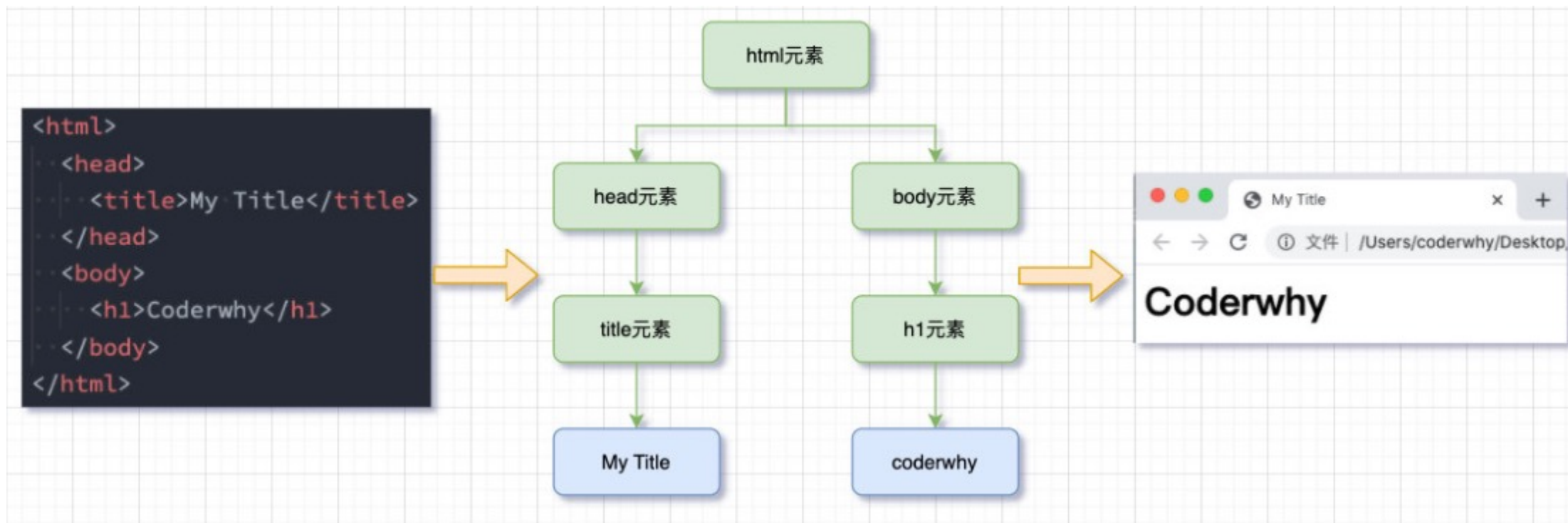


Vue3源码学习

王红元 coderwhy

真实的DOM渲染

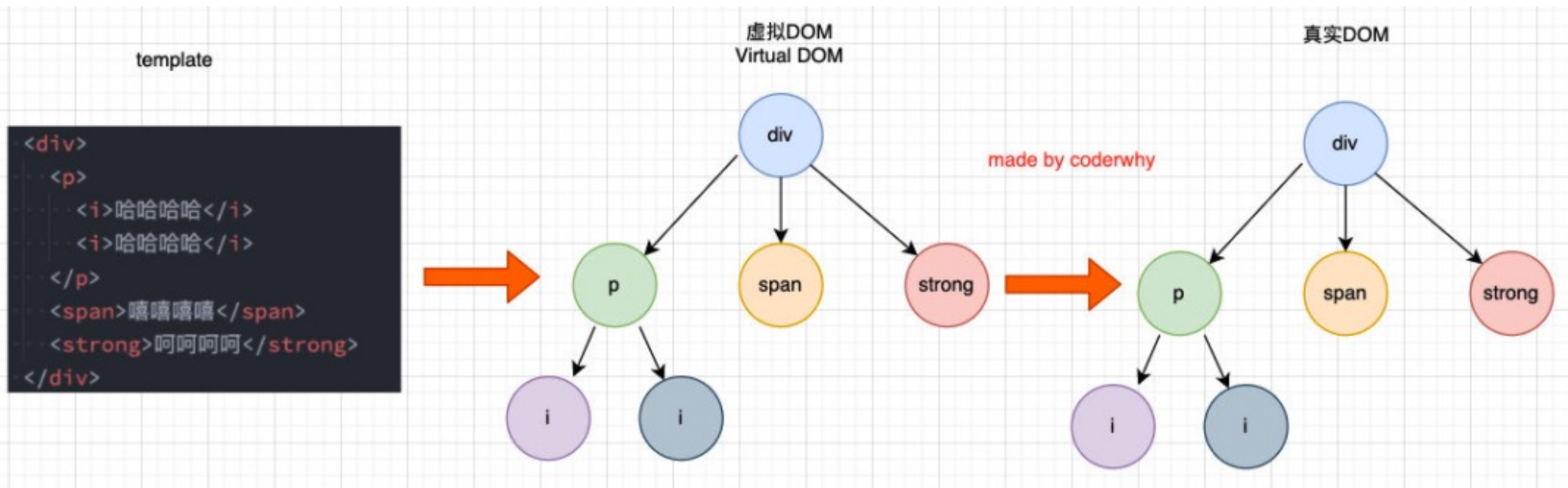
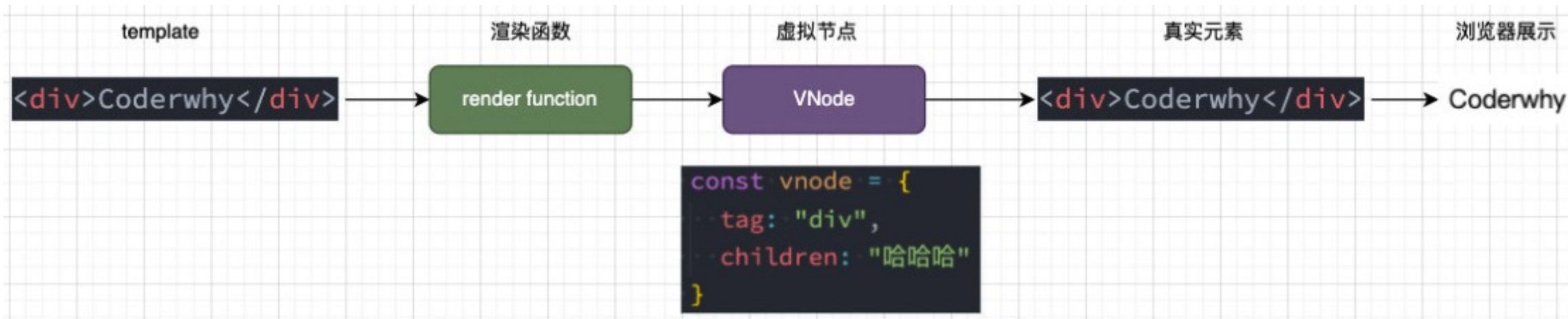
- 我们传统的前端开发中，我们是编写自己的HTML，最终被渲染到浏览器上的，那么它是什么样的过程呢？



虚拟DOM的优势

- 目前框架都会引入虚拟DOM来对真实的DOM进行抽象，这样做有很多的好处：
- 首先是可以对真实的元素节点进行抽象，抽象成VNode（虚拟节点），这样方便后续对其进行各种操作：
 - 因为对于直接操作DOM来说是有很多的限制的，比如diff、clone等等，但是使用JavaScript编程语言来操作这些，就变得非常的简单；
 - 我们可以使用JavaScript来表达非常多的逻辑，而对于DOM本身来说是非常不方便的；
- 其次是方便实现跨平台，包括你可以将VNode节点渲染成任意你想要的节点
 - 如渲染在canvas、WebGL、SSR、Native（iOS、Android）上；
 - 并且Vue允许你开发属于自己的渲染器（renderer），在其他平台上渲染；

