学士論文　2017 年度（平成 29 年度）

モダン Web 開発における CSS 設計思想による パフォーマンス最適化手法の提案

慶應義塾大学 総合政策学部

瀬下 明紗子

徳田 · 村井 · 楠本 · 中村 · 高汐 · バンミーター · 植原 · 三次 · 中澤 · 武田
合同研究プロジェクト

2018 年 1 月

学士論文　2017 年度（平成 29 年度）

モダン Web 開発における CSS 設計思想による パフォーマンス最適化手法の提案

論文要旨

本研究の目的は，CSS(Cascading Style Sheet) コードを最適化し，Web サイトの表示速度の向上を目指すことである．

Web が人々の日常において欠かせないものとなるにつれて，フロントエンド開発は急速に巨大化し複雑化してきた．その中で Web パフォーマンスの重要性は増し，フロントエンドにおいても意識すべきこととなっている．

本研究では未だ議論が未成熟である CSS コード開発におけるパフォーマンス改善手法について検証することで，パフォーマンス改善に有効な方法論を探ることを目的とする．

研究の手法として，パフォーマンスにおける CSS 最適化手法について調査，また既存の設計方針を調査・分類した上で独自の設計手法を提案し，それをもとに web サイトを実装した．評価として同一デザインの web サイトを実装し，Google Chrome を用いて表示スピードの計測実験を行い，得られたデータを比較した．

実験の結果として提案する方針は決定的な改善策ではないが有用であるとわかった．

キーワード

CSS 設計，パフォーマンス最適化，HTTP/2

慶應義塾大学 総合政策学部

瀬下 明紗子

Abstract Of Bachelor's Thesis Academic Year 2017

Proposal of CSS Architecture Optimization in Modern Web Development

Summary

The purpose of this study is to optimize CSS source codes to maximize its rendering speed.

As for increasing Web importance in our lives, front-end development is growing rapidly and becoming more complex. Furthermore, Web page performance is becoming increasingly important, although Web developers are not aware of this fact.

Therefore, CSS methodology is still immature because of the lack of the consciousness.

The purpose of this research is to find a methodology effective for performance improvement of CSS, by verifying discussed methods in CSS code development.

This paper investigates CSS optimization methods in performance and classifies existing CSS methodologies. From these classifications, the paper shows an implementation of a website based on the proposed proprietary designing method.

The evaluation was done by measuring the rendering speed of two different websites with the same design that were implemented in different CSS methodologies. The measurement was done by using Google Chrome to compare the obtained data.

The paper concludes that although the proposed method resulting from the experiment is not a definite improvement, the measurement has still contributed to the discussions of CSS methodology.

Keywords

CSS Architecture, CSS Methodologies, Optimization, HTTP/2

Faculty of Policy Management
Keio University

Asako Seshimo

目 次

図 目 次

表 目 次

第1章　序論

本章では，本研究の背景を示し，その課題と本研究の目的について述べる．

1.1　背景

インターネットが普及また発達し Web が日常的に利用されていくにつれて，Web ページのインターフェイスを担うフロントエンドは領域を拡大し複雑化してきた．フロントエンド開発に利用される主な言語である HTML，CSS，JavaScript は，ブラウザで表示するための文書と単なるその装飾から現在ではネイティブアプリの開発にまで利用されるようになっている．

それに呼応して主要言語は加速度的な変化を遂げてきた．特に JavaSctipt の標準である ECMA Script は，2015 年の ES6 登場以降 2017 年まで毎年新しいバージョンを勧告している [1]．同様に HTML 及び CSS も日々新しい要素が検討され実装されている．CSS では CSS3 と呼ばれる CSS2.1 の言語拡張を目的とした発展系が勧告されつつあり，丸角やシャドウ，アニメーションなどの機能を新たに追加した [2]．今後 CSS4 と呼ばれるさらに新たな更新が予定されており，変数のような役割を果たすカスタムプロパティなどの仕様が議論されている．

フロントエンド自体の領域の拡大に加え，言語の拡張によってフロントエンド開発におけるコーディングの規模は加速度的な増大の傾向にある．これは単なる文書とその装飾の表現を担うのみであった初期に比べ，コードの設計やパフォーマンスに関する方法論がより重要なものとなっていることを意味する．しかしこの中で，JavaScript のコーディング手法やフレームワーク，及びパフォーマンスへの影響は活発に議論されているが，CSS におけるパフォーマンスへの影響についてはあまり議論が進んでいない．今後 CSS の機能が増加し CSS コーディングがプログラミングの形態に近づいていくにつれ議論の重要性が高まっていくと考えられる．

1.2　本研究の目的

本研究では前節で述べた通り議論が活発でない CSS のパフォーマンス最適化に着目する．CSS はその性質から容易に保守困難になり肥大化してパフォーマンスにまで影響を与える可能性があり，コードそのものに秩序をもたらす設計手法が必要とされている．しかし，その手法は確立されておらず，開発者により単発的に方法論が提示されている．

本研究はこうした設計手法・思想の乱立と HTTP/2 の普及など基盤となる関連技術の発展を背景に，Web ブラウザでの表示パフォーマンスを基準に一定の指針の確立を目指すものである．この研究は効率のいい CSS コーディングのための新たな標準を生み出すことや CSS コーディングにおけるパフォーマンス改善が容易になること，また CSS を GUI から自動出力する際のアルゴリズム構築にも寄与するものと考える．

1.3 本論文の構成

本論文の構成を以下に示す．第 2 章では，本研究において前提となるフロントエンド環境及び CSS コーディングにおける根本的な問題点とそれに対する既存の解決策についてまとめる．第 3 章では CSS コーディングにおけるパフォーマンス最適化手法をまとめ，その先行研究及び独自の実験からその検証を行なう．第 4 章では CSS 設計方針に着目し，その特徴について調査する．第 5 章では第 3 章及び第 4 章における調査，検証結果を元に，有用と思われる設計手法の条件を提示し，定めた条件に沿ってモデルサイトを実装する．第 6 章では実装したモデルサイトについて，既存手法により実装したものと比較し評価をとる．第 7 章では本論文のまとめを示し，課題及び今後の展望を述べる．

第2章　フロントエンド開発の現状と本研究における問題点

本章では，Web サイト制作における CSS 設計開発に関連する技術や手法，また本研究で着目する CSS 設計思想について述べる．

2.1　フロントエンド開発環境

本節では本研究において前提とする背景をまとめる．

2.1.1　コンポーネント

多くのフレームワーク及びライブラリが登場する中で，共通して見られるようになってきた概念がコンポーネントである．コンポーネントは，複雑かつ肥大化するコード群の管理可能性を高めるため，インターフェースをある特定の機能ごとに切り分け独立して実装するものである．これは JavaScript ではコンポーネント指向のプログラミングまたはフレームワークと呼ばれ，CSS では設計思想がコンポーネントについて説明している．

コンポーネントを Web 標準で実装する試みが Web Components[3] である．Web Components では HTML を ShadowDOM と呼ばれる技術で全体のコードから独立させ，その独立した個々の要素の中に独自のスコープを発生させることができる．

このような歴史，そして標準化の流れから，これからのフロントエンドはリソースをコンポーネント，つまり機能ごとのかたまりに切り分ける形を前提とすると考えられる．

2.1.2　HTTP/2

2015 年に公開された HTTP/2 によって，上述のフロントエンド開発にさらなる変化の要因がもたらされた．HTTP/2 の特徴のうち主要なものに全てのサイトでの https の適用，ヘッダの縮小，ストリームによる多重通信，サーバプッシュなどが挙げられる．HTTP/2 は既に標準化され普及が進んでおり，今後 Web の基盤となっていくと考えられる．

このうちフロントエンドではストリームによる多重通信が特に大きな意味を持つ．これまでリソースファイルの数が通信にかかる負荷が高いことを前提として開発されてきたために複雑なコードによって非効率な回避策を講じていたが，多重通信により負荷を考慮に入れず

に済むようになれば，そうした回避策は通信への最適化というメリットに非効率なデメリットが勝り不要な技術となる．

この HTTP/2 の登場は本研究において最も注目する背景であり，CSS 開発におけるその影響は第 3 章にて詳しく述べる．

2.1.3 web パフォーマンスにおける CSS

CSS コードは次節にて述べるとおり肥大化しやすく，Web サイトが大規模になるに連れて純粋なリソースファイルとしての巨大化による通信の圧迫が生じる可能性がある．またブラウザの仕様により CSS ファイルの読み込み中に DOM のレンダリングがブロックされるレンダリングブロックの問題から，CSS も JavaScript 同様 Web パフォーマンスに影響を与えうる．

2.2 CSS の問題点

本節では，変動するフロントエンド開発の中で，CSS において根本的な解決するべき問題となる要素についてまとめる．

2.2.1 良い CSS

CSS の問題点を明らかにするため，まず目指すべき良い CSS はどのようなものか定義する．またそれに対比した典型的な悪い例から，CSS コードが引き起こしうる問題について述べる．

まず良い CSS の条件として，Philop Walton が挙げた以下の 4 つの重要な要素 [4] を採用する．

Predictable 予測可能性

Reusable 再利用可能性

Maintainable 保守可能性

Scalable 拡張可能性

予測可能性はコードが一見して予測できる通りに動作することを指す．再利用可能性は文字どおりコードを複数箇所で利用できるようにすることである．保守可能性は修正やスタイルの追加を行う際に，変更する部分以外のリファクタリングを必要としないこと．拡張可能性はサイトの規模の変化や重なるデザインの追加に耐えうる設計であることを指す．

次に，上に挙げた良い CSS の条件が全く守られていない悪例を以下に示す．

最終的に適用されるスタイルの判定が困難な例

HTML

```
<section id="id">
  <div class="selector">
    <span class="span">
      この文字は何色になるのか？
    </span>
  </div>
</section>
```

CSS

```
.span { color: red; !important}
div .selector { color: blue;}
#id { color: pink;}
.selector span { color: black;}
```

HTML 中に定義された文字列に適用されるスタイルを判断しようと試みると，このコードは予測可能性に乏しいということがわかる．スタイルが多重に指定されており，かつ ID，クラス及び要素セレクタと指定に利用しているセレクタの種類が混在しているため，一見してどのスタイルが優先され最終的に適用されるのか判定が困難なのである．

スタイル指定が HTML 側の階層に依存している 2 行目及び 4 行目のコードは，仮に同じスタイルを階層構造が異なる別の箇所に適用しようとしてもそのまま使うことができず，別の箇所のためのスタイルが同一で新たなセレクタ指定を記述するか，元のセレクタを書き換えて複数箇所に対応可能なコードにしなければならない．また ID をセレクタに使用している 3 行目は，1 つの Web ページ中に同名の ID を複数使用することができないことから ID に直接依存しており，その ID が指定された要素以外の要素にスタイルを指定することができない．これはコードの再利用可能性，そして新しい追加に既存コードのメンテナンスが必要という点で保守可能性に抵触する．

最後に拡張可能性であるが，ここまでに述べた通り予測可能性や保守可能性に乏しいのでコードの拡大に耐えうる状態ではない．また，に適用する .span などクラスセレクタの命名が指定するスタイルの役割を説明しておらず意味がないため，規模の拡大に伴って命名を続けるうちにクラス名が同一のスタイルが発生しコードに予想外の挙動をもたらす可能性がある．

以下に例示したコードをリファクタリングした例を示す．

リファクタリング例

HTML

```
<section id="caution">
  <div class="wrapper">
    <span class="alert">
      この文字は何色になるのか？
    </span>
  </div>
</section>
```

CSS

```
.alert { color: red;}
```

大きく変更した点はクラス及び ID の命名に機能的意味をもたせたこと，また不要なスタイルを削除したことである．これにより一目で要素に適用されるスタイルが判断可能となり，予測可能性が大幅に向上し，再利用可能性，保守可能性及び拡張可能性も同様に向上した．

この例で削除したような不要なコードは実際の開発でも発生する．2012 年の Ali Mesbah, Shabnam Mirshokraie による研究では，実運用される web サイトにおいて 1 つのサイトにつき CSS コードの 60%は使われていない [5] という結果が出ている．このような不要なコードは，例として入れ子になったある要素の中で親要素の指定を子要素の指定が上書きし，かつ変更や拡張に応じて親要素の指定に該当する文書が存在しなくなるといった命名などに気をつけていても防げない原因が考えられるが，予測可能性を担保することで修正を容易にし極力減らすことは可能である．

