Webデザイン実習3B

2019/7/15

Kazuma Sekiguchi
class@cieds.jp

課題

- 今までにやったJSの技法を活かしてサイトを作成
  - インタラクションのあるサイトを作成
  - インタクションは効果的に利用すること
  - jQueryの使用OK
  - JSの機能は2つ以上あること
- テーマ
  - 夏に楽しめるもの、こと、場所の紹介サイト
  - 自分で作成したキャラクターなどは許可
  - トップページのみ作成すればOK

最終課題

- 画像などは自分で描いたもの、撮ったもの、商用利用可のもののみ利用可
  キャラクター関係は不可（特にネズミー関係）（自分で作ったキャラクターならOK）
- デザイン（ai、psd、sketch、XDなど編集可能なもの）とそれのPDF化したもの、HTML,CSS,js,画像群を提出
- デザインカンプはアウトライン化、ラスタライズ化しないこと
  - 保存バージョンは最新のものでOK

提出日など

- Topページはindex.htmlとすること
  - 直下に配置すること
  - 日本語ファイル名は不可
- 画像はimagesフォルダにまとめて配置すること
- カンプ（aiとかpsdとか）は直下に配置すること
- カンプと最終的に完成したものが多少異なるのはOK
  - 全然違うのはダメだけど

提出日など

- 2019年7月29日までに提出
  - 直接関口に提出してください（メール提出は不可）
  - その前に提出するのはOK
  - 早めに出しておくことをお奨めします

見るところとか

- きちんとタグが使い分けできているか
- CSSが書けているかどうか。妥当なCSSになっているかどうか
- 表示したときに崩れていないか、画像が表示されるか
- ファイル名、フォルダー名が英数字だけで構成されているかどうか
- デザインがWeb的かどうか
- JSがきちんと作成されているか、きちんと動作するか
- scriptタグなどの指定に間違いが無いか。jqueryを複数読み込んだりしていないか
- jqueryの初期化指定が正しくできているか
- jsを効果的に使用できているか

夏休みの課題

- ウェブサイトを30サイト閲覧し、そのウェブサイトに関してレポート
- A4用紙に余白を四方それぞれ20mm取り、ウェブサイトのスクリーンショット（メインなページ）を貼り付け
  - Word等で作成するのが楽
- 下部に「閲覧日」、「URL」、「タイトル」、「良いと思った点や改善点など（3行程度）」を記入
- 1ファイルのPDF形式にして提出（ファイル名を学籍番号＋名前にすること）

WebSitesのまとめサイト

- WebDesignClip（日本のサイト）
  - http://webdesignclip.com/
- WebDesignFile（海外サイト）
  - http://www.webdesignfile.com/
- I/O 3000 | Webデザインギャラリー
  - http://io3000.com/
- Webデザインリンク集
  - http://bm.straightline.jp/

提出日など

- 夏休み明けの最初の授業に提出

CSSメタ言語

- CSSを効率的に記述するための記述方法
- CSSを効率的に書くためのものなので、新しい言語という訳では無い

- いくつかの種類が存在
  - Sass ，Scss ，Less辺りが有名で良く使われる

今はまずこれ

CSSの問題

- CSSの仕様上、継承を多く利用してスタイルを適用する
  - 継承の記述が煩雑
  - 継承がCSSファイルだけを見ているだけでは把握しづらい（一覧性に乏しい）
- クラスの動的な適用、排除によるスタイル定義
  - 状態にあわせた（DOM構造の変化に耐える）クラスを多数作成する必要性の向上
  - 多重クラスの利用がしやすくなったことで、ますますCSSに記述する量が増大

高速化の壁

- CSSは複数ファイルに分散して記述が可能
  - CSSファイルに限らず、ファイルの数を減らした方が表示は速くなる
  - できればCSSファイルは1つにしたい
    - 1つにしたらCSSの行数が膨大になり、管理、メンテナンスが面倒になる
  - 最終出力で結合して1つにできれば、メンテナンスが容易なまま高速化が可能
  - 圧縮もついでにしてくれたら良い

