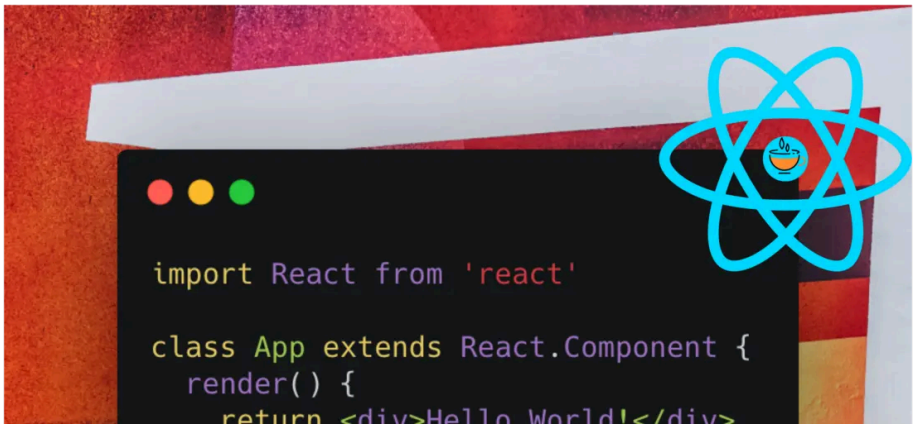
一杯茶的时间，上手 React 框架开发

一只图雀 2020-03-10 阅读 21 分钟



React（也被称为 React.js 或者 ReactJS）是一个用于构建用户界面的 JavaScript 库。起源于 Facebook 内部项目，最初用来架设 Instagram 的网站，并于 2013 年 5 月开源。React 性能较高，并且它的声明式、组件化特性让编写代码变得简单，随着 React 社区的发展，越来越多的人投入 React 的学习和开发，使得 React 不仅可以用来开发 Web 应用，还能开发桌面端应用，TV应用，VR应用，IoT应用等，因此 React 还具有一次学习，随处编写的特性。本教程将带你快速入门 React 开发，通过 20-30 分钟的学习，你不仅可以了解 React 的基础概念，而且能开发出一个待办事项小应用，还在想什么了？马上学起来吧！本文所有代码已放在 GitHub 仓库中。

> 此教程属于 React 前端工程师学习路线的一部分，欢迎来 Star 一波，鼓励我们继续创作出更好的教程，持续更新中~

Hello, World

我们将构建什么?

在这篇教程中，我们将展示给你如何使用 React 构建一个待办事项应用，下面最终项目的展示成果：

你也可以在这里看到我们最后构建的结果：最终结果。如果你现在对代码还不是很理解，或者你还不熟悉代码语法，别担心！这篇教程的目标就是帮助你理解 React 和它的语法。

我们推荐你在继续阅读这篇教程之前先熟悉一下这个待办事项，你甚至可以尝试添加几个待办事项！你可能注意到当你添加了2个待办事项之后，会出现不同的颜色；这就是 React 中条件渲染的魅力。

当你熟悉了这个待办事项之后你就可以关闭它了。在这篇教程的学习中，我们将从一个 Hello World 代码模板开始，然后带领你初始化开发环境，这样你就可以开始构建这个待办事项了。

你将学到什么?

你将学习所有 React 的基础概念，其中又分为三个部分：

- 编写组件相关：包括 JSX 语法、Component、Props
- 组件的交互：包括 State 和生命周期
- 组件的渲染：包括列表和 Key、条件渲染
- 和 DOM & HTML 相关：包括事件处理、表单。

前提条件

我们假设你熟系 HTML 和 JavaScript，但即使你是从其他编程语言转过来的，你也能看懂这篇教程。我们还假设你对一些编程语言的概念比较熟悉，比如函数、对象、数组，如果对类了解就更好了。

如果你需要复习 JavaScript，我们推荐你阅读这篇指南。你可能注意到了我们使用了一些 ES6 的特性 -- 一个最近的 JavaScript 版本。在这篇教程，我们会使用 arrow functions，classes，和 const。你可以使用 Babel REPL 来检查 ES6 代码编译之后的结果。

环境准备

首先准备 Node 开发环境，访问 Node 官方网站下载并安装。打开终端输入如下命令检测 Node 是否安装成功：

```
node -v # v10.16.0
npm -v # 6.9.0
```

> **注意**
> Windows 用户需要打开 cmd 工具，Mac 和 Linux 是终端。

如果上面的命令有输出且无报错，那么代表 Node 环境安装成功。接下来我们将使用 React 脚手架 -- Create React App（简称 CRA）来初始化项目，同时这也是官方推荐初始化 React 项目的最佳方式。

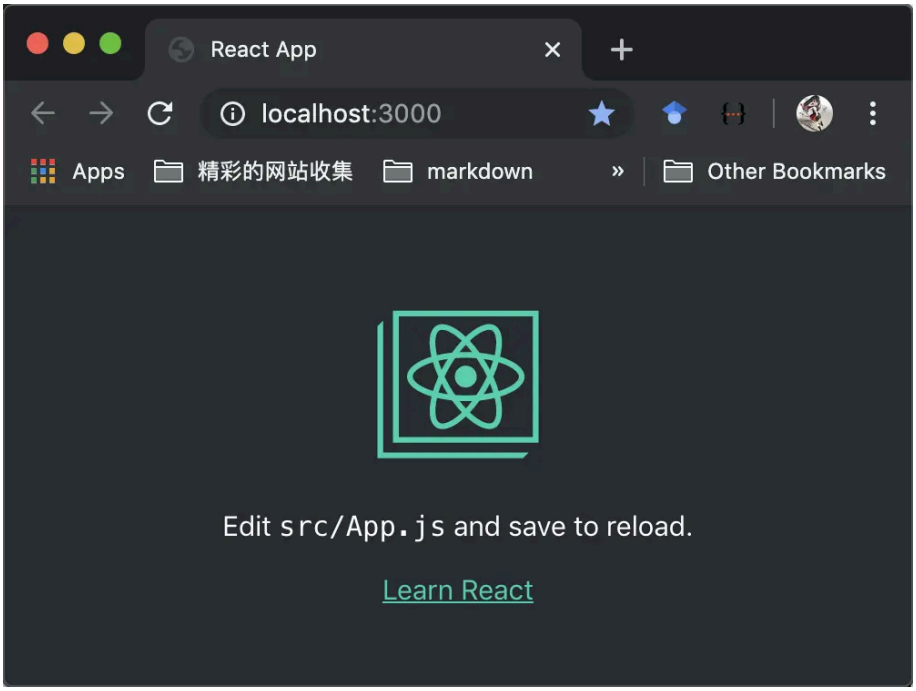
在终端中输入如下命令：

```
npx create-react-app my-todolist
```

