源码解析

Preact: https://preactjs.com/guide/v10/getting-started

版本： 10.8.1

与react的区别：https://preactjs.com/guide/v10/differences-to-react

Hooks Demo

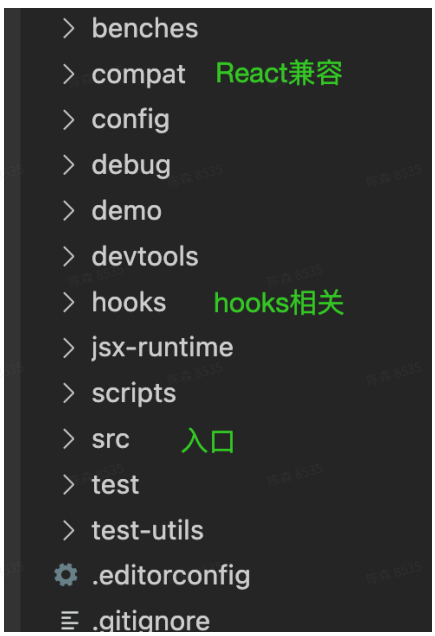
```
import { h } from 'preact'; // 就是createElement 函数的别名
import {useEffect, useState} from "preact/hooks"; //与 preact 主体是分来的 实现逻辑解耦，可自B
import style from './style.css';

const Demo = ({ user }) => {
  const [time, setTime] = useState(Date.now());
  const [count, setCount] = useState(10);

  useEffect(() => {

  }, []);

  return (
    <div class={style.profile}>
      <h1>Profile: {user}</h1>
      <p>This is the user profile for a user named { user }.</p>

      <div>Current time: {new Date(time).toLocaleString()}</div>

      <p>
        <button onClick={() => setCount((count) => count + 1)}>Click Me</button>
        {' '}
        Clicked {count} times.
      </p>
    </div>
  );
}

export default Demo;

```

Class Demo

```jsx
import { createElement, Component, Fragment } from 'preact';

export default class extends Component {
  state = { number: 0 };

  componentDidMount() {
    setInterval(_ => this.updateChildren(), 1000);
  }

  updateChildren() {
    this.setState(state => ({ number: state.number + 1 }));
  }

  render(props, state) {
    return (
      <div>
        <div>{state.number}</div>
        <>
          <div>one</div>
          <div>{state.number}</div>
          <div>three</div>
        </>
      </div>
    );
  }
}
```

项目结构：

结构风格： 分工明确，清晰解耦，内部通过可插拔方式进行逻辑组合



前置理解：

options.js 内部定义了一系列在render过程中执行的钩子，hooks中会针对特定钩子添加一些hook相关的逻辑

```
export interface Options extends preact.Options {
  /** Attach a hook that is invoked before render, mainly to check the arguments. */
  /* 在render 函数体中 第一行 进行调用 */
  _root?(
    vnode: ComponentChild,
    parent: Element | Document | ShadowRoot | DocumentFragment
  ): void;
  /** Attach a hook that is invoked before a vnode is diffed. */
  /* 在diffed之前调用*/
  _diff?(vnode: VNode): void;
  /** Attach a hook that is invoked after a tree was mounted or was updated. */
  /* 在提交之后调用 此时dom树已经是新的了*/
  _commit?(vnode: VNode, commitQueue: Component[]): void;

  /*_commit _render 在类react hooks的 实现中非常重要*/
  /** Attach a hook that is invoked before a vnode has rendered. */
  /* 在vnode render之前调用*/
  _render?(vnode: VNode): void;
  /** Attach a hook that is invoked before a hook's state is queried. */
  _hook?(component: Component, index: number, type: HookType): void;
  /** Bypass effect execution. Currenty only used in devtools for hooks inspection */
  _skipEffects?: boolean;
  /** Attach a hook that is invoked after an error is caught in a component but before cal
ling lifecycle hooks */
  _catchError(
    error: any,
    vnode: VNode,
    oldVNode: VNode | undefined,
    errorInfo: ErrorInfo | undefined
  ): void;
}

