Передовые инженерные школы

МИНОБРНАУКИ РОССИИ

УНИВЕРСИТЕТ ИННОПОЛИС

Занятие 21

Управление состоянием
Flux архитектура
Redux | Action | ActionCreators

План занятия

ПЕРЕДОВАЯ
ИНЖЕНЕРНАЯ ШКОЛА
УНИВЕРСИТЕТА ИННОПОЛИС

1. Состояния и управление ими
2. Flux архитектура
3. Redux
4. Actions
5. ActionsCreators

Состояние

Современные приложения

В современных приложениях между клиентом и сервером общение строится именно на данных, а не на отрендеренных кусках разметки. Чаще всего для такого общения выбирают JSON.

работает так:

1. Клиент делает первичный запрос на сервер.
2. Сервер отвечает HTML-страницей, иногда с набором каких-то данных внедрённых в виде JS-объекта в конце страницы.
    1. Страница может быть либо отрисована на сервере и тогда клиенту придёт готовый HTML.
    2. Либо отрисовкой будет заниматься сам клиент с помощью какой-нибудь библиотеки, например, React. В этом случае от сервера приходит просто набор необходимых данных.
3. Пользователь совершает какое-то действие, например, просит отсортировать таблицу.
4. Клиент в ответ на это действие решает, какой запрос отправить на сервер, строит этот запрос и отправляет его.
5. Сервер принимает этот запрос, обрабатывает его и отправляет на клиент порцию новых данных.
6. Клиент принимает данные и перерисовывает часть страницы по ним сам. То есть он уже не заменяет один кусок разметки другим готовым, а рисует разметку сам.

Состояние

Плюсов такого общения (когда передаются только данные) несколько:

- *Сервер и клиент становятся независимыми друг от друга*. Сервер может ничего не знать об устройстве страниц, ему достаточно лишь уметь работать с БД и обрабатывать данные (первичная отрисовка может быть сделана самим сервером с помощью SSR)
- *Количество информации, которое приходится передавать и принимать, меньше* — а это уменьшает объём трафика
- *Логика приложения на сервере может быть проще*, потому он и клиент становятся менее зависимы друг от друга в плане формата данных

Состояние

ПЕРЕДОВАЯ ИНЖЕНЕРНАЯ ШКОЛА УНИВЕРСИТЕТА ИННОПОЛИС

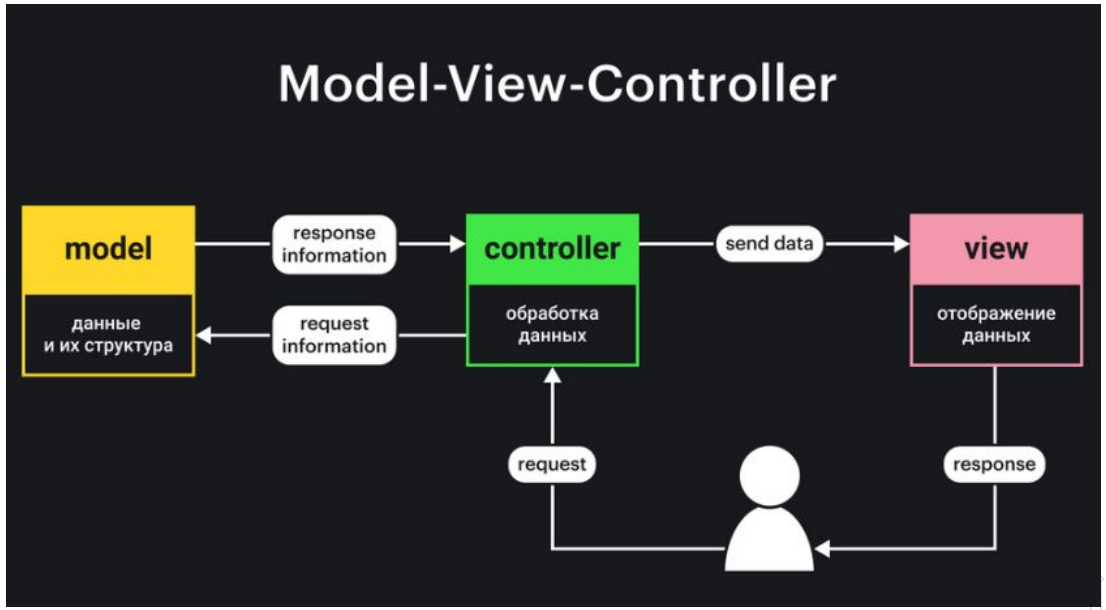
Разделение ответственности

Секция статьи "Разделение ответственности"

Такое общение между сервером и клиентом очень напоминает паттерн MVC.

MVC (Model-View-Controller) — структура приложения, в которой за данные, их обработку и их вывод отвечают три разных сущности.

- Модель (model) отвечает за данные и их структуру.
- Представление (view) — за их отображение.
- Контроллер (controller) — за их обработку.



