Оглавление

1 Загрузка данных **3**
1.1 Поток данных 4
1.2 Загрузка данных 8
1.3 React-refetch 14
1.4 Summary 14

Глава 1

Загрузка данных

Цель этой главы - рассмотреть различные способы загрузки данных в React приложении.

Чтобы лучше понимать, как работать с загружаемыми данными, нам нужно будет разобраться как в целом распространяются данные по дереву компонентов в React.

Важно понимать, как родительские компоненты могут коммуницировать с потомками. А также как несвязанные между собой напрямую потомки могут передавать друг другу данные.

Мы посмотрим на конкретные примеры загрузки данных и улучшение структуры компонент, которые загружают данные, с HoC.

И в конце мы посмотрим на удобные библиотеки, такие как react-refetch, которые могут сохранить нам много времени, предоставляя ядро работы с сетью.

В этой главе мы рассмотрим следующие пункты:

- Как Однонаправленный поток данных в React упрощает для понимания структуру приложения
- Как дочерние элементы могут взаимодействовать с родителем через функции обратного вызова
- Как множество дочерних компонент могут делить данные между собой через общий родительский элемент
- Как создать универсальный HoC, с помощью которого можно будет загружать данные из любого API

- Как работает библиотека react-refetch, и как она может упростить работу с сетевыми запросами в нашем приложении

1.1 Поток данных

В последних двух главах мы разбирались, как создавать переиспользуемые компоненты и как эффективно их комбинировать.

Теперь мы поговорим о том, как выстроить правильный поток данных (data flow) между множеством компонент внутри нашего приложения.

React использует очень интересный паттерн, чтобы распространять данные от коренных элементов к дочерним. Этот паттерн обычно называют **Однонаправленный поток данных (Unidirectional Data Flow)**, и в этой части мы посмотрим на него детальнее.

Как видно из названия данные в React компонентах передаются в одном направлении, от корневых элементов к дочерним. У этого подхода есть множество преимуществ, так как это упрощает поведение компонент и их взаимоотношения, делая код более предсказуемым и поддерживаемым.

Каждый компонент получает данные от родительского компонента в виде параметров, которые не должен модифицировать. Также каждый компонент может при необходимости иметь собственное состояние. На основе состояния и полученных параметров компоненты может создать новые данные и передать их дальше по дереву элементов.

Во всех примерах, которые мы видели на текущий момент, данные передавались только от родительских компонент дочерним.

Однако, что делать, если появилась необходимость передать данные от дочернего элемента родительскому?, или родительский элемент должен быть обновлен при изменении состояния дочернего?, или два дочерних элемента хотят передать данные друг другу? Мы ответим на все эти вопросы в процессе разбора примеров из жизни.

Мы начнем с простого компонента, у которого нет потомков, и шаг за шагом преобразуем его в компонент чистый и структурированный.

Мы должны посмотреть, какие паттерны на каждом из шагов преобразования компонента подходят больше всего.

Давайте погрузимся в создание компонента счетчика *Counter*, который имеет две кнопки для увеличения и уменьшения счетчика и значению 0 по умолчанию.

Начнем с создания класса, который наследует *React.Component*:

```
class Counter extends React.Component
```

В конструкторе мы зададим начальное значение счетчика и привяжем к компоненту обработчики событий:

```
constructor(props) {
  super(props)
  this.state = {
    counter: 0,
  }
  this.handleDecrement = this.handleDecrement.bind(this)
  this.handleIncrement = this.handleIncrement.bind(this)
}
```

Обработчики событий будут также просты, им достаточно изменять состояние компонента, увеличивая или уменьшая значение счетчика:

```
handleDecrement() {
  this.setState({
    counter: this.state.counter - 1,
  })
}
handleIncrement() {
  this.setState({
    counter: this.state.counter + 1,
  })
}
```

И в конце нам нужно создать метод *render*, в котором мы будем отображать текущее состояние счетчика и кнопки для его изменения:

```
render() {
  return (
    <div>
      <h1>{this.state.counter}</h1>
      <button onClick={this.handleDecrement}>-</button>
      <button onClick={this.handleIncrement}>+</button>
    </div>
  )
}
```

Взаимодействие потомка с родителем (callbacks)

В общем и целом этот компонент работает, но он делает несколько вещей:

- Содержит во внутреннем состоянии счетчик
- Отвечает за отображение данных
- Содержит логику по увеличению и уменьшению счетчика

Стоит стремиться к тому, чтобы компоненты оставались небольшими и отвечающими за определенную вещь. Это улучшает поддерживаемость приложения и увеличивает шансы пережить изменения требований с меньшей кровью.

