お前自分ちのページャUIが本当に速いと思ってんの？

@mizchi

自己紹介

- id: mizchi
  - フロントエンドエンジニア
  - 片手間じゃないJavaScript書く人

技術

- シングルページアプリケーション(SPA) や JavaScriptでエディタとか作るのが仕事

- デザイナではないのでこのスライドが見づらくても文句言うな

- 俺は見づらいと思った

- **<u>今日話すこと</u>**
  - 片手間じゃないリストビュー設計
  - リストビューとデータバインディング
  - リストビューとインフィニットスクロール
- **<u>今日話さないこと</u>**
  - サーバー側のチューニングあれこれ
  - なんかREST叩いてJSON返ってくればいいよ(雑)

ページャの種類

- 進む/戻る
- リストビュー
- インフィニットスクロール

ページャの種類

- 進む/戻る
- リストビュー
- インフィニットスクロール

発展形

発展形

1. 進む/戻る

・こんなページがあったとする

タイトル

本文

・ ページャですね

タイトル

本文

ボタン押す

ん？

お

お、お

お、おう

タイトル

おう…

タイトル

本文

…

タイトル

本文

ほぼ画面全体作り直してる…

最小ステップを考える

タイトル

ここと

本文

ここだけ差し替えればいいはず？

- 本文記事を取得するだけのAPI叩く
  - GET /api/entries/42 みたいな
- クライアントサイドテンプレーティング
  - イニシャルページと同じテンプレートで再展開
  - pushStateでURL書き換え

適当に作る(擬似コード)

```
var template = ‘<div>{{content}}</div>’;
var n = 42;
$.get(‘/api/entries/’ + n.toString())
    .then(function(data){
      $(‘.title’).text(data.title);
      $(‘.content’).html(Mustache.render(template, {
         content: data.content
      });
});
```

適当に作る(擬似コード)

```
var template = ‘<div>{{content}}</div>’;
var n = 42;
$.get(‘/api/entries/’ + n.toString())
    .then(function(data){
      $(‘.title’).text(data.title);
      $(‘.content’).html(Mustache.render(template, {
         content: data.content
      });
});
```

ここがどんどん辛くなる！！

- イベントコールバックの再付与
- エスケープ処理
- 要件膨らむことでかさんでいく前処理
- etc…

- 人類が管理するには辛い領域
  - でもjQuery得意な人は頑張っちゃう
  - 本人以外読めないjQuery書くのは人類に有害
- どうやって解決する？
  - データバインドしようぜ！

データバインド

- **データバインド**
- データとビューの同期
  - MVVM
- ビューモデルが常にアトミックな編集点
- Angular/Knockout/Backbone.stickit/epoxy.js

ビューモデル

```
{
  title:      'I am title',
  content: 'I am content'
}
```

ビューモデル

```
{
  title:      'I am title',
  content: 'I am content'
}
```

出力

```
<div class='container'>
  <h2>I am title</h2>
  <p>I am content</p>
</div>
```

ビューモデル

```
{
  title:      'I am title',
  content: 'I am content'
}
```

テンプレート

```
<div class='container'>
  <h2>{{ title }}</h2>
  <p> {{ content }}</p>
</div>
```

監視

差分の伝達

Mediator

ビューモデル

```
{
  title:      'I am title',
  content: 'I am content'
}
```

テンプレート

```
<div class='container'>
  <h2>{{ title }}</h2>
  <p> {{ content }}</p>
</div>
```

監視

差分の伝達

Mediator

意味論的に意識するのは
ここだけ！！！

- **テンプレートエンジンとデータバインドの違い**

  - テンプレートエンジン

    - 生成するのは**HTMLの文字列**

    - 一度書いたら終わり

  - データバインド

    - 管理するのは**DOMそのもの**

    - イベントのコールバックも管理する

    - DOM生成後もビューモデルと同期し続ける

DOM ≠ HTML

- 文字列の挿入 => ブラウザが認識 => DOM へ変換
- 画面のちらつき = ブラウザの認識時間
- サーバサイドの人もこれだけは覚えて帰って

- **テンプレートエンジンとデータバインドの違い**

  - テンプレートエンジン

    - 生成するのは**HTMLの文字列**

    - 一度書いたら終わり

  - データバインド

    - 管理するのは**DOMそのもの**

    - イベントのコールバックも管理する

    - DOM生成後もビューモデルと同期し続ける

データバインドの目的

- 前提: **生DOMを操作するのは難しい！**
- 人間のためのモデルで抽象したい
- 人間は馬鹿なので差分管理できない
- ライブラリが差分をレンダリングすればよい！

データバインドで達成できること

- モデル操作だけでDOMの状態をコントロールする
- プログラマはモデルの意味論だけ考えてコードを書けるようになる
  - entryViewModel.title = ‘This is next title’;

タイトル

本文

ボタン押す

次のタイトル

次の本文

頑張って高速化したぞ！

- 高速化だけだと「速すぎて気持ち悪い」になる可能性
    - 画面の変化がなさすぎで「遷移」のアフォーダンスがない
        - ユーザーが気づかない可能性
- じゃあどうする？
- **「UX」の話をしていいのは、ここから先！！！**

