React

运行机制

- React DOM 在渲染所有输入内容之前，默认会进行转义。所有的内容在渲染之前都被转换成了字符串
- react组件更新是重新调用render方法，不是重新挂在整个组件，挂在组件在组件的生命周期中只会发生一次

react的原则或理念

- React 认为渲染逻辑本质上与其他 UI 逻辑内在耦合
  - 比如，在 UI 中需要绑定处理事件、在某些时刻状态发生变化时需要通知到 UI，以及需要在 UI 中展示准备好的数据
- React 组件一直更像是函数
- 组件内部命名
  - 建议从组件自身的角度命名 props，而不是依赖于调用组件的上下文命名。

react组件

- 是什么
  - 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。
- 写法
  - 函数组件
    - 这个函数组件写法和class组件的写法是等价的

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
```

    - 它接收唯一带有数据的 "props"（代表属性）对象与并返回一个 React 元素
    - 它本质上就是 JavaScript 函数
  - class组件
    - 这个class组件的写法和函数组件的写法是等价的

```
class Welcome extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}
```

    - 将函数组件转成class组件

    - https://codepen.io/gaearon/pen/amqdNA?editors=0010 示例
    - 最大的麻烦事this的指向和绑定 劣势
- API
  - 当 React 元素为用户自定义组件时，它会将组件上的属性（attributes）以及子组件（children）转换为单个对象传递给组件，这个对象被称之为 "props"。

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}

const element = <Welcome name="Sara" />;
```

  - state
    - state 是私有的，并且完全受控于当前组件。
    - 注意点：
      - 不要直接修改 State，而是应该使用 setState()
      - 构造函数是唯一可以给 this.state 赋值的地方，其他地方只能修改用setState修改state的值
      - State 的更新可能是异步的
        - 出于性能考虑，React 可能会把多个 setState() 调用合并成一个调用
        - 不要依赖state或props的值来更新到的state。

```
// Wrong
this.setState({
  counter: this.state.counter + this.props.increment,
});
```

        - 可以让setState接受一个函数来使用state的值：

```
this.setState((state, props) => ({
  counter: state.counter + props.increment
}));
```

- 使用习惯
  - 根据经验来看，如果 UI 中有一部分被多次使用（Button，Panel，Avatar），或者组件本身就足够基本（App，FeedStory，Comment），那么它就是一个可提取出独立组件的候选项。
- 注意点
  - React 会将以小写字母开头的组件视为原生 DOM 标签 组件名称必须以大写字母开头
  - 所有 React 组件都必须像纯函数一样保护它们的 props 不被更改

React.createElement() & React元素

```
const element = React.createElement(
  'h1',
  {className: 'greeting'},
  'Hello, world!'
);
```

- "React元素"

```
// 注意：这是简化过的结构
const element = {
  type: 'h1',
  props: {
    className: 'greeting',
    children: 'Hello, world!'
  }
};
```

  - 会预先执行一些检查，以帮助你编写无错代码，但实际上它创建了一个这样的对象（这些对象被称为 "React 元素"）：
- React元素
  - 是什么
    - 是个对象

```
// 注意：这是简化过的结构
const element = {
  type: 'h1',
  props: {
    className: 'greeting',
    children: 'Hello, world!'
  }
};
```

    - "React 元素"描述了你希望在屏幕上看到的内容。React 通过读取这些对象，然后使用它们来构建 DOM 以及保持随时更新。
  - 根节点
    - "根" DOM 节点。该节点内的所有内容都将由 React DOM 管理。

```
const element = <h1>Hello, world</h1>;
const root = ReactDOM.createRoot(
  document.getElementById('root')
);
root.render(element);
```

    - 将一个 React 元素渲染到根 DOM 节点中 https://codepen.io/gaearon/pen/edyBaE?editors=1010
    -
  - 性质
    - React 元素是创建开销极小的普通对象。
    - 与DOM元素相比
      - React DOM 会负责更新 DOM 来与 React 元素保持一致。
      - 一旦被创建，你就无法更改它的子元素或者属性。一个元素就像电影的单帧：它代表了某个特定时刻的 UI。
      - React 元素是不可变对象
  - 例子：
    - 更新 UI 唯一的方式是创建一个全新的元素，并将其传入 root.render()。
    - https://codepen.io/gaearon/pen/gwoJeZ?editors=1010
    - 不要怀疑，React就这保性。一直创建，一直渲染，能这么霸气是因为React有个好性能的diff算法。
    - 在这个例子中，尽管每一秒都会新建一个描述整个 UI 树的元素，React DOM 只会更新实际改变了的内容，也就是例子中的文本节点。