Предположим, что нам нужны такие же кнопки плюса и минуса в другом блоке приложения.

Было бы прекрасно, если бы мы смогли переиспользовать кнопки, которые созданы внутри компонента *Counter*, но встает вопрос: если мы вынесем кнопки за границы компонента, то как узнать, когда они были нажаты, чтобы изменить состояние счетчика?

Посмотрим, как мы можем это сделать.

Создадим компонент *Buttons*, который будет отображать необходимые нам кнопки, но вместо того, чтобы определять функции для обработки событий нажатия внутри этого компонента, он будет ожидать их из параметров:

```
const Buttons = ({ onDecrement, onIncrement }) => (
  <div>
    <button onClick={onDecrement}>-</button>
    <button onClick={onIncrement}>+</button>
  </div>
)
Buttons.propTypes = {
  onDecrement: React.PropTypes.func,
  onIncrement: React.PropTypes.func,
}
```

Это обычная компонент-функция, которая передает кнопкам через параметр *onClick* функции, которые получает из параметров.

Теперь мы можем интегрировать этот компонент с кнопками в наш компонент *Counter*:

```
render() {
  return (
    <div>
      <h1>{this.state.counter}</h1>
      <Buttons
```

```
          onDecrement={this.handleDecrement}
          onIncrement={this.handleIncrement}
        />
      </div>
    )
}
```

Как видно, в компоненте *Counter* меняется только блок с кнопками на новый компонент, которому через параметры передаются обработчики событий.

Компонент с кнопками теперь сам по себе ничего не знает о том, кто его использует и лишь уведомляет о нажатиях.

Таким образом, если нам нужно передавать какие-либо данные из дочернего компонента в родительский, то мы можем передать потомку функцию для обратного вызова и реализовать всю остальную логику внутри родительского компонента.

Общий предок

Теперь компонент *Counter* выглядит уже гораздо лучше. Осталось вынести из него часть, отвечающую за отображение.

Чтобы сделать это, мы можем создать компонент *Display*, который будет получать значение и выводить его на экран:

```
const Display = ({ counter }) => <h1>{counter}</h1>
Display.propTypes = {
  counter: React.PropTypes.number,
}
```

Так как нам не нужно хранить состояние внутри этого компонента, мы можем использовать компонент-функцию. Также стоит сказать, что конкретно в этом примере не так много смысла выносить отображение одного $h1$ элемента в отдельный компонент, но в общем случае у вас тут могут быть еще стили, логика смены цвета в зависимости от значение и так далее.

В общем случае, мы должны стремиться делать компоненты так, чтобы они не знали о том, кто именно источник данных, в этом случае их можно будет значительно проще переиспользовать в разных частях приложения.

Теперь мы можем заменить старую разметку в компоненте *Counter* новым компонентом *Display*:

```
render() {
  return (
    <div>
      <Display counter={this.state.counter} />
      <Buttons
        onDecrement={this.handleDecrement}
        onIncrement={this.handleIncrement}
      />
    </div>
  )
}
```

Как вы можете видеть, два дочерних компонента (*Display* и *Buttons*) коммуницируют посредством общего предка, компонента *Counter*.

Когда на компонент *Buttons* кто-либо кликает, он через функцию обратного вызова уведомляет об этом компонент *Counter*, который обновляет данные и передает их компоненту *Display*. Это очень распространенный и эффективный паттерн в React, который позволяет работать с общими данными компонентам, которые не обладают прямой связью.

Данные распространяются от родительских компонентов к дочерним, но последние через функции обратного вызова могут попросить родительский компонент изменить состояние и вызвать тем самым перерисовку других компонентов.

Таким образом, если у нас есть данные, который нужны двум или более компонентам, мы должны найти общего для них родительский компонент и хранить состояние там. В этом случае этот компонент сможет через параметры передавать данные в актуальном состоянии всем дочерним компонентам.

1.2 Загрузка данных

В предыдущей части мы посмотрели, как мы можем передавать данные между компонентами.

