React.js Course

Занятие 3. Жизненный цикл, согласование, продвинутый рендеринг

Автор программы и преподаватель - Помазкин Илья

Структура занятия

1. Архитектура клиент-сервер, запросы
2. Жизненный цикл
3. Согласование
4. Рефы

Архитектура клиент-сервер

Фронтенд выполяет функцию интерйфейса - он не хранит данные, не производит сложные вычисления. Для этого - есть бекенд. Обычно бекнд - это сервер, запущенный на удаленном компьютере. Этот сервер умеет общаться с внешним миром - он умеет обрабатывать сетевые запросы.

Для отображения фронтенда мы используем веб-браузер. Мы переходим по какому-то URL, а браузер делает запросы к бекенду, загружает страницу и показывает ее нам. Такие запросы называются синхронными, для их выполнения браузеру нужно перезагружать страницу, что бы сделать запрос и обработать ответ.

Но мы также можем делать AJAX-запросы. Это асинхронные запросы, которые не требуют перезагрузки страницы. Они обычно используются для интерактивной загрузки и отправки данных, например: варианты для автокомплита, отправкап формы без перезагрузки страницы и т.д.

И в обоих случаях у нас есть бекенд и фронтенд, клиент и сервер.

Создание асинхронных запросов в JavaScript

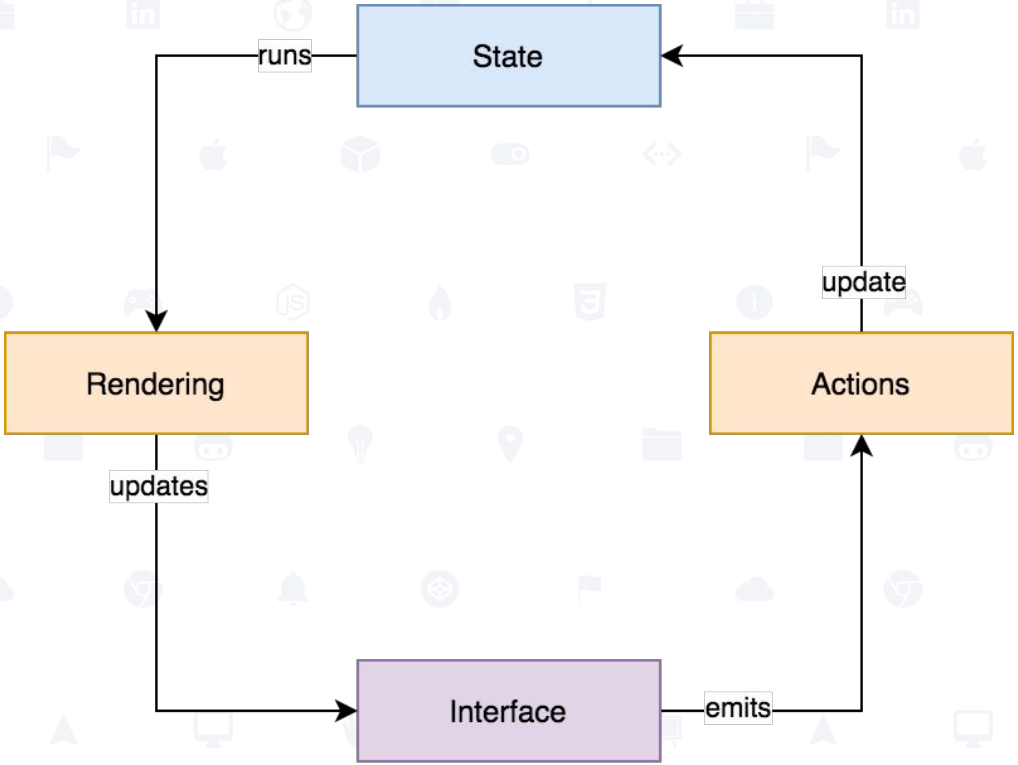
В JavaScript у нас есть несколько вариантов для создания запросов:

- XMLHttpRequest - нативно, но не очень удобно
- fetch - тоже нативно, чуть удобнее, можно использовать промисы, но все еще будет много шаблонного кода
- $.ajax - для любителей jQuery :)
- axios - очень удобная библиотека, поддерживает из коробки отправку данных с формы, CORS и прочие плюшки
- superagent - еще одна библиотека

Мы будем использовать axios.

