React

1. 回顾

- 函数、对象、数组，以及 class 的一些内容
- 箭头函数（arrow functions）、class、let 语句和 const 语句

2. 以史为镜，可以知得失

3. JavaScript Library

https://reactjs.org/

- Declarative
- Component-Based
- Learn Once, Write Anywhere

3.a Declarative 声明式

React makes it painless to create interactive UIs. Design simple views for each state in your application, and React will efficiently update and render just the right components when your data changes.

Declarative views make your code more predictable and easier to debug.

- painless
- design
- efficiently update and render when data change
- more predictable and easier to debug

Readable, Maintainable, Reusable

Imperative 命令式

现实世界中，我们大部分编码都是命令式的。

Vanilla JavaScript 是命令式的，我们命令 DOM 更新

```
document.querySelector('#resume-page').classList.add('page--active');
```

SQL 语句是声明式的，我们告诉数据库要什么，然后数据库就会给你对应的数据，而不是通过数据库的 API 去取。

```
SELECT * FROM Products WHERE name='Alipay'
```

一家命令式餐馆，从客人进入店开始

1. 老板告诉（命令）客人坐在 N 号桌。
2. 客人向老板点餐 「交互」。
3. 老板告诉（命令）后厨做几道什么样的菜。
4. 老板告诉（命令）厨师放什么原材料，多少调料，几成熟。
5. 老板上菜。

Readable, Maintainable, Reusable

一家声明式餐馆，从客人进入店开始

1. 当 Waiting Zone 有客人时，服务员A带客人入座。
2. 当客人入座服务员B管理区域时，服务员B点单。
3. 当客人点好单，服务员B讲单据送到后厨。
4. 当后厨看到新的单据时，根据单据做菜。
5. 当服务员C看到 Pick Up 区有菜品时，根据单据上菜。

Readable, Maintainable, Reusable

```
document.querySelector('#resumePage').classList.add('page--active');
document.querySelector('[href="resume"]').classList.add('navItem--
active');
```

```
<a
  href="home"
  class={`navItem ${activePage === 'homepage' && 'navItem---active'}`}
>
  Home
</a>
<a
  href="resume"
  class={`navItem ${activePage === 'resume' && 'navItem---active'}`}
>
  Resume
</a>
// ...

<div class={`page ${activePage === 'homepage' && 'page---active'}`}>
// ...
</div>
<div class={`page ${activePage === 'resumePage' && 'page---active'}`}>
// ...
</div>
```

3.b Component Based

网页程序在业务发展的过程中体积越来越庞大，其中堆叠了大量的业务逻辑代码，不同业务模块的代码相互调用，相互嵌套，代码之间的耦合性越来越高，调用逻辑会越来越混乱。当某个模块需要升级的时候，改动代码的时候往往会有牵一发而动全身的感觉。特别是多人合作的情况下，代码的维护会让人奔溃。

宜家家具

家具被划分成无数个小零件（Component），不同家具的小零件（Component）因为规格相同可以互相使用，安装方便。

乐高

相同规格的积木（Component）通过不同的组合方式组成不同的产品。

大型工程

京港澳跨海大桥。由大桥，引道组成。大桥由桥面，桥墩，....。整体桥面由多个小桥面组成。每个小桥面由...组成，每个...由...组成，这种每个小产品（Component）由分布在各地不同的工厂生产，最大化效率。

Readable, Maintainable, Reusable

```
<header class="nav">
  <div class="nav__left">
    <div class="logo">
      <span class="logo__highlight">Tifa</span>
      <span>Lockhart</span>
    </div>
  </div>
  <div class="nav__right">
    <div class="navbar">
      <a class="navbar__item" href="HOME">Home</a>
      <a class="navbar__item" href="RESUME">Resume</a>
      // ...
    </div>
  </div>
</header>
<div class={`page ${activePage === 'homepage' && 'page---active'}`}>
// ...
</div>
<div class={`page ${activePage === 'resumePage' && 'page---active'}`}>
// ...
</div>
```

```
<Nav>
<Homepage />
<ResumePage />
```

```
// Nav
<div class="nav">
  <div class="nav__left">
    <Logo>
  </div>
  <div class="nav__right">
```

```
    <Navbar>
  </div>
</div>
```

```
// NavBar
<div class="navbar">
  <NavBarItem href="HOME">Home</NavBarItem>
  <NavBarItem href="RESUME">Resume</NavBarItem>
  // ...
</div>
```

Component

- Single Responsibility
- 就近维护，面向功能（责任）

```
// NavBarItem
NavBarItem.css
NavBarItem.js
NavBarItem.html
```

3.c Learn Once, Write Anywhere

- Server Side Rendering (SSR)
- React Native (RN)

React —> 生成 UI --> React * —> 生成的UI挂载到 *

—javascript —> 操作DOM来更新
React DOM HTML Web
React Native, —> React —> 搭建在IOS或者Android上
Mobile
- Andorid java
-IOS object c/swift

-SSR Service Side Rendering

3.d create-react-app

npx

Node package manager
yarn or npm to manage the external dependency

4. React 哲学

Component! Component! Component!

