性能优化相关API

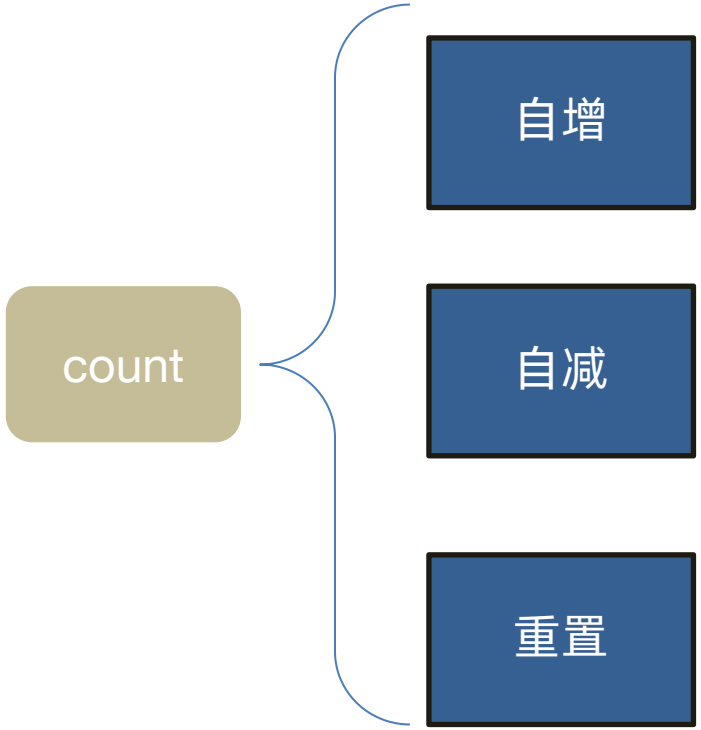
useMemo/React.memo/useCallback

黑马程序员
www.itheima.com

传智教育旗下
高端IT教育品牌

useReducer

作用：和useState的作用类似，用来管理相对复杂的状态数据



useReducer-基础用法

1. 定义一个reducer函数（根据不同的action返回不同的新状态）
2. 在组件中调用useReducer，并传入reducer函数和状态的初始值
3. 事件发生时，通过dispatch函数分派一个action对象（通知reducer要返回哪个新状态并渲染UI）

```
function reducer (state, action) {
  // 根据不同的action type 返回新的state
  switch (action.type) {
    case 'INC':
      return state + 1
    case 'DEC':
      return state - 1
    default:
      return state
  }
}
```

```
const [state, dispatch] = useReducer(reducer, 0)
```

