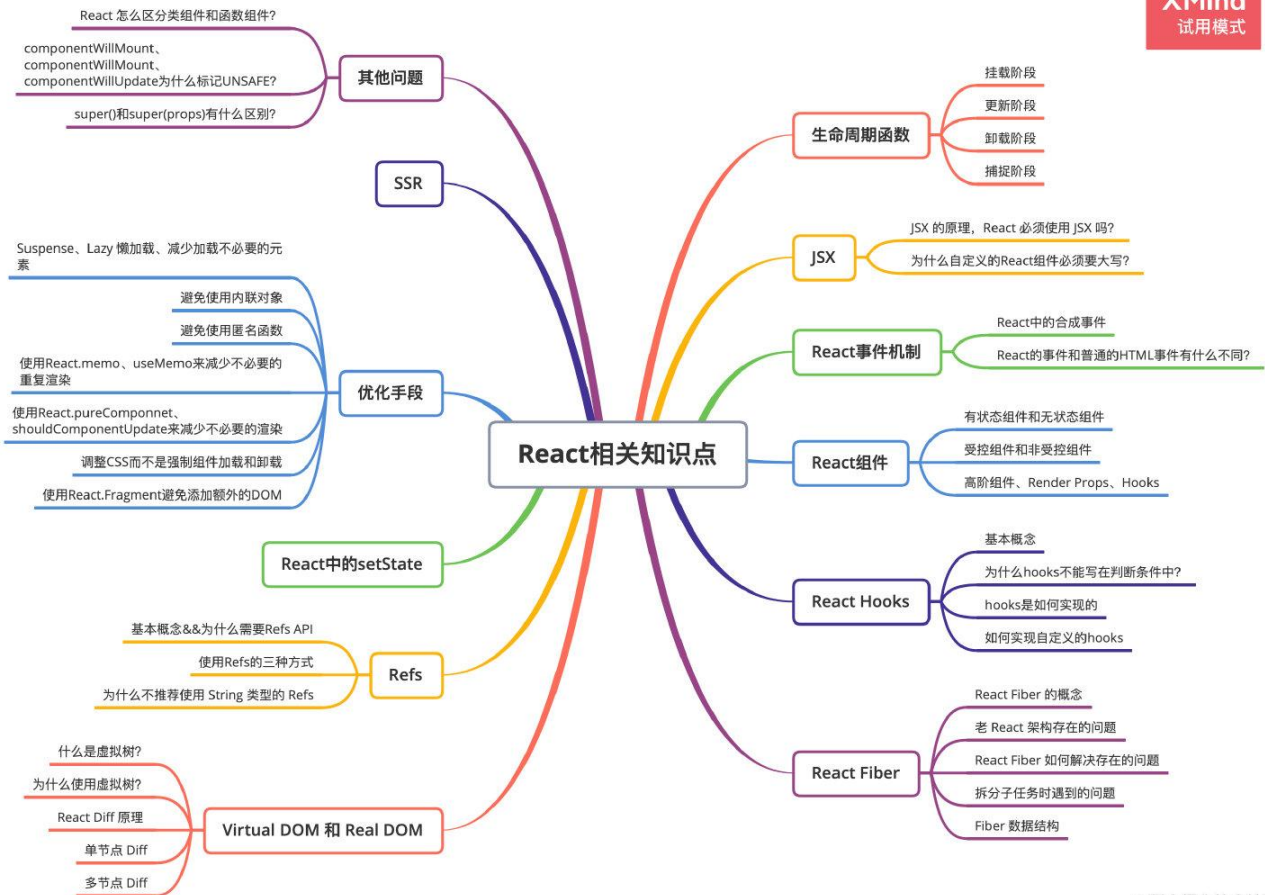


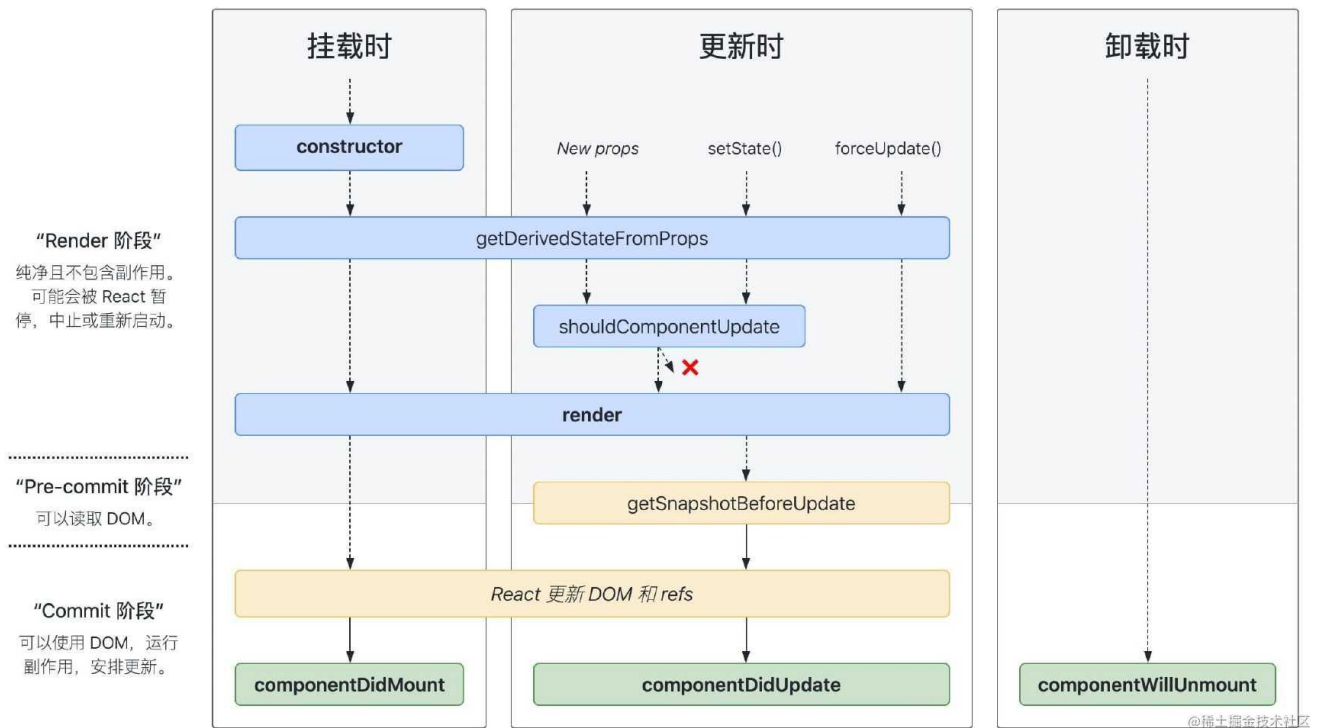
React相关知识

XMind
试用模式




@稀土掘金技术社区

正如人的生老病死一样，React中的组件从创建到销毁也要经历一些特殊的阶段。官方给我们提供了一些方法，让我们在组件的这些阶段中添加自己的代码。



挂载阶段 (Mounting)

- `constructor(props)`
- `static getDerivedStateFromProps(props, state)`
-  `UNSAFE_componentWillMount()`
- `render()`
- `componentDidMount()`

更新阶段 (Updating)

-  `UNSAFE_componentWillReceiveProps(nextProps)`
- `static getDerivedStateFromProps(props, state)`
- `shouldComponentUpdate(nextProps, nextState)`
-  `UNSAFE_componentWillUpdate(nextProps, nextState)`
- `render()`
- `getSnapshotBeforeUpdate(prevProps, prevState)`
- `componentDidUpdate(prevProps, prevState, snapshot)`

卸载阶段 (Unmounting)

- `componentWillUnmount()`

捕捉错误 (Error Handling)

- `static getDerivedStateFromError(error)`
- `componentDidCatch(error, info)`

JSX相关

JSX 的原理，React 必须使用 JSX 吗？

React 并不强制使用 JSX，实际上 JSX 只是 `React.createElement(component, props, ...children)` 函数的语法糖。

```
<MyButton color="blue" shadowSize={2}>
  Click Me
</MyButton>
```

js复制代码

会变编译为

```
React.createElement(MyButton, {
  color: "blue",
  shadowSize: 2
}, "Click Me");
```

js复制代码

Babel 在线语法转化

JSX 通过 Babel 中的 `@babel/plugin-transform-react-jsx` 插件转化成 React 代码。

为什么自定义的React组件必须要大写？

`React.createElement` 的第一个参数的类型是 `String/ReactClass type`。

- `String` 类型 React 会当做原生的 DOM 节点进行解析
- `ReactClass type` 类型 自定义组件

简而言之，babel在编译过程中会判断 JSX 组件的首字母，如果是小写，则当做原生的DOM标签解析，就编译成字符串。如果是大写，则认为是自定义组件，编译成对象。

