

lesson1 React组件化

lesson1 React组件化

课堂目标

资源

知识点

快速开始

组件化优点

组件跨层级通信 - Context

Context API

`React.createContext`

`Context.Provider`

`Class.contextType`

`Context.Consumer`

`useContext`

使用Context

`pages/ContextTypePage.js`

`pages/ConsumerPage.js`

消费多个Context

`pages/UseContextPage`

注意事项

总结

表单组件设计与实现

antd表单使用

antd3表单组件设计思路

antd4表单组件实现

实现my-rc-field-form

实现Form/index

实现Form

实现FieldContext

实现useForm

实现Field

回顾

作业

下节课内容

课堂目标

掌握组件化开发中多种实现技术

1. 掌握context，跨层级传递
2. 掌握高阶组件
3. 了解组件化概念，能设计并实现自己需要的组件

资源

1. [create-react-app](#)
2. [HOC](#)
3. [ant design](#)
4. [课堂代码地址](#)

知识点

快速开始

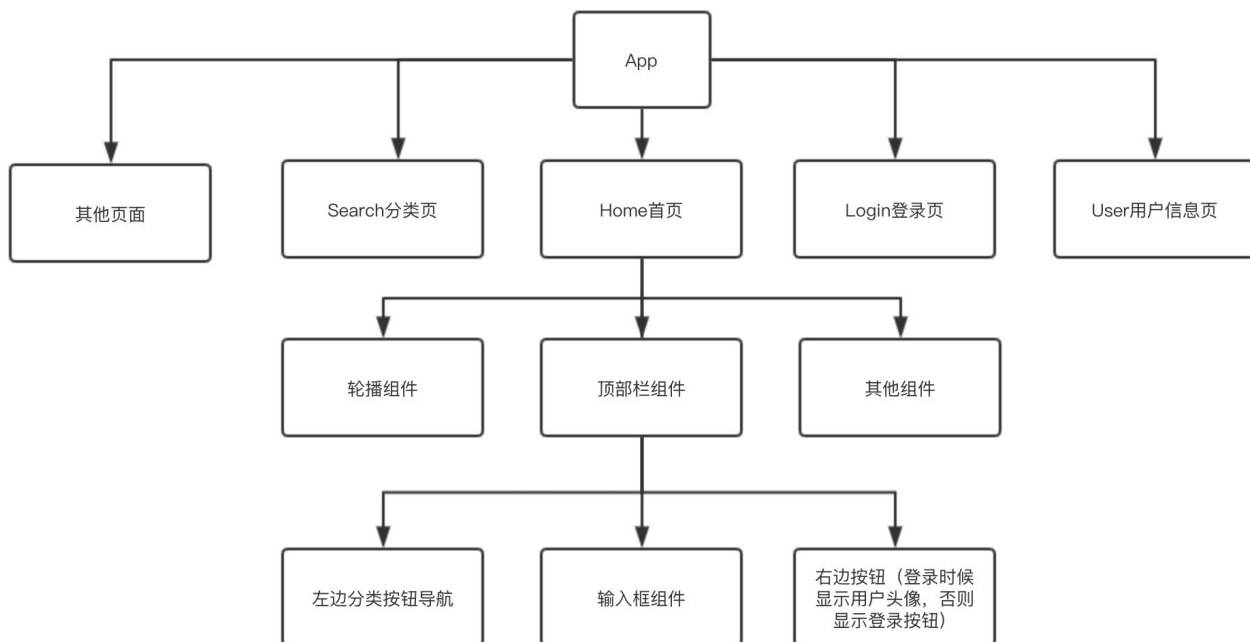
(<https://www.html.cn/create-react-app/docs/getting-started/>)

```
npx create-react-app lesson1  
  
cd lesson1  
  
yarn start
```

组件化优点

1. 增强代码重用性，提高开发效率
2. 简化调试步骤，提升整个项目的可维护性
3. 便于协同开发
4. 注意点：降低耦合性

组件跨层级通信 - Context



在一个典型的 React 应用中，数据是通过 props 属性自上而下（由父及子）进行传递的，但这种做法对于某些类型的属性而言是极其繁琐的（例如：地区偏好，UI 主题），这些属性是应用程序中许多组件都需要的。Context 提供了一种在组件之间共享此类值的方式，而不必显式地通过组件树的逐层传递 props。