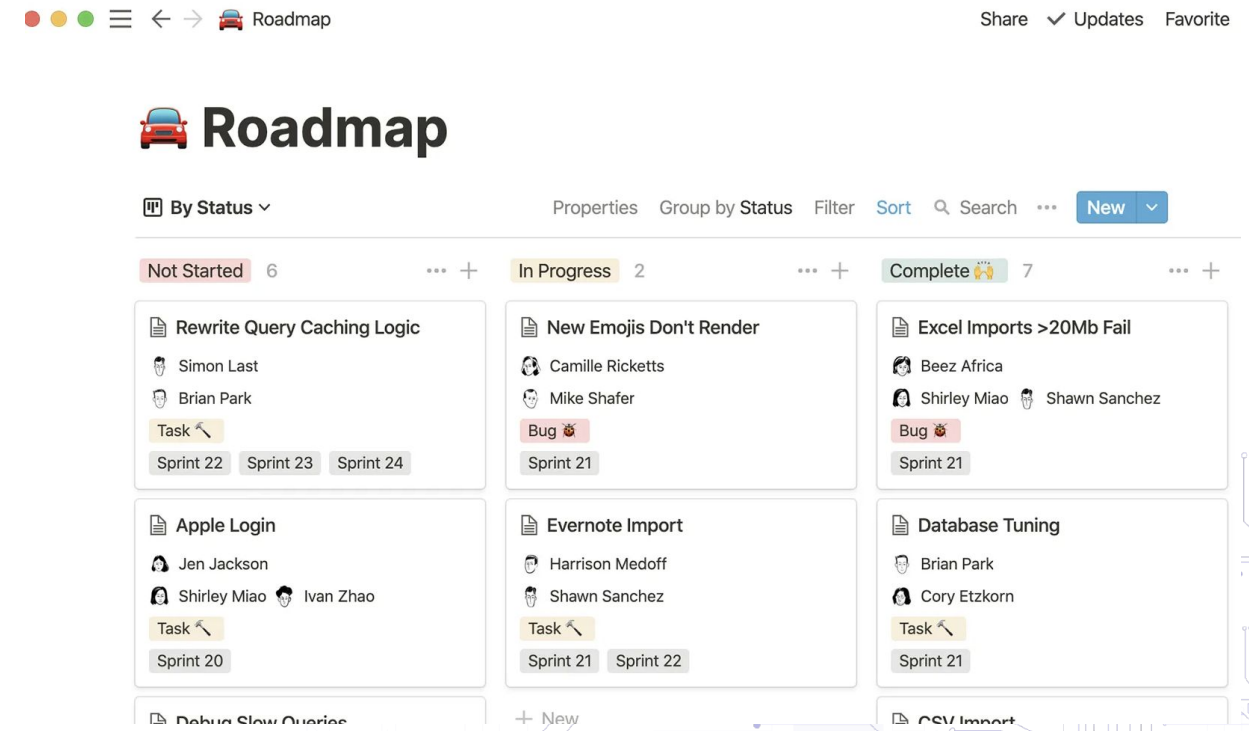
Если попробовать сравнить классическую клиент-серверную архитектуру с MVC, то можно сравнить БД с моделью, сервер с контроллером, а клиент с представлением. В самых первых приложениях, в принципе, так и было. Сейчас, однако, клиент стал сложнее, поэтому MVC может быть и часть архитектуры клиентского приложения.

Состояние

MVC на клиенте
Возьмём для примера Notion

Веб-версия работает прямо в браузере, однако это полноценный текстовый редактор.
Большая часть работы при вводе текста происходит на клиенте.

Если забыть про невозможность сохранить написанное на сервере, то попробовать написать текстовый редактор можно было и тогда.
Но, скорее всего, результатом был бы кусок разметки с инлайновыми стилями и текстом внутри.



Состояние

Состояние приложения
Текстовый редактор из примера — это приложение с состоянием.

Состояние приложения — это все данные этого приложения на текущий момент.
В случае с текстовым редактором — это весь текст, который пользователь ввёл, а также результаты преобразований над этим текстом.

Текстовые редакторы не единственный вид таких приложений.
Любое приложение, *которое хранит что-то перед отправкой (или даже без отправки) на сервер* — это приложение с состоянием.

Плюсы в выделении состояния те же:
- данные перестают зависеть от того, как их представлять;
- их проще обрабатывать и хранить;
- представление проще менять.

Как правило, состояние в приложениях на JS — это какой-то объект или массив объектов в памяти.

Состояние

ПЕРЕДОВАЯ ИНЖЕНЕРНАЯ ШКОЛА
УНИВЕРСИТЕТА ИННОПОЛИС

```
// Например, так могло бы выглядеть
// состояние текстового редактора:
const State = {
  lastModified: '2020-08-24T18:15:00',
  blocks: [
    {
      type: 'heading',
      data: {
        text: 'Какой-то заголовок',
      },
    },
    {
      type: 'paragraph',
      data: {
        text: 'Какой-то параграф текста под заголовком',
      },
    },
  ],
}
```

Состояние

Вид состояния не только удобен в использовании внутри приложения, но ещё и прост в сохранении с помощью JSON.

Приложения, в которых за состояние отвечает отдельный модуль, проще развивать и изменять.

Сейчас есть много подходов и инструментов для управления состоянием.
Из самых популярных можно назвать:

- Redux (и основанные на его подходе Vuex, NgRx)
- MobX
- Overmind

Состояние

Вид состояния не только удобен в использовании внутри приложения, но ещё и прост в сохранении с помощью JSON.

Приложения, в которых за состояние отвечает отдельный модуль, проще развивать и изменять.

Сейчас есть много подходов и инструментов для управления состоянием.
Из самых популярных можно назвать:

- Redux (и основанные на его подходе Vuex, NgRx)
- MobX
- Overmind

Состояние

JSON
Самый популярный формат данных.
Он немногословен, понятен и человеку, и компьютеру, много языков с ним уже умеют работать.

В вебе JSON, стандарт, потому что используется как формат по умолчанию во многих фреймворках.