就像写 OOP 一样，写 React 首先考虑的就是 Component。

Readable, Maintainable, Reusable

React 最棒的部分之一是引导我们思考如何构建一个应用。

https://zh-hans.reactjs.org/docs/thinking-in-react.html

- 第一步：将设计好的 UI 划分为组件层级
- 第二步：用 React 创建一个静态版本
- 第三步：确定 UI state 的最小（且完整）表示
- 第四步：确定 state 放置的位置
- 第五步：添加反向数据流

4.a 第一步: 将设计好的 UI 划分为组件层级

参考宜家和大型化工程，按功能划分。 参考乐高，按复用划分。

Tree View Structure

- App
  - Nav
    - Logo
    - NavBar
      - NavBarItem
  - Homepage | ResumePage ... (Page)
  - Footer

4.b 第二步: React 创建一个静态版本

4.b.1 JSX

Function Component 返回一个 React Component，这是一种对渲染内容的轻量级描述。大多数的 React 开发者使用了一种名为 “JSX” 的特殊语法。

JSX 可以让你更轻松地书写这些结构。语法 `<div />` 会被编译成 `React.createElement('div')`。

JSX 给我们更大的便利维护代码，因为 HTML in JS。但是需要注意，一些 HTML 的 keywords 在 JSX 是不一样的，不如 class。

```
// Logo.js
import React from 'react';

const Logo = () => (
  <div className="logo">
    <span className="logo__highlight">Tifa</span>
    <span>Lockhart</span>
  </div>
);

export default Logo;
```

```
return React.createElement('div', {className: 'logo'}, [
  React.createElement('span', /* ... h1 children ... */),
  React.createElement('span', /* ... ul children ... */),
]);
```

使用 JSX 组合和渲染自定义的 React 组件

```
// Nav.js
import Nav from 'react';
import Logo from './Logo';
```

```
const Nav = () => (
  <header className="nav">
    <div className="nav__left">
      <Logo />
    </div>
    <div className="nav__right">
      <NavBar />
    </div>
  </header>
);

export default Nav;
```

4.b.2 props

想象一下乐高，我们如何更高效的复用积木（Component）。答案是组件的多样性。

我们可以定义一个积木，但是一个是灰色的1-4，而另外一个是绿色2-2。

通过定义不同的组件属性来实现组件的多样性。

组件，从概念上类似于 JavaScript 函数。它接受任意的入参（即 “props”），并返回用于描述页面展示内容的 React 元素。

在 React 应用中，数据通过 props 的传递，从父组件流向子组件。

在 JSX 中可以可以使用大括号来加入JavaScript表达式。

- 遇到{}，JS 表达式。
- 遇到<>，HTML 表达式。

```
<a class="navbar__item" href="HOME">Home</a>
```

```
//NavBarItem
import React from 'react';

const NavBarItem = (props) => (
  <a className="navbar__item" href={props.href}>{props.children}</a>
);

export default NavBarItem;

// NavBar
<div className="navBar">
  <NavBarItem href="HOME">Home</NavBarItem>
  <NavBarItem href="Resume">Resume</NavBarItem>
</div>
```

4.b.3 TODO: css module

4.b.4 babel 编译

- JSX 不能直接执行
- 为什么要 `import React from 'react';`

Babel is a JavaScript compiler

Babel is a toolchain that is mainly used to convert ECMAScript 2015+ code into a backwards compatible version of JavaScript in current and older browsers or environments. Here are the main things Babel can do for you:

- Transform syntax
- Polyfill features that are missing in your target environment (through @babel/polyfill)
- Source code transformations (codemods)
- And more! (check out these videos for inspiration)

https://babeljs.io/repl

4.c 第三步: 确定 UI state 的最小（且完整）表示

4.c.1 Handling Event

React 元素的事件处理和 DOM 元素的很相似，但是有一点语法上的不同。

https://zh-hans.reactjs.org/docs/events.html#supported-events

```
const Item = ({
  href,
  children,
}) => {
  const handleItemClick = (event) => {
    event.preventDefault(); // 阻止浏览器默认行为

    console.log('The link was clicked.');
  };

  return (
    <a className="item" href={href} onClick={handleItemClick}>
      {children}
    </a>
  );
}
```