React中的事件机制

React中的合成事件

- React合成事件首先抹平了浏览器之间的兼容性问题，在上层提供了统一的接口，开发者们由此便不必再关注烦琐的兼容性问题，可以专注于业务逻辑的开发。
- 利用 **事件委托** 的优化手段，JSX 上写的事件并没有绑定在对应的真实 DOM 上，而是通过事件代理的方式，将所有的事件都统一绑定在了 **document** 上（React 17 绑定在最顶层的 component 上）。这样的方式不仅减少了内存消耗，还能在组件挂载销毁时统一订阅和移除事件。

事件委托有两个局限性：1. 并不是所有的事件都是冒泡的 2. **mousemove**、**mouseout** 这样的事件，有冒泡但是对性能损耗较高，不适合利用委托事件。React中是如何解决的呢？对于不冒泡的事件仍然绑定在具体的元素上？——**mousemove**、**mouseout** 仍然用事件委托实现？

React的事件和普通的HTML事件有什么不同

- 对于事件名称命名方式，原生事件为全小写，react 事件采用小驼峰；
- 对于事件函数处理语法，原生事件为字符串，react 事件为函数；
- react 事件不能采用 **return false** 的方式来阻止浏览器的默认行为，而必须要地明确地调用 **preventDefault()** 来阻止默认行为。

React组件

有状态组件和无状态组件

- **有状态组件**：组件中存在state,该组件需要与用户进行交互
- **无状态组件**：一个组件中不包含state，即该组件只用于显示，不用与用户交互

受控组件和非受控组件

- **受控组件**：**<input>** 或 **<select>** 都要绑定一个 **onChange** 事件，每当表单的状态发生改变，都会被写入组件的 **state** 中，而组件的展示又由 **state** 决定，这种组件在 React 中称为受控组件。

在受控组件中，组件渲染出的状态与他的 value 相对应，react 通过这种方式消除了组件的局部状态，使得应用的整个状态可控。

- **非受控组件**：没有通过 **state** 去绑定value,表单元素的状态依然由表单元素自己管理的组件被称为非受控组件。

高阶组件、Render Props、Hooks

- **高阶组件 (HOC)**：是 React 中用于复用逻辑的一种高级技巧。HOC并不是 React API 的一部分，更像是一种设计模式。具体来说，高阶组件接受一个组件，返回一个带有更多功能更的新组件。
- **Render Props**：通过props接受一个返回react element 的函数，来动态决定自己要渲染的结果。
- **Hooks**：React Hooks 是 React 16.8.0 新增的特性。它可以让你在不编写 Class 的情况下使用 state 以及其他的 React 特性。

React Hooks

基本概念

React Hooks 是 React 16.8.0 新增的特性。它可以让你在不编写 class 的情况下使用 state 以及其他的 React 特性。

Hooks规则

- 只在顶层调用HOOK

不要在循环，条件或嵌套函数中调用 Hook。通过遵循此规则，您可以确保每次组件渲染时都以相同的顺序调用 Hook。这就是允许 React 在多个 `useState` 和 `useEffect` 调用之间能正确保留 Hook 状态的原因。

- 只在 React Functions 调用 Hooks

不要在常规 JavaScript 函数中调用 Hook。你可以在 React 函数式组件中调用 Hooks、从自定义的 Hooks 中调用 Hooks。

为什么Hooks不能写在判断条件中？

首先，我们看 `hooks` 的数据结构，`hooks` 在内部是通过 `next` 属性连接起来的，如果写在判断、循环、嵌套中会导致数组取值错位。

初次渲染的时候，按照 `useState`, `useEffect` 的顺序，把 `state`, `deps` 等按照顺序塞到 `memoizedState` 数据中。