```

主体文件： src/index.js

目录结构


src
diff
catch-error.js
children.js
index.js
props.js
cjs.js
clone-element.js
component.js
constants.js
create-context.js
create-element.js
index.d.ts
index.js
internal.d.ts
jsx.d.ts
options.js
render.js
util.js

render

```
export function render(vnode, parentDom, replaceNode) {
    /**  在render 之前一个钩子函数 可以进行参数校验*/
  if (options._root) options._root(vnode, parentDom);

  // 判断是否是ssr
  let isHydrating = typeof replaceNode === 'function';

  // 为了能够支持在同一个上多次调用 `render()`
  // DOM 节点，我们需要获取对前一棵树的引用。我们的确是
 // 通过为指向的 DOM 节点分配一个新的 `_children` 属性
 // 到最后渲染的树。默认情况下，此属性不存在，这
 // 表示我们是第一次挂载一棵新树。
  let oldVNode = isHydrating
    ? null
    : (replaceNode && replaceNode._children) || parentDom._children;

  vnode = (
    (!isHydrating && replaceNode) ||
    parentDom
  )._children = createElement(Fragment, null, [vnode]);
// 这个使用parentDom的_children属性已经指向[vnode]了
// _children自定义属性
  let commitQueue = []; // components list
  diff(
    parentDom,
    vnode,
    oldVNode || EMPTY_OBJ,
    EMPTY_OBJ,
    parentDom.ownerSVGElement !== undefined,// issvg
    !isHydrating && replaceNode // excessDomChildren 这个参数用来做dom复用的作用
      ? [replaceNode]
      : oldVNode
      ? null
      : parentDom.firstChild
      ? slice.call(parentDom.childNodes)// 如果parentDom有子节点就会把整个子节点作为待复用的节点
      : null,
    commitQueue,
    !isHydrating && replaceNode // oldDom
      ? replaceNode
      : oldVNode
      ? oldVNode._dom
      : parentDom.firstChild, // oldVNode 存在 就取 oldVNode._dom (更新)
      // 不然就取 parentDom.firstChild 第一次渲染的时候
    isHydrating
  );

  // Flush all queued effects
  commitRoot(commitQueue, vnode);
}
```

create-element

```
一句话概括： 创建一个 Vnode js 对象

// 文件内的 这一个定义函数 在component中有用到
export function Fragment(props) {
  return props.children;
}
```

