# RxJSの紹介

2018-06-25

能城

# RxJSとは

- “**R**eactive E**x**tensions for **J**ava**S**cript”
  リアクティブプログラミング用のJavaScriptのライブラリ

- 「リアクティブプログラミング」とは？

# 概要

- RxJSを用いるとどう開発効率が上がるのか、具体例を用いて紹介
- 普通に実装した場合とRxJSを使った場合を比較

## キーワード

RxJS, Observable, JavaScript, リアクティブプログラミング

# 作りたいもの

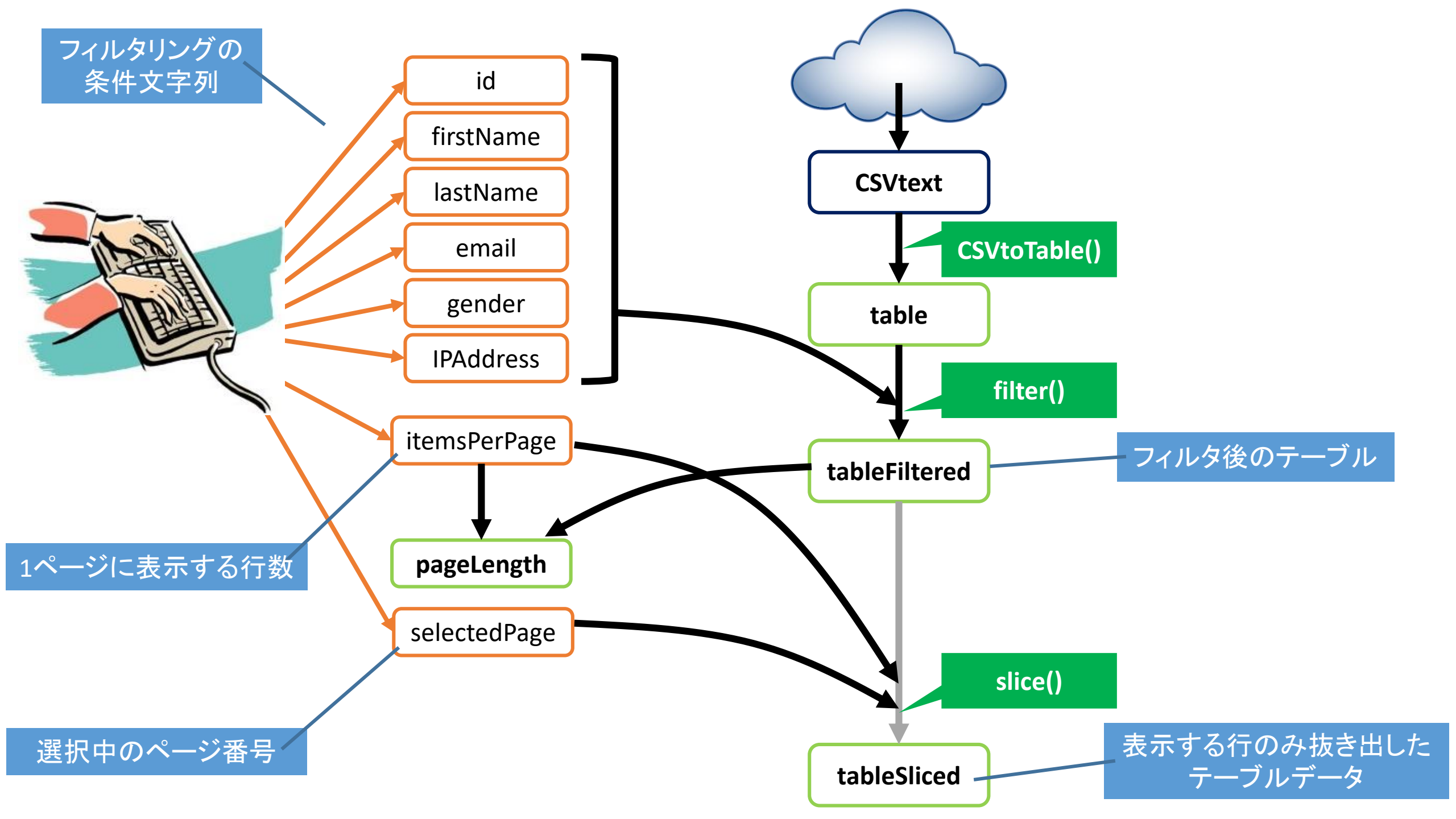
data-table-app

# 目次

1. とりあえず愚直に実装
2. 修正版
3. RxJSを使ってリアクティブに実装

# 目次

1. とりあえず愚直に実装
2. 修正版
3. RxJSを使ってリアクティブに実装


フィルタリングの
条件文字列
id
firstName
lastName
email
gender
IPAddress
itemsPerPage
pageLength
selectedPage
1ページに表示する行数
選択中のページ番号
CSVtext
CSVtoTable()
table
filter()
tableFiltered
フィルタ後のテーブル
slice()
tableSliced
表示する行のみ抜き出した
テーブルデータ

# 変数宣言

```
// 各列のテキストボックスの値
let id = '';
let firstName = '';
let lastName = '';
let email = '';
let gender = '';
let IPAddress = '';

let itemsPerPage = 50;  // 1ページに表示する列数
let pageNumber = 1;  // 現在のページ番号
let pageLength = 1;  // ページ数

let table = [];  // テーブルデータ
let tableFiltered = [];  // フィルタ後のテーブルデータ
let tableSliced = [];  // tableFilteredのうち現在のページの部分
```

