入門

JavaScriptドリル

第8回 インクリメンタルサーチの実装

今回の問題は「インクリメンタルサーチの実装」です。データ検索と処理と、input要素への入力やクリックのハンドリングといったUIを扱う処理が並存します。これらをどうわかりやすくコードにするかがポイントです。

2017年12月07日発行


宇野 陽太　フロントエンド・エンジニア

このシリーズの記事一覧をみる

★ このシリーズをお気に入りに登録

未読にする

目次

はじめに

「ドリル」形式の問題を通して、JavaScriptプログラムのさまざまな解決のパターンを学んでいく連載の第8回です。

さて、今回も、普段からJavaScriptを書いている方であれば簡単に書ける、また学習中の方でも解けるレベルを想定した問題です。

今回の問題のテーマは「インクリメンタルサーチの実装」です。

「インクリメンタルサーチ」について軽く説明すると、検索したい単語を全文入力していなくても、入力途中の語でリアルタイムに検索する検索方法のことを指します。

それでは問題を見ていきましょう。

問題:インクリメンタルサーチの実装

問題

次の配列を使いインクリメンタルサーチを実装してください。

```
const list1 = [
'中村 享介',
'高津戸 壮',
'小山田 晃浩',
'外村 奈津子',
'德田 和規',
'山田 順久',
'山田 敬美',
'小原 司',
'坂巻 翔大郎',
'中島 直博',
'宇野 陽太',
'丸山 陽子',
'大杉 充',
'森 大典',
'杉浦 有右嗣',
'矢倉 眞隆',
'奥野 賢太郎',
'藤田 智朗',
]

const list2 = [
'Kyosuke Nakamura',
'Takeshi Takatsudo',
'Akihiro Oyamada',
'Natsuko Sotomura',
'Kazunori Tokuda',
'Yukihisa Yamada',
'Takami Yamada',
'Tsukasa Obara',
'Shotaro Sakamaki',
'Naohiro Nakajima',
'Yota Uno',
'Yoko Maruyama',
'Makoto Oosugi',
'Hironori Mori',
'Yuji Sugiura',
'Masataka Yakura',
'Kentaro Okuno',
'Tomoaki Fujita',
]
```

仕様

- 検索入力欄として、ひとつのinput要素をもつ

- 入力欄に文字を入力したら、その配列内から入力文字列を含む項目を列挙する
- `list1`, `list2`のどちらがソースになっても動作する
- 候補の列挙中に入力欄の内容を増減させたら、それに応じて新しい候補を表示する
- 候補をクリックしたら候補を消し、入力欄にクリックした値が入力されている状態にする

以降、コードを紹介する場合には、各コードの配列の始めの要素は以外は省略します。

この問題のポイント

今回のドリルは、UIの実装です。これまでのドリルとは違い、データから特定の語を検索するといったデータを扱う処理と、input要素への入力やクリックのハンドリングといったUIを扱う処理が必要です。これらの処理が混在することになるので、いかにわかりやすく記述するかがポイントとなります。

小原の解答：filterの第二引数の利用

まずは、普段JavaScriptを書くことがないUIデザイナー小原の解答です。

```
<div class="box">
  <input type="text" class="input" placeholder="🔍 ">
  <ul class="output">
  </ul>
</div>
```

```
const names = [
  'Kyosuke Nakamura',
  // 要素省略
]
```

```
const $elements = {
  'input': document.querySelector('.input'),
  'output': document.querySelector('.output')
}

function init() {
  $elements.input.addEventListener('input', oninput, fa
  $elements.output.addEventListener('click', onlistclic
  render(names)
}

function oninput() {
  const re = new RegExp(this.value, 'i')
  const filterd = names.filter(compare, re)
  render(filterd)
}

function onlistclick(event) {
    const clickedName = event.target.textContent
  $elements.input.value = clickedName
  render([])
}

function compare(item) {
  return this.exec(item)
}

function render(ary) {
  $elements.output.textContent = null //一旦全部消す
  const lists = ary.map(name => {
    const li = document.createElement('li')
    li.textContent = name
    return li
  })
  $elements.output.append(...lists)
```

```
}

init()
```

各種イベントハンドラと、名前比較用のcompare()関数、レンダリング用のrender()関数で構成されています。名前を検索する処理は、input要素のハンドラである、oninput()関数の中に書かれています。