等待命令运行完成，接着输入如下命令开启项目：

```
cd my-todolist && npm start
```

CRA 会自动开启项目并打开浏览器，你应该可以看到下面的结果：



??? 恭喜你！成功创建了第一个 React 应用！

现在 CRA 初始化的项目里有很多无关的内容，为了开始接下来的学习，我们还需要做一点清理工作。首先在终端中按 `ctrl + c` 关闭刚刚运行的开发环境，然后在终端中依次输入如下的命令：

```
# 进入 src 目录
cd src

# 如果你在使用 Mac 或者 Linux:
rm -f *

# 或者，你在使用 Windows:
del *

# 然后，创建我们将学习用的 JS 文件
# 如果你在使用 Mac 或者 Linux:
touch index.js

# 或者，你在使用 Windows
type nul > index.js

# 最后，切回到项目目录文件夹下
cd ..
```

此时如果在终端项目目录下运行 `npm start` 会报错，因为我们的 `index.js` 还没有内容，我们在终端中使用 `ctrl +c` 关闭开发服务器，然后使用编辑器打开项目，在刚刚创建的 index.js 文件中加入如下代码：

```
import React from "react";
import ReactDOM from "react-dom";

class App extends React.Component {
  render() {
    return <div>Hello, World</div>;
  }
}

ReactDOM.render(<App />, document.getElementById("root"));
```

我们看到 `index.js` 里面的代码分为三个部分。

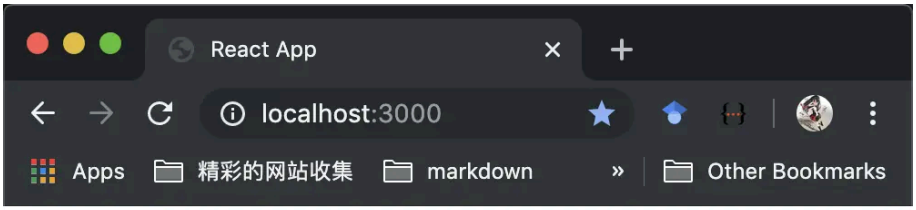
首先是一系列导包，我们导入了 `react` 包，并命名为 React，导入了 `react-dom` 包并命名为 ReactDOM。对于包含 React 组件（我们将在之后讲解）的文件都必须在文件开头导入 React。

然后我们定义了一个 React 组件，命名为 App，继承自 React.Component，组件的内容我们将会在后面进行讲解。

接着我们使用 ReactDOM 的 render 方法来渲染刚刚定义的 App 组件，`render` 方法接收两个参数，第一个参数为我们的 React 根级组件，第二个参数接收一个 DOM 节点，代表我们将把和 React 应用挂载到这个 DOM 节点下，进而渲染到浏览器中。

> **注意**
> 上面代码的三个部分中，**第一部分和第三部分在整篇教程中是不会修改的**，同时在编写任意 React 应用，这两个部分都是必须的。后面所有涉及到的代码修改都是关于第二部分代码的修改，或者是在第一部分到第三部分之间插入或删除代码。

保存代码，在终端中使用 `npm start` 命令开启开发服务器，现在浏览器应该会显示如下内容：



Hello, World

准备工作已经就绪！

你可能对上面的代码细节还不是很清楚，别担心，我们将马上带你领略 React 的神奇世界！

JSX 语法

首先我们来看一下 React 引以为傲的特性之一 -- JSX。它允许我们在 JS 代码中使用 XML 语法来编写用户界面，使得我们可以充分的利用 JS 的强大特性来操作用户界面。

一个 React 组件的 render 方法中 `return` 的内容就为这个组件所将渲染的内容。比如我们现在的代码：

```
render() {
    return <div>Hello, World</div>;
}
```

这里的 `<div>Hello, World</div>` 是一段 JSX 代码，它最终会被 Babel 转译成下面这段 JS 代码:

```
React.createElement(
  'div',
  null,
  'Hello, World'
)
```

`React.createElement()` 接收三个参数：

- 第一个参数代表 JSX 元素标签。
- 第二个参数代表这个 JSX 元素接收的属性，它是一个对象，这里因为我们的 `div` 没有接收任何属性，所以它是 `null`。
- 第三个参数代表 JSX 元素包裹的内容。

`React.createElement()` 会对参数做一些检查确保你写的代码不会产生 BUG，它最终会创建一个类似下面的对象：

```
{
  type: 'div',
  props: {
    children: 'Hello, World'
  }
};
```

这些对象被称之为 “React Element”。你可以认为它们描述了你想要在屏幕上看到的内容。React 将会接收这些对象，使用它们来构建 DOM，并且对它们进行更新。

> **注意**
> 我们推荐你使用 “Babel” 查看 JSX 的编译结果。

App 组件最终返回这段 JSX 代码，所以我们使用 ReactDOM 的 `render` 方法渲染 App 组件，最终显示在屏幕上的就是 `Hello, World"` 内容。

JSX 作为变量使用

因为 JSX 最终会被编译成一个 JS 对象，所以我们可以把它当做一个 JS 对象使用，它享有和一个 JS 对象同等的地位，比如可以将其赋值给一个变量，我们修改上面代码中的 render 方法如下：

```
render() {
  const element = <div>Hello, World</div>;
  return element;
}
```

保存代码，我们发现浏览器中渲染的内容和我们之前类似。

在 JSX 中使用变量

我们可以使用大括号 `{}` 在 JSX 中动态的插入变量值，比如我们修改 render 方法如下：

```
render() {
  const content = "World";
  const element = <div>Hello, {content}</div>;
  return element;
}
```

保存代码，发现浏览器中效果依然不变。

JSX 中使用 JSX

我们可以在 JSX 中再包含 JSX，这样我们编写任意层次的 HTML 结构：

```
render() {
    const element = <li>Hello, World</li>
    return (
      <div>
        <ul>
          {element}
        </ul>
      </div>
    )
  }
```

JSX 中添加节点属性

我们可以像在 HTML 中一样，给元素标签加上属性，只不过我们需要遵守驼峰式命名法则，比如在 HTML 上的属性 `data-index` 在 JSX 节点上要写成 `dataIndex`。

```
const element = <div dataIndex="0">Hello, World</div>;
```

> **注意**
> 在 JSX 中所有的属性都要更换成驼峰式命名，比如 `onclick` 要改成 `onClick`，唯一比较特殊的就是 `class`，因为在 JS 中 `class` 是保留字，我们要把 `class` 改成 `className` 。

```
const element = <div className="app">Hello, World</div>;
```

实战

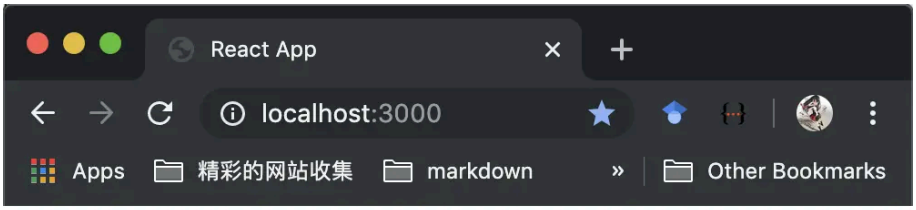
我们使用这一节讲解的 JSX 知识，来继续完成我们的待办事项应用。

在编辑器中打开 `src/index.js` ，对 App 组件做如下改变：

```
class App extends React.Component {
  render() {
    const todoList = ["图雀", "图雀写作工具", "图雀社区", "图雀文档"
    return (
      <ul>
        <li>Hello, {todoList[0]}</li>
        <li>Hello, {todoList[1]}</li>
        <li>Hello, {todoList[2]}</li>
        <li>Hello, {todoList[3]}</li>
      </ul>
    );
  }
}
```

