**Локальный стейт, состояние и жизненный цикл**

Сегодня мы разберемся что такое локальный стейт в React. Напишем с его помощью выпадающее меню. Разберемся какие сложности бывают с вызовом методов класса.

У реакта есть 2 типа данных - это props и state. Сегодня мы разберем что такое локальный стейт и как с ним работать в react.

Если вам нужно чтобы какие то изменения происходили только внутри конкретной компоненты, то всегда стоит использовать локальный стейт.

Давайте рассмотрим примеры:

Для того, чтобы вывести на экран что-нибудь мы используем ReactDOM.render().

**function** *tick*() {  
 **const** element = (  
 <**div**>  
 <**h1**>Hello, world!</**h1**>  
 <**h2**>It is {**new** Date().toLocaleTimeString()}.</**h2**>  
 </**div**>  
 );  
 **ReactDOM**.**render**(  
 element,  
 **document**.getElementById(**'root'**)  
 );  
}  
  
setInterval(*tick*, 1000);

<https://codepen.io/gaearon/pen/gwoJZk?editors=0010>

В данном примере каждую секунду вызывается функция tick(), которая каждый раз рендерит элемент с новым значение таймера. Но мы хотим прописать код часов один раз и получить такие часы, которые будут самообновляться. Например:

**ReactDOM**.**render**(  
 <**Clock** />,  
 **document**.getElementById(**'root'**)  
);

Чтобы реализовать задуманное, нам нужно добавить к компоненту Clock состояние. Состояние это тоже самое, что и свойства, но оно является закрытым и полностью контролируется компонентом.

Мы упоминали ранее, что компоненты определяются как классы с некоторыми дополнительными характеристиками. Локальное состояние подразумевает функцию, доступную только для классов. Создадим наш компонент с помощью класса:

**class** Clock **extends** React.**Component** {  
 render() {  
 **return** (  
 <**div**>  
 <**h1**>Hello, world!</**h1**>  
 <**h2**>It is {**this**.**props**.**date**.toLocaleTimeString()}.</**h2**>  
 </**div**>  
 );  
 }  
}

<https://codepen.io/gaearon/pen/zKRGpo?editors=0010>

**Добавление локального состояния к классу**

Теперь мы переместим date из свойства в состояние, следуя трем действиям:

**1. Замените this.props.date на this.state.date в алгоритм render()**

**class** Clock **extends** React.**Component** {  
 render() {  
 **return** (  
 <**div**>  
 <**h1**>Hello, world!</**h1**>  
 <**h2**>It is {**this**.**state**.**date**.toLocaleTimeString()}.</**h2**>  
 </**div**>  
 );  
 }  
}

**2. Добавьте class constructor, которому присваиваются инициалы this.state**

**class** Clock **extends** React.**Component** {  
 constructor(props) {  
 **super**(props);  
 **this**.**state** = {**date**: **new** Date()};  
 }  
  
 render() {  
 **return** (  
 <**div**>  
 <**h1**>Hello, world!</**h1**>  
 <**h2**>It is {**this**.**state**.**date**.toLocaleTimeString()}.</**h2**>  
 </**div**>  
 );  
 }  
}

*Компоненты класса всегда должны отдавать сигнал в конструктор с props*

**3. Уберите свойство date из элемента <Clock />**

**ReactDOM**.**render**(  
 <**Clock** />,  
 **document**.getElementById(**'root'**)  
);

Позже мы добавим код таймера обратно к самому компоненту. Результат будет выглядеть так:

**class** Clock **extends** React.**Component** {  
 constructor(props) {  
 **super**(props);  
 **this**.**state** = {**date**: **new** Date()};  
 }  
 render() {  
 **return** (  
 <**div**>  
 <**h1**>Hello, world!</**h1**>  
 <**h2**>It is {**this**.**state**.**date**.toLocaleTimeString()}.</**h2**>  
 </**div**>  
 );  
 }  
}  
**ReactDOM**.**render**(  
 <**Clock** />,  
 **document**.getElementById(**'root'**)  
);

<https://codepen.io/gaearon/pen/KgQpJd?editors=0010>

Далее, мы сделаем так, чтобы Clock установил свой таймер и самообновлялся каждую секунду.

**Жизненый цикл класса**

Мы хотим установить таймер, когда Clock выводятся в DOM впервые.

Мы также хотим, чтобы таймер очищался каждый раз, когда DOM, произведенный от Clock, удаляется.

**class** Clock **extends** React.**Component** {  
 constructor(props) {  
 **super**(props);  
 **this**.**state** = {**date**: **new** Date()};  
 }  
 componentDidMount() {  
  
 }  
 componentWillUnmount() {  
  
 }  
 render() {  
 **return** (  
 <**div**>  
 <**h1**>Hello, world!</**h1**>  
 <**h2**>It is {**this**.**state**.**date**.toLocaleTimeString()}.</**h2**>  
 </**div**>  
 );  
 }  
}

Эти алгоритмы называются привязками жизненного цикла.

