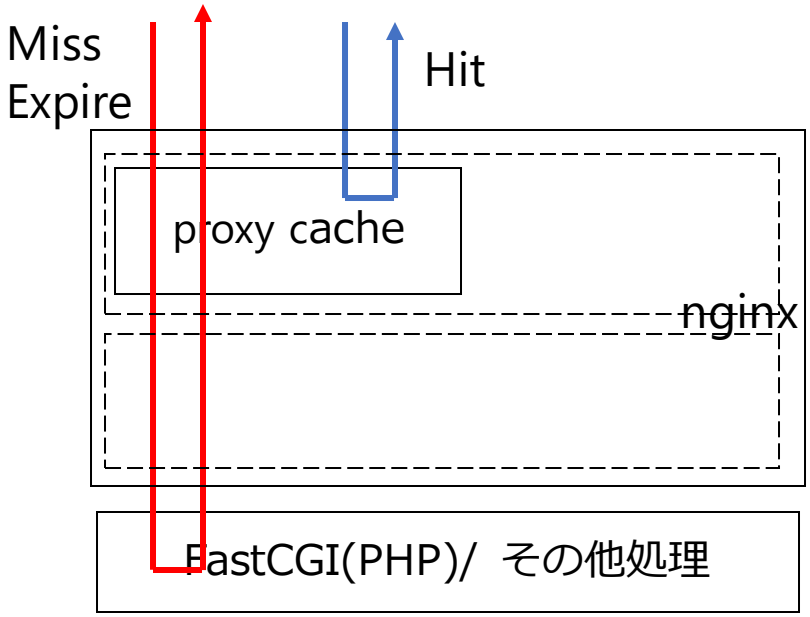
proxy cacheを利用する

バックエンドのサーバに処理を渡した結果をキャッシュ

プロキシサーバとバックエンド（upstream）のウェブサーバ



1つの設定ファイルにフロントエンドのプロキシサーバとバックエンドのウェブサーバの設定を記述



nginx

B1/C1&C2

Expiresを設定する

Expires ＝ HTTPクライアントにコンテンツのキャッシュ有効期限を返却するためのHTTPレスポンスヘッダ

- ブラウザキャッシュを利用するため高速にコンテンツを表示
- 通信を削減
- リクエストを削減し、サーバ負荷の軽減
- 「PageSpeed Insights」では、静的コンテンツに Expires ヘッダが付けられていることを評価

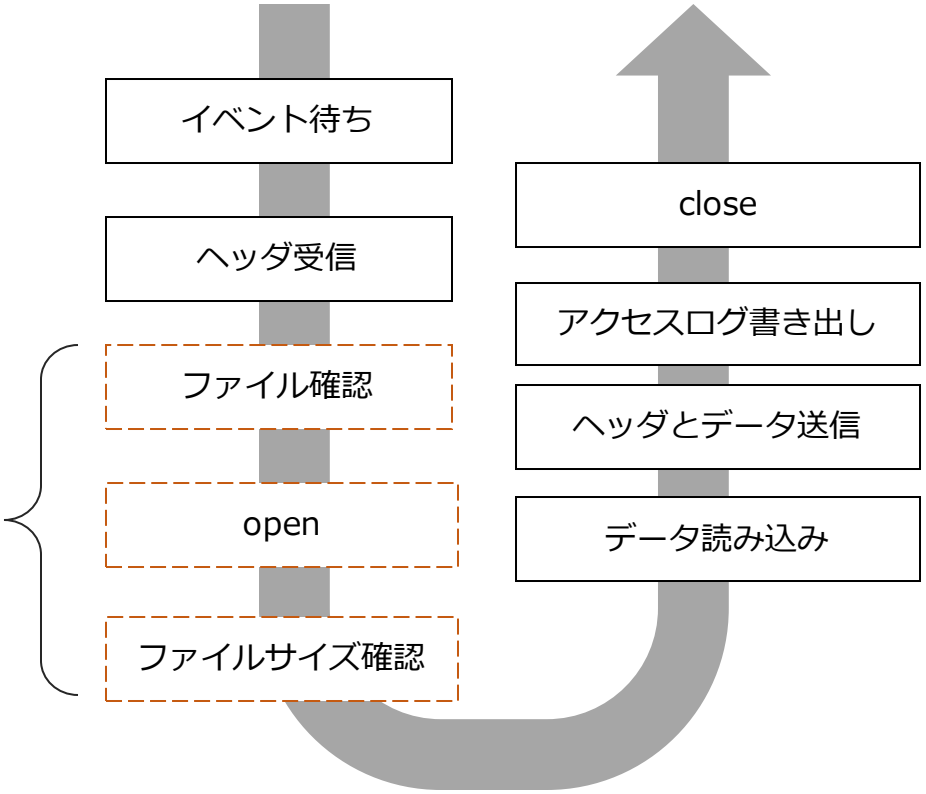
```
location ~* ¥.(jpg|jpeg|gif|png|css|js|swf|ico|pdf|svg|eot|ttf|woff)$ {
    expires 60d;
    access_log off;
}
```

nginx

B1/C2

open file cacheを設定する

ファイルを閉じる処理を遅らせ、キャッシュすることで
次のアクセス時にファイルを開く処理を省略



特定のファイルにアクセスが集中する場合に有効
CSSファイルなど

```
open_file_cache max=100000 inactive=20s;
open_file_cache_valid 30s;
open_file_cache_min_uses 2;
open_file_cache_errors on;
```

nginx

A1/C1

Luaによるアプリケーション化

lua-nginx-module → nginxの処理フローにフック

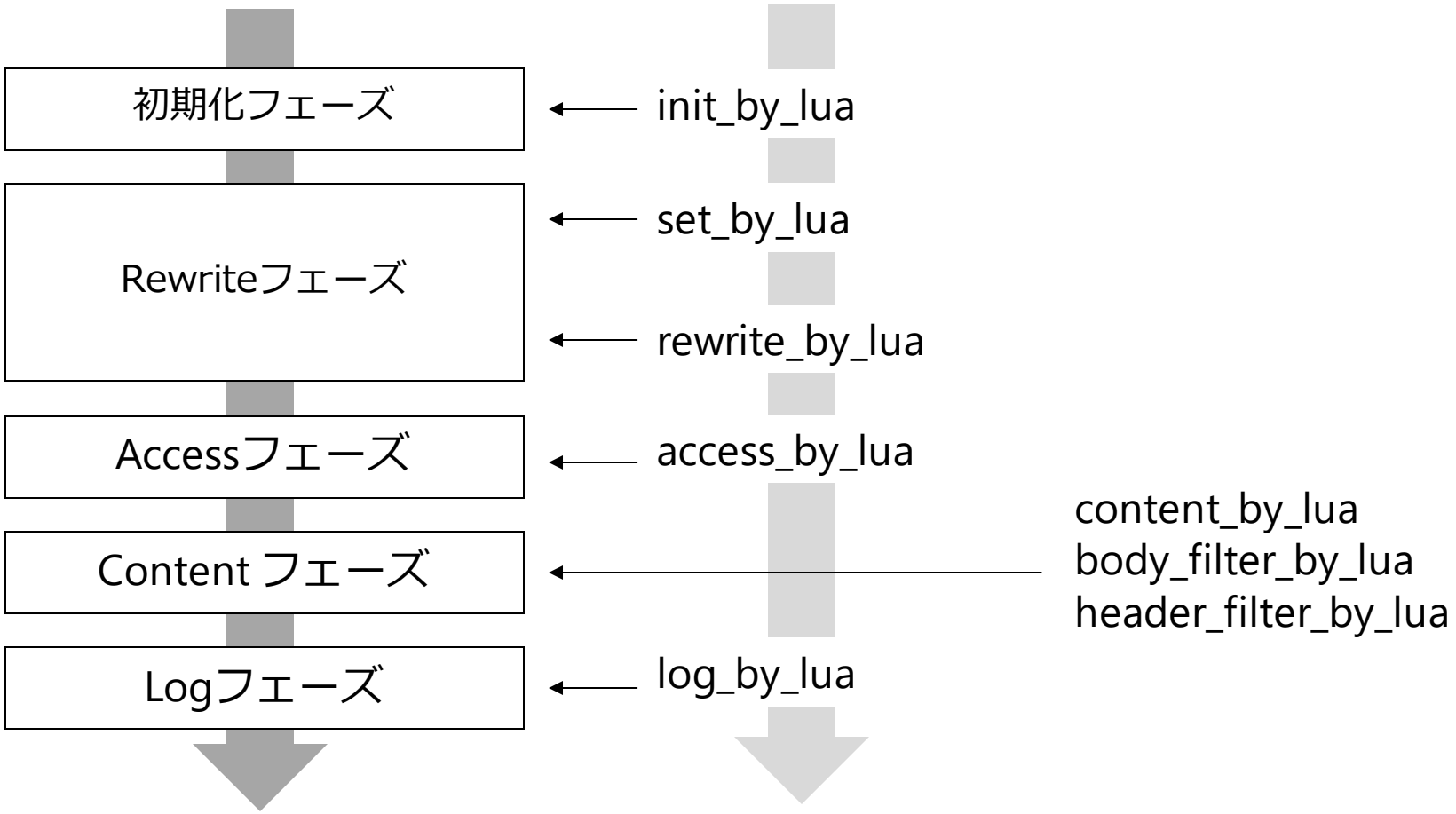
nginxをより高度なサーバアプリケーションフレームワークとして利用

```
set $expire 'off';
set $access_log 1;
rewrite_by_lua_block {
    local uri = ngx.var.uri
    local regex = [=[¥.(jpg|jpeg|gif|png|css|js|swf|ico|pdf|svg|eot|ttf|woff|woff2|webp|jp2|jxr|map)$]=]
    local option = 'jo'
    if uri and ngx.re.match( uri, regex, option ) then
        ngx.var.expire = '60d'
        ngx.var.access_log = 0
    end
}
expires $expire;
access_log  /home/kusanagi/server/log/nginx/ssl_access.log main if=$access_log;
```

nginx

A1/C1

Luaによるアプリケーション化



nginx

A1/C1

LuaをLuaJITに変更する

LuaJIT＝ Lua の JITコンパイラ(Just-In-Time)