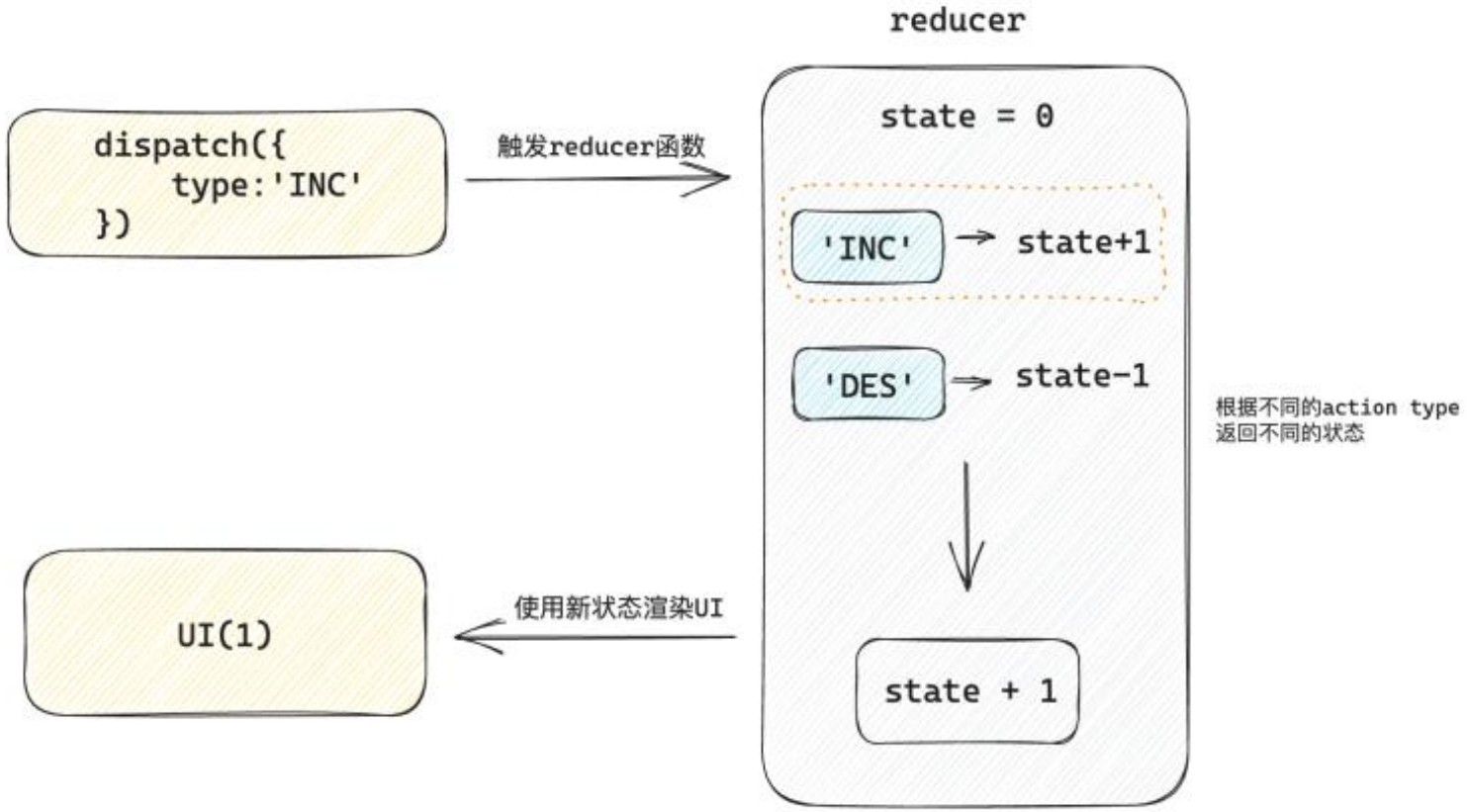
```
dispatch({ type: 'INC' })
```

useReducer-分派action时传参

```
dispatch({
  type: 'SET',
  payload: 100
})
```

```
function reducer (state, action) {
  // 根据不同的action type 返回新的state
  switch (action.type) {
    case 'INC':
      return state + 1
    case 'DEC':
      return state - 1
    case 'SET':
      return action.payload
    default:
      return state
  }
}
```