- 「速さ」は最高のユーザー体験
- API/ビューのコンポーネントが綺麗に実装されてないと、その先の演出に制約がかかる
- 「できた」「めっちゃバグりますねこれ」「修正しました」「今回は怖いので次に」永遠にリリースされない

という話を踏まえて

2. リストビュー

の話をします

こういうデータがあったら

```
[
  {val: 0},

  {val: 1},
{val: 2},
        ]
```

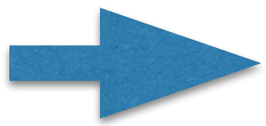
こういう出力がほしい

```
[
   {val: 0},

   {val: 1},

        {val: 2}

              ]
```



```
<ul>
   <li>0</li>
   <li>1</li>
   <li>2</li>
</ul>
```

もちろんデータの変更に応じてビューが生成されてほしい

雑な実装

```
var data = [{val: 0}, {val: 1} {val: 2}];
var $ul = $('<ul/>');
data.forEach(function(item){
  $ul.append('<li>' + escape(item.val) + '</li>');
});
```

雑な実装

```
var data = [{val: 0}, {val: 1} {val: 2}];
var $ul = $('<ul/>');
data.forEach(function(item){
  $ul.append('<li>' + escape(item.val) + '</li>');
});
```

これで本当にいいのか？

よくあるユースケース

- ソートしたい
- フィルタしたい
- 末尾までスクロールしたらインフィニットスクロールしたい

よくあるユースケース

- ソートしたい
- フィルタしたい
- 末尾までスクロールしたらインフィニットスクロールしたい

アッ

リストビューは難しい

- たとえばjQuery#appendし続ける場合
  - 状態を復元するためのアトミック性がない
  - ロールバックとか無理
    - エンバグしたときに理由がわからないことが多い
- 常に初期化から始めるアトミックな処理の場合
  - DOMツリーを構築し直すことになるので目に見えて遅い
  - CSS再適用で画面のちらつきが激しい

リストビューも
データバインドしようぜ！

データ

```
[
  {val: 0},
  {val: 1},
  {val: 2},
          ]
```

データ

[

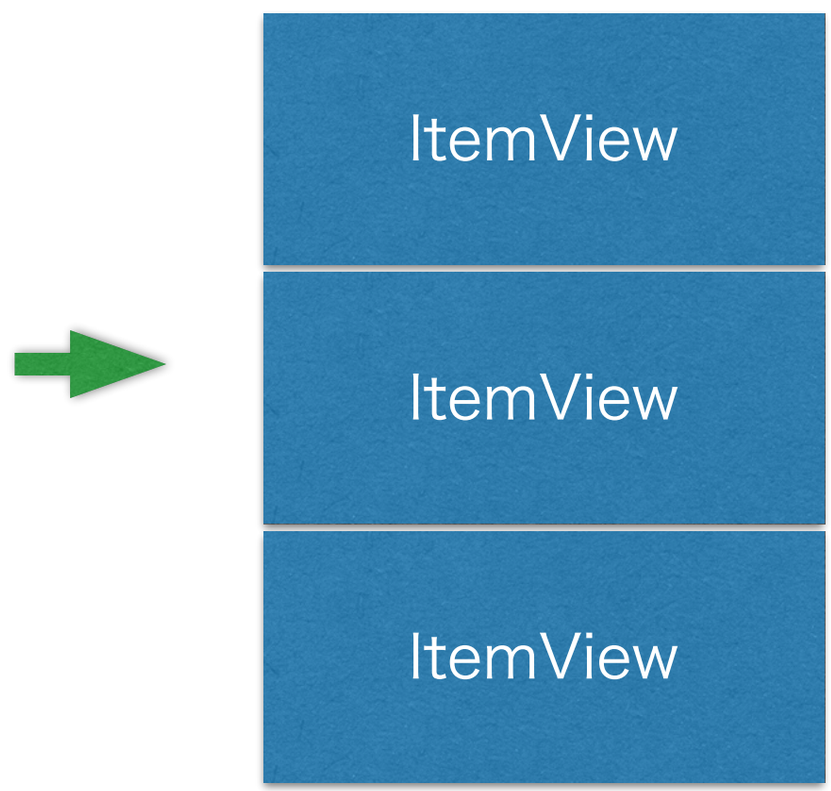
{val: 0},

{val: 1},

{val: 2},

]