特徴的なのは、リストを絞り込む処理です。

```
function oninput() {
  const re = new RegExp(this.value, 'i')
  const filterd = names.filter(compare, re)
  render(filterd)
}

// 中略

function compare(item) {
  return this.exec(item)
}
```

filter()*は、array.filter(callback[, thisObject]);という構文になります。配列の要素に対して、コールバック関数を実行し、その結果がtrueになった要素からなる、新しい配列を生成します。

***注：filter()**
filter()については、次の記事に詳しく解説されています。参照してみてください。

- いまどきの配列操作 第2回 配列を別の形に変える

よくあるfilter()の使い方は、第一引数にコールバック関数を渡し、その結果をbooleanで返すというものです。compare()関数がこのコールバック関数にあたります。

このコードではさらに、第二引数を使ってコールバック関数のthisをバインドしています。第二引数に渡している、つまりcompare()関数のthisとして渡されているのは、入力値から作成した正規表現オブジェクトです。this.exec(item)という処理は、RegExpのexecメソッドを使って、itemが正規表現にマッチするか検索した結果を返す処理となります。

候補をクリックしたときの、イベントハンドラは次のようになっています。

```
function init() {
  $elements.input.addEventListener('input', oninput, fa
  $elements.output.addEventListener('click', onlistclic
  render(names)
}

// 中略

function onlistclick(event) {
  const clickedName = event.target.textContent
  $elements.input.value = clickedName
  render([])
}
```

候補の要素の挿入先である$elements.outputに対して、クリックイベントをハンドリングしています。

候補の要素は動的に追加されるため、あらかじめイベントハンドラを設定しておくことができません。そこで、親要素に対してイベントハンドラを設定し、event.targetを使って、イベントを発生させた子要素を特定するという手法をとることがあります。この手法を、イベントデリゲーションといいます。

filter()の第二引数の利用は仕様的には問題ありませんが、あまり一般的ではありません。読みやすさを意識するならcompare()に引数を渡すよ

うに置き換えても良いかもしれません。

山田順久の解答:メソッドの使い分け

続いて山田順久の解答です。

```
<div id="incremental-form">
  <input id="js-input" type="text" placeholder="Yukihis
  <div id="js-options"></div>
</div>


const list1 = [
'中村 享介',
// 要素省略
]

const list2 = [
'Kyosuke Nakamura',
// 要素省略
];

function showOptions(names) {
  const el = document.getElementById('js-options');
  el.innerHTML = '';

  for (const name of names) {
    const nameEl = document.createElement('div');
    nameEl.innerText = name;
    el.append(nameEl);
  }
```

```
  if (names.length) {
    el.style.display = 'block';
  } else {
    el.style.display = 'none';
  }
}

function init() {
  const inputEl = document.getElementById('js-input');
  const optionsEl = document.getElementById('js-options

  inputEl.addEventListener('keyup', event => {
    const input = event.target.value;
    const re = new RegExp(input, 'i');

    if (!input) {
      return showOptions([]);
    }

    const matched = [...list1, ...list2].filter(name =>
    showOptions(matched);
  });

  optionsEl.addEventListener('click', event => {
    const name = event.target.innerText;
    inputEl.value = name;
    optionsEl.innerHTML = '';
    optionsEl.style.display = 'none';
  });
}

document.addEventListener('DOMContentLoaded', init);
```

構成は前述した小原のコードと似ていて、イベントハンドラと、候補のリス

トを表示するshowOptions()関数からなっています。文字列比較に正規表現を使用している点も同じですが、filter()メソッドのコールバック関数の扱い方が異なっています。

filter()メソッドを使用している箇所を見てみましょう。

```
const matched = [...list1, ...list2].filter(name => re.
```

コールバック関数を個別の関数として分割しない代わりに、正規表現オブジェクトをコールバック関数の中で使用しています。条件式が短いコードで表現できる場合は、無理に関数を分割しなくても、簡潔でわかりやすいコードを書くことができます。

また、細かいところですが、小原のコードに出てきたexec()メソッドは、「正規表現にマッチした文字列を返し、マッチしなかったらnullを返す」メソッドです。一方で、山田のコードに出てきたtest()メソッドは、「正規表現にマッチしているかどうかを真偽値で返す」メソッドです。

exec()メソッドを使用しても、filter()メソッドは意図した結果を返すため仕様は満たせます。しかし、test()メソッドを使用したほうが、「マッチしているかどうか調べる」という意図をより明確に表すことができます。