4.c.2 Interaction

我们需要一个地方保存页面当前的状态。

- 当 Item 被点击时，我们设置 active 值。

- 当 active 时，我们添加 active className。

```
import React from 'react';
import classNames from 'classnames'; // yarn add classnames
import styles from './Item.css';

const cx = classNames.bind(styles); //
https://www.npmjs.com/package/classnames#alternate-bind-version-for-css-
modules-

const Item = ({
  href,
  children,
}) => {
  let active = false;

  const handleItemClick = (event) => {
    event.preventDefault(); // 阻止浏览器默认行为

    console.log('The link was clicked.');
    active = true; // 当 `Item` 被点击时，我们设置 `active` 值。

    // active 已经被设置成功，但是页面并不能正常工作。
    console.log('DEBUG: handleItemClick -> active', active);
  };

  return (
    <a
      className={cx({
        item: true,
        active: active, // 当 `active` 时，我们添加 `active` className。
      })}
      href={href}
      onClick={handleItemClick}
    >
      {children}
    </a>
  );
}
```

为什么不能正常工作？因为我们需要告诉浏览器去用最新的值刷新页面 (命令式)。

React 提供了声明式的 API，但是所有 API 都是基于 Vanilla JS。React 需要知道什么时候去命令浏览器刷新 DOM。

4.c.3 Declarative Vanilla JS

4.c.4 state

React.Component

我们可以通过继承 React.Component 来使用一些 React 提供的 High-Level API。

```
import React from 'react';
import classNames from 'classnames'; // yarn add classnames
import styles from './Item.css';

const cx = classNames.bind(styles); //
https://www.npmjs.com/package/classnames#alternate-bind-version-for-css-
modules-

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

    this.state = {
      active: false,
    };
  }

  handleItemClick = (event) => {
    event.preventDefault();
    this.setState({
      active: true,
    });
  }

  render = () => {
    const { href, children } = this.props;
    const { active } = this.state;

    return (
      <a
        className={cx({
          item: true,
          active, // 当 `active` 时，我们添加 `active` className。
        })}
        href={href}
        onClick={this.handleItemClick}
      >
        {children}
      </a>
    );
  }
}
```

继承父类时调用父类构造函数

```
class Person {
  constructor(name) {
    this.name = name
```

```
  }
}

class SayHiPerson extends Person {
  constructor() {
    this.sayHi();
  }

  sayHi() {
    console.log(`Greeting ${this.name}`);
  }
}

const alice = new SayHiPerson('alice')
```

this! this! this!

bind() 方法创建一个新的函数，在 bind() 被调用时，这个新函数的 this 被指定为 bind() 的第一个参数，而其余参数将作为新函数的参数，供调用时使用。

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

    this.state = {
      active: false,
    };

    this.handleItemClick = this.handleItemClick.bind(this);
  }

  handleItemClick(event) {
    event.preventDefault();

    // this?
    this.setState({
      active: true,
    });
  }

  render() {
    const { href, children } = this.props;
    const { active } = this.state;

    return (
      <a
        className={cx({
          item: true,
          active: active,
        })}
        href={href}
```

```
          onClick={this.handleItemClick}
        >
          {children}
        </a>
      );
    }
  }
```

state

组件中的 state 包含了随时可能发生变化的数据。state 由用户自定义，它是一个普通 JavaScript 对象。 永远不要直接改变 this.state。仅允许在 contractor 中直接赋值初始化 this.state。

Immutable

Immutable 可以给你的应用带来性能提升，也可以带来更简单的编程和调试体验。这是因为，与那些在整个应用中可被随意篡改的数据相比，永远不变的数据更容易追踪，推导。

特别来说，在 Web 应用中对于 Immutable 的使用，可以让复杂的变化检测机制得以简单快速的实现。从而确保代价高昂的 DOM 更新过程只在真正需要的时候进行（这也是 React 性能方面优于其他类库的基石）。

setState

- setState() 会对一个组件的 state 对象安排一次更新。当 state 改变了，该组件就会重新渲染。而直接更改 state 并不会触发组件重新渲染。
- setState() 将会合并参数和当前 state。

调用 setState 其实是异步的 —— 不要指望在调用 setState 之后，this.state 会立即映射为新的值。

```
incrementCount() {
  // 注意：这样 *不会* 像预期的那样工作。
  this.setState({count: this.state.count + 1});
}

handleSomething() {
  // 假设 `this.state.count` 从 0 开始。
  this.incrementCount();
  this.incrementCount();
  this.incrementCount();
  // 当 React 重新渲染该组件时，`this.state.count` 会变为 1，而不是你期望的 3。

  // 这是因为上面的 `incrementCount()` 函数是从 `this.state.count` 中读取数据的，
  // 但是 React 不会更新 `this.state.count`，直到该组件被重新渲染。
  // 所以最终 `incrementCount()` 每次读取 `this.state.count` 的值都是 0，并将它
设为 1。
}
```

使用回调函数作为参数