このように，CSS 開発はスタイルの指定方法によって必要以上に大量のコードが書かれ，大量であるためにコードは管理困難になり，場当たり的対処によってさらにコードが増大していくという悪循環の可能性を孕んでいる．

この問題の原因はグローバルな値指定，詳細度による複雑なスタイル決定，値やスタイルの抽象化を行う機能がないといった CSS の性質に切り分けることができる．次節以降にその詳細を述べる．

2.2.2 グローバルな値指定

CSS のスタイル指定では 1 つの HTML に定義されたスタイル全てが考慮の対象になる．つまり全てのスタイルがグローバルである．そのためブラウザ標準の指定からユーザ指定，開発者が作成し適用したあらゆるスタイルコード全てに値が衝突する可能性があり，開発者は常にそれを意識しなくてはならない．

2.2.3　詳細度による複雑なスタイル決定

全てがグローバルなスタイル群は，カスケーディングの概念と詳細度によって最終的に特定の要素に適用されるものが決定される．カスケーディングはブラウザによるスタイルなどスタイルを指定するファイルの種類ごとに優先順位を定め，コードの継承と上書きについて定義するものである．詳細度はこのカスケーディングに優先してスタイルの適用順を定めるものである．この詳細度がコードの予測可能性や保守可能性を脅かす要因となる．

2.2.4　値やスタイルの抽象化を行う機能がない

CSS には一般的なプログラミング言語などに見られる値の抽象化を行う機能がない．具体的にいえばループや条件分岐，変数や関数の指定ができず，コード全体の構造を管理することができないのである．CSS コードの見通しがどうしても悪い原因に，この抽象化がないということが挙げられる．特に表示する要素自体の変更が発生しやすい Web アプリケーションでは，こうした CSS コード自体に抽象的な構造を持たせ操作することが必要とされている．

2.3　CSS の問題を解決する既存技術

本節ではここまでに述べた CSS の問題を解決するための既存技術についてまとめる．

2.3.1　既存技術の種別と概要

以下に列挙するのは主要な既存技術とその概要である．

CSS 設計方針
: CSS 設計方針は開発者自身がコーディングにおいて遵守するべき指針群であり，汎用的なコーディング規約である．コンポーネントという単位で CSS を区切ることでコード可読性を飛躍的に向上し，またスタイルの相互作用をそのコンポーネント内のみに抑えるため保守性をも高めることができる．コーディング規約とは異なり，

フレームワーク
: CSS フレームワークは CSS における汎用的な機能やレイアウトを任意に使用可能なものとして用意したコード群である．グリッドシステムなど汎用性のある枠組みのみを抜き出して使う場合や，アプリケーションの仮組みでフォームやメニューバーなどデザインをそのまま利用する場合がある．代表的なものに Bootstrap[6] がある．

プリプロセッサ
: プリプロセッサは，開発者があらかじめ設定された独自のルールに基づいて記述したコードをブラウザが解釈する CSS コードに変換する仕組みである．主に CSS の言語機

能を拡張し，変数や for 文を擬似的に実現するために利用される．Sass[7] などの CSS メタ言語に代表される．

ポストプロセッサ
: ポストプロセッサは，開発者が記述，あるいはプリプロセッサによって変換された CSS コードにさらに処理を施し，よりブラウザに最適化する仕組みである．ブラウザ間で標準化されていないスタイルのベンダープレフィクスを自動付与する Autoprefixer[8] などがある．またこの Autoprefixer から発展して，CSS コードのパースを担い API の提供によりプリプロセッサ及びポストプロセッサの機能をカスタマイズして利用できる汎用システムである PostCSS[9] が利用されている．

CSS Lint
: CSS Lint は CSS における文法的間違いや体裁，あいまいな表現などを規定するルール，およびそれに反する記述を指摘するツールを指す．C 言語においてソースコードに対し厳密なチェックを行うプログラムを Lint と呼ぶが，その CSS 版である．Web サービスとして提供されているものの他，ポストプロセッサでの処理の一部として利用される．代表として CSS Lint[10]，stylelint[11] が挙げられる．

CSS in JS
: CSS in JS は，JavaScript で CSS を管理するという一連の思想およびツールである．この思想は大きく 2 派に分かれる．一方は JavaScript ライブラリである React[12] での開発から生じたもの [13] で，完全に JavaScript にスタイル指定を委ねるため JavaScript コード上でスタイルを記述する．一方は，CSS は CSS としてスタイルを記述し，それを JavaScript で操作することによって擬似的な名前空間の実現などを行う．後者は CSS modules[14] に代表される．

ビルドシステム
: ビルドシステムは，前述の CSS メタ言語の機能などによって分割されたファイルやその他のリソース，またプリプロセッサやポストプロセッサに該当するツールの処理をまとめて実行するプログラムである．代表的なものに gulp[15] がある．より広範囲のリソースを管理するものに webpack[16] がある．

スタイルガイド
: スタイルガイドは，CSS が具体的にどのように HTML に適用され，どのようなデザインの実現が期待されるかを一覧として示すマニュアルのようなもの，およびそれを自動生成するツールである．CSS コード中にコメントとして必要な要素を記述するものが多い．フレームワークの利用マニュアルもこのスタイルガイドの形を取っていることがある．

このように多くの技術が開発され，利用されている．またここに列挙した技術は 2018 年時点で利用されているものであるが，全ての技術が確立されているわけではなく，相互の兼ね合いや JavaScript の開発環境への適応のため日々統合や衰退，そして新しい概念の登場

が繰り返されている．この動きは特にプリプロセッサ及びポストプロセッサ，CSSinJS，ビルドシステムについて顕著である．

2.3.2 CSS 開発の流れ

本項では前項に述べた既存技術の役割をより明確にするため，実際の開発の流れに沿って概要を説明する．

まず，定められたデザインを元に Web ページを開発する例について，想定される基本的な流れを以下にまとめる．

1. デザインの切り分け
2. 必要であればフレームワークの選定
3. CSS 設計思想の選択及びコーディング規約の設定
4. プリプロセッサによる記述・変換
5. ポストプロセッサによる加工
6. テスト，実サイトに適用

スタイルガイドを生成する場合，プリプロセッサによる記述にその記述が含まれる．また JavaScript で全てのスタイルを指定する CSSinJS については，プリプロセッサによる記述やポストプロセッサによる加工ではなく JavaScript フレームワークでの記述が行われる．CSS lint はポストプロセッサに含まれる．

2.4 本研究における課題

本項では本章でまとめた背景と CSS 開発における問題点及び既存技術から，本研究において着目する点及び解決するべき課題について述べる．

本研究が根底に掲げる課題は，CSS のパフォーマンス改善である．CSS がパフォーマンスに影響を与える要素は前述した通りであり，これは 1 つの web ページおけるファイルの数，コード全体のファイルサイズ，レンダリングブロックの 3 つに区別することができる．この中で本研究ではファイルの数及びレンダリングブロックの問題に着目する．本章でまとめた既存の CSS コーディング関連技術は多くがファイルサイズの削減を解決するものであり，本研究で着目する他の 2 つの問題はあまり検討されていないためである．また，この 2 つの問題はどちらも CSS コードの分割に関する問題であり，双方を同時に検討することに意義があると考えられる．さらに，HTTP/2 の登場によりファイルリソースの増減に関する方法論の転換が行われるとされるため，これらのパフォーマンスにおける問題を改めて検討することは重要であると考える．

また，CSS コードの分割について検討するにあたって，既存技術として述べた CSS 設計思想について着目する．CSS 設計思想は実運用上の web サイトに適用する CSS ファイルの分割について検討するものではないが，デザインをいかに解釈しコードを管理容易な単位に分割するかを検討しているためである．さらに，設計思想は CSS 開発において他の言語による開発に比べても非常に重要であると考える．CSS の言語的拡張やスコープの実現は多くの技術によって実現がなされてきているが，CSS は HTML の構造に依存しており，また JavaScript と異なり CSS から HTML の要素自体を操作することはできないという原則は変わらない．これは開発者自身が依存関係を意識しなければならないということであり，開発者自身による設計が大きな意味を持つということである．CSSinJS のように HTML 及び CSS を含むすべてを JavaSctipt で操作するという解決策も提示されてはいるが，現状 JavaScript 自体にそのような運営に最適化された機能があるわけではなく特定のフレームワークやライブラリといった拡張技術に依存しているというデメリットがある．

2.5 本章のまとめ

本章では，CSS コーディング最適化を考えるにあたり前提となるフロントエンド開発の現状，CSS の根本的な問題点及びそれに対する既存の解決策についてまとめ，本研究において着目する課題について述べた．本研究では HTTP/2 による今後のフロントエンド開発の転換を念頭に起きつつ，既存の最適化手法及び CSS 設計方針を比較検討することにより，より良い最適化手法について考察する．

第3章　CSS開発における最適化手法の検証

本章では，CSS 開発においてパフォーマンス向上の観点から定石とされる方法論を整理し，それらを検証する先行研究の紹介及び独自の実験について述べる．

3.1　最適化手法

本節では既存の最適化手法として，パフォーマンスの視点から CSS 開発において取り入れられる主なコーディング規約をまとめる．

3.1.1　ネストの限定

CSS コードにおいて，セレクタを多重にネストすることは推奨されていない．セレクタ指定におけるネストとは，以下のような状態を指す．

ネストの深いスタイル指定

```
#id section div div div span {~}
```

このようなセレクタ指定を行うと，ブラウザは 1 つひとつセレクタを検証して該当する HTML 要素を選択するため処理に時間がかかる．また，左から解釈すれば`#id`によって候補を絞っているように見えるが，実際にはブラウザは右からセレクタを解釈するため，まず全ての`<span>`要素を検索し，`<div>`要素の子要素であるものが見つかったら次に全ての`<div>`を同様に検索して，最後に`<~ id="id">`が指定された要素の子要素であることを検索する．ネストはコードの再利用可能性などの観点から多用するべきではないが，パフォーマンスにおいても効率が悪い．

3.1.2　ユニバーサルセレクタの扱い

本項では CSS セレクタの処理速度の違いについて述べる．CSS セレクタは 4 種類存在するが，それぞれ処理にかかる速度が異なる．以下のリストに示すように，ID セレクタが最も速く，ユニバーサルセレクタが最も遅くなる．

1. ID(最も速い)

2. クラス

3. タグ

4. ユニバーサル (最も遅い)

ユニバーサルセレクタはパフォーマンスの観点から多用するべきではない．

3.1.3 ファイル結合

Web サイトにおいて，表示に要する時間を左右するのは通信容量ではなく通信の過程で発生するレイテンシである [17]．ブラウザ，つまりクライアントがサーバとの接続を確立し，リクエストを送信し，データを受信する一連の処理の繰り返しが蓄積することによって，全体の表示時間が遅延するのである．`HTTP1.0` 及び `HTTP1.1` では通信接続の確立を原則 1 つずつ行うため，通信する必要のあるファイルの増加に対応してレイテンシが増大していく．このためデータは可能な限り結合しファイルの数を減らすことが肝要であった．具体的には CSS ファイルは 1 つの web サイトにつき 1 つのファイルにまとめること，あるいは最大限 HTML にインライン化することが挙げられる．

ここで，HTTP/2 では通信の多重化が可能であるため，同時に多くのファイルを処理できるとされている．そのため，HTTP1.1 での対策はデメリットをもたらす可能性がある．CSS スプライトは複雑な CSS コードを要しレンダリングの負荷も余分にかかるため，小さなアイコンファイルの呼び出しにかかる負荷を無視できるのであれば，ファイルは適切に分割し 1 つのファイルで 1 つの画像を表示するべきである．また，画像や CSS のインライン化についても，個別にファイルを管理することによる見通しのよさを享受すべきであると言われる．

3.1.4 CSS スプライト

CSS スプライトは，アイコンなどの小さな画像を複数使用する場合，いくつかの画像を 1 つにまとめ CSS で表示領域を限定して複数の画像を 1 つのファイルで表示するものである．通信を要するファイル数を削減するメリットがある一方で，煩雑なファイル作成及び CSS コードの記述を要するデメリットがある．

3.1.5 コンポーネントごとの link タグ配置

本項では，第 2 章で述べたレンダリングブロックの緩和策を示す．これは CSS ファイルが巨大である場合，サイトにアクセスした時にすぐにコンテンツが表示されずユーザ体験を損ねる可能性がある．これを解決するために CSS の読み込みを分散及び遅延させる必要がある．CSS ファイルを適切に分割し読み込む場所をコンポーネントごとに設定することが有効であると考えられる．