Luaと比べ、 LuaJITは10倍以上高速

	Version	コンパイル時間 (s)	実行時間 (s)
lua	5.2.3	0	67.669
luajit	2.0.2	0	4.962

この頃 流行りの 言語たち（他）でベンチマーク （Dart, Go, Julia, Nim, Python, Rust 他）

nginx

A1/C1

PCRE(LuaJIT) を PCRE JITに変更する

ngx_luaの正規表現

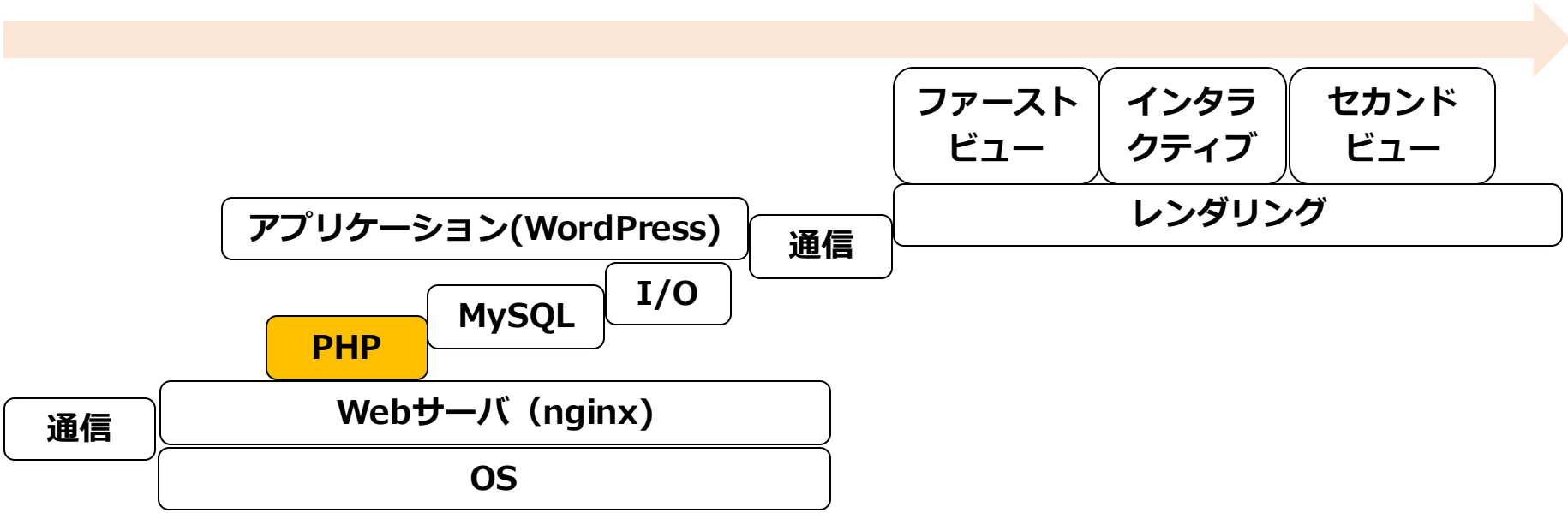
Lua　　string.match
PCRE　　re.match
PCRE JIT　re.match

パス	pcre_jit	かかった時間(sec)
string_match	-	2.18
re_match	off	12.86
re_match	on	12.63
re_match_optimize	off	5.55
re_match_optimize	on	4.48

OpenRestyのre.matchとstring.matchの性能差
https://qiita.com/toritori0318/items/0ecb337acd86e276e38e

PHP

- PHP5系からPHP7系へ
- OPCache の導入
- APCuの導入
- PCRE JITの利用

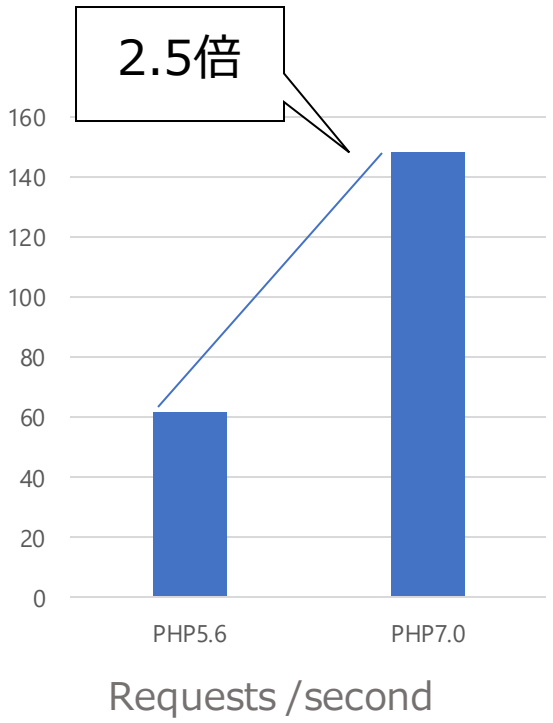


PHP

A1/C3

PHP5系からPHP7系へ

性能向上やメモリ使用量の削減など大きな改善



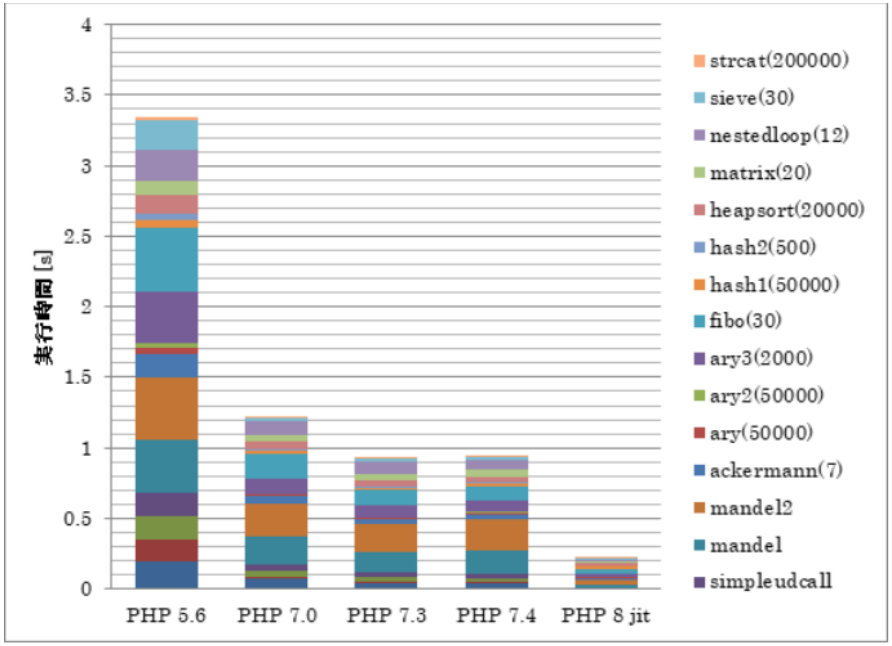


図1　実行速度ベンチマーク結果の例
(Zend/bench.php)

https://column.prime-strategy.co.jp/archives/column_3179

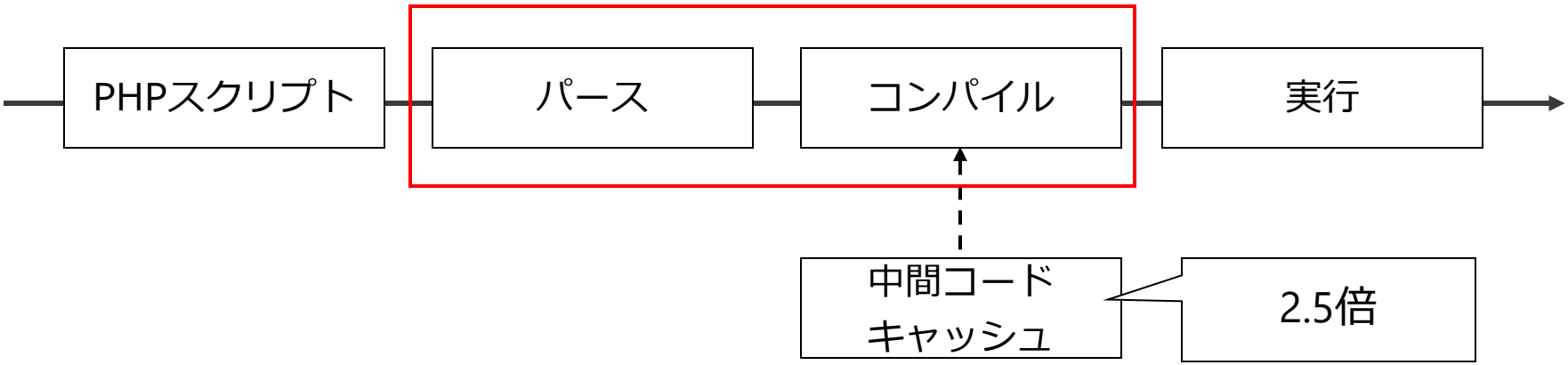
PHP

A1/C1

OPCacheの導入

中間コード（オペコード）のキャッシュ

コンパイル後のコードをメモリに保存して再利用



PHPのソースコードに変更が加えられていた場合は
キャッシュされた中間コードは破棄

→ OPCache の有無でPHPの出力結果は変わらない

PHP

B1/C2

APCu（ユーザーキャッシュ）の導入

オブジェクトや変数をメモリに保存して、別のリクエストから再利用するキーバリュー型（KVS)のキャッシュ

```
//キャッシュに保存
	apcu_store($key,  $var, $ttl);

//キャッシュから取得
	$var = apcu_fetch($key);
```