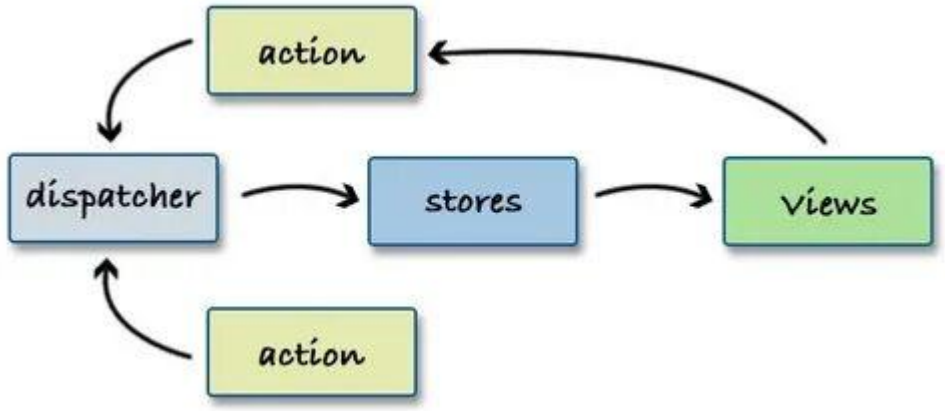
Состояние из примера выше в формате JSON выглядело бы так:

```
{
  "lastModified": "2020-08-24T18:15:00",
  "blocks": [
    {
      "type": "heading",
      "data": {
        "text": "Какой-то заголовок"
      }
    },
    {
      "type": "paragraph",
      "data": {
        "text": "Какой-то параграф текста под заголовком"
      }
    }
  ]
}
```

Flux архитектура

Flux - это в основном *концепция архитектуры приложений*, разработанная в Facebook, но тот же термин также относится и к библиотеке, представляющей официальную реализацию.

Facebook сделал Flux как попытку решить проблемы, вызванные шаблоном MVC в их массивной базе кода. Они боролись с проблемами, когда действия приводили к каскадным обновлениям, что приводило к непредсказуемым результатам и коду, который становилось трудно отлаживать.



Flux архитектура

ПЕРЕДОВАЯ ИНЖЕНЕРНАЯ ШКОЛА УНИВЕРСИТЕТА ИННОПОЛИС

Строительные блоки Flux

На высоком уровне Flux разбивается на четыре основные части: действия, диспетчер, хранилища и отображения:

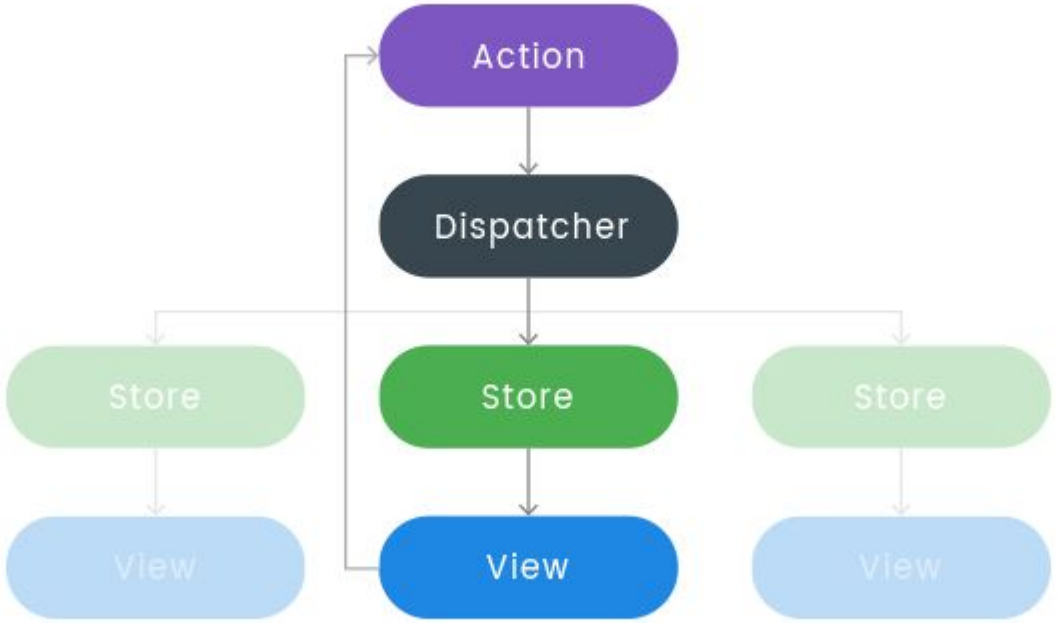
- **Действия** описывают действие, которое происходит в приложении.
- **Диспетчер** - это однопользовательский реестр обратных вызовов. Он действует как посредник, передавая действия всем хранилищам, которые подписались на него.
- **Хранилища** управляют состоянием и логикой, необходимой для его обновления для определенных частей приложения.

Представления - это простые старые компоненты React.

- В Flux все данные передаются в одном направлении:
- **Действия** передаются **диспетчеру**, удобные классы, называемые **создателями действий**.
- **Диспетчер** *отправляет* действия всем **хранилищам**, которые подписались на него.

- **хранилища** имеют отношение к конкретному действию, которое было получено, они обновляют свое состояние и сигнализируют **представления**,

Flux архитектура



Flux архитектура

Действия

Данные отправляются «по проводам» в одном направлении с использованием обычных объектов JavaScript, называемых действиями.