# CSVファイル取得

```
const url =
   'https://dl.dropboxusercontent.com/s/i480egosiyzkpu3/sample.csv';

const req = new XMLHttpRequest();
req.open('get', url, true);
req.send(null);
req.addEventListener('load', event => {  // getが完了したときに行う処理
   const csvText = event.target.responseText;
   table = CSVtoTable( csvText );
   filter();  // tableFilteredを更新
   slice();  // tableSlicedを更新
   renderAll();  // 表示を更新
});
```

# テキストボックス入力イベントに対する処理を登録

```
document.getElementById('first_name')  // first nameの入力欄
    .addEventListener('input', event => {  // input => 1文字ごとに発火
        firstName = event.target.value;
        filter();  // tableFilteredを更新
        updatePageLength();  // pageLengthを更新
        slice();  // tableSlicedを更新
        renderAll(); // 表示を更新
});
```

```
document.getElementById('items-per-page')  // 表示行数の入力欄
    .addEventListener('input', event => {
        itemsPerPage = event.target.valueAsNumber;
        updatePageLength();  // pageLengthを更新
        slice();  // tableSlicedを更新
        renderAll(); // 表示を更新
});
```

# 変数の値を表示

```
const renderAll = () => {
    // テーブルを表示
    setTableData( document.getElementById('my-data-table'), tableSliced );

    // フィルタ後の行数を表示
    document.getElementById('nof-items').innerText = tableFiltered.length;

    // ページ数を表示
    document.getElementById('page-length').innerText = (pageLength || 1);

    // 1ページあたりの表示行数を指定するテキストボックスの値を更新
    document.getElementById('items-per-page').value = (itemsPerPage || 50);

    // ページ番号を指定するテキストボックスの値を更新
    document.getElementById('page-number').value = (pageNumber || 1);
};
```

# 目次

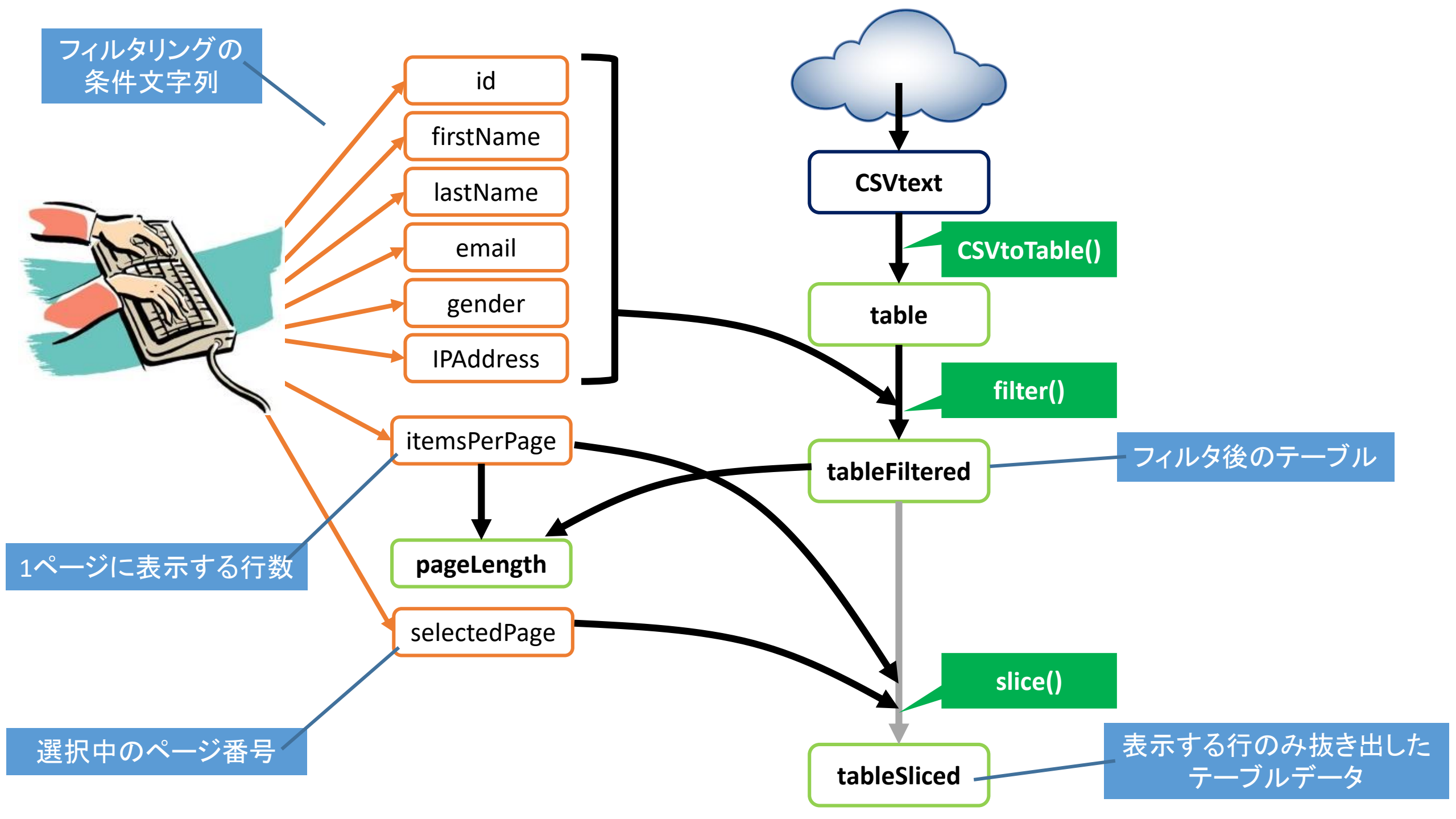
1. とりあえず愚直に実装
2. 修正版
3. RxJSを使ってリアクティブに実装