Для генерации данных и в качестве бекенда будем использовать https://mockapi.io.

Напоминание - концепция интерактивности



1. Пользователь видит **интерфейс** и делает в нем какие-то действия
2. **Действия** обновляют состояние
3. Изменение **состояния** запускает рендер
4. **Рендер** обновляет интерфейс

Напоминание - классовые компоненты

React-компоненты есть двух типов: классовые и функциональные. На этом уроке рассмотрим классовые.

Классовый компонент:

```
import React, { Component } from "react";

export class Button extends Component {
   render() {
       const { children, ...rest } = this.props;

       return (
           <button {...rest}>{children}</button>
       );
   }
}
```

Так же у класса есть дополнительные методы - мы разберем их чуть позже.

Класс должен наследовать от React.Component.
За рендер отвечает метод render - он должен вернуть React-элемент.

Пропсы записываются в свойство объекта “props”.
К ним можно обратиться через this.props в методе render.

Дочерние элементы доступны в props.children.

Жизненный цикл компонента

Классовый компонент за время своей жизни проходит 3 фазы:

1. Монтирование - это первый рендер и вставка результата рендера в DOM
2. Обновление - это все последующие рендеры и обновление ранее вставленного в DOM
3. Размонтирование - это удаление ранее вставленного из DOM

Каждая фаза может иметь до 3 этапов:

1. Render - влияет на рендер, может быть приостановлен, прерван или перезапущен самим React
2. Pre-commit - можно производить чтение DOM до его обновления
3. Commit - можно делать что-то после обновления DOM: ручное изменение DOM, запросы к серверу, обновление состояния и т.д.

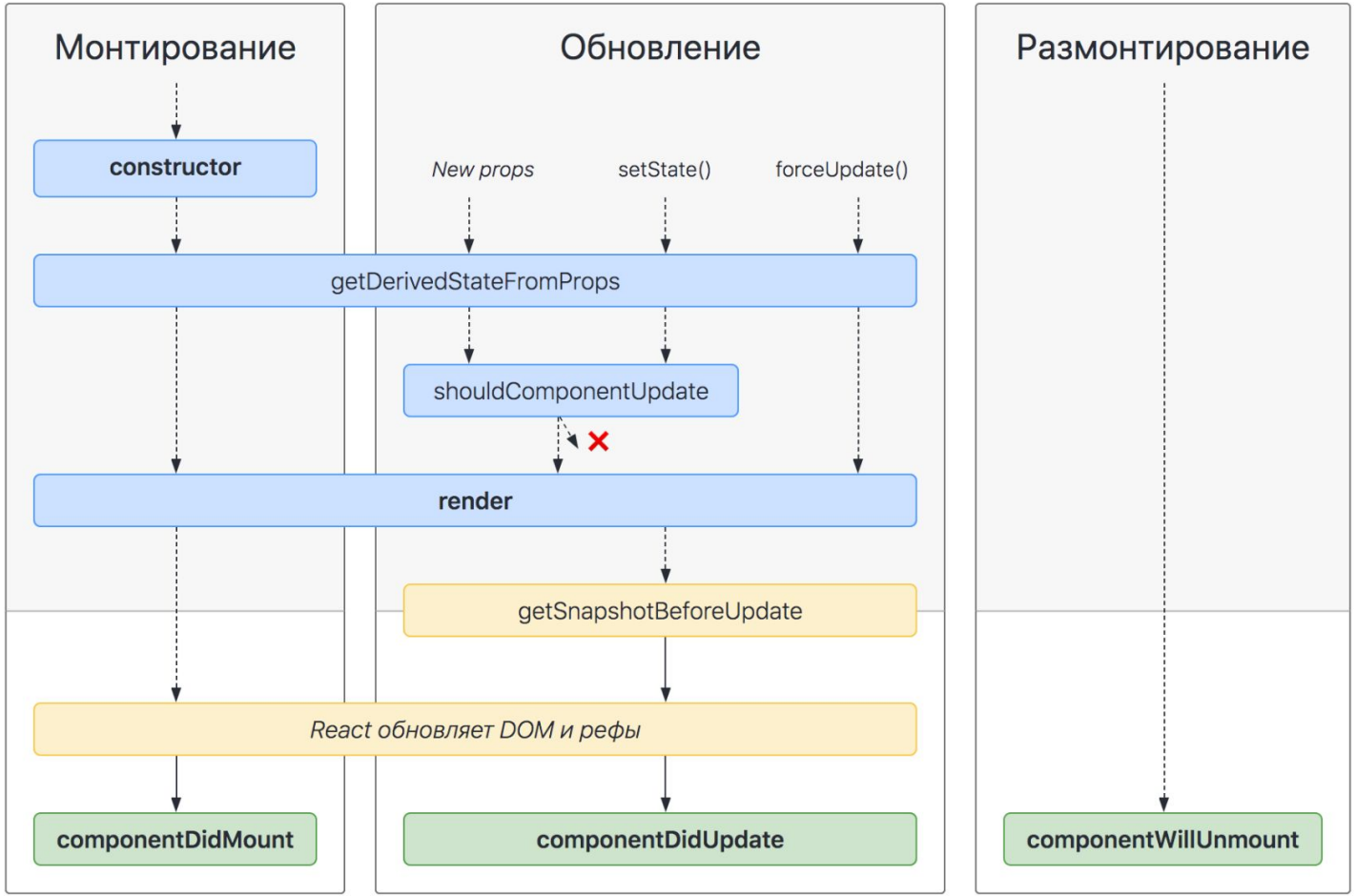
Жизненный цикл компонента - фазы и этапы