PHP

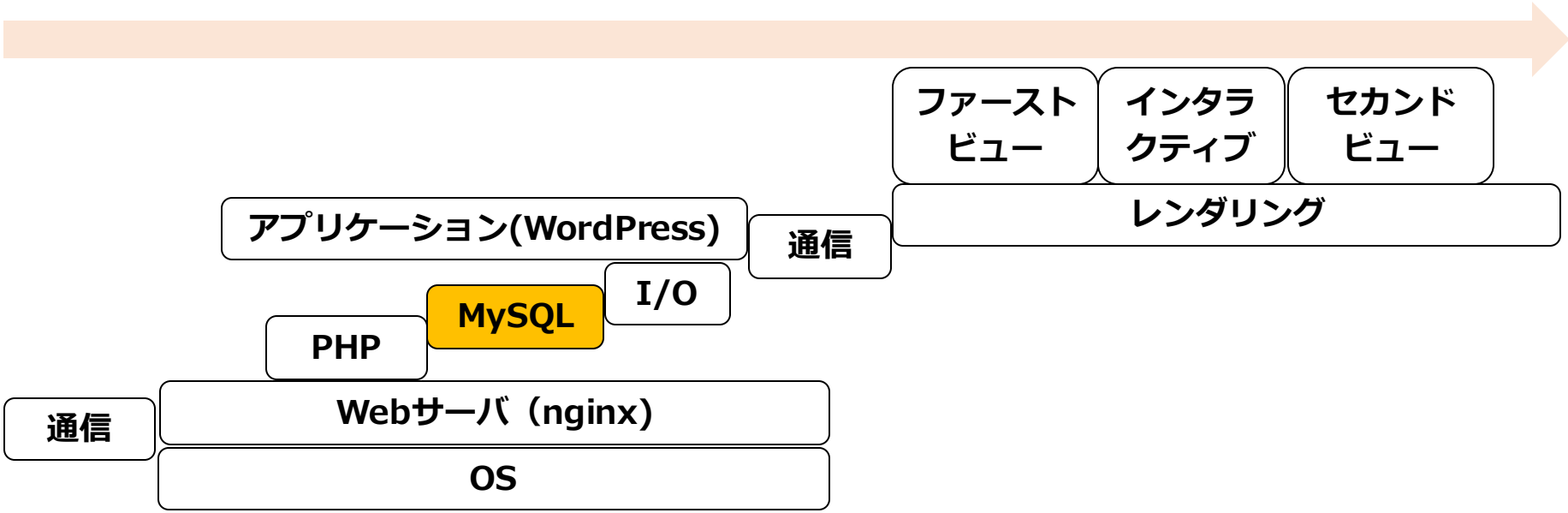
A1/C1

PCRE(PHP) を PCRE JITに変更する

- PCRE　Perl互換の正規表現
- PCRE JIT　正規表現のJITコンパイラ
  正規表現のマッチングのパフォーマンスの向上
  preg_match関数で10倍のパフォーマンスも

MySQL

- Query Cache
  - InnoDB
  - MyIsam



MySQL

A1/C1&C2

Query Cacheの導入

参照系クエリ（SELECT文）の実行結果をメモリにキャッシュし再利用

- パフォーマンスは数%から1000%以上
  （クエリが増えるほど効果が大きくなる）
- データベースの更新でクエリの結果が影響を受けるときにキャッシュを破棄

→クエリキャッシュの有効／無効によってクエリの結果は変わらない

【設定の目安】想定されるデータベースサイズの10%

```
[mysqld]
query_cache_size = 64M
```

MySQL (InnoDB)

A1/C2

InnoDB Buffer Poolの設定

頻繁にアクセスされるデータとインデックスをメモリーにキャッシュ

innodb_buffer_pool_size

メモリにキャッシュするため、大きいほど有利

【設定の目安】想定されるデータベースサイズの120%

```
[mysqld]
innodb_buffer_pool_size = 512M
```

MySQL (MyIsam)

A1/C2

OSのディスク（ページ）キャッシュの利用

MyIsam

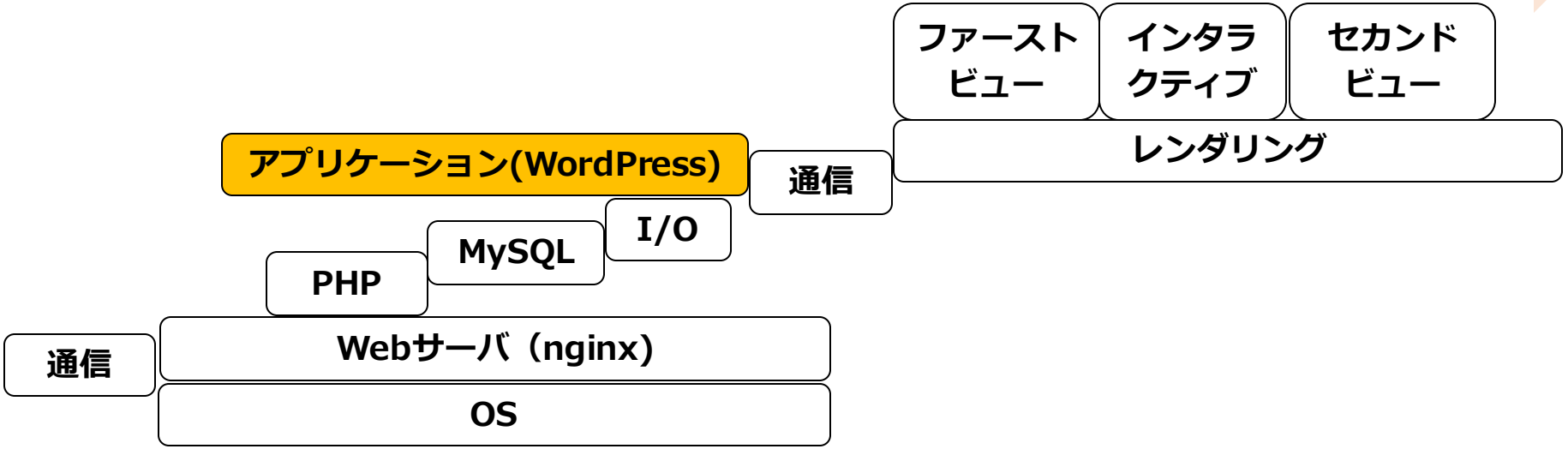
- テーブルデータのキャッシュ→OSのディスク（ページ）キャッシュを利用

```
# free -m
              total        used        free      shared  buff/cache   available
Mem:           7802        5782         187         379        1833        1334
Swap:          2047         451        1596
```

アプリケーション（WordPress）

- 翻訳キャッシュ
- ページキャッシュ
- 部分キャッシュ（Transients API）
- オブジェクトキャッシュ

- SQLクエリの削除、変更
- wpdbの拡張/クエリキャッシュ
- I/Oに基づく最適化処理



アプリケーション（WordPress）

B1/C1

翻訳キャッシュの導入

WordPressを日本語で表示する場合、辞書ファイル（ja.mo）をリアルタイムに読み込むため、処理に時間がかかる。

→翻訳ファイルの読み込みをキャッシュし、高速化させる

アプリケーション（WordPress）

B1/C1

ページキャッシュの導入

生成したHTMLを一定期間再利用する。

WP Super Cacheなどのプラグインを利用することが多い。

- ページの性質により適切な有効期限を設定
- HTMLの再利用という性質上、設定に注意が必要

  トラブルの例）
  - 管理画面は対象外→管理画面が外部で閲覧可能に
  - ログインユーザごとの機能があるサイト
    →コメントが反映されない
  - EC サイト（カート） →他のユーザの購入データが表示

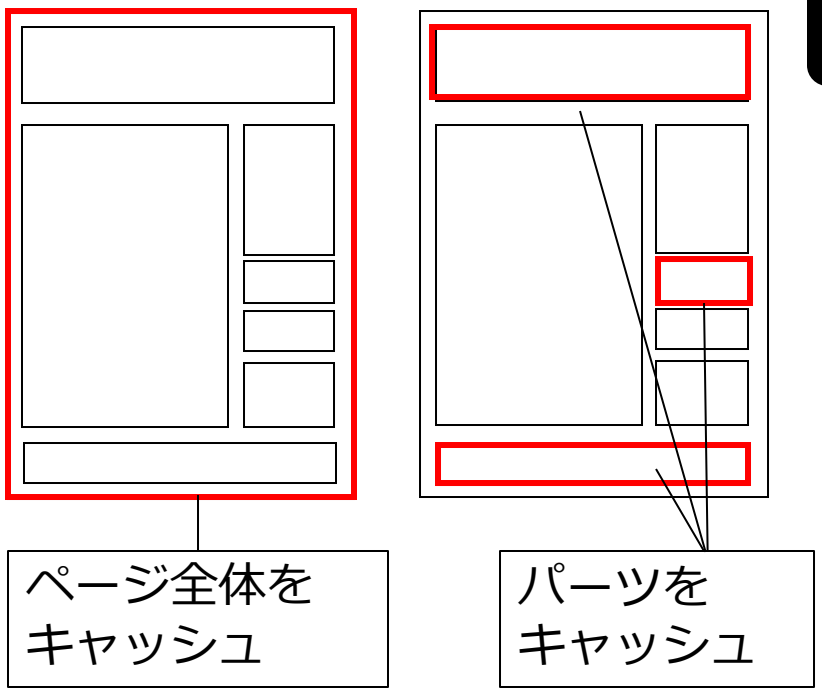
アプリケーション（WordPress）

B2/C1＆C2

部分キャッシュ（Transients API）の導入

WordPressの処理結果を部分的にデータベースにキャッシュ

ページキャッシュと部分キャッシュ




```
set_transient( $transient, $value,
$expiration );
```

```
5.6488840580秒 # Transients API使用なし
0.0009789467秒 # Transients API使用あり、APCu使用なし
```