虚拟DOM的渲染过程



三大核心系统

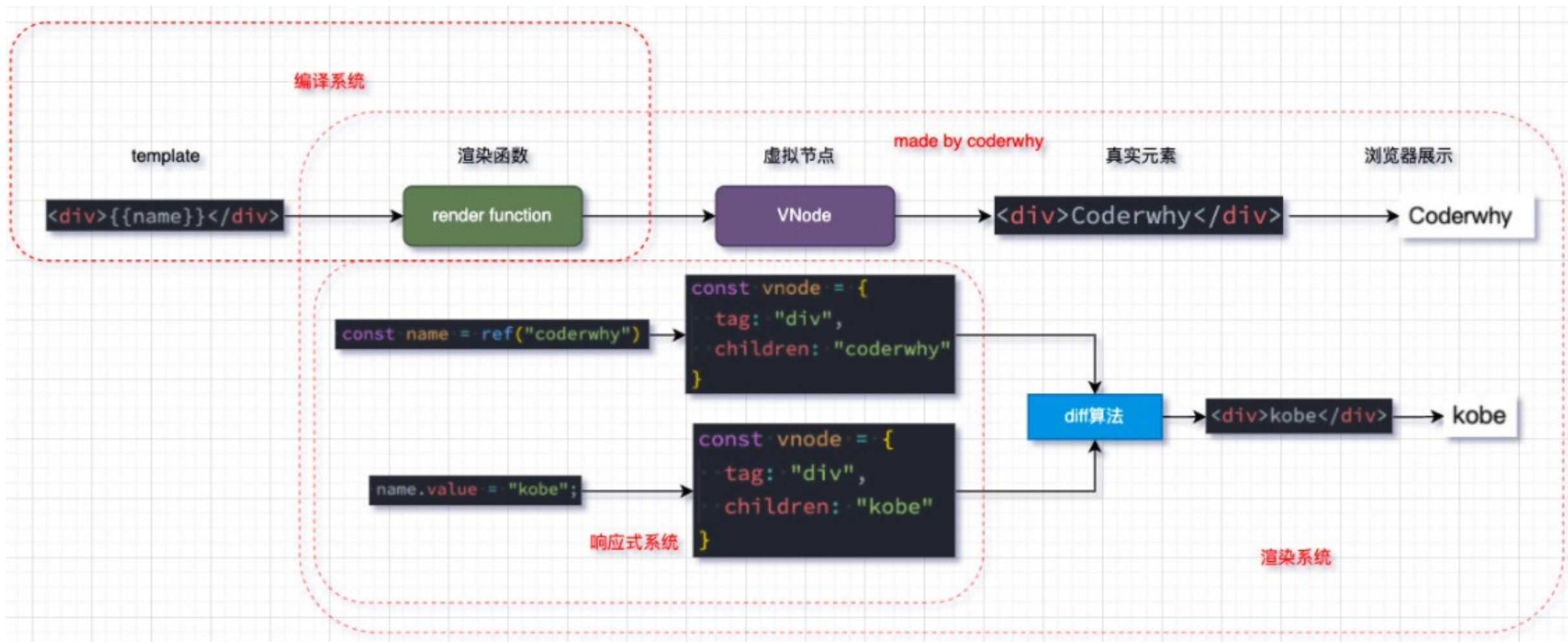
■ 事实上Vue的源码包含三大核心：

- Compiler模块：编译模板系统；
- Runtime模块：也可以称之为Renderer模块，真正渲染的模块；
- Reactivity模块：响应式系统；



三大系统协同工作

■ 三个系统之间如何协同工作呢：





实现Mini-Vue

■ 这里我们实现一个简洁版的Mini-Vue框架，该Vue包括三个模块：

- 渲染系统模块；
- 可响应式系统模块；
- 应用程序入口模块；

■ **渲染系统**，该模块主要包含三个功能：

- 功能一：h函数，用于返回一个VNode对象；
- 功能二：mount函数，用于将VNode挂载到DOM上；
- 功能三：patch函数，用于对两个VNode进行对比，决定如何处理新的VNode；

h函数 – 生成VNode

■ h函数的实现：

□ 直接返回一个VNode对象即可

```
const h = (tag, props, children) => {  
  return {  
    tag,  
    props,  
    children  
  }  
}
```

Mount函数 – 挂载VNode

■ mount函数的实现：

■ 第一步：根据tag，创建HTML元素，并且存储到vnode的el中；

■ 第二步：处理props属性

- 如果以on开头，那么监听事件；
- 普通属性直接通过 setAttribute 添加即可；

■ 第三步：处理子节点

- 如果是字符串节点，那么直接设置textContent；
- 如果是数组节点，那么遍历调用 mount 函数；

```
const mount = (vnode, container) => {  
  const el = vnode.el = document.createElement(vnode.tag);  
  if (vnode.props) {  
    for (const key in vnode.props) {  
      const value = vnode.props[key];  
      if (key.startsWith('on')) {  
        el.addEventListener(key.slice(2).toLowerCase(), value);  
      } else {  
        el.setAttribute(key, value);  
      }  
    }  
  }  
  if (vnode.children) {  
    if (typeof vnode.children === 'string') {  
      el.textContent = vnode.children;  
    } else {  
      vnode.children.forEach(child => {  
        mount(child, el);  
      })  
    }  
  }  
  container.appendChild(el);  
}
```