ссылка на источник

"Этап Render"
Чистый и не имеет побочных эффектов. Может быть приостановлен, прерван и перезапущен самим React.

"Этап Pre-commit"
Можно производить чтение DOM.

"Этап Commit"
Может работать с DOM, выполнять побочные эффекты, назначать обновления.



Методы LC - constructor

Фаза: монтирование, этап - render. Вызывается при создании экземпляра компонента.

Что обязательно:
- вызов super(props)

Что можно делать:
- присваивать начальное состояние
- привязывать методы к экземпляру класса (this.handleSome.bind(this))
- устанавливать начальное, дефолтное состояние исходя из пропсов

Что нельзя делать:
- вызывать this.setState
- запускать сайд-эффекты и добавлять обработчики событий
- тупо копировать props в state, так как при обновлении пропсов они не будут повторно копироваться в state.

Методы LC - static getDerivedStateFromProps

Фазы: монтирование и обновление, этап - render. Вызывается перед render, при монтировании и каждом обновлении.

Получает 2 аргумента - props и state, на основе которых можно вернуть состояние для обновления.

Что обязательно:
- вернуть из метода или null для того, что бы ничего не обновлять, или объект для обновления состояния

Что можно делать:
- только одно - устанавливать состояние на основе пропсов

Что нельзя делать:
- обращаться к this - так как это статичный метод

Методы LC - shouldComponentUpdate

Фаза: обновление, этап - render. Вызывается перед рендером каждый раз, когда изменилось состояние или пропсы. Указывает, нужно ли вызывать метод render или нет. Нужен только для оптимизации производительности.

Получает 2 аргумента - nextProps, nextState, то есть - новые пропсы и новое состояние.

Что обязательно:
- вернуть true, если хотим что бы рендер был вызван или false - если не хотим

Что можно делать:
- на основе новых пропсов и состояния и старых пропсов и состояния решить, стоит ли вызывать рендер

React.PureComponent имеет метод shouldComponentUpdate, который уже реализовал логику поверхностного сравнения старых и новых state и props. Так что можно просто наследоваться от него.

Методы LC - render

Фазы: монтирование и обновление, этап - render. Это собственно сам рендер. Метод должен вернуть React-элемент.

Что обязательно:
- вернуть валидный React-элемент

Что можно делать:
- рендерить HTML на основе пропсов и состояния
- добавлять обработчики событий

Что нельзя делать:
- вызывать this.setState
- запускать сайд-эффекты

Методы LC - componentDidMount

Фаза: монтирование, этап - commit. Вызывается после вставки компонента в DOM.

Что можно делать:

- вызывать this.setState (вызовет доп. рендер, при этом результаты первого рендера юзеру не покажут, а покажут сразу результаты доп. рендера)
- запускать сайд-эффекты и добавлять обработчики событий
- инициализировать сторонние библиотеки, которые изменяют DOM (например плагины jQuery)
- изменять DOM вручную

Методы LC - getSnapshotBeforeUpdate

Фаза: обновление, этап - pre-commit. Вызывается перед обновлением DOM. Он нужен, что бы вытащить какую-то информацию из DOM до обновления, например положение полосы прокрутки.