SCSS

- CSSにはスコープ概念が存在しない
  - 間違えて他のスタイルを上書きすることもある
  - 命名規則（BEMなど）がさまざま登場してきた理由でもある
- CSSファイルを分離して作成し、最後に必要なCSSを結合していくのが読み出しにも効率的
  - モジュール単位でCSSスタイルを作成
  - いくつかのモジュールでSCSSファイルを作成して結合
  - 最終的に1つのCSSファイルのみを生成する

```
@import “_base.scss;”
```

CSSを楽に記述

- 入れ子を簡単に
- 変数を利用可能に
- 同じ表現の使い回しを楽に
- 複数ファイルを結合したい
- できれば圧縮もしたい
- ベンダープレフィックスとか面倒

SassとかLessとか

- Sass（サス）、Less（レス）はともにCSSを簡単に書くためのCSSメタ言語
  - Sassのバージョンアップ版がScss（サス）
  - 通常サスと呼んだときはSCSSを指すことが多い
- バージョンによって機能に違いがある点は注意
  - 変換ソフトによっては、最新版が使えるとは限らない
- 今はSCSSで記述するのが一般的

Sassとか

- CSSとは違う記述を行う
  - SassはCSSと同じ記述方法も可能（意味無いけど）
  - CSSとして利用する場合は変換が必要
    - 変換をコンパイルと呼ぶ（実際のコンパイルはC言語などで記述したプログラムを実行できるように変換すること。単なる変換とはちょっと違う）
  - 変換せずにJSを通して利用する手もあるが遅くなるためお薦めしない
  - 変換は通常専用のソフトを用いる
    - 今はNode-sassを利用するのが流行り
    - Ruby-scssは開発が止まった

SCSS

- 現状のさまざまなCSSに対する一つの解として使われているのがSass/SCSS
- SCSSなどを利用することで、ある程度CSSの抱える問題を克服することが可能
  - 全ての問題は克服できない→CSSのそもそもの設計的問題
- Autoprefixerなどを併用することで、ベンダープレフィックスを自動的に付与することも可能

TaskRunner

- Web制作で決まった処理を自動化して行うための仕組み
- 拡張モジュールを入れることでさまざまな処理を自動化することが可能
  - ScssからCSSへの変換
  - CSSの圧縮、JSの圧縮
- gulp.jsが有名だが、npm scriptでも対応可能
  - どちらもNode.js上で動作するアプリ
  - 動作させるには行いたいことを記述したJSONファイルが必要、動作はCUI画面から行う（通称黒い画面）
  - 昔はGruntというのもあった

流れ

Sass,Lessで記述し保存

コンパイラ（トランスパイラ）で変換

CSSを生成

生成されたCSSを利用

変換

- オンラインでも変換可能
  - http://c2c.briangonzalez.org/など
- インストールして動作するタイプも種類豊富
  - Prepros
  - Koala
  - CodeKit(Mac)
  - Scout
- DreamweaverでもCC2017以降は変換可能

- 変換ソフトによっては、Sassなどを保存したら自動的に変換して、ブラウザーをリロードしてくれるものも
  - 表示確認が楽

出力

- Scssで記述した場合、最終的に利用するのはCSS
  - ScssからCSSへトランスパイルするときに形式を選択可能
  - nested，expanded，compact，compressed
  - 通常はCompressedを選択（コメントや改行、余分な文字など全て排除される）
- Scssの場合、CSSの変更、修正はSCSSファイルで行う
  - CSSを直接編集しないこと（怒られます）
  - CSSが読みづらくても何ら問題ない（Scssが読めれば問題ない）

Scssの記述

Scss

```
$bgcolor:#EEE;
$mainSize:980px;
body{
        background-color:$bgcolor;
        h1{
                text-align:center;
        }
}
.wrapper{
        width:$mainSize;
        margin:0 auto;
        main{
                width:($mainSize / 2);
        }
}
```

CSS

```
body {
        background-color: #eeeeee;
}
body h1 {
        text-align: center;
}

.wrapper {
        width: 980px;
        margin: 0 auto;
}
.wrapper main {
        width: 490px;
}
```