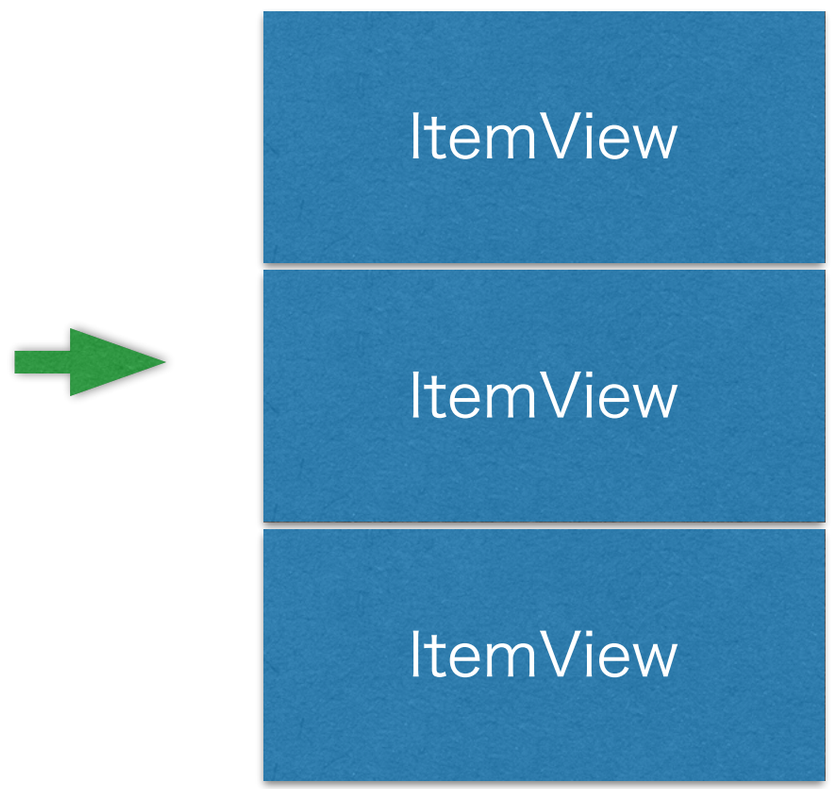
ListView



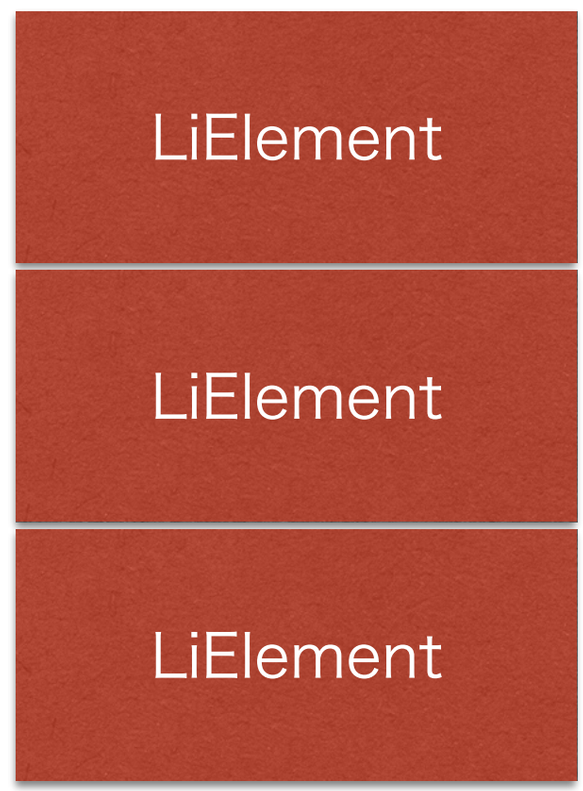
データ

```
[
  {val: 0},
  {val: 1},
  {val: 2},
]
```

ListView



DOM



データ

ListView

DOM

データを修正すると

```
[
  {val: 0},
  {val: 1},
  {val: 2},
        ]
```

ItemView

ItemView

ItemView

LiElement

LiElement

LiElement

データ

[

{val: 0},

{val: 1},

{val: 2},

{val: 3},

]

追加

ListView

ItemView

ItemView

ItemView

DOM

LiElement

LiElement

LiElement

データ

[

{val: 0},

{val: 1},

{val: 2},

{val: 3},

]

ListView

ItemView

ItemView

ItemView

DOM

LiElement

LiElement

データの修正を検知

データ

[

{val: 0},

{val: 1},

{val: 2},

{val: 3},

]

ListView

DOM

DOMに伝搬

ItemView

ItemView

ItemView

LiElement

LiElement

LiElement

リストビューのデータバインドで達成したいこと

- 変更差分「だけ」をレンダリングさせる
  - 内部的にはDOMのヒープを確保したり捨てたりする
  - 「DOMを破棄せずに」ソートが可能になる
  - 「DOMを破棄せずに」フィルタが可能になる
- それらの操作が見た目上はモデル操作で完結する