Получает 2 аргумента - prevProps, prveState, то есть - предыдущие пропсы и предыдущее состояние.

Что обязательно:
- вернуть null или что-то другое

Что можно делать:
- на основе пропсов и состояния брать какую-то информацию из DOM и возвращать ее

Что нельзя делать:
- запускать сайд-эффекты и прочее

То, что вернет этот метод будет 3 аргументом метода componentDidUpdate.

Методы LC - componentDidUpdate

Фаза: обновление, этап - commit. Вызывается сразу после обновления.

Получает 3 аргумента - prevProps, prevState, snapshot. Аргемент snapshot будет равен тому, что возвращает метод getSnapshotBeforeUpdate, если он реализован.

Что можно делать:
- вызывать this.setState, обернутое в условие (вызовет доп. рендер, при этом результаты предыдущего рендера юзеру не покажут, а покажут сразу результаты доп. рендера)
- запускать сайд-эффекты
- изменять DOM вручную

Что нельзя делать:
- вызывать this.setState без проверки на необходимость обновления состояния - иначе попадем в бесконечный цикл

Методы LC - componentWillUnmount

Фаза: размонтирование, этап - commit. Вызовется прямо перед удалением компонента из DOM.

Что можно делать:
- удалять обработчики глобальных событий
- очистка таймеров
- остановка сайд-эффектов

Что нельзя делать:
- вызывать this.setState, так как повторный рендер уже не произойдет

Методы LC - старые и небезопасные методы

- UNSAFE_componentWillMount - вызывается в фазе монтирования перед метором render
- UNSAFE_componentWillUpdate(nextProps, nextState) - вызывается а фазе обновления перед рендером
- UNSAFE_componentWillReceiveProps(nextProps) - вызывается в фазе обновления перед тем как компонент получит новые пропсы

НЕ ИСПОЛЬЗУЙТЕ ЭТИ МЕТОДЫ!!!

Методы LC - forceUpdate

Обычно повторный рендер запускается при изменении пропсов или состояния. И иногда нужно запустить рендер при изменении каких-либо других, внутреннх данных. Для этого есть метод forceUpdate.

Он запускает фазу обновления компонента, но при этом в ней игнорируется вызов shouldComponentUpdate.

Во всех дочерних элементах фаза обновления будет отработана как обычно.

Жизненный цикл - логика фазы обновления

1. Меняются пропсы или состояние
2. Вызывается getDerivedStateFromProps, который может изменить что-то в новом состоянии
3. Вызывается shouldComponentUpdate.
   a. Если вернул true - идем дальше
   b. Если вернул false - останавливаем фазу тут
4. Вызывается render. Он возвращает дерево React-элементов
5. Вызывается getSnapshotBeforeUpdate. Поскольку DOM еще не обновлен из него можно запомнить какую-то информацию.
6. Запускается алгоритм ***согласования*** для сравнения старого и нового деревьев.
7. Вносятся изменения в DOM.
8. Обновляются ref-ы
9. Вызывается componentDidUpdate.

Согласование

Virtual DOM - это деревья React-элементов, которые создаются функцией React.createElement и возвращаются из метода render. При обновлении состояния или пропсов React повторно запускает метод render и он создает новое дерево React-элементов. И теперь React должен сравнить старое и новое дерево и найти различия. В зависимости от того, какие они будут - React внесет изменения уже в DOM.

Изменения вносятся по определенным правилам:

1. Если у элемента поменялся тип - старый элемент вместе со своим деревом будет размонтирован и элемент с новым типом и деревом будет примонтирован
2. Если у элмента поменялся проп key - будет тоже самое что в п. 1
3. Если тип элемента не изменился - React обновит только измененные атрибуты
4. Если изменились свойства в атрибуте style - React обновит только измененые своства style
5. При работе с массивами элементов React по key различает элементы списка и отслеживает добавление, удаление, изменение порядка, изменение пропсов.

Как еще можно использовать проп key

С помощью пропа key, если он задан, React понимает, как идентифицировать какой-то компонент.

Это особенно важно при рендере списков - что бы React понимал, что обновлять.

Но prop key можно использовать не только для списоков. Если компоненту просто указать проп key - ничего не измениться. Но если потом этот проп будет изменен - React размонтирует компонент со старым ключом и примонтирует компонент с новым ключом.