更新的时候，按照顺序，从 `memoizedState` 中把上次记录的值拿出来。

如何自定义一个Hook？

当我们想要在两个 JavaScript 函数之间共享逻辑时，我们会将共享逻辑提取到第三个函数。组件和 Hook 都是函数，所以这种方法也适用于它们！

自定义 Hook 是一个 JavaScript 函数，其名称以“use”开头，可以调用其他Hook。例如：

js复制代码

```
import { useState, useEffect } from 'react';

function useFriendStatus(friendID) {
  const [isOnline, setIsOnline] = useState(null);

  function handleStatusChange(status) {
    setIsOnline(status.isOnline);
  }

  useEffect(() => {
    ChatAPI.subscribeToFriendStatus(friendID, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(friendID, handleStatusChange);
    };
  });

  return isOnline;
}
```

React Hooks 是如何实现的？

React Fiber

React Fiber 的概念

从架构角度来看，Fiber 是对 React 核心算法（调和过程）的重写。从编码的角度讲，Fiber 是 React 内部定义的一种数据结构，它是 Fiber 树结构的节点单位，也就是 React16 新架构下的虚拟DOM。

老 React 架构存在的问题

JavaScript 引擎与页面渲染引擎两个线程是互斥的，当一个线程执行时，另一个线程只能挂起等待。如果 JavaScript 线程长时间地占用了主线程，那么渲染层的更新就不得不长时间等待，导致页面响应度变差，让用户感觉到卡顿。

而这也是 React15中 Stack Reconciler（栈调和）所面临的问题。React15 中在渲染组件时，从开始渲染到完成整个过程是一气呵成的，无法中断。如果组件较大，那么 JS 线程会一直占用主线程，等到整颗树计算完成后，才会将执行权交给渲染引擎线程。这就会导致一些用户交互、动画等任务无法立即得到处理，导致卡顿的情况。

React Fiber 如何解决存在的问题

把渲染更新任务拆分成多个子任务，每次只做一小部分，做完后看是否还有剩余时间。如果有继续下一个任务，如果没有则将控制权交给主线程，等主线程不忙的时候再继续执行。这种策略叫做 **Cooperative Scheduling**（合作调度），操作系统常用任务调度策略之一。

合作调度主要是用来分配任务的，当有更新的任务进来的时候，不会马上去做 Diff 操作，而是先把更新放入 **Update Queue** 中，然后交给 **Scheduler** 去处理。**Scheduler** 会根据当前主线程的使用情况去处理这次更新。为了实现这种特性，使用了 **requestIdleCallback** API。对于不支持这个API 的浏览器，React 会加上 **pollyfill**。

拆分子任务时遇到的问题

- 如何拆分成子任务？一个子任务多大合适？

在 reconciliation 阶段的每个工作循环中，每次处理一个 Fiber，处理完可以中断/挂起整个工作循环。

- 怎么判断该帧是否有剩余时间？

使用了 **requestIdleCallback** API，对于不支持这个API 的浏览器，React 会加上 **pollyfill**。

- 有剩余时间怎么去调度应该执行哪一个任务？

React 为每个 **Fiber** 都设置了**优先级**。在调度任务的时候，在任务队列中选出高优先级的 **Fiber Node** 执行。

调用 **requestIdleCallback** 获取所剩时间，若执行时间超过了 **deathLine**，或者突然插入更高优先级的任务，则执行中断，保存当前结果，修改 **Fiber node** 的 **tag** 标记，设置为 **pending** 状态，迅速收尾并再调用 **requestIdleCallback**，等主线程释放出来再继续恢复任务执行时，检查 **tag** 是被中断的任务，会接着继续做任务或者重做。