Их задача - описать событие, которое имело место в приложении, и переносить новые данные в хранилища.

Каждое действие должно иметь тип и необязательный ключ `payload`, содержащий данные. Действие похоже на приведенное ниже:

```
{
  actionType: "UPDATE_TITLE",
  payload: "This is a new title."
}
```

Flux архитектура

ПЕРЕДОВАЯ ИНЖЕНЕРНАЯ ШКОЛА УНИВЕРСИТЕТА ИННОПОЛИС

Тип действия должен быть представлен описательной и последовательной строкой в верхнем регистре, аналогичной общему соглашению определяющих констант.

Являются уникальными идентификаторами, которые хранилища будут использовать для определения действия и соответствующего ответа.

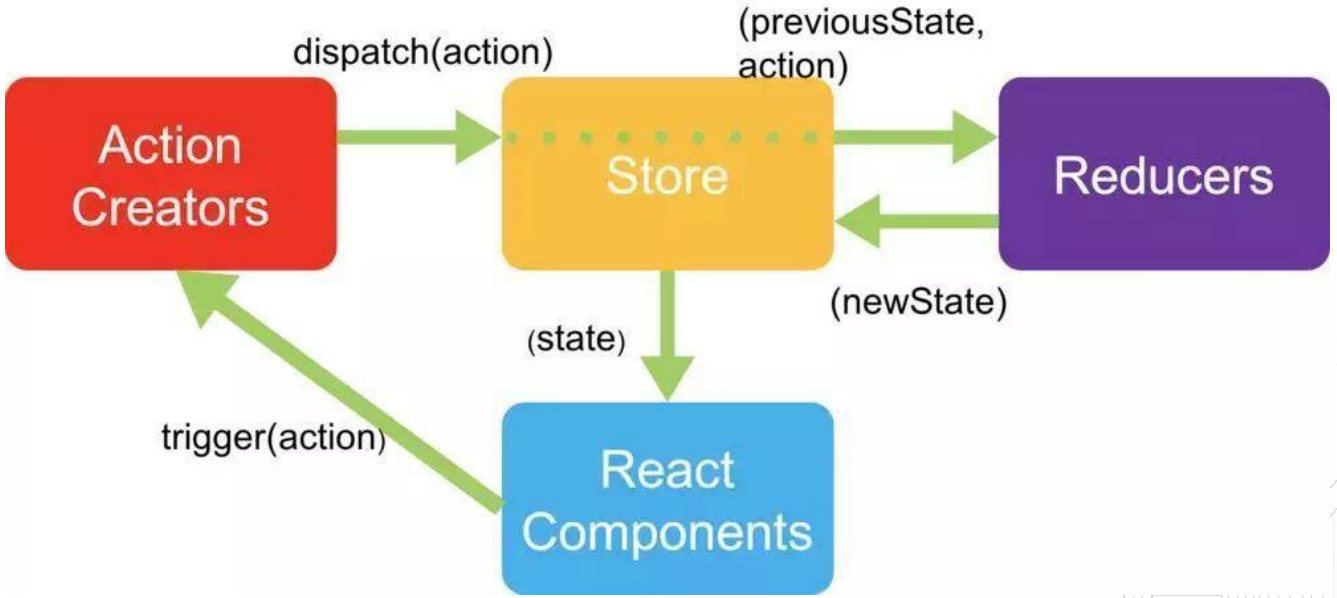
Общей практикой является определение всех типов действий в объекте констант и ссылка на этот объект вместо приложения для обеспечения согласованности.

В файле index.js используйте следующий код для создания первого типа действия:

```
export default {
    ADD_NEW_ITEM: 'ADD_NEW_ITEM'
}
```

Flux архитектура

Действия (action) передаются диспетчеру с помощью удобных классов-помощников, называемых **создателями действий (actionCreators)**, которые обрабатывают простую задачу создания и отправки действия диспетчеру.



Flux архитектура

Диспетчер

Используется для координации связи между создателями действий и хранилищами.

Используем его для регистрации обратного вызова обработчика действий хранилища, а также для отправки действий в хранилища, которые подписались.

API-интерфейс диспетчера прост и в нем доступно всего пять методов:

```
register(): регистрирует обратный вызов обработчика действия хранилища.
unregister(): Отменить регистрацию обратного вызова у хранилища.
waitFor(): Ожидает, что указанный обратный вызов (ы) запускается первым.
dispatch(): Выполняет действие.
isDispatching(): Проверяет, отправляет ли диспетчер какое-либо действие.
```

Flux архитектура

ПЕРЕДОВАЯ ИНЖЕНЕРНАЯ ШКОЛА УНИВЕРСИТЕТА ИННОПОЛИС

Наиболее важными являются register() и dispatch(), поскольку они используются для обработки большинства основных функций.

```
let _callbacks = [];
class Dispatcher {
    // Register a store callback.
  register(callback) {
    let id = 'callback_' + _callbacks.length;
    _callbacks[id] = callback;
    return id;
  }
  // Dispatch an action.
  dispatch(action) {
    for (var id in _callbacks) {
      _callbacks[id](action);
    }
  }
}
```

Flux архитектура

ПЕРЕДОВАЯ ИНЖЕНЕРНАЯ ШКОЛА
УНИВЕРСИТЕТА ИННОПОЛИС

Метод register() сохраняет все обратные вызовы в приватном массиве _callbacks

dispatch() выполняет итерацию и вызывает каждый обратный вызов, сохраненный для принятого действия.

```
import { Dispatcher } from 'flux';
export default new Dispatcher();
```

Импортируем диспетчера из библиотеки flux, которая была установлена раньше с помощью yarn , а затем экспортирует новый экземпляр.