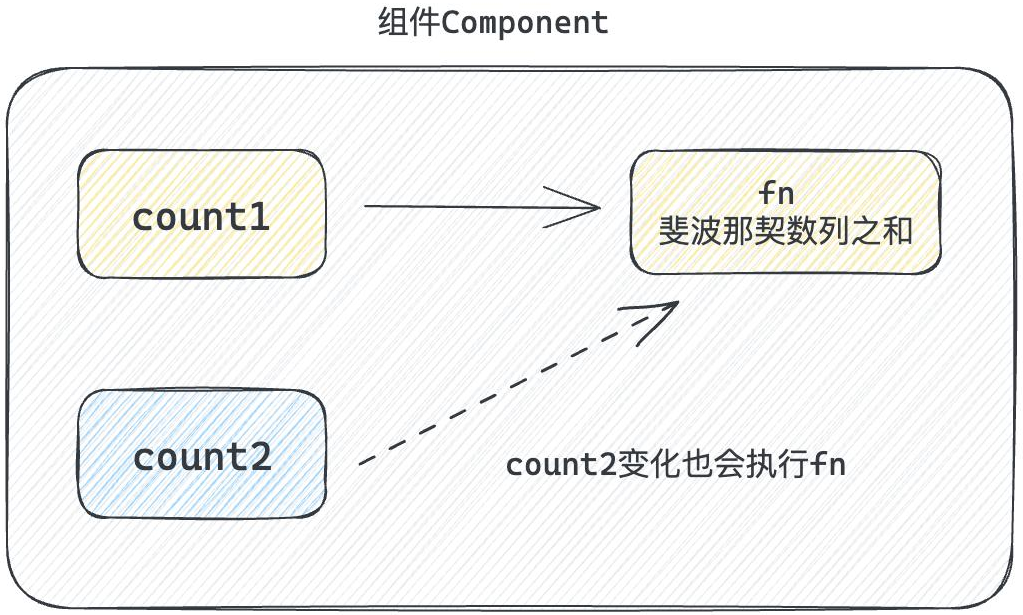
useReducer-小结



useMemo

作用：在组件每次重新渲染的时候缓存计算的结果

看个需求：



useMemo - 基础语法

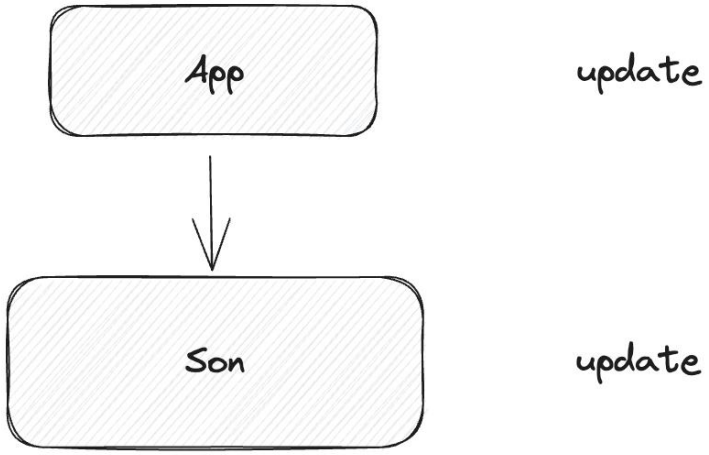
```
useMemo(() => {
  // 根据count1返回计算的结果
}, [count1])
```

说明：使用useMemo做缓存之后可以保证只有count1依赖项发生变化时才会重新计算

React.memo

作用：允许组件在Props没有改变的情况下跳过渲染

React组件默认的渲染机制：只要父组件重新渲染子组件就会重新渲染



思考：如果Son组件本身并不需要做渲染更新，是不是存在浪费？

React.memo - 基础语法

```
const MemoComponent = memo(function SomeComponent (props) {
  // ...
})
```

说明：经过memo函数包裹生成的缓存组件只有在props发生变化的时候才会重新渲染

React.memo - props的比较机制

机制: 在使用memo缓存组件之后，React会对每一个 prop 使用 Object.is 比较新值和老值，返回true，表示没有变化

prop是简单类型

Object.is(3, 3) => true 没有变化

prop是引用类型（对象 / 数组）

Object([], []) => false 有变化，React只关心引用是否变化

useCallback

作用：在组件多次重新渲染的时候缓存函数

```jsx
const Input = memo(function Input ({ onChange }) {
  console.log('子组件重新渲染了')
  return <input type="text" onChange={(e) => onChange(e.target.value)} />
})

function App () {
  // 传给子组件的函数
  const changeHandler = (value) => console.log(value)
  // 触发父组件重新渲染的函数
  const [count, setCount] = useState(0)
  return (
    <div className="App">
      {/* 把函数作为prop传给子组件 */}
      <Input onChange={changeHandler} />
      <button onClick={() => setCount(count + 1)}>{count}</button>
    </div>
  )
}
```

useCallback - 基础语法

```jsx

const Input = memo(function Input ({ onChange }) {
  console.log('子组件重新渲染了')
  return <input type="text" onChange={(e) => onChange(e.target.value)} />
})

function App () {
  // 传给子组件的函数
  const changeHandler = useCallback((value) => console.log(value), [])
  // 触发父组件重新渲染的函数
  const [count, setCount] = useState(0)
  return (
    <div className="App">
      {/* 把函数作为prop传给子组件 */}
      <Input onChange={changeHandler} />
      <button onClick={() => setCount(count + 1)}>{count}</button>
    </div>
  )
}
```