Теперь можно разобраться, как в React осуществляется загрузка данных из сети, и в какой части приложения расположить логику загрузки данных.

Примеры в этой главе для http запросов мы будем использовать функцию $fetch$, которая является современной альтернативой функции $XMLHttpRequest$.

На данный момент функция $fetch$ нативно поддерживается толь-

ко браузерами Chrome и FireFox, поэтому если вы хотите поддерживать другие браузеры, вы должны использовать **полифил (polyfill)** от GitHub:

```
https://github.com/github/fetch
```

Мы также будем использовать публичное API GitHub для использования его внутри нашего приложения. Например, мы можем использовать сервис для получения списка **гистов (gists)** пользователя:

```
https://api.github.com/users/:username/gists
```

Гисты - небольшие кусочки кода, которыми пользователи могут делиться между собой через систему GitHub.

Первым компонентом, который мы создадим, будет простой компонент со списком гистов пользователя *gaearon (Dan Abramov)*.

Приступим. Для начала создадим соответствующий класс:

```
class Gists extends React.Component
```

В конструкторе мы создадим начальное состояние, состоящие из пустого списка гистов:

```
constructor(props) {
  super(props)

  this.state = { gists: [] }
}
```

Есть два метода жизненного цикла компонента, в которых можно осуществлять загрузку данных: $componentWillMount$ и $componentDidMount$.

Первый срабатывает перед первой отрисовкой компонента, а второй сразу после окончания монтирования компонента.

Звучит разумным просто использовать первый, так как мы хотим начать загрузку как можно раньше, но есть нюанс.

По факту метод $componentWillMount$ вызывается и на клиенте и на сервере в случае отрисовки на стороне сервера (server-side rendering).

Детальнее об отрисовке на стороне сервера мы поговорим в Главе 8. Сейчас отметим, что вызов асинхронного API в момент отрисовки на стороне сервера может привести к непредсказуемому результату.

Поэтому мы будем использовать $componentDidMount$, чтобы быть уверенными, что вызов API произойдет только на клиенте.

Также стоит учесть, что в реальном проекте вы скорее всего захотите показывать индикатор загрузки во время вызова API. Сделать это можно одним из способов, описанных в Главе 2, в этой главе мы их опустим.

Как мы сказали раньше, мы хотим загрузить список гистов пользователя gaearon функцией *fetch*:

```
componentDidMount() {
  fetch('https://api.github.com/users/gaearon/gists')
    .then(response => response.json())
    .then(gists => this.setState({ gists }))
}
```

Этот код требует некоторых пояснений. Когда срабатывает метод *componentDidMount*, мы вызываем функцию *fetch* адресом нужного нам сервиса.

Функция *fetch* возвращает *Promise*, который в случае успешного выполнения возвращает объект *response* с результатом запроса. Затем из этого объекта с помощью функции *json* можно получить данные в формате JSON.

Затем этот json можно сохранить во внутреннее состояние компонента, чтобы он был доступен из метода *render*:

```
render() {
  return (
    <ul>
      {this.state.gists.map(gist => (
        <li key={gist.id}>{gist.description}</li>
      ))}
    </ul>
  )
}
```

В методе *render* мы просто обходим список гистов и оборачиваем описание каждого из них в тег <li>.

Вы могли обратить внимание на атрибут *key* элементов <li>. Это делается в целях улучшения производительности и будет разобрано подробнее в конце книги.

Если вы удалите этот атрибут, то получите предупреждение в консоли разработчика, но приложение продолжит работать.

Компонент работает, но он пока далек от идеального. Как мы уже увидели в предыдущих главах, мы можем как минимум разнести логику и отображение в разные компоненты, что сделает их проще и более тестируемыми.

Но для нас сейчас более важно то, что мы скорее всего хотим загружать данные из разных участков нашего приложения, но не хотим дублировать соответствующий код в каждый компонент.

Основной способ, который мы можем использовать для переиспользования какой-либо логики в множестве компонент, это компоненты высшего порядка (HoC).

В данном случае HoC будет загружать данные из сети и передавать дочерним компонентам через параметры.

Давайте посмотрим, как это может выглядеть.

Как мы уже знаем, HoC - это функция, которая принимает аргументом компонент (и возможно какие-то параметры) и возвращает его с расширенным функционалом.