Patch函数 – 对比两个VNode

- patch函数的实现，分为两种情况
- n1和n2是不同类型的节点：
 - 找到n1的el父节点，删除原来的n1节点的el；
 - 挂载n2节点到n1的el父节点上；
- n1和n2节点是相同的节点：
 - 处理props的情况
 - ✓ 先将新节点的props全部挂载到el上；
 - ✓ 判断旧节点的props是否不需要在新节点上，如果不需要，那么删除对应的属性；
 - 处理children的情况
 - ✓ 如果新节点是一个字符串类型，那么直接调用 `el.textContent = newChildren`；
 - ✓ 如果新节点不同一个字符串类型：
 - 旧节点是一个字符串类型
 - 将el的textContent设置为空字符串；
 - 就节点是一个字符串类型，那么直接遍历新节点，挂载到el上；
 - 旧节点也是一个数组类型
 - 取出数组的最小长度；
 - 遍历所有的节点，新节点和旧节点进行patch操作；
 - 如果新节点的length更长，那么剩余的新节点进行挂载操作；
 - 如果旧节点的length更长，那么剩余的旧节点进行卸载操作；- 代码实现见下页

Patch的实现

```
const patch = (n1, n2) => {  
  if (n1.tag === n2.tag) {  
    const el = n2.el = n1.el;  
  
    // props  
    const oldProps = n1.props || {};  
    const newProps = n2.props || {};  
  
    for (const key in newProps) {  
      const oldValue = oldProps[key];  
      const newValue = newProps[key];  
      if (newValue !== oldValue) {  
        el.setAttribute(key, newValue);  
      }  
    }  
  
    for (const key in oldProps) {  
      if (!(key in newProps)) {  
        el.removeAttribute(key);  
      }  
    }  
  }  
}
```

```
// children  
const oldChildren = n1.children;  
const newChildren = n2.children;  
if (typeof newChildren === "string") {  
  el.textContent = newChildren;  
} else {  
  if (typeof oldChildren === "string") {  
    el.innerHTML = "";  
    newChildren.forEach(child => {  
      mount(child, el);  
    })  
  } else {  
    const commonLength = Math.min(oldChildren.length, newChildren.length);  
    for (let i = 0; i < commonLength; i++) {  
      patch(oldChildren[i], newChildren[i]);  
    }  
    if (newChildren.length > oldChildren.length) { ...  
    }  
    if (newChildren.length < oldChildren.length) { ...  
    }  
  }  
} else {  
  const n1Parent = n1.el.parentElement;  
  n1Parent.removeChild(n1.el);  
  mount(n2, n1Parent);  
}
```

依赖收集系统

```
class Dep {  
  constructor() {  
    this.subscribers = new Set();  
  }  
  
  depend() {  
    if (activeEffect) {  
      this.subscribers.add(activeEffect);  
    }  
  }  
  
  notify() {  
    this.subscribers.forEach(effect => {  
      effect();  
    })  
  }  
}
```

```
const dep = new Dep();  
let activeEffect = null;  
function watchEffect(effect) {  
  activeEffect = effect;  
  dep.depend();  
  effect();  
  activeEffect = null;  
}  
  
watchEffect(() => {  
  console.log("依赖回调1");  
})  
  
watchEffect(() => {  
  console.log("依赖回调2");  
})  
  
dep.notify();
```


响应式系统Vue2实现

```
function reactive(raw) {  
  Object.keys(raw).forEach(key => {  
    const dep = getDep(raw, key);  
    let value = raw[key];  
    Object.defineProperty(raw, key, {  
      get() {  
        dep.depend();  
        return value;  
      },  
      set(newValue) {  
        value = newValue;  
        dep.notify();  
      }  
    })  
  })  
  
  return raw;  
}
```

```
const targetMap = new WeakMap();  
  
function getDep(target, key) {  
  let depsMap = targetMap.get(target);  
  if (!depsMap) {  
    depsMap = new Map();  
    targetMap.set(target, depsMap);  
  }  
  
  let dep = depsMap.get(key);  
  if (!dep) {  
    dep = new Dep();  
    depsMap.set(key, dep);  
  }  
  
  return dep;  
}
```

