REACTJS*, JAVASCRIPT*

Flux для глупых людей

ПЕРЕВОД

ElianL 30 января 2015 в 13:09 170k

Оригинал: Andrew Ray

Пытаясь разобраться с библиотекой от Facebook ReactJS и продвигаемой той же компанией архитектурой «Flux», наткнулся на просторах интернета на две занимательные статьи: «ReactJS For Stupid People» и «Flux For Stupid People». Чуть раньше я поделился с хабравчанами переводом первой статьи, настала очередь второй. Итак, поехали.

Flux для глупых людей

TL;DR Мне, как глупому человеку, как раз не хватало этой статьи, когда я пытался разобраться с Flux. Это было не просто: хорошей документации нет и многие ее части перемещаются.

Это продолжение статьи «ReactJS For Stupid People».

Должен ли я использовать Flux?

Если ваше приложение работает с динамическими данными, тогда, вероятно, вы должны использовать Flux.

Если ваше приложение просто набор статичных представлений и

вы не сохраняете и не обновляете данные, тогда нет. Flux не даст вам какой-либо выгоды.

Почему Flux?

Юмор в том, что Flux — это не самая простая идея. Так зачем же все усложнять?

90% iOS приложений — это данные в табличном виде. Инструменты для iOS имеют четко определенные представления и модель данных, которые упрощают разработку приложений.

Для frontend'a (HTML, JavaScript, CSS) у нас такого нет. Вместо этого у нас есть большая проблема: никто не знает, как структурировать frontend приложение. Я работал в этой сфере в течении многих лет и «лучшие практики" никогда нас этому не учили. Вместо этого нас „учили" библиотеки. jQuery? Angular? Backbone? Настоящая проблема — поток данных — до сих пор ускользает от нас.

Что такое Flux?

Flux — это термин, придуманный для обозначения однонаправленного потока данных с очень специфичными событиями и слушателями. Нет Flux библиотек *(прим. перев.: на данный момент их полно)*, но вам будет нужен Flux Dispatcher и любая JavaScript event-библиотека.

Официальная документация написана как чей-то поток сознания и

является плохой отправной точкой. Но когда вы уложите Flux в свою голову, она может помочь заполнить некоторые пробелы.

Не пытайтесь сравнивать Flux с MVC-архитектурой. Проведение параллелей только еще больше запутает вас.

Давайте нырнем поглубже! Я буду по порядку объяснять все концепции и сводить их к одной.

1. Ваши представления отправляют события

Dispatcher по своей сути является event-системой. Он траслирует события и регистрирует колбэки. Есть только один глобальный dispatcher. Вы можете использоватьdispatcher от Facebook. Он очень легко инициализируется:

```
var AppDispatcher = new Dispatcher();
```

Скажем, в вашем приложении есть кнопка “New Item”, которая добавляет новый элемент в список.

```
<button onClick={ this.createNewItem }>New Item</button>
```

Что происходит при клике? Ваше представление отправляет специальное событие, которое содержит в себе название события и данные нового элемента:

```
createNewItem: function( evt ) {
```

```
    AppDispatcher.dispatch({
        eventName: 'new-item',
        newItem: { name: 'Marco' } // example data
    });

}
```

2. Ваше хранилище(store) реагирует на отправленные события

Как и „Flux“, „**Store**“ — это просто термин, придуманный Facebook. Для нашего приложения нам необходимы некоторый набор логики и данные для списка. Это и есть наше хранилище. Назовем его *ListStore*.

Хранилище — это синглтон, а это значит, что вам можно не объявлять его через оператор *new*.

```
// Global object representing list data and logic
var ListStore = {

   // Actual collection of model data
    items: [],

   // Accessor method we'll use later
    getAll: function() {
        return this.items;
    }

}
```

Ваше хранилище будет реагировать на посланное событие:

```
var ListStore = …

AppDispatcher.register( function( payload ) {
```

```
    switch( payload.eventName ) {

        case 'new-item':

           // We get to mutate data!
            ListStore.items.push( payload.newItem );
            break;

    }

    return true; // Needed for Flux promise resolution

});
```

Это традиционный подход к тому, как Flux вызвает колбэки. Объект *payload* содержит в себе название события и данные. А оператор *switch* решает какое действие выполнить.