React中使用Context实现祖代组件向后代组件跨层级传值。Vue中的provide & inject来源于Context。

Context API

`React.createContext`

创建一个 Context 对象。当 React 渲染一个订阅了这个 Context 对象的组件，这个组件会从组件树中离自身最近的那个匹配的 `Provider` 中读取到当前的 context 值。

`Context.Provider`

Provider 接收一个 `value` 属性，传递给消费组件，允许消费组件订阅 context 的变化。一个 Provider 可以和多个消费组件有对应关系。多个 Provider 也可以嵌套使用，里层的会覆盖外层的数据。

当 Provider 的 `value` 值发生变化时，它内部的所有消费组件都会重新渲染。Provider 及其内部 consumer 组件都不受制于 `shouldComponentUpdate` 函数，因此当 consumer 组件在其祖先组件退出更新的情况下也能更新。

Class.contextType

挂载在 class 上的 `contextType` 属性会被重赋值为一个由 `React.createContext()` 创建的 Context 对象。这能让你使用 `this.context` 来消费最近 Context 上的那个值。你可以在任何生命周期中访问到它，包括 render 函数中。

你只通过该 API 订阅单一 context。

Context.Consumer

这里，React 组件也可以订阅到 context 变更。这能让你在[函数式组件](#)中完成订阅 context。

这个函数接收当前的 context 值，返回一个 React 节点。传递给函数的 `value` 值等同于往上组件树离这个 context 最近的 Provider 提供的 `value` 值。如果没有对应的 Provider，`value` 参数等同于传递给 `createContext()` 的 `defaultValue`。

useContext

接收一个 context 对象（`React.createContext` 的返回值）并返回该 context 的当前值。当前的 context 值由上层组件中距离当前组件最近的 `<MyContext.Provider>` 的 `value` prop 决定。只能用在 function 组件中。

使用Context

创建Context => 获取Provider和Consumer => Provider提供值 => Consumer消费值

范例：共享主题色

```
import React, {Component} from "react";
import ContextTypePage from "../ContextTypePage";
import {ThemeContext, UserContext} from "../Context";
import useContextPage from "../UseContextPage";
import ConsumerPage from "../ConsumerPage";

export default class ContextPage extends Component {
  constructor(props) {
    super(props);
    this.state = {
      theme: {
        themeColor: "red"
      },
      user: {
        name: "xiaoming"
      }
    };
  }
}
```

```

changeColor = () => {
  const {themeColor} = this.state.theme;
  this.setState({
    theme: {
      themeColor: themeColor === "red" ? "green" : "red"
    }
  });
};

render() {
  const {theme, user} = this.state;
  return (
    <div>
      <h3>ContextPage</h3>
      <button onClick={this.changeColor}>change color</button>
      <ThemeContext.Provider value={theme}>
        <ContextTypePage />
        <UserContext.Provider value={user}>
          <UseContextPage />
          <ConsumerPage />
        </UserContext.Provider>
      </ThemeContext.Provider>
      <ContextTypePage />
    </div>
  );
}
}

```

//Context.js

```

import React from "react";

export const ThemeContext = React.createContext({themeColor: "pink"});

export const UserContext = React.createContext();

```

pages/ContextTypePage.js

```

import React, {Component} from "react";
import {ThemeContext} from "../Context";

export default class ContextTypePage extends Component {
  static contextType = ThemeContext;
  render() {

```

```

    const {themeColor} = this.context;
    return (
      <div className="border">
        <h3 className={themeColor}>ContextTypePage</h3>
      </div>
    );
  }
}

```

pages/ConsumerPage.js

```

import React, {Component} from "react";
import {ThemeContext, UserContext} from "../Context";

export default class ConsumerPage extends Component {
  render() {
    return (
      <div className="border">
        <ThemeContext.Consumer>
          {themeContext => (
            <>
              <h3 className={themeContext.themeColor}>ConsumerPage</h3>
              <UserContext.Consumer>
                {userContext => <HandleUserContext {...userContext} />}
              </UserContext.Consumer>
            </>
          )}
        </ThemeContext.Consumer>
      </div>
    );
  }
}

function HandleUserContext(userCtx) {
  return <div>{userCtx.name}</div>;
}