可以看到，我们使用 `const` 定义了一个 `todoList` 数组常量，并且在 JSX 中使用 `{}` 进行动态插值，插入了数组的四个元素。

最后保存代码，浏览器中的效果应该是这样的：



- Hello, 图雀
- Hello, 图雀写作工具
- Hello, 图雀社区
- Hello, 图雀文档

> **提示**
> 无需关闭刚才使用 `npm start` 开启的开发服务器，修改代码后，浏览器中的内容将会自动刷新！

你可能注意到了我们手动获取了数组的四个值，然后逐一的用 `{}` 语法插入到 JSX 中并最终渲染，这样做还比较原始，我们将在后面列表和 Key 小节中简化这种写法。

在这一小节中，我们了解了 JSX 的概念，并且实践了相关的知识。我们还提出了组件的概念，但是并没有深入讲解它，在下一小节中我们将详细地讲解组件的知识。

Component

React 的核心特点之一就是组件化，即我们将巨大的业务逻辑拆分成一个个逻辑清晰的小组件，然后通过组合这些组件来完成业务功能。

React 提供两种组件写法：1）函数式组件 2）类组件。

函数式组件

在 React 中，函数式组件会默认接收一个 `props` 参数，然后返回一段 JSX：

```
function Todo(props) {
  return <li>Hello, 图雀</li>;
}
```

关于 `props` 我们将在下一节中讲解。

类组件

通过继承自 React.Component 的类来代表一个组件。

```
class Todo extends React.Component {
  render() {
    return <li>Hello, 图雀</li>;
  }
}
```

我们发现，在类组件中，我们需要在 `render` 方法里面返回需要渲染的 JSX。

组件组合

我们可以组合不同的组件来完成复杂的业务逻辑：

```
class App extends React.Component {
  render() {
    return (
```

```
      <ul>
        <Todo />
        <Todo />
      </ul>
    );
  }
}
```

在上面的代码中，我们在类组件 App 中使用了我们之前定义的 Todo 组件，我们看到，组件以 `<Component />` 的形式使用，比如 Todo 组件使用时为 `<Todo />`，我们在 Todo 组件没有子组件时使用这种写法；当 Todo 组件需要包含子组件时，我们需要写成下面的形式：

```
class App extends React.Component {
  render() {
    return (
      <ul>
        <Todo>Hello World</Todo>
        <Todo>Hello Tuture</Todo>>
      </ul>
    );
  }
}
```

组件渲染

我们在第一节中讲到，通过 `ReactDOM.render` 方法接收两个参数：1）根组件 2) 待挂载的 DOM 节点，可以将组件的内容渲染到 HTML 中。

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

实战

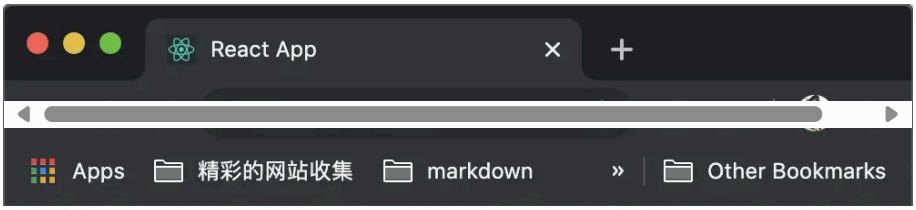
我们运用在这一节中学到的组件知识来继续完成我们的待办事项应用。我们编写一个 Todo 类组件，用于代表单个待办事项，然后在 App 类组件中使用 Todo 组件。

打开 `src/index.js` 文件，实现 Todo 组件，并调整 App 组件代码如下：

```
class Todo extends React.Component {
  render() {
    return <li>Hello, 图雀</li>;
  }
}
```

```
class App extends React.Component {
  render() {
    const todoList = ["图雀", "图雀写作工具", "图雀社区", "图雀文档"
    return (
      <ul>
        <Todo />
        <Todo />
        <Todo />
        <Todo />
      </ul>
    );
  }
}

ReactDOM.render(<App />, document.getElementById("root"));
```

保存代码，然后你应该可以在浏览器中看到如下结果：



- Hello, 图雀
- Hello, 图雀
- Hello, 图雀
- Hello, 图雀

你可能注意到我们暂时没有使用之前定义的 todoList 数组，而是使用了四个相同的 Todo 组件，我们使用继承自 `React.Component` 类的形式定义 Todo 组件，然后在组件的 `render` 中返回了 `<li>Hello, 图雀</li>`，所以最终浏览器中会渲染四个 `"Hello, 图雀"`。并且因为 Todo 组件不需要包含子组件，所以我们写成了 `<Todo />` 的形式。

现在4个待办事项都是一样的内容，这有点单调，你可能会想，如果可以像调用函数那样可以通过传参对组件进行个性化定制就好了，你的想法是对的！我们将在下一节中引出 props，它允许你给组件传递内容，从而进行个性化内容定制。

Props

React 为组件提供了 Props，使得在使用组件时，可以给组件传入属性进行个性化渲染。

函数式组件中使用 Props

函数式组件默认接收 `props` 参数，它是一个对象，用于保存父组件传递下来的内容：

```
function Todo(props) {
  return (
    <li>Hello, {props.content}</li>
  )
}

<Todo content="图雀" />
```

我们给 Todo 函数式组件传递了一个 `content` 属性， 它的值为 `"图雀"` ，所有传递的属性都会合并进 `props` 对象中，然后传递给 Todo 组件，这里 `props` 对象是这样的 `props = { content: "图雀" }` ，如果我们再传递一个属性：

```
<Todo content="图雀" from="图雀社区" />
```

最终 `props` 对象就会变成这样：`props={ content: "图雀", from: "图雀社区" }`。

> **注意**
> 如果给组件传递 `key` 属性是不会并入 props 对象中的，所以我们在子组件中也取不到 `key` 属性，我们将在列表和 Key 一节 中详细讲解。

类组件中使用 Props

类组件中基本和函数式组件中的 Props 保持一致，除了是通过 `this.props` 来获取父组件传递下来的属性内容：

```
class Todo extends React.Component {
  render() {
    return <li>Hello, {this.props.content}</li>;
  }
}
```

```
<Todo content="图雀" />
```

实战

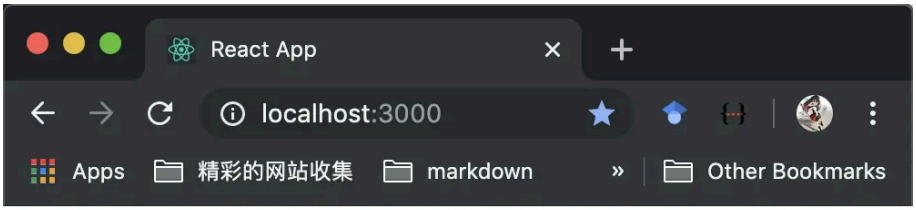
我们运用这一节中学到的 Props 知识来继续完成我们的待办事项应用。

打开 `src/index.js` 文件，分别调整 Todo 和 App 组件，修改后代码如下：

```
import React from "react";
import ReactDOM from "react-dom";

class Todo extends React.Component {
  render() {
    return <li>Hello, {this.props.content}</li>;
  }
}

class App extends React.Component {
  render() {
    const todoList = ["图雀", "图雀写作工具", "图雀社区", "图雀文档"
    return (
      <ul>
        {todoList.map((item) => (
          <Todo content={item} />
        ))}
      </ul>
    );
  }
}

ReactDOM.render(<App />, document.getElementById("root"));
```