Flux архитектура

Новый файл Actions.js и следующий код:

```
import Dispatcher from '../dispatcher';
import ActionTypes from '../constants';
class WalletActions {
    addNewItem(item) {
    // Note: This is usually a good place to do API calls.
    Dispatcher.dispatch({
      actionType: ActionTypes.ADD_NEW_ITEM,
      payload: item
    });
  }
}
export default new WalletActions();
```

Flux архитектура

Класс WalletActions предоставляет метод addNewItem(), который обрабатывает три основные задачи:

- Он получает item в качестве аргумента.
- Он использует диспетчера для отправки действия с типом действия ADD_NEW_ITEM
- Затем он отправляет полученный item в качестве пейлоада вместе с типом действия.

Flux архитектура

ПЕРЕДОВАЯ ИНЖЕНЕРНАЯ ШКОЛА
УНИВЕРСИТЕТА ИННОПОЛИС

Хранилища
Их роль заключается в обработке логики и хранении состояния для определенного компонента верхнего уровня в вашем приложении.

Все хранилища Flux должны определить метод обработчика действий, который затем будет зарегистрирован диспетчером.

Функция обратного вызова в основном состоит из оператора switch для полученного типа действия.

Если выполняется конкретный тип действия, он действует соответствующим образом и обновляет локальное состояние. Хранилище передает событие, чтобы сигнализировать представления об обновленном состоянии, чтобы они могли соответствующим образом его обновлять.

Чтобы транслировать события, хранилища должны расширять логику эмиттера событий. Существуют различные библиотеки эмиттеров событий

Flux архитектура


```
import { EventEmitter } from 'events';
import Dispatcher from '../dispatcher';
import ActionTypes from '../constants';
const CHANGE = 'CHANGE';
let _walletState = [];
class WalletStore extends EventEmitter {
    constructor() {
    super();
    // Registers action handler with the Dispatcher.
    Dispatcher.register(this._registerToActions.bind(this));
  }
  // Switches over the action's type when an action is dispatched.
  _registerToActions(action) {
    switch(action.actionType) {
      case ActionTypes.ADD_NEW_ITEM:
        this._addNewItem(action.payload);
      break;
    }
  }
  // Adds a new item to the list and emits a CHANGED event.
  _addNewItem(item) {
    item.id = _walletState.length;
    _walletState.push(item);
    this.emit(CHANGE);
  }
  // Returns the current store's state.
  getAllItems() {
    return _walletState;
  }
  // Calculate the total budget.
  getTotalBudget() {
    let totalBudget = 0;
    _walletState.forEach((item) => {
      totalBudget += parseFloat(item.amount);
    });
    return totalBudget;
  }
  // Hooks a React component's callback to the CHANGED event.
  addChangeListener(callback) {
    this.on(CHANGE, callback);
  }
  // Removes the listener from the CHANGED event.
  removeChangeListener(callback) {
    this.removeListener(CHANGE, callback);
  }
}
export default new WalletStore();
```

Flux архитектура

Начинаем с импорта необходимых зависимостей, необходимых для хранилища, начиная с эмиттера события Node, диспетчера, за которым следуют ActionTypes.

При инициализации класс WalletStore начинается с регистрации обратного вызова _registerToAction() в диспетчере. За кулисами этот обратный вызов будет добавлен в массив _callbacks диспетчера.

Метод имеет один оператор switch над типом действия, полученным от диспетчера при отправке действия. Если он соответствует типу действия ADD_NEW_ITEM, то затем запускается метод _addNewItem() и ему передается полученный пейлоад.

Функция _addNewItem() устанавливает id для элемента, добавляет его к списку существующих элементов и затем запускает событие CHANGE.

Далее методы getAllItems() и getTotalBudget() являются базовыми геттерами, которые будем использовать для извлечения состояния текущего хранилища и общего бюджета.

Последние два метода addChangeListener() и removeChangeListener() будут использоваться для связывания компонентов React с WalletStore, чтобы они получали уведомление при изменении данных хранилища.

Flux архитектура

Представления контроллеров

React позволяет разбить части приложения на различные компоненты.

В Flux компоненты, расположенные в верхней части цепочки, хранят большую часть логики, необходимой для создания действий и получения новых данных;

Поэтому они называются представлениями контроллера. Эти представления напрямую подключаются к хранилищам и прослушивают события изменений, вызванные при обновлении хранилищ.

Когда происходят просмотры контроллера вызывают метод setState, который запускает метод render() для запуска и обновления представления и отправки данных дочерним компонентам через props. Затем React и Virtual DOM делают свою магию и максимально обновляют DOM.

Flux архитектура

ПЕРЕДОВАЯ ИНЖЕНЕРНАЯ ШКОЛА
УНИВЕРСИТЕТА ИННОПОЛИС

Соберем все вместе

Отправка действия
Возвращаясь к компоненту <AddNewItem/>, теперь можно включить модуль WalletActions и использовать его для генерации нового действия в методе _addNewItem().

```
import React from 'react';
import WalletActions from '../actions/walletActions';
// ...
_addNewItem(event) {
    event.preventDefault();
  this.state.item.description = this.state.item.description || '-';
  this.state.item.amount = this.state.item.amount || '0';
  WalletActions.addNewItem(this.state.item);
  this.setState({ item : this._getFreshItem() });
}
// ...
```

Flux архитектура

ПЕРЕДОВАЯ ИНЖЕНЕРНАЯ ШКОЛА
УНИВЕРСИТЕТА ИННОПОЛИС

Прослушивание изменений в хранилище

В WalletStore, элемент добавляется в список, его состояние изменяется, а событие CHANGE запускается, но его никто не слушает.