```
incrementCount() {
  // this.setState({
  //   count: this.state.count + 1
  // });

  this.setState((state) => {
    // 重要：在更新的时候读取 `state`，而不是 `this.state`。
    return {count: state.count + 1}
  });
}

handleSomething() {
  // 假设 `this.state.count` 从 0 开始。
  this.incrementCount();
  this.incrementCount();
  this.incrementCount();

  console.log(this.state.count);

  // 如果你现在在这里读取 `this.state.count`，它还是会为 0。
  // 但是，当 React 重新渲染该组件时，它会变为 3。
}
```

使用第二个参数

```
// setState(obj) // -> 拿不到最新的值
// setState(callbackFunction) // 用最新的值调用 callback, cb state 最新的

render();
count = 0;

this.setState({count: this.state.count + 1});
incrementCount count (cb(newState), cb1);
newState = 1
this.state = 0;

incrementCount count (cb(newState));
newState = 2;
this.state = 0;
incrementCount count (cb(newState))
newState = 3;
this.state = 0;

render();
this.state = newState;
cb1();
console.log(this.state) -> 3;

count = 3;
```

```
  incrementCount() {
    this.setState((state) => {
      return {
        count: state.count + 1,
      };
    });
  }

  handleSomething() {
    // 假设 `this.state.count` 从 0 开始。
    this.incrementCount();
    this.incrementCount();
    this.incrementCount();

    // 如果你现在在这里读取 `this.state.count`，它还是会为 0。
    // 但是，当 React 重新渲染该组件时，它会变为 3。
  }
```

4.d 第四步: 确定 state 放置的位置

状态提升 State Lifting

通常，多个组件需要反映相同的变化数据，这时我们建议将共享状态提升到最近的共同父组件中去。

Source of truth

在 React 应用中，任何可变数据应当只有一个相对应的唯一“数据源”。

通常，state 都是首先添加到需要渲染数据的组件中去 (最小且完整状态)。然后，如果其他组件也需要这个 state，那么你可以将它提升至这些组件的最近共同父组件中。

如果某些数据可以由 props 或 state 推导得出，那么它就不应该存在于 state 中。

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

    this.state = {
      activePage: 'HOME',
    };

    this.handleItemClick = this.handleItemClick.bind(this);
  }

  handleItemClick(href) {
    this.setState({
      activePage: href,
    });
  }

  render() {
```

```
      const { activePage } = this.state;

      <div className={styles.navBar}>
        <Item href="HOME" onClick={handleItemClick} active={activePage ===
"HOME"}>Home</Item>
        <Item href="RESUME" onClick={handleItemClick} active={activePage ===
"RESUME"}>Resume</Item>
        ...
      </div>
    }
  }

  const Item = ({
    onClick,
    active,
    href,
    children,
  }) => (
    <a
      className={cx({
        item: true,
        active: active,
      })}
      href={href}
      onClick={(event) => {
        event.preventDefault();
        onClick(href);
      }}
    >
      {children}
    </a>
  )
```

state VS props VS class property

- props 是只读的，用于传递数据。props 可以是父组件的 state。
- state 是可以改变的动态数据，用于渲染页面。
- class property?

Snapshot

- 为什么父组件的 state 可以作为 props 传递给子组件?

渲染多个组件

我们使用 Javascript 中的 map() 方法来遍历数组，创建多个元素组件。

当你创建一个元素时，必须包括一个特殊的 key 属性。key 帮助 React 识别哪些元素改变了，比如被添加或删除。因此你应当给数组中的每一个元素赋予一个确定的标识。

- 唯一标识 (例如，database ID)

```
const items = [{
  text: 'Home',
  href: 'HOME',
}, {
  text: 'Resume',
  href: 'RESUME',
}, {
  text: 'Blog',
  href: 'BLOG',
}];

class NavBar extends React.Component {
  ...

  render() {
    const { active } = this.state;

    <div className={styles.navBar}>
      {items.map(({ text, href }) => (
        <Item href={href} onClick={this.handleItemClick} active=
{activePage === href} key={href}>{text}</Item>
      ))}
    </div>
  }
}
```

4.d.4 lifecycle

生命周期的三个状态

- Mounting：插入 DOM
- Updating：重新渲染（更新）DOM
- Unmounting：移出 DOM

React 为每个状态都提供了两种处理函数，`will` 函数在进入状态之前调用，`did` 函数在进入状态之后调用，三种状态共计五种处理函数。

- componentWillMount()
- componentDidMount()
- componentWillUpdate(object nextProps, object nextState)
- componentDidUpdate(object prevProps, object prevState)
- componentWillUnmount()

注意 React 不提供 componentDidUnmount 方法

此外，React 还提供两种特殊状态的处理函数。

- componentWillReceiveProps(object nextProps) 已加载组件收到新的参数时调用
- shouldComponentUpdate(object nextProps, object nextState) 组件判断是否重新渲染时调用

