基础知识

State 是当前组件的数据，State 与 props 类似，但是 state 是私有的，并且完全受控于当前组件。

- state 是组件的私有数据
- props 是父组件传递过来的数据
- 可以将当前组件的 state 数据，通过 props 传递给子组件

#遍历状态

为了方便理解，可以将状态理解为组件数据

#声明定义

下面在组件 components/User/index.js 中声明用户列表数据

```
export default class List extends Component {
  constructor() {
    super()
    this.state = {
      users: [
        { id: 1, name: "后盾人", age: 18 },
        { id: 2, name: "向军大叔", age: 19 }
      ]
    }
  }
}
...

```

#遍历数据

还是在组件 components/User/index.js 中使用 map 来遍历渲染数据

- map 是 JS 函数需要使用 {} 包裹
- 需要设置 key 值标识唯一性
- 使用 user 属性向子组件 components/User/user.js 中传递 props 数据

```
render() {
    return (
      <div>
```

```
        <table border="1" width="100%">
          <caption>{this.props.children}</caption>
          <thead>
            <tr>
              <th>编号</th>
              <th>姓名</th>
              <th>年龄</th>
            </tr>
            {this.state.users.map(user => {
              return <User key={user.id} user={user} />
            })}
          </thead>
        </table>
      </div>
    )
  }

```

子组件 components/User/user.js 接收 props 并进行设置

```
export default class User extends Component {
  render() {
    return (
      <tr align="center">
        <td>{this.props.user.id}</td>
        <td>{this.props.user.name}</td>
        <td>{this.props.user.age}</td>
      </tr>
    )
  }
}

```

最终渲染效果如下图

使用展开语法来简化传递，在父组件 components/User/index.js 中设置属性是使用点语法展开

- 可以将属性分别独立传递给子组件的 props

```
render() {
```

```
...

  {this.state.users.map(user => {
    return <User key={user.id} {...user} />
  })}
...
}

```

在子组件中直接使用 props 的值即可

```
export default class User extends Component {
  render() {
    return (
      <tr align="center">
        <td>{this.props.id}</td>
        <td>{this.props.name}</td>
        <td>{this.props.age}</td>
      </tr>
    )
  }
}

```

#用户统计

修改 components/User/index.js 组件来添加用户统计，注意类属性使用 className

```
<table border="1" width="100%">
  <caption>{this.props.children}</caption>
  ...
</table>

<div className="count">共计{this.state.users.length}人</div>

```

修改样式文件 components/User/index.css

```
div.count {
```

```
    text-align: center;
    margin-top: 20px;
}

```

最终效果

#修改状态

通过删除用户来体验使用事件修改状态

#修改数件

在 components/User/index.js 组件中添加删除按钮标签

```
<div className="count">共计{this.state.users.length}人</div>
<div className="delAll">
  <button>删除全部</button>
</div>

```

修改样式文件 components/User/index.css

```
div.delAll {
    text-align: center;
    padding-top: 10px;
    margin-top: 20px;
    border-top: solid 1px #ddd;
}

div.delAll button {
    background: #16a085;
    color: white;
    padding: 5px;
    border: none;
    cursor: pointer;
}

```

最终效果如图

#添加事件

为删除按钮绑定事件，并在类中添加事件处理程序，事件处理程序要使用 onClick 形式的属性

- 修改数据要使用 setState 方法完成，不允许直接赋值
- 下例中使用箭头函数定义方法，是为了设置 this 为当前组件对象
- 如果事件方法没有定义箭头函数，调用事件时要使用 this.delAll.bind(this) 来绑定 this

```
export default class List extends Component {
  ...
  this.setState({
  users: []
  })
  ...
}

```

最终效果如图

this.setState 方法也支持传递函数，上一个 state 作为第一个参数 props 做为第二个参数。

```
delAll = () => {
  this.setState((state, props) => ({
    users: []
  }))
}

```

#绑定 THIS

上面的事件处理方法使用箭头函数定义的，就可以很方便的得到 this，这是推荐的做法，我们再看其他一些情况

事件绑定 this

如果事件方法没有定义箭头函数

```
delAll() {
  this.setState(
```

```
      ...
    )
}

```

调用事件时要使用 this.delAll.bind(this) 来绑定 this

```
<button onClick={this.delAll.bind(this)}>删除全部</button>

```

构造函数绑定

也可以在构造函数 constructor 中完成绑定

```
() {
  super()
  this.state = {
    users: [
      { id: 1, name: "后盾人", age: 18 },
      { id: 2, name: "向军大叔", age: 19 }
    ]
  }
  this.delAll = this.delAll.bind(this)
}

```

在标签中使用时就不需要 bind 了

```
<button onClick={this.delAll}>删除全部</button>

```

#异步修改

修改状态是异步操作的，通过下面的代码我们会看可以通过 console 打印出数据，证明删除是异步的

```
delAll = () => {
  this.setState((state, props) => ({
    users: []
  }))
  console.log(this.state.users)
```

```
}

```

真正修改状态后做的可以在 this.setState 方法的第二个函数参数来完成

```
delAll = () => {
  this.setState(
    (state, props) => ({
      users: []
    }),
    () => {
      console.log("状态修改完成后执行")
    }
  )
  console.log(this.state.users)
}