响应式系统Vue3实现

```
function reactive(raw) {  
  return new Proxy(raw, {  
    get(target, key, receiver) {  
      const dep = getDep(target, key);  
      dep.depend();  
      return Reflect.get(target, key, receiver);  
    },  
    set(target, key, value, receiver) {  
      const dep = getDep(target, key);  
      const result = Reflect.set(target, key, value, receiver);  
      dep.notify();  
      return result;  
    }  
  })  
}
```




为什么Vue3选择Proxy呢？

- Object.defineProperty 是劫持对象的属性时，如果新增元素：
 - 那么Vue2需要再次 调用defineProperty，而 Proxy 劫持的是整个对象，不需要做特殊处理；
- 修改对象的不同：
 - 使用 defineProperty 时，我们修改原来的 obj 对象就可以触发拦截；
 - 而使用 proxy，就必须修改代理对象，即 Proxy 的实例才可以触发拦截；
- Proxy 能观察的类型比 defineProperty 更丰富
 - has：in操作符的捕获器；
 - deleteProperty：delete 操作符的捕捉器；
 - 等等其他操作；
- Proxy 作为新标准将受到浏览器厂商重点持续的性能优化；
- 缺点：Proxy 不兼容IE，也没有 polyfill, defineProperty 能支持到IE9