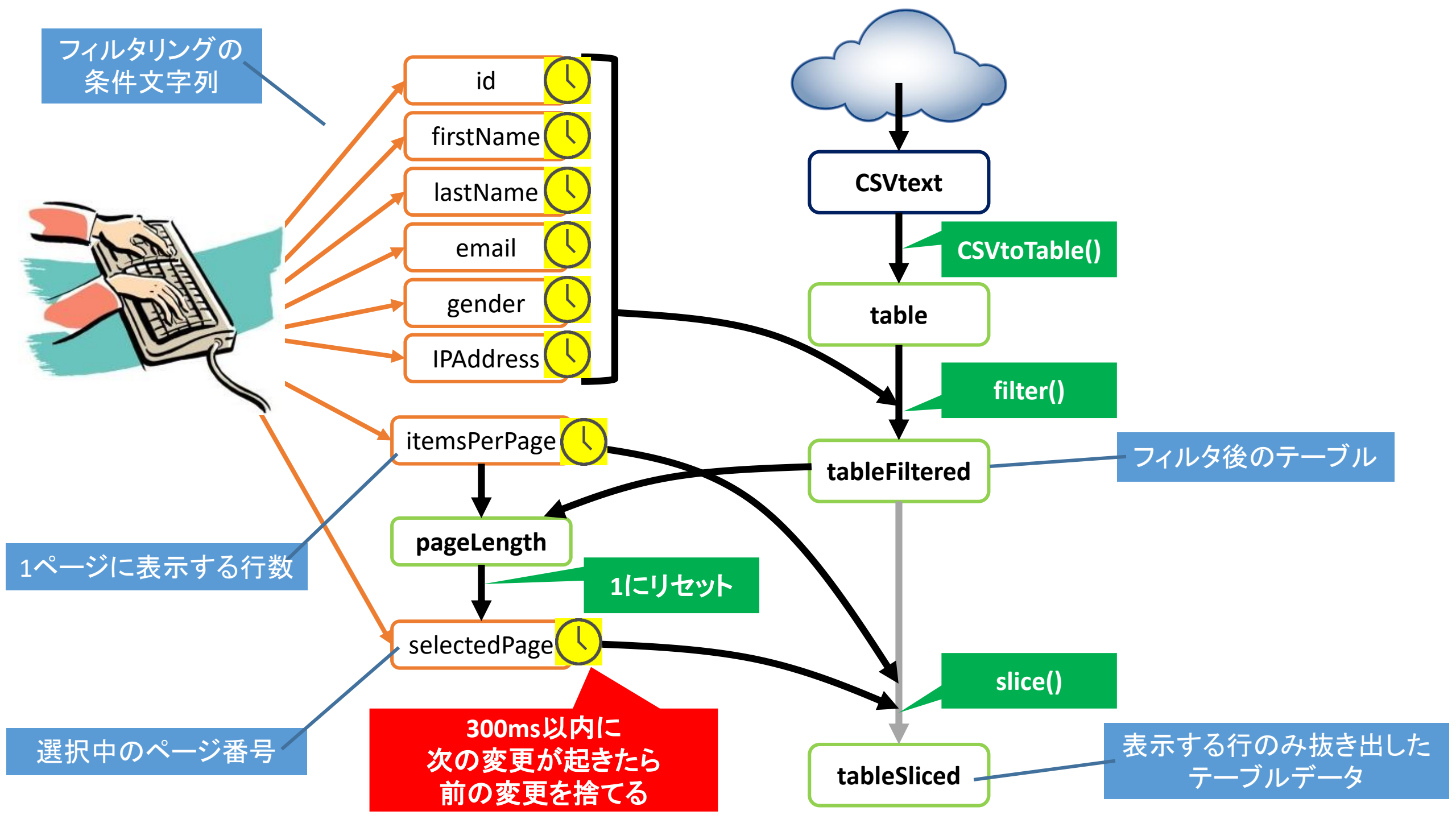
# 問題点

- テキストボックスに1文字入力するたび filter が走る
  データが大きいとブラウザ画面が固まる…

  比較
  - data-table-app (version 1)
  - data-table-app（完成版）

- フィルタや表示行数変更が起きた時に、
  存在しないページを選択したままになってしまう
  （これは pageLength の変更時に pageNumber を1にするだけなので簡単）


フィルタリングの
条件文字列
id
firstName
lastName
email
gender
IPAddress
itemsPerPage
pageLength
selectedPage
1ページに表示する行数
選択中のページ番号
CSVtext
CSVtoTable()
table
filter()
tableFiltered
フィルタ後のテーブル
slice()
tableSliced
表示する行のみ抜き出した
テーブルデータ


フィルタリングの
条件文字列
id
firstName
lastName
email
gender
IPAddress
itemsPerPage
pageLength
1にリセット
selectedPage
1ページに表示する行数
選択中のページ番号
300ms以内に
次の変更が起きたら
前の変更を捨てる
CSVtext
CSVtoTable()
table
filter()
tableFiltered
フィルタ後のテーブル
slice()
tableSliced
表示する行のみ抜き出した
テーブルデータ

# Before（再掲）

```
document.getElementById('first_name')  // first nameの入力欄
    .addEventListener('input', event => {
        firstName = event.target.value;
        filter();
        slice();
        renderAll();
});
```

# After（黄色枠部分を追加）

```
let timerId;
document.getElementById('first_name') // first nameの入力欄
    .addEventListener('input', event => {
        clearTimeout(timerId);
        timerId = setTimeout( () => {
            firstName = event.target.value;
            filter();
            slice();
            renderAll();
        }, 300 );
    });
```