3.1.6　キャッシュ効率の向上

ブラウザが一度訪れた web サイトのデータを保存しておき次回アクセス時に利用できるようにするキャッシュを有効に利用するため，一部のコードの変更がもたらす影響を最小限に抑えることでパフォーマンスの改善を行うことができる．CSS ファイルにおいても変更が生じる頻度ごとにコードの記述ファイルを分割することでキャッシュ効率の向上を図ることができる．

3.1.7　まとめ

ここまでに述べたように，パフォーマンス改善の方法論には対立する方法論が存在する．ネットワークに関わる部分であり，HTTP1.1 以前での方法論とされるファイルを極力結合する方法論は，HTTP/2 以降に提唱されるファイル分割の方法論，そしてレンダリングブロックの緩和策であるコンポーネントごとの link タグ配置及びキャッシュ効率の向上の前提となる CSS ファイルの分割に対立するものである．パフォーマンス改善への最適解を探るにあたって，これらの方法論を改めて検証する必要がある．

3.2　先行研究

本節では，ここまでに述べた Web パフォーマンス最適化手法について検証した先行研究をまとめる．

3.2.1　パフォーマンス測定指標

Web パフォーマンスの計測は様々な要因が関係するため，ノイズのない確実な指標を設定するのが困難である．これまでにパフォーマンス測定の指標には大きく分けて 2 種類の指標が提唱されている．

1 つが W3C が標準化している `Web Performance API`[18] により取得できる各種の数値である．この API は W3C により標準化されているためにどのブラウザでも利用でき，実際にユーザがアクセスする際の状況に近い数値が取りやすいというメリットがある．取得できる数値は以下の通りである．

navigationStart
: 直前のドキュメントをアンロードしようとした時刻．直前に表示されているドキュメントが存在しない場合，fetchStart と同じ値を返す．

unloadEventStart
: 直前のドキュメントのアンロードを開始した時刻．直前に表示されているドキュメントがない，または直前に表示されているドキュメントとオリジンが異なる場合は 0 を返す．

unloadEventEnd

直前のドキュメントのイベントアンロードが終了した時刻．直前に表示されているドキュメントがない，または直前に表示されているドキュメントとオリジンが異なる場合は 0 を返す．

redirectStart

リダイレクト開始時刻．リダイレクトしない場合は 0 を返す．

redirectEnd

リダイレクト終了時刻．リダイレクトしない場合は 0 を返す．

fetchStart

関連するキャッシュの検索を開始した時刻．

domainLookupStart

ドメイン名解決の開始時刻．

domainLookupEnd

ドメイン名解決の終了時刻．

connectStart

ドキュメント取得のためのサーバとの接続確立開始時刻

connectEnd

ドキュメント取得のためのサーバとの接続確立終了時刻

secureConnectionStart

通信が HTTPS で行われている場合，接続を安全なものにするためのハンドシェイクプロセスを開始した時刻

requestStart

ドキュメントのリクエスト開始時刻

responseStart

サーバからドキュメントの最初の 1 バイトを受信した時刻

responseEnd

サーバからドキュメントの最後の 1 バイトを受信した時刻

domLoading

`Current document readiness` 要素が `loading` に設定された時刻

domInteractive

`Current document readiness` 要素が `interactive` に設定された時刻

domContentLoadedEventStart

`DOMContentLoaded` イベントの起動時刻

domContentLoadedEventEnd

`DOMContentLoaded` イベントの処理完了時刻

domComplete

`Current document readiness` 要素が `complete` に設定された時刻

loadEventStart

ロードイベントの起動時刻

loadEventEnd

ロードイベントの処理完了時刻

これらの数値は単独ではなく，差分を比較検討することによって様々な観点からパフォーマンスを計測することができる．

もう 1 つに挙げられるのが，2012 年に Google 社がパフォーマンス測定サービス WebPageTest[19] に導入したページ表示速度を計測する `Speed Index`[20] である．

Navigation Time API でページの読み込みが完了するまでの数値は正確に測定することができるが，実際にページの表示が開始された時刻を正確に測ることはできない．そこでブラウザ上でどれほど表示がなされたかビデオ処理を用いて取得する Render Start が開発され，それを記録とした図 3.1 のような Film Strip が利用可能となった．

図 3.1: Film Strip の例．[20] より引用．

Speed Index はこの Film Strip を利用し，一定の時間感覚でページが表示されている割合を測定し，そのグラフ上の面積を指標化したものである．これにより，単にページが表示される時間だけではなく，ユーザがどれだけ速くより多くのページ内要素を目にすることができるかを測れるようになった．具体的には図 3.1 の例において，以下の図 3.2 のように，ページの表示完了時刻は一致していても 1 秒時点で 93%表示されていたものの方が 18%のみ表示されていたものに比べ数値が低くなる．Speed Index の値が低ければ低いほどユーザの体感的な待ち時間が少なくなると言えるため，パフォーマンスが優れていると言える．

以上のように，フロントエンドにおける Web パフォーマンスは Navigation Timing API あるいは Speed Index などの指標を組み合わせて測定される．

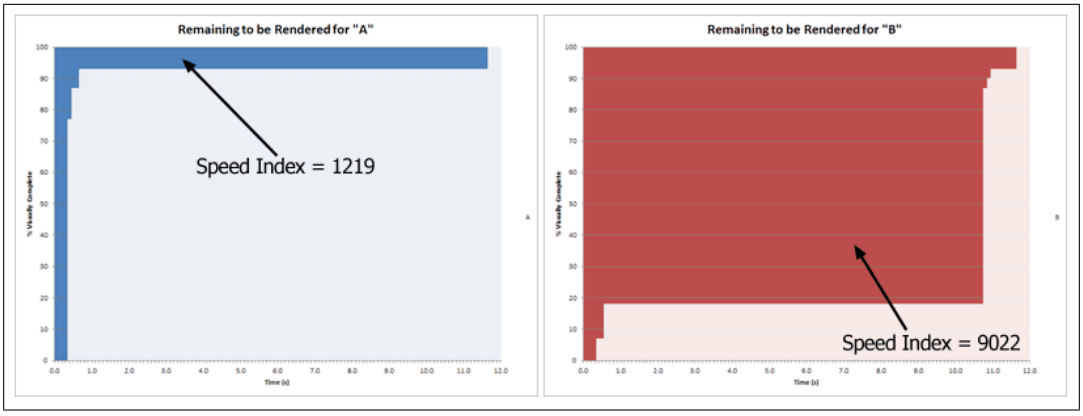

図 3.2: Film Strip の具体例．[20] より引用．

3.2.2 HTTP/2 環境における HTTP1.1 での最適化手法検証研究

HTTP/2 環境における HTTP1.1 での最適化手法を検証した先行研究として，HTTP1.1 以前における定石を HTTP/2 環境で検証した Robin Marx, Peter Quax, Axel Faes and Wim Lamotte らによる研究 [21] がある．

この研究では HTTP1.1 におけるパフォーマンス最適化手法として CSS スプライトを用いる画像の結合，CSS 及び JavaScript ファイルの結合について SSL を用いない HTTP1.1 及び SSL を用いた HTTP1.1，そして HTTP/2 環境におけるパフォーマンスを測定し，比較検討している．測定の条件は図 3.3 の通り，プロトコル及び通信条件，ブラウザを複数使用し組み合わせたものである．この測定の指標には，前項に挙げた Navigation Timing API から `loadEventEnd`，及び Speed Index が利用されている．

Protocols	HTTP/1.1 (cleartext), HTTPS/1.1, HTTPS/2
Browsers	Chrome (51 - 54), Firefox (45 - 49)
Test drivers	Sitespeed.io (3), Webpagetest (2.19)
Servers	Apache (2.4.20), NGINX (1.10), NodeJS (6.2.1)
Network	- `DUMMYNET` (cable and cellular) (Webpagetest) - fixed `TC NETEM` (cable and cellular) - dynamic `TC NETEM` (cellular) (Goel et al., 2016)
Metrics	All Navigation Timing values (Wang, 2012), SpeedIndex, firstPaint, visualComplete, other Webpagetest metrics (Meenan, 2016)

図 3.3: 最適化手法の検討実験における条件．[21] より引用．

この研究の結果，CSS スプライト及び CSS ファイルや JavaScript ファイルのは HTTP/2 においても有効であること，ただし HTTP/2 においては，通信環境が悪化した際に受ける影響が HTTP/1.1 よりも少ないことがわかっている．

CSS ファイルの結合における `loadEventEnd` の測定については図 3.4 に見られるように，

シンプルなスタイルを含むファイルの適用において 30-40 以上のファイル数を適用した際に http1.1 と比較してはっきりとした差があらわれることがわかっている．

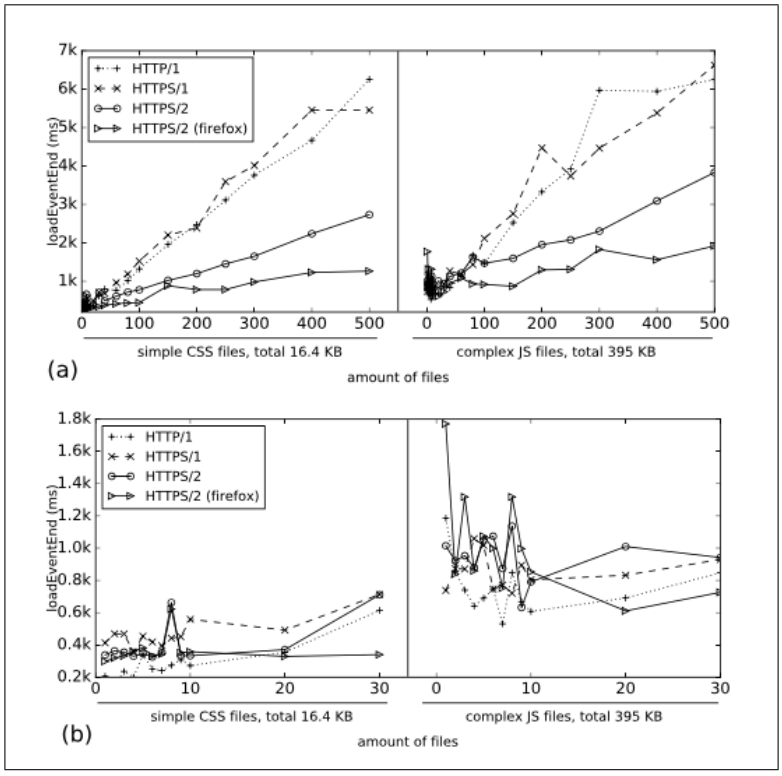

図 3.4: 多数の CSS 及び JS ファイル結合に関する測定実験結果．[21] より引用．

このグラフを見ると，HTTP/1.1 に比較して HTTP/2 でのファイル数の増加によって生じる通信時間の増加は非常に緩やかである．また特筆すべきは firefox での HTTP/2 での通信時間であり，ファイル数が 100 を超えるまでほとんど増加しない．

また同一の条件で Speed Index についても測定されており，CSS ファイルについては `loadEventEnd` の場合と同じ傾向を示すことがわかっている．

以上のことから，HTTP/2 は HTTP1.1 に比較して，確かにより多くのファイルを分割した状態において高いパフォーマンスを示すということがわかる．ただし分割した方が早いということはなく，ファイルの結合は依然有効であることがわかった．よって HTTP/2 においてはファイル分割がパフォーマンス改善に有効であるという方法論は成立しない．

3.3 検証実験

本節では，前節でまとめた先行研究の結果と対立する方法論である，CSS ファイルを分割しそれぞれ対応するコンポーネントの直前に適用することでレンダリングブロックを抑えパフォーマンスを向上する方法論について検証する．

3.3.1 検証に用いる指標

本実験において，検証に用いる指標はNavigation Timing APIよりロード時間計測の指標として`loadEventEnd`から`navigationStart`を引いた差分及びSpeed Indexとする．ロード時間の計測は`loadEventEnd`及び`navigationStart`の差分を採用するのは，ユーザがページを表示しようと行動を起こした瞬間からのより正確な数値が測れると考えるためである．

3.3.2 実験環境

前節に挙げた先行研究における実験環境である`Speeder Framework`は情報が掲載されているが，現在はオープンソースでの利用が停止されているため，本実験では独自に環境を構築する．構築した実験環境は以下の表の通りである．

表 3.1: パフォーマンス計測実験環境

プロトコル	HTTP/2
ブラウザ	Google Chrome 63
サーバー	Virtual Box, Vagrant, Node.js(v9.4.0)