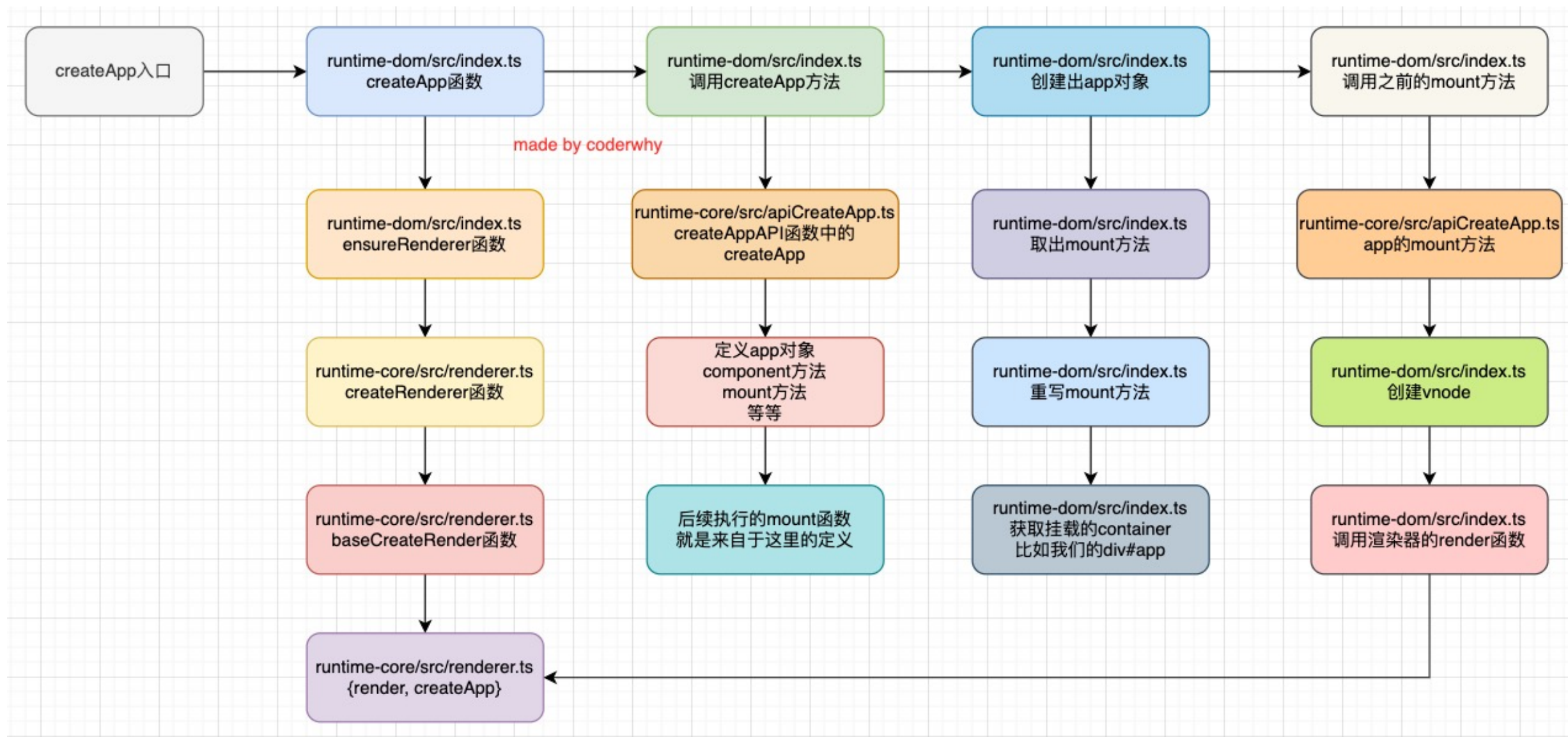
框架外层API设计

- 这样我们就知道了，从框架的层面来说，我们需要有两部分内容：

- createApp用于创建一个app对象；
- 该app对象有一个mount方法，可以将根组件挂载到某一个dom元素上；

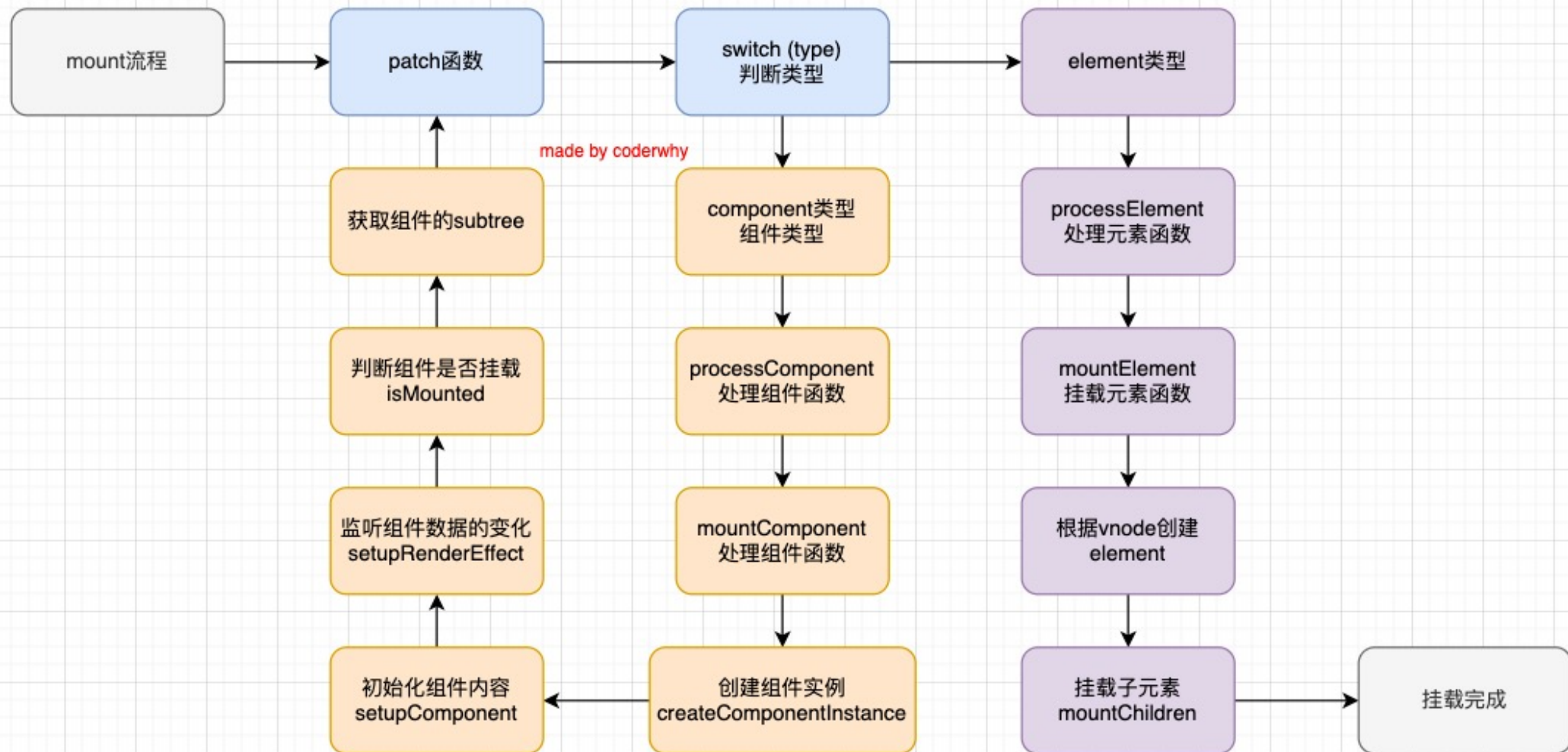
```
const createApp = (rootComponent) => {  
  return {  
    mount(selector) {  
      let isMounted = false;  
      let preVNode = null;  
  
      watchEffect(() => {  
        if (!isMounted) {  
          preVNode = rootComponent.render();  
          debugger;  
          mount(preVNode, document.querySelector(selector));  
          isMounted = true;  
        } else {  
          const newVNode = rootComponent.render();  
          patch(preVNode, newVNode);  
          preVNode = newVNode;  
        }  
      })  
    }  
  };  
}
```