Diff

- React DOM 会将元素与它们之前的状态进行比较，并只会进行必要的更新来使 DOM 达到预期的状态。

纯函数

```
function sum(a, b) {
  return a + b;
}
```

- 这样的函数被称为"纯函数"，因为该函数不会尝试更改入参，且多次调用下相同的入参始终返回相同的结果。

事件

- 注意
  - React 事件的命名采用小驼峰式（camelCase），而不是纯小写
  - 阻止默认行为
    - 必须显式的使用 preventDefault

```
function Form() {
  function handleSubmit(e) {
    e.preventDefault();
    console.log('You clicked submit.');
  }

  return (
    <form onSubmit={handleSubmit}>
      <button type="submit">Submit</button>
    </form>
  );
}
```

  - this的绑定
    - 必须谨慎对待 JSX 回调函数中的 this。在 JavaScript 中，class 的方法默认不会绑定 this。如果你忘记绑定 this.handleClick 并把它传入了 onClick，当你调用这个函数的时候 this 的值为 undefined
      - https://codepen.io/gaearon/pen/xEmzGg?editors=0010
    - 使用方法：
      - 在构造函数中给事件bind（this）

```
class LoggingButton extends React.Component {
  constructor(props) {
    super(props);
    // 为了在回调中使用 `this`，这个绑定是必不可少的
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    console.log('this is:', this);
  }

  render() {
    // 此语法确保 `handleClick` 内的 `this` 已被绑定。
    return (
      <button onClick={this.handleClick}>
        Click me
      </button>
    );
  }
}
```

      - 需要安装插件 使用 class fields 语法

```
class LoggingButton extends React.Component {
  // 此语法确保 `handleClick` 内的 `this` 已被绑定。
  handleClick = () => {
    console.log('this is:', this);
  }

  render() {
    return (
      <button onClick={this.handleClick}>
        Click me
      </button>
    );
  }
}
```

      - 给子组件传该方法会造成性能问题 使用箭头函数

```
class LoggingButton extends React.Component {
  handleClick() {
    console.log('this is:', this);
  }

  render() {
    // 此语法确保 `handleClick` 内的 `this` 已被绑定。
    return (
      <button onClick={() => this.handleClick()}>
        Click me
      </button>
    );
  }
}
```

        - 在大多数情况下，这没什么问题，但如果该回调函数作为 prop 传入子组件时，这些组件可能会进行额外的重新渲染
  - 写法

```
<button onClick={activateLasers}>
  Activate Lasers
</button>
```

  - 传参
    - 两种方式

```
<button onClick={(e) => this.deleteRow(id, e)}>Delete Row</button>
<button onClick={this.deleteRow.bind(this, id)}>Delete Row</button>
```

生命周期

- componentDidMount() 在组件已经被渲染到 DOM 中后运行
- componentWillUnmount() 组件从 DOM 中被移除后触发