メソッドのちょっとした使い分けからも、コードの読みやすさという点で見ると違いが出てきます。慣れてきた方は使い分けを意識してみると良いでしょう。

小山田の解答：データ処理とUI処理の分割

次は小山田の解答です。

```
<input>
<div class="suggestions"></div>


( function () {

  const list1 = [
  '中村 享介',
  // 要素省略
  ]

  const list2 = [
  'Kyosuke Nakamura',
  // 要素省略
  ]

  const input = document.querySelector( 'input' );
  const suggestionsEl = document.querySelector( '.sugge


  input.addEventListener( 'input', ( event ) => {

    if ( ! input.value ) {

      suggestionsEl.innerHTML = '';
      return;

    };

    const suggestions = getSuggestions( input.value );
    suggestionsEl.innerHTML = suggestions.reduce( ( a,

      return a + `<li><button type="button" class="appl
```

```
    }, '' );

  } );

  suggestionsEl.addEventListener( 'click', ( event ) =>

    if ( event.srcElement.classList.contains( 'applyBut

      input.value = event.srcElement.innerHTML;

    }

  } );

  function getSuggestions( text ) {

    const _list1 = list1.filter( el => new RegExp( text
    const _list2 = list2.filter( el => new RegExp( text

    return _list1.concat( _list2 );

  }

} )();
```

これまでのコードと異なり、イベントハンドラの中では、要素の生成とレンダリングを行っています。

```
const suggestions = getSuggestions( input.value );
suggestionsEl.innerHTML = suggestions.reduce( ( a, b )

  return a + `<li><button type="button" class="applyBut

}, '' );
```

reduce()*は配列の各要素に対して実行した関数の結果を、蓄積して単一の値を生成します。

***注：reduce()**
reduce()については、次の記事に詳しく解説されています。参照してみてください。

- いまどきの配列操作 第2回 配列を別の形に変える

配列から文字列を生成する方法としては、配列メソッドのmap()とjoin()を使って、array.map(...).join('')のように書くという方法もあります。ここでは、reduce()を使うことで「文字列を生成している」ということを、すっきりと書くことができています。一方で、reduce()は配列からあらゆる型の値を生成できるため、reduce()でどのような処理を行っているかを注意深く読む必要があります。

リストを絞り込む処理は、getSuggestions()関数に書かれています。DOMに依存せず、配列の処理だけを行っているので、読みやすくなっています。

```
function getSuggestions( text ) {

  const _list1 = list1.filter( el => new RegExp( text )
  const _list2 = list2.filter( el => new RegExp( text )

  return _list1.concat( _list2 );
```

```
}
```

複雑なデータやUIを扱うようなアプリケーションを作る際には、このようにデータを扱う処理とUIを制御する処理を分けるというアプローチは有用です。テストや保守も行いやすくなることから、積極的に取り入れていきたいところです。

宇野の解答：処理分割が不十分

次は筆者（宇野）の解答です。

```
<p>
課題： 次の配列を使いインクリメンタルサーチを実装してください。
</p>
<select id="searchTarget"></select>
<input id="searchText" type="text">
<div id="result"></div>


(function(){
  const list1 = [
    '中村 享介',
    // 要素省略
  ]

  const list2 = [
    'Kyosuke Nakamura',
    // 要素省略
  ]

  const listCollection = [list1, list2]
```

```
document.addEventListener('DOMContentLoaded', main)

function main() {
  const $input        = _getSearchInputElement()
  const $searchTarget = _getSearchTargetContainerElem

  _render()
  _attachEvents()

  function _render() {
    renderSearchTargets(listCollection)
    renderSearchResult(search())
  }

  function _attachEvents() {
    $input.addEventListener('keyup', onChangeSearchIn
    $searchTarget.addEventListener('change', onChange
    document.addEventListener('click', onClickName)
  }
}

function onChangeSearchInput(ev) {
  const searched = search(ev.target.value)
  renderSearchResult(searched)
}

function onChangeSearchTarget(ev) {
  renderSearchResult(search())
}

function onClickName(ev) {
  if (!ev.target.classList.contains('button')) {
    return
  }
```

```
  _getSearchInputElement().value = ev.target.textCont
  renderSearchResult([])
}

function search(value = '') {
  const $searchTarget = _getSearchTargetContainerElem
  return listCollection[$searchTarget.value].filter(t
    return target.includes(value)
  })
}

function renderSearchTargets(collection) {
  const $target = _getSearchTargetContainerElement()
  collection.forEach((_, i) => {
    const $option = _createSearchTargetElement(i)
    if (i === 0) {
      $option.selected = 'selected'
    }
    $target.appendChild($option)
  })
}

function renderSearchResult(values) {
  const $target = _getResultContainerElement()
  if ($target.hasChildNodes()) {
    Array.from($target.childNodes).forEach($el => $ta
  }

  const $container = _createListContainerElement(valu
  $target.appendChild($container)
}

function _getSearchTargetContainerElement() {
  return document.getElementById('searchTarget')
}
```

```
function _createSearchTargetElement(id) {
  const $option = document.createElement('option')
  $option.value = id
  $option.innerHTML = `list${id + 1}`
  return $option
}

function _getSearchInputElement() {
  return document.getElementById('searchText')
}

function _getResultContainerElement() {
  return document.getElementById('result')
}

function _createListContainerElement(values) {
  const $container = document.createElement('ul')
  values.forEach(value => {
    const $list = _createListElement(value)
    $container.appendChild($list)
  })
  return $container
}

function _createListElement(value) {
  const $container = document.createElement('li')
  $container.appendChild(_createButtonElement(value))
  return $container
}

function _createButtonElement(value) {
  const $container = document.createElement('button')
  $container.textContent = value
  $container.classList.add('button')
```

```
    return $container
  }

})()
```