注意到我们又重新开始使用之前定义的 todoList 数组，然后给每个 Todo 组件传递一个 `content` 属性，分别赋值数组的每一项，最后在 Todo 组件中使用我们传递下来的 `content` 属性。

保存修改的内容，你应该可以在浏览器中看到如下的内容：



- Hello, 图雀
- Hello, 图雀写作工具
- Hello, 图雀社区
- Hello, 图雀文档

可以看到，我们的内容又回来了，和我们之前在 JSX 一节 中看到的内容一样，但是这一次我们成功使用了组件来渲染接收到的 Props 内容。

State 和生命周期

React 通过给类组件提供 State 来创造交互式的内容 -- 即内容可以在渲染之后发生变化。

定义 State

通过在类组件中添加 `constructor` 方法，并在其中定义和初始化 State：

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

    this.state = {
      todoList: ["图雀", "图雀写作工具", "图雀社区", "图雀文档"]
    };
  }
```

这里 constructor 方法接收的 props 属性就是我们在上一节中讲到的那个 props；并且 React 约定每个继承自 React.Component 的组件在定义 constructor 方法时，要在方法内首行加入 `super(props)` 。

接着我们 `this.state` 来定义组件的 state，并使用 `{ todoList: ["图雀", "图雀写作工具", "图雀社区", "图雀文档"] }` 对象来初始化 state。

使用 State

我们可以在一个组件中的多处地方通过 `this.state` 的方式来使用 state，比如我们在这一节中将讲到的生命周期函数中，比如在 render 方法中：

```
class App extends React.Component {
  constructor(props) {
    super(props);

    this.state = {
      todoList: ["图雀", "图雀写作工具", "图雀社区", "图雀文档"]
    };
  }

  render() {
    return (
      <ul>
        <Todo content={this.state.todoList[0]} />
        <Todo content={this.state.todoList[1]} />
        <Todo content={this.state.todoList[2]} />
        <Todo content={this.state.todoList[3]} />
      </ul>
    );
  }
}
```

我们通过 `this.state.todoList` 可以获取我们在 `constructor` 方法中定义的 state，可以看到，我们使用 `this.state.todoList[0]` 的方式替代了之前的 `todoList[0]`。

更新 State

我们通过 `this.setState` 方法来更新 state，从而使得网页内容在渲染之后还能变化：

```
this.setState({ todoList: newTodoList });
```

> **注意**
>
> 关于 `this.setState` 我们需要注意以下几点：
>
> 1）这里不能够通过直接修改 `this.state` 的方式来更新 `state`:
>
> ```
> // 错误的
> this.state.todoList = newTodoList;
> ```

2）State 的更新是合并更新的：

比如原 `state` 是这样的：

```
constructor(props) {
  super(props);
  this.state = {
    todoList: [],
    nowTodo: '',
  };
}
```

然后你调用 `this.setState()` 方法来更新 `state`:

```
this.setState({ nowTodo: "Hello, 图雀" });
```

React 将会合并更新，即将 `nowTodo` 的新内容合并进原 `this.state`，当更新之后，我们的 `this.state` 将会是下面这样的：

```
this.state = { todoList: [], nowTodo: "Hello, 图雀" };
```

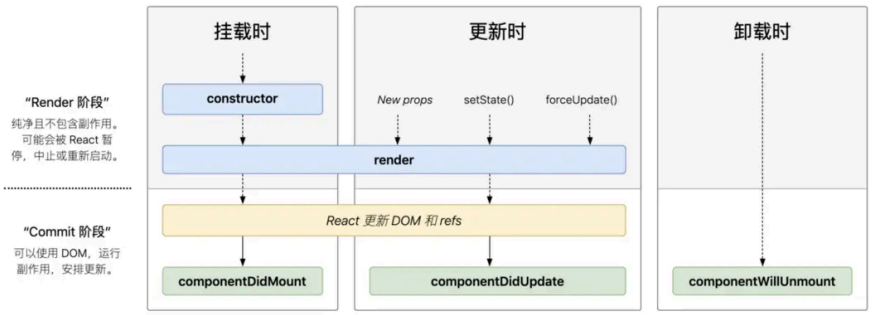
不会因为只单独设置了 `nowTodo` 的值，就将 `todoList` 给覆盖掉。

生命周期函数

React 提供生命周期函数来追踪一个组件从创建到销毁的全过程。主要包含三个方面：

- 挂载（Mounting）
- 更新（Updating）
- 卸载（Unmounting）

一个简化版的生命周期的图示是这样的：



注意

> React 生命周期相对而言比较复杂，我们这里不会详细讲解每个部分，上面的图示用户可以试着了解一下，对它有个大概的印象。
>
> 查看完整版的生命周期图示请参考这个链接：点击查看。

这里我们主要讲解挂载和卸载里面常用的生命周期函数。

挂载

其中挂载中主要常用的有三个方法：

- `constructor()`
- `render()`
- `componentDidMount()`

`constructor()` 在组件创建时调用，如果你不需要初始化 `State` ，即不需要 `this.state = { ... }` 这个过程，那么你不需要定义这个方法。

`render()` 方法是挂载时用来渲染内容的方法，每个类组件都需要一个 render 方法。

`componentDidMount()` 方法是当组件挂载到 DOM 节点中之后会调用的一个方法，我们通常在这里发起一些异步操作，用于获取服务器端的数据等。

卸载

卸载只有一个方法：

- `componentWillUnmount()`

componentWillUnmount 是当组件从 DOM 节点中卸载之前会调用的方法，我们一般在这里面销毁定时器等会导致内存泄露的内容。

实战

我们运用在这一节中学到的 State 和生命周期知识来继续完成我们的待办事项应用。

打开 `src/index.js`，修改代码如下：

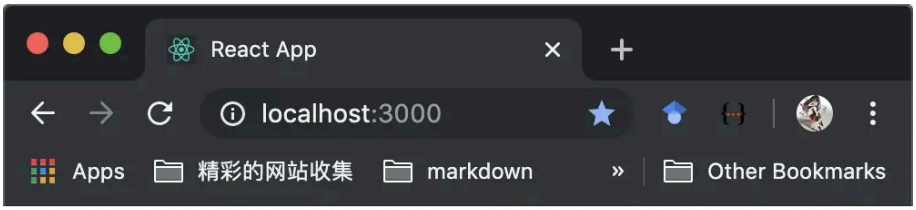
```
import React from "react";
import ReactDOM from "react-dom";