フレームワーク毎に
いろんなアプローチがある

Backbone拡張系

Chaplin/Marionette

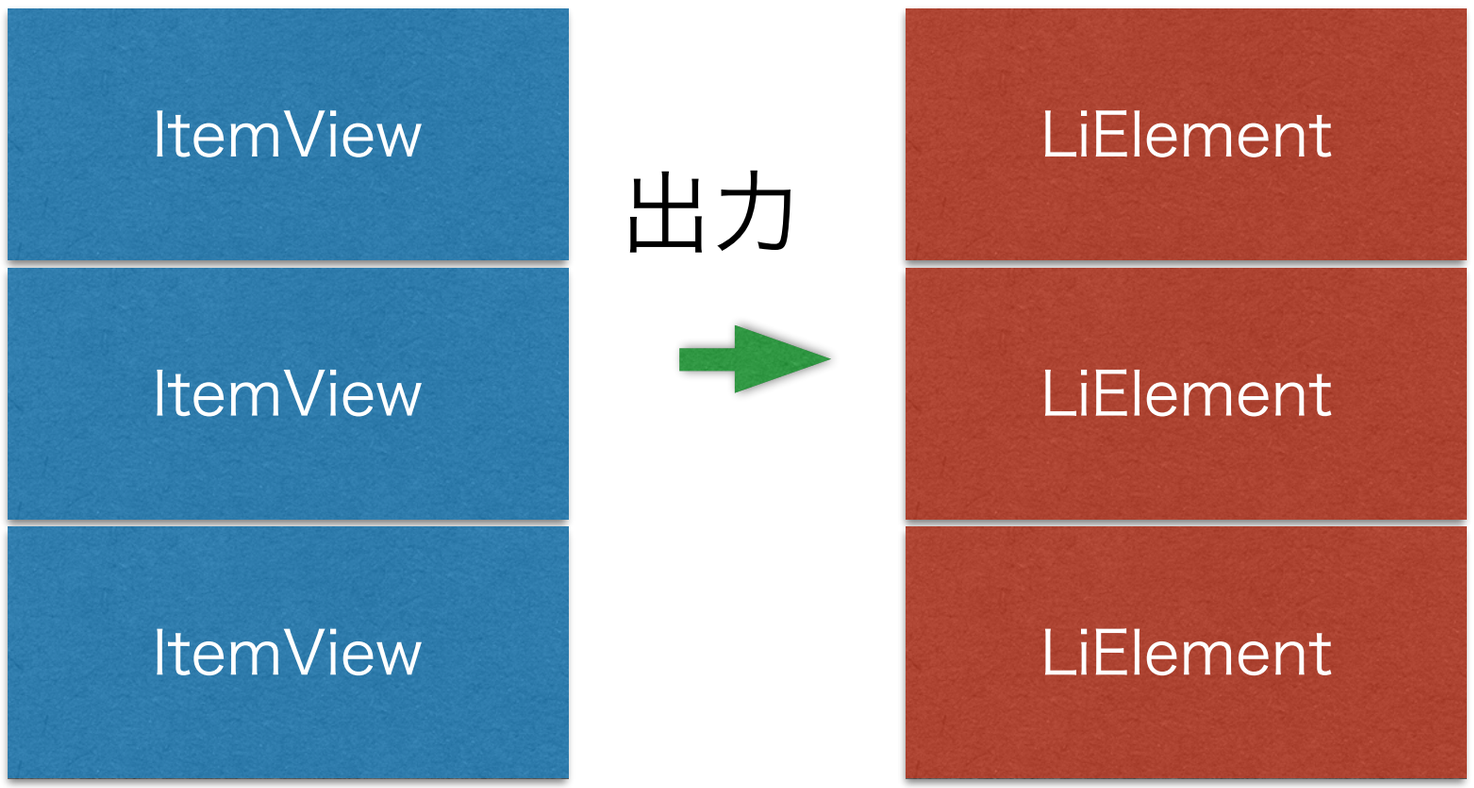
Chaplin/Marionette

Backbone.Collection CollectionView DOM

```
new Collection(
    [
      {val: 0},
      {val: 1},
      {val: 2},
    ] )
```