源码阅读之createApp



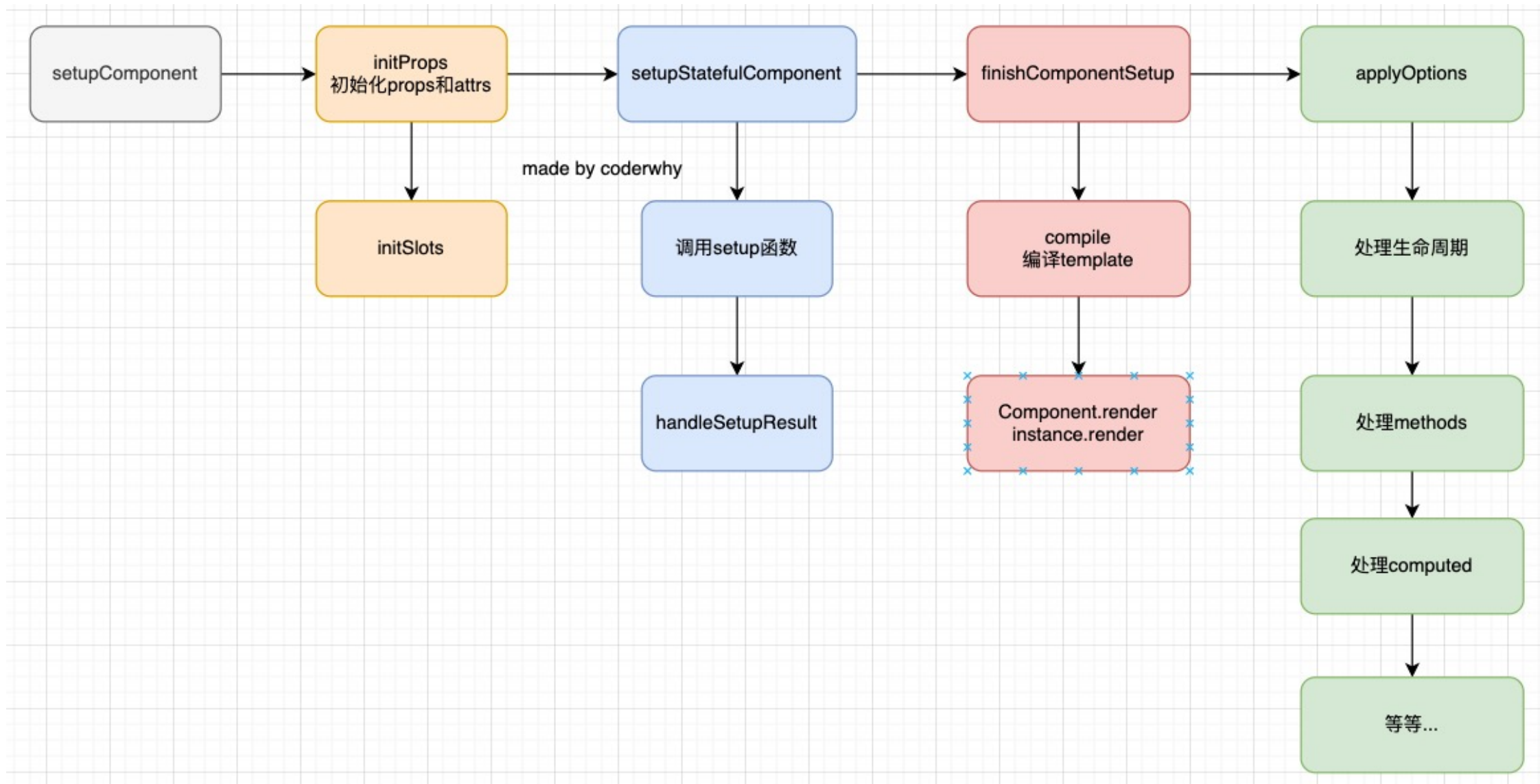
源码阅读之挂载根组件

以下操作均在渲染器renderer中完成



- `const app = {props: {message: String}}`
- `instance`
- `// 1.处理props和attrs`
- `instance.props`
- `instance.attrs`
- `// 2.处理slots`
- `instance.slots`
- `// 3.执行setup`
- `const result = setup()`
- `instance.setupState = proxyRefs(result);`
- `// 4.编译template -> compile`
- `<template> -> render函数`
- `instance.render = Component.render = render函数`
- `// 5.对vue2的options api进行知识`
- `data/methods/computed/生命周期`

组件化的初始化



Compile过程

```
// 对template进行解析, 生成ast
const ast = isString(template) ? baseParse(template, options) : template