Добавив слушателя изменений внутри компонента <ItemsList/>

```
import React from 'react';
import WalletStore from '../stores/walletStore';
class ItemsList extends React.Component {
    constructor(props) {
    super(props);
    this.state = {
      items: WalletStore.getAllItems()
      };
        this._onChange = this._onChange.bind(this);
  }
  _onChange() {
    this.setState({ items: WalletStore.getAllItems() });
  }
  componentWillMount() {
    WalletStore.addChangeListener(this._onChange);
  }
  componentWillUnmount() {
    WalletStore.removeChangeListener(this._onChange);
  }
  render() {
        // ...
    }
}
export default ItemsList;
```

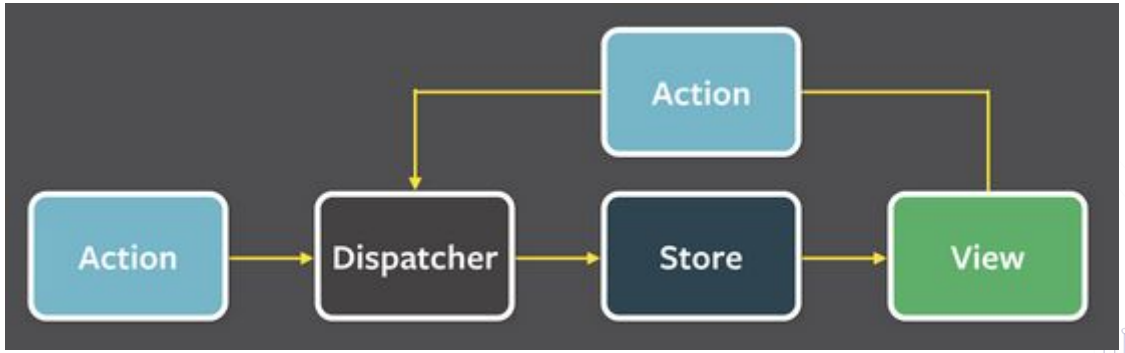
Flux архитектура

ПЕРЕДОВАЯ ИНЖЕНЕРНАЯ ШКОЛА УНИВЕРСИТЕТА ИННОПОЛИС

Обновленный компонент закрывает поток однонаправленных потоков Flux.

все шаг за шагом:

- Подключаем модуль WalletStore.
- Локальное состояние компонента обновляется,
- метод _onChange() используется для обновления состояния новыми данными из хранилища



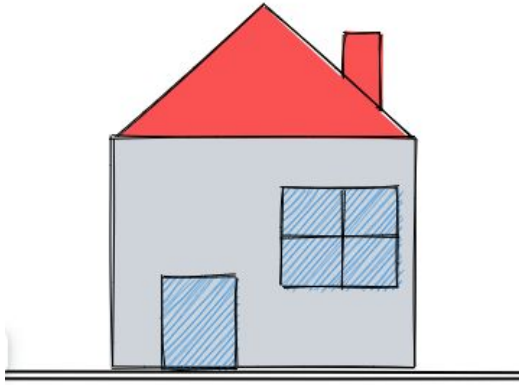
Redux

ПЕРЕДОВАЯ ИНЖЕНЕРНАЯ ШКОЛА
УНИВЕРСИТЕТА ИННОПОЛИС

Что такое Redux?

Redux - это библиотека управления состоянием, которая позволяет управлять состоянием приложений JavaScript более эффективно и предсказуемо.

Представьте, что вы строите дом и вам нужно отслеживать все материалы, которые вы используете, и сколько денег вы тратите. Вместо того, чтобы отслеживать все в своей голове или на листе бумаги, вы могли бы использовать бухгалтерскую книгу для отслеживания каждой транзакции. Redux работает аналогично, отслеживая состояние вашего приложения в одном месте, называемом "хранилище".

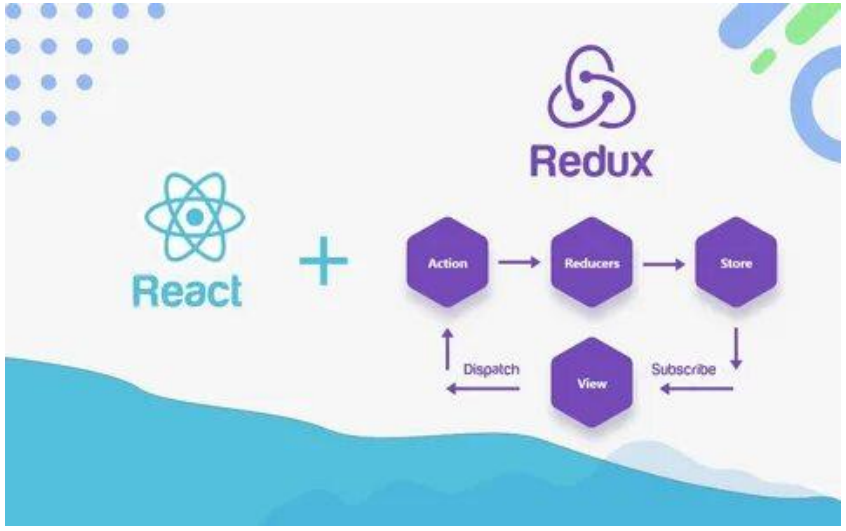


Redux

Допустим, создаете сайт электронной коммерции. Возможно, потребуется отслеживать товары в корзине пользователя, их платежную информацию и данные о доставке.