Scss

- {の使い方次第で入れ子を表現可能
  - {を閉じないでそのままタグを記述すると親セクレターを付与したセレクターになる
- 変数を利用可能
  - 「$変数名:変数の値」で記述しておけば、どこからでも利用可能
  - 上部にまとめて記述しておけば、メンテナンスしやすい
  - 変数の値を変更すれば、一気に変更が可能
- 計算式が利用可能
  - 四則演算が可能
  - 引き算と割り算は（）でくくること
  - 単位は自動で付与される（割られる対象についている単位が使われる）

SCSSの注意点

- 入れ子が簡単に掛けるため、入れ子が複雑になりやすい
- Sass自体のバージョンがあるため、新しい機能だと利用できないこともある
  - Sass自体をバージョンアップすればOK
- SCSSであまり複雑な関数などを作ると直感的に把握しづらくなるため、分かりやすくなるように心掛ける
  - 関数化して、別ファイルにしておく（隠蔽しておく）

複数のファイルに分ける

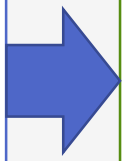
- SCSSでは複数のファイルに分け、読み込むことが可能
- アンダーバーをファイル名先頭に付与することで、パーシャルファイルとして利用できる
  - SCSSを保存してもCSSとしてトランスパイルされない
  - 単独で使用しないファイル（読み込まれて使用する）に使う
  - _base.scss
- 用途にあわせてファイルを分割することで、管理がしやすくなる
  - 要らないCSSファイルを読み込まない、なども可能
- 読み込む際は、@importの後にアンダーバーと拡張子を外して指定する

```
@import “base”
body{
p{
```

Scss

- @import
  - 指定したCSSファイルまたはScssファイルを読み込んでくる
  - 結合が可能
  - CSSの@importとは異なり、記述した位置にファイルが読み込まれるので注意
  - ScssではCSSの@importは多分使えないはず

```
@import base.css
@import layout.scss
```



```
body{/*base.css*/
        margin:0;
        padding:0;
}
.box{/*layout.scss*/
        width:600px;
}
```

Scss

- プロパティも入れ子が可能
  - background-colorなどのハイフン区切りのプロパティはbackgroundの子としてcolorなどを指定可能

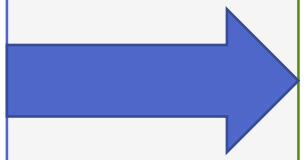
```
#main{
background:{
        color:#EEE;
        image:url(“back.jpg”);
        }
}
```

```
#main {
        background-color: #EEE;
        background-image: url(“back.jpg”);
}
```

Scss

- &セレクター
  - 親セレクターを指したい場合に利用

```
a{
        color:#FFF;
        &:hover{
                color:#F00;
        }
}
```



```
a {
        color: #FFF;
}
a:hover {
        color: #F00;
}
```

Scss

- mixin
  - mixinとは表現の集合体
  - 何度も使うような表現をまとめて記述しておける
    - 修正する場合もmixinを修正すれば全部修正される

```
@mixin normalBox{
        background-color:#FFF;
        font-size:12px;
        box-shadow:1px 1px 1px #000;
        border-radius:3px;
}
#main{
        @include normalBox;
        width:980px;
}
```

```
#main {
        background-color: #FFF;
        font-size: 12px;
        box-shadow: 1px 1px 1px black;
        border-radius: 3px;
        width: 980px;
}
```

SCSS制御文的なもの

- @mixin ，@include

```
@mixin base{
background-color:#ccc;
width:800px;
}
main{
        @include base;
        margin:0 auto;
}
```

SCSS制御文的なもの

- @mixin ，@include
- 外部から値を与えることが可能
- 初期値を設定すれば、@includeで値を与えなくても初期値が使われる

```
@mixin base($color:#F00){
background-color:#ccc;
color:$color;
}
@mixin baseBox($size:900px){
width:$size;
margin:0 auto;
}
body{
        @include base(#999);
        @include baseBox(1160px);
}
```