このコードも、小山田のコードのように、リストを絞り込む処理とUIを制御する処理を分けようとしています。絞り込みは、search()関数で行い、renderSearchResult()関数が結果の描画を担当しています。

絞り込みを行っているsearch()関数を見てみましょう。

```
function search(value = '') {
  const $searchTarget = _getSearchTargetContainerElemen
  return listCollection[$searchTarget.value].filter(tar
    return target.includes(value)
  })
}
```

実はこのコードには、仕様にはない「リストを選択するUI」が含まれていて、$searchTargetには、その要素が格納されています。

本来はリストを絞り込むという処理だけを行うはずのsearch()関数が、要素を参照ししまっていて、完全にデータとUIの処理が分かれているとはいえません。先ほど「分けようとしています」と表現したのはこれが理由です。

きちんと分けるのであれば、$searchTarget.valueまたはlistCollection[$searchTarget.value]をsearch()関数が受け取るようにして、DOMの参照はsearch()関数の外側で行うべきでしょう。

坂巻の解答：datalist要素の使用

次は、坂巻の解答です。坂巻は、JavaScriptよりもHTMLやCSSを担当することが多く、他の解答者とは違った特徴的な解答をしています。

```
<input type="text" id="search" list="search-list">


const list1 = [
'中村 享介',
// 要素省略
]

const list2 = [
'Kyosuke Nakamura',
// 要素省略
]

const list = list1;
const form = document.getElementById("search");
const datalist = document.createElement('datalist');

datalist.setAttribute('id', 'search-list');

list.forEach((v, idx) => {
  let option = document.createElement('option');
  option.innerText = v;
  option.value = v;
  datalist.appendChild(option);
});

document.body.appendChild(datalist);
```

坂巻のコードは、今回の解答中もっとも少ないコード量となっています。その理由は、datalist要素*を使用している点にあります。

***注：datalist要素**
他のコントロールから使えるoption要素群を内包する要素です。次のページなども参考にしてください。

- datalist 要素 - HTML | MDN

datalist要素は、input要素などに入力を行う際、入力の候補を示すための要素です。対応する要素に何か入力があった場合に表示され、候補を選択するとその値が入力欄へと反映されます。まさに今回のドリルにぴったりの要素といえるでしょう。