Fiber 数据结构

js复制代码

```
type Fiber = {|
  tag: WorkTag, // 标识fiber类型，根据 ReactElement 组件的 type 进行生成
  key: null | string, // 该节点的唯一表示，用于diff算法的优化
  elementType: any, // 一般来讲和ReactElement组件的 type 一致
  type: any, // 一般来讲和fiber.elementType一致。一些特殊情形下，比如在开发环境下为了
  stateNode: any, // 与fiber关联的局部状态节点(比如：HostComponent类型指向与fiber节

  return: Fiber | null, // 该节点的父节点
  child: Fiber | null, // 该节点的第一个子节点
  sibling: Fiber | null, // 该节点的下一个子节点
  index: number, // 该节点的下标
  ref:
    | null
```

```

    | (((handle: mixed) => void) & { _stringRef: ?string, ... })
    | RefObject,
pendingProps: any, // 从`ReactElement`对象传入的 props. 用于和`fiber.memoizedProps`
memoizedProps: any, // 上一次生成子节点时用到的属性, 生成子节点之后保持在内存中
updateQueue: mixed, // 存储state更新的队列, 当前节点的state改动之后, 都会创建一个updateQueue
memoizedState: any, // 用于输出的state, 最终渲染所使用的state
dependencies: Dependencies | null, // 该fiber节点所依赖的(contexts, events)等
mode: TypeOfMode, // 二进制位Bitfield, 继承至父节点, 影响本fiber节点及其子树中所有节点

// Effect 副作用相关
flags: Flags, // 标志位
subtreeFlags: Flags, // 替代16.x版本中的 firstEffect, nextEffect. 当设置了 enableLegacySubtree
deletions: Array<Fiber> | null, // 存储将要被删除的子节点. 当设置了 enableNewReconciler

nextEffect: Fiber | null, // 单向链表, 指向下一个有副作用的fiber节点
firstEffect: Fiber | null, // 指向副作用链表中的第一个fiber节点
lastEffect: Fiber | null, // 指向副作用链表中的最后一个fiber节点

// 优先级相关
lanes: Lanes, // 本fiber节点的优先级
childLanes: Lanes, // 子节点的优先级
alternate: Fiber | null, // 指向内存中的另一个fiber, 每个被更新过fiber节点在内存中
|}

```

Real DOM 和 Virtual DOM

什么是虚拟树？

虚拟DOM，就是用来表示 DOM 的 JS 对象。当状态变更的时候，重新渲染这个 JS 对象，再去更新 DOM；

为什么使用虚拟树？

- 性能方面（增量更新，批量更新）
- 提高开发体验，开发效率
- 跨端，跨平台

React Diff 原理

首先，两颗树如果要完全比对的话，算法的复杂度最少也为 $O(n^3)$ ，如果直接使用该算法的话，1000个元素所需的计算量已经达到了10亿的量级，为了解决这个问题，React的 **diff** 策略是基于三个假设的：

- **假设一**：只对同级的元素进行 **Diff**，假设某 **DOM节点** 在前后两次更新中发生了跨越层级，那么 **React** 不会去尝试复用它（得不偿失）。
- **假设二**：两个不同类型的元素会产生出不同的树。假设元素由 **div** 变成了 **p**，或者由 **函数组件** 变成了 **类组件**，那么React 会删除原来的节点，直接挂载新节点。

- **假设三**：开发者可以 `key prop` 来暗示哪些子元素在多次渲染下能保持稳定。

单节点 Diff

单节点diff时，首先会判断 `current` 树上是否有该节点，其次判断 `key`（所以，使用 `map` 来渲染 `jsx` 时务必加上 `key`），`elementType` 是否是相同的，如果是，则复用该节点。否则，删除 `current` 树该层级下的所有节点。

多节点 Diff

虽然在正常的 Diff 过程中有 **新增**、**删除**、**更新** 三种情况，但是 **React** 团队在开发过程中发下，相比于 **新增** 和 **删除**，**更新** 组件发生的频率更高，所以 Diff 算法会优先判断是否属于 **更新**。具体参考：

React多节点Diff

Refs

基本概念

Refs 提供了一种方式，允许我们访问DOM节点或者在 `render` 方法中创建的 **React** 元素。

一般来说，我们无需通过直接操作DOM。直接操作DOM也与react的Virtual Dom理念相悖，但是某些时候，我们仍然需要用ref去获取真实的DOM节点。

使用Refs的三种方式

- **过时 API：String 类型的 Refs**

js复制代码

```
class MyComponent extends Component {
  renderRow = (index) => {
    // This won't work. Ref will get attached to DataTable rather than MyCom
    return <input ref={'input-' + index} />;

    // This would work though! Callback refs are awesome.
    return <input ref={input => this['input-' + index] = input} />;
  }

  render() {
    return <DataTable data={this.props.data} renderRow={this.renderRow} />
  }
}
```

- **回调函数：**

```

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

    this.textInput = null;

    this.setTextInputRef = element => {
      this.textInput = element;
    };

    this.focusTextInput = () => {
      // 使用原生 DOM API 使 text 输入框获得焦点
      if (this.textInput) this.textInput.focus();
    };
  }

  componentDidMount() {
    // 组件挂载后，让文本框自动获得焦点
    this.focusTextInput();
  }

  render() {
    // 使用 `ref` 的回调函数将 text 输入框 DOM 节点的引用存储到 React
    // 实例上（比如 this.textInput）
    return (
      <div>
        <input
          type="text"
          ref={this.setTextInputRef}
        />
        <input
          type="button"
          value="Focus the text input"
          onClick={this.focusTextInput}
        />
      </div>
    );
  }
}

