

仮想 DOM を自作した

WD3A 岡崎 流依

# 目次

1. 作成に至った背景
2. 完成した物
3. 実際のコード
4. 解説

作成に至った背景

作成に至った背景

普段から仮想 DOM を扱うライブラリを使って開発しているから作ってみたい...

完成した物

# カウント: 0

increment



decrement

# コードの解説と設計

1. 要件定義
2. ディレクトリ構造
3. コードの流れ
4. コードの解説

## 要件定義

- HTML には親となる要素を作成する
- 仮想 DOM から実際の DOM を生成する
- 仮想 DOM の差分を検知し、差分を適用させる
- ブラウザの表示を更新することができる
- state を作成する



## ディレクトリ構造

```
src
├── app.ts //bundleされる親ファイルです。
├── main.ts //importすると仮想DOMが描画されます。
├── func //関数を格納しています
│   ├── createElement.ts
│   ├── createVNode.ts
│   ├── diff.ts
│   └── mount.ts
├── state //lifecycle の状態や描画中の DOM のデータを格納しています
│   └── index.ts
├── types //型情報を格納しています
│   └── VirtualNode.ts
```



## 仮想 DOM の型情報

htmlTagName : div や h1 など

props : class や type などの属性が入ります

children : タグの中身が入ります。要素か文字列が入るので再起的な型にしています。

(VirtualNode または string の配列)

```
// types/VirtualNode.ts

export type VirtualNode = {
  htmlTagName: string;
  props: {
    [key: string]: string;
  };
  children: (VirtualNode | string)[];
};
```

# 仮想 DOM を作成するヘルパー関数

自分で定義していくのは大変なのでヘルパー関数を作成します

```
// func/createVNode.ts

const h = (
  tagName: VirtualNode['tagName'],
  props: VirtualNode['props'],
  children: VirtualNode['children']
): VirtualNode => ({
  tagName,
  props,
  children,
});
```

## 仮想 DOM から実際の DOM 要素を作成する関数(全体)

```
// func/createElement.ts

const createElement = (vnode: VirtualNode | string): HTMLElement | Text => {
  if (typeof vnode === 'string') return document.createTextNode(vnode);
  const el = document.createElement(vnode.tagName);
  if (vnode.props)
    for (const [key, value] of Object.entries(vnode.props)) {
      el.setAttribute(key, value);
    }
  if (vnode.children)
    for (const child of vnode.children) {
      el.appendChild(createElement(child));
    }
  return el;
};
```

## **vnode の型に string があるのは何故？**

children がなくなるまで再起的に実行されるので、文字列データが引数に来る恐れがあるため

```
const createElement = (vnode: VirtualNode | string): HTMLElement | Text => {  
  ...  
};
```

**vnode は仮想 DOM || string なので、string の場合は TextNode を返却するようにする**

```
const createElement = (vnode: VirtualNode | string): HTMLElement | Text => {  
  if (typeof vnode === 'string') return document.createTextNode(vnode);  
  
  ...  
};
```

仮想 DOM から実際の DOM 要素を作成する関数

## createElement(仮想 DOM のタグ名)で要素を作成する

```
const createElement = (vnode: VirtualNode | string): HTMLElement | Text => {  
  ...  
  const el = document.createElement(vnode.htmlTagName);  
  
  ...  
};
```



## 仮想 DOM に props が存在した場合は属性を追加していく

※ `el.setAttribute(key, value);` ではなく、`el[key] = value` とすると typescript がエラーを吐くのでメソッドを使いましょう。（プロパティの型情報が不確定でエラーになる）

```
const createElement = (vnode: VirtualNode | string): HTMLElement | Text => {  
  ...  
  
  if (vnode.props)  
    for (const [key, value] of Object.entries(vnode.props)) {  
      el.setAttribute(key, value);  
    }  
  
  ...  
};
```

## children が存在する場合要素がなくなるまで処理を回す

```
const createElement = (vnode: VirtualNode | string): HTMLElement | Text => {  
  ...  
  
  if (vnode.children)  
    for (const child of vnode.children) {  
      el.appendChild(createElement(child));  
    }  
  
  ...  
};
```

## 処理の過程で生成した HTML 要素を返却する

```
const createElement = (vnode: VirtualNode | string): HTMLElement | Text => {  
  ...  
  return el;  
};
```

