РАЗРАБОТКА ВЕБ-САЙТОВ*, REACTJS*, БЛОГ КОМПАНИИ RUVDS.COM

Шаблоны проектирования в React

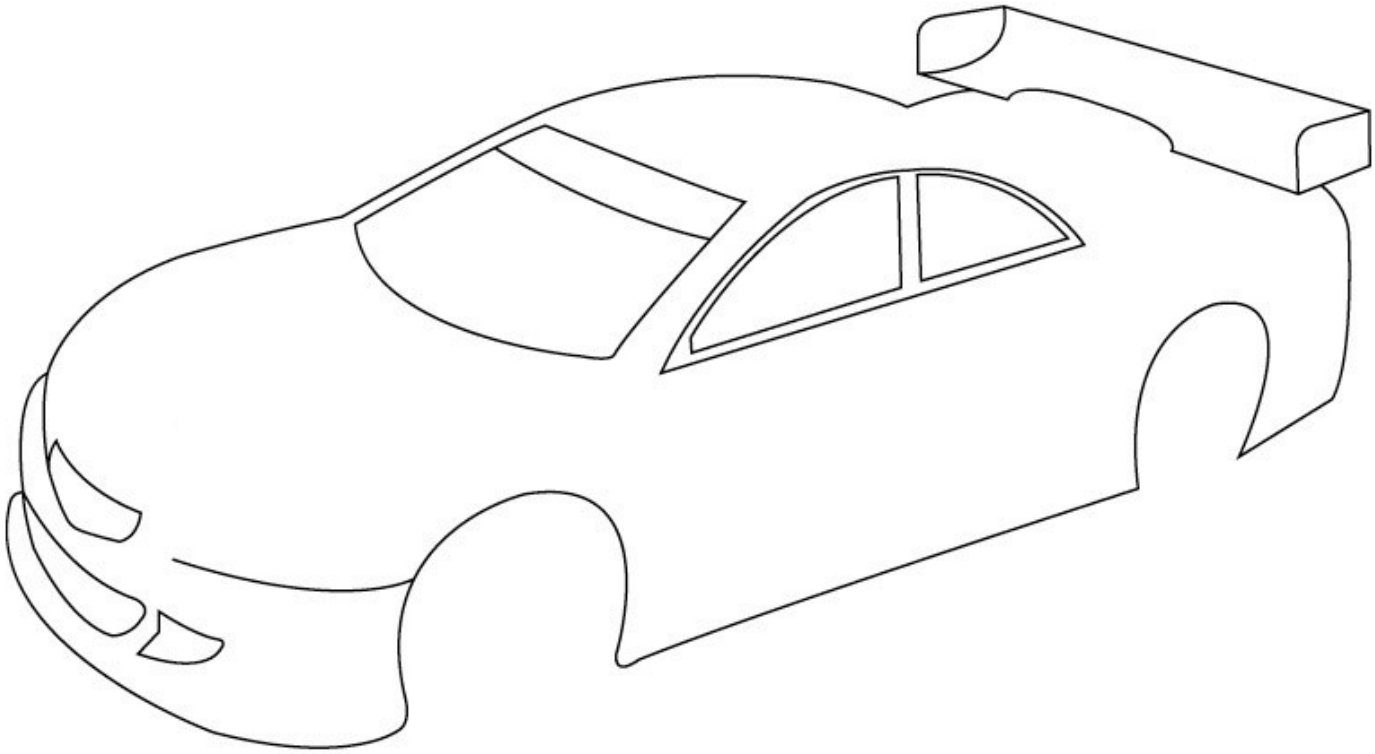
ПЕРЕВОД

ru_vds 16 февраля в 11:35 11k

Оригинал: Alex Moldovan

Шаблоны проектирования, которые возникли и развились в экосистеме React за время её существования, улучшают читабельность и чистоту кода, облегчают повторное использование компонентов.

Автор этого материала говорит, что начал работать с React около трёх лет назад. В то время ещё не было устоявшихся практик, изучая которые и следуя которым можно было бы улучшить качество своих разработок.



Сообществу React понадобилось около двух лет для того, чтобы выработать несколько идей, которые теперь стали популярными. Тут можно отметить переход с `React.createClass` к классам ES6 и к чистым функциональным компонентам, отказ от миксинов и упрощение API. Теперь, учитывая то, что число React-разработчиков постоянно растёт, то, что в развитие этого проекта вкладываются серьёзные силы, можно наблюдать эволюцию нескольких интересных шаблонов проектирования. Этим шаблонам и посвящён данный материал.