既にタイマー予約されている（IDがtimerIdの）タスクをキャンセル

- 300ms後に実行するように予約
- timerIdを更新

# すべてのテキストボックスに適用

```
{
   let timerId;
   document.getElementById('id')
      .addEventListener('input', event => {
         clearTimeout(timerId);
         timerId = setTimeout( () => {
            id = event.target.value;
            filter();
            slice();
            renderAll();
         }, timeToDebounce );
      });
}
```

```
{
   let timerId;
   document.getElementById('email')
      .addEventListener('input', event => {
         clearTimeout(timerId);
         timerId = setTimeout( () => {
            email = event.target.value;
            filter();
            slice();
            renderAll();
         }, timeToDebounce );
      });
}
```

```
{
   let timerId;
   document.getElementById('items-per-page')
      .addEventListener('input', event => {
         clearTimeout(timerId);
         timerId = setTimeout( () => {
            itemsPerPage = event.target.valueAsNumber;
            pageNumber = 1;
            slice();
            renderAll();
         }, timeToDebounce );
      });
}
```

```
{
   let timerId;
   document.getElementById('first_name')
      .addEventListener('input', event => {
         clearTimeout(timerId);
         timerId = setTimeout( () => {
            firstName = event.target.value;
            filter();
            slice();
            renderAll();
         }, timeToDebounce );
      });
}
```

```
{
   let timerId;
   document.getElementById('gender')
      .addEventListener('input', event => {
         clearTimeout(timerId);
         timerId = setTimeout( () => {
            gender = event.target.value;
            filter();
            slice();
            renderAll();
         }, timeToDebounce );
      });
}
```

```
{
   let timerId;
   document.getElementById('page-number')
      .addEventListener('input', event => {
         clearTimeout(timerId);
         timerId = setTimeout( () => {
            pageNumber = event.target.valueAsNumber;
            slice();
            renderAll();
         }, timeToDebounce );
      });
}
```

```
{
   let timerId;
   document.getElementById('last_name')
      .addEventListener('input', event => {
         clearTimeout(timerId);
         timerId = setTimeout( () => {
            lastName = event.target.value;
            filter();
            slice();
            renderAll();
         }, timeToDebounce );
      });
}
```

```
{
   let timerId;
   document.getElementById('ip_address')
      .addEventListener('input', event => {
         clearTimeout(timerId);
         timerId = setTimeout( () => {
            IPAddress = event.target.value;
            filter();
            slice();
            renderAll();
         }, timeToDebounce );
      });
}
```

やってられない

# 不満点

- 大した事をやっていないのに記述量が多い
  これだけの問題なら共通部分を関数化すればよいが…

- コードを読んだときに意図が分かりにくい
  「300ms以内の連続した入力では発火しない」という意味を読み取るのが難しい（コメントが無かったら死ぬ）

（そもそもバージョン1も2も）

- 各イベントに対応する処理をすべてその場で書いている
  ＝ 変数の依存関係の全体像（さっきの図）を常に意識しなければならない
  → ソースコード内の相互依存性が高い