6200倍

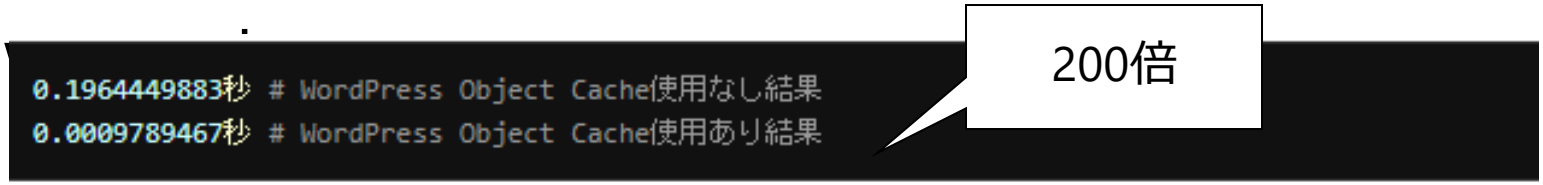
アプリケーション（WordPress）

A1/C1＆C2

オブジェクトキャッシュ（WordPress Object Cache）の導入

取得に時間のかかるデータをメモリ上に保存

- データベースのクエリ実行時間の削減
- ネットワーク経由のデータ転送時間の削減



PHPプロセスのメモリ内キャッシュ→セッション内のみ

アプリケーション（WordPress）

B 1/C1＆C2

オブジェクトキャッシュ（WordPress Object Cache） の永続化

共有メモリAPCu+機能拡張

（ドロップイン・プラグイン

WordPress APCu Object Cache Backend）

→セッションをまたいだWordPress Object Cacheの保持

```
5.6488840580秒 # Transients API使用なし
0.0009789467秒 # Transients API使用あり、APCu使用なし
0.0000348091秒 # Transients API使用あり、APCu使用あり
```