出力

出力



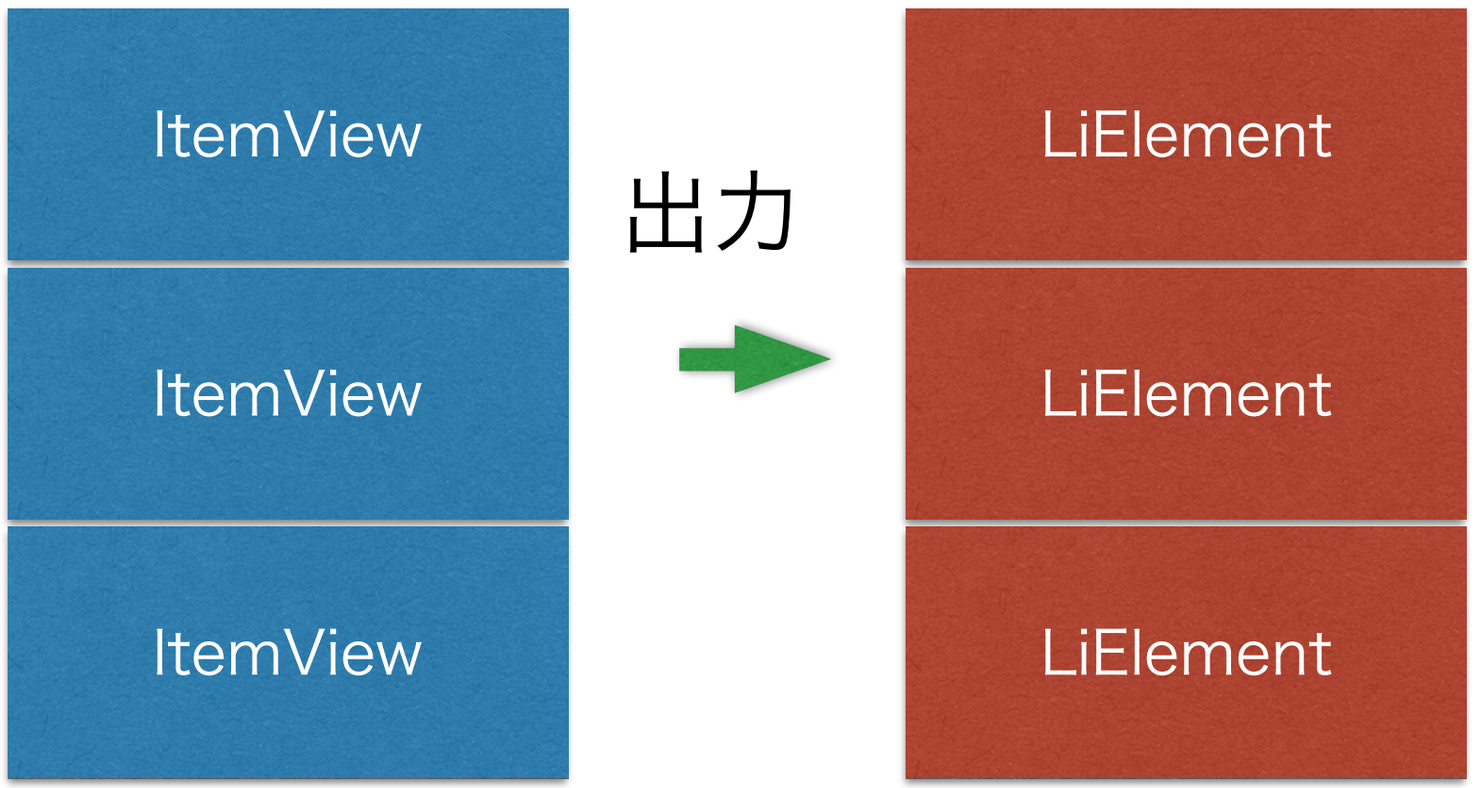
ユーザーが操作するのは専用のCollection

Chaplin/Marionette

Backbone.Collection　CollectionView　　DOM

```
new Collection(
    [
      {val: 0},
      {val: 1},
      {val: 2},
    ]  )
```

出力



CollectionViewのインスタンスはCollectionに従属

Chaplin/Marionette方式

メリット

- Collectionへの操作がアトミック
  - 要素/要素数を変更するだけでDOMが出力される
- データフローが常に一方向
  - Collection -> CollectionView -> DOM

Chaplin/Marionette方式

デメリット

- 専用のCollectionのAPIを覚える必要
- CollectionとItemViewの内部構造の対応が複雑
  - どこでchange/reset発火するか?
- リストビューが入れ子になったときに複雑 x 複雑で人間の管理能力を超える感
  - ポリモーフィックなItemViewめっちゃ辛かった

mizchi/horn.js

- データバインドを行うビューライブラリ
- リストビューが簡単に書けるように特化

```
# ListView
list = new StatusList
list.size(2) # generate automatically 2 blank view.
list.update [{name: 1},{name: 2},{name: 3}] # genera
list.addItem {name: 4} # add more
list.attach 'body'
```

仕組み

```
size: (n) ->
  return @views.length unless n?

  if @views.length > n
    for i in [1..@views.length-n]
      @views.pop().remove()

  else if @views.length < n
    for i in [1..n-@views.length]
      @addItem()
```

減ってたらView削除

足りなかったらView生成

mizchi/horn.jsの今後

- vue.jsがあるので使わなくていい
- vue.js最高

Vue/Angular方式

Vue/Angular

データ　　Vue/Angular　　DOM

[

{val: 0},

{val: 1},

{val: 2},

]

監視

出力

Mediator

LiElement

LiElement

LiElement

ユーザーが操作するのはJavaScriptのデータ

Vue/Angular

- メリット
  - 生のデータ構造を扱うので**APIが直感的**
  - 特定の構造体を要求されないので採用コストが軽い

Vue/Angular

- **デメリット**
- どうあがいても実装が汚い
  - Vue.jsはオブジェクトをgetter/setterまみれにする
  - Angular は裏のループで差分を監視 (Dirty Check)
- 行儀よく実装するにはES7 の Object.observe/DynamicProxyを待つ必要がある

ES7 Object.observe/DynamicProxy

- **Object.observe**
  - プリミティブな値を外から監視する仕組み
  - Chrome 35から有効化
- **DynamicProxy**
  - オブジェクトに対する操作をラップできる
- どちらもPolyfillの実装が難しい