说明：使用useCallback包裹函数之后，函数可以保证在App重新渲染的时候保持引用稳定

React.forwardRef

使用ref暴露DOM节点给父组件

黑马程序员
www.itheima.com

传智教育旗下
高端IT教育品牌

forwardRef - 场景说明

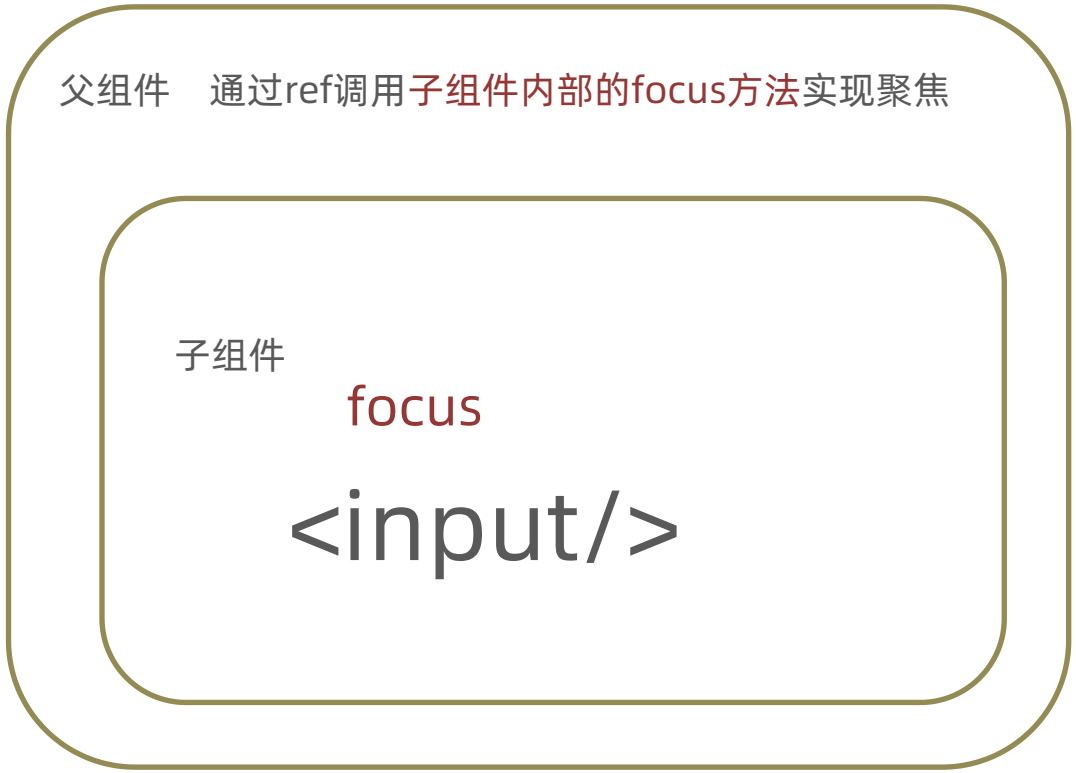
forwardRef - 语法实现

```
1  // 子组件
2  const Input = forwardRef((props,
3    return <input type="text" ref=
4  })
5
6  // 父组件
7  function App () {
8    const inputRef = useRef(null)
9    return (
10     <>
11       <Input ref={inputRef} />
12     </>
13   )
14 }
15
```

useInperativeHandle

通过ref暴露子组件中的方法

useInperativeHandlle - 场景说明



useInperativeHandlle - 场景说明

```
1  // 子组件
2  const Input = forwardRef((props, ref) => {
3    const inputRef = useRef(null)
4    // 实现聚焦逻辑函数
5    const focusHandler = () => {
6      inputRef.current.focus()
7    }
8    // 暴露函数给父组件调用
9    useImperativeHandle(ref, () => {
10     return {
11       focusHandler
12     }
13   })
14   return <input type="text" ref={inputRef} />
15 })
```