30倍

アプリケーション（WordPress）

A 1/C3

SQLクエリの削除、変更をする

■使われてないSQLを発行しているフックを解除

pre_get_posts,queryにフックして条件を変更

→get_postやwp_query、postmeta の取得数の制限

→ get_postやwp_query、postmetaの先読みを抑制

■クエリの書き換え

「WHERE」句や「ORDER BY」句を簡素化しても得られる結果に大きな差が発生しない場合、これらの条件自体を変更してクエリの高速化を図る

アプリケーション（WordPress）

B 1/C1＆C2

wpdbの拡張/クエリキャッシュ

WordPress Object Cacheでキャッシュできない参照系（SELECT）のクエリをキャッシュ

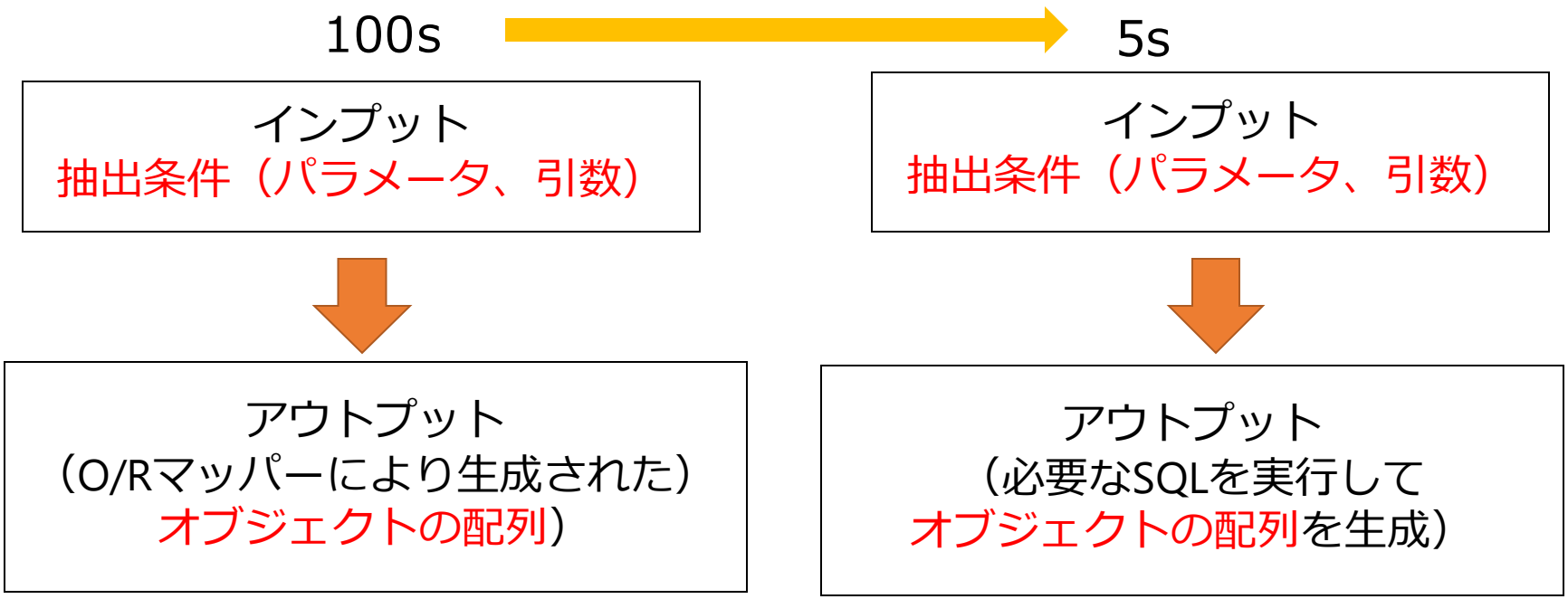
データベースのクエリキャッシュのパージ、枯渇時にもキャッシュ保持

- 性能向上の時間の延長
- ネットワーク経由のデータ転送時間の削減

アプリケーション（WordPress）

B 2/C3

I/Oに基づく最適化処理
（同じI/Oを異なるアルゴリズムで代替）



その他

B 1/C1＆C2

CDN を利用する

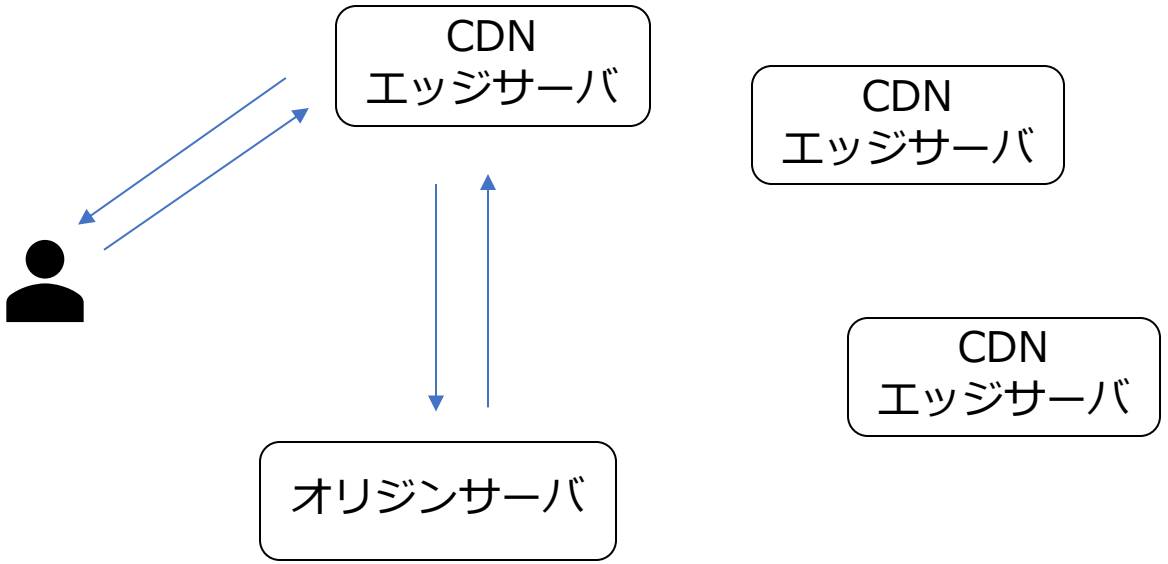
エッジサーバからキャッシュしたコンテンツを配信

・画像、CSS/JSなどのリソース

・HTML

キャッシュの範囲と時間、パージの方法の設計

- Cloudfront
- Cloudflare
- Fastly
- Akamai



キャッシュ戦略のまとめ

A1 → A2 → B1 → B2

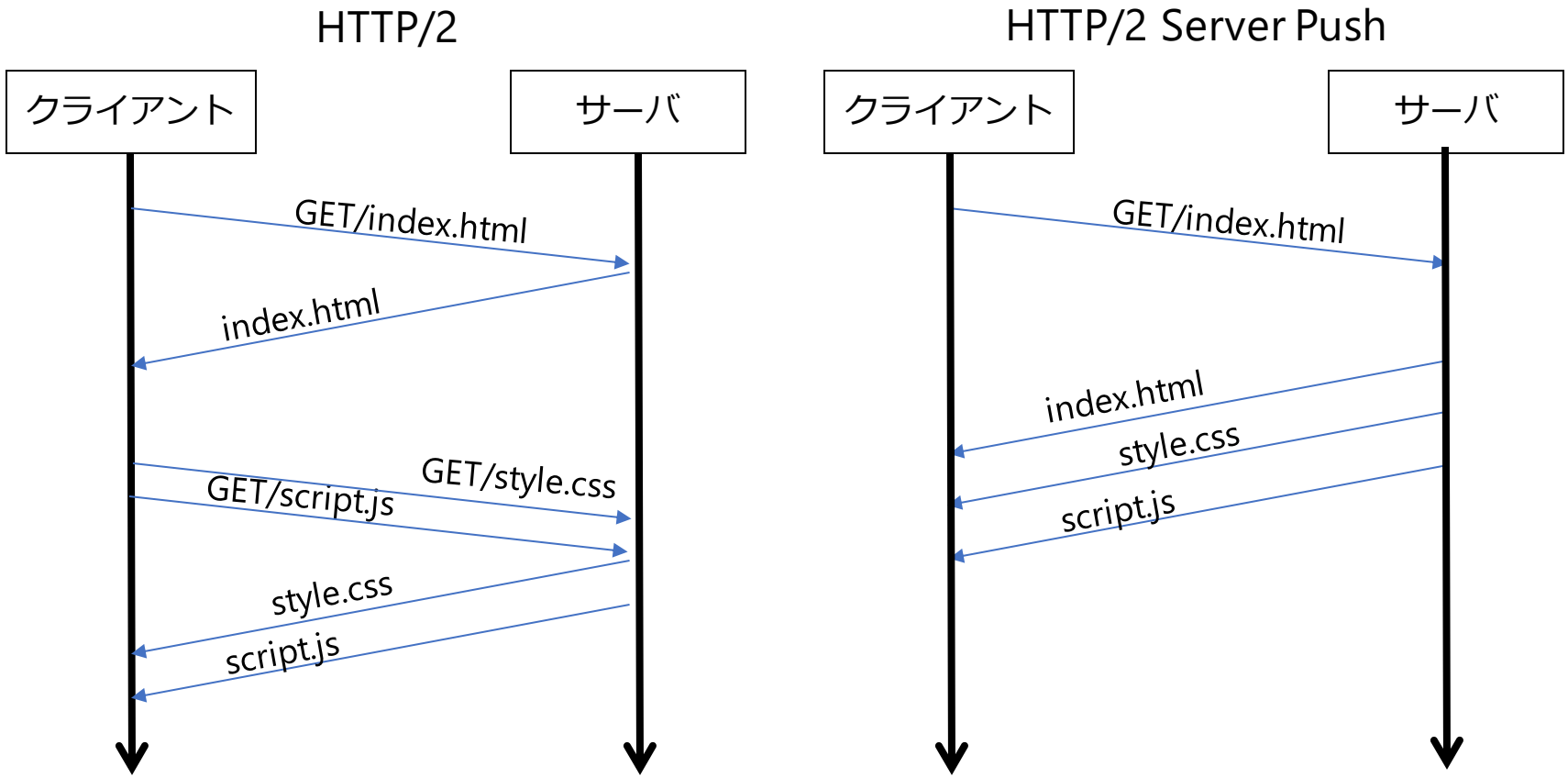
	A （最新のデータによるアウトプットが保証）	B （過去の情報を再利用）
1 修正不要	• ディスク（ページ）キャッシュの活用 (C2) • PHP5系からPHP7系へ(C3) • OPcache(PHP 5.4以降) (C1) • Query Cache (C1&C2) • InnoDB Buffer Pool (C2) • SQLクエリの削除、変更 (C3) • オブジェクトキャッシュ（単一セッション）(C1&C2) • Luaによるアプリケーション化(C1) • LuaをLuaJITに変更 • PCRE(LuaJIT) を PCRE JITに変更(C1) • PCRE(PHP) を PCRE JITに変更(C1) • CPUのコア数よりCPUの周波数の高いものを選択 (C3)	• fastcgi cache (C1&C2) • proxy cache (C1&C2) • expires (C1&C2) • open file cache (C2) • 翻訳キャッシュ (C1) • ページキャッシュ (C1&C2) • オブジェクトキャッシュの永続化 (C1&C2) • APCu（インメモリのKVS） (C2) • Wpdbの拡張/クエリキャッシュ (C1&C2) • CDN (C1&C2)
2 修正要	• PHP5系からPHP7系へ(C3)	• 部分キャッシュ（Transients API）(C1&C2) • I/Oに基づく最適化処理（同じI/Oを異なるアルゴリズムで代替） (C3)

ネットワークの高速化

HTTP/2

HTTP/2による通信の多重化とサーバプッシュ

```
listen       443 ssl http2;
```



ネットワーク帯域の確保

クライアント側が5Gでもバックエンドが
100Mbpsでは4G/3Gと変わりません。

10Mbps
↓
100Mbps
↓
1Gbps

画像リソース

同一画像フォーマットによる最適化

■JPEG

ライブラリの変更 libjpeg → mozjpeg

- サイズの変更
- qualityの変更

  cjpeg -quality 60
- メタ情報の削除（位置情報など）

  jpegtran -copy none
- データの内部保持形式の調整

  jpegtran -optimize

cjpeg	非可逆圧縮　ファイルの圧縮、Jpegへの変換など
jpegtran	可逆圧縮 メタ情報の削除、加工など 「PageSpeed Insights」で推奨

画像リソース

- 次世代画像フォーマットによる最適化

■PNG

- サイズの変更
- 色数の変更　pngquant　true colorからindex color（減色パレット）へ

  pngquant 256 --speed 1
- メタ情報の削除（位置情報など）

  optipng –strip all
- データの内部保持形式の調整optipng

  optipng -o7

pngquant	非可逆圧縮　メディアンカット法による画像の減色
optipng	可逆圧縮　最適化、メタ情報の削除、PNG への変換

次世代画像フォーマット

次世代画像フォーマット対応に変換し、ブラウザによりだし分け

■JPEG

JPEG2000/JPEGXR

■PNG

WebP(libwebp)

	WebP	JPEG2000	JPEGXR
IE	×	×	○
Edge	×	×	○
Firefox	○	×	×
Chrome	○	×	×
Safari	×	○	×

User Agent, Acceptヘッダなどで判定、map, lua, try filesなどで出し分け

画像リソース

画像最適化の効果

218KB→15.3KB

konsinkai_425.png	GET	200	png	202...	218 KB

konsinkai_425.png	GET	200	webp	lazy...	15.3 KB

画像リソース

Lazy load

スクロールに応じて、画像データを遅延読込する

- jQuery（JavaScript）
- WordPressプラグイン

CSS/JS/HTMLのミニファイ・最適化

■CSS

- 利用していないCSSの除外
  - Chromeのcoverageで調査
  - UnCSS
- Minify （ホワイトスペースの削除）
  - CSSnano

■JS

- 利用していないJSの除外
  - Chromeのcoverageで調査
- Minify （ホワイトスペースの削除）
  - terser/Uglify JS