Привязка componentDidMount() выполнятся после того, как результат выполнения компонента выводится в DOM. Здесь как раз подходящее место для таймера

componentDidMount() {  
 **this**.**timerID** = setInterval(  
 () => **this**.tick(),  
 1000  
 );  
}

Обратите внимание, что мы сохраняем ID таймера в this.

В то время как React устанавливает this.props, а this.state имеет особое значение, вы можете вручную добавлять дополнительные поля к классу, если вы хотите хранить то, что не используется для визуального вывода.

Если вы ничего не используете в render(), то не нужно ничего добавлять.

Мы разберем таймер в привязке жизненного цикла componentWillUnmount():

componentWillUnmount() {  
 clearInterval(**this**.timerID);  
}

Наконец, мы внедрим алгоритм tick(), который выполняется каждую секунду. Он будет использовать this.setState() со схематичными обновлениями локального состояния компонента:

**class** Clock **extends** React.**Component** {  
 constructor(props) {  
 **super**(props);  
 **this**.**state** = {**date**: **new** Date()};  
 }  
 componentDidMount() {  
 **this**.**timerID** = setInterval(  
 () => **this**.tick(),  
 1000  
 );  
 }  
 componentWillUnmount() {  
 clearInterval(**this**.**timerID**);  
 }  
 tick() {  
 **this**.setState({  
 **date**: **new** Date()  
 });  
 }  
 render() {  
 **return** (  
 <**div**>  
 <**h1**>Hello, world!</**h1**>  
 <**h2**>It is {**this**.**state**.**date**.toLocaleTimeString()}.</**h2**>  
 </**div**>  
 );  
 }  
}  
**ReactDOM**.**render**(  
 <**Clock** />,  
 **document**.getElementById(**'root'**)  
);  
<https://codepen.io/gaearon/pen/amqdNA?editors=0010>

**Изменение состояния**

Для изменения состояния компонента есть метод setState().

Нельзя изменять состояние напрямую

*// Не правильно***this**.**state**.**comment** = **'Hello'**;

*// Правильно***this**.setState({**comment**: **'Hello'**});

Для того, чтобы перезаписать стейт используем деструктуризацию

**this**.**state** = {  
 **title**: **'time'**,  
 **count**:0  
}

**this**.setState({...**this**.**state**, **count**: 2})

или функцию обратного вызова

**let** prev = **this**.**state**;  
**this**.setState((prev) => ({  
 **count**: prev.count+2  
}));

Состояние недоступно ни одному компоненту, кроме того, который устанавливает состояние и управляет им, но компонент способен передавать свое состояние, как свойство (props) дочерним элементам.

**Методы жизненного цикла**

Различные методы выполняются в определенных местах жизненного цикла компонента.

componentWillMount()

Вызывается один раз, на клиенте и сервере, непосредственно перед началом рендеринга. Если вы вызовите setState внутри этого метода, render() будет видеть обновленное состояние и будет выполнен только один раз, несмотря на изменение состояния.

componentDidMount()

Вызывается один раз, только на клиенте (не на сервере), сразу же после того, как происходит инициализация рендеринга. На данном этапе в жизненном цикле компонент имеет представление DOM, к которому вы можете получить доступ с помощью this.getDOMNode().

Если вы хотите интегрироваться с другими фреймворками JavaScript, установите таймеры используя setTimeout или setInterval, или отправьте AJAX запросы, выполняйте эти операции в этом методе.

componentWillReceiveProps(object nextProps)

Вызывается, когда компонент получает новые свойства. Этот метод не вызывается для инициализации рендера.

Используйте это как возможность реагировать на перемещение свойства до вызова render() путем обновления состояния с помощью this.setState(). Старые свойства могут быть доступны через this.props. Вызов this.setState() в этой функции не будет вызывать дополнительный рендеринг.

componentWillReceiveProps: *function*(nextProps) {  
 **this**.setState({  
 **likesIncreasing**: nextProps.likeCount > **this**.**props**.likeCount  
 });  
}

shouldComponentUpdate(object nextProps, object nextState)

Вызывается перед рендерингом при получени новых свойств или состояния. Этот метод не вызывается для начального рендеринга

Используйте это как возможность return false, когда вы уверены, что переход на новые свойства и состояние не потребует обновления компонента.

shouldComponentUpdate: *function*(nextProps, nextState) {  
 **return** nextProps.**id** !== **this**.**props**.**id**;  
}

Если shouldComponentUpdate возвращает false, то render() будет полностью пропускаться до следующего изменения состояния. Кроме того, componentWillUpdate и componentDidUpdate не будут вызываться.