component.js

```
// 基础组件 构造函数 类
export function Component(props, context) {
  this.props = props;
  this.context = context;
}

// 接下来在 Component的原型上添加 一些 特定函数
//callback state更新后的回调
Component.prototype.setState = function(update, callback) {
  // 仅在第一次克隆状态到 nextState 时
  let s;
  if (this._nextState != null && this._nextState !== this.state) {
    s = this._nextState;
  } else {
    s = this._nextState = assign({}, this.state);
  }
  //  ...
  if (update) {
    // 这就是把处理后的state 放到nextState
    assign(s, update);
  }

  if (this._vnode) {
    if (callback) this._renderCallbacks.push(callback);
    // 把当前组件加入待渲染队列并渲染
    enqueueRender(this);
  }
};

Component.prototype.forceUpdate = function(callback) {
  if (this._vnode) {
    this._force = true;
    if (callback) this._renderCallbacks.push(callback);
    // 设置_force来标记是强制渲染，然后加入渲染队列并渲染。如果_force为真，则在diff渲染中不会触发组
    // 因为 forceUpdate 永远不应该调用shouldComponentUpdate
    enqueueRender(this);
  }
};

Component.prototype.render = Fragment;

// 待渲染组件队列
let rerenderQueue = [];

//异步调度器。如果支持Promise则会用Promise，否则用setTimeout
const defer = //等同于 Promise.resolve().then。
  typeof Promise == 'function'
    ? Promise.prototype.then.bind(Promise.resolve())
    : setTimeout;

// 将组件排入渲染队列
```

```
// c 为待重新渲染的队列
export function enqueueRender(c) {
    // 如果_dirty为false则设为true
    // 然后把组件加入队列中
    // 自加加rerenderCount并且如果为0则触发渲染
  if (
    (!c._dirty &&
      (c._dirty = true) &&
      rerenderQueue.push(c) &&
      !process._rerenderCount++) ||  // process._rerenderCount 为 0  才会为true ,>0 的值为
    prevDebounce !== options.debounceRendering
  ) {
    prevDebounce = options.debounceRendering;
    //执行process
    (prevDebounce || defer)(process);
  }
}

// 通过重新渲染所有排队的组件来刷新渲染队列
// 遍历队列渲染组件
function process() {
  let queue;
  while ((process._rerenderCount = rerenderQueue.length)) {
    //按深度排序，最顶级的组件的最先执行
    queue = rerenderQueue.sort((a, b) => a._vnode._depth - b._vnode._depth);
    rerenderQueue = []; // 重置
    // 暂时不要更新 `renderCount`。保持其值非零以防止不必要的
    // process() 在 `queue` 仍然被消耗时被调度调用。
    queue.some(c => {
       //如果组件需要渲染则渲染它
      if (c._dirty) renderComponent(c);
    });
  }
}


// 渲染组件
function renderComponent(component) {
  let vnode = component._vnode,
    oldDom = vnode._dom,
    parentDom = component._parentDom;

  if (parentDom) {
    let commitQueue = [];
    const oldVNode = assign({}, vnode);
    oldVNode._original = vnode._original + 1;
    // oldVNode 一开始比较难理解
    // 保证oldVNode._component 存在，去走 update的 过程
    // 两者_original 不等是为了跳过diff中的一个case
    //比较渲染
    diff(
      parentDom,
```

```
104         vnode,
105         oldVNode,
106         component._globalContext,
107         parentDom.ownerSVGElement !== undefined,
108         vnode._hydrating != null ? [oldDom] : null,
109         commitQueue,
110         oldDom == null ? getDomSibling(vnode) : oldDom,
111         vnode._hydrating
112       );
113       // 渲染完成时执行did生命周期和setState回调
114       commitRoot(commitQueue, vnode);
115
116       // 如果newDom与oldDom不一致，则调用updateParentDomPointers
117       if (vnode._dom != oldDom) {
118       // 更新 vnode._parent 的_dom
119         updateParentDomPointers(vnode);
120       }
121     }
122   }
123
```