■HTML

- Minify （ホワイトスペースの削除）
  - PageSpeed Module

gzip/brotli（通信の圧縮）

リソース最適化の手法

1. 直接リソースに変更を加える BETTER
2. 最適化したリソースを別に用意して、直接リソースの変更を行わない BEST

   nginx map、try files、lua

   ・acceptヘッダ

   ・cookie

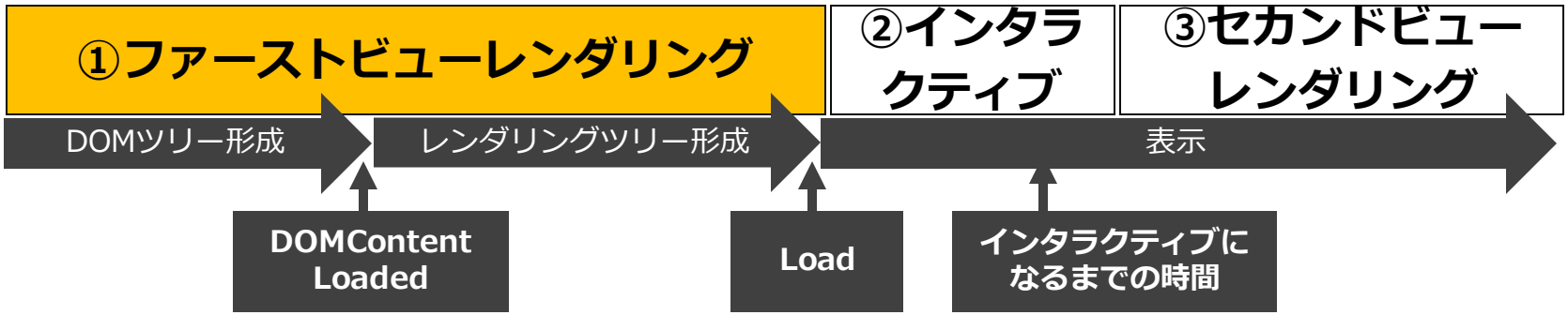
   ・user agent

   ・url

   などをもとにリソース毎、ユーザーごとに出しわける

レンダリングの高速化

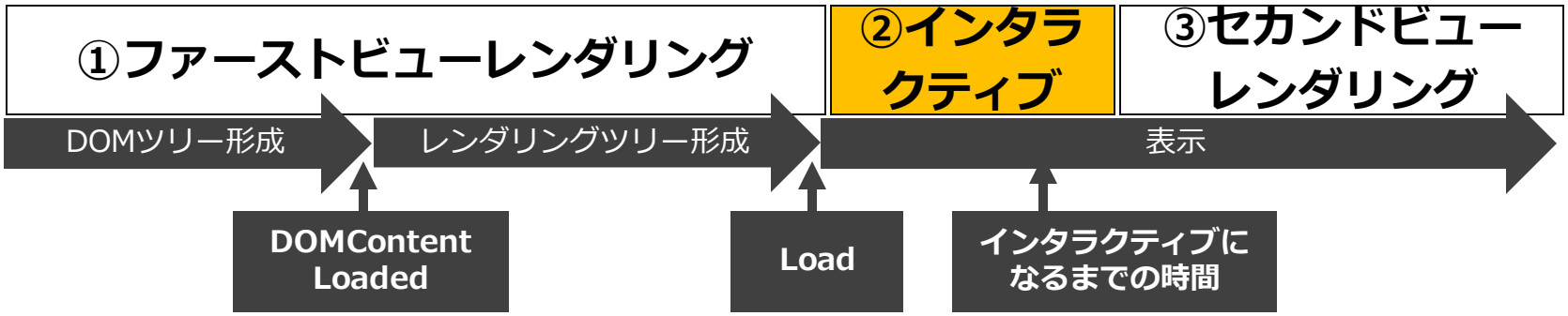
レンダリングプロセスの分解と再構成



ファーストビューに必要なリソースのみ最短で実行する。

それ以外をloadイベント以降に遅らせることで、DOMContentLoadedイベントとloadイベントまでの時間を最短にする。

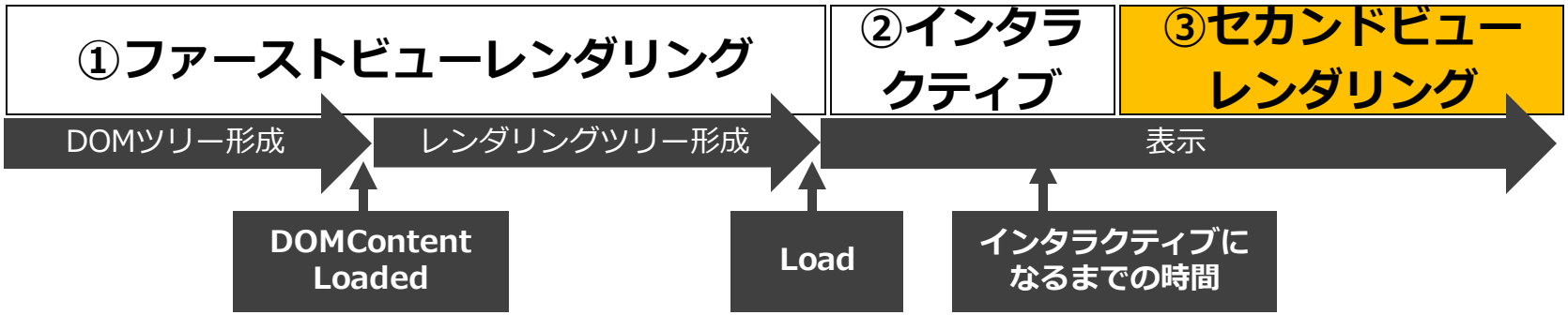
レンダリングプロセスの分解と再構成

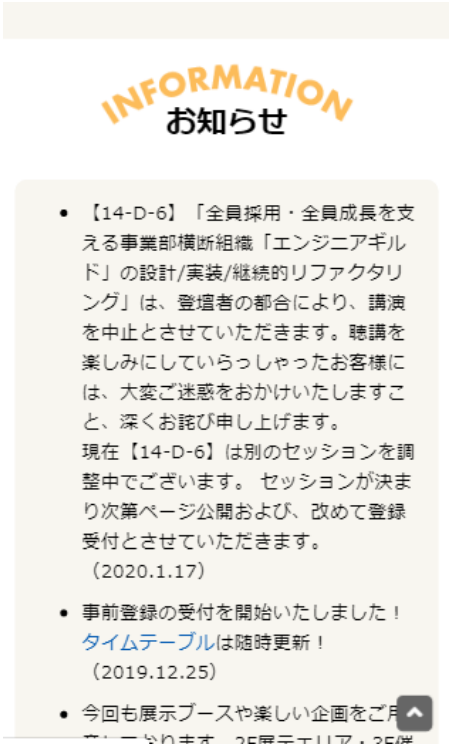


Loadイベントからは、ユーザーが操作できる状態を確保する。

ユーザーの操作が始まるか、数秒間はファーストビューに関係しないJSやwebfontなどのリソースの実行を遅延させる。

レンダリングプロセスの分解と再構成

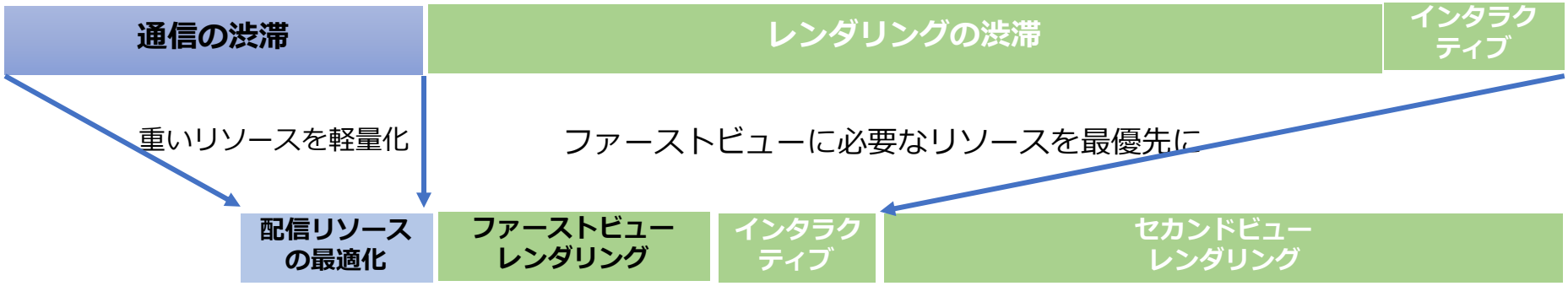




最後にセカンドビューの交通整理を行ってユーザー操作や時間の経過とともにリソースの直列、並列での実行処理をおこなう。

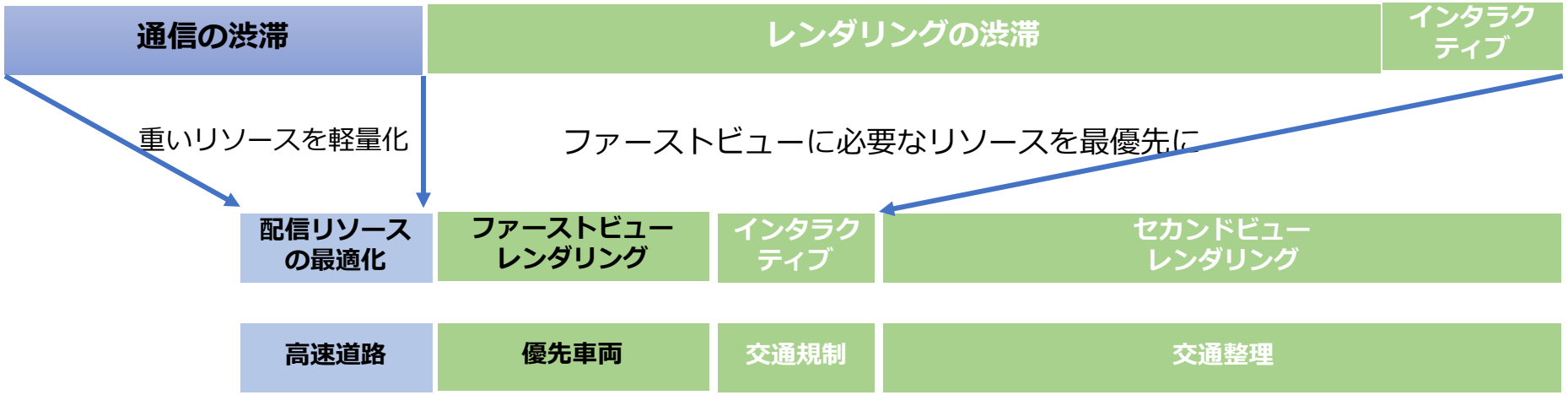
ネットワーク・レンダリングの渋滞の解消

重いリソースを軽量化、渋滞を解消し、ファーストビューとインタラクティブを最短にする



ネットワーク・レンダリングの渋滞の解消

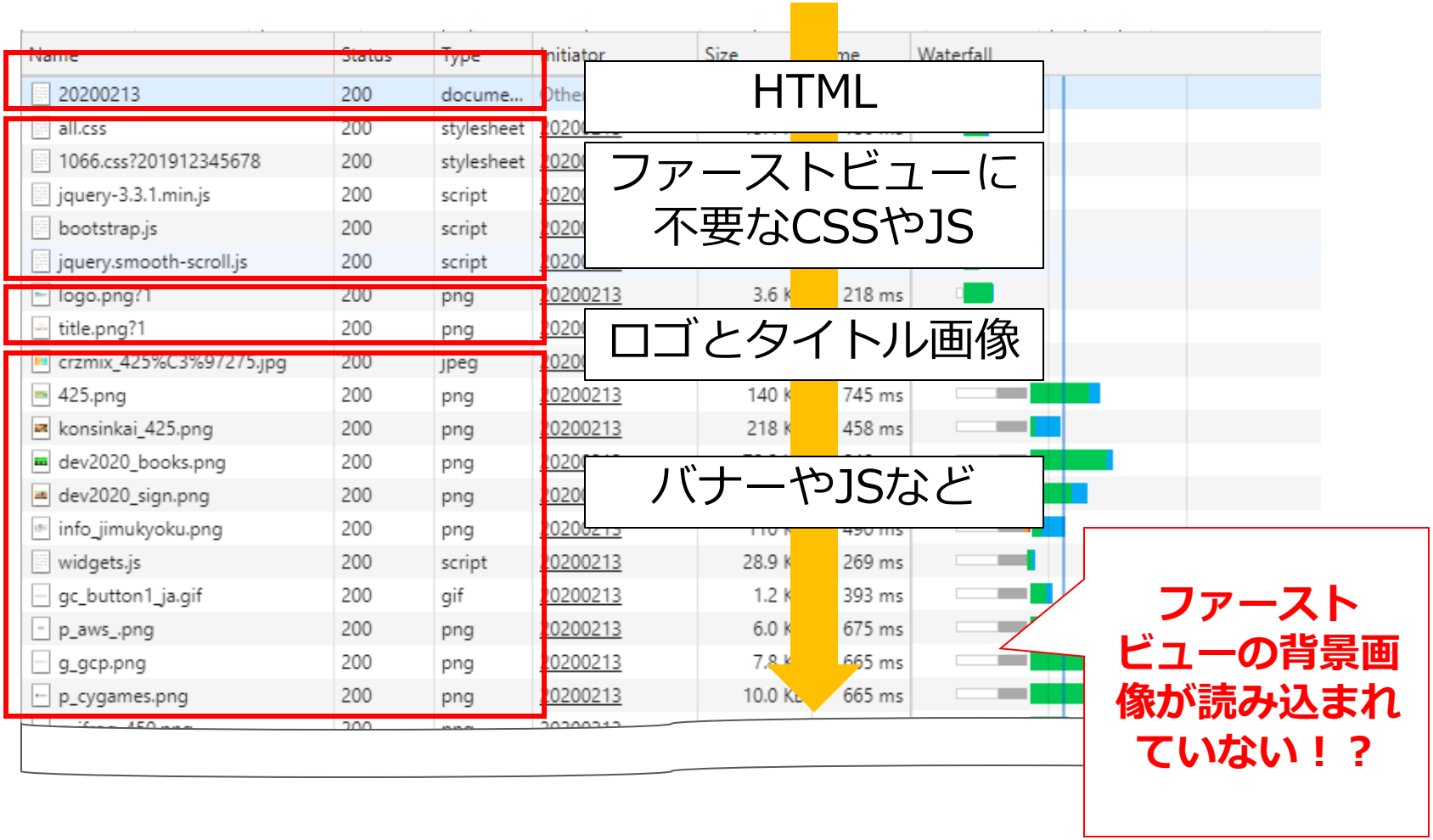
重いリソースを軽量化、渋滞を解消し、ファーストビューとインタラクティブを最短にする