我们得到最重要的经验是，过时的组件生命周期往往会带来不安全的编码实践，具体函数如下：

- componentWillMount
- componentWillReceiveProps
- componentWillUpdate

```
class Parent extends React.Component {
  construct(props) {
    super(props);

    this.state = {
      value: '',
    };

    this.handleValueChange = this.handleValueChange.bind(this);
  }

  handleValueChange(newValue) {
    this.setState({
      value: newValue,
    });
  }

  render() {
    return (
      <div>
        <ChildA value={this.state.value} onChange={this.handleValueChange}
/>
        <ChildB value={this.state.value} onChange={this.handleValueChange}
/>
      </div>
    )
  }
}

<Parent>
  <ChildA value={this.state.value} onChange={this.handleValueChange} />
  <ChildB value={this.state.value} onChange={this.handleValueChange} />
</Parent>

class ChildB extends React {
  componentWillUpdate(nextProps, nextState) {
    if (nextProps.value === 'ABC') {
      this.props.onChange('');
    }
  }

  ...
}
```

ChildA: Parent.setState -> render() {能不能成功取消这次 render?} ChildB: Parent.setState -> componentWillUpdate -> render() {取消这次 render} -> Parent.setState -> render()

如果 ChildA 和 ChildB 是同步渲染的话，ChildA 有能力告诉 ChildB 取消这次渲染，直到下次渲染 但是，在一些情况中如果 ChildA 和 ChildB 是异步渲染的话，ChildA 没有能力（很难）与 ChildB 通讯， 那么 ChildA 和 ChildB 就会出现不一致的状态

ChildA: Parent.setState render() -> render() ChildB: Parent.setState render() -> ChildB.componentDidUpdate -> Parent.setState -> render()

4.d.5 CAP 定理

- 一致性（Consistency） （等同于所有节点访问同一份最新的数据副本）
- 可用性（Availability）（每次请求都能获取到非错的响应——但是不保证获取的数据为最新数据）
- 分区容错性（Partition tolerance） （以实际效果而言，分区相当于对通信的时限要求。系统如果不能在时限内达成数据一致性，就意味着发生了分区的情况，必须就当前操作在C和A之间做出选择。）

4.d.6 Fetch Data 异步渲染更新

4.e 第五步: 添加反向数据流

4.e.1 一等公民

Pure function is the one and only first-class citizen

4.e.2 TODO: context

5. Test

现在有许多种测试 React 组件的方法。大体上可以被分为两类：

1. 渲染组件树 在一个简化的测试环境中渲染组件树并对它们的输出做断言检查。
2. 运行完整应用 在一个真实的浏览器环境中运行整个应用（也被称为“端到端（end-to-end）”测试）

5.a 迭代速度 vs 真实环境

一些工具在做出改动和看到结果之间提供了非常快速的反馈循环，但没有精确的模拟浏览器的行为。 另一些工具，也许使用了真实的浏览器环境，但却降低了迭代速度，而且在持续集成服务器中不太可靠。

5.b Mock

对组件来说，“单元测试”和“集成测试”之间的差别可能会很模糊。

如果你在测试一个表单，用例是否应该也测试表单里的按钮呢？一个按钮组件又需不需要有他自己的测试套件？重构按钮组件是否应该影响表单的测试用例？

5.c 测试运行器

- 使用 `Jest` 作为测试运行器
- 渲染到 `jsdom`
- 使用辅助函数提供的能力通过一系列的浏览器事件来模拟用户交互行为

5.d React Testing Library

5.e E2E 测试

E2E（端对端）测试对于测试更长的工作流程非常有用，特别是当它们对于你的业务（例如付款或注册）特别重要时。对于这些测试，你可能会希望测试真实浏览器如何渲染整个应用、从真实的 API 端获取数据、使用 session 和 cookies 以及在不同的链接间导航等功能。你可能还希望不仅在 DOM 状态上进行断言，而同时也在后端数据上进行校验。

5.f Cypress

5.g Making your UI tests resilient to change

https://kentcdodds.com/blog/making-your-ui-tests-resilient-to-change/

7. Redux

React 只是 DOM 的一个抽象层，并不是 Web 应用的完整解决方案。

在大型项目中，React 中两个方面会让人意外的头疼。

- 代码结构
- 组件之间的通信

"只有遇到 React 实在解决不了的问题，你才需要 Redux 。"

by Redux 的创造者 Dan Abramov

7.a Redux 设计模式

7.a.1 state

```
{
  todos: [{
    text: 'Eat food',
    completed: true
  }, {
    text: 'Exercise',
    completed: false
  }],
}
```

state 就像 “Model”，区别是它并没有 setter（修改器方法）。因此其它的代码不能随意修改它，造成难以复现的 bug。

7.a.2 action

要想更新 state 中的数据，我们需要发起一个 action。Action 就是一个普通 JavaScript 对象用来描述发生了什么。

```
{ type: 'ADD_TODO', text: 'Go to swimming pool' }
{ type: 'TOGGLE_TODO', index: 1 }
```