const todoList = ["图雀", "图雀写作工具", "图雀社区", "图雀文档"];
```

```
class Todo extends React.Component {
  render() {
    return <li>Hello, {this.props.content}</li>;
  }
}

class App extends React.Component {
  constructor(props) {
    super(props);

    this.state = {
      todoList: []
    };
  }

  componentDidMount() {
    this.timer = setTimeout(() => {
      this.setState({
        todoList: todoList
```

可以看到我们主要改动了五个部分：

- 将 todoList 移动到组件外面。
- 定义 constructor 方法，并且通过设置 `this.state = { todoList: [] }` 来初始化组件的 State，这里我们将 `todoList` 初始化为空数组。
- 添加 `componentDidMount` 生命周期方法，当组件挂载到 DOM 节点之后，设置一个时间为 2S 的定时器，并赋值给 `this.timer`，用于在组件卸载时销毁定时器。等到 2S 之后，使用 `this.setState({ todoList: todoList })` 来使用我们刚刚移动到组件外面的 todoList 来更新组件的 `this.state.todoList`。
- 添加 `componentWillUnMount` 生命周期方法，在组件卸载时，通过 `clearTimeout(this.timer)` 来清除我们之前设置的定时器，防止出现内存泄露。

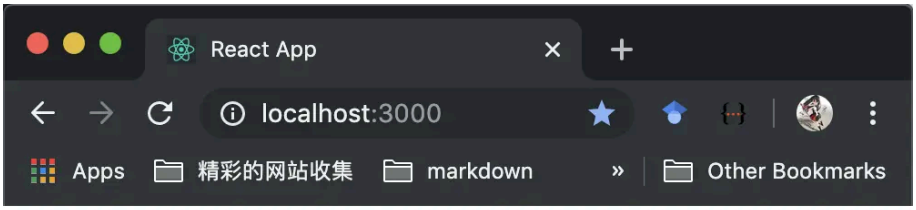
保存修改的代码，我们应该会看到浏览器中有一个内容更新的过程，在组件刚刚创建并挂载时，浏览器屏幕上应该是这样的：



- Hello,
- Hello,
- Hello,
- Hello,

因为我们在 `this.state` 初始化时，将 `todoList` 设置为了空数组，所以一开始 `"Hello"` 后面的 `this.props.content` 内容为空，我们就出现了四个 `"Hello, "`。

然后当过了 2S 之后，我们可以看到熟悉的内容出现了：



- Hello, 图雀
- Hello, 图雀写作工具
- Hello, 图雀社区
- Hello, 图雀文档

因为在过了 2S 之后，我们在定时器的回调函数中将 `todoList` 设置为了定义在组件外面的那个 `todoList` 数组，它有四个元素，所以显示在浏览器上面的内容又是我们之前的样子。

恭喜你！成功创建了自己第一个交互式的组件！

读到这里，你有可能有点累了，试着离开座椅，活动活动，喝杯咖啡，精彩稍后继续？

列表和 Key

目前我们有四个 Todo 组件，我们是一个一个取值然后渲染，这显得有点原始，并且不可扩展，因为当我们的 todoList 数组很大的时候（比如 100 个元素），一个一个获取就显得不切实际了，这个时候我们就需要循环介入了。

渲染组件列表

JSX 允许我们渲染一个列表：

```
render() {
    const todoList = ["图雀", "图雀写作工具", "图雀社区", "图雀文档"

    // 请注意：我们这里在 `map` 遍历时用了箭头函数简洁返回写法，直接用
    const renderTodoList = todoList.map((todo) => (
      <Todo content={todo} />
    ));
    return (
      <ul>
        {renderTodoList}
      </ul>
    );
  }
```

我们通过对 todoList 进行 `map` 遍历，返回了一个 Todo 列表，然后使用 `{}` 插值语法渲染这个列表。

当然我们可以在 JSX 中使用表达式，所以上面的代码可以写成这样：

```
render() {
    const todoList = ["图雀", "图雀写作工具", "图雀社区", "图雀文档"
    return (
      <ul>
        {todoList.map((todo) => (
          <Todo content={todo} />
        ))}
      </ul>
    );
  }
```

加上 Key

React 要求给列表中每个组件加上 `key` 属性，用于标志在列表中这个组件的身份，这样当列表内容进行了修改：增加或删除了元素时，React 可以根据 `key` 属性高效的对列表组件进行创建和销毁操作：

```
render() {
    const todoList = ["图雀", "图雀写作工具", "图雀社区", "图雀文档"
    return (
      <ul>
        {todoList.map((todo, index) => (
          <Todo content={todo} key={index} />
        ))}
      </ul>
    );
  }
```

这里我们使用了列表的 `index` 作为组件的 `key` 值，React 社区推荐的最佳实践方式是使用列表数据元素的唯一标识符作为 `key` 值，如果你的数据是来自数据库获取，那么列表元素数据的主键可以作为 `key`。

这里的 `key` 值不会作为 props 传递给子组件，React 会在编译组件时将 `key` 值从 props 中排除，即最终我们的第一个 Todo 组件的 `props` 如下：

```
props = { content: "图雀" }
```

而不是我们认为的：

```
props = { content: "图雀", key: 0 }
```

实战

我们运用这一节学到的列表和 Key 的知识来继续完成我们的待办事项应用。

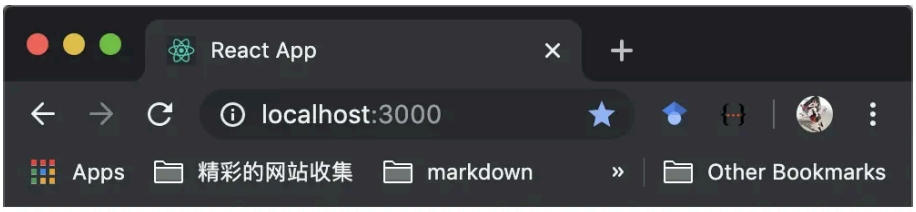
打开 `src/index.js`，代码修改如下：

```
import React from "react";
import ReactDOM from "react-dom";

const todoList = ["图雀", "图雀写作工具", "图雀社区", "图雀文档"];

class Todo extends React.Component {
```

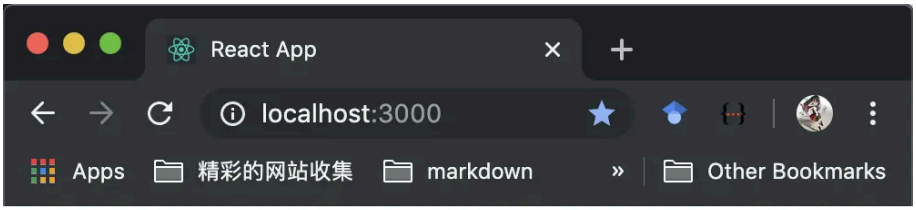
```
  render() {
    return <li>Hello, {this.props.content}</li>;
  }
}

class App extends React.Component {
  constructor(props) {
    super(props);

    this.state = {
      todoList: []
    };
  }

  componentDidMount() {
    this.timer = setTimeout(() => {
      this.setState({
        todoList: todoList
      });
```

可以看到，我们将之前的手动获取元素渲染改成了使用内嵌表达式的方式，通过对 `this.state.todoList` 列表进行 map 操作生成Todo 组件列表，然后使用列表的 `index` 作为组件的 `key` 值，最后渲染。

保存内容，查看浏览器里面的内容，我们可以看到内容会有一个变化的过程，一开始是这样的：



你会发现一片空白，然后过了 2S 变成了下面这样：



- Hello, 图雀
- Hello, 图雀写作工具
- Hello, 图雀社区
- Hello, 图雀文档

这是因为，一开始 `this.state.todoList` 在 constructor 中初始化时是空数组， `this.state.todoList` 进行 map 操作时返回空数组，所以我们的浏览器中没有内容，当组件挂载之后，等待 2S，我们更新 `this.state.todoList` 内容，就看到浏览器里面获得了更新。

条件渲染

在 React 中，我们可以根据不同的情况，渲染不同的内容，这也被成为条件渲染。

if-else 条件渲染