似たようなもので@extend

- @mixinと似ているが使い道が違う
  - 既存のclass表現などを変更するときに使用する
  - @mixinは既存のclassに表現を追加するときに使用する
  - @mixinはそもそも読み込まれるという前提がある（@extendは普通のクラスなどを読み込むので、読み込まれる前提がない）

```
. extendTableHead {
    @extend .tableHead
    font-size: 10px;
}
```

別に指定したclass定義

それの文字サイズを変更して、新しいクラスを定義

```
@mixin tablehead{
  background-color:#f90;
}
.tableheadline{
  @include tablehead;
  font-size:14px;
}
```

Scss

- 関数を利用可能
  - 使わなくても十分機能的に高機能なため、無理に使うことも無い
  - 変化に強いCSSにはなるので、フレームを作成する際には便利かも

```
$totalWidth: 940px;
$columnCount: 10;

@function getColumnWidth($width, $count) {
  // $columnWidthを計算
  $padding: 10px;
  $columnWidth: floor(($width - ($padding * ($count - 1))) / $count);
  @debug $columnWidth;
  @return $columnWidth;
}

.grid {
  float: left;
  width: getColumnWidth($totalWidth, $columnCount);
}
```

SCSSの関数利用

- SCSSではさまざまな関数を利用することが可能
  - ビルドイン関数
  - ユーザー定義関数
  - JavaScriptに近いがより簡単に記述が可能
- トランスパイラ実行時に実行されてCSSとして展開される
  - JSなどと異なりCSS時に動的に値を変えるような動きをするわけではない
  - CSSのcalc()を使えばある程度の計算はCSS上で可能になっている

SCSS関数（ビルトイン）

- 色関連の関数が豊富
  - rgb関数:rgb(255,200,100)→HEX指定に変換する
  - mix関数:mix(#000,#FFF)→混ぜた色ができる
  - hsl関数：hsl(127,86%,20%)→HEX指定に変換する
  - lighten関数：lighten(#663355,15%)→15%明るくしたHEX値を返す
  - darken関数：darken(#446373,10%) →10%暗くしたHEX値を返す
  - complement(#552638)→補色の色のHEX値を返す

SCSS関数（ビルトイン）

- round関数：round(4.5)→数値を四捨五入
- ceil関数：ceil(7.21)→数値を切り上げ
- floor関数：floor(4.752)→数値を切り捨て
- min関数（20px,53px,10px,$base)→最小の値を返す
- max関数（20px,53px,10px,$base)→最大の値を返す

SCSS制御文的なもの

- @function

```
$mainSize:800px;
@function wrapperSize($size){
	@return $size / 2;
}
main{
	width:wrapperSize($mainSize);
	margin:0 auto;
}
```

SCSS制御文的なもの

- @functionを利用することで、独自の関数を作成することが可能
  - 利用する時は関数名を指定すれば利用可能
  - 関数内では@returnで必ず返り値を指定する
  - 通常戻り値はプロパティの値として利用することが多い
    - セレクターとして利用もできる
  - 通常は、()で引数を受け取るようにして、関数内部の値を変更させる
    - 変更ができない関数は利用する意味が無い

@mixinと@functionの使い分け

- @mixinはCSSの固まりをそのまま持ってくることが可能
- @functionはそのときに値を与えて単一の値だけを計算させることが可能
  - 何らかのCSSプロパティの値として利用する
  - 他のビルトイン関数と組み合わせて利用する
- CSS自体のスタイルをそのまま再利用するのであれば@mixinを利用する
  - @mixin内にも@functionは利用できるため組み合わせることももちろん可能

ループ

- 繰り返して処理をする仕組み
- #{変数}みたいに記述する（インターポレーション）とセレクター側で変数を利用可能

```
@for $i from 1 through 5{      //5回繰り返す。
        .item_#{$i}{
                margin:$i*10px;//$iは1回ずつ増えていく
        }
}
```

配列

- 配列を作成可能
  - 単一の変数に複数の値（文字列）を入れることができる仕組み
  - 通常は@eachと組み合わせて利用する

```
$items = ‘apple’,’orange’,’berry’;
@function urlimgbase($image){
        @return url(“../images/”+$image);
}
@each $item in $items{
.#{$item}{
        background-image:urlimgbase($item+’.jpg);
}
```