hooks

- 概念 Hook 是一些可以让你在函数组件里"钩入" React state 及生命周期等特性的函数
- 特性 Hook 不能在 class 组件中使用
- 优点
  - 完全可选的。你无需重写任何已有代码就可以在一些组件中尝试 Hook
  - 100% 向后兼容的。Hook 不包含任何破坏性改动。
  - 为共享状态逻辑提供更好的原生途径
    - 什么是状态逻辑
    - Hook 使你在无需修改组件结构的情况下复用状态逻辑。
  - 相关代码写在一起而不是分散到不同的生命周期中 Hook 将组件中相互关联的部分拆分成更小的函数（比如设置订阅或请求数据），而并非强制按照生命周期划分
  - 不再需要将 React 与状态管理库结合使用 可以使用 reducer 来管理组件的内部状态
  - Hook 使你在非 class 的情况下可以使用更多的 React 特性
  - 拥抱函数，和React理念更加契合
  - Hook 提供了问题的解决方案，无需学习复杂的函数式或响应式编程技术
- 生命周期
- 用法（API）
  - props 父级传参
  - state 组件内的状态（响应数据）
  - context 组件上下文
  - refs 是什么
  - 生命周期
- 使用规则：
  - 只能在函数最外层调用 Hook。不要在循环、条件判断或者子函数中调用。
  - 只能在 React 的函数组件中调用 Hook。不要在其他 JavaScript 函数中调用
- hook有哪些
  - react内置的hook
    - useState方法
      - 通过在函数组件里调用它来给组件添加一些内部 state
      - React 会在重复渲染时保留这个 state
      - 它类似 class 组件的this.setState，但是它不会把新的 state 和旧的 state 进行合并
      - 唯一的参数就是初始 state
    - useEffect
      - 给函数组件增加了操作副作用的能力
      - 它跟 class 组件中的 componentDidMount、componentDidUpdate 和 componentWillUnmount 具有相同的用途，只不过被合并成了一个 API
      - 在完成对 DOM 的更改后运行 默认情况下，React 会在每次渲染后调用副作用函数 包括第一次渲染的时候
      - 可以达何到组件的 props 和 state
  - 自定义hook
- 其他提及的内容 reducer 管理组件的内部状态
- 和类组件的对比 calss组件的缺点
  - class中this的指向很容易造成疑惑
  - 代码冗余
  - 对优化措施不友好
    - 不能很好地压缩
    - 热更新不稳定

JSX

- 是什么 是一个 JavaScript 的语法扩展
- 使用场景 在 React 中配合使用 JSX JSX 可以生成 React "元素"
- 优点
  - JSX 可以很好地描述 UI 应该呈现出它应有交互的本质形式
  - JSX 可能会使人联想到模板语言，但它具有 JavaScript 的全部功能。
- 缺点
- 机制 Babel 会把 JSX 转译成一个名为 React.createElement() 函数调用

```
const element = (
  <h1 className="greeting">
    Hello, world!
  </h1>
);
```

  - 等同于：

```
const element = React.createElement(
  'h1',
  {className: 'greeting'},
  'Hello, world!'
);
```

- 使用注意点 对于DOM上的属性：
  - 不要在大括号外添加上引号，你应该仅使用引号（对于字符串值）或大括号（对于表达式）中的一个，对于同一属性不能同时使用这两种符号
  - 因为 JSX 语法上更接近 JavaScript 而不是 HTML，所以 React DOM 使用 camelCase（小驼峰命名）来定义属性的名称

使用注意点

- 使用JSX时对于DOM上的属性：
  - 不要在大括号外添加上引号，你应该仅使用引号（对于字符串值）或大括号（对于表达式）中的一个，对于同一属性不能同时使用这两种符号
  - 因为 JSX 语法上更接近 JavaScript 而不是 HTML，所以 React DOM 使用 camelCase（小驼峰命名）来定义属性的名称
- 组件 组件名称必须以大写字母开头
- state（状态）
  - 不要直接修改 State，而是应该使用 setState()
  - 构造函数是唯一可以给 this.state 赋值的地方，其他地方只能修改用setState修改state的值
  - State 的更新可能是异步的
    - 出于性能考虑，React 可能会把多个 setState() 调用合并成一个调用
    - 不要依赖state或props的值来更新到的state。

```
// Wrong
this.setState({
  counter: this.state.counter + this.props.increment,
});
```

      - 可以让setState接受一个函数来使用state的值：

```
this.setState((state, props) => ({
  counter: state.counter + props.increment
}));
```

- 事件

自己的理解

- 有状态组件和无状态组件： 区别点在于组件中有没有使用state

条件渲染

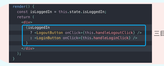
- 示例
  - 写成函数式组件

  - 用变量来代替

  - &&运算符

```
{unreadMessages.length > 0 &&
  <h2>
    You have {unreadMessages.length} unread messages.
  </h2>
}
```

  - 三目运算符



  - 控制组件显隐