Developers Summit 2020 のサイトを見てみましょう

ソースコードや
デベロッパーツールで
実際に確認してみてください

https://event.shoeisha.jp/devsumi/20200213



https://event.shoeisha.jp/devsumi/20200213

47ba260f9caf737caf12bc327322...	200	png	20200213	4.7 KB	180 ms
429d0d2fa112dd4ba545323b6e...	200	jpeg	20200213	2.0 KB	
b9829cdd4fab25228d68cda8c52...	200	png	20200213		
3071564f5f0aee05c9a543f4746b...	200	jpeg	20200213		
avatar?s=50	200	jpeg	20200213	2.0 KB	
dcd5f2ae371fc2c395235a28a4af...	200	jpeg	20200213	1.8 KB	
c_sone.png	200	png	20200213	33.0 KB	
c_kawamura.png	200	png	20200213	46.4 KB	
c_orita_akihiro.png	200	png	20200213	31.9 KB	
c_yoshida_yuya_.png	200	png	20200213	36.4 KB	
c_kuroda_itsuki.png	200	png	20200213	22.8 KB	
c_kyon.png	200	png	20200213	47.4 KB	
c_sugawara_nobisuke.png	200	png	20200213	48.6 KB	686 ms
c_nagase_seiichiro.png	200	png	20200213		705 ms
twitter.png	200	png	20200213		704 ms
facebook.png	200	png	20200213		747 ms
hatena.png	200	png	20200213	2.1 KB	748 ms
bookmark_button.js	200	script	20200213	10.8 KB	365 ms
cx.js	200	script	20200213:1		
analytics.js	200	script	20200213:		
fbevents.js	200	script	20200213:		
sdk.js	200	script	20200213:1...	2.1 KB	
embed?pb=!1m14!1m8!1m3!1d...	200	document	20200213		
back.png	200	png	20200213		
h2-info.png	200	png	20200213		
h2-sns.png	200	png	20200213	1.2 KB	
fa-solid-900.woff2	200	font	20200213		831 ms
h2-outline.png	200	png	20200213		696 ms
h2-sponsor.png	200	png	20200213		700 ms
h2-comm.png	200	png	20200213	2.6 KB	701 ms
h2-code.png?1	200	png	20200213	4.0 KB	696 ms

バナー画像

フッターのSNS画像

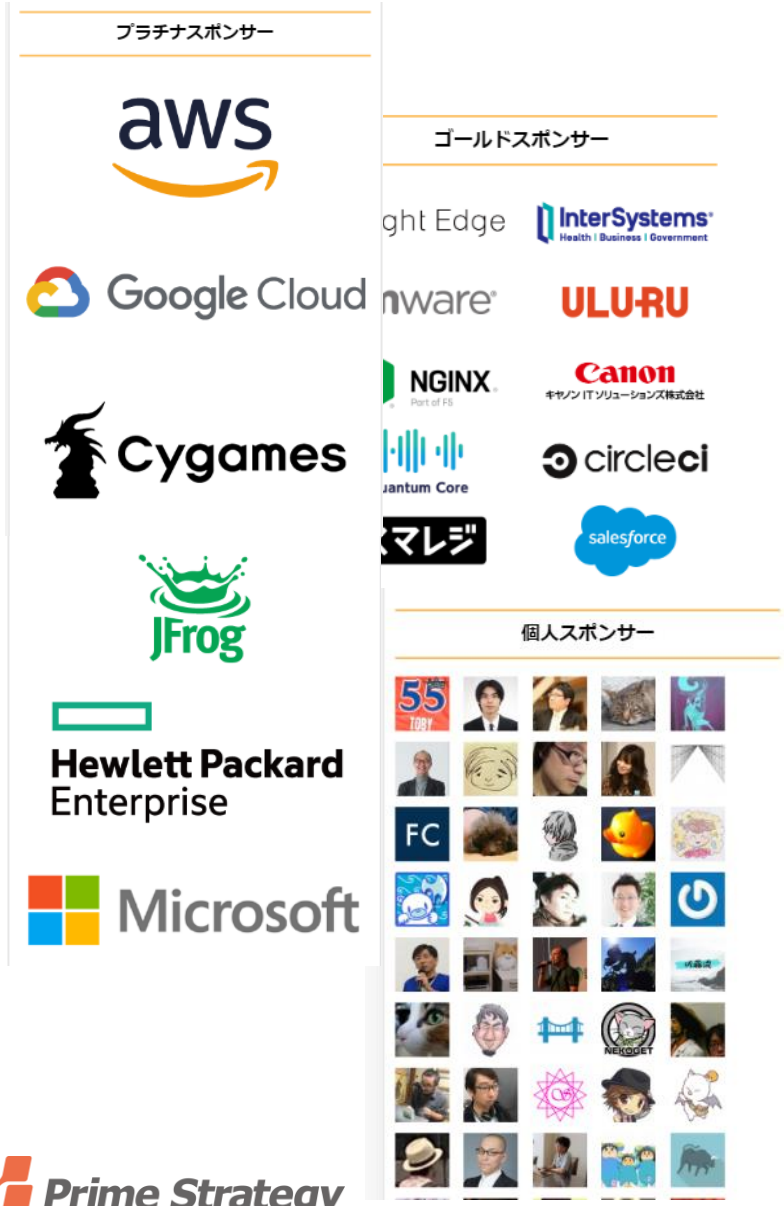
計測やSNSのJS

ココ！！

見出し画像など

リソースの大渋滞が起きている

画像の最適化と遅延読み込み



画像の最適化と
次世代フォーマットへの変換
↓
遅延読み込み（Lazy load)