```

传递一个函数。这个函数中接受 React 组件实例或 HTML DOM 元素作为参数，以使它们能在其他地方被存储和访问。

- **createRef API:**

```

class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.myRef = React.createRef();
  }
  render() {
    return <div ref={this.myRef} />;
  }
}

```

```
}  
}
```

该方法在 React 16.3 版本引入。但仅在class 组件中生效，函数组件中可以使用 forwardRef

为什么不推荐使用 String 类型的Refs

使用这种方法存在一些问题：

```
class MyComponent extends Component {  
  renderRow = (index) => {  
    // This won't work. Ref will get attached to DataTable rather than MyCom  
    return <input ref={'input-' + index} />;  
  
    // This would work though! Callback refs are awesome.  
    return <input ref={input => this['input-' + index] = input} />;  
  }  
  
  render() {  
    return <DataTable data={this.props.data} renderRow={this.renderRow} />  
  }  
}
```

js复制代码

1. 由于它无法知道this，所以需要React去跟踪当前渲染的组件。这使得React变得比较慢。
2. 上述例子，string类型的refs写法会让ref被放置在DataTable组件中，而不是MyComponent中。通过MyComponent不能访问到对应的ref。
3. 如果一个库在传递的子组件（子元素）上放置了一个ref，那用户就无法在它上面再放一个ref了。但函数式可以实现这种组合。

React 作者gaearon对该问题的回复：[github.com/facebook/re...](https://github.com/facebook/react/issues/12217)

React 中的 setState

setState 是同步的还是异步的？

在 React 中，大多数情况下 setState 是合并操作、延迟更新的，setState后并不能马上拿到结果。但是 React 中的 setState 也并不是绝对异步的。在 React 源代码中通过 `isBatchingUpdates` 来判断 setState 是现存进 state 队列还是直接更新。如果值为 true 则执行异步操作，为 false 则直接更新。

在 React 能控制的地方，`isBatchingUpdates` 为 true，比如 React 合成事件、声明周期函数里。在 React 无法控制的情况下，比如原生事件：`addEventListener`、`setInterval`、`setTimeout` 等事件中，就只能同步更新。

多次 `setState` 会发生什么情况？

由于 `setState` 是合并操作、延迟更新的，所以正常情况多次 `setState` 会以最后一次赋值的 `state` 为准。

为什么 `setState` 要设计成异步的？

1. **性能优化**：一般认为，做异步设计是为了性能优化、减少渲染次数。假设 `setState` 是同步的，那意味着每执行一次都会重新 diff + dom 修改，这从性能上来说是非常不好的。
2. **保持内部一致性**：如果将 `state` 改为同步更新，那尽管 `state` 的更新是同步的，但是 `props` 不是。

优化手段

`Suspense`、`Lazy` 懒加载、减少加载不必要的元素

延迟加载实际上不可见的组件，React 加载的组件越少，首次进入的加载速度就越快。

避免使用内联对象

使用内联对象时，React 会在每次渲染时重新创建对此对象的引用，这会导致接受此对象的组件将其视为不同的对象，该组件对于 `prop` 的浅比较永远返回 false，导致组件一直重新渲染。

常见的场景有：显示设定内联样式，就会使组件重新渲染，可能会导致性能问题（见下文例子），为了解决这个问题，我们可以保证该对象只初始化一次，指向相同引用。另外一种传递一个对象，同样会在渲染时创建不同的引用，也有可能性能问题，我们可以利用 ES6 扩展运算符将传递的对象解构。

js复制代码

```
// Bad !!!
function Component(props) {
  const aProp = { someProp: 'someValue' };
  return <AnotherComponnet style={{ margin: 0 }} aProp={aProp} />
}

// Good !!!
const styles = { margin: 0 };
function Component(props) {
  const aProp = { someProp: 'someValue' };
  return <AnotherComponnet style={style} {...aProp} />
}
```

