

Create App

- 安装Node.js
- `npx create-react-app my-app` 新建应用
- 删除src文件夹下所有内容

<https://reactjs.org/tutorial/tutorial.html#what-is-react>

Overview

React is a declarative, efficient, and flexible JavaScript library for building user interfaces. It lets you compose complex UIs from small and isolated pieces of code called “components”.

React has a few different kinds of components, but we'll start with `React.Component` subclasses:

```
class ShoppingList extends React.Component {
  render() {
    return (
      <div className="shopping-list">
        <h1>Shopping List for {this.props.name}</h1>
        <ul>
          <li>Instagram</li>
          <li>WhatsApp</li>
          <li>Oculus</li>
        </ul>
      </div>
    );
  }
}

// Example usage: <ShoppingList name="Mark" />
```

We use components to tell React what we want to see on the screen. When our data changes, React will efficiently update and re-render our components.

Here, `ShoppingList` is a **React component class**, or **React component type**. A component takes in parameters, called `props` (short for “properties”), and returns a hierarchy of views to display via the `render` method.

The `render` method returns a *description* of what you want to see on the screen. React takes the description and displays the result. In particular, `render` returns a **React element**, which is a lightweight description of what to render. Most React developers use a special syntax called “JSX” which makes these structures easier to write. The `<div />` syntax is transformed at build time to `React.createElement('div')`. The example above is equivalent to:

```
return React.createElement('div', {className: 'shopping-list'},
  React.createElement('h1', /* ... h1 children ... */),
  React.createElement('ul', /* ... ul children ... */),
);
```

[See full expanded version.](#)

JSX comes with the full power of JavaScript. You can put *any* JavaScript expressions within braces inside JSX. Each React element is a JavaScript object that you can store in a variable or pass around in your program.

The `ShoppingList` component above only renders built-in DOM components like `<div />` and ``. But you can compose and render custom React components too. For example, we can now refer to the whole shopping list by writing `<ShoppingList />`. Each React component is encapsulated and can operate independently; this allows you to build complex UIs from simple components.

Inspecting the Starter Code

open `src/index.js` in your project folder

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
class Square extends React.Component {
  render() {
    return (
      <button className="square">
        {/* TODO */}
      </button>
    );
  }
}

class Board extends React.Component {
  renderSquare(i) {
    return <Square />;
  }

  render() {
    const status = 'Next player: X';

    return (
      <div>
        <div className="status">{status}</div>
        <div className="board-row">
          {this.renderSquare(0)}
          {this.renderSquare(1)}
          {this.renderSquare(2)}
        </div>
        <div className="board-row">
          {this.renderSquare(3)}
          {this.renderSquare(4)}
          {this.renderSquare(5)}
        </div>
        <div className="board-row">
          {this.renderSquare(6)}
          {this.renderSquare(7)}
          {this.renderSquare(8)}
        </div>
      </div>
    );
  }
}
```

```

}

class Game extends React.Component {
  render() {
    return (
      <div className="game">
        <div className="game-board">
          <Board />
        </div>
        <div className="game-info">
          <div>{/* status */}</div>
          <ol>{/* TODO */}</ol>
        </div>
      </div>
    );
  }
}

// =====

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

```

This Starter Code is the base of what we're building. We've provided the CSS styling so that you only need to focus on learning React and programming the tic-tac-toe game.

By inspecting the code, you'll notice that we have three React components:

- Square
- Board
- Game

The Square component renders a single `<button>` and the Board renders 9 squares. The Game component renders a board with placeholder values which we'll modify later. There are currently no interactive components.

Passing Data Through Props

To get our feet wet, let's try passing some data from our Board component to our Square component.

- In Board's `renderSquare` method, change the code to pass a prop called `value` to the Square
- Change Square's `render` method to show that value by replacing `{/* TODO */}` with `{this.props.value}`

```

class Square extends React.Component {
  render() {
    return (
      <button className="square">
        {this.props.value}
      </button>
    );
  }
}

```