```

上面代码最终打印结果

```
(2) [{…}, {…}]
状态修改完成后执行

```

#组件通信

下面通过添加用户来体验组件的通信

- state 只能单向向下递
- state 只能有所属组件修改
- 子组件修改父组件 state，需要调用父组件修改方法

#表单输入

REACT 对表单的处理需要注意几点

- 表单值要通过 onChange 事件设置
- 表单值要与组件的 state 关联

下面是组件 components/Add/index.js 中的表单设置

```
export default class Add extends Component {
```

```
  constructor() {
    super()
    this.state = {
      value: "后盾人"
    }
  }
  change = e => {
    this.setState({
      value: e.target.value
    })
  }
  render() {
    return (
      <div className="add">
        <input value={this.state.value} onChange={this.change} />
        <button>{this.props.btnTitle}</button>
      </div>
    )
  }
  ...
}

```

#组件调整

组件只能向父组件一层层传递，但我们的添加组件与列表组件是同级，由以下几种方法

1. 在这两个组件上再添加一个父组件，然后将用户列表数据移动到这个组件中
2. 将添加用户组件 Add/index.js 置入到用户列表组件 List/index.js 中做为它的子组件

第二种方式是比较合理，也是省事的方式

修改 App.js 根组件，去掉对 Add/index.js 组件的使用

```
...
<div className="app">
  <List btnTitle="后盾人@向军大叔" />
</div>
...

```

修改 List/index.js 组件并导入 Add/index.js 组件

- 这时 Add/index.js 的组件中的按钮文本，将由 List/index.js 将 App.js 设置的值使用 props 传递下去

```
...
import Add from "../Add"
export default class List extends Component {
 ...
        render() {
    return (
      <div>
        <Add btnTitle={this.props.btnTitle} />
        ...
      </div>
    )
  }
  ...
}

```

最终的页面效果没有什么变化

#状态传递

现在父子组件关系已经确立，可以方便传递 state 了

- 子组件不能直接修改父组件的 state
- 父组件定义方法来让子组件调用
- 父组件将这个方法以 props 的形式传递给子组件

父组件

下面修改 List/index.js 父组件首先定义添加用户的方法

- 本例中包含姓名和年龄，但输入框只有一个，所以要求以**姓名@年龄**来输入

```
add = param => {
 //拆分获取用户名与年龄
  const info = param.split("@")
  const user = {
    id: this.state.users.length + 1,
    name: info[0],
    age: info[1]
```

```
  }
  this.setState({
        //添加新用户
    users: [...this.state.users, user]
  })
}

```

List/index.js 父组件向子组件传递添加用户的方法

```
<Add btnTitle={this.props.btnTitle} add={this.add} />

```

子组件

下面来定义添加用户的子组件

- 定义 handleAdd 方法，当点击按钮时通知父组件添加用户

```
add = e => {
  this.props.add(this.state.value)
}
render() {
  return (
    <div className="add">
      <input value={this.state.value} onChange={this.changeValue} />
      <button onClick={this.add}>{this.props.btnTitle}</button>
    </div>
  )
}

```

现在功能已经完成了

#键盘事件

下面来实现表单中敲回车来添加元素，首先添加键盘事件处理方法 enterAdd

- 回车的 ASCII 码是 13
- 复用按钮事件

```
//回车添加用户
enterAdd = e => e.keyCode === 13 && this.add()

add = () => {
  this.props.add(this.state.value)
  //将表单清空
  this.setState({
    value: ""
  })
}

```

#REF

下面来实现点击按钮后表单获取焦点

- 首先引及函数 createRef
- 创建 ref 对象
- 绑定表单

```
import React, { Component, createRef } from "react"
...

export default class Add extends Component {
  constructor() {
    ...
    this.input = createRef()
  }
  ...
  render() {
    return (
      <div className="add">
        <input
          placeholder="用户名@年龄"
          value={this.state.value}
          onChange={this.changeValue}
          onKeyUp={this.enterAdd}
          ref={this.input}
```

```
        />
        ...
      </div>
    )
  }
  add = () => {
    ...
    //让表单获取焦点
    this.input.current.focus()
  }
  ...
}

```

#删除用户

下面来实现用户删除操作，加深对组件通信的理解

修改 components/user.js 添加按钮表单

```
...
<td>
  <button className="del-user-button">删除</button>
</td>
...

```

添加 components/user.css 文件设置按钮样式

```
.del-user-button {
    background: #d35400;
    color: #fff;
    padding: 5px 10px;
}

```

界面效果如下

下面来实现删除逻辑，首先修改 components/User/index.js 组件，添加删除方法

```

```

```
//根据子组件传递的用户编号删除
del = id => {
  this.setState(state => {
    return {
      users: state.users.filter(user => user.id !== id)
    }
  })
}

```

然后方法以 props 的形式传递给子组件，用于子组件调用该方法来删除用户

```
<thead>
  <tr>
    <th>编号</th>
    <th>姓名</th>
    <th>年龄</th>
    <th>操作</th>
  </tr>
  {this.state.users.map(user => {
    return <User key={user.id} {...user} del={this.del} />
  })}
</thead>

```

最终删除效果如下