Class API

编写类组件

黑马程序员
www.itheima.com

传智教育旗下
高端IT教育品牌

类组件基础结构

类组件就是通过JS中的类来组织组件的代码

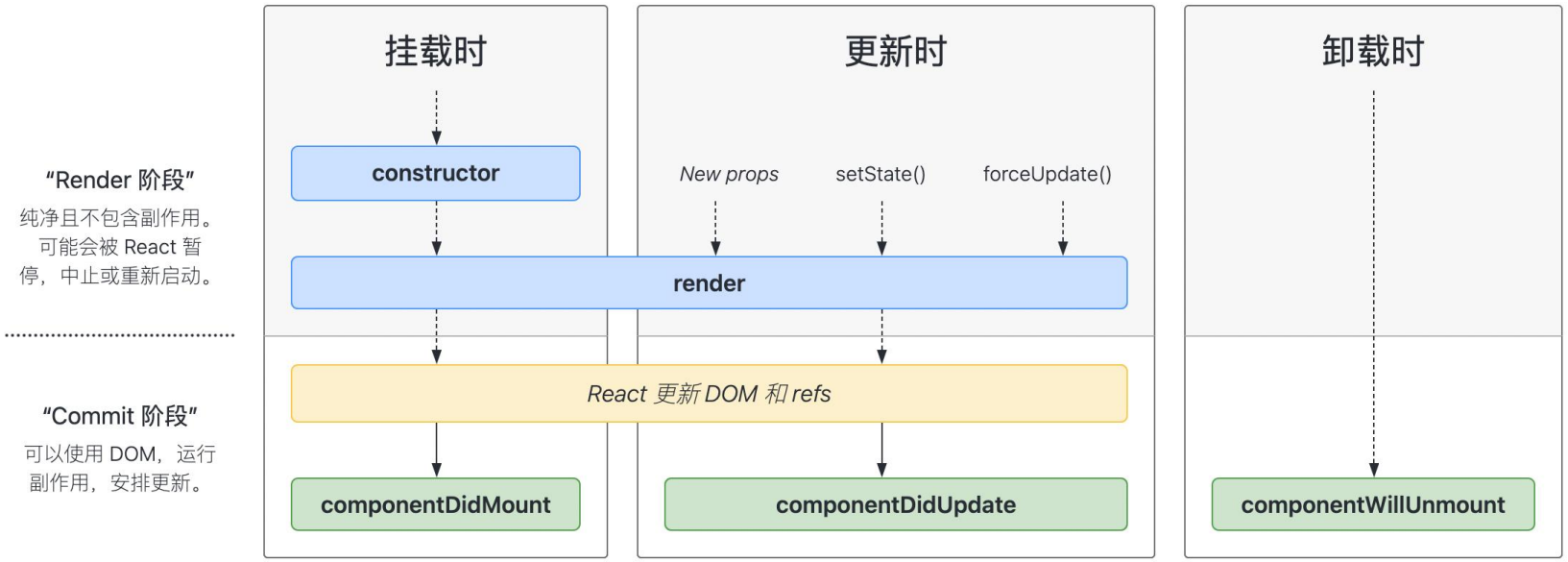
```
class Counter extends Component {
  // 定义状态变量
  state = {
    count: 0
  }
  // 事件回调
  clickHandler = () => {
    this.setState({
      count: this.state.count + 1
    })
  }

  // UI模版(JSX)
  render () {
    return <button onClick={this.clickHandler}>{this.state.count}</button>
  }
}
```

1. 通过类属性state定义状态数据　2. 通过setState方法来修改状态数据　3. 通过render来写UI模版（JSX语法一致）

类组件的生命周期函数

概念：组件从创建到销毁的各个阶段自动执行的函数就是生命周期函数



1. componentDidMount：组件挂载完毕自动执行 – 异步数据获取
2. componentWillUnmount: 组件卸载时自动执行 – 清理副作用