```
render() {
    if (this.props.content === "图雀") {
      return <li>你好, {this.props.content}</li>;
    } else {
      return <li>Hello, {this.props.content}</li>;
    }
  }
```

在上面的代码中，我们判断 `this.props.content` 的内容，当内容为 `"图雀"` 时，我们渲染 `"你好，图雀"`，对于其他内容我们依然渲染 `"Hello，图雀"`。

三元表达式条件渲染

我们还可以直接在 JSX 中使用三元表达式进行条件渲染：

```
  render() {
      return this.props.content === "图雀"? (
        <li>你好, {this.props.content}</li>
      ) : (
        <li>Hello, {this.props.content}</li>
      );
    }
```

当然三元表达式还可以用来条件渲染不同的 React 元素属性：

```
  render() {
      return (
        <li className={this.state.isClicked ? 'clicked' : 'notclick
      )
    }
```

上面我们判断组件的 `this.state.isClicked` 属性，如果 `this.state.isClicked` 属性为 `true`，那么我们最终渲染 `"clicked"` 类：

```
  render() {
      return (
        <li className="clicked"}>Hello, {this.props.content}</li>
      )
    }
```

如果 `this.state.isClicked` 为 `false`，那么最终渲染 `notclicked` 类：

```
  render() {
      return (
        <li className="notclicked">Hello, {this.props.content}</li>
      )
    }
```

实战

我们运用本节学到的知识继续完成我们的待办事项应用。

打开 `src/index.js` ，对 Todo 和 App 组件作出如下修改：

```
  class Todo extends React.Component {
    render() {
      if (this.props.index % 2 === 0) {
```

```
      return <li style={{ color: "red" }}>Hello, {this.props.co
    }

    return <li>Hello, {this.props.content}</li>;
  }
}

class App extends React.Component {
  constructor(props) {
    super(props);

    this.state = {
      todoList: []
    };
  }

  componentDidMount() {
    this.timer = setTimeout(() => {
      this.setState({
        todoList: todoList
      });
    }, 2000);
```

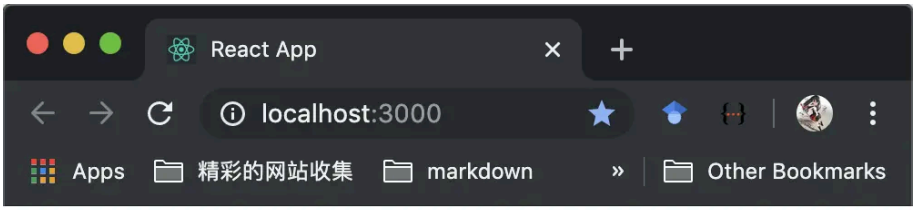
我们在首先在 App 组件中给 Todo 组件传入了一个 `index` 属性，然后在 Todo 组件的 render 方法中，对 `this.props.index` 进行判断，如果为偶数，那么渲染一个红色的文字，如果为奇数则保持不变。

这里我们通过给 `li` 元素的 `style` 属性赋值一个对象来实现在 JSX 中设置元素的 CSS 属性，我们可以通过同样的方式设置任何 CSS 属性：

```
// 黑底红字的 Hello, 图雀
<li style={{ color: "red", backgroundColor: "black"}}>Hello, 图雀
```

> **注意**
> 这里有一点需要做一下改变，就是像 `background-color` 这样的属性，要写成驼峰式 `backgroundColor`。对应的比如 `font-size`，也要写成 `fontSize`。

保存代码，你应该可以在浏览器中看到如下内容：



- Hello, 图雀
- Hello, 图雀写作工具
- Hello, 图雀社区
- Hello, 图雀文档

我们看到浏览器中的效果确实是在偶数项（数组中 0 和 2 项）变成了红色的字体，而（数组中 1 和 3 项）还是之前的黑色样式。

事件处理

在 React 元素中处理事件和在 HTML 中类似，就是写法有点不一样。

JSX 中的事件处理

这里的不一样主要包含以下两点：

- React 中的事件要使用驼峰式命名：`onClick`，而不是全小写：`onclick`。
- 在 JSX 中，你传递的是一个事件处理函数，而不是一个字符串。

在 HTML 中，我们处理事件是这样的：

```
<button onclick="handleClick()">点我</button>
```

在 React 中，我们需要写成下面这样：

```
function Button() {
  function handleClick() {
    console.log('按钮被点击了');
  }

  return (
```

```
    <button onClick={handleClick}>点我</button>
  )
}
```

注意到我们在上面定义了一个函数式组件，然后返回一个按钮，并在按钮上面定义了点击事件和对应的处理方法。

> **注意**
> 这里我们的的点击事件使用了驼峰式的 `onClick` 来命名，并且在 JSX 中传给事件的属性是一个函数：`handleClick`，而不是之前 HTML 中单纯的一个字符串：`"handleClick()"`。

合成事件

我们在以前编写 HTML 的事件处理时，特别是在处理表单时，常常需要禁用浏览器的默认属性。

> **提示**
> 比如一般表单提交时都会刷新浏览器，但是我们有时候希望提交表单之后不刷新浏览器，所以我们需要禁用浏览器的默认属性。

在 HTML 中我们禁用事件的默认属性是通过调用定义在事件上的 `preventDefault` 或者设置事件的 `cancelBubble`：

```
// 当点击某个链接之后，禁止打开页面
document.getElementById("myAnchor").addEventListener("click", fun
  event.preventDefault()
});
```

在 JSX 中，事件处理和这个类似：

```
function Link() {
  function handleClick(event)  {
    event.preventDefault();
    console.log('链接被点击了，但是它不会跳转页面，因为默认行为被禁用
  }

  return (
    <a onClick={handleClick} href="https://tuture.co">点我</a>
  )
}
```

实战

我们运用这一节中学到的知识来继续完成我们的待办事项应用。

我们之前的待办事项的 todoList 数组都是直接硬编码在代码里，不可以进行增删改，这相当死板，一个更真实的 todoList 应该要具备增加功能，这一功能实现需要两个步骤：

- 允许用户输入新的待办事项。
- 将这个输入的待办事项加入到现有的 todoList 列表里面。

在这一小节中，我们将来实现第一个步骤的内容。

打开 `src/index.js` ，对 App 组件内容作出如下修改：