diff : 实现生成/更新dom

```
// 比较两个虚拟node 把差别更新到dom 上
export function diff(
  parentDom,
  newVNode,
  oldVNode,
  globalContext,
  isSvg,
  excessDomChildren,
  commitQueue,
  oldDom,
  isHydrating
) {
  let tmp,
    newType = newVNode.type;
  // 在开始diff之前的钩子
  if ((tmp = options._diff)) tmp(newVNode);

  try {
    // 如果是类组件或者函数组件
    outer: if (typeof newType == 'function') {
      let c, isNew, oldProps, oldState, snapshot, clearProcessingException;
      let newProps = newVNode.props;

      tmp = newType.contextType;
      //找到所属的provider组件
      let provider = tmp && globalContext[tmp._id];
       //有tmp时，如果提供provider时为provider的value，不然为createContext的defaultValue
       // 没有则为父节点传递下来的context
      let componentContext = tmp
        ? provider
          ? provider.props.value
          : tmp._defaultValue
        : globalContext;

      //如果已经存在实例化的组件
      if (oldVNode._component) {
        c = newVNode._component = oldVNode._component;
        clearProcessingException = c._processingException = c._pendingError;
      } else {
         // class组件
        if ('prototype' in newType && newType.prototype.render) {
          newVNode._component = c = new newType(newProps, componentContext); // eslint-di
        } else {
          //函数组件的话会实例化Component
          newVNode._component = c = new Component(newProps, componentContext);
          c.constructor = newType;
          // 设置render
          c.render = doRender;
        }
        // 如果某个子孙节点组件设置了contextType静态属性，会调用sub方法把该组件添加到订阅数组中。
        //订阅，当provider组件value改变时，渲染组件
```

```
52          if (provider) provider.sub(c);
53
54          c.props = newProps;
55          if (!c.state) c.state = {};
56          c.context = componentContext;
57          c._globalContext = globalContext;
58          isNew = c._dirty = true;// 设置标识位
59          c._renderCallbacks = []; // 初始化组件上的回调数组
60        }
61
62        // Invoke getDerivedStateFromProps
63        if (c._nextState == null) {
64          c._nextState = c.state;
65        }
66        if (newType.getDerivedStateFromProps != null) {
67          if (c._nextState == c.state) {
68            c._nextState = assign({}, c._nextState);
69          }
70
71          assign(
72            c._nextState,
73            newType.getDerivedStateFromProps(newProps, c._nextState)
74          );
75        }
76
77        oldProps = c.props;
78        oldState = c.state;
79
80
81        // 如果是新创建的组件
82        if (isNew) {
83          if (
84            newType.getDerivedStateFromProps == null &&
85            c.componentWillMount != null
86          ) {
87            c.componentWillMount();
88          }
89
90          if (c.componentDidMount != null) {
91          // componentDidMount 塞进回调数组
92            c._renderCallbacks.push(c.componentDidMount);
93          }
94        } else { // 不是新创建的组件
95          if (
96            newType.getDerivedStateFromProps == null &&
97            newProps !== oldProps &&
98            c.componentWillReceiveProps != null
99          ) {
100           //没有设置getDerivedStateFromProps并且设置了componentWillReceiveProps则执行此生命周
101           c.componentWillReceiveProps(newProps, componentContext);
102         }
103         // 如果不是forceUpdate并且设置了shouldComponentUpdate则执行此生命周期在返回false的情况
```

```
        // 简单来说。就是 shouldComponentUpdate 返回false
        if (
          (!c._force &&
            c.shouldComponentUpdate != null &&
            c.shouldComponentUpdate(
              newProps,
              c._nextState,
              componentContext
            ) === false) ||
          newVNode._original === oldVNode._original
        ) {
          c.props = newProps;
          c.state = c._nextState;
          // More info about this here: https://gist.github.com/JoviDeCroock/bec5f2ce9354
          if (newVNode._original !== oldVNode._original) c._dirty = false;
          c._vnode = newVNode;
          newVNode._dom = oldVNode._dom;
          newVNode._children = oldVNode._children;
          newVNode._children.forEach(vnode => {
            if (vnode) vnode._parent = newVNode;
          });
          if (c._renderCallbacks.length) {
            commitQueue.push(c);
          }

          // 就不继续走下去了
          break outer;
        }

        if (c.componentWillUpdate != null) {
        // 执行 componentWillUpdate 生命周期
          c.componentWillUpdate(newProps, c._nextState, componentContext);
        }

        if (c.componentDidUpdate != null) {
        // 继续添加。回调。在commit 操作 更新dom之后会陆续执行这些回调函数
          c._renderCallbacks.push(() => {
            c.componentDidUpdate(oldProps, oldState, snapshot);
          });
        }
      }

      c.context = componentContext;
      c.props = newProps;
      c._vnode = newVNode;
      c._parentDom = parentDom;
    /**  在render 之前执行的 钩子*/
      let renderHook = options._render,
        count = 0;
      if ('prototype' in newType && newType.prototype.render) {
        c.state = c._nextState;
        c._dirty = false;
```

```
156
157         if (renderHook) renderHook(newVNode);
158        //执行render
159         tmp = c.render(c.props, c.state, c.context);
160       } else {
161         do {
162           c._dirty = false;
163
164           if (renderHook) renderHook(newVNode);
165
166           tmp = c.render(c.props, c.state, c.context);
167
168           // Handle setState called in render,
169           c.state = c._nextState;
170         } while (c._dirty && ++count < 25);
171       }
172
173       // Handle setState called in render,
174       c.state = c._nextState;
175
176      // 如果是Provider组件，然后调用getChildContext获取ctx对象并向下传递
177       if (c.getChildContext != null) {
178         globalContext = assign(assign({}, globalContext), c.getChildContext());
179       }
180
181       if (!isNew && c.getSnapshotBeforeUpdate != null) {
182         snapshot = c.getSnapshotBeforeUpdate(oldProps, oldState);
183       }
184
185       let isTopLevelFragment =
186         tmp != null && tmp.type === Fragment && tmp.key == null;
187       let renderResult = isTopLevelFragment ? tmp.props.children : tmp;
188
189       // 待补充
190       diffChildren(
191         parentDom,
192         Array.isArray(renderResult) ? renderResult : [renderResult],
193         newVNode,
194         oldVNode,
195         globalContext,
196         isSvg,
197         excessDomChildren,
198         commitQueue,
199         oldDom,
200         isHydrating
201       );
202
203       c.base = newVNode._dom;
204
205       // We successfully rendered this VNode, unset any stored hydration/bailout state:
206       newVNode._hydrating = null;
207
```

```
      if (c._renderCallbacks.length) {
      // 如果存在 待执行的回调。就把这个这个组件实例放进 commitQueue
        commitQueue.push(c);
      }

      if (clearProcessingException) {
        c._pendingError = c._processingException = null;
      }

      c._force = false;
    } else if (
      excessDomChildren == null &&
      newVNode._original === oldVNode._original
    ) {
      newVNode._children = oldVNode._children;
      newVNode._dom = oldVNode._dom;
    } else {
      // 对比html标签节点，'div' 这种
      //diffElementNodes 待补充
      newVNode._dom = diffElementNodes(
        oldVNode._dom,
        newVNode,
        oldVNode,
        globalContext,
        isSvg,
        excessDomChildren,
        commitQueue,
        isHydrating
      );
    }

     // 支持 diff 之后的钩子函数
    if ((tmp = options.diffed)) tmp(newVNode);
  } catch (e) {
    newVNode._original = null;
    // if hydrating or creating initial tree, bailout preserves DOM:
    if (isHydrating || excessDomChildren != null) {
      newVNode._dom = oldDom;
      newVNode._hydrating = !!isHydrating;
      excessDomChildren[excessDomChildren.indexOf(oldDom)] = null;
      // ^ could possibly be simplified to:
      // excessDomChildren.length = 0;
    }
    options._catchError(e, newVNode, oldVNode);
  }
}


export function commitRoot(commitQueue, root) {
  /**在dom 更新之后执行 _commit hook*/
  if (options._commit) options._commit(root, commitQueue);
  commitQueue.some(c => {
```

```
    // _renderCallbacks是指在preact中指每次 render 后，
    // 同步执行的操作回调列表，例如setState的第二个参数 cb、
    //  或者一些render后的生命周期函数、或者forceUpdate的回调）
    try {
      commitQueue = c._renderCallbacks;
      c._renderCallbacks = [];
      // 按顺序支持 _renderCallbacks 回调
      commitQueue.some(cb => {
        cb.call(c);
      });
    } catch (e) {
      options._catchError(e, c._vnode);
    }
  });
}



/** The `.render()` method for a PFC backing instance. */
//  专门针对 FC 的实现，执行函数 走一遍 render渲染流程，走一遍hooks 的调用
function doRender(props, state, context) {
  return this.constructor(props, context);
}
```