Ключевая концепция: *Хранилище — это не модель. Хранилище содержит модели*.

Ключевая концепция: *Хранилище — единственная сущность в вашем приложении, которая знает как изменить данные.* ***Это самая важная часть Flux.*** *Событие, которые мы послали, не знает как добавить или удалить элемент*

.

Если, например, разным частям вашего приложения нужно хранить путь до некоторых картинок и другие метаданные, вы создаете другое хранилище и называете его ImageStore. Хранилище представляет собой отдельный „домен“ вашего приложения. Если ваше приложение большое, домены, возможно, будут для вас очевидны. Если приложение маленькое, то,

возможно, вам хватит и одного хранилища.

Только хранилища регистрируют колбеки в dispatcher. Ваши представления никогда не должны вызвать AppDispatcher.register. Dispatcher только для отправки сообщений из представлений в хранилища. Ваши представления будут реагировать на другой вид событий.

3. Ваше хранилище посылает событие „Change“

Мы почти закончили. Сейчас наши данные точно меняются, осталось рассказать об этом миру.

Ваше хранилище посылает событие, но не использует dispatcher. Это может сбить с толку, но это „Flux way“. Давайте дадим нашему хранилищу возможность инициировать событие. Если вы используете MicroEvents.js, то это просто:

```
MicroEvent.mixin( ListStore );
```

Теперь инициализируем наше событие „change“:

```
    case 'new-item':

            ListStore.items.push( payload.newItem );

           // Tell the world we changed!
            ListStore.trigger( 'change' );

            break;
```

Ключевая концепция: *Мы не передаем данные вместе с событием. Наше представление беспокоиться только о том, что что-то изменилось.*

4. Ваше представление реагирует на событие „change“

Сейчас мы должны отобразить список. Наше представление **полностью перерисуется**. когда список изменится. Это не опечатка.

Во-первых, давайте подпишемся на событие „change“ из нашего ListStore сразу после создания компонента:

```
 componentDidMount: function() {
    ListStore.bind( 'change', this.listChanged );
}
```

Для простоты мы просто вызовем forceUpdate, который вызовет перерисовку:

```
listChanged: function() {
   // Since the list changed, trigger a new render.
    this.forceUpdate();
},
```

Не забываем удалять слушателя, когда компонент удаляется:

```
componentWillUnmount: function() {
    ListStore.unbind( 'change', this.listChanged );
},
```

Что теперь? Давайте посмотрим на нашу функцию render, которую я намерено оставил напоследок:

```
render: function() {

   // Remember, ListStore is global!
   // There's no need to pass it around
    var items = ListStore.getAll();

   // Build list items markup by looping
   // over the entire list
    var itemHtml = items.map( function( listItem ) {

       // "key" is important, should be a unique
       // identifier for each list item
        return <li key={ listItem.id }>
            { listItem.name }
          </li>;

    });

    return <div>
        <ul>
            { itemHtml }
        </ul>

        <button onClick={ this.createNewItem }>New Item</button>
    </div>;
}
```

Мы пришли к полному циклу. Когда вы добавляете новый элемент, представление отправляет событие, хранилище подписано на это событие, хранилище изменяется, хранилище создает событие „change“ и представление, подписанное на событие „change“, перерисовывается.

Но тут есть одна проблема. Мы полностью перерисовываем

представление каждый раз, когда список изменяется! Разве это не ужасно неэффективно?

Конечно, мы вызываем функцию render снова и, конечно, весь код в этой функции выполняется. Но R**eact изменяет реальный DOM, если только результат вызова render будет отличатся от предыдущего.** Ваша функция render, на самом деле, генерирует „виртуальный DOM“, который React сравнивает с предыдущим результатом вызова функции render. Если два виртуальных DOMa различаются, React изменит реальный DOM — и только в нужных местах.