类组件的组件通信

概念：类组件和Hooks编写的组件在组件通信的思想上完全一致

1. 父传子：通过prop绑定数据
2. 子传父：通过prop绑定父组件中的函数，子组件调用
3. 兄弟通信：状态提升，通过父组件做桥接

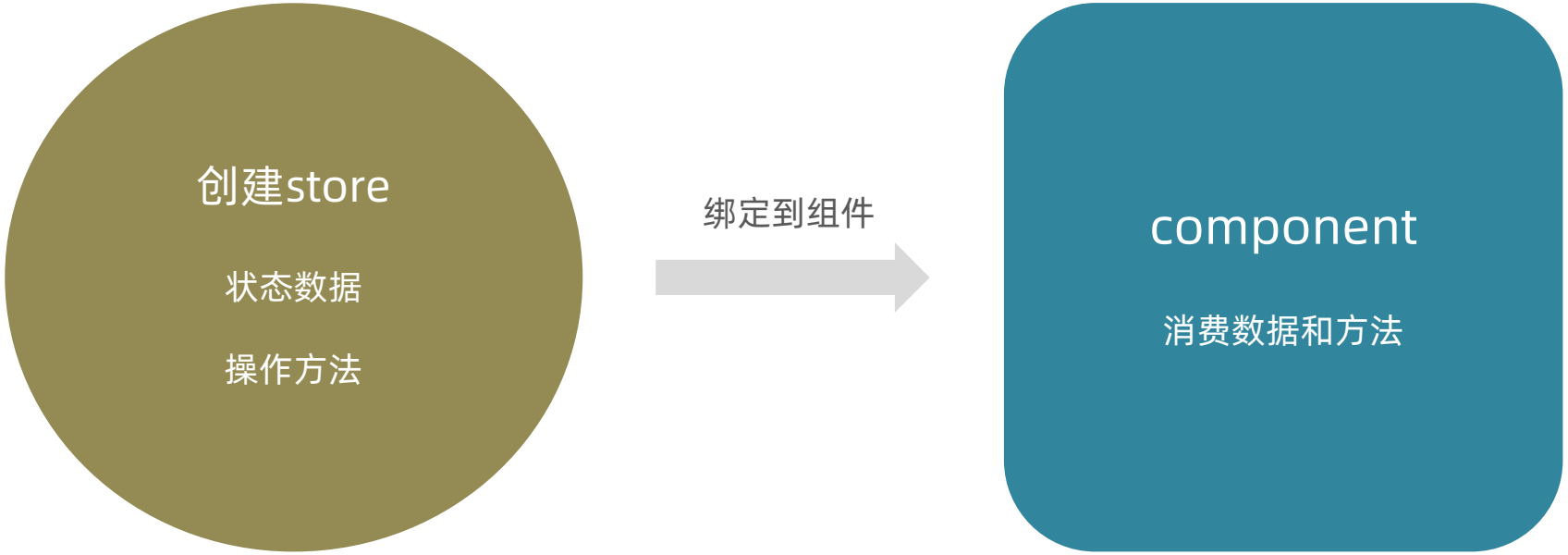
zustand

极简的状态管理工具

黑马程序员
www.itheima.com

传智教育旗下
高端IT教育品牌

zustand-快速上手



zustand-异步支持

对于异步的支持不需要特殊的操作，直接在函数中编写异步逻辑，最后只需要调用set方法传入新状态即可

```
const useStore = create((set) => {
  return {
    // 状态数据
    channelList: [],
    // 异步方法
    fetchChannelList: async () => {
      const res = await fetch(URL)
      const jsonData = await res.json()
      // 调用set方法更新状态
      set({
        channelList: jsonData.data.channels
      })
    }
  }
})
```

zustand-切片模式

场景：当单个store比较大的时候，可以采用 切片模式 进行模块拆分组合，类似于模块化