То есть изменением пропа key можно указать React-у, что нужно полностью переподключить какой-то компонент.

Рефы

Нужны для прямого доступа к DOM-узлам или созданным в методе render React-элементам. Например, что бы руками изменить что-то в DOM или передать DOM-узел в какой-то сторонний плагин для подключения. Случаи, когда могут потребоваться рефы:

- работа с неуправляемыми полями формы
- сложная пошаговая анимация
- взаимодействие с HTML API: аудио, видео, прочее
- интеграция с сторонними библиотеками, которые могут изменять DOM: jQuery-плагины, текстовые маски, прочее
- везде, где нужно руками залезть в DOM

Рефы - объектный реф

Для создания рефа есть фукнция React.createRef. Она возвращает объект с одним единственным полем - current, которое по умолчанию равно null.

Рефы удобно создавать в конструкторе или поле класса, что бы они были потом доступны из любого метода класса. После этого в методе render нужно передать в проп ref целевого тега ссылку на созданный реф.

После установки рефов, что бы обратиться к значению рефа - обращайтесь к полю refObj.current.

```
class Animator extends PureComponent {
 rootEl = React.createRef(); // создаем объектный реф

 componentDidMount() {
   // обращаемся к значению рефа
   let bcr = this.rootEl.current.getBoundingClientRect();
 }

 render() {
   return (
     // передаем в проп ref ссылку на объектный реф
     <div ref={this.rootEl} className="animator" />
   );
 }
}
```

Рефы - коллбек-реф

Так же в ref можно передавать коллбек-функцию, которая будет вызвана в момент установки рефов. Первым аргументом функции будет DOM-элемент или экземпляр компонента.

Со значением можно делать что угодно, но обычно его просто сохраняют в свойстве экземпляра компонента.

```
class Animator extends PureComponent {
 componentDidMount() {
   // обращаемся к сохраненному значению рефа
   let bcr = this.rootEl.getBoundingClientRect();
 }

 render() {
   return (
     // передаем в проп ref коллбек-реф
     <div ref={(el) => {
       this.rootEl = el; // сохраняем элемент в свойстве rootEl
     }} className="animator" />
   );
 }
}
```

Рефы - привязка к DOM и к компонентам

Если мы передаем реф тегу - в нем будет храниться ссылка на DOM-узел.
Если передаем реф классовому уомпоненту - в нем будет храниться экземпляр компонента.

```
class Button extends PureComponent {
 render() { return <button>...</button> }
}

class Animator extends PureComponent {
 rootEl = React.createRef();
 buttonEl = React.createRef();

 componentDidMount() {
   let rootNode = this.rootEl.current; // тут будет DOM-узел
   let buttonComponent = this.buttonEl.current; // тут будет экземпляр Button
 }

 render() {
   return (
     <Fragment>
       <div ref={this.rootEl} className="animator" />
       <Button ref={this.buttonEl} />
     </Fragment>
   );
 }
}
```

Рефы - перенаправление рефов

Если нам нужно, что бы ref классового компонента указывал не на экземпляр, а на DOM-узел или другой компонент - можно сделать перенаправление рефа с помощью функции React.forwardRef.

```
class Button extends PureComponent {
 render() {
   let {
     ref, // forwardRef добавит проп ref, который мы перенаправим в тег
     ...rest
   } = this.props;
   // передаем проп ref в тег
   return <button ref={ref}>...</button>;
 }
}
Button = React.forwardRef(Button); // передали класс компонента в forwardRef
// forwardRef сработает как декоратор и добавит классу нужное поведение

class Animator extends PureComponent {
 buttonEl = React.createRef();

 componentDidMount() {
   let buttonComponent = this.buttonEl.current; // тут будет DOM-узел
 }

 render() {
   return <Button ref={this.buttonEl} />;
 }
}
```

Рефы - общий алгоритм работы с рефами

1. Создание рефа и передача его в проп
   a. Создаем объектный реф и присваиваем его в свойство класса, передаем в проп
   b. Или передаем в проп коллбек-реф, внутри которого делаем сохранение значения рефа
   c. если нужно - делаем перенаправлене рефа
2. Доступ к рефу в фазе commit
   a. Через ref.current для объектных
   b. Через прямой доступ к сохранненому значению для коллбек-рефов
3. Делаем что-то с DOM-узлом: инициализируем библиотеки, меняем DOM, что угодно.