```

消费多个Context

```

<ThemeProvider value={theme}>
  <ContextTypePage />
  <ConsumerPage />

  {/*多个Context */}
  <UserProvider value={user}>
    <MultipleContextsPage />
  </UserProvider>
</ThemeProvider>

```

如果两个或者更多的 context 值经常被一起使用，那你可能要考虑一下另外创建你自己的渲染组件，以提供这些值。

pages/UseContextPage

```

import React, {useState, useEffect, useContext} from "react";
import {ThemeContext, UserContext} from "../Context";

export default function UseContextPage(props) {
  const themeContext = useContext(ThemeContext);
  const {themeColor} = themeContext;
  const userContext = useContext(UserContext);
  return (
    <div className="border">
      <h3 className={themeColor}>UseContextPage</h3>
      <p>{userContext.name}</p>
    </div>
  );
}

```

注意事项

因为 context 会使用参考标识（reference identity）来决定何时进行渲染，这里可能会有一些陷阱，当 provider 的父组件进行重渲染时，可能会在 consumers 组件中触发意外的渲染。举个例子，当每一次 Provider 重渲染时，以下的代码会重渲染所有下面的 consumers 组件，因为 `value` 属性总是被赋值为新的对象：

```
class App extends React.Component {
  render() {
    return (
      <Provider value={{something: 'something'}}>
        <Toolbar />
      </Provider>
    );
  }
}
```

为了防止这种情况，将 value 状态提升到父节点的 state 里：

```
class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      value: {something: 'something'},
    };
  }

  render() {
    return (
      <Provider value={this.state.value}>
        <Toolbar />
      </Provider>
    );
  }
}
```

总结

在React的官方文档中，[Context](#) 被归类为高级部分(Advanced)，属于React的高级API，建议不要滥用。

后面我们要学习到的react-redux的 `<Provider />`，就是通过 Context 提供一个全局态的 store，路由组件react-router通过 Context 管理路由状态等等。在React组件开发中，如果用好 Context，可以让你的组件变得强大，而且灵活。

表单组件设计与实现

antd表单使用

实现用户名密码登录，并实现校验。

//FormPage.js

class

```
import React, {Component, useEffect} from "react";
import {Form, Input, Button} from "antd";

const FormItem = Form.Item;

const nameRules = {required: true, message: "请输入姓名! "};
const passworRules = {required: true, message: "请输入密码! "};

export default class AntdFormPage extends Component {
  formRef = React.createRef();

  componentDidMount() {
    this.formRef.current.setFieldsValue({name: "default"});
  }

  onReset = () => {
    this.formRef.current.resetFields();
  };

  onFinish = val => {
    console.log("onFinish", val); //sy-log
  };

  onFinishFailed = val => {
    console.log("onFinishFailed", val); //sy-log
  };

  render() {
    console.log("AntdFormPage render", this.formRef.current); //sy-log
    return (
      <div>
        <h3>AntdFormPage</h3>
        <Form
          ref={this.formRef}
          onFinish={this.onFinish}
          onFinishFailed={this.onFinishFailed}
          onReset={this.onReset}>
          <FormItem label="姓名" name="name" rules={[nameRules]}>
            <Input placeholder="name input placeholder" />
          </FormItem>
          <FormItem label="密码" name="password" rules={[passworRules]}>
```

```

        <Input placeholder="password input placeholder" />
      </FormItem>
      <FormItem>
        <Button type="primary" size="large" htmlType="submit">
          Submit
        </Button>
      </FormItem>
      <FormItem>
        <Button type="default" size="large" htmlType="reset">
          Reset
        </Button>
      </FormItem>
    </Form>
  </div>
);
}
}

```

function实现:

注意 `useForm` 是React Hooks的实现, 只能用于函数组件。

```

export default function AntdFormPage(props) {
  const [form] = Form.useForm();

  const onFinish = val => {
    console.log("onFinish", val); //sy-log
  };
  const onFinishFailed = val => {
    console.log("onFinishFailed", val); //sy-log
  };
  const onReset = () => {
    form.resetFields();
  };

  useEffect(() => {
    form.setFieldsValue({name: "default"});
  }, []);

  return (
    <Form
      form={form}
      onFinish={onFinish}
      onFinishFailed={onFinishFailed}
      onReset={onReset}>
      <FormItem label="姓名" name="name" rules={[nameRules]}>
        <Input placeholder="name input placeholder" />
      </FormItem>
    </Form>
  );
}