注意: どちらもライブラリ実装者向けの機能

facebook/react

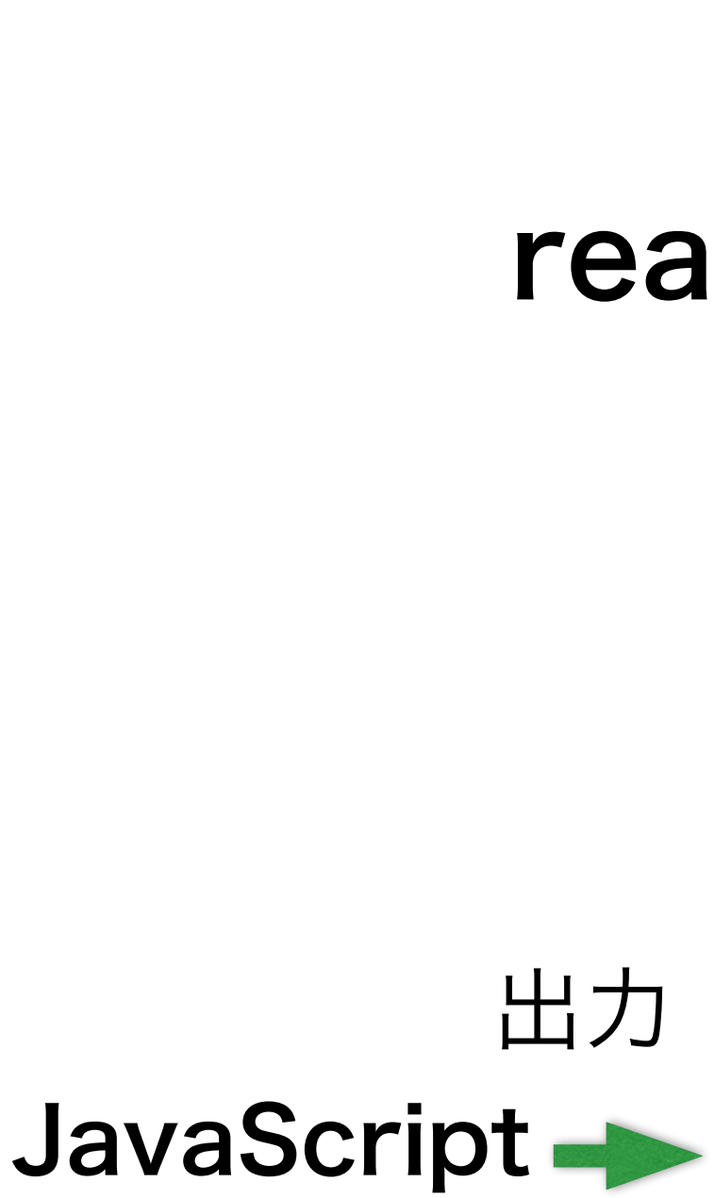
react

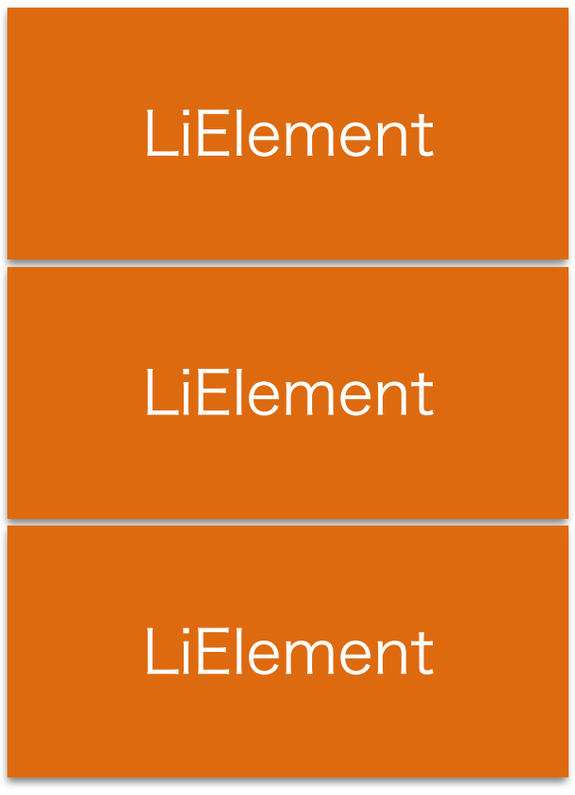
Virtual DOM

DOM

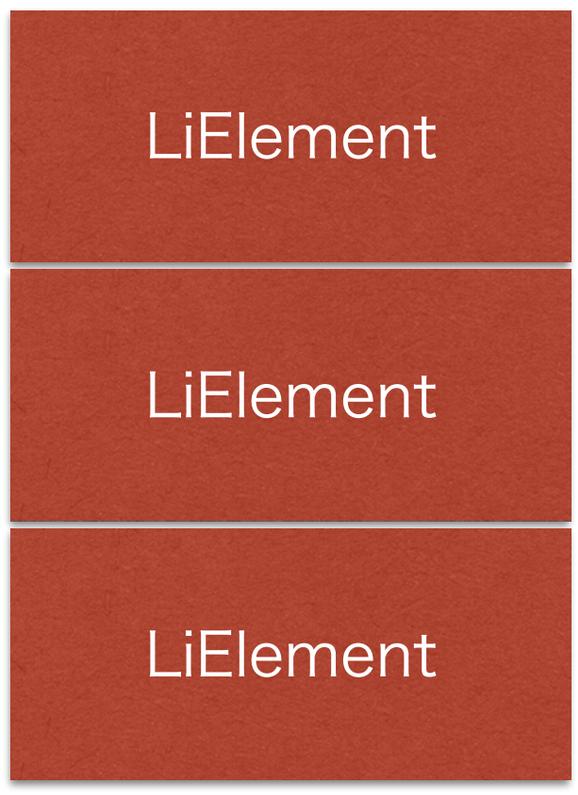
出力

JavaScript





差分出力



facebook/react

- (今まで話したモデル抽象ではない点に注意)
- 仮想のDOMと実際のDOMを分離
- ユーザーは常にVirtualDOMを操作する
- reactはVirtualDOMと実際のDOMの差分を生成する

facebook/react

Reactのメリット

- 今までと同じようなDOMを出力するアプローチに対して、裏で最適化がかかる
- 嬉しい…

facebook/react

Reactの辛い点

- 専用のjsx構文を要求される(altjsに厳しい)
- jsxを使わない場合はVirtualDOMインスタンスを直接触る必要がある
- 生DOM触ると仮想DOMと状態がズレる
- 生DOM触るライブラリと合わせると何が起こる？？？