```

}

class Board extends React.Component {
  rendersquare(i) {
    return <Square value={i} />;
  }

  render() {
    const status = 'Next player: x';

    return (
      <div>
        <div className="status">{status}</div>
        <div className="board-row">
          {this.renderSquare(0)}
          {this.renderSquare(1)}
          {this.renderSquare(2)}
        </div>
        <div className="board-row">
          {this.renderSquare(3)}
          {this.renderSquare(4)}
          {this.renderSquare(5)}
        </div>
        <div className="board-row">
          {this.renderSquare(6)}
          {this.renderSquare(7)}
          {this.renderSquare(8)}
        </div>
      </div>
    );
  }
}

class Game extends React.Component {
  render() {
    return (
      <div className="game">
        <div className="game-board">
          <Board />
        </div>
        <div className="game-info">
          <div>{/* status */}</div>
          <ol>{/* TODO */}</ol>
        </div>
      </div>
    );
  }
}

// =====

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

```

Before:

Next player: X

After: You should see a number in each square in the rendered output.

Next player: X

0	1	2
3	4	5
6	7	8

Making an Interactive Component

Let's fill the Square component with an "X" when we click it. First, change the button tag that is returned from the Square component's `render()` function to this:

```
class Square extends React.Component {
  render() {
    return (
      <button className="square" onClick={function() { alert('click'); }}>
        {this.props.value}
      </button>
    );
  }
}
```

Note

To save typing and avoid the [confusing behavior of `this`](#), we will use the [arrow function syntax](#) for event handlers here and further below:

```
class Square extends React.Component {
  render() {
    return (
      <button className="square" onClick={() => alert('click')}>
        {this.props.value}
      </button>
    );
  }
}
```

Notice how with `onClick={() => alert('click')}`, we're passing *a function* as the `onClick` prop. React will only call this function after a click. Forgetting `() =>` and writing `onClick={alert('click')}` is a common mistake, and would fire the alert every time the component re-renders.

As a next step, we want the Square component to “remember” that it got clicked, and fill it with an “X” mark. To “remember” things, components use **state**.

React components can have state by setting `this.state` in their constructors. `this.state` should be considered as private to a React component that it's defined in. Let's store the current value of the Square in `this.state`, and change it when the Square is clicked.

First, we'll add a constructor to the class to initialize the state:

```
class Square extends React.Component {
  ///////////////////////////////////////////////////Change Below
  constructor(props) {
    super(props);
    this.state = {
      value: null,
    };
  }
  ///////////////////////////////////////////////////Change Above
  render() {
    return (
      <button className="square" onClick={() => alert('click')}>
        {this.props.value}
      </button>
    );
  }
}
```

Now we'll change the Square's `render` method to display the current state's value when clicked:

- Replace `this.props.value` with `this.state.value` inside the `<button>` tag.
- Replace the `onClick={...}` event handler with `onClick={() => this.setState({value: 'X'})}`.
- Put the `className` and `onClick` props on separate lines for better readability.

After these changes, the `<button>` tag that is returned by the Square's `render` method looks like this:

```

class Square extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      value: null,
    };
  }

  render() {
    return (
      <button
        className="square"
        onClick={() => this.setState({value: 'X'})}
      >
        {this.state.value}
      </button>
    );
  }
}

```

Note

In [JavaScript classes](#), you need to always call `super` when defining the constructor of a subclass. All React component classes that have a `constructor` should start with a `super(props)` call.

Now we'll change the Square's `render` method to display the current state's value when clicked:

- Replace `this.props.value` with `this.state.value` inside the `<button>` tag.
- Replace the `onClick={...}` event handler with `onClick={() => this.setState({value: 'X'})}`.
- Put the `className` and `onClick` props on separate lines for better readability.

After these changes, the `<button>` tag that is returned by the Square's `render` method looks like this:

```

class Square extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      value: null,
    };
  }

  render() {
    return (
      <button
        className="square"
        onClick={() => this.setState({value: 'X'})}
      >
        {this.state.value}
      </button>
    );
  }
}

```

By calling `this.setState` from an `onClick` handler in the Square's `render` method, we tell React to re-render that Square whenever its `<button>` is clicked. After the update, the Square's `this.state.value` will be `'X'`, so we'll see the `x` on the game board. If you click on any Square, an `x` should show up.

When you call `setState` in a component, React automatically updates the child components inside of it too.

Developer Tools

The React Devtools extension for [Chrome](#) and [Firefox](#) lets you inspect a React component tree with your browser's developer tools.

The React DevTools let you check the props and the state of your React components.

After installing React DevTools, you can right-click on any element on the page, click "Inspect" to open the developer tools, and the React tabs ("🔗 Components" and "🔗 Profiler") will appear as the last tabs to the right. Use "🔗 Components" to inspect the component tree.