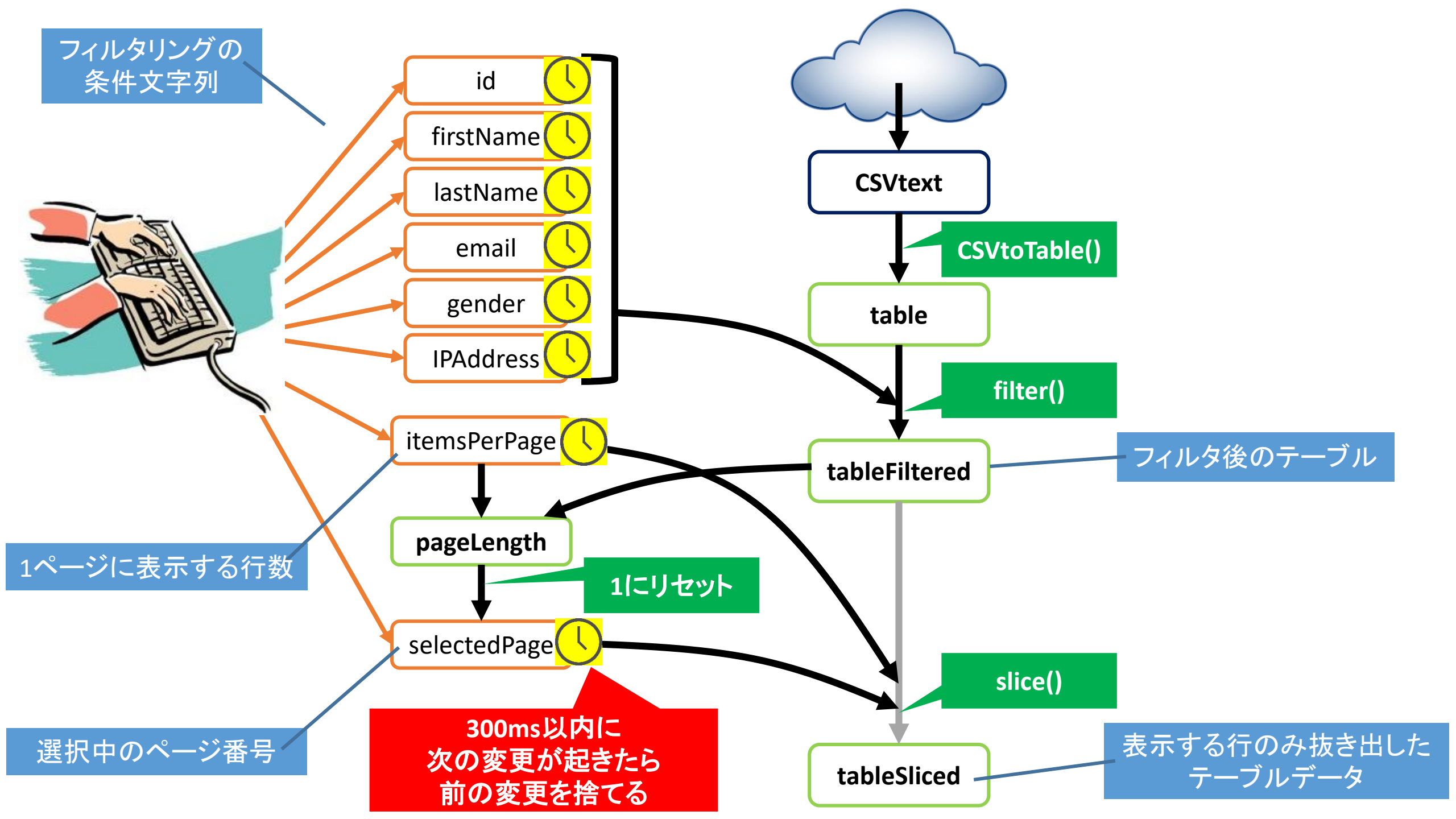
# 目次

1. とりあえず愚直に実装
2. 修正版
3. RxJSを使ってリアクティブに実装

# RxJSによる実装では

- ただの変数の代わりに“**Observable**”を使う
  - **Observable**とは、
    - 「値が更新されたときに発火する（アクションを起こす）変数」
    - 「値の更新を別の**Observable**変数へ伝播する変数」

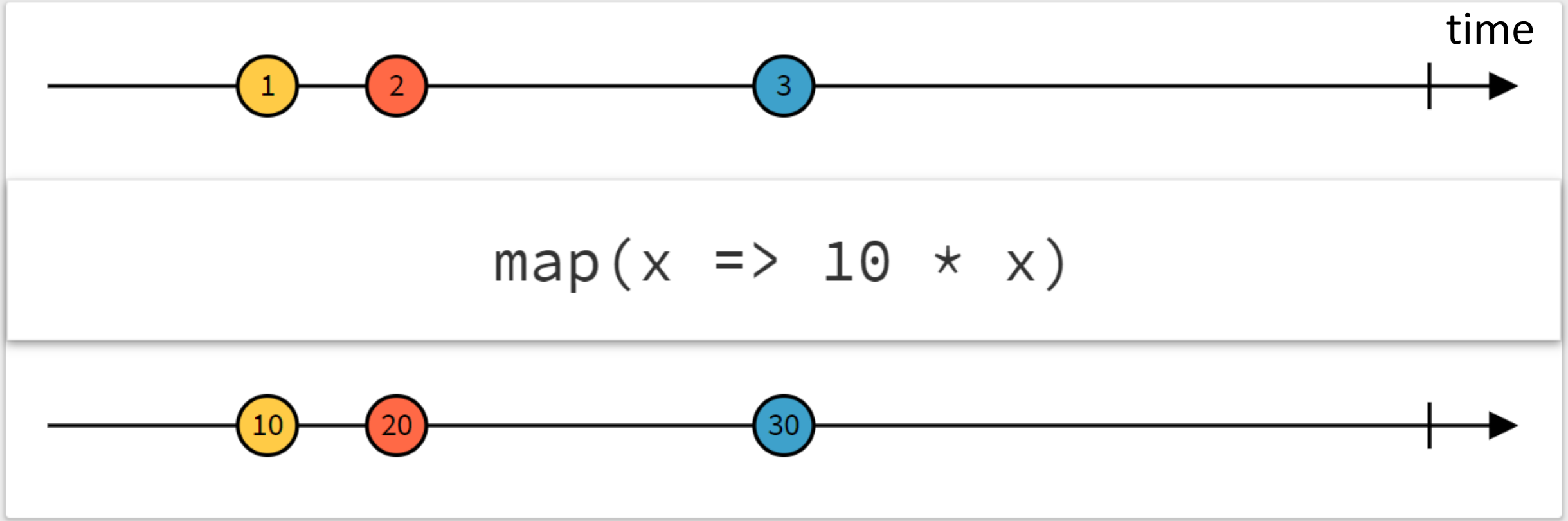
    あるいは
    - 「値の更新を監視できる変数」
- 依存関係を記述する