分岐

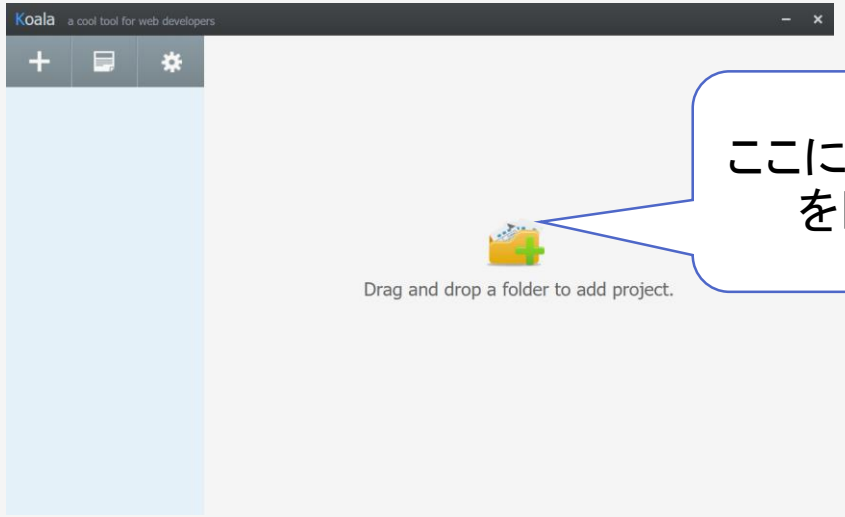
- @if,@elseを利用した分岐が可能
  - 条件に合わせて利用する関数を分けたり、書き出すCSSを変えることが可能

```
for $i from 1 through 10{
.box_#{$i}{
    width:100px;
    @if($i == 10){
        margin-bottom:10px;
    }
    @else{
        margin-bottom:5px;
    }
}
```

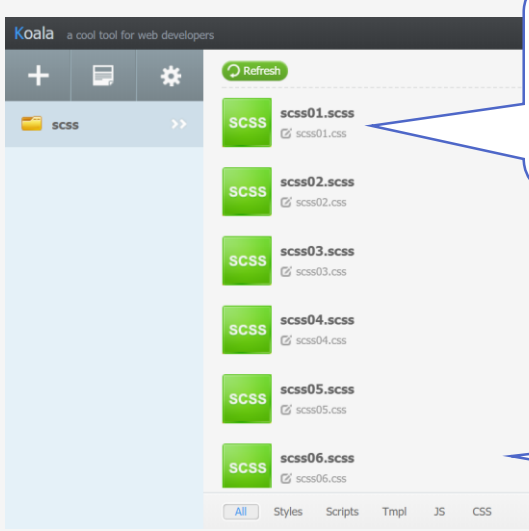
Scss

- Scssの保存時、拡張子は、scssとすること（sassではないので注意）
- Scssのコメントは「/* */」のほか、「//」タイプの1行コメント文も可能
  - //タイプの場合は、/* */に変換される（コメントを削除する変換もある）
- 通常Scssファイルの保存先フォルダとCSSファイル保存先フォルダは分けるので、パスに注意