避免使用匿名函数

类似于避免使用内联对象，当一个参数为函数时，为了保持对作为 prop 传递给React组件的函数的相同引用，可以将其声明为该类组件的方法，或者使用 useCallback 钩子。

js复制代码

```
// Bad !!!
function Component(props) {
  return <AnotherComponent onChange={() => props.callback(props.id)} />
}

// Good !!!
function Component(props) {
  const handleChange = useCallback(() => props.callback(props.id), [props.id])
  return <AnotherComponent onChange={handleChange} />
}

// Good !!!
class Component extends React.Component {
  handleChange = () => {
    this.props.callback(this.props.id)
  }
  render() {
    return <AnotherComponent onChange={this.handleChange} />
  }
}
```

使用React.memo、useMemo来减少不必要的重复渲染

如果你的组件在相同 props 的情况下得到相同的结果，那么你可以通过将其包装在 `React.memo` 中调用，以此通过记忆组件渲染结果的方式来提高组件的性能表现。这意味着在这种情况下，React 将跳过渲染组件的操作并直接复用最近一次渲染的结果。

默认情况下其只会对复杂对象做浅层对比，如果你想要控制对比过程，那么请将自定义的比较函数通过第二个参数传入来实现。 [文档](#)

js复制代码

```
function MyComponent(props) {
  /* 使用 props 渲染 */
}

function areEqual(prevProps, nextProps) {
  /*
   如果把 nextProps 传入 render 方法的返回结果与
   将 prevProps 传入 render 方法的返回结果一致则返回 true，
   否则返回 false
  */
}

export default React.memo(MyComponent, areEqual);
```

使用React.pureComponent、shouldComponentUpdate来减少不必要的渲染

父组件状态的每次更新，都会导致子组件的重新渲染，即使是传入相同 props。这对于大型组件例如组件树来说是非常消耗性能的。可以在 `shouldComponentUpdate` 中将上一次的 props 和 state 和下一次的 props 和 state 的值进行比对来决定是否更新组件。

React.PureComponent 中的 `shouldComponentUpdate()` 仅作对象的浅层比较。如果对象中包含复杂的数据结构，则有可能因为无法检查深层的差别，产生错误的比对结果。仅在你的 `props` 和 `state` 较为简单时，才使用 React.PureComponent。

调整CSS而不是强制组件加载和卸载

当我们想要对某个模块进行显隐操作时，如果加载/卸载的组件“很重”，这类操作可能非常消耗性能甚至导致延迟，在这些情况下，最好通过CSS隐藏它。

使用React.Fragment避免添加额外的DOM

React 中规定组件中必须有一个父元素，当我们同时显示多个元素时，就不得不额外添加一个“多余”的父元素。

```
function Component() {
  return (
    <div>
      <h1>Hello world!</h1>
      <h1>Hello there!</h1>
      <h1>Hello there again!</h1>
    </div>
  )
}
```

js复制代码

实际上页面上的元素越多，加载所需的时间就越多。为了减少不必要的加载时间，我们可以使用 React.Fragment 来避免创建不必要的元素。

```
function Component() {
  return (
    <React.Fragment>
      <h1>Hello world!</h1>
      <h1>Hello there!</h1>
      <h1>Hello there again!</h1>
    </React.Fragment>
  )
}
```

js复制代码

服务端渲染 SSR

待完善...

React其他相关问题

React 怎么区分类组件和函数组件？

由于本质上 `Class` 组件和 `Function` 组件都是函数，所以只能在 `Component` 组件上加上 `isReactComponent` 属性来区分函数组件和类组件。

```
function Component(props, context, updater) {  
  ...  
}  
// 加上属性 用来区分函数组件和类组件  
Component.prototype.isReactComponent = {};
```

js复制代码

`componentWillMount`、`componentDidMount`、`componentWillUpdate`为什么标记UNSAFE？

首先，这三个声明周期函数是在 `React` 的 `render` 阶段执行的。其次，在 `concurrent|blocking` 模式下，`render` 中的任务可能执行一半的时候会被高优先级的任务打断，然后重新执行 `render` 阶段。所以这几个生命周期函数可能会存在被多次调用的情况。

`super()`和`super(props)`有什么区别？

在 `React` 组件挂载之前，会调用它的构造函数。在为 `React.Component` 子类实现构造函数时，应在其他语句之前调用 `super(props)`。否则，`this.props` 在构造函数中可能会出现未定义的 bug。

如果不初始化 `state` 或不进行方法绑定，则不需要为 `React` 组件实现构造函数。