Условный рендеринг

Начнём с условного рендеринга (conditional rendering). Мне доводилось видеть следующий сценарий во множестве проектов. Речь идёт о том, что работая с React и JSX, разработчики всё ещё склонны размышлять об этих технологиях в терминах HTML и

JavaScript. В результате, вполне естественно то, что условную логику отделяют от возвращаемого кода.

```
const condition = true;

const App = () => {
  const innerContent = condition ? (
    <div>
      <h2><font color="#3AC1EF">Show me</font></h2>
      <p>Description</p>
    </div>
  ) : null;

  return (
    <div>
      <h1>This is always visible</h1>
      { innerContent }
    </div>
  );
};
```

Подобные конструкции, в которых, в начале каждой функции `render()`, имеются условные тернарные операторы, имеют свойство быстро выходить из-под контроля. Для того, чтобы понять, будет ли выводиться тот или иной элемент, постоянно нужно просматривать код функции.

В качестве альтернативы можете попробовать следующий шаблон, задействующий особенности языка.

```
const condition = true;

const App = () => (
  <div>
    <h1>This is always visible</h1>
    {
      condition && (
        <div>
```

```
            <h2><font color="#3AC1EF">Show me</font></h2>
            <p>Description</p>
          </div>
        )
      }
    </div>
);
```

Если `condition` является `false`, то до вычисления второго операнда оператора `&&` система не доходит. Если `true`, тогда второй операнд, то есть, JSX-код, который мы хотим рендерить, возвращается.

Это позволяет смешивать логику пользовательского интерфейса с описанием интерфейсных элементов, используя декларативный подход. При этом JSX стоит воспринимать так, как будто это неотъемлемая часть кода. В конечном итоге, речь идёт об обычном JavaScript.

Передача свойств вниз по дереву компонентов

Наш следующий шаблон проектирования — это передача свойств вниз по дереву компонентов (passing down props). Когда приложение растёт и развивается, в итоге оказывается, что оно состоит из небольших компонентов, действующих как контейнеры для других компонентов. В результате через компоненты приходится передавать большие объёмы свойств, предназначенных для потомков этих компонентов и не используемых компонентами-родителями. В подобной ситуации можно прибегнуть к деструктурированию свойств, к применению

оператора расширения.

```
const Details = ( { name, language } ) => (
  <div>
    <p>{ name } works with { language }</p>
  </div>
);

const Layout = ( { title, ...props } ) => (
  <div>
    <h1>{ title }</h1>
    <Details { ...props } />
  </div>
);

const App = () => (
  <Layout
    title="I'm here to stay"
    language="JavaScript"
    name="Alex"
  />
);
```

В этом примере можно изменить свойства, необходимые для `Details` и обеспечить отсутствие ссылок на эти свойства в нескольких компонентах.

Деструктурирование свойств

Поговорим о шаблоне деструктурирования свойств (destructuring props). Со временем приложение меняется, то же самое происходит и с компонентами. Компонент, написанный пару лет назад, может использовать состояние, но в текущих условиях он может быть преобразован в компонент без состояния. Часто можно видеть и обратную ситуацию.

Задействуя возможности деструктурирования свойств, можно использовать полезный приём, который я применяю для того, чтобы, в долгосрочной перспективе, облегчить себе работу над проектом. Свойства можно деструктурировать для компонентов обоих типов.

```
const Details = ( { name, language } ) => (
  <div>
    <p>{ name } works with { language }</p>
  </div>
);

class Details extends React.Component {
  render() {
    const { name, language } = this.props;
    return (
      <div>
        <p>{ name } works with { language }</p>
      </div>
    )
  }
}
```

Обратите внимание на то, что строки 2-4 и 11-13 (то есть, теги `<div>`) идентичны. Этот шаблон облегчает трансформирование компонентов. Кроме того, при таком подходе ограничивается использование `this` внутри компонента.