```
▼ <Game>
  ▼ <div className="game">
    ▼ <div className="game-board">
      ▼ <Board>
        ▼ <div>
          <div className="status">Next player: X</div>
          ▼ <div className="board-row">
            ▶ <Square>...</Square>
            ▶ <Square>...</Square>
            ▶ <Square>...</Square>
          </div>
          ▼ <div className="board-row">
            ▶ <Square>...</Square>
            ▶ <Square>...</Square>
            ▶ <Square>...</Square>
          </div>
          ▼ <div className="board-row">
            ▶ <Square>...</Square>
            ▶ <Square>...</Square>
            ▶ <Square>...</Square>
          </div>
        </div>
      </Board>
    </div>
    ▼ <div className="game-info">
      <div/>
      <ol/>
    </div>
  </div>
</Game>
```

Completing the Game

We now have the basic building blocks for our tic-tac-toe game. To have a complete game, we now need to alternate placing "X"s and "O"s on the board, and we need a way to determine a winner.

Lifting State Up

Currently, each Square component maintains the game's state. To check for a winner, we'll maintain the value of each of the 9 squares in one location.

We may think that Board should just ask each Square for the Square's state. Although this approach is possible in React, we discourage it because the code becomes difficult to understand, susceptible to bugs, and hard to refactor. Instead, the best approach is to store the game's state in the parent Board component instead of in each Square. The Board component can tell each Square what to display by passing a prop, [just like we did when we passed a number to each Square](#).

To collect data from multiple children, or to have two child components communicate with each other, you need to declare the shared state in their parent component instead. The parent component can pass the state back down to the children by using props; this keeps the child components in sync with each other and with the parent component.

Lifting state into a parent component is common when React components are refactored — let's take this opportunity to try it out.

Add a constructor to the Board and set the Board's initial state to contain an array of 9 nulls corresponding to the 9 squares:

添加高亮部分。

```
class Board extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      squares: Array(9).fill(null),
    };
  }

  renderSquare(i) {
    return <Square value={i} />;
  }
}
```

When we fill the board in later, the `this.state.squares` array will look something like this:

```
[
  'o', null, 'x',
  'x', 'x', 'o',
  'o', null, null,
]
```

The Board's `renderSquare` method currently looks like this:

```
renderSquare(i) {
  return <Square value={i} />;
}
```

In the beginning, we [passed the `value` prop down](#) from the Board to show numbers from 0 to 8 in every Square. In a different previous step, we replaced the numbers with an "X" mark [determined by Square's own state](#). This is why Square currently ignores the `value` prop passed to it by the Board.

We will now use the prop passing mechanism again. We will modify the Board to instruct each individual Square about its current value (`'x'`, `'o'`, or `null`). We have already defined the `squares` array in the Board's constructor, and we will modify the Board's `renderSquare` method to read from it:

```
renderSquare(i) {
  return <Square value={this.state.squares[i]} />;
}
```

```
class Square extends React.Component {
```

```

// TODO: remove the constructor
constructor(props) {
  super(props);
  this.state = {
    value: null,
  };
}

render() {
  // TODO: use onClick={this.props.onClick}
  // TODO: replace this.state.value with this.props.value
  return (
    <button className="square" onClick={() => this.setState({value: 'X'})}>
      {this.state.value}
    </button>
  );
}
}

class Board extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      squares: Array(9).fill(null),
    };
  }

  rendersquare(i) {
    return <Square value={this.state.squares[i]} />;
  }

  render() {
    const status = 'Next player: X';

    return (
      <div>
        <div className="status">{status}</div>
        <div className="board-row">
          {this.renderSquare(0)}{this.renderSquare(1)}{this.renderSquare(2)}
        </div>
        <div className="board-row">
          {this.renderSquare(3)}{this.renderSquare(4)}{this.renderSquare(5)}
        </div>
        <div className="board-row">
          {this.renderSquare(6)}{this.renderSquare(7)}{this.renderSquare(8)}
        </div>
      </div>
    );
  }
}

class Game extends React.Component {
  render() {
    return (
      <div className="game">
        <div className="game-board">
          <Board />
        </div>
      </div>
    );
  }
}

```

```

        <div className="game-info">
          <div>{/* status */}</div>
          <ol>{/* TODO */}</ol>
        </div>
      </div>
    );
  }
}

// =====

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

```

效果：空白的九个格子，点击格子变叉