```
class App extends React.Component {
  constructor(props) {
    super(props);

    this.state = {
      nowTodo: "",
      todoList: []
    };
  }

  componentDidMount() {
    this.timer = setTimeout(() => {
      this.setState({
        todoList: todoList
      });
    }, 2000);
  }

  componentWillUnmount() {
    clearTimeout(this.timer);
  }

  handleChange(e) {
    this.setState({
      nowTodo: e.target.value
```

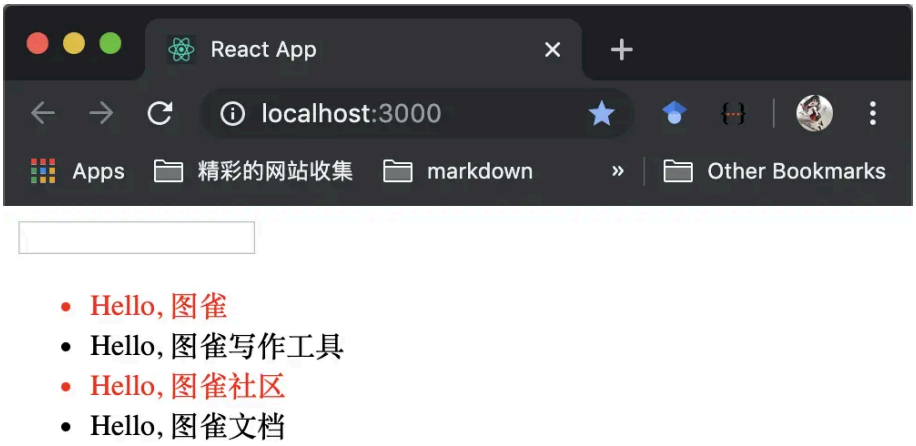
可以看到，我们新加入的代码主要有四个部分：

- 首先在 state 里面添加了一个新的属性 `nowTodo`，我们将用它来保存用户新输入的待办事项。
- 然后我们在 render 方法里面，在返回的 JSX 中使用 `div` 将内容包裹起来，接着加入了一个 `div`，在里面加入了一个 `input` 和 一个 `div`，input 用于处理用户的输入，我们在上面定义了 `onChange` 事件，事件

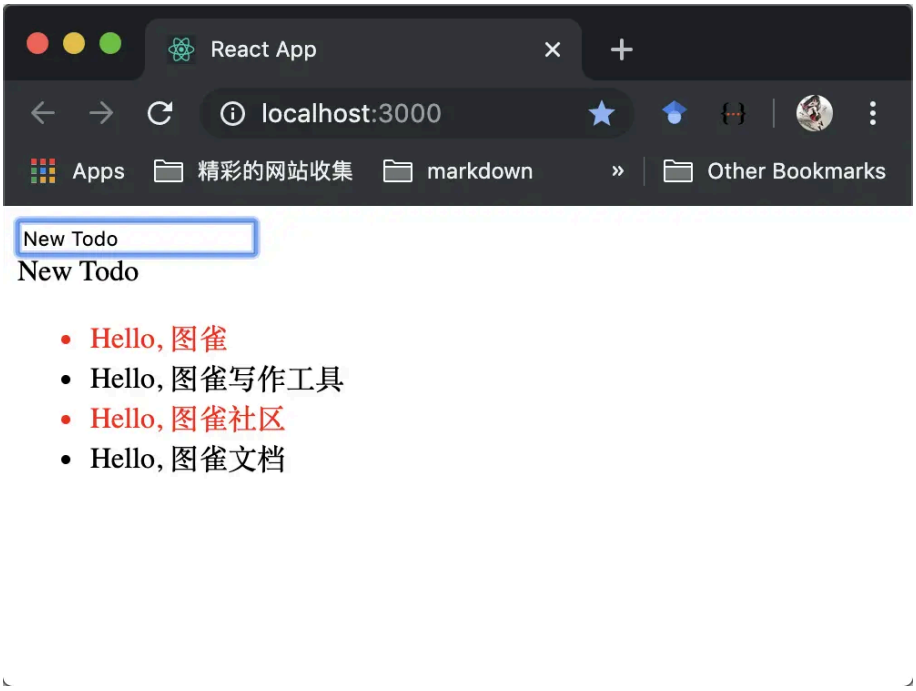
处理函数是一个箭头函数，它接收事件 `e`，然后用户输入时，会在函数里面调用 `this.handleChange` 方法，将事件 `e` 传给这个方法。

- 在 `handleChange` 里面通过 `this.setState` 使用 input 的值来更新 `nowTodo` 的内容。
- 最后在 div 里面使用 `{}` 插值语法展示 `nowTodo` 的内容。

保存代码，打开浏览器，你应该可以看到如下的内容：



当你尝试在输入框中键入内容时，输入框的下面应会显示相同的内容：



这是因为当我们在输入框里面输入内容时，我们使用了输入框的值更新 `this.state.nowTodo`，然后在输入框之下展示 `this.state.nowTodo` 的值。

表单

接下来我们来完成增加新的待办事项的功能的第二个步骤：允许用户将新输入的待办事项加入到 todoList 列表中。

打开 `src/index.js` 文件，对 App 组件做出如下修改：

```
class App extends React.Component {
  constructor(props) {
    super(props);

    this.state = {
      nowTodo: "",
      todoList: []
    };
  }

  componentDidMount() {
    this.timer = setTimeout(() => {
      this.setState({
        todoList: todoList
      });
    }, 2000);
  }