解答のコードでは、このdatalist要素を、名前の入った配列を元に動的に作成することで、インクリメンタルサーチを実装しています。

要件にマッチした要素を使うという、普段HTMLに触れる機会の多い坂巻らしい解答です。

森の解答：実験的でトリッキー

最後は森の解答です。森は、一般的なイベントハンドリングやDOM操作を行わない、ということをコンセプトに解答したそうです。

インクリメンタルサーチを行うための要素を定義し、AngularやVue.jsのように、HTMLからデータを渡せるような作りになっています。

コードは大きく2つに分かれていて、UIの挙動を定義する処理と、UIに対してデータをバインディングする処理からできています。

なお、このコードを実行するには、babelなどのトランスパイラが必要です。

```
<div class="ex">
  <div>
```

```
    <h4>漢字の方</h4>
    <IncrementalSearch list="list1" placeholder="山田 太
  </div>
  <div>
    <h4>ローマ字の方</h4>
    <IncrementalSearch list="list2" placeholder="yamada
  </div>
</div>


const list1 = [
'中村 享介',
// 要素省略
];

const list2 = [
'Kyosuke Nakamura',
// 要素省略
];


class AbstractUi {
  _tempEl = document.createElement('span');
  escapeForHtml (v) {
    this._tempEl.textContent = v;
    return this._tempEl.innerHTML;
  }
}

class IncrementalSearch extends AbstractUi {

  constructor (el, id) {
    super();
    this.el = el;
    this.id = id;
```

```
    this.list = window[el.getAttribute('list')];
    this.placeholder = this.escapeForHtml(el.getAttribu
    this.el.innerHTML = this.getTemplate();
  }

  onFilterKeyup (el) {
    this.renderList(el.value);
  }

  onListClick (el) {
    this.selectList(el.textContent);
  }

  selectList (selectedValue) {
    this.el.querySelector('input').value = selectedValu
    this.renderList('');
  }

  renderList (filterValue) {
    const id = this.id;
    this.el.querySelector('ul').innerHTML = this.filter
      const html = this.escapeForHtml(data);
      return `<li onClick="${id}.onListClick(this)">${h
    }).join('');
  }

  filterList (value) {
    const v = value.toLowerCase().trim();
    return !v.length ? [] :
      this.list.filter(data => new RegExp(v).test(data.
  }

  getTemplate () {
    const id = this.id;
    const placeholder = this.placeholder;
```

```
      return `
        <input onKeyUp="${id}.onFilterKeyup(this)" placeh
        <ul></ul>
      `;
    }
  }

  class UI {

    static uiNames = [
      'IncrementalSearch'
    ];

    static _currentId = 0;

    static init () {
      this.uiNames.forEach(name => {
        document.querySelectorAll(name).forEach(el => {
          const id = 'id' + (this._currentId ++);
          this[id] = new window[name](el, 'UI.' + id);
        });
      });
    }
  }

  UI.init();
```

インクリメンタルサーチを行う要素を独自に定義するという実験的な方法をとっています。要素の定義にCustom Elementsを使用していないため、HTMLとしては未知の要素となってしまう点は注意が必要です。

まずはIncrementalSearchクラスから見ていきましょう。レンダリングやイベントハンドリング、データの絞り込みなどの、インクリメンタルサーチに必要な処理は、すべてIncrementalSearchクラスにまとまっていま

す。

コンストラクタ関数は次のようになっています。

```
  constructor (el, id) {
    super();
    this.el = el;
    this.id = id;
    this.list = window[el.getAttribute('list')];
    this.placeholder = this.escapeForHtml(el.getAttribute
    this.el.innerHTML = this.getTemplate();
  }
```

インスタンス化する際にDOMを受け取り、クラスのメンバ変数に格納しています。DOMをコンストラクタ引数として受け取ることで、クラスが特定のDOMに依存することを防いでいるのが見て取れます。

一方で、リストは`this.list = window[el.getAttribute('list')];`と、グローバルオブジェクトを直接参照しています。テンプレートで`list="list1"`のように指定したら、該当するリストを参照できるようにしたのだと思いますが、これもコンストラクタ引数で受け取ったものをメンバ変数へと格納することで、グローバルオブジェクトへの依存を防げるように感じました。データが必ずしもグローバルスコープにあるとは限らないので、工夫の余地はありそうです。

続いてはレンダリングを行っている`renderList()`メソッドです。

```
  renderList (filterValue) {
    const id = this.id;
    this.el.querySelector('ul').innerHTML = this.filterLi
      const html = this.escapeForHtml(data);
      return `<li onClick="${id}.onListClick(this)">${htm
```

```
    }).join('');
  }
```

絞り込んだ配列を使って、要素を生成している点は他の解答者と同様です。特徴的なのは、クリックイベントのハンドリングにonclick属性を使っている点です。onclick属性でハンドリングを行うには、イベントハンドラがグローバルスコープからアクセス可能となっている必要があります。後述しますが、UIクラス経由でidにグローバルスコープからのパスが入り、それを文字列に展開することで、グローバルスコープからアクセスできるという仕組みのようです。

次は絞り込みを行っているfilterList()メソッドを見てみましょう。

```
  filterList (value) {
    const v = value.toLowerCase().trim();
    return !v.length ? [] :
      this.list.filter(data => new RegExp(v).test(data.tc
  }
```