```

```

    </FormItem>
    <FormItem label="密码" name="password" rules={[passworRules]}>
      <Input placeholder="password input placeholder" />
    </FormItem>
    <FormItem>
      <Button type="primary" size="large" htmlType="submit">
        Submit
      </Button>
    </FormItem>
    <FormItem>
      <Button type="default" size="large" htmlType="reset">
        Reset
      </Button>
    </FormItem>
  </Form>
);
}

```

antd3表单组件设计思路

- 表单组件要求实现数据收集、校验、提交等特性，可通过高阶组件扩展
- 高阶组件给表单组件传递一个input组件包装函数接管其输入事件并统一管理表单数据
- 高阶组件给表单组件传递一个校验函数使其具备数据校验功能

但是antd3的设计有个问题，就是局部变化会引起整体变化，antd4改进了这个问题。

antd4表单组件实现

antd4的表单基于rc-field-form, [github源码地址](#)。

安装rc-field-form, `yarn add rc-field-form`。

使用useForm，仅限function：

```

import React, {Component, useEffect} from "react";
// import Form, {Field} from "rc-field-form";
import Form, {Field} from "../components/my-rc-field-form/";
import Input from "../components/Input";

const nameRules = {required: true, message: "请输入姓名! "};
const passworRules = {required: true, message: "请输入密码! "};

export default function MyRCFieldForm(props) {
  const [form] = Form.useForm();

  const onFinish = val => {
    console.log("onFinish", val); //sy-log
  }

```

```

};

// 表单校验失败执行
const onFinishFailed = val => {
  console.log("onFinishFailed", val); //sy-log
};

useEffect(() => {
  console.log("form", form); //sy-log
  form.setFieldsValue({username: "default"});
}, []);

return (
  <div>
    <h3>MyRCFieldForm</h3>
    <Form form={form} onFinish={onFinish} onFinishFailed={onFinishFailed}>
      <Field name="username" rules={[nameRules]}>
        <Input placeholder="input UR Username" />
      </Field>
      <Field name="password" rules={[passwordRules]}>
        <Input placeholder="input UR Password" />
      </Field>
      <button>Submit</button>
    </Form>
  </div>
);
}

```

class实现:

```

export default class MyRCFieldForm extends Component {
  formRef = React.createRef();
  componentDidMount() {
    console.log("form", this.formRef.current); //sy-log
    this.formRef.current.setFieldsValue({username: "default"});
  }

  onFinish = val => {
    console.log("onFinish", val); //sy-log
  };

  // 表单校验失败执行
  onFinishFailed = val => {
    console.log("onFinishFailed", val); //sy-log
  };

  render() {
    return (

```

```

<div>
  <h3>MyRCFieldForm</h3>
  <Form
    ref={this.formRef}
    onFinish={this.onFinish}
    onFinishFailed={this.onFinishFailed}>
    <Field name="username" rules={[nameRules]}>
      <Input placeholder="Username" />
    </Field>
    <Field name="password" rules={[passwordRules]}>
      <Input placeholder="Password" />
    </Field>
    <button>Submit</button>
  </Form>
</div>
);
}
}

```

实现my-rc-field-form

实现Form/index

```

import React from "react";
import _Form from "./Form";
import Field from "./Field";
import useForm from "./useForm";

const Form = React.forwardRef(_Form);
Form.Field = Field;
Form.useForm = useForm;

export {Field, useForm};
export default Form;

```

实现Form

```

import React from "react";
import useForm from "./useForm";
import FieldContext from "./FieldContext";

export default function Form({children, onFinish, onFinishFailed, form}, ref) {
  const [formInstance] = useForm(form);

  React.useImperativeHandle(ref, () => formInstance);

```

```

formInstance.setCallback({
  onFinish,
  onFinishFailed
});
return (
  <form
    onSubmit={event => {
      event.preventDefault();
      event.stopPropagation();
      formInstance.submit();
    }}>
    <FieldContext.Provider value={formInstance}>
      {children}
    </FieldContext.Provider>
  </form>
);
}

```

实现FieldContext

```

import React from "react";

const warnFunc = () => {
  console.log("-----err-----"); //sy-log
};

const FieldContext = React.createContext({
  registerField: warnFunc,
  setFieldsValue: warnFunc,
  getFieldValue: warnFunc,
  getFieldsValue: warnFunc,
  submit: warnFunc
});

export default FieldContext;