diffChildren【待补充】

```
diff子节点
```

diffElementNodes【待补充】

```
diff dom元素节点，非组件节点
```

create-context.js

```
import { enqueueRender } from './component';

export let i = 0;

export function createContext(defaultValue, contextId) {
  contextId = '__cC' + i++;

  const context = {
    _id: contextId,
    _defaultValue: defaultValue,
    /** @type {import('./internal').FunctionComponent} */
    Consumer(props, contextValue) {
      return props.children(contextValue);
    },
    /** @type {import('./internal').FunctionComponent} */
    Provider(props) {
      if (!this.getChildContext) {
        let subs = [];
        let ctx = {};
        ctx[contextId] = this;

        this.getChildContext = () => ctx;

        this.shouldComponentUpdate = function(_props) {
          //当value不相等时
          if (this.props.value !== _props.value) {
            subs.some(enqueueRender);
          }
        };
        // render 消费context 的组件时调用 该sub
         // 将 该组件实例添加到队列中，当value改变时渲染该组件
        this.sub = c => {
          subs.push(c);
          let old = c.componentWillUnmount;
          // 当组件卸载后从队列中删除，然后执行老的componentWillUnmount
          c.componentWillUnmount = () => {
            subs.splice(subs.indexOf(c), 1);
            if (old) old.call(c);
          };
        };
      }

      return props.children;
    }
  };

  return (context.Provider._contextRef = context.Consumer.contextType = context);
}
```