入力値とリストの文字列をそれぞれtoLowerCase()メソッドで小文字に変換しています。小文字に変換することで、大文字と小文字を区別せずに比較できるようになっています。正規表現を使った解答でも、iオプションを使用して大文字小文字を区別せずに比較できるようにしていました。仕様には明記されていませんが、解答者のUIに対する配慮が窺えます。

また、IncrementalSearchクラスはAbstractUiクラスを継承しています。次はAbstractUiクラスを見てみましょう。

```
  class AbstractUi {
    _tempEl = document.createElement('span');
    escapeForHtml (v) {
      this._tempEl.textContent = v;
```

```
    return this._tempEl.innerHTML;
  }
}
```

名前から察するにAbstractUiクラスは、IncrementalSearch以外のUIクラスが現れたときに、共通の処理を提供するためのクラスのようです。「Abstract（抽象）」とついていますが、ここでは「共通処理をくくっている」と考えるのが良さそうです。

escapeForHtml()メソッドでは、引数をspan要素のtextContentに入れたあと、span要素のinnerHTMLを返しています。

何をしているのかわかりづらいですが、ここではHTML文字列に含まれる<や>などの文字列を、<や>といったHTML内でも使用できる文字列に置換しています。このような処理をサニタイジング（無害化）と呼び、一般的なフレームワークやライブラリでも同様の処理が行われています。

replace()で置換していったほうが、コードとしての可読性はあがります。ですが、この方法だとブラウザの処理に任せることができるので、漏れをなくせるというメリットがあります。

どちらの書き方がいいかはケースバイケースですが、このような書き方もあるということは、覚えておいて損はないでしょう。

ちなみに、リストの中にはHTML文字列は含まれていませんが、ここにも森なりの配慮が伺えます。

最後にUIクラス*を見てみます。

```
class UI {

  static uiNames = [
    'IncrementalSearch'
  ];
```

```
  static _currentId = 0;

  static init () {
    this.uiNames.forEach(name => {
      document.querySelectorAll(name).forEach(el => {
        const id = 'id' + (this._currentId ++);
        this[id] = new window[name](el, 'UI.' + id);
      });
    });
  }
}

UI.init();
```

***注：UIクラス**
AbstractUi というクラスがあるので、ここでUI という名前を使うのは避けたほうが良かったかもしれません。

staticでいろいろと定義されていますが、メインとなるのはinit()メソッドのようです。

init()メソッドでは、uiNamesに定義した文字列で要素を検索し、それぞれの要素名と同名のクラスをインスタンス化しています。インスタンス化する際にユニークなIDを生成し、このIDをキーとしたプロパティを宣言しています。

さらに、このIDは、頭にUI.という文字列を追加して、コンストラクタ引数へと渡しています。前述したrenderList()メソッドで出てきたidの正体はここで生成したIDです。このIDがグローバルスコープにいるUIクラスのメンバ変数までのパスを表し、onclick属性のハンドラを参照する、という仕組みになっているようです。

筆者の感想としては、やはりグローバルオブジェクトへの依存が気になり

ました。この依存をなくせるように工夫すれば、より良いコードになると思います。

全体を通して実験的な内容となっていたため、処理を細かく解説してみました。実際の業務では、このような処理はライブラリやフレームワークに任せ、自分で書くことは滅多にないでしょう。ですが、その中には実務に直接役立つようなコードも入っています。興味を持った方は、普段自分が使っているライブラリのコードを読んでみると、学びを得られるかもしれません。

おわりに

今回の問題は、これまでの問題とは違い「UIを実装する」というものでした。

UIの実装で頻出する、イベントハンドリングやDOMの操作・生成などは、ライブラリやフレームワークのサポートを受けて、簡単に書くことができるようになっています。しかし、それらの処理の実装方法を知ることは、とても大切です。

先ほども述べましたが、この記事を読んで興味を持った方は、ぜひ普段自分が使っているライブラリのコードを読んでみてください。

また、ここで紹介した以外にも良い方法があると思いますので、チャレンジされた方は`#codegrid`を付けて、ぜひツイートしてください。

宇野 陽太 フロントエンド・エンジニア

大手ソフトウェアダウンロード販売会社、ソーシャルアプリケーションプロバイダーなどで、デザイナー、マークアップ・エンジニア、フロントエンド・エンジニアとして、主にスマートフォン向けゲームの開発に携わる。2015年1月に株式会社ピクセルグリッドに入社。 MVCフレームワークを用いたJavaScriptの実装や、メンテナブルなCSSの設

計を得意とする。