По умолчанию, shouldComponentUpdate всегда возвращает true, чтобы предотвратить коварные ошибки, когда state мутирует на месте, но если вы внимательны, чтобы всегда относиться к state как к неизменному и читать только из props и state в render(), то вы можете переопределить shouldComponentUpdate в реализацию, которая сравнивает старые свойства и состояние с их заменой.

Если производительность упадет, особенно с десятками или сотнями компонентов, используйте shouldComponentUpdate, чтобы ускорить ваше приложение.

componentWillUpdate(object nextProps, object nextState)

Вызывается непосредственно перед рендерингом, когда новые свойства или состояние будет получено. Этот метод не вызывается для начала рендеринга.

Используйте это как возможность выполнить подготовку перед обновлением.

Примечание:

Вы не можете использовать this.setState() в этом методе. Если вам нужно обновить состояние в ответ на изменение свойства, используйте componentWillReceiveProps.

componentDidUpdate(object prevProps, object prevState)

Вызывается сразу после возникновения обновление. Этот метод не вызывается для начала рендеринга.

Используйте это как возможность работать с DOM, когда компонент уже обновлен.

componentWillUnmount()

Вызывается непосредственно перед тем, как компонент демонтируется из DOM.

Выполняйте любую необходимую очистку в этом методе такие как: отключение таймеров или очистки любых DOM элементов, которые были созданы в componentDidMount.

**ОБРАБОТКА СОБЫТИЙ**

Обработка событий на реакте похожа на JS.

Основные отличия:

1. В реакте вы используете camelCase в качестве названия события и обработчик события пишите в {фигурных скобках}, а не в кавычках

**JS:**

<button **onclick**=**"activateLasers()"**>  
 Activate Lasers  
</button>

**React:**

<**button onClick=**{activateLasers}>  
 Activate Lasers  
</**button**>

2. В React Вы не можете вернуть false к предыдущему состоянию по умолчанию. Явно необходим алгоритм preventDefault.

**JS:**

<**a href="#" onclick="console.log('The link was clicked.'); return false"**>  
 Click me  
</**a**>

**React:**

**function** *ActionLink*() {  
 **function** *handleClick*(e) {  
 e.preventDefault();  
 **console**.log(**'The link was clicked.'**);  
 }  
 **return** (  
 <**a href="#" onClick=**{*handleClick*}>  
 Click me  
 </**a**>  
 );  
}

Здесь е - это синтетическое событие реакта. Оно подходит для всех браузеров

3. Как правило, при использовании React, вам не нужно отсылать сигнал к addEventListener, чтобы добавить обработчики событий к DOM элементу после его создания. Вместо этого, просто выполните обработчик когда элемент впервые отображается.

4. Когда вы определяете компонент, используя класс ES6, общим образцом для обработчика событий и будет алгоритмом класса. Например, компонент Toggle отображает кнопку, которая позволяет пользователям выбирать между состояниями «ON» и «OFF»:

**class** Toggle **extends** React.**Component** {  
 constructor(props) {  
 **super**(props);  
 **this**.**state** = {**isToggleOn**: **true**};  
**this**.handleClick = **this**.handleClick.bind(**this**);  
 }  
 handleClick() {  
 **this**.setState(prevState => ({  
 **isToggleOn**: !prevState.**isToggleOn** }));  
 }  
 render() {  
 **return** (  
 <**button onClick=**{**this**.handleClick}>  
 {**this**.**state**.**isToggleOn** ? **'ON'** : **'OFF'**}  
 </**button**>  
 );  
 }  
}  
**ReactDOM**.**render**(  
 <**Toggle** />,  
 **document**.getElementById(**'root'**)  
);

<https://codepen.io/gaearon/pen/xEmzGg?editors=0010>

В данном примере мы делаем подстановку контекста (this) при помощи метода bind

**this**.handleClick = **this**.handleClick.bind(**this**);

Это можно реализовать и при помощи стрелочной функции ES6:

**class** LoggingButton **extends** React.**Component** {  
 *handleClick* = () => {  
 **console**.log(**'this is:'**, **this**);  
 }  
 render() {  
 **return** (  
 <**button onClick=**{**this**.*handleClick*}>  
 Click me  
 </**button**>  
 );  
 }  
}

**Задания**

**1. Сделать выпадающее меню. Несколько блоков, при наведении на которые будет выпадать список.**

**2. Меню для мобильных устройств, которое будет плавно появляться слева по клику на кнопку**

**3. Сделать счетчик, который будет считать количество кликов по кнопке**

**4. Сделать 12 счетчиков**