3. インフィニットスクロール

インフィニットスクロールついての知見

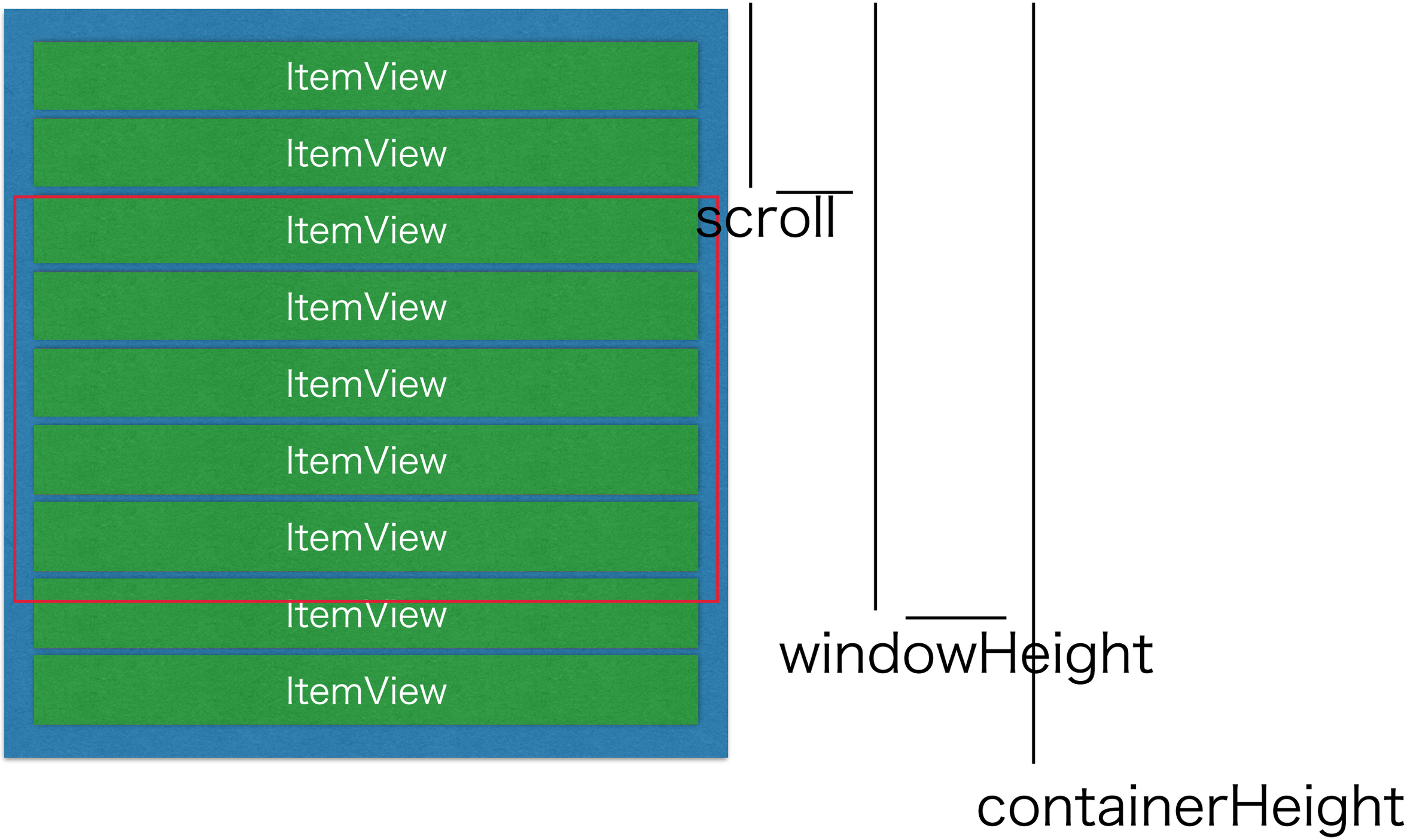
- **毎回要件がブレる**
  - body直下？コンテナーの中？
- **末尾の定義がブレる**
  - リストの最後の要素の頭
  - リストの最後の要素の末尾
  - 余裕持って「末尾の400pxぐらい先に」とか
- **ほとんどのライブラリが特定要件でしか使えない**
  - (自分は)自前で作ったほうが速い