ロード時間の計測は直接実験用HTMLファイルにスクリプトを適用することで行う．また，Speed Indexの計測はGoogle Chromeのディベロッパーモードを利用しデータを取得，そしてPierre-Marie Dartusらによるnode.jsモジュールである`speedline`を利用して計算することで行った．

3.3.3 実験結果

実験に用いたHTML及びCSSファイルには，以下のようなHTML要素を10個含んだHTMLファイルを用意しそれぞれの要素に連番でクラスを付与，そして各クラスに10行程度のスタイルを付与したCSSファイルを作成した．

実験用HTMLコードの一部

```
<div class="class1">
<span class="class1_1">Lorem ipsum dolor sit amet, consectetur
 adipisicing elit. Qui voluptate corporis, iure similique ab
 beatae asperiores eius aperiam vitae laudantium.
 Fugiat accusantium molestiae reprehenderit,
 omnis! Quasi minus, voluptas blanditiis adipisci?</span>
</div>
```

この HTML 及び CSS ファイルについて，全ての CSS コードを 1 つのファイルに結合した状態で<head>内に CSS ファイルを呼び出したもの，CSS コードを 10 個の HTML 要素に合わせて 10 のファイルに分割し，それぞれの要素の直前で CSS ファイルを呼び出したもの，そして CSS コードを 10 のファイルに分割し，それを全て<head>内で呼び出したものについて，それぞれロード時間及び Speed Index を 5 回ずつに渡って計測した．計測結果は図 3.5 及び図 3.6 の通りである．

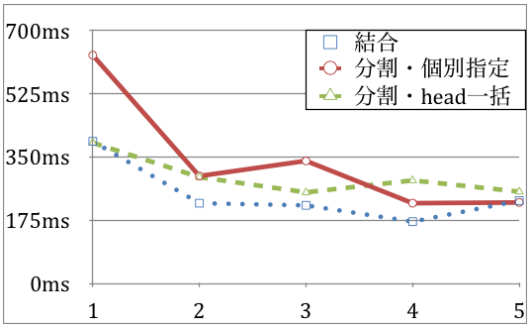

図 3.5: ロード時間計測結果

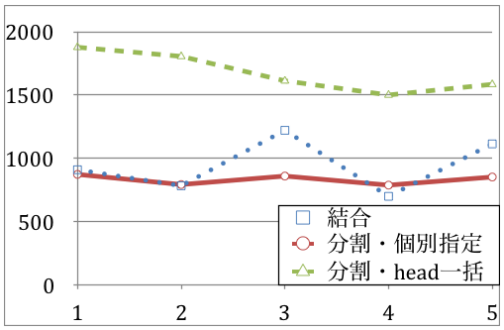

図 3.6: Speed Index 計測結果

まず，ロード時間の計測については，わずかであるが先行研究にある通りファイルを結合したものが最もパフォーマンスが良い傾向にある．しかし 1 回目を除いて大きな差はないことがわかる．

次に Speed Index の計測については，ファイルを分割し<head>内で全て指定したものが圧倒的に高い数値を示し，パフォーマンスが悪いことがわかった．分割した上で個別に指定したものと全て結合したものについてはほぼ差異がなく，分割したものが若干安定している．この差異はブラウザでのロードの順序と複数ファイルにおける処理と単一ファイルに置ける処理の違いにより生じるのではないかと考える．

以上の結果により，ファイル分割をする際は<head>内にファイルを呼び出すのではなくコンポーネントごとに呼び出す必要があることがわかった．しかし結合した場合と分割し個別にファイルを呼び出した場合の優位な差は見られず，ファイルの分割が結合に比べパフォーマンスの最適化に有効とは言えないとわかった．

3.4　本章のまとめ

本章では CSS コーディングにおいて Web パフォーマンスを改善するための最適化手法についてまとめ，特に通信に関わる部分について検証を行った．検証の結果，HTTP/2 環境下においてもファイルの結合が有効であること，ただしスタイルの宣言をコンポーネントごとに行う場合はファイル分割を行なってもパフォーマンスが変化しないということがわかった．これにより，キャッシュ効率の観点から分割を採用するのは可能ではないかと考える．

第4章　既存のCSS設計思想

本章では，既存の設計思想について述べる．

4.1　調査した設計思想

設計思想は世界中の多くの開発者により個々に提唱されており，また開発プロジェクトそれぞれにおいてアレンジを加え利用されることもあるので，全てを挙げるのは不可能である．しかし，ここでは調査できた主な既存の設計思想を列挙する．

OOCSS[22]
Object Oriented CSS を掲げ，CSS コードで定義するスタイルにおける機能と外観の区分を提唱する

BEM[23]
Block，Element，Modifier の 3 要素を定めたシンプルかつ厳格な命名規則を主体とした思想

SMACSS[24]
Base，Layout，Module，State，Theme の 5 つにスタイルを分類し管理する

FLOCSS[25]
Foundation，Layout，Object の 3 つにスタイル機能を区分し，Object に属するものとして Component，Progect，Utility の区別を定義する

ITCSS[26]
詳細度順に CSS を記述することを提唱し，スタイルの機能を Settings，Tools，Generic，Elements，Objects，Components，Utilities の項目に分類する．

SUITCSS[27]
Style tools for UI component とし，Utility，Component の大きな 2 分類，そして Component をさらに modifier，desendant，state の 3 種に区分する命名規則を提唱する．またその命名規則適用のため独自のプリプロセッサを提供している．

Maintainable CSS[28]
Module，Component，State, Modifier の 4 要素で構成される命名規則を提示する．

MCSS[29]

マルチレイヤーを主題に掲げ，Base，Project，Cosmetic の 3 つのレイヤーを定義する．

AMCSS[30]

Attribute Modules という独自のモジュールを提供し，HTML タグの属性を新たに設定する形で CSS を記述する．

Enduring CSS[31]

Micronamespace，Component，ChildNode の 3 要素で構成された，HTML 要素の意味よりも CSS コーディング自体の利便性をより意識した命名規則を核にした思想．スタイルの重複を許可する．

FUN[32]

Enduring CSS の作者による CSS 開発において意識するべきことの指摘であり，セレクタ間の詳細度などの力関係が均一であること (Flat hierarchy of selectors)，Atomic CSS のような 1 つのクラスに 1 つのスタイルを定義した汎用クラスの使用 (Utility styles)，名前空間を付与したコンポーネント群 (Name-spaced components) を掲げる．

Atomic CSS[33]

1 クラスにつき 1 つのスタイルを定義する思想

4.2 設計思想の構成要素

本節では，調査した設計思想が CSS 開発におけるどのような条件を主に提示しているかについて述べる．

4.2.1 文法的なコーディング規約

多くの設計思想は，CSS の言語的性質を考慮した禁止事項の設定などを含む文法的な事柄を定めたコーディング規約を含んでいる．例としてネストの量を規定するものや要素セレクタの使用に関する制限が挙げられる．

またほとんどの設計思想で採用されている重要な前提がクラスセレクタの使用である．再利用可能性のない ID セレクタによるスタイル指定や負荷の高い属性セレクタを禁じ，あくまでもクラスセレクタの命名によってコードを構成することで，詳細度の問題やネストによる不要な制約及び混乱を回避することができる．例外として HTML タグに要素を追加し要素セレクタを利用する AMCSS がある．

4.2.2 命名規則

クラスセレクタを中心に CSS を設計するにあたって，最も困難であり重要なのがクラスの命名である．BEM，Maintainable CSS，Enduring CSS はこの命名規則を核に思想を提

案している他，全ての設計思想でクラスの命名について少なからず触れている．

調査した中で最も多くの思想に影響を与えている命名規則はロシアの Yandex 社が提唱する BEM である．この思想は以下のように命名規則を提示する．

BEM の命名規則

```
.block__element_modifier {~}
.block__element {~}
.block_modifier {~}
```

まずはコンポーネント及びモジュールにあたる単位として `block` で名前空間を形成し，子要素にあたる `element` 及び色違いなどの小さな分岐要素である `modifier` を区別して定義する．この `element` 及び `modifier` は入れ子にすることも可能である．定義した要素が `element` なのか `modifier` なのか判断可能にするために，`element` は接続にアンダーバー 2 つを使用し，`modifier` はアンダーバー 1 つを使用することを定義している．

このようにアンダーバーやハイフンを駆使していくつかの要素を判別可能な状態で使い分ける手法は数多くの設計思想で採用されており，以下に示す Enduring CSS の命名規則もその一例である．

EnduringCSS の命名規則

```
.nsp-Component_ChildNode {~}
```

この規則において nsp とは `namespace` の省略形であり，Enduring CSS においてはこの，BEM では `block` にあたる要素の命名において，省略形を用いることを提唱している．`-Component` は `__element` にあたり，`_ChildNode` は `_modifier` にあたる要素であり，詳細な定義は異なるが外観の構成は類似している．

また別の例として Atomic CSS の命名が挙げられる．この設計思想は 1 つのクラスに 1 つのスタイルを定義するため，クラス名は以下のように指定するスタイルの内容がわかる最小限の省略形を利用する．

AtomicCSS の命名規則

```
.BgC { background-color: #0280ac;}
.MgT12 { margin-top: 12px;}
```

さらに AMCSS はタグ要素の追加における命名規則について定義している．

命名規則は各設計思想の特徴を明確に表すものであると言える．

4.2.3　複数セレクタによるスタイルの分割

今回調査した設計思想の中ではクラスの役割を最小限に留める Atomic CSS を除く全ての設計思想において，スタイルを機能ごとに切り分け管理する方法が定義されている．その単位の名称は本章冒頭のリストに示したように設計思想によって様々であるが，機能ごとの切り分けという点で第 2 章でフロントエンド開発についてについて述べた際に触れたコンポーネントの概念に当てはまるものであると考える．

コンポーネントは第 2 章で述べた通り，複雑かつ肥大化するコード群の管理可能性を高めるため，インターフェースをある特定の機能ごとに切り分けたものを意味する．本稿ではこれを広義に捉え，以降類似する概念は名称が異なっても基本的にコンポーネントと記載することとする．名称が異なる例として OOCSS におけるオブジェクトが挙げられる．OOCSS のドキュメントではオブジェクトを以下のように定義している．

> Basically, a CSS“object”is a repeating visual pattern, that can be abstracted into an independent snippet of HTML, CSS, and possibly JavaScript. That object can then be reused throughout a site.

この定義は本稿におけるコンポーネントの捉え方に合致するものである．

こうしたコンポーネントによるスタイルの切り分けは，よい CSS コードを実現する上で重要な役割を占める．命名規則と合わせることでグローバルな CSS に擬似的なスコープをもたらし，開発において考慮にいれる範囲を大きく狭める．各設計思想のコンポーネントはファイルとディレクトリによるツリーで表現され，CSS ファイルそのものの管理を統制することにもつながる．

こうしたコンポーネントを定義する思想の多くはプリプロセッサの使用を前提としている．プリプロセッサを利用することでファイルの統合を容易に行うことができる他，変数や関数のような定義が可能であるため，これを前提とすることでより効率的なコンポーネント運営が可能となる．

このコンポーネントの思想が HTTP/2 におけるコーディング最適化への手がかりになると考えられる．

4.3　コンポーネント定義の傾向

本項では CSS 設計思想におけるスタイル分割としてのコンポーネントについて理解を深めるため，調査した既存の設計思想のうちコンポーネントにあてはまる概念を有するものについて種類ごとに分類しまとめる．

4.3.1　配置と装飾の分割

CSS で指定するスタイルは 2 種類に分類できる．それが配置及び装飾である．この考え方は OOCSS に起源を発し，多くの CSS 設計思想がこれに影響を受けている．これは，主にスタイルの再利用性を高めるものである．

4.3.2　レイヤー思考

コンポーネントの捉え方の 1 つとして，レイヤーの概念を用いる考え方がある．スタイルは想定されるその適用範囲や使用されるセレクタの汎用性から，基本の色指定や文字サイズなどページ全体に適用されるもの，レイアウトなど再利用性が低くページ固有のもの，独立したコンポーネントとして広範囲に使用されうるものなどに分類できる．この分類はひとつひとつをデザインのレイヤーと捉えることができる．コンポーネント分割やファイル構成に触れる設計思想はほとんどがこのレイヤーで捉えることができる構造を採用している．この考え方を定義する代表的な思想として MCSS が挙げられる．