Шаблон «провайдер»

Шаблон проектирования «провайдер» (provider) относится к возможностям React, которые появились сравнительно недавно. Выше мы разбирали пример, в котором свойства нужно отправлять потомкам некоего компонента. Эта операция

усложняется при росте числа компонентов-получателей свойств. Как быть, если, скажем, свойства надо передать пятнадцати компонентам? В подобной ситуации полезно будет воспользоваться API React Context. Нельзя сказать, что эта возможность React пригодится в любой ситуации, но тогда, когда это нужно, она оказывается очень кстати.

Тут надо сказать, что о появлении у `Context` нового API, содержащего реализацию шаблона проектирования «провайдер» было объявлено сравнительно недавно. Этот шаблон должен быть знаком тем, кто использует нечто вроде React Redux или Apollo. Для того чтобы разобраться с новым API, используя существующие возможности, можете поэкспериментировать с этим кодом.

В рассматриваемой нами модели компонент верхнего уровня называется провайдером (provider). Он записывает в контекст какие-то значения. Компонент-потомок, называемый потребителем (consumer), берёт эти значения из контекста.

Пока синтаксис работы с контекстом выглядит немного странно, однако в следующих версиях React ожидается реализация именно этого шаблона.

Компоненты высшего порядка

Разговор о шаблоне «компонент высшего порядка» (High Order Component, HOC) стоит начать с идеи повторного использования кода. Вместе с отказом от старой фабричной функции

`React.createElement()`, команда React отказалась и от поддержки миксинов. Они были, до некоторой степени, стандартным подходом к композиции компонентов через обычную композицию объектов. Компоненты высшего порядка теперь предназначены для удовлетворения потребностей в повторном использовании функционала множеством компонентов.

Компонент высшего порядка — это функция, которая принимает входной компонент и возвращает расширенную или изменённую версию этого компонента. У того, что мы тут называем «компоненты высшего порядка», есть много названий, я предпочитаю воспринимать их как декораторы.

Если вы используете Redux, вы узнаете компонент высшего порядка в функции `connect`, которая, принимая компонент, добавляет к нему некие свойства.

Реализуем простой компонент высшего порядка, который может добавлять свойства к существующим компонентам.

```
const withProps = ( newProps ) => ( WrappedComponent ) => {
  const ModifiedComponent = ( ownProps ) => ( // модифицированная
версия компонента
    <WrappedComponent { ...ownProps } { ...newProps } /> // исходные
свойства + новые свойства
  );

  return ModifiedComponent;
};

const Details = ( { name, title, language } ) => (
  <div>
    <h1>{ title }</h1>
    <p>{ name } works with { language }</p>
```

```
  </div>
);

const newProps = { name: "Alex" }; // это добавлено компонентом
высшего порядка
const ModifiedDetails = withProps( newProps )( Details ); //
компонент высшего порядка каррирован для улучшения читабельности

const App = () => (
  <ModifiedDetails
    title="I'm here to stay"
    language="JavaScript"
  />
);
```

Если вам нравится функциональное программирование, то вам придётся по душе и работа с компонентами высшего порядка. Тут можно вспомнить о Recompose — отличном пакете, дающем в распоряжение разработчика такие компоненты высшего порядка, как `withProps`, `withContext`, `lifecycle` и так далее.

Взглянем на пример повторного использования функционала.

```
function withAuthentication(WrappedComponent) {
  const ModifiedComponent = (props) => {
    if (!props.isAuthenticated) {
      return <Redirect to="/login" />;
    }

    return (<WrappedComponent { ...props } />);
  };

  const mapStateToProps = (state) => ({
    isAuthenticated: state.session.isAuthenticated
  });

  return connect(mapStateToProps)(ModifiedComponent);
}
```

Обратите внимание на то, что `withAuthentication` можно использовать в ситуациях, когда в маршруте надо вывести данные, не предназначенные для чужих глаз. Эти данные будут доступны только пользователям, вошедшим в систему.

Тут приведён пример сквозной функциональности, реализованной в одном месте и подходящей для повторного использования во всём приложении.

Однако у компонентов высшего порядка есть и недостатки. Каждый такой компонент приводит к созданию дополнительного компонента React в DOM/vDOM. Это, по мере роста приложения, может вести к потенциальным проблемам с производительностью.

Некоторые дополнительные проблемы с компонентами высшего порядка описаны в этом материале. Тут, в частности, предлагается заменить компоненты высшего порядка шаблоном, который мы сейчас рассмотрим.