  componentWillUnmount() {
    clearTimeout(this.timer);
  }

  handleChange(e) {
    this.setState({
      nowTodo: e.target.value
```

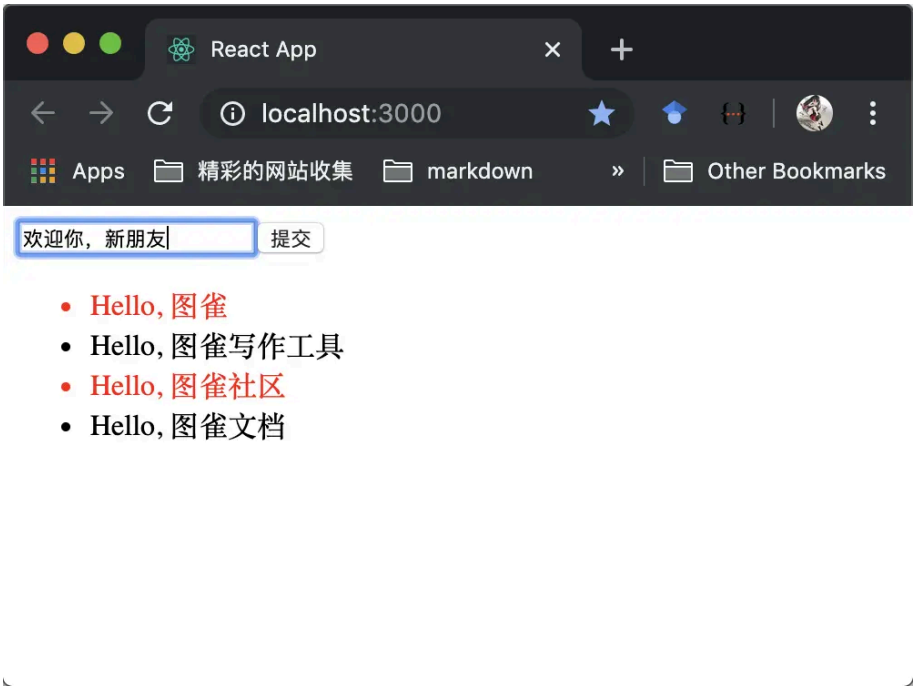
上面的变化主要有三个部分：

- 首先我们将 `render` 方法中返回的 JSX 的最外层的 `div` 替换成了 `form`，然后在上面定义了 `onSubmit` 提交事件，并且通过一个箭头函数接收事件 `e` 来进行事件处理，在 form 被提交时，在箭头函数里面会调用 `handleSubmit` 方法， 并将 e 传递给这个函数。
- 在 handleSubmit 方法里面，我们首先调用了 `e.preventDefault()` 方法，来禁止浏览器的默认事件处理，这样我们在提交表单之后，浏览器就不会刷新，之后是将现有的 `this.sate.todoList` 列表加上新输入的 `nowTodo`，最后是使用 `this.setState` 更新 `todoList` 和 `nowTodo`；这样我们就可以通过输入内容添加新的待办事项了。
- 接着我们将之前的展示 `this.state.nowTodo` 的 div 替换成 提交按钮 `button`，并将 button 的 `type` 设置为 `submit` 属性，表示在点击这个

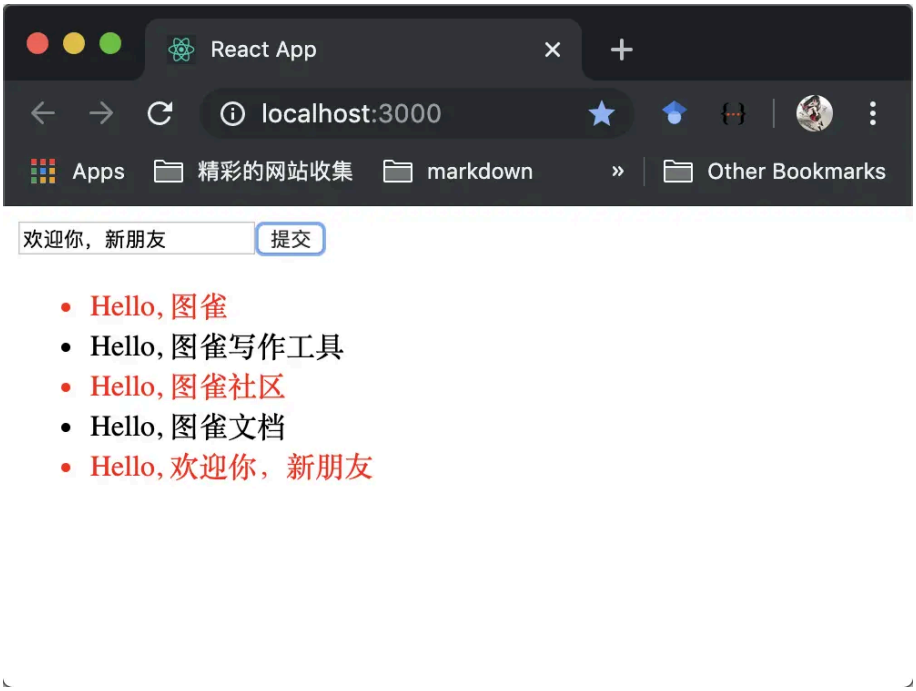
button 之后，会触发表单提交；将新输入的内容加入现有的待办事项中。

> **注意**
> 我们在 `handleSubmit` 方法里面使用 `this.setState` 更新状态时，将 `nowTodo` 设置为了空字符串，代表我们在加入新的待办事项之后，将清除现有输入的 nowTodo 待办事项内容。

保存代码，打开浏览器，在输入框里面输入点东西，你应该可以看到下面的内容：



当你点击提交按钮之后，新的待办事项会加入到现有的 todoList 列表中，你应该可以看到下面的内容：



恭喜你！你成功使用 React 完成了一个简单的待办事项应用，它可以完成如下的功能：

- 异步获取将要展示的待办事项：todoList
- 将待办事项展示出来
- 偶数项待办事项将会展示成红色
- 可以添加新的待办事项

做得好！我们希望你现在已经对 React 的运行机制有了一个比较完整的了解，也希望本篇教程能够为你踏入 React 开发世界提供一个好的开始！感谢你的阅读！

后记

受限于篇幅，我们的待办事项还不完整，如果你有额外的时间或者你想要练习你新学到的 React 知识，下面是一些使我们的待办事项变得完整的一些想法，我们将按实现难度给这些功能排序：

- 在添加新的待办事项之后，清空输入框里面的内容，方便下一次输入。这样涉及到 React 受控组件的知识。
- 允许对单个事项进行删除。这涉及到子组件修改父组件的状态知识。
- 允许用户对单个事项进行修改。
- 允许用户对待办事项进行搜索。

> 想要学习更多精彩的实战技术教程？来图雀社区逛逛吧。

前端 react.js

赞 7 收藏 6 分享

阅读 3.8k • 更新于 2020-03-12

一只图雀
863 声望 • 1.2k 粉丝
我们图雀社区是一个供大家分享用 Tuture 写作工具撰写教程的一个平台。在这里，读者们可以尽情享受高质量的实战教...
关注作者

下一篇 »
一杯茶的时间，上手 Node.js

引用和评论

推荐阅读

Taro 小程序开发大型实战（九）：使用 Authing 打造具有微信登录的企业级用户系统
一只图雀 • 赞 1 • 阅读 4.4k

深入理解React Diff算法
nero • 赞 36 • 阅读 15.8k • 评论 4

前端开发方法集
寒青 • 赞 23 • 阅读 2.6k

React中的高优先级任务插队机制
nero • 赞 32 • 阅读 15.1k • 评论 14

关于小程序如何做到强制更新
南玖 • 赞 13 • 阅读 828 • 评论 1

【你不知道的canvas】之更换绿屏视频背景
雾岛听风 • 赞 15 • 阅读 6.3k • 评论 8

【动画进阶】巧用 CSS/SVG 实现复杂线条光效动画
chokcoco • 赞 14 • 阅读 924

3 条评论

得票 | 最新

撰写评论 ...

提交评论

评论支持部分 Markdown 语法：`**粗体** _斜体_ [链接](http://example.com) `代码` - 列表 > 引用`。你还可以使用 @ 来通知其他用户。

powerformarc：SegmentFault 上看到的最好的 React 入门教程，没有之一?

回复 • 2020-03-11

一只图雀（作者）：@powerformarc 感谢您的认可，您的认可是对我们专注原创以及高质量教程的动力，敬请期待哈！！！

回复 • 2020-03-11

灿若星空丶：太棒了，没想到SegmentFault 还有这么好的文章，作者快更新啊

回复 • 2020-03-12