Koala

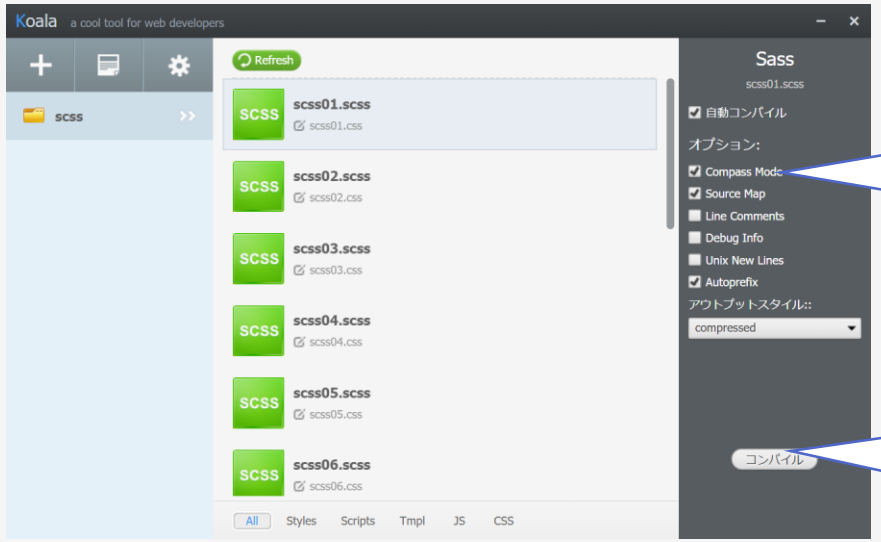


ここにフォルダーをドロップ



対象となるファイルが検出される

以降はファイルを編集して保存すれば自動的にコンパイルされる



変換時の設定を行う「環境設定」で設定したものが反映されている

コンパイルを押せば、変換される

Node.js

- 本来は大量の接続に耐えるWebサーバー
  - C10K問題（同時10,000接続に耐える）への解として出てきた
  - 言語としてJavaScriptを用いたサーバーサイドスクリプト環境を提供している
  - 現在もサーバーサイドJavaScriptとして出てくることもある

  - 中身的にはChromeで使われているv8エンジンを元にしている

Node.js

- フロントの世界では。。。
  - JavaScript実行環境とされる
  - JavaScriptはブラウザーから実行するのが一般的だが、それをブラウザー以外で他の機能を付け加えた上で利用可能とした
    - CommonJS（ファイルの書き込みや読込など）
    - 厳密にはNodeはCommonJSから外れる、、らしい
- JavaScriptでアプリを作成できることからフロントの世界で異様に使われている
  - アプリと言ってもGUIは無いため、全てコマンドで操作する

NPM(Node Package Manager)

- Node上で動作するアプリは大量
  - それぞれが依存関係を有していることがある
  - 依存＝他のファイルが無いと動作しない関係
    - リンクに似ているが、リンクは無くても動作するが、依存関係の場合は無いと動作しない
  - 依存関係をきちんと処理してアプリを入れていく必要があるが面倒
    - 1にあれを入れ、2にあれを入れ、、3にあれを入れ、、、、。
- NPMを使うことで、依存関係を処理してアプリを導入してくれる
  - こういうのをパッケージマネージャーと呼ぶ
  - Linuxなどで良く使われている考え方

NPMで導入

- Nodeは実行環境、NPMはアプリ管理ツールとでも
  - NodeをインストールすればNPMは勝手に付いてくる
- Nodeはインストールが必要
  - 比較的アップデートが早い
  - 導入するならLTS版を入れること（LTS＝長期サポート版）
  - Windowsはそのままインストーラーで導入
  - macはhomebrew経由で入れるのがお奨め
    - バージョンの切替ができる（Windowsでは結構面倒）

macにhomebrew経由で入れる

- homebrewが入っていない場合は入れる
  - ターミナルで以下を入力

```
$ /usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

  - インストールが終わったらnodebrewを入れる

```
$ brew install nodebrew
```

  - パスを通しておきましょう

```
$ echo 'export PATH=$HOME/.nodebrew/current/bin:$PATH' >> ~/.bash_profile
```

  - Node本体を入れる

```
$ nodebrew install-binary latest
```

macにhomebrew経由で入れる

- homebrewが入っていない場合は入れる
  - 入っているNodeを確認