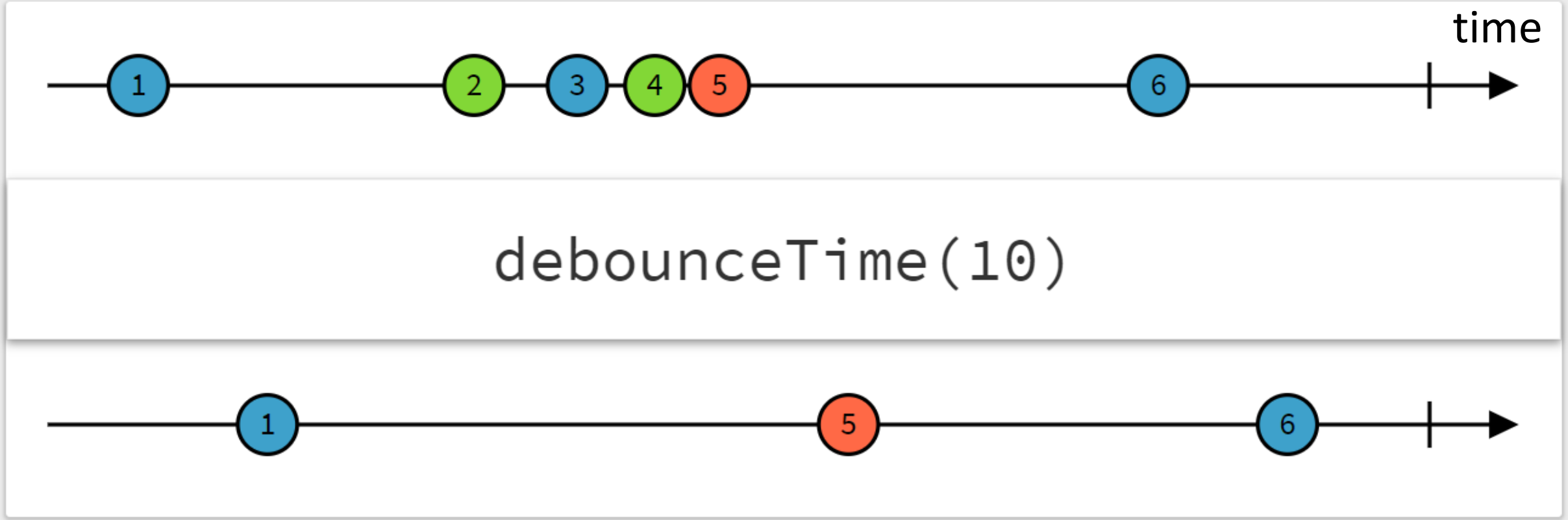

フィルタリングの
条件文字列
id
firstName
lastName
email
gender
IPAddress
itemsPerPage
pageLength
1にリセット
selectedPage
1ページに表示する行数
選択中のページ番号
300ms以内に
次の変更が起きたら
前の変更を捨てる
CSVtext
CSVtoTable()
table
filter()
tableFiltered
フィルタ後のテーブル
slice()
tableSliced
表示する行のみ抜き出した
テーブルデータ

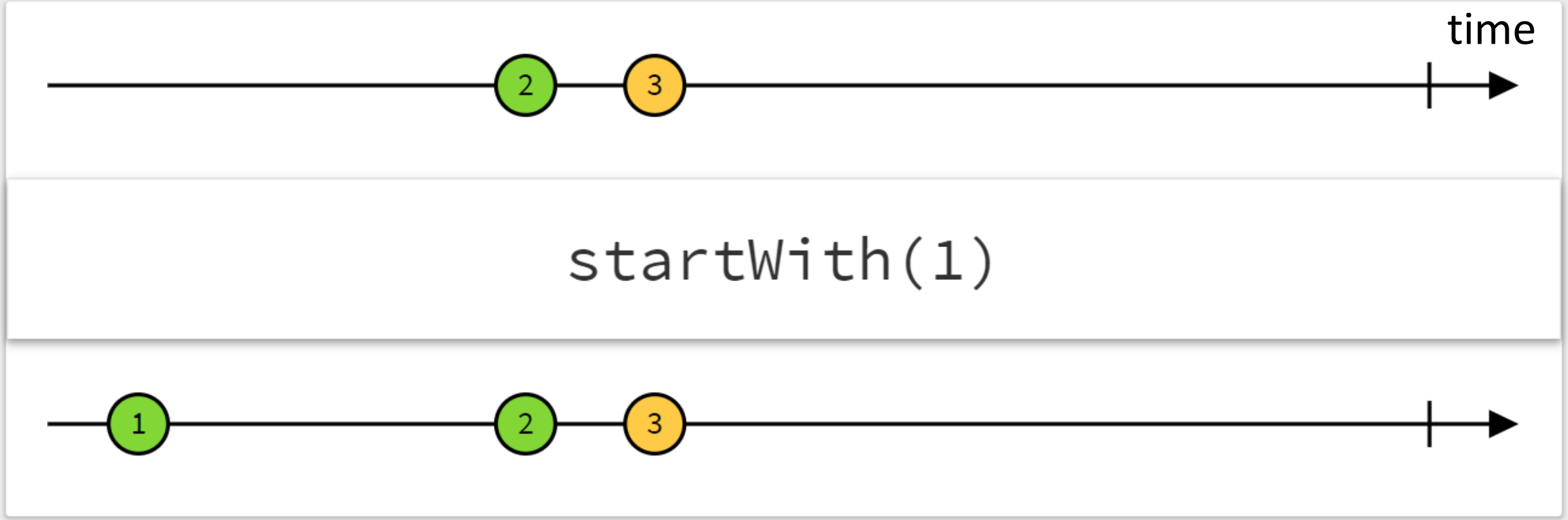
```
let timerId;
document.getElementById('first_name') // first nameの入力欄
    .addEventListener('input', event => {
        clearTimeout(timerId);
        timerId = setTimeout( () => {
            firstName = event.target.value;
            filter();
            slice();
            renderAll();
        }, 300 );
    });
```