Шаблон «render props»

Шаблон «render props», или, как его ещё называют, «функция как потомок», позволяет достичь того же самого, что достижимо с помощью компонентов высшего порядка. Эти шаблоны взаимозаменяемы. Сравнивая их, я не могу отдать абсолютное предпочтение одному из них. Оба шаблона используются для того, чтобы сделать код чище и улучшить возможности его повторного использования.

Если в двух словах, то идея использования шаблона render props заключается в передаче управления вашей функцией рендеринга другому компоненту, который затем возвращает управление через свойство, являющееся функцией. Некоторые предпочитают применять для достижения того же эффекта динамические свойства, некоторые просто используют `this.props.children`.

Пожалуй, лучше всего будет проиллюстрировать всё это на примере.

```
class ScrollPosition extends React.Component {
  constructor( ) {
    super( );
    this.state = { position: 0 };
    this.updatePosition = this.updatePosition.bind(this);
  }

  componentDidMount( ) {
    window.addEventListener( "scroll", this.updatePosition );
  }

  updatePosition( ) {
    this.setState( { position: window.pageYOffset } )
  }

  render( ) {
    return this.props.children( this.state.position )
  }
}

const App = () => (
  <div>
    <ScrollPosition>
      { ( position ) => (
        <div>
          <h1>Hello World</h1>
          <p>You are at { position }</p>
        </div>
      ) }
```

```
    </ScrollPosition>
  </div>
);
```

Здесь, в качестве свойства, используется `children`. В компонент `<ScrollPosition>` мы отправляем функцию, которая принимает `position` как параметр.

Шаблон render props может быть использован в ситуациях, где нужна некая подходящая для повторного использования логика внутри компонента, при этом данный компонент не планируется оборачивать в компонент высшего порядка. Примеры использования шаблона render props можно найти в библиотеке React-Motion. Вот пример создания компонента `Fetch`, иллюстрирующий использование асинхронных потоков с шаблоном render props.

В итоге хочется отметить, что с одним и тем же компонентом можно использовать несколько функций-потомков. Шаблон render props даёт разработчику неограниченные возможности по композиции и повторному использованию функционала.

Итоги

В этом материале мы рассмотрели несколько шаблонов React, некоторые из которых существуют уже давно, а некоторые появились сравнительно недавно. Полагаем, знакомство с шаблонами полезно, так как они предлагают разработчику надёжные способы решения типичных задач, использование которых экономит время и улучшает код.

Уважаемые читатели! Какими шаблонами вы пользуетесь в ваших проектах, созданных на базе React?

Комментарии (4)

Похожие публикации

Анализ производительности React 16 приложений с помощью инструментов разработчика Chrome

4

ПЕРЕВОД

ru_vds • 5 декабря 2017 в 12:12

Основы React: всё, что нужно знать для начала работы

60

ПЕРЕВОД

ru_vds • 23 ноября 2017 в 13:25

Отладка React-приложений в VS Code

10

ПЕРЕВОД

ru_vds • 9 ноября 2017 в 12:20

Популярное за сутки

Наташа — библиотека для извлечения структурированной информации из текстов на русском языке

14

alexkuku • вчера в 16:12

Unit-тестирование скриншотами: преодолеваем звуковой барьер. Расшифровка доклада

4

lahmatiy • вчера в 13:05

Люди не хотят чего-то действительно нового — они хотят привычное, но сделанное иначе

25

ПЕРЕВОД

Smileek • вчера в 10:32

Руководство по SEO JavaScript-сайтов. Часть 2. Проблемы, эксперименты и рекомендации

2

ПЕРЕВОД

ru_vds • вчера в 12:04

Как адаптировать игру на Unity под iPhone X к апрелю

0

P1CACHU • вчера в 16:13

Лучшее на Geektimes

Стивен Хокинг, автор «Краткой истории времени», умер на 77 году жизни 33

HostingManager • вчера в 13:49

Обзор рынка моноколес 2018 70

lozga • вчера в 06:58

«Битва за Telegram»: 35 пользователей подали в суд на ФСБ 40

alizar • вчера в 15:14

Стивен Хокинг и его работа — что дал ученый человечеству? 8

marks • вчера в 14:46

Sunlike — светодиодный свет нового поколения 17

AlexeyNadezhin • вчера в 20:32