```
$ nodebrew ls
```

  - 使うバージョンを指定する

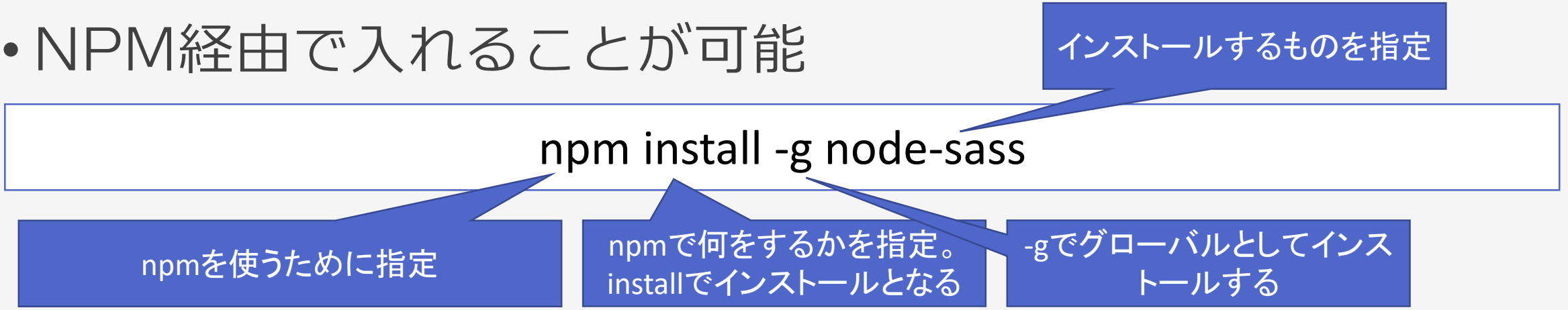
```
$ nodebrew use v7.1.0
```

  - 確認しておきましょう

```
$ node -v
```

NPMでNode-Sassを入れる

- SCSSをCSSにトランスパイルできるのがNode-sass
- NPM経由で入れることが可能



  - npmコマンドでインストールする
    - -g（ハイフンが入るので注意）でグローバルインストールになる
    - ローカルインストールとグローバルインストールがある
    - 通常はローカルインストールを行うが、どのプロジェクトでも利用するようなアプリはグローバルインストールしておくと楽

NPMでNode-Sassを入れる

- ちょっと待っているとインストールが完了する
  - 実体は変なところに入る
  - C:¥Users¥（ユーザー名）¥AppData¥Roaming¥npm
- 何かで指定する場合（WebStormとか）はここを指定する
  - AppDataが隠しフォルダーであるため、直接打たないとダメな場合もある
- 使用する際は、nodeから利用する
  - ダブルクリックしたら使える、というようなものではない

Node-Sassの使い方

- コマンドで入力
  - 対象となるフォルダーまで移動する
  - cdコマンドを利用する

```
$ cd フォルダーパス
```

    - ls（Windowsではdir）でフォルダー内のファイルを表示

  - 移動したらコマンド

```
node-sass style.scss result.css --output-style expanded
```

    - これで書き出される
    - --output-styleで書き出し方を指定可能

Node-Sassの使い方

- オプションで-wまたは-watchを付けると、ファイルを保存すると勝手にCSSを作成してくれる

```
node-sass style.scss result.css --output-style compressed -w
```

  - あとは勝手に保存すれば書き出されるので、楽
  - 止めたいときは、nodeを落とす

NPM Script

- NPMで導入した各パッケージを組み合わせて一連の動作をさせることができるもの
  - Nodeと各パッケージだけで運用できるため比較的楽
  - 生成されるファイルを渡してnpm initコマンドを打って貰えば、勝手に同じパッケージが導入されるため、複数人で同じ環境を
    容易に作ることが可能
    - 1人ががんばってscriptを記述すれば、みんなで使える
    - 各社秘伝のソースがあったりする
- 編集したSCSSをトランスパイルして、プレフィックスを付与して、自動的にブラウザーをリロードして読込直す
  - みたいなことが可能

NPM Scriptのめんどうさ

- 全てテキストでコマンドを記入していく必要がある
  - 意外にエラーが起きやすい
  - エラーは動かしてみて初めて分かることも
  - それぞれのパッケージで使用できるスイッチ（オプション）が違うため、調べるのが結構面倒
  - 一部は設定ファイルを別ファイルに記述する

NPMコマンドでScript導入

- 何らかのパッケージを導入