## 差分を検知する関数(全体)

```
const diff = (
  el: HTMLElement | ChildNode,
  oldVNode: VirtualNode | string | undefined,
  newVNode?: VirtualNode | string
) => {
  if (!newVNode && !oldVNode) return;

  if (!newVNode) return el.remove();
  if (!oldVNode) return el.appendChild(createElement(newVNode));

  const replace = () => el.replaceWith(createElement(newVNode));

  if (typeof oldVNode === 'string' || typeof newVNode === 'string')
    if (oldVNode !== newVNode) return replace();
    else return;

  if (oldVNode.htmlTagName !== newVNode.htmlTagName) return replace();

  const oldProps = JSON.stringify(Object.entries(oldVNode.props).sort());
  const newProps = JSON.stringify(Object.entries(newVNode.props).sort());
  if (oldProps !== newProps) return replace();

  [...el.childNodes].forEach((child, index) => {
    diff(child, oldVNode.children[index], newVNode.children[index]);
  });
};
```

## 仮想 DOM が存在しない場合は処理を終了する

```
const diff = (  
  el: HTMLElement | ChildNode,  
  oldVNode: VirtualNode | string | undefined,  
  newVNode?: VirtualNode | string  
) => {  
  if (!newVNode && !oldVNode) return;  
  
  ...  
};
```

## どちらかの値がない場合は即更新する

新しい仮想 DOM(newVNode)の値がない場合は要素を削除する。

古い仮想 DOM(oldVNode)の値が undefined の場合は新しい仮想 DOM で HTML 要素を生成する

```
const diff = (  
  el: HTMLElement | ChildNode,  
  oldVNode: VirtualNode | string | undefined,  
  newVNode?: VirtualNode | string  
) => {  
  ...  
  
  if (!newVNode) return el.remove();  
  if (!oldVNode) return el.appendChild(createElement(newVNode));  
  
  ...  
};
```

## TextNode の判定

string の値が異なる場合は要素を置き換える

```
const diff = (  
  el: HTMLElement | ChildNode,  
  oldVNode: VirtualNode | string | undefined,  
  newVNode?: VirtualNode | string  
) => {  
  ...  
  
  const replace = () => el.replaceWith(createElement(newVNode));  
  
  if (typeof oldVNode === 'string' || typeof newVNode === 'string')  
    if (oldVNode !== newVNode) return replace();  
    else return;  
  
  ...  
};
```

## html の要素名の判定

html の要素名が違う場合は要素を置き換える

```
const diff = (  
  el: HTMLElement | ChildNode,  
  oldVNode: VirtualNode | string | undefined,  
  newVNode?: VirtualNode | string  
) => {  
  ...  
  
  if (oldVNode.htmlTagName !== newVNode.htmlTagName) return replace();  
  
  ...  
};
```



## props(属性)の判定

JSON.stringify を用いてオブジェクト同士の判定をする場合、順序性を保証できないため entries でオブジェクトを配列に変換し、sort をすることで順序性を保証することができる。

```
const diff = (  
  el: HTMLElement | ChildNode,  
  oldVNode: VirtualNode | string | undefined,  
  newVNode?: VirtualNode | string  
) => {  
  ...  
  
  const oldProps = JSON.stringify(Object.entries(oldVNode.props).sort());  
  const newProps = JSON.stringify(Object.entries(newVNode.props).sort());  
  if (oldProps !== newProps) return replace();  
  
  ...  
};
```

## children の判定

```
const diff = (  
  el: HTMLElement | ChildNode,  
  oldVNode: VirtualNode | string | undefined,  
  newVNode?: VirtualNode | string  
) => {  
  ...  
  
  [...el.childNodes].forEach((child, index) => {  
    diff(child, oldVNode.children[index], newVNode.children[index]);  
  });  
};
```

## global state

```
const globalState: {  
  lifecycle: 'beforeMount' | 'afterMount';  
  oldVNode: VirtualNode;  
  newVNode: VirtualNode;  
  viewElement: HTMLElement | Text;  
  state: [string, string | number][];  
} = {  
  lifecycle: 'beforeMount',  
  oldVNode: defaultElement,  
  newVNode: defaultElement,  
  viewElement: createElement(defaultElement),  
  state: [],  
};
```

## 差分を監視する

30ms 毎に差分がないかをチェックする。

lifecycle が beforeMount(HTML の DOM が生成前)の場合は mount 関数を実行し、仮想 DOM をマウントする。

```
const main = () => {
  const { lifecycle, viewElement, oldVNode, newVNode } = globalState;
  if (lifecycle === 'beforeMount') mount(viewElement);
  else diff(viewElement, oldVNode, newVNode);
};

(() => {
  main();
  window.setInterval(() => main(), 30);
})();
```

完成～！！🎉🎉🎉