Вместо передачи этой информации от компонента к компоненту с помощью props, Redux позволяет хранить их в одном центральном месте, где к ним можно легко получить доступ и обновлять.

Это упрощает управление сложными состояниями и упорядочивает ваше приложение.
Важно отметить, что Redux не ограничивается React, и можно использовать его с другими фреймворками или даже с ванильным JavaScript.



Redux

Почему хорошо использовать Redux?

Redux может помочь упростить процесс управления состоянием, особенно при работе со сложными и взаимосвязанными компонентами. Вот несколько причин, по которым вы, возможно, захотите использовать Redux в своем приложении:

1. **Централизованное управление состоянием:** С помощью Redux вы можете поддерживать состояние всего вашего приложения в одном хранилище, упрощая управление данными между компонентами и доступ к ним.

2. **Предсказуемые обновления состояния:** Redux имеет четкий поток данных, что означает, что изменения состояния могут произойти только тогда, когда вы создаете действие и отправляете его через Redux. Это упрощает понимание того, как данные вашего приложения будут меняться в ответ на действия пользователя.

3. **Упрощенная отладка:** с Redux DevTools у вас есть четкая запись всех изменений состояния вашего приложения. Это упрощает поиск и устранение проблем в вашем коде, экономя ваше время и усилия в процессе отладки.

4. **Повышение производительности:** Сводя к минимуму количество обновлений состояния и уменьшая потребность в детализации реквизитов, Redux помогает повысить производительность вашего приложения.

Redux

ПЕРЕДОВАЯ ИНЖЕНЕРНАЯ ШКОЛА
УНИВЕРСИТЕТА ИННОПОЛИС

Как работает Redux?

Redux позволяет поддерживать единое централизованное хранилище, которое управляет состоянием всего приложения.

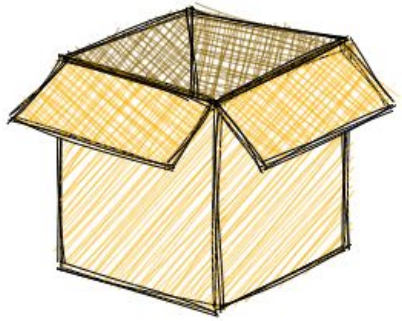
Все компоненты приложения могут обращаться к этому хранилищу и обновлять или извлекать из него данные по мере необходимости.

Ключевыми компонентами, которые обеспечивают этот централизованный подход к управлению состоянием, являются:

1. Storage
2. Actions
3. Dispath
4. Reducers

Redux

Redux Store похож на гигантский контейнер, в котором хранятся все данные для приложения.



Представьте хранилище как коробку с разными отделениями для разных типов данных. В этих ячейках можно хранить любые данные, которые хотите, и они могут содержать различные типы данных, такие как строки, числа, массивы, объекты и даже функции.

Кроме того, хранилище является единственным источником достоверной информации о состоянии приложения. Это означает, что любой компонент приложения может получить к нему доступ для извлечения и обновления данных.

Redux

Actions
это объект, который описывает, какие изменения необходимо внести в состояние приложения. Оно отправляет данные из приложения в хранилище Redux и служит единственным способом обновления хранилища.

Действие должно иметь свойство "type", описывающее выполняемое действие. Это свойство "type" обычно определяется как строковая константа, чтобы обеспечить согласованность и избежать опечаток.

В дополнение к свойству "тип" действие может иметь свойство "полезная нагрузка". Свойство "полезная нагрузка" представляет данные, которые предоставляют дополнительную информацию о выполняемом действии. Например, если тип действия является ADD_TASK, полезной нагрузкой может быть объект, содержащий "идентификатор" нового элемента задачи, "текст" и "статус завершения".

```
{
  type: 'ADD_TASK',
  payload: {
    id: 1,
    text: 'Buy groceries',
    completed: false
  }
}
```

Redux

Action creators - это функции, которые создают и возвращают объекты action.

пример создания действий, принимает текст задачи и возвращает объект действия для добавления задачи в хранилище Redux:

```
function addTask(taskText) {
  return {
    type: 'ADD_TASK',
    payload: {
      id: 1,
      text: taskText,
      completed: false
    }
  }
}
```

Redux

В Redux dispatch - это функция, предоставляемая storage, которая позволяет отправлять действие для обновления состояния приложения.

При вызове dispatch хранилище выполняет действие через все доступные reducers, которые, в свою очередь, соответствующим образом обновляют состояние.

dispatch это почтальон, который доставляет почту в разные отделы крупной компании.

Точно так же, как почтальон доставляет почту в разные отделы, dispatch доставляет действия различным отправителям в Redux Store.

Каждый reducer подобен отделу в компании, который обрабатывает почту и обновляет свою часть данных компании.

Redux

ПЕРЕДОВАЯ ИНЖЕНЕРНАЯ ШКОЛА УНИВЕРСИТЕТА ИННОПОЛИС

В Redux редуктор - это функция, которая принимает текущее состояние приложения и действие в качестве аргументов и возвращает новое состояние на основе действия.

```
const initialState = {
  count: 0
};

function counterReducer(state = initialState, action) {
  switch(action.type) {
    case 'INCREMENT':
      return { ...state, count: state.count + 1 };
    case 'DECREMENT':
      return { ...state, count: state.count - 1 };
    default:
      return state;
  }
}
```

Redux

В приведенном коде есть простой reducer под названием "counterReducer", который управляет состоянием переменной count. Он принимает два аргумента: state и action. state аргумент представляет текущее состояние приложения, в то время как action аргумент представляет действие, отправленное для изменения состояния.