イベントからObservableを生成する関数

```
const firstName$
    = fromEvent( document.getElementById('first_name'), 'input' )
        .pipe(   map( event => event.target.value ),
                 debounceTime(300),  // 300ms以内の連続した発火を抑制
                 startWith('') );  // 初期値を''に
```


time
1
2
3
map(x => 10 * x)
10
20
30

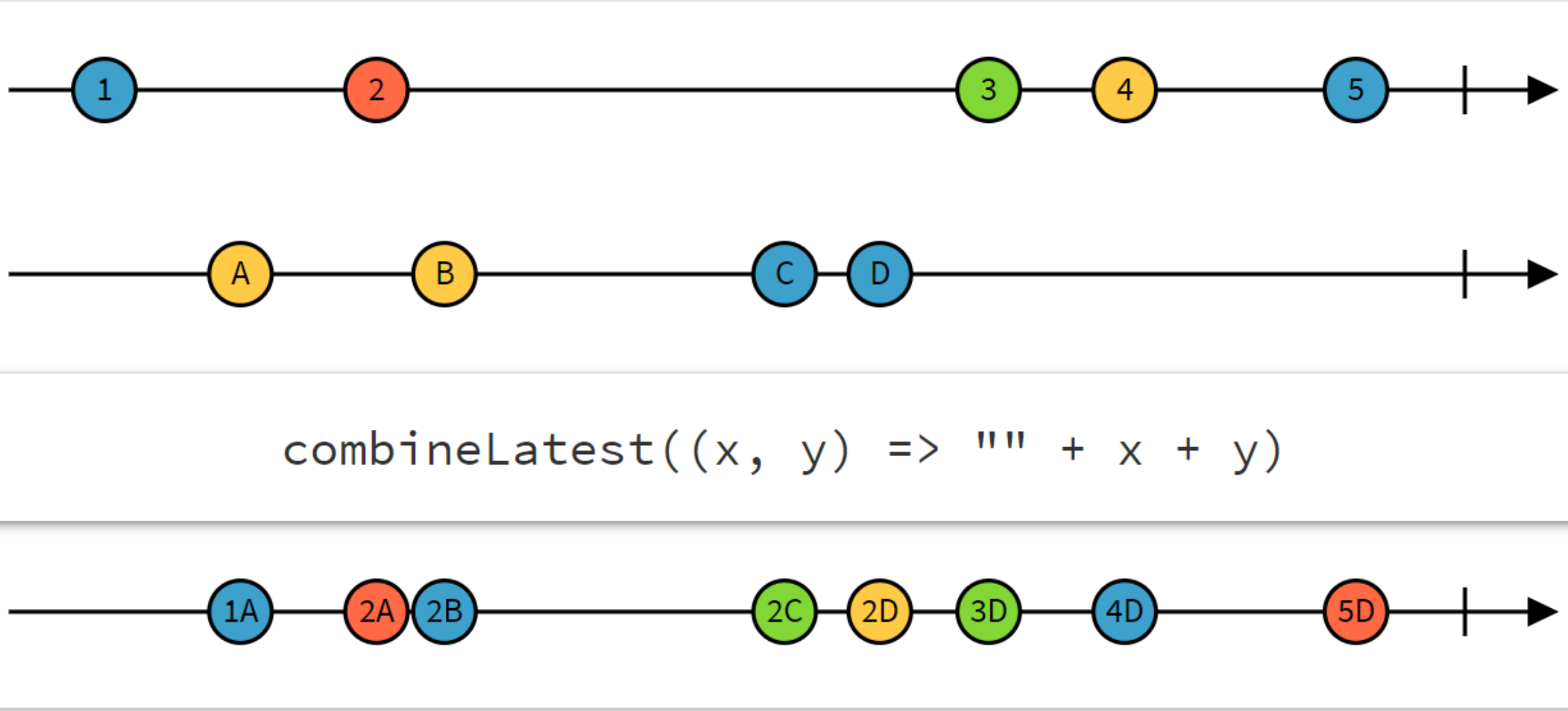

time
1
2
3
4
5
6
debounceTime(10)
1
5
6


time
2
3
startWith(1)
1
2
3

```
const csvText$ = (() => {
      const req = new XMLHttpRequest();
      req.open('get', url, true);
      req.send(null);
      return fromEvent( req, 'load' )
                .pipe(  map( event => event.target.responseText ),
                        startWith('') );  // 初期値を''に
   })();

const table$ = csvText$.pipe( map( CSVtoTable ) );
```