```
$ npm install –D node-sass
```

  - -Dを付けることで、このフォルダーにだけインストールされる
  - -gだと全体になるが、できれば、-D推奨

- initコマンドでpackage.jsonを作成

```
$ npm init
```

  - packagename:区別できるような英数字を記入（括弧内はデフォルト値。そのままEnterすればそれが入る）
  - packagename以外はデフォルトで大体平気

package.json

- npmを管理しているJSONファイル
  - 中身を編集して利用する
  - dependenciesまたは、devDependenciesにインストールしたパッケージが書き込まれていく
  - 他者にこのファイルを渡して、npm initして貰えば、勝手にインストールしてくれる仕組み
  - npm script自体は、scriptsに記述する

postcssを入れる

- CSSに後処理をしてくれるパッケージ
  - 良く使われるのは、autoprefixer
  - 自動的にベンダープレフィックスを付与してくれる
  - autoprefixerはpostcssのプラグイン扱いなので、postcssを導入し、プラグインとして導入する

```
$ npm install –D postcss-cli
$npm install –D autoprefixer
```

  - postcss.config.jsファイルを作成し、そこに設定を記述する
    - どこまでのベンダープレフィックスを補うかなどを指定可能

postcss.config.js

```
module.exports = {
  plugins: [
    require('autoprefixer')({
      "browsers": [
        "last 3 versions"
      ]
    })
  ]
}
```

```
module.exports = {
  plugins: [
    require('autoprefixer’)({
        grid: true
        })
  ],
};
```

- browsersでいろいろと設定が可能
  - last 3 versionsだと現行ブラウザーから3つ分のベンダープレフィックスが付与される
  - 右の記述だと、CSSGridのベンダープレフィックスが付与される

npm scriptでscssからcssにしてベンダープレフィックス

- scssを記述してcssにする

```
"node-sass --include-path scss scss/style.scss dist/style.css --output-style compressed"
```

- postcssでベンダープレフィックスを付与する

```
"postcss dist/style.css -o dist/style.css --use autoprefixer --no-map"
```

  - 出力先とかは任意に設定変更する
  - この場合、scssはscss/style.scssに書き込み、dist/style.cssとして変換したものを書き込み、ベンダープレフィックスを付与して上書きする

- NPM scriptでこの2つを繋げる

npm scriptでscssからcssにしてベンダープレフィックス

- package.jsonのscriptに記述

```
"build:scss": "node-sass --include-path scss scss/style.scss dist/style.css --output-style compressed",
"build:postcss": "postcss dist/style.css -o dist/style.css --use autoprefixer --no-map",
```

- コロンの左側にコマンド名（任意）を付ける
  - 右側にコマンドを記述する
  - 末尾に「,」を付与する

```
"all": "npm run build:scss && npm run build:postcss",
```

  - 2つをまとめるような記述を行う
  - ＆＆で繋げる

保存時に自動的に実行

- onchangeかnodemonパッケージを導入

```
$ npm install –D onchange
```

- package.jsonのscriptに記述

```
"watch": "onchange scss/style.scss -- npm run all",
```

- onchangeでファイルの改変を検知する
  - onchangeの後でファイル名を指定するとそれが改変されたときにnpmコマンドを実行させる

browsersync

- 変更時にブラウザーを呼び出して自動的にリロードするパッケージ

```
$ npm install –D browser-sync
```

- package.jsonのscriptに記述

```
"serve": "browser-sync start --server --files ¥"**/*¥" ",
```

- --filesで更新を検知するファイルを指定
  - **/*を指定しておけば、大体のファイルを検知できる
  - 検知したら自動的にリロードする
  - localhost:3000で表示
    - localhost:3001で管理画面を表示する

変更検知したらブラウザーをリロード

- ビルドしつつ、ブラウザーリロードを行うなど、複数の仕組みを動作させるときに使用

```
$ npm install –D concurrently
```

- package.jsonのscriptに記述

```
"dev":"concurrently ¥"npm run watch¥" ¥"npm run serve¥""
```

- npm run watchとnpm run serveが動く
  - SCSSやhtmlの変更したときに検知してサーバーがリロードされる