hooks系列：hooks/src/index.js

1. 标志位变量：

```
 currentComponent  // 当前正在渲染的组件
 currentIndex  // 当前组件正在执行的hook

previousComponent //上一个组件

let currentHook = 0;
let afterPaintEffects = [];

let EMPTY = [];

let oldBeforeDiff = options._diff;
let oldBeforeRender = options._render;
let oldAfterDiff = options.diffed;
let oldCommit = options._commit;
let oldBeforeUnmount = options.unmount;


const RAF_TIMEOUT = 100;
let prevRaf;
```

2. 针对options中的钩子中加入 hooks 相关逻辑

```
// 在diff执行执行 执行_diff 钩子
options._diff = vnode => {
  currentComponent = null; // 重置
  if (oldBeforeDiff) oldBeforeDiff(vnode);
};

// 开始render 函数执行之前 调用_render 函数
options._render = vnode => {
  if (oldBeforeRender) oldBeforeRender(vnode);
//  进行每次 render 的初始化操作。
// 包括执行/清理上次未处理完的 effect、初始化 hook 下标为 0、取得当前 render 的组件实例。
  currentComponent = vnode._component;
  currentIndex = 0;
  // 在每一次render过程中是从0 开始的 ，每执行一次useXX 后加一
  // 每个hook 在多次render中对于记录前一次的执行状态正是通过 currentComponent._hooks
  // 实现的
  const hooks = currentComponent.__hooks;
  if (hooks) {
    if (previousComponent === currentComponent) {
      hooks._pendingEffects = [];// 复位 当前组件hooks
      currentComponent._renderCallbacks = [];
      hooks._list.forEach(hookItem => {
        hookItem._pendingValue = EMPTY;
        hookItem._pendingArgs = undefined;
      });
    } else {
      // 执行上一次遗留的清理操作 就是 useEffect 中返回的清理函数
      hooks._pendingEffects.forEach(invokeCleanup);
      // 执行 effect
      hooks._pendingEffects.forEach(invokeEffect);
      hooks._pendingEffects = [];
    }
  }
  previousComponent = currentComponent;
};


// diffed 钩子
// diff 函数执行最后，在dom 更改之后会立即调用diffed 钩子
options.diffed = vnode => {
  if (oldAfterDiff) oldAfterDiff(vnode);

  const c = vnode._component;
    // 下面会提到useEffect就是进入_pendingEffects队列
  if (c && c.__hooks) {
    if (c.__hooks._pendingEffects.length) afterPaint(afterPaintEffects.push(c));
    // afterPaint 表示本次帧绘制完，下一帧开始前执行
    // 将含有_pendingEffects的组件推进全局的afterPaintEffects队列中s
    c.__hooks._list.forEach(hookItem => {
      if (hookItem._pendingArgs) {
        hookItem._args = hookItem._pendingArgs;
```

```
52        }
53        //  不是初始化默认值的话。就把 _value 设置成正确的值
54        // 这时组件 render() 渲染函数已经在执行过一遍了，hooks【useMemo】已经有了 _pendingValue
55        // 赋值操作 便于下一次使用
56        if (hookItem._pendingValue !== EMPTY) {
57          hookItem._value = hookItem._pendingValue;
58        }
59        hookItem._pendingArgs = undefined;
60        hookItem._pendingValue = EMPTY;
61      });
62    }
63    // 清理 组件 标志位
64    previousComponent = currentComponent = null;
65  };
66
67  options._commit = (vnode, commitQueue) => {
68    commitQueue.some(component => {
69      try {
70        // 执行上次的_renderCallbacks的清理函数
71        component._renderCallbacks.forEach(invokeCleanup);
72        // _renderCallbacks有可能是setState的第二个参数这种的、或者生命周期、或者forceUpdate的回调
73        // 通过_value判断是hook的回调则在此出执行
74        // 其他的 就放到外面执行
75        component._renderCallbacks = component._renderCallbacks.filter(cb =>
76          cb._value ? invokeEffect(cb) : true
77        );
78      } catch (e) {
79        commitQueue.some(c => {
80          if (c._renderCallbacks) c._renderCallbacks = [];
81        });
82        commitQueue = [];
83        options._catchError(e, component._vnode);
84      }
85    });
86
87    if (oldCommit) oldCommit(vnode, commitQueue);
88  };
89
90  options.unmount = vnode => {
91    if (oldBeforeUnmount) oldBeforeUnmount(vnode);
92
93    const c = vnode._component;
94    if (c && c.__hooks) {
95      let hasErrored;
96      // _cleanup 是 effect 类hook的清理函数，也就是我们每个effect的callback 的返回值函数
97      c.__hooks._list.forEach(s => {
98        try {
99        // 执行 useEffect 返回的清理函数
100         invokeCleanup(s);
101       } catch (e) {
102         hasErrored = e;
103       }
```

```
    });
    if (hasErrored) options._catchError(hasErrored, c._vnode);
  }
};_value
```