强制使用 action 来描述所有变化带来的好处是可以清晰地知道应用中到底发生了什么。如果一些东西改变了，就可以知道为什么变。action 就像是描述发生了什么的指示器。

7.a.3 reducer

我们创建一个把 action 和 state 串起来的函数，这就是 reducer。reducer 只是一个接收 state 和 action，并返回新的 state 的函数。

```
// { type: 'ADD_TODO', text: 'Go to swimming pool' }
// { type: 'TOGGLE_TODO', index: 0 }
function todos(state = [], action) {
  switch (action.type) {
    case 'ADD_TODO':
      return [
        ...state,
        {
          text: action.text,
          completed: falsem
        }
      ]
    case 'TOGGLE_TODO':
      return state.map((todo, index) =>
        action.index === index
          ? { text: todo.text, completed: !todo.completed }
          : todo
      )
    default:
      return state
  }
}
```

Immutable

Immutable 可以给你的应用带来性能提升，也可以带来更简单的编程和调试体验。这是因为，与那些在整个应用中可被随意篡改的数据相比，永远不变的数据更容易追踪，推导。

特别来说，在 Web 应用中对于 Immutable 的使用，可以让复杂的变化检测机制得以简单快速的实现。从而确保代价高昂的 DOM 更新过程只在真正需要的时候进行（这也是 React 性能方面优于其他类库的基石）。

7.a.4 store

```
function todoApp(state = {}, action) {
  return {
    todos: todos(state.todos, action),
```

```
  }
}

const state = {};

state = todoApp(state, { type: 'ADD_TODO', text: 'Go to swimming pool' });
state = {
  todos: [{ text: 'Go to swimming pool', completed: false }],
};
state = todoApp(state, { type: 'TOGGLE_TODO', index: 0 });
state = {
  todos: [{ text: 'Go to swimming pool', completed: true }],
};

console.log(state);

// Or
import { combineReducers } from 'redux'

function todoApp(state = {}, action) {
  return {
    todos: todos(state.todos, action),
  }
}
```

TODO: 多个 state

7.b react-redux

React Redux 组件 `<Provider>` 通过借用 React Context，从而实现所有组件都可以访问当前 store。

```
import React from 'react'
import { render } from 'react-dom'
import { Provider } from 'react-redux'
import { createStore } from 'redux'
import todoApp from './reducers'
import App from './components/App'

let store = createStore(todoApp)

render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('root')
)
```

- 容器组件（Smart/Container Components）
- 展示组件（Dumb/Presentational Components）

容器组件从 Redux 读取数据并派发到展示组件。

7.b.1 展示组件

Function Component 只从 props 获取数据。

```
import React from 'react'
import PropTypes from 'prop-types'
import Todo from './Todo'

const TodoList = ({ todos, onTodoClick }) => (
  <ul>
    {todos.map((todo, index) => (
      <Todo key={index} {...todo} onClick={() => onTodoClick(index)} />
    ))}
  </ul>
)

TodoList.propTypes = {
  todos: PropTypes.arrayOf(
    PropTypes.shape({
      id: PropTypes.number.isRequired,
      completed: PropTypes.bool.isRequired,
      text: PropTypes.string.isRequired
    }).isRequired
  ).isRequired,
  onTodoClick: PropTypes.func.isRequired
}

export default TodoList
```

7.b.2 容器组件

通过 Redux 注入 props。

```
import { connect } from 'react-redux'

const toggleTodo = (id) => {
  return { type: 'TOGGLE_TODO', index: id };
};

const mapStateToProps = state => {
  return {
    todos: state.todos,
  };
};

const mapDispatchToProps = dispatch => {
  return {
    onTodoClick: (id) => {
```

```
      dispatch(toggleTodo(id)),
    },
  };
};

// Or

const mapDispatchToProps = (dispatch) => {
  return bindActionCreators({
    onTodoClick: (id) => toggleTodo(id),
  }, dispatch)
}

const VisibleTodoList = connect(
  mapStateToProps,
  mapDispatchToProps
)(TodoList)

export default VisibleTodoList
```

connect API 连接传入组件和 Redux，其中第一个参数为 `mapStateToProps`，第二个参数为 `mapDispatchToProps`。

mapStateToProps

组件将会监听 Redux store 的变化。任何时候，只要 Redux store 发生改变，mapStateToProps 函数就会被调用。该回调函数必须返回一个纯对象，这个对象会与组件的 props 合并。

如果指定了该回调函数中的第二个参数 ownProps，则该参数的值为传递到组件的 props。

mapDispatchToProps

我们通过 dispatch 一个 action 更新 store， dispatch 是 redux 提供的 API，一个方法用来将 action 和 reducer 串起来。