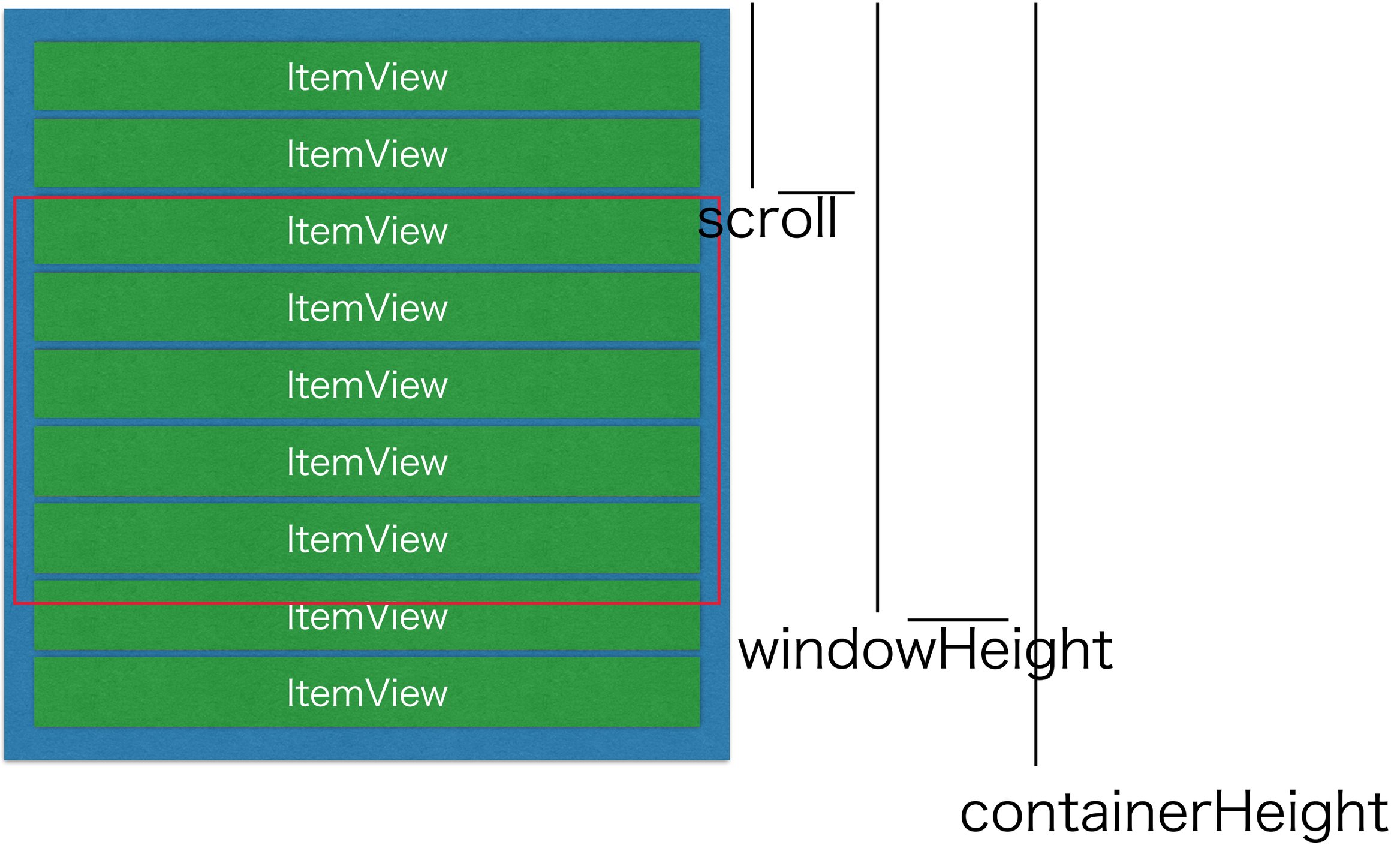
ワーストケース

- インフィニットスクロール発火の度にリストを再生成
  - DOMが破棄されるのでコンテナが一旦縮む
  - DOMを破棄する前後でスクロール位置を保存して描画後に復元するなどして対処
- ユーザーとしては画面がガコッガコッとズレまくるので最悪なユーザー体験
- しかもスクロールするほど要素が増えて遅くなる


ItemView
ItemView
ItemView
ItemView
ItemView
ItemView
ItemView
ItemView
ItemView

赤線: 画面


ItemView
ItemView
ItemView
ItemView
ItemView
ItemView
ItemView
ItemView
ItemView
scroll
windowHeight
containerHeight



発火条件

containerHeight - 任意の先読み幅 < scroll + windowHeight
(100~300px)

擬似コード

```
var lock = false;
var cursor = 0;
$container.on('scroll', function(){
  if(lock) return; //多重読み込みを防止
  if(発火条件){
    lock = true; // ロックをとる
    $.get('/items', {cursor: cursor++})
      .then(function(data)){
        //ビューモデルを操作してビューを追加
        listView.items = listView.items.concat(data.items);
        lock = false; ロック解除
});};})
```

インフィニットスクロールはつくれる！

- 土台作っとかないと複雑度が爆発する
- ロックを取らないと無限に発火する
- 自分で作った方が楽　※個人の感想です

インフィニットスクロールの発展形

インフィニットスクロールの発展形

上下左右の先読み予測

- 上下左右のデータ構造
  - バインディング作らないと工数爆発
- 2次元のデータプリフェッチ
- 最高に辛い

昔これに似たページャ作った

二次元のクオータビュー

- 無限に伸びるので二次元のリスト構造
- マスターデータ(固定) + 周辺リージョンのソーシャルデータ先読み
- 1セルでdivが6層 + 25セル * 25セル = 37500div
- 最初は画面遷移のたびに4s
- 裏でdivのヒープ作って再利用で初期化4s+遷移40ms
- 回転行列は決め打ち

最高のページャにしましょう(完)