Мы будем использовать частичное применение (partial application) для того, чтобы первым вызовом передать параметры, а компонент уже вторым:

```
const withData = url => Component => (...)
```

Мы назвали HoC *withData*, так как он будет передавать данные в параметре *data*.

Функция принимает аргументом url, по которому нужно загрузить данные, и компонент, которому эти данные нужно передать.

Реализация этого HoC будет очень похожа на сам компонент. Разница будет лишь в том, что url теперь приходит в виде аргумента функции, и в методе *render* мы отрисовыываем дочерний компонент.

Функция *withData* будет возвращать класс, который наследует React.Component:

```
class extends React.Component
```

В конструкторе мы определим начальное состояние с пустым списком данным:

```
constructor(props) {
  super(props)

  this.state = { data: [] }
}
```

Заметим, что мы заменили *gists* внутри состояния на *data*, так как мы хотим создать универсальный HoC и нам незачем привязываться к конкретному названию (прим. пер. однако он будет работать только со списками, я бы заменил пустой массив на null, но кто я такой чтобы идти против творца).

Аналогично исходному компоненту, мы инициируем загрузку данных внутри метода *componentDidMount* и сохраняем внутри состояния после успешной загрузки:

```
componentDidMount() {
  fetch(url)
    .then(response => response.json())
    .then(data => this.setState({ data }))
}
```

Важный нюанс, url не забит гвоздями внутри этого компонента, а приходит как параметр HoC. Это основа переиспользуемости этого компонента внутри приложения.

И в конце мы отображаем полученный в аргументах компонент, передавая ему все параметры и новые данные:

```
render() {
  return <Component {...this.props} {...this.state} />
}
```

Собственно, HoC готов. Теперь мы можем обернуть им любой компонент, чтобы передать ему данные из любого сетевого сервиса.

Давайте посмотрим, как сделать это.

Для начала создадим глупый компонент, который получает данные и отображает их аналогично изначальному компоненту:

```
const List = ({ data: gists }) => (
  <ul>
    {gists.map(gist => (
      <li key={gist.id}>{gist.description}</li>
    ))}
  </ul>
)

List.propTypes = {
  data: React.PropTypes.array,
}
```

Мы можем использовать компонент-функцию, так как мы не собираемся хранить внутри компонента какие-либо данные или определять обработчики событий.

Параметр, который мы получаем из HoC, называется *data*, что не очень удобно для использования внутри компонента, но мы можем без проблем его переименовать благодаря возможностям ES2015.

Теперь мы можем посмотреть, как мы можем воспользоваться нашим HoC *withData*, для того, чтобы передать данные новому компоненту *List*.

Благодаря частичному использованию функции мы можем сначала создать HoC с нужным url, а потом использовать для любого компонента:

```
const withGists = withData(
  'https://api.github.com/users/gaearon/gists'
)
```

И в конце концов мы можем обернуть им наш новый компонент, чтобы создать новый:

```
const ListWithGists = withGists(List)
```

Теперь мы можем добавить расширенный компонент в любое место приложения, и он будет работать.

Наш HoC *withData* великолепен, но он может загружать данные только из статических url, когда в реальности загрузка данных может зависеть от различных параметров, которые бы можно было передать через *props*.

К сожалению, *props* не доступны в момент создания HoC, но они доступны внутри метода *componentDidMount* в момент вызова сетевого запроса.

Что мы можем сделать, так это научить работать наш HoC с двумя типами URL: строкой, как это работает сейчас, и функцией, которая будет создавать url в зависимости от пришедших параметров.

Для того, чтобы сделать это, достаточно поправить метод *componentDidMount*:

```
componentDidMount() {
  const endpoint = typeof url === 'function'
    ? url(this.props)
    : url

  fetch(endpoint)
    .then(response => response.json())
    .then(data => this.setState({ data }))
}
```

Таким образом, если к нам пришла строка, то мы используем его как раньше, если же функция, то мы передаем ей параметры, чтобы получить url.

Мы можем использовать обновленный HoC следующим образом:

```
const withGists = withData(
  props => `https://api.github.com/users/${props.username}/
    gists`
)
```

И имя пользователя мы можем передать через параметры компонента *ListWithGists*:

```
<ListWithGists username="gaearon" />
```

1.3 React-refetch

1.4 Summary