以下にレイヤー適用の例を説明する．まず図 4.1 のように Web ページのデザインをコンポーネントごとに分割する．

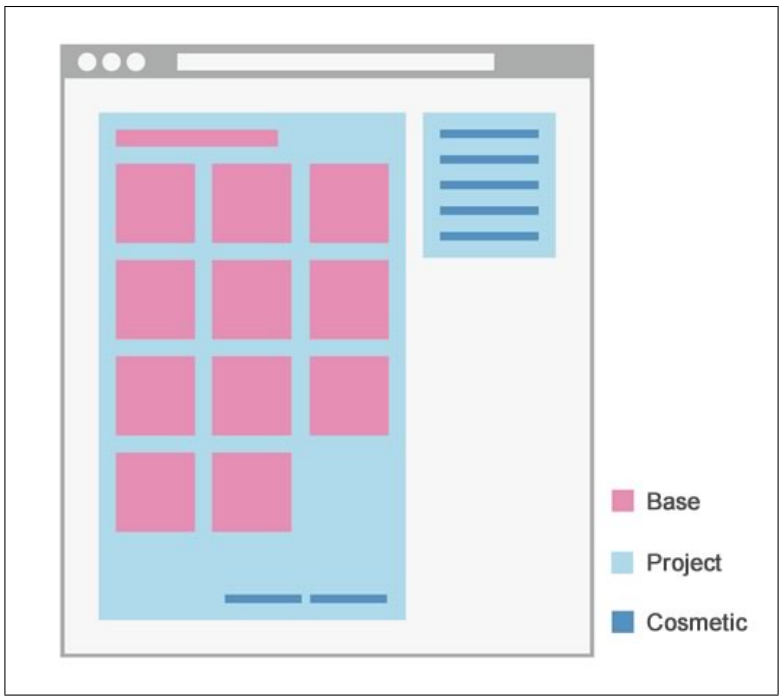

図 4.1: CSS コンポーネント例．[29] より引用．

このコンポーネントは図 4.2 のようにレイヤー構造として考える．MCSS では基本の Base，Project，Cosmetic に加え，ブラウザが指定するデフォルト CSS をリセットするゼロレイヤーや，ブラウザ固有の問題などに対処するスタイルをまとめる context レイヤーを定義している．

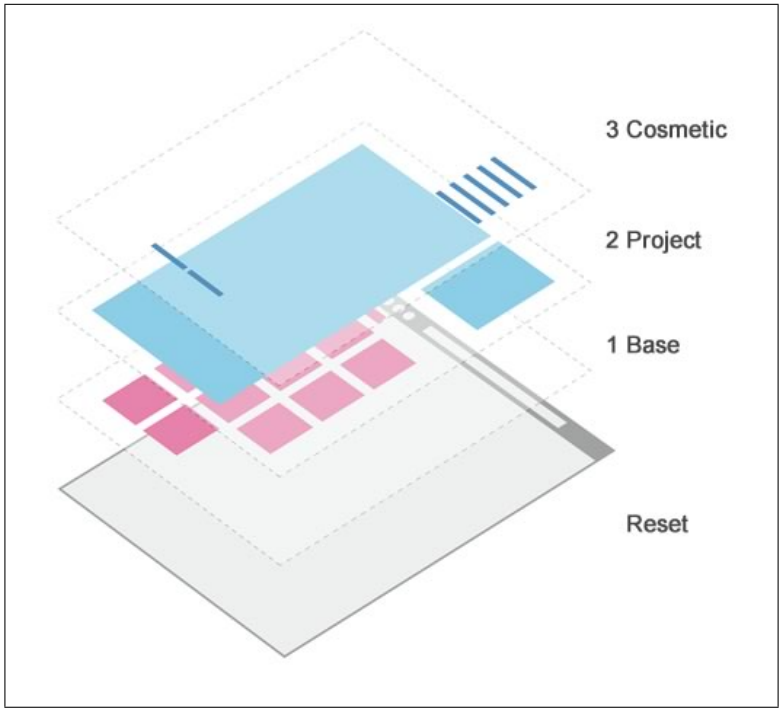

図 4.2: CSS レイヤー図．[29] より引用．

このレイヤー構造により，個々のスタイルの役割をよりはっきりさせ，セレクタの定義を明確にすることができる．また MCSS では同一のレイヤー内でのコンポーネント同士の上書きは容認するが，異なるレイヤーのコンポーネントで定義されたスタイルを上書きすることを禁止し，コンポーネント間の干渉関係をも統制している．

4.3.3 詳細度順

詳細度順のスタイル適用は，スタイル適用の見通しを良くするものである．

第 2 章で述べた通り，最終的に要素に適用されるスタイルは，まず詳細度が高いものが計算され，そしてカスケーディングによりスタイルコードがより後[1]に読み込まれたものが選択される．このことを踏まえ，コードを詳細度順及び優先度に記述し，特定のスタイルの詳細度とカスケーディングの優先度を一致させることで，どのスタイルが適用されるか判別を容易にすることができる．

この詳細度順を主題に掲げているのが ITCSS である．この設計思想が定義するスタイルの定義順は，図 4.3 に引用する概念図が示すように逆三角形の形状で説明することができる．`Explisitness` つまりスタイルが適用されるセレクタの抽象度が高いものから低いものへ，`Specifity` つまり詳細度が低いものから高いものへ，`Reach` つまりそのスタイルが適用される範囲が広いものから狭いものへと順にスタイルを定義する．レイヤーによるコンポーネント分割から派生し，指標の増加という点でこれを強化したものと見ることができる．

[1] !important 属性が付加されたスタイル群についてはより前に読み込まれたスタイルが適用される [34]

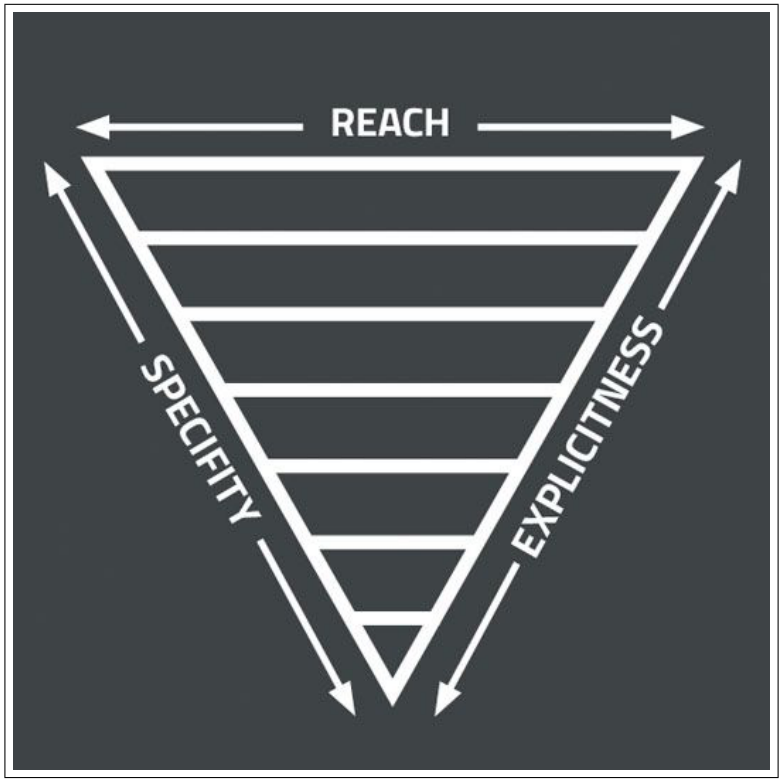

図 4.3: ITCSS による詳細度順のコーディング概念図．[26] より引用．

ITCSS はこの概念に当てはめ，図 4.4 のように名付けた階層構造を定義している．

この詳細度順は常に定義するスタイルのブラウザが読み込む優先度について考慮しなければならないため開発者の負荷が高いが，詳細度をグラフにして表示するツールが開発されているなど，多くの開発者から支持されている．

4.3.4 1 クラス 1 スタイル

Atomic CSS の考え方は上記のどれも含まない例外に当たる．この設計思想は 1 つのクラスにつき 1 つのスタイルのみを定義することを提唱しており，スタイルの適用は全て HTML 上でクラス指定をすることによってなされる．具体的なコードは以下のようになる．

Atomic CSS 適用例

HTML

```
<div class="C00f Fs12">
  青色，12px の文字
</div>
```

CSS

```
.C00f { color: #00f;}
.Fs12 { font-size: 12px;}
```

CSS 開発の方策の中に汎用クラスとして数値の異なる margin スタイル指定クラスを複数

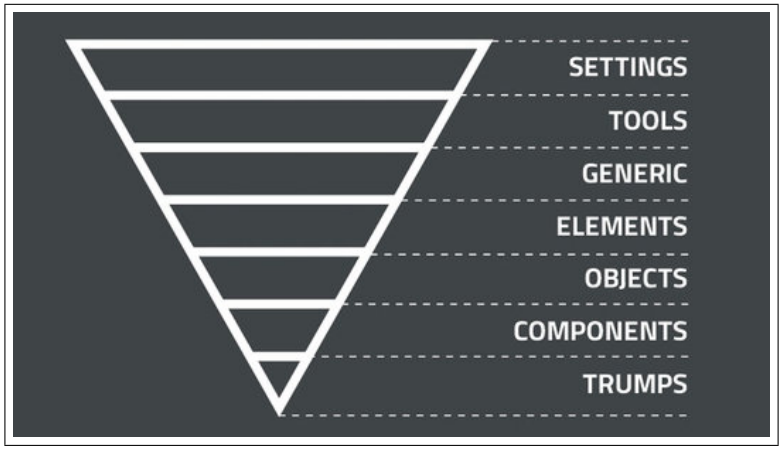

図 4.4: ITCSS における階層命名図．[26] より引用．

用意するものがあるが，それとは異なり，この設計思想では実際に使用するスタイルのみクラスとして用意する．そのため CSS としては大幅なコードの削減が見込める．この思想をコーディングに適用した結果ファイルサイズが減少した例として，Gesture という Web サービスを提供する John Polacek は，Atomic CSS の適用後サービスの CSS ファイルサイズがそれまでの 10%にまで縮小でき，その後 1 年半の運用の中でコードが減ることはあっても増えることがなかったと提示している [35]．

この設計思想におけるスタイルの適用は HTML 側のクラス指定に委ねられるため，HTML に直接コーディングするよりも，JavaSctipt でコードを運用する `CSSinJS` の考え方と非常に親和性が高いものである．

4.4 本章のまとめと考察

既存の設計思想を調査及び整理することによって，設計思想は分類できるいくつかの要素によって構成されていることがわかった．これらの要素はひとつの思想に同時に採用することができないものもあるが，組み合わせることによって CSS コードの統制を強めることができる．また，設計思想は総合していかに CSS を分割するかを規定するものであると言える．これらの考え方を利用し，パフォーマンスの上でより効率的な分割を検討し，新たな思想として取り入れることは有効であると考えられる．

今回調査した CSS 設計思想とその構成要素の内本研究において重要となるコンポーネント分割に関するものを表 4.1 にまとめる．これにより，設計思想 FLOCSS が本研究の目的であるコンポーネント分割を利用したファイル分割の適用に必要な要素を全て備えているということがわかる．第 5 章では第 3 章の結果を元に手法を提案し，この FLOCSS に適用する．

表 4.1: CSS 設計思想分類

思想名	基本文法規則	命名規則	ファイル管理	配置と装飾	レイヤー	詳細度
OOCSS				○		
SMACSS	○	○		○	○	
BEM		○			○	
MCSS			○	○	○	
AMCSS	○	○		○		
SUITCSS		○				
FUN	○	○				
ECSS	○	○	○			
maintainableCSS	○	○	○			
AtomicCSS	○	○				
FLOCSS	○	○	○	○	○	○
ITCSS	○		○	○	○	○

第5章　提案手法

本章では，第 4 章及び第 3 章での検証を元に新たな手法を提案する．

5.1　提案手法の概要

ここでは，提案する手法の概要を述べる．

第 4 章及び第 3 章の結果を元に，以下の規約を提案する．

- ネスト及びユニバーサルセレクタの禁止
- クラスセレクタの使用
- コンポーネント及びモジュールのファイル分割
- 分割したファイルのコンポーネントごとの適用
- コンポーネント間のスタイルの依存関係の禁止 (ブラウザのデフォルト CSS を打ち消すリセット CSS を除く)