7.c 异步数据流

默认情况下，createStore() 所创建的 Redux store 没有使用 middleware，所以只支持 同步数据流。

A thunk is a function that wraps an expression to delay its evaluation.

```
// calculation of 1 + 2 is immediate
// x === 3
let x = 1 + 2; // 3

// calculation of 1 + 2 is delayed
// foo can be called later to perform the calculation
// foo is a thunk!
let foo = () => 1 + 2;

console.log();
```

```
// ...
```

函数是一等公民

```
const sum = (a, b) => a + b;
// const myCalculator = (sum) => (a, b) => sum(a, b);
// const myCalculator(1, 2);

const Calculator = ({ 加法, 减法, 除法, 乘法 }) => ({
  点击加号: (a, b) => 加法(a, b),
});

const sum = (a, b) => a + b;
const calculator = Calculator({ 加法: sum });

calculator.点击加号(a, b);
```

React Redux 通过 dispatch 一个 action 到 reducer 来更新 state。

如果我们将 dispatch 保存起来，稍后使用呢?

```
class DumbForecast extends React.Component {
  async getForecast() {
    const { city } = this.props;
    const { data } = await getForecast(city.id);

    dispatch({
      type: 'GET_FORECAST_SUCCESSFUL',
      data,
    });
  }

  componentDidMount() {
    this.getForecast();
  }

  render() {
    const { forecast } = this.props;

    return (
      // ...
    );
  }
}

const mapStateToProps = (state) => ({
  forecast: state.forecast,
  city: state.city,
```

```
});

const SmartForecast = connect(mapStateToProps);

export default SmartForecast;
```

大胆一点，把 getForecast 移出来

```
class DumbForecast extends React.Component {
  async getForecast() {
    const { city, getForecast } = this.props;
    await getForecast(city.id);
  }

  componentDidMount() {
    this.getForecast();
  }

  render() {
    const { forecast } = this.props;

    return (
      // ...
    );
  }
}

const mapStateToProps = (state) => ({
  forecast: state.forecast,
  city: state.city,
});

const getWeather = (weather) => ({
  type: 'GET_WEATHER',
  data: weather,
});

const mapDispatchToProps = (dispatch) => ({
  getWeather: (wether) => dispatch(getWeather(weather))
});

// ---

const getWeather = (weather) => ({
  type: 'GET_WEATHER',
  data: weather,
});

const mapDispatchToProps = (dispatch) => ({
  getWeather: (id) => APIGetWeather(id)
    .then((weather) => dispatch(getWeather(weather)))
});
```

```
// --
import APIGetWeather from './APIGetWeather';

// dispatch?

const getWeather = (id) => (dispatch) => {
  dispatch({});

  return APIGetWeather(id)
    .then((weather) => {
      return dispatch({
        type: 'GET_WEATHER',
        data: weather,
      });
    })
}

dispatch((dispatch) => {
  dispatch({});

  return APIGetWeather(id)
    .then((weather) => {
      return dispatch({
        type: 'GET_WEATHER',
        data: weather,
      });
    })
})

dispatch({
  type: 'GET_WEATHER',
  data: weather,
});

const dispatch = (action) => {
  if (typeof action === 'function') {
    刷新store(action(dispatch));

    return;
  }

  // 同步
  刷新store(action);
}

const mapDispatchToProps = (dispatch) => ({
  getWeather: (id) => dispatch(getWeather(id)),
  // getWeather: (id) => getWeather(id)(dispatch),
});

const mapDispatchToProps = (dispatch) => ({
  getForecast: async (city) => {
    const { data } = await getForecast(city.id);
```

```
    dispatch({
      type: 'GET_FORECAST_SUCCESSFUL',
      data,
    });
  }
})

const SmartForecast = connect(mapStateToProps);
```

redux-thunk

```
import { createStore, applyMiddleware } from 'redux';
import thunk from 'redux-thunk';
import reducer from './reducers/index';

const store = createStore(reducer, applyMiddleware(thunk));
```

```
class DumbForecast extends React.Component {
  componentDidMount() {
    const { getForecast } = this.props;

    getForecast();
  }

  render() {
    const { forecast } = this.props;

    return (
      // ...
    );
  }
}

const mapStateToProps = (state) => ({
  forecast: state.forecast,
  city: state.city,
});

const getForecast = (dispatch, getState) => {
  const { city } = getState();

  const { data } = await getForecast(city.id);

  dispatch({
    type: 'GET_FORECAST_SUCCESSFUL',
    data,
  });
};
```

```
const mapDispatchToProps = (dispatch) => ({
  getForecast: dispatch(getForecast),
});

const SmartForecast = connect(mapStateToProps);
```