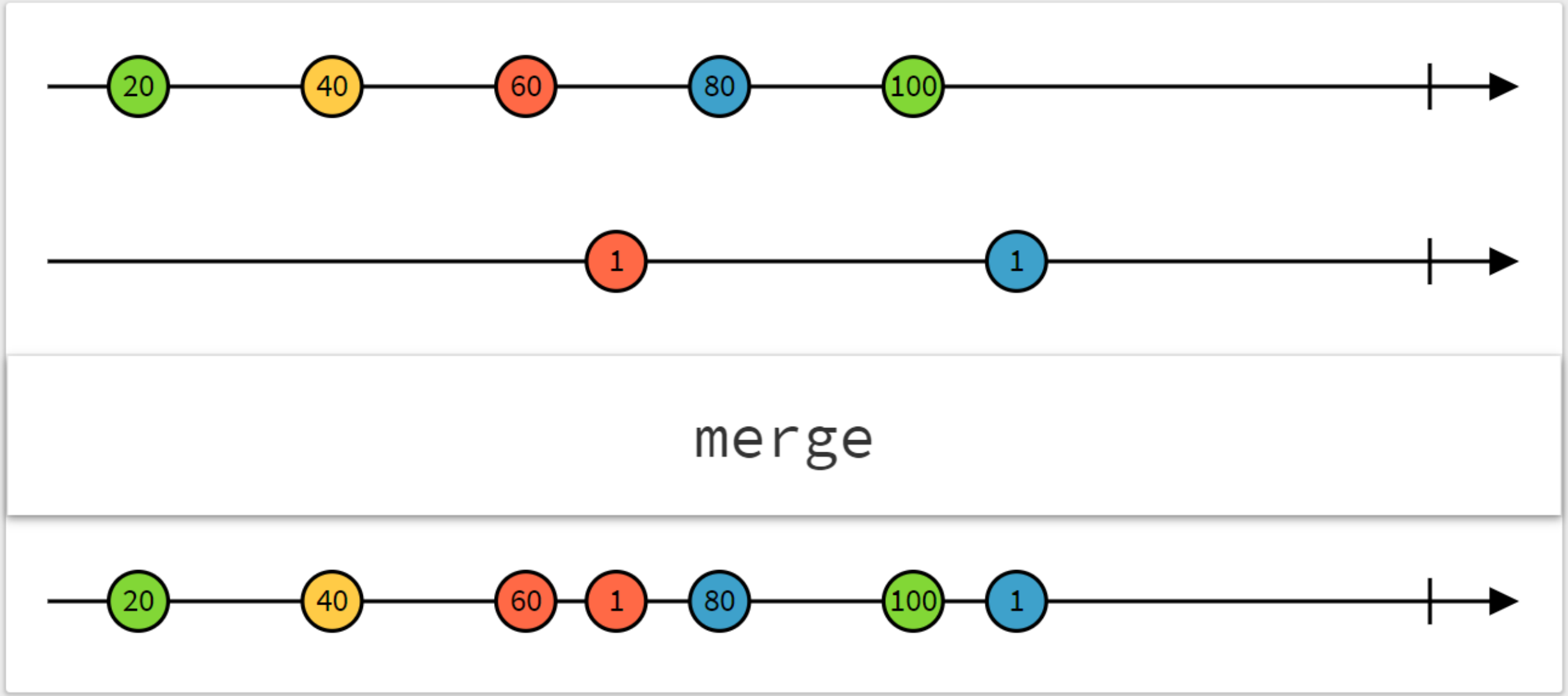
```
const tableFiltered$ = combineLatest(
        table$,
        id$,
        firstName$,
        lastName$,
        email$,
        gender$,
        IPAddress$,
        (table, ...input) =>
           table.filter( line => filterFunction( line, ...input ) ) );
```


1
2
3
4
5
A
B
C
D
combineLatest((x, y) => "" + x + y)
1A
2A
2B
2C
2D
3D
4D
5D

```
const itemsPerPage$
   = fromEvent( document.getElementById('items-per-page'), 'input' )
        .pipe(   map( event => event.target.valueAsNumber ),
                 map( val => (val || 50) ),
                 debounceTime(300),
                 startWith(50) );
```

```
const pageLength$ = combineLatest(
     tableFiltered$,
     itemsPerPage$,
     (table, itemsPerPage) =>
        Math.ceil( table.length / itemsPerPage ) )
   .pipe( startWith(1) );
```

```
const pageNumber$ = merge(
      fromEvent( document.getElementById('page-number'), 'input' )
         .pipe(  map( event => event.target.valueAsNumber ),
                 map( val => (val || 1) ),
                 debounceTime(300) ),
      pageLength$.pipe( map( _ => 1 ) ) )
   ).pipe( startWith(1) );
```


20
40
60
80
100
1
1
merge
20
40
60
1
80
100
1

```
const tableSliced$ = combineLatest(itemsPerPage$, pageNumber$)
      .pipe(  withLatestFrom( tableFiltered$ ),
              map( ([[itemsPerPage, pageNumber], tableFiltered]) =>
                 tableFiltered.slice(
                 itemsPerPage * (pageNumber - 1),
                 itemsPerPage * pageNumber ) )
            );
```

Observableの発火時の処理を登録するメソッド

```
tableSliced$.subscribe( tableSliced =>
   setTableData( document.getElementById('my-data-table'), tableSliced ) );

tableFiltered$.subscribe( tableFiltered =>
   document.getElementById('nof-items').innerText = tableFiltered.length);

pageLength$.subscribe( val =>
   document.getElementById('page-length').innerText = (val || 1) );

itemsPerPage$.subscribe( val =>
   document.getElementById('items-per-page').valueAsNumber = (val || 50) );

pageNumber$.subscribe( val =>
   document.getElementById('page-number').valueAsNumber = (val || 1) );
```

# Observableの優れた点

- 「自身を変更する要因」の組み合わせで定義するので
  - どんな値になりうるかはその定義部分だけ見ればわかる
  - 定義部分以外の箇所で値が書き換わる心配が全く無い
  - 「変数Aの値を更新する処理」と「変数Aの値の変更時の処理」を分けて書ける（ことになる）

- JavaScript内の変数の値を画面に反映し忘れる心配が少ない
  - 表示処理をObservable発火時の処理として登録するため

- 「親→子」の依存関係だけ書いていけばよい
  ソースコード内の相互依存性が低い

# RxJS導入方法

1. index.html に1行追加して、

```
<script src="https://unpkg.com/rxjs/bundles/rxjs.umd.min.js"></script>
```

2. JavaScriptのソースコード内でimportするだけ

```
const { Observable, fromEvent, combineLatest, merge } = rxjs;

const { map, startWith, debounceTime, withLatestFrom } = rxjs.operators;
```

# 発展

- Angularを使うとさらにJavaScriptソースコードをすっきりさせられる

```
tableSliced$.subscribe( tableSliced =>
   setTableData( document.getElementById('my-data-table'), tableSliced ) );
```

という部分は、テンプレート(html)内で “async pipe” を使って書ける

```
<table id="my-data-table">
   <tr *ngFor="let line of (tableSliced$ | async)">
      ...
   </tr>
</table>
```

→ JavaScript部分はロジックに集中できる

# 参考リンク

- http://rxmarbles.com
- https://github.com/ReactiveX/rxjs