ネスト及びユニバーサルセレクタの禁止，クラスセレクタの使用は，CSS の処理におけるブラウザ側の負荷を最大限減らすこと，またセレクタ種類の混乱により詳細度の問題が生じコードが管理困難になることを防ぐためである．

また，コンポーネント及びモジュールのファイル分割は及び分割したファイルのコンポーネントごとの適用は第 3 章の検証実験によるものである．

コンポーネント間のスタイルの依存関係の禁止は，CSS ファイルをコンポーネントの出現順に定義することによってファイル適用の順番が実装時と異なる可能性に対処するものである．あるコンポーネントのスタイルがそのコンポーネント内で完結しており，別のコンポーネントに影響を及ぼすことがなければ，ファイルの順番が前後することによりカスケーディングが狂い予期せぬレンダリング結果をもたらすことがない．ただし，全てのコンポーネントに対して確実に先に宣言されるリセット CSS についてはこの依存関係が混乱する可能性が低いため例外とする．

5.2　CSS 設計手法への適用

本研究において提案手法を検証するにあたって，第 4 章で述べた通り提案手法を既存の CSS 設計思想の 1 つである FLOCSS に適用する．本節ではその具体的な方法を述べる．

5.2.1 文法規則

今回適用する FLOCSS では ID セレクタの利用及びネストを一部許容しているが，本手法の適用によりどちらも使用不可とする．ただし，同一コンポーネント内での隣接セレクタの利用や擬似要素の適用については例外とする．

5.2.2 ファイル適用

本項ではコンポーネントをいかにファイルに分割し，それをコード中に適用するかを述べる．今回適用する FLOCSS におけるコンポーネント分割は以下の通りである．

FLOCSS におけるコンポーネント分割

```
1. Foundation - reset/normalize/base...
2. Layout - header/main/sidebar/footer...
3. Object
    3-1. Component - grid/button/form/media...
    3-2. Project - articles/ranking/promo...
    3-3. Utility - clearfix/display/margin...
```

以上において，Foundation はサイト全体に適用されるべきコンポーネント，Layout は全てのページで使われる大きな単位でのコンポーネント，Object 以下はより小さな単位のコンポーネントを指す．本手法の適応においては，この内 Foundation のみを<head>内に適用し，その他を各コンポーネントが最初に使用される直前に適用するものとする．具体例として FLOCSS ドキュメントにおいて挙げられているファイル管理例に適用する例を示す．FLOCSS ドキュメント上のファイル管理例は以下の通りである．

モデルサイトのコンポーネント構成

```
css
├── foundation
│   ├── _base.scss
│   └── _reset.scss
├── layout
│   ├── _header.scss
│   ├── _footer.scss
│   ├── _main.scss
│   └── _sidebar.scss
└── object
    ├── component
    │   ├── _button.scss
    │   ├── _dialog.scss
    │   ├── _grid.scss
    │   └── _media.scss
    ├── project
    │   ├── _articles.scss
    │   ├── _comments.scss
    │   ├── _gallery.scss
    │   └── _profile.scss
    └── utility
        ├── _align.scss
        ├── _clearfix.scss
        ├── _margin.scss
        ├── _position.scss
        ├── _size.scss
        └── _text.scss
```

この例ではプリプロセッサである SCSS の利用が前提とされているが，本研究においては記載されている scss ファイルをそのまま css に変換するものとする．HTML 上では以下のように適用する．

```
<!DOCTYPE html>
<html>
<head>
        <meta charset="UTF-8">
        <title>コーディング例</title>
        <link type="text/css" rel="stylesheet"
                        href="css/Foundation/reset.css" media="all">
        <link type="text/css" rel="stylesheet"
```

```
                        href="css/Foundation/base.css" media="all">
</head>
<body>

<link type="text/css" rel="stylesheet"
                        href="css/Layout/header.css" media="all">
<section class="header">
        <h1ヘッダー></h1>
</section>

<link type="text/css" rel="stylesheet"
                        href="css/Object/utility/text.css" media="all">
<link type="text/css" rel="stylesheet"
                        href="css/Layout/main.css" media="all">
<section class="main">
        <p class="textコンテンツ"></p>
</section>

:
:
:
```

以上のように，提案手法適用後の HTML コードでは<body>内に複数の<link>タグが使用され，CSS ファイルがその都度呼び出される形になる[1]．

5.3 モデルサイトの作成

本節では，提案する手法を元に実装したモデルサイトについて述べる．

5.3.1 デザイン

実験用に実装したサイトのデザインは，企業や学校の公式サイトのデザインを想定し作成した．実際にブラウザでレンダリングした結果のスクリーンショットを図 5.1 に示す．このデザインをベースに構成の異なる合計 5 つのページを作成した．残り 4 つの外観については付録にてスクリーンショットを添付する．

ページはそれぞれ実運用サイトを想定し index，news，article，image，about と名付けを行なっている．

5.3.2 コンポーネント設計

上述したデザインを元に，以下のようにコンポーネント及びを定義した．

[1]実際に CSS がロードされるタイミング，またその時点でのレンダリングに関する挙動はブラウザによって異なる

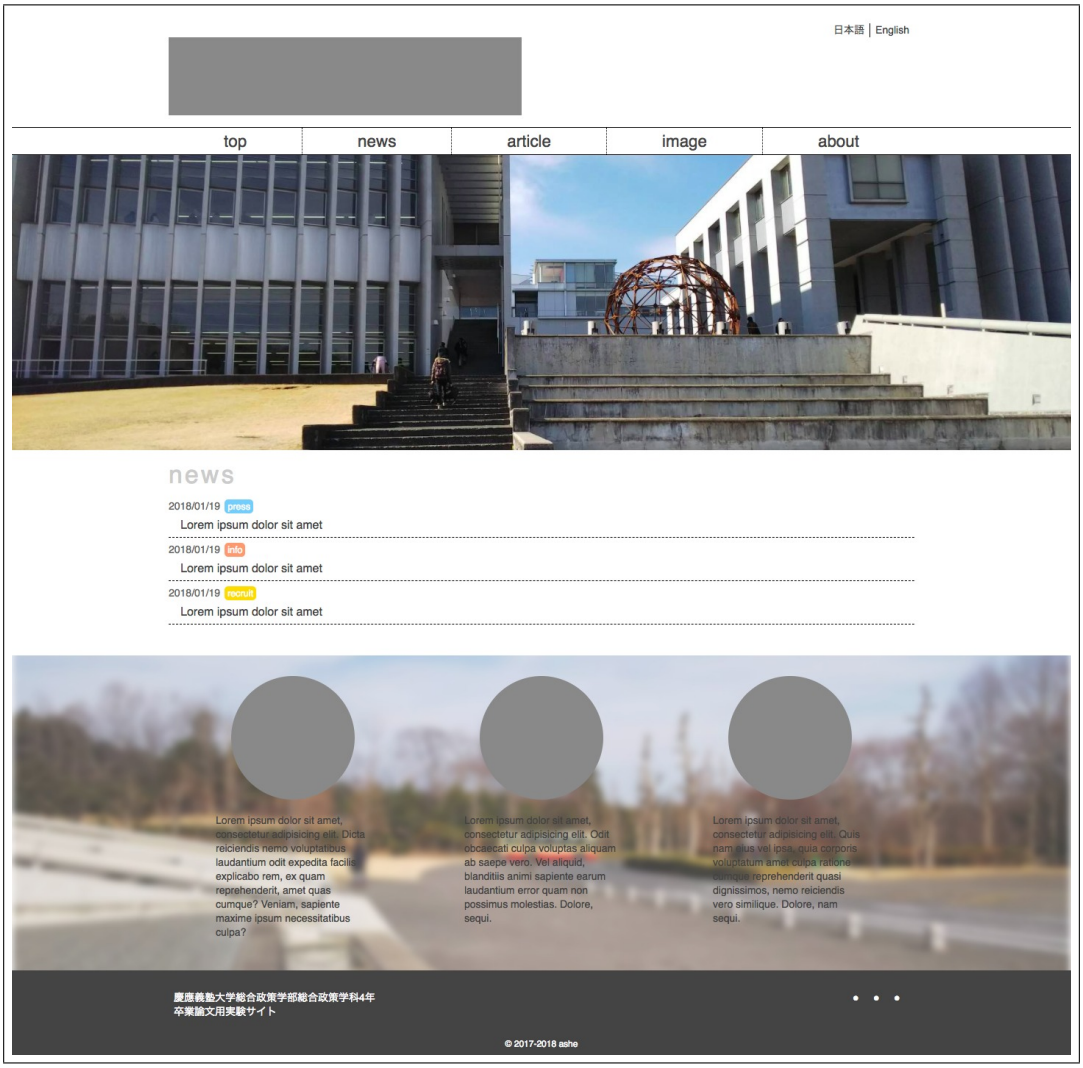

図 5.1: サイトデザイン 1

モデルサイトのコンポーネント構成

```
.
├── Foundation
│   └── foundation.css
├── Layout
│   ├── footer.css
│   ├── header.css
│   ├── main.css
│   ├── menu.css
│   └── sidebar.css
└── Object
    ├── Component
    │   ├── head-article.css
    │   └── head-top.css
    ├── Project
    │   ├── about.css
    │   ├── article.css
    │   ├── imglist.css
    │   ├── cover.css
    │   ├── introduction.css
    │   └── news.css
    └── Utility
        ├── margin.css
        └── centering.css
```

これを適用するモデルサイトの HTML ファイル内で呼び出される CSS ファイルの数は，それぞれ index，news，article が 8，image，about が 7 である．

5.4 本章のまとめ

本章では，CSS の体系化を念頭におきつつパフォーマンス向上をはかる新たな独自の設計手法を提案した．その上で，設計手法を第 4 章で選出した既存の CSS 設計思想に適用し，モデルサイトの制作を行った．

第6章　評価

本章では，提案した新しいCSS設計手法のパフォーマンスを測定し，既存のCSS設計手法との比較実験を行う．

6.1　実験概要

本節では評価実験の概要について述べる．

本実験では，第5章において作成したモデルサイトについて，サイトデザインは同一だが本手法を適用せずCSSファイルを結合した状態で適用したものを新たに作成し，第3章での検証実験と同一の環境でパフォーマンスを測定する．

計測に使用する環境は表3.1に示した通りであり，計測の指標としてロード時間及びSpeed Indexを採用する．計測はそれぞれ5回ずつ行い，数値の総合を比較評価する．

6.2　実験結果

本節では実験の結果判明した数値をまとめる．

6.2.1　ロード時間の比較

提案手法を適用したものとそうでないものにおけるロード時間の計測結果を以下の図6.1-図6.5に示す．グラフは計測に用いたHTMLファイル毎の結果であり，縦軸に算出したSpeed Index，横軸に実験を行った回数を置く．計測時間の単位は全てmsである．

グラフにより，全てのページを通して，ファイルの結合を行なっている場合が結合を行わない提案手法に対して若干の優位を示す，あるいはほぼ変わらないことがわかる．

以上の結果から，モデルサイトへの適用においても先行研究及び検証実験で判明していた通り，ロード時間の改善においてファイル分割よりも結合を行なった方が優れた結果を示すことがわかった．ただしその差は特に憂慮すべきと言えるほど顕著ではなく誤差程度と考えられる範囲にあるため，実験に用いたモデルサイトの規模であればCSSファイルの分割はロード時間に影響を与えないと言える．