Ключевая концепция: *Когда хранилище изменяется, ваши представления не должны заботиться том, какое событие произошло: добавление, удаление или изменение. Они должны просто полностью перерисоваться. Алгоритм сравнения „вирутального DOM“ справится с тяжелыми расчетами и изменит реальный DOM. Это сделает вашу жизнь проще и уменьшит головную боль.*

И еще: что вообще такое „Action Creator“?
Помните, когда мы нажимали нашу кнопку, мы отправляли специальное событие:

```
AppDispatcher.dispatch({
    eventName: 'new-item',
    newItem: { name: 'Samantha' }
});
```

Это может привести к часто повторяющемуся коду, если много

ваших представлений использует это событие. Плюс, все представления должны знать о формате. Это неправильно. Flux предлагает абстракцию, названную **action creators**, которая просто абстрагирует код выше в функцию.

```
ListActions = {

    add: function( item ) {
        AppDispatcher.dispatch({
            eventName: 'new-item',
            newItem: item
        });
    }

};
```

Теперь, ваше представление просто вызывает *ListAction.add({name: “...”})* и не переживает о синтаксисе отправки сообщений.

Оставшиеся вопросы

Все, о чем говорит нам Flux, это как управлять потоком данных. Но он не отвечает на вопросы:

- Как вам загружать данные на сервер и как их сохранять на сервере?
- Как управлять связью между компонентами с общим родителем?
- Какую event-библиотеку использовать? Имеет ли это значение?
- Почему Facebook не включил все это в свою библиотеку?
- Должен ли я использовать слой модели наподобие Backbone в качестве модели в нашем хранилище?

Ответ на все эти вопросы: развлекайтесь!

PS: Не используйте forceUpdate
Я использовал forseUpdate ради простоты. Правильное решение будет считать данные из хранилища и скопировать их в state компонента, а в функции render прочитать данные из state. Вы можете посмотреть, как это работает в этом примере.

Когда ваш компонент загружается, хранилище копирует данные в state. Когда хранилище изменяется, данные полностью переписываются. И это лучше, потому что внутри forceUpdate выполняется синхронно, а setState — более эффективный.

Вот и все!

В дополнение можете посмотреть Example Flux Applcation от Facebook.

Надеюсь, после прочтения этой статьи вам будет проще понять файловую структуру проекта.

Документация Flux содержит несколько полезных примеров, глубоко закопанных внутри.

Если этот пост помог вам понять Flux, то подписывайтесь на меня в твиттере.

Проголосовать:

+32

Поделиться:

Сохранить:

Комментарии (35)

Похожие публикации

Haskell для ВКонтакте, JavaScript и ReactJS, Или «Чужой против Симпсонов»

1

ИЗ ПЕСОЧНИЦЫ

eryx67 • 11 декабря 2015 в 12:01

Новые возможности платформы VoxImplant: Instant Messaging и Presence + демо на ReactJS/Flux

2

aylarov • 15 мая 2015 в 14:47

ReactJS для глупых людей

17

ИЗ ПЕСОЧНИЦЫ

ElianL • 28 января 2015 в 16:33

Популярное за сутки

Яндекс открывает Алису для всех разработчиков. Платформа Яндекс.Диалоги (бета) 69

BarakAdama • вчера в 10:52

Почему следует игнорировать истории основателей успешных стартапов 20

ПЕРЕВОД

m1rko • вчера в 10:44

Как получить телефон (почти) любой красотки в Москве, или интересная особенность MT_FREE 24

ИЗ ПЕСОЧНИЦЫ

cab404 • вчера в 20:27

Java и Project Reactor 10

zealot_and_frenzy • вчера в 10:56

Пользовательские агрегатные и оконные функции в PostgreSQL и Oracle 6

erogov • вчера в 12:46

Лучшее на Geektimes

Как фермеры Дикого Запада организовали телефонную сеть на колючей проволоке

31

NAGru • вчера в 10:10

Энтузиаст сделал новую материнскую плату для ThinkPad X200s

49

alizar • вчера в 15:32

Кто-то посылает секс-игрушки с Amazon незнакомцам. Amazon не знает, как их остановить

85

Pochtoycom • вчера в 13:06

Илон Маск продолжает убеждать в необходимости создания колонии людей на Марсе

139

marks • вчера в 14:19

Дела шпионские (часть 1)

16

TashaFridrih • вчера в 13:16