3. 辅助函数

```
afterPaint
// preact 的diff 是同步的 ，是宏任务
// newQueueLength ===1 是为了保证afterPaint 内的 afterNextFrame(flushAfterPaintEffects)
// 只执行一遍
//flushAfterPaintEffects 是在宏任务中执行的
// 一次将所有含有pendingEffect的组件进行回调进行
function afterPaint(newQueueLength) {
  if (newQueueLength === 1 || prevRaf !== options.requestAnimationFrame) {
    prevRaf = options.requestAnimationFrame;
    // 执行下一帧结束后，清空 useEffect的回调
    (prevRaf || afterNextFrame)(flushAfterPaintEffects);
  }
}

let HAS_RAF = typeof requestAnimationFrame == 'function';

function afterNextFrame(callback) {
  const done = () => {
    clearTimeout(timeout);
    if (HAS_RAF) cancelAnimationFrame(raf);
    //是在宏任务中执行的
    setTimeout(callback);
  };
  // 如果在100ms内 当前帧 requestAnimationFrame 没有结束（例如窗口不可见的情况下）
  // 则直接执行flushAfterPaintEffects
  const timeout = setTimeout(done, RAF_TIMEOUT);

  let raf;
  if (HAS_RAF) {
    raf = requestAnimationFrame(done);
  }
}

function flushAfterPaintEffects() {
  let component;
  /**
 * 执行队列内所有组件的上一次的`_pendingEffects`的清理函数
  执行本次的`_pendingEffects`。
 */
  while ((component = afterPaintEffects.shift())) {
    if (!component._parentDom) continue;
    try {
      // 清理上一次的_pendingEffects
      component.__hooks._pendingEffects.forEach(invokeCleanup);
      // 执行当前_pendingEffects
      component.__hooks._pendingEffects.forEach(invokeEffect);
      component.__hooks._pendingEffects = [];
    } catch (e) {
      component.__hooks._pendingEffects = [];// 因为下一次render 执行过程中 会重新往里面添加
      options._catchError(e, component._vnode);
    }
```

```
  }
}


function invokeCleanup(hook) {
  const comp = currentComponent; // __cleanup的执行可能会影响到currentComponent
  let cleanup = hook._cleanup;
  // 执行清理函数
  if (typeof cleanup == 'function') {
    hook._cleanup = undefined;
    cleanup();
  }
  currentComponent = comp;
}


function invokeEffect(hook) {
  const comp = currentComponent;// _value的执行可能会影响到currentComponent
  hook._cleanup = hook._value(); // 举例：就是useEffect中传入的callback 函数。返回值就是清理函数
  currentComponent = comp;
}
```