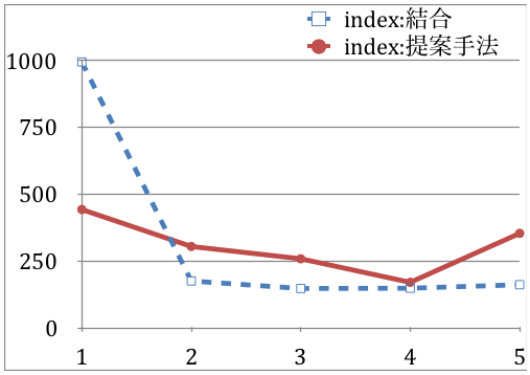

図 6.1: ロード時間計測結果: index

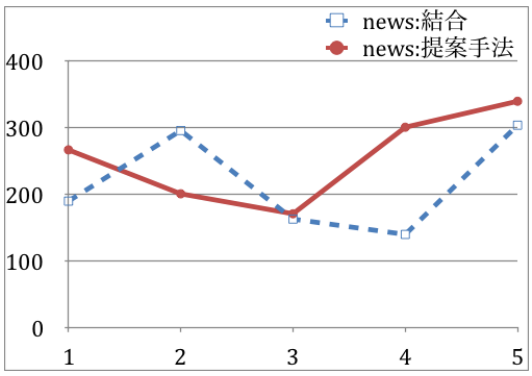

図 6.2: ロード時間計測結果: news

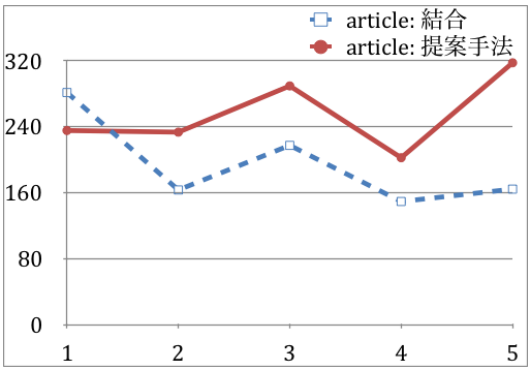

図 6.3: ロード時間計測結果: article

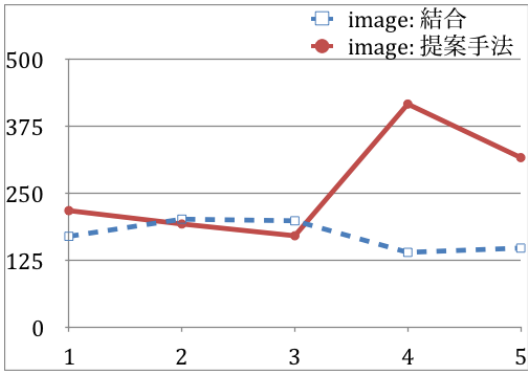

図 6.4: ロード時間計測結果: image

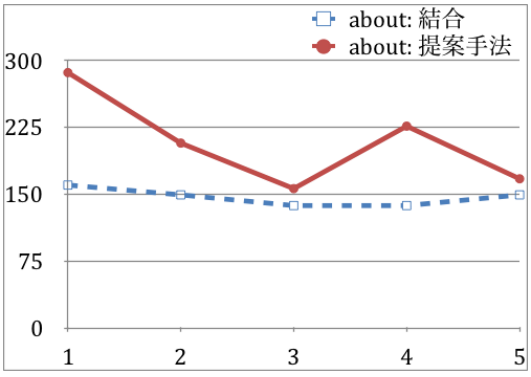

図 6.5: ロード時間計測結果: about

6.2.2 Speed Index の比較

提案手法を適用したものとそうでないものにおけるロード時間の計測結果を以下の図 6.6-図 6.10 に示す．グラフは計測に用いた HTML ファイル毎の結果であり，縦軸に算出した Speed Index，横軸に実験を行った回数を置く．

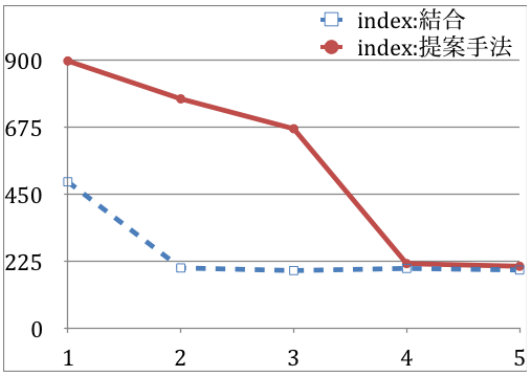

図 6.6: Speed Index 計測結果: index

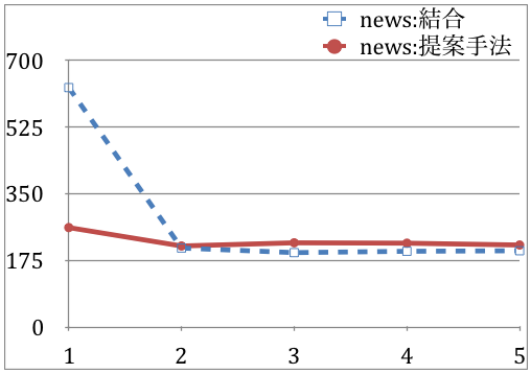

図 6.7: Speed Index 計測結果: news

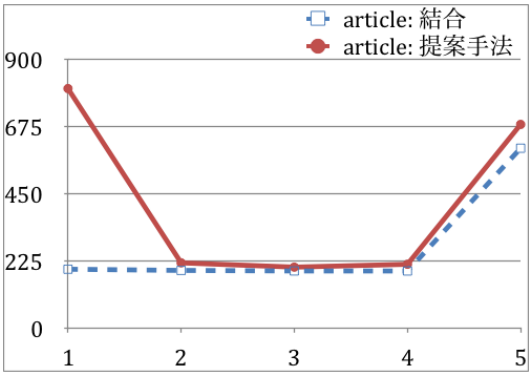

図 6.8: Speed Index 計測結果: article

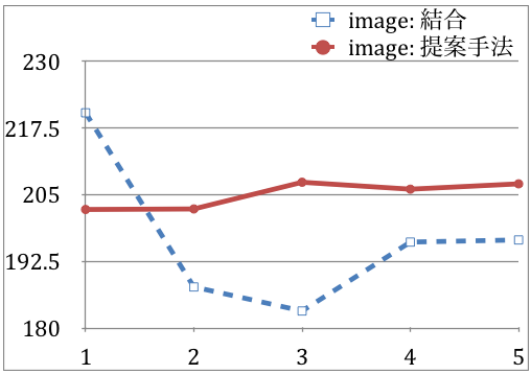

図 6.9: Speed Index 計測結果: image

グラフを見ると，index，article，image のページにおいてはファイルの結合を行なった場合が提案手法を適用したものに比べ同等，あるいは若干の優位を示していることがわかる．news，about については逆に提案手法を適用したものが若干の優位を示している．この 2 グループにコーディング上の特徴の違いはなく，この差異は計測における誤差であると考えられる．

以上の結果によって，ファイルを結合した場合，結合せず提案手法を適用した場合の双方においてレンダリングパフォーマンスに大きな差異は生じないということがわかった．

6.3 本章のまとめ

本章では提案した手法と既存手法による実装をブラウザでのアクセス時間を計測することにより比較した．比較の結果提案手法はパフォーマンスにおいて顕著な改善を行うものでは

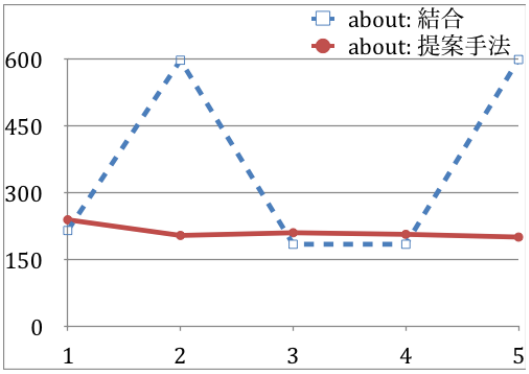

図 6.10: Speed Index 計測結果: about

ないが，同時に既存の手法と比較して大きく劣るものではないということがわかった．このことから，コーディング規約や設計方針を定めるに当たって，キャッシュの効率やファイル管理の視点に立ち CSS ファイルの分割を導入することは Web サイトのパフォーマンスに影響を与えないと考えられる．つまり，提案手法はパフォーマンスの解決策ではないが，実際のコーディングにおいて有用性のあるものであるといえる．

第7章　結論

本章では，本研究のまとめと，今後の課題及び展望について述べる．

7.1　本研究のまとめ

本節で本研究のまとめを示す．

本研究では CSS 開発が大規模になるにつれパフォーマンスへの影響が高まることを懸念し，HTTP/2 以降のモダンな Web 開発におけるパフォーマンス最適化手法について探った．まず CSS コーディングにおける既存のパフォーマンス最適化手法についての調査及び検証，また既存の CSS 設計思想の検討を行った．またそれを踏まえ新たなパフォーマンス最適化手法とその CSS 設計手法における適用を行い，モデルサイトを実装してパフォーマンスの計測実験を行った．

実験の結果，HTTP/2 が普及するであろう今後の Web 開発においてもファイルの結合に始まる既存のパフォーマンス最適化手法は有効であることがわかった．しかし，ファイル分割を行った際のパフォーマンスが大きく劣っているということもないため，CSS コードが頻繁に変更されキャッシュの効率が重要な場合などに CSS ファイルの分割を検討することはパフォーマンスの観点から有効であると考えられる．

7.2　今後の課題と展望

今回は静的で比較的小規模なコードで実験をしたが，動的な Web アプリケーションなどより大規模な実運用サイトで継続的に測定調査を行うことにより，さらなる優位性の検証ができるのではないかと考える．

また，実験環境について，単一のブラウザ及び単一のネットワーク環境について計測を行ったが，より多くのブラウザやより多くの通信環境を用いて実験することによって，提案手法が有効になる状況やパフォーマンス上推奨されない状況などがよりはっきりするのではないかと考える．特にスマートフォンの普及により最重要視されつつあるモバイル環境での適用について本研究の計測に含まれていないため，より実用的な指標の提案のためにそうした異なる環境での計測が必要だと考える．

謝辞

本論文の執筆にあたりご指導頂いた，慶應義塾大学環境情報学部教授 村井純博士，同学部客員教授 徳田英幸博士，同学部教授 中村修博士，同学部教授 楠本博之博士，同学部准教授 高汐一紀博士，同学部准教授 Rodney D.Van Meter III 博士，同学部准教授 植原啓介博士，同学部教授 三次仁博士，同学部准教授 中澤仁博士，同学部教授 武田圭史博士，同大学環境情報学部講師 斉藤賢爾博士，同大学政策・メディア研究科特任准教授 鈴木茂哉博士，同研究科 特任准教授 佐藤雅明博士，同研究科 特任講師 Achmad Husni Thamrin 博士，同研究科特任助教 空閑洋平博士，同研究科 中島博敬氏，永山翔太博士に感謝いたします．

研究について日頃からご指導頂きました，鈴木茂哉博士に重ねて感謝致します．お忙しい合間にお時間を頂戴し，多岐に渡るご指導を賜りました．その多大なるご厚情を賜ることにより研究を続けられたことを，心より感謝申し上げます．

Kumo 研究グループの皆様，徳田・村井・楠本・中村・高汐・バンミーター・植原・ 三次・中澤・武田 合同研究プロジェクトの皆様に感謝致します．Kumo 研究グループの同期，尾崎周也氏，増田雄一氏，桑原誠尚氏，Rickey Rowland 氏に重ねて感謝いたします．未熟な私を常に同期として迎え入れてくれた彼らと研究生活を共にできたことは幸運でした．感謝いたします．

最後に，私の研究生活及び学生生活を支えていただいた慶應義塾大学のみなさま，友人，家族に感謝いたします．

以上を持ちまして，謝辞といたします．

参考文献

[1] http://www.ecma-international.org/publications/standards/Ecma-262-arch.htm. 2018 年 1 月 19 日アクセス.

[2] https://developer.mozilla.org/ja/docs/Web/CSS/CSS3. 2018 年 1 月 19 日アクセス.

[3] webcomponents.org. https://www.webcomponents.org/. 2018 年 1 月 19 日アクセス.

[4] Philip Walton. CSS Architecture. https://philipwalton.com/articles/css-architecture/, 2012.

[5] Ali Mesbah and Shabnam Mirshokraie. Automated analysis of CSS rules to support style maintenance. In *2012 34th International Conference on Software Engineering (ICSE)*, 2012.

[6] Bootstrap. https://getbootstrap.com/. 2018 年 1 月 19 日アクセス.

[7] Hampton Catlin, Natalie Weizenbaum, Chris Eppstein, and numerous contributors. Sass: Syntactically Awesome Style Sheets. http://sass-lang.com/. 2018 年 1 月 19 日アクセス.

[8] Andrey Sitnik. Autoprefixer. https://github.com/postcss/autoprefixer. 2018 年 1 月 19 日アクセス.

[9] PostCSS. https://github.com/postcss/postcss. 2018 年 1 月 19 日アクセス.

[10] Nicholas C. Zakas and Nicole Sullivan. CSS Lint. http://csslint.net/. 2018 年 1 月 19 日アクセス.

[11] David Clark Maxime Thirouin and Richard Hallows. stylelint. https://stylelint.io/, 2015.

[12] Facebook Inc. React - A JavaScript library for building user interfaces. https://reactjs.org/. 2018 年 1 月 19 日アクセス.

[13] vjeux. React: CSS in JS. https://speakerdeck.com/vjeux/react-css-in-js, 2014.