6. CSS in JS

6.a 关注点

Separation of Concerns（关注点分离）

各种技术只负责自己的领域，不要混合在一起，形成耦合。对于网页开发来说，主要是三种技术分离。

- HTML 语言：负责网页的结构，又称语义层
- CSS 语言：负责网页的样式，又称视觉层
- JavaScript 语言：负责网页的逻辑和交互，又称逻辑层或交互层

这种写法是不被接受的。

```
<h1 style="color:red;font-size:46px;"  onclick="alert('Hi')">
  Hello World
</h1>
```

就近维护 关注点混合

```
import React from 'react';

const styles = {
  header: {
    color: 'red',
    fontSize: '16px',
  },
};

const MyHeader = () => (
  <h1 style={styles.header}>
    Hello world!
  </h1>
);
```

React 在一个文件里面封装了结构、样式和逻辑，完全违背了"关注点分离"的原则。

但是，这有利于**组件的隔离**。每个组件包含了所有需要用到的代码，不依赖外部，组件之间没有耦合，很方便复用。

由于 CSS 的封装非常弱，导致了一系列的第三方库，用来加强 React 的 CSS 操作。它们统称为 CSS in JS，意思就是使用 JS 语言写 CSS。

我们使用的 `css module` 就是其中一种。

https://github.com/MicheleBertoli/css-in-js

6.b PostCSS, less, sass?

CSS in JS 使用 JavaScript 的语法，是 JavaScript 脚本的一部分，不用从头学习一套专用的 API，也不会多一道编译步骤。

6.c styled-components

- 使用`模板字符串`和`Scss-like 语法`
- 标签+样式组合后是一个大写字母开头表示的组件，比如`<Link />`
- 可以使用 `JavaScript` 的一些语法特性
- theme

```
import styled from 'styled-components';

const Link = styled.a`
  font-size: 16px;
  color: blue;

  &:hover {
    color: red; // <Thing> when hovered
  }

  text-decoration: ${(props) => props.underline ? 'underline' : 'none'};
`;

<Link underline href="https://google.com">Google</Link>
```

6.c.1 Styling normal React components

```
const Button = ({
  className,
  children,
  ...props
}) => (
  <button className={className} {...props}>{children}</button>
);

const CloseButton = styled(Button)`
  font-size: 16px;
  border-radius: 100%;
  ...
`;
```

```
const Card = () => (
  <div>
    ...
    <CloseButton>X</CloseButton>
  </div>
)
```

6.c.2 Styling styled components

```
const Button = styled.button`
  font-size: 16px;
`;

const NakedButton = styled(Button)`
  outline: 0;
  border: 0;
  cursor: pointer;
  padding: 0;
  text-align-left;
`;
```

6.c.3 Theming

```
// ./src/theme.js

export default {
  color: {
    primary: '#a6d4fa',
    secondary: '#f6a5c0',
    error: '#e57373',
    warning: '#ffb74d',
    info: '#64b5f6',
    success: '#81c784',
    white: '#FFFFFF',
    gray: '#e0e0e0',
    black: '#000000',
    border: '#DADADA',
  },
}

// ./src/index.js
import { ThemeProvider } from 'styled-components';

<ThemeProvider theme={theme}>
  <App />
</ThemeProvider>
```

```
// ./src/Button.js
import styled from 'styled-components';
import { rgba, darken } from 'polished';

const disabledStyles = (props) => {
  const backgroundColor = '#3c3c3c';

  return css`
  cursor: not-allowed;
  background-color: ${backgroundColor};
  color: ${props => rgba(props.theme.color.white, 0.8)};

  &:hover {
    background-color: ${backgroundColor};
  }
`;
}

const Button = styled.button`
  ${(props) => {
    const fontSize = {
      sm: '0.75rem',
      base: '1rem',
      lg: '1.25rem',
    }

    const padding = {
      sm: '4px 5px',
      base: '6px 8px',
      lg: '8px 11px',
    }

    return css`
      font-size: ${fontSize[props.size]};
      padding: ${padding[props.size]};
    `;
  }}

  ${(props) => {
    switch(props.variant) {
      case 'primary':
        return css`
          background-color: ${(props) => props.theme.color.primary};
          border-color: ${(props) => props.theme.color.primary};

          $:hover {
            background-color: ${(props) => darken(0.2,
props.theme.color.primary)};
          }
        `;

      case 'disabled':
        return disabledStyles(props);
```

```
        case 'secondary':
        // TODO:

        case 'info':
        // TODO:

        case 'success':
        // TODO:

        case 'danger':
        // TODO:

        default:
          return css`
            border: 1px solid ${(props) => props.theme.color.border};
            background-color: ${(props) => props.theme.color.white};
            color: ${props => rgba(props.theme.color.black, 0.8)};

            $:hover {
              background-color: ${(props) => props.theme.color.gray};
            }
          `;
      }
    }}

    &:disabled {
      ${disabledStyles}
    }
  `;
```

6.d TODO: Styled System