```

实现useForm

```

import React from "react";

class FormStore {
  constructor() {
    this.store = {}; //存储state数据，以key value形式存储
    this.fieldEntities = []; // 存储field实例
  }
}

```

```

    this.callbacks = {}; // 存储回调
  }

  setCallbacks = newCallbacks => {
    this.callbacks = {
      ...this.callbacks,
      ...newCallbacks
    };
  };

  // * 注册和取消注册 订阅和取消订阅 一定要成对出现
  registerFieldEntity = entity => {
    //注册
    this.fieldEntities.push(entity);
    return () => {
      // 取消注册
      this.fieldEntities = this.fieldEntities.filter(item => item !== entity);
      delete this.store[entity.props.name];
    };
  };

  getFieldValue = name => {
    return this.store[name];
  };

  getFieldsValue = () => {
    return {...this.store};
  };

  // set函数, newStore可以定义多个state
  setFieldsValue = newStore => {
    // 合并
    // step1: 数据更新
    this.store = {
      ...this.store,
      ...newStore
    };
    // step2: 组件也要更新
    // 对应组件才需要更新, 需要加筛选
    this.fieldEntities.forEach(entity => {
      Object.keys(newStore).forEach(k => {
        if (k === entity.props.name) {
          entity.onStoreChange();
        }
      });
    });
  };

  valide = callback => {

```

```

    let err = [];
    // todo 校验 作业

    return err;
  };

  submit = () => {
    const {onFinish, onFinishFailed} = this.callbacks;
    const err = this.validate();
    // 先校验this.store
    // 校验通过 执行onFinish
    // 校验失败 执行 onFinishFailed

    if (err.length === 0) {
      // 成功
      onFinish(this.getFieldsValue());
    } else {
      // 失败
      onFinishFailed(err, this.getFieldsValue());
    }
  };

  getForm = () => {
    return {
      getFieldValue: this.getFieldValue,
      getFieldsValue: this.getFieldsValue,
      setFieldsValue: this.setFieldsValue,
      registerFieldEntity: this.registerFieldEntity,
      setCallbacks: this.setCallbacks,
      submit: this.submit
    };
  };
}

export default function useForm(form) {
  const formRef = React.useRef();
  if (!formRef.current) {
    if (form) {
      formRef.current = form;
    } else {
      const formStore = new FormStore();
      formRef.current = formStore.getForm();
    }
  }

  return [formRef.current];
}

```


实现Field

```
import React, {Component} from "react";
import FieldContext from "../FieldContext";

export default class Field extends Component {
  static contextType = FieldContext;

  componentDidMount() {
    const {registerField} = this.context;
    this.cancelRegisterField = registerField(this);
  }

  componentWillUnmount() {
    if (this.cancelRegisterField) {
      this.cancelRegisterField();
    }
  }

  onStoreChange = () => {
    this.forceUpdate();
  };

  getControlled = () => {
    const {name} = this.props;
    const {getFileValue, setFieldsValue} = this.context;

    return {
      value: getFileValue(name), //取数据
      onChange: event => {
        // 存数据
        const newValue = event.target.value;
        setFieldsValue({[name]: newValue});
      }
    };
  };

  render() {
    console.log("field render"); //sy-log
    const {children} = this.props;

    const returnChildNode = React.cloneElement(children, this.getControlled());
    return returnChildNode;
  }
}
```

回顾

lesson1 React组件化

课堂目标

资源

知识点

快速开始

组件化优点

组件跨层级通信 - Context

Context API

`React.createContext`

`Context.Provider`

`Class.contextType`

`Context.Consumer`

`useContext`

使用Context

pages/ContextTypePage.js

pages/ConsumerPage.js

消费多个Context

pages/UseContextPage

注意事项

总结

表单组件设计与实现

antd表单使用

antd3表单组件设计思路

antd4表单组件实现

实现my-rc-field-form

实现Form/index

实现Form

实现FieldContext

实现useForm

实现Field

回顾

作业

下节课内容

作业

1. 实现validate函数，使其能够实现基础校验。注意事项，交的代码片段一定要有validate函数的实现!!!

下节课内容

lesson2: redux