[14] css-modules. https://github.com/css-modules/css-modules. 2018 年 1 月 19 日アクセス.

[15] gulp.js. https://gulpjs.com/. 2018 年 1 月 19 日アクセス.

[16] webpack. https://webpack.js.org/. 2018 年 1 月 19 日アクセス.

[17] IIya Grigorik. ハイパフォーマンスブラウザネットワーキング. オライリー・ジャパン, 2014. 和田祐一郎，株式会社プログラミングシステム社 訳.

[18] Zhiheng Wang. Navigation Timing: W3C Recommendation 17 December 2012. https://www.w3.org/TR/navigation-timing/, 2012.

[19] WebPagetest - Website Performance and Optimization Test. https://www.webpagetest.org/. 2018 年 1 月 19 日アクセス.

[20] WebPagetest Documentation: Speed Index. https://sites.google.com/a/webpagetest.org/docs/using-webpagetest/metrics/speed-index. 2018 年 1 月 19 日アクセス.

[21] Axel Faes Robin Marx, Peter Quax and Wim Lamotte. Concatenation, Embedding and Sharding: Do HTTP/1 Performance Best Practices Make Sense in HTTP/2? In *Proceedings of the 13th International Conference on Web Information Systems and Technologies*, pp. 160–173, 2017.

[22] Nicole Sullivan. Object Oriented CSS. http://oocss.org. 2018 年 1 月 19 日アクセス.

[23] Vladimir Starkov Vsevolod Strukchinsky. Get BEM. http://getbem.com/. 2018 年 1 月 19 日アクセス.

[24] Jonathan Snook. Scarable and modular architecture for css - a flexible guide to developing sites small and large. 2012.

[25] 谷拓樹. flocss - CSS organization methodology. https://github.com/hiloki/flocss. 2018 年 1 月 19 日アクセス.

[26] Harry Roberts. Manage large CSS projects with ITCSS. http://www.creativebloq.com/web-design/manage-large-css-projects-itcss-101517528, 2016.

[27] Nicolas. SUIT CSS - Style tools for UI components. http://suitcss.github.io/. 2018 年 1 月 19 日アクセス.

[28] Adam Silver. Maintainable CSS. https://maintainablecss.com/. 2018 年 1 月 19 日アクセス.

[29] Robert Haritonov. MCSS - Multilayer CSS. http://operatino.github.io/MCSS/ja/. 2018 年 1 月 19 日アクセス.

[30] Ben Schwarz Glen Maddern. AMCSS - Attribute Modules for CSS. https://amcss.github.io/. 2018 年 1 月 19 日アクセス.

[31] Ben Frain. Enduring CSS - A guide to writing style sheets for large scale, rapidly changing, long-lived web projects. http://ecss.io/, 2015.

[32] Ben frain. Enduring CSS: writing style sheets for rapidly changing, long-lived projects. https://benfrain.com/enduring-css-writing-style-sheets-rapidly-changing-long-lived-projects/, 2014.

[33] Yahoo! Inc. Atomic CSS. https://acss.io/, 2015.

[34] Elika J. Etemad / fantasai (Invited Expert) and Tab Atkins Jr. (Google). CSS Cascading and Inheritance Level 3: W3C Candidate Recommendation, 19 May 2016. https://www.w3.org/TR/css-cascade-3/, 2016.

[35] John Polacek. By The Numbers: A Year and Half with Atomic CSS. https://medium.com/@johnpolacek/by-the-numbers-a-year-and-half-with-atomic-css-39d75b1263b4, 2017.

[36] 谷拓樹. Web 制作者のための CSS 設計の教科書: モダン Web 開発に欠かせない「修正しやすい CSS」の設計手法. 株式会社インプレス, 2014.

[37] 森下雅章. [詳解] モダン CSS: 記法，スタイルの管理，PostCSS. 技術評論社, 2016.

[38] Speeder: A framework for automated Web Page Optimization testing. https://speeder.edm.uhasselt.be/. 2018 年 1 月 19 日アクセス.

[39] Yiping Chen Hugues de Saxce , Iuniana Oprescu. Is HTTP/2 Really Faster Than HTTP/1.1? In *2015 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pp. 293–299, 2015.

[40] Hypertext Transfer Protocol Version 2 (HTTP/2). http://www.rfc-editor.org/rfc/rfc7540.txt, 2015.

[41] Paul Irish Pierre-Marie Dartus. WebPagetest - Website Performance and Optimization Test. https://github.com/paulirish/speedline. 2018 年 1 月 19 日アクセス.

[42] きたけー. WebPagetest を使わずに Speed Index を算出する - kitak blog. http://kitak.hatenablog.jp/entry/2016/12/26/201925, 2016.

[43] 竹洞陽一郎. Speed Index とは何か? — ブログ — 株式会社 Spelldata. https://spelldata.co.jp/blog/blog-2017-05-05.html, 2017.

[44] Inessa Brown. Methods to Organize CSS. https://css-tricks.com/methods-organize-css/, 2017.

[45] Chris Coyier. Efficiently Rendering CSS. https://css-tricks.com/efficiently-rendering-css/, 2010.

[46] Jake Archibald. The future of loading CSS. https://jakearchibald.com/2016/link-in-body/, 2016.

付 録A

A.1　実装サイトのデザイン外観

実装したWebページ全てのスクリーンショットを以下に示す.

図 A.1: サイトデザイン 2

図 A.2: サイトデザイン 3

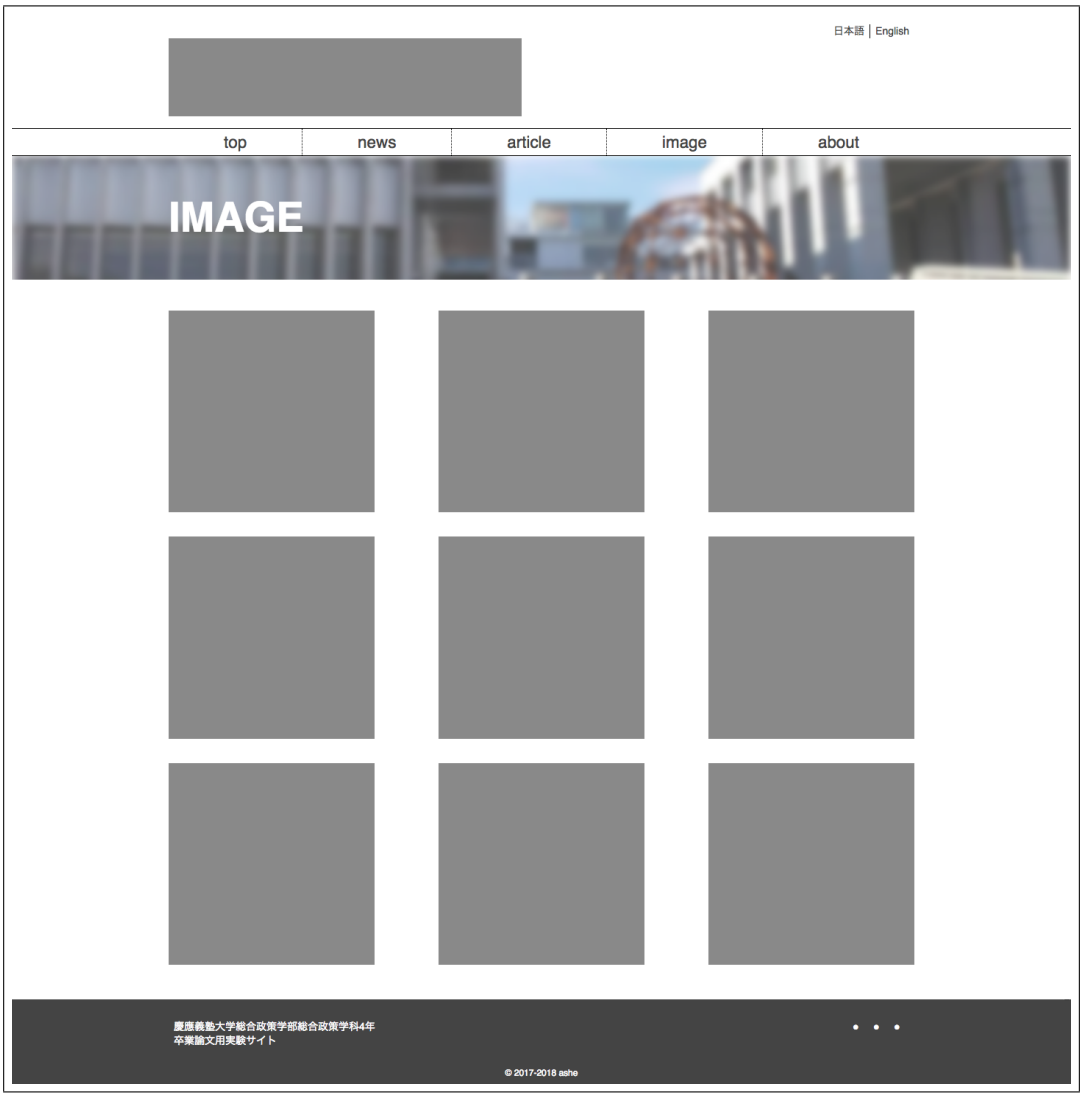

図 A.3: サイトデザイン 4

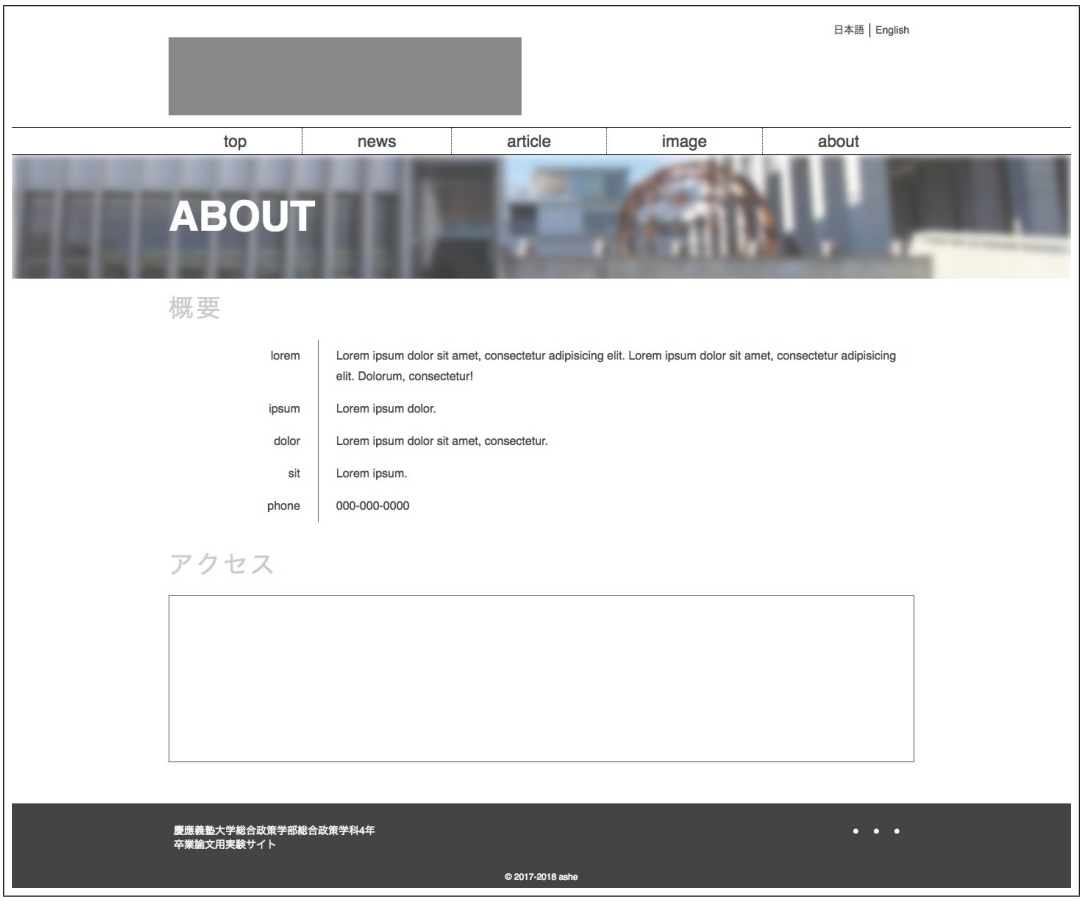

図 A.4: サイトデザイン 5