Затем reducer использует инструкцию switch для проверки "типа" действия и на основе этого типа соответствующим образом обновляет состояние.

Например, если тип действия - “INCREMENT", reducer возвращает новый объект состояния с количеством, увеличенным на 1.

если тип действия "DECREMENT", reducer возвращает новый объект состояния с уменьшенным на 1 значением.

Redux

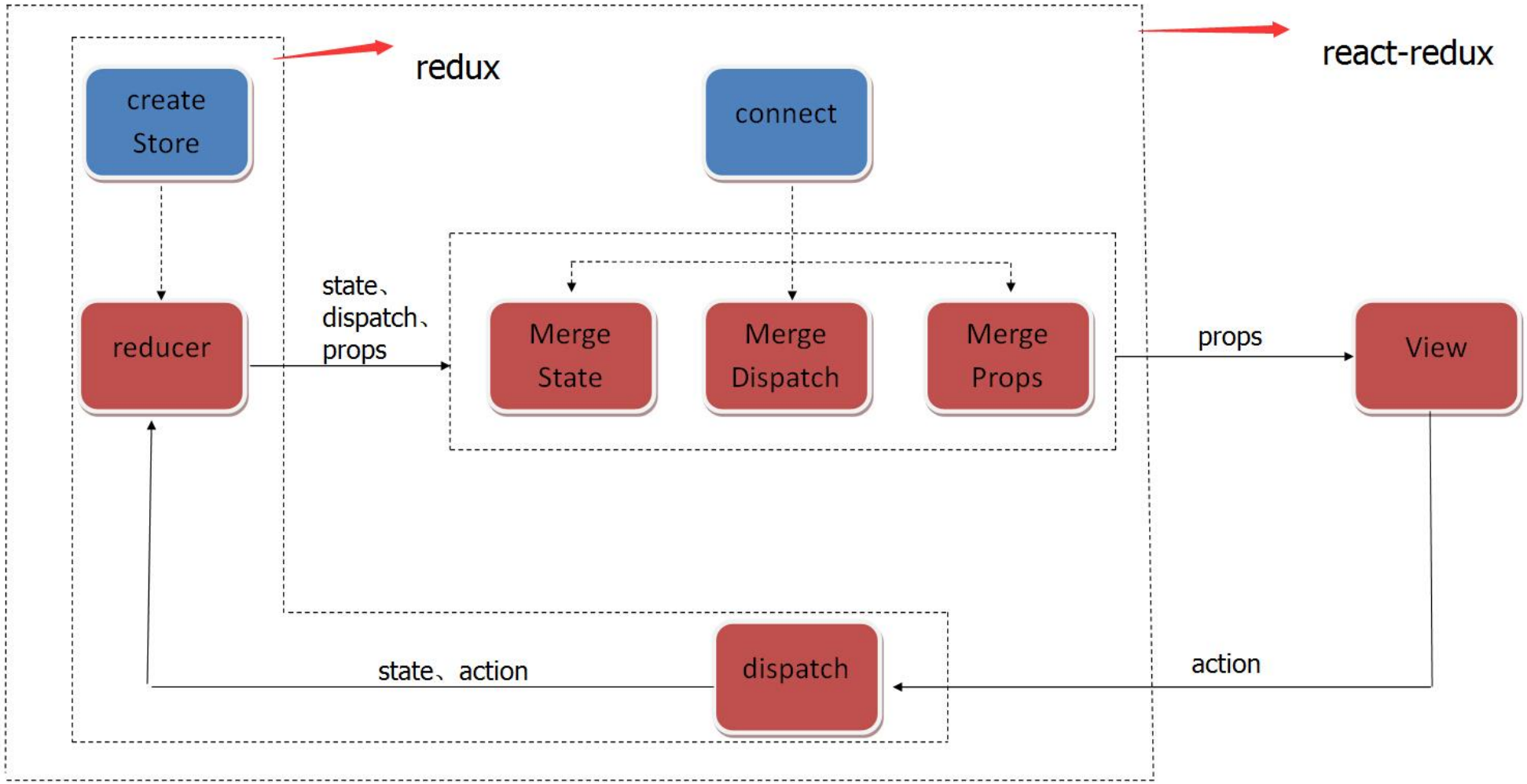
В приведенном коде есть простой reducer под названием "counterReducer", который управляет состоянием переменной count. Он принимает два аргумента: state и action. state аргумент представляет текущее состояние приложения, в то время как action аргумент представляет действие, отправленное для изменения состояния.

Затем reducer использует инструкцию switch для проверки "типа" действия и на основе этого типа соответствующим образом обновляет состояние.

Например, если тип действия - “INCREMENT", reducer возвращает новый объект состояния с количеством, увеличенным на 1.

если тип действия "DECREMENT", reducer возвращает новый объект состояния с уменьшенным на 1 значением.

Redux


redux
react-redux
create Store
connect
reducer
state、dispatch、props
Merge State
Merge Dispatch
Merge Props
props
View
dispatch
state、action
action

Передовые инженерные школы

МИНОБРНАУКИ РОССИИ

УНИВЕРСИТЕТ ИННОПОЛИС

Спасибо за внимание