2 框架设计的核心要素

```
createApp(App).mount("#not-exist");
```

开发环境和生产环境

vue 在输出资源时， 会输出两个版本：

- vue.global.js(开发环境)
- vue.global.prod.js(生产环境)

处于开发环境时，会把_DEV_常量设置为 true.

框架良好的 tree-shaking

```
01 ├── demo
02 │   └── package.json
03 │   └── input.js
04 │   └── utils.js
```

实现 tree-shaking 条件 模块必须时 ES module

良好的 typescript 支持

```
function foo(val: any) {
  return val;
}
```

3 设计思路

3.1 声明式描述 UI

- DOM 元素
- 属性
- 事件
- 元素层级结构

```
const title = {
  //标签名称
  tag: "h1",
  props: {
    onclick: handler,
  },
  //子节点
  children: [
    {
      tag: "span",
    },
  ],
};
```

对应 vue.js 模板：

```
<h1 @click="handler">
  <span></span>
</h1>
```

使用 javascript 描述对象 UI 更加灵活：

```
let level = 3;
const title = {
  tag: `h${level}`,
};
```

渲染器

```
const vnode = {
  tag: "div",
  props: {
    onclick: () => alert("hello"),
  },
  children: "click me",
};
```

将上述虚拟 DOM 渲染成真实 DOM:

```
const vnode = {
    tag:'div',
    props:{
        onclick : () => alert('hello')
    },
    children:'click me'
}

function renderer(vnode , container){
    //使用vnode.tag 作为标签名称创建DOM元素
    const el = document.createElement(vnode.tag)

    //遍历vnode.props,将属性/事件添加到DOM元素中
    for(const key in vnode.props){
        if(/^on/.test(key)){
            //如果key 以on 开头 , 说明它是事件
            el.addEventListener(
                key.substr(2).toLowerCase(),
                //事件名称 onclick --->

                vnode.props[key]
                //事件处理函数
            )
        }
    }

    //处理children
    if(typeof vnode.children === 'string'){
        //如果children是字符串 , 说明它是元素的文本子节点
        el.appendChild(document.createTextNode(vnode.children))}
```

```
    else if(Array.isArray(vnode.children)){
        //递归调用renderer 渲染子节点，  使用当前元素el 作为挂载点
    vnode.children.forEach(child => renderer(child , el))
    }

    //将元素挂载到挂载点下
    container.appendChild(el)
}

//vnode:虚拟DOM对象
//container:一个真实DOM元素作为挂载点

//调用renderer函数
renderer(vnode , document.body)
//body作为挂载点
```

3.3 组件的本质

组件就是一组 **DOM** 元素的封装

如果 vnode.tag 类型是字符串，则他描述的是普通标签元素，此时调用 mountElement 完成渲染；若类型是函数，则描述的是组件，此时调用 mountComponent 函数完成渲染：

```
function renderer(vnode, container) {
  if (typeof vnode.tag === "string") {
    mountElement(vnode, container);
  }
  if (typeof vnode.tag === "function") {
    mountComponent(vnode, container);
  }
}
```

响应系统的作用和实现

副作用函数

```
function effect() {
  document.body.innerText = "hello vue3";
}
```

当 effect 函数执行时会修改 body 文本内容，但除了 effect 之外任何函数都可读取或设置 body 的文本内容。 即 **effect 函数执行会直接或间接影响其他函数的执行。**

4.2 响应式数据的基本实现

以下为例：

```
const obj = { text: "hello world" };
function effect() {
  document.body.innerText = "obj.text";
}
```

一个响应式系统的工作流程：

- 当读取操作放生时， 将副作用函数收集到桶中；
- 当设置操作发生时从桶中取出副作用函数并执行。

如何让 **obj** 变为响应式数据？

- 当副作用函数 effect 执行时， 会触发字段 obj.text读取操作；
- 当修改 obj.text 值，会触发 obj.text设置操作

关键：拦截一个对象的读取和设置操作

采用 **proxy**实现：

```
//存储副作用函数的桶、
const bucket = new Set();

//原始数据
const data = {
  text: "hello world",
};

//对原始数据的代理
const obj = new Proxy(data, {
  //拦截读取操作
  get(target, key) {
    //将副作用函数effect 添加到存储副作用函数的桶中
    bucket.add(effect);
    //返回属性值
    return target[key];
  },
  //拦截设置操作
  set(target, key, newVal) {
    target[key] = newVal;
```

```
    //把副作用函数从桶里取出并执行
    bucket.forEach((fn) => fn());
    //返回true代表设置操作成功
    return true;
  },
});

function effect() {
  document.body.innerText = obj.text;
}
//执行副作用函数,触发读取
effect();
//1秒后修改响应式数据
setTimeout(() => {
  obj.text = "hello vue3";
}, 1000);
```