JS / Webフォント の遅延と交通整理

SNSやGoogleマップはスクロールしてから表示されればよい

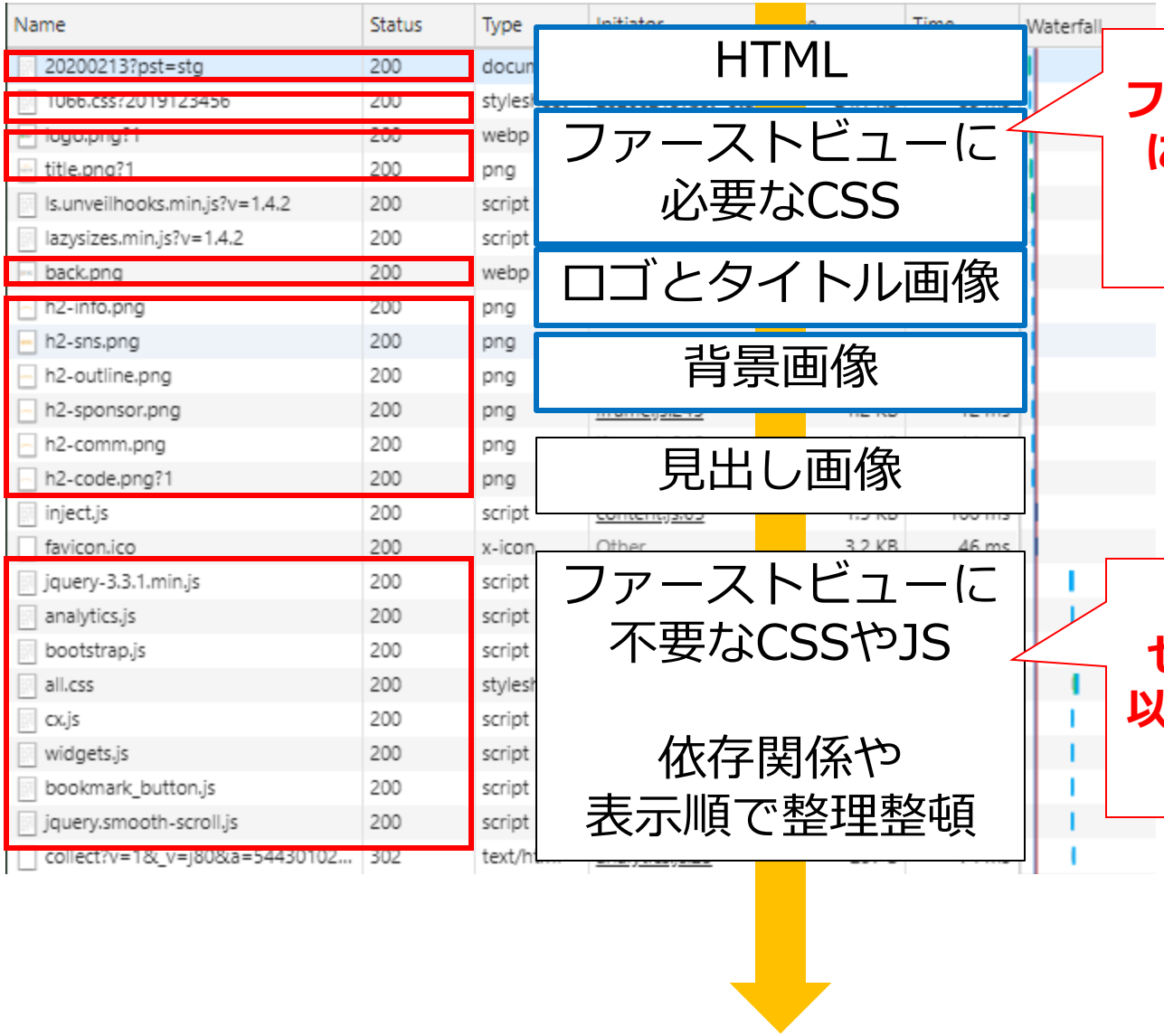
↓

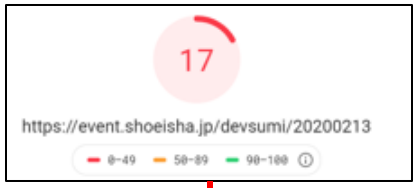
セカンドビュー以降に遅延させる

↓

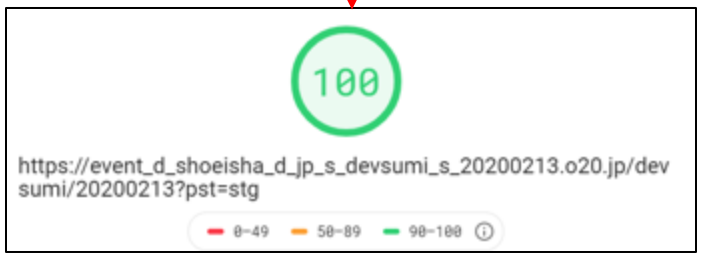
SNSのウィジェット
Googleマップ
SNS アイコン
（交通整理）

デモページ

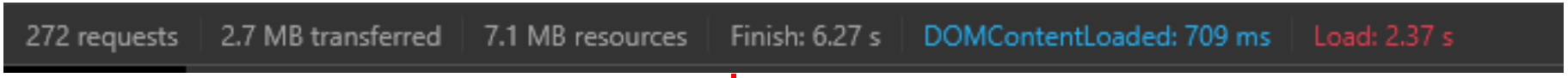

Name
Status
Type
Waterfall
20200213?pst=stg 200 docun
1066.css?2019123456 200 styles
webp
title.png?1 200 png
ls.unveilhooks.min.js?v=1.4.2 200 script
lazysizes.min.js?v=1.4.2 200 script
back.png 200 webp
h2-info.png 200 png
h2-sns.png 200 png
h2-outline.png 200 png
h2-sponsor.png 200 png
h2-comm.png 200 png
h2-code.png?1 200 png
inject.js 200 script
favicon.ico 200 x-icon Other 3.2 KB 46 ms
jquery-3.3.1.min.js 200 script
analytics.js 200 script
bootstrap.js 200 script
all.css 200 styles
cx.js 200 script
widgets.js 200 script
bookmark_button.js 200 script
jquery.smooth-scroll.js 200 script
302 text/h
HTML
ファーストビューに
必要なCSS
ロゴとタイトル画像
背景画像
見出し画像
ファーストビューに
不要なCSSやJS
依存関係や
表示順で整理整頓
ファーストビュー
に必要なものを
最速で表示
セカンドビュー
以降は順番を制御


17
https://event.shoeisha.jp/devsumi/20200213
0-49
50-89
90-100


First Contentful Paint 4.6 秒
First Meaningful Paint 4.6 秒
速度インデックス 8.3 秒
CPU の初回アイドル 12.4 秒
インタラクティブになるまでの時間 20.7 秒
初回入力遅延の最大推定時間 300 ミリ秒


100
https://event_d_shoeisha_d_jp_s_devsumi_s_20200213.o20.jp/devsumi/20200213?pst=stg
0-49
50-89
90-100


First Contentful Paint 1.3 秒
First Meaningful Paint 1.3 秒
速度インデックス 1.5 秒
CPU の初回アイドル 1.3 秒
インタラクティブになるまでの時間 1.3 秒
初回入力遅延の最大推定時間 80 ミリ秒


272 requests
2.7 MB transferred
7.1 MB resources
Finish: 6.27 s
DOMContentLoaded: 709 ms
Load: 2.37 s


100 requests
298 KB transferred
4.6 MB resources
Finish: 7.87 s
DOMContentLoaded: 435 ms
Load: 648 ms

手法

1. 直接アプリケーションを修正する BETTER
2. PHP→アウトプットバッファリング BEST

    WordPress → hook
    nginxのluaならどのような処理系でも出力されるHTMLなどをオリジナルのシステムに割り込んで（バッファ制御して）書き換えることができる。

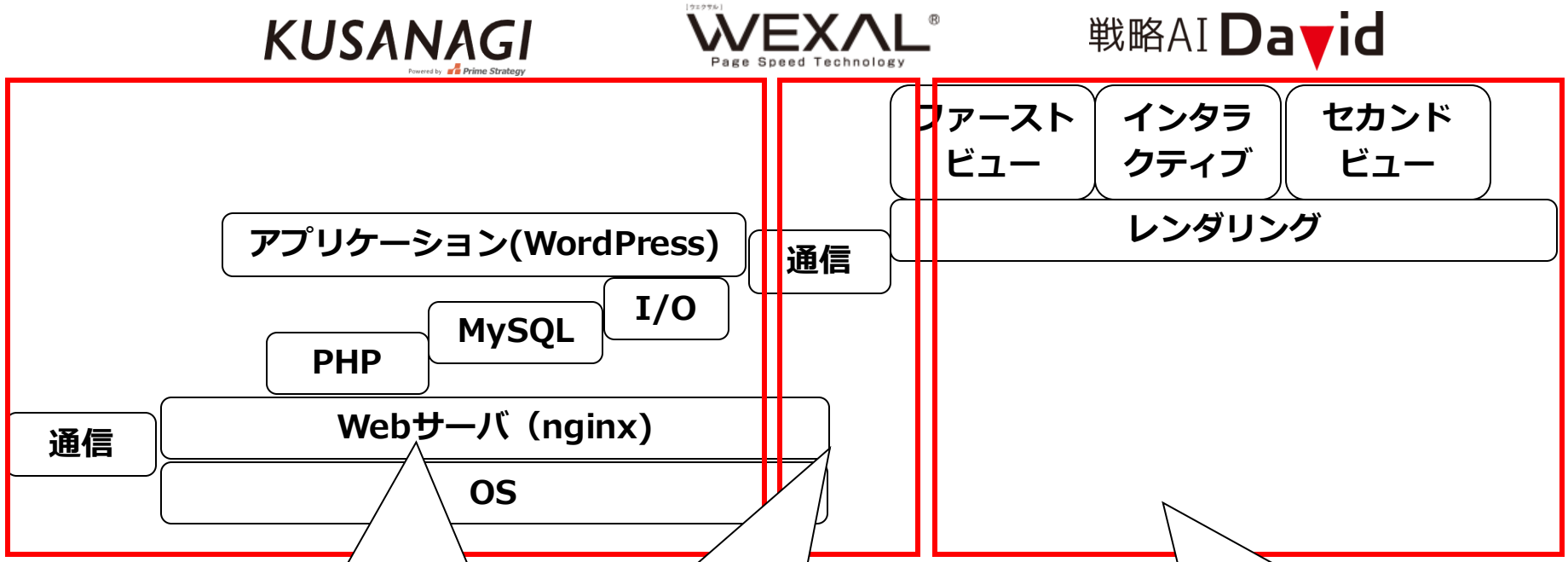
高速化の自動化

ここまで紹介した施策を複数のサイトに
個別に、継続的に行うのは困難

バックエンド・・構成済みのインスタンス
CDNの導入
画像/リソースの最適化・・自動化
レンダリング・・・？

KUSANAGI Premium Edition

(KUSNAGI, WEXAL Page Speed Technology, David)



キャッシュなしで10～15倍、キャッシュありで1000倍～のパフォーマンスを実現する構成済みサーバOS

稼働しているシステムに一切の改変を加えることなく、リアルタイム・動的に、Webのリソースのすべてを最適化

Webサイトを解析し、ファーストビュー高速化の戦略を立案する戦略AI

KUSANAGI Premium Editionの利用で
本日お話したことはすべて実現可能ですが・・

・ファイルの圧縮などの自動化
　→gulp（タスクランナー）
・画像の最適化の自動化
　→inotifywait でファイル監視

などで自動化をすることもできます。

本日のまとめ

- Webサイトの高速化をするには、ネットワーク・バックエンド・レンダリングを重層的にチューニング
- バックエンドの高速化はキャッシュ戦略。効果と性質で使い分ける
- ネットワークの高速化はリソースの最適化・軽量化
- レンダリングの高速化はファーストビューの高速化戦略
- 高速化は自動化、戦略はAIに

ご清聴ありがとうございました