4. useXXX 源码实现

```
// 这个函数是在组件每次执行useXxx的时候，
// 首先执行这一步获取 hook 的状态的（以useEffect为例子）。
// 所有的hook都是使用这个函数先获取自身 hook 状态
function getHookState(index, type) {
  if (options._hook) {
    // 附加一个在查询钩子状态之前调用的钩子。
    options._hook(currentComponent, index, currentHook || type);
  }
  currentHook = 0; // 我理解这个变量的是存在为了 弥补 type 不传的情况 去区分

 // hook最终是挂在组件的__hooks属性上的，因此，每次渲染的时候只要去读取函数组件本身的属性就能获取上
  const hooks =
    currentComponent.__hooks ||
    (currentComponent.__hooks = {
      // 每个组件的hook存储
      _list: [],
      // useEffect 等
      _pendingEffects: []
    });

  // 初始化的时候，创建一个空的hook
  if (index >= hooks._list.length) {
    hooks._list.push({ _pendingValue: EMPTY });
  }
  return hooks._list[index];
}

export function useState(initialState) {
  currentHook = 1;
  return useReducer(invokeOrReturn, initialState);
}

export function useReducer(reducer, initialState, init) {
  const hookState = getHookState(currentIndex++, 2);
  hookState._reducer = reducer;
  if (!hookState._component) {
    hookState._value = [
       // invokeOrReturn 在第5项 工具函数中有介绍
      !init ? invokeOrReturn(undefined, initialState) : init(initialState),
      action => {
        // reducer函数计算出下次的state的值
        const nextValue = hookState._reducer(hookState._value[0], action);
        if (hookState._value[0] !== nextValue) {
          hookState._value = [nextValue, hookState._value[1]];
          // setState开始进行下一轮更新
          // 调用组件的setState方法进行组件的diff和相应更新操作
        // (这里是preact和react不太一样的一个地方，preact 的函数组件在内部和 class 组件一样使用
          hookState._component.setState({});
        }
      }
    ];
```

```
52
53      hookState._component = currentComponent;
54    }
55  // 返回当前的state
56    return hookState._value;
57  }
58
59  // useEffect 的 callback 执行是在本次渲染结束之后，下次渲染之前执行。
60  export function useEffect(callback, args) {
61    const state = getHookState(currentIndex++, 3);
62    // 判断 依赖项数组的值 是否改变
63    if (!options._skipEffects && argsChanged(state._args, args)) {
64      state._value = callback;
65      state._pendingArgs = args; //
66      // _pendingEffects则是本次重绘之后，下次重绘之前执行
67      currentComponent.__hooks._pendingEffects.push(state);
68    }
69  }
70
71  // useLayoutEffect则是在本次会在浏览器 layout 之后，painting 之前执行，是同步的。
72  export function useLayoutEffect(callback, args) {
73    const state = getHookState(currentIndex++, 4);
74    if (!options._skipEffects && argsChanged(state._args, args)) {
75      state._value = callback;
76      state._pendingArgs = args;
77      //_renderCallbacks 是在_commit 钩子中执行
78      // renderCallback 就是render后的回调
79      currentComponent._renderCallbacks.push(state);
80    }
81  }
82
83
84  export function useMemo(factory, args) {
85    const state = getHookState(currentIndex++, 7);
86     // 判断依赖项是否改变， 只是普通的===比较，
87     // 如果依赖的引用类型并且改变引用类型的上的属性 将不会执行callback
88    if (argsChanged(state._args, args)) {
89      // 改变后执行callback的函数返回值
90      state._pendingValue = factory();
91      //存储本次依赖的数据值
92      state._pendingArgs = args;
93      state._factory = factory;
94      return state._pendingValue;
95    }
96
97    return state._value;
98  }
99
100
101 export function useCallback(callback, args) {
102   currentHook = 8;
103   // 直接返回这个callback 而不是执行
```

```
  return useMemo(() => callback, args);
}
```

5. 工具函数

```
function argsChanged(oldArgs, newArgs) {
  return (
    !oldArgs ||
    oldArgs.length !== newArgs.length ||
    newArgs.some((arg, index) => arg !== oldArgs[index])
  );
}

function invokeOrReturn(arg, f) {
  return typeof f == 'function' ? f(arg) : f;
}

```