const [nodeTransforms, directiveTransforms] = getBaseTransformPreset(
  prefixIdentifiers
)

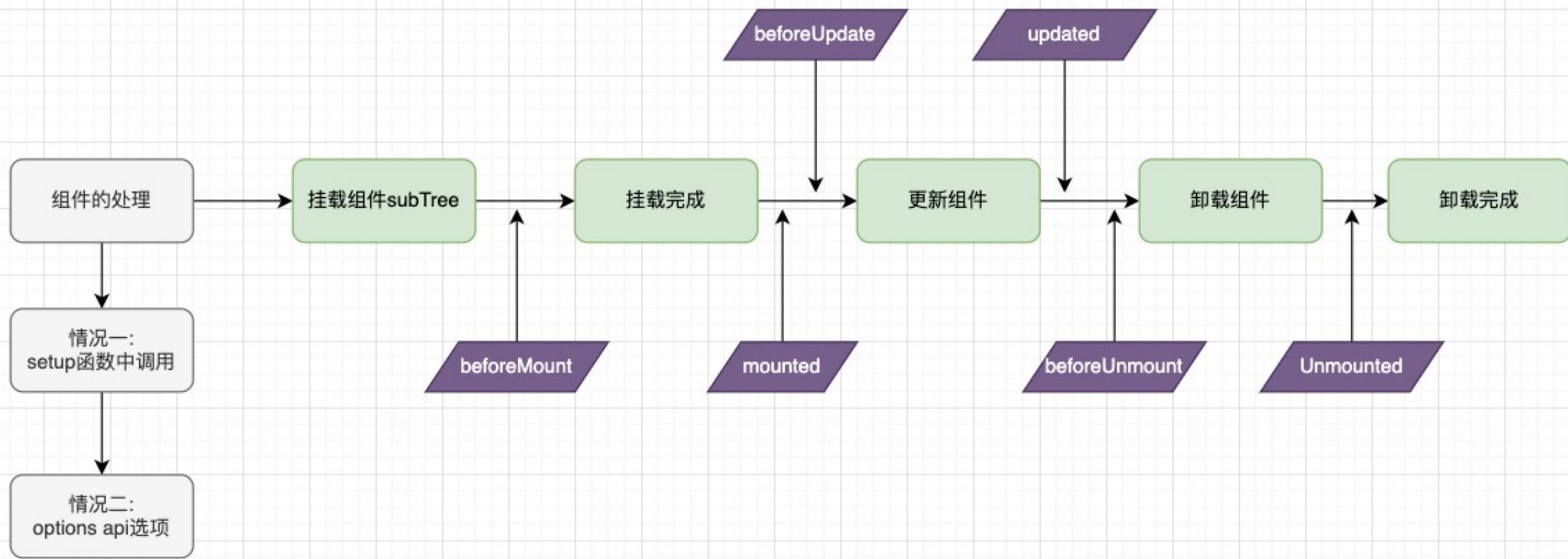
// AST转换
transform(...)

// 生成代码
return generate(
  ast,
  extend({}, options, {
    prefixIdentifiers
  })
)
```


Block Tree分析

```
1 (function anonymous(  
2 ) {  
3   const _Vue = Vue  
4   const { createVNode: _createVNode } = _Vue  
5  
6   const _hoisted_1 = /*#__PURE__*/_createVNode("div", null, "Coderwhy", -1 /* HOISTED */)   
7   const _hoisted_2 = /*#__PURE__*/_createVNode("h2", null, "Coderwhy", -1 /* HOISTED */)   
8  
9   return function render(_ctx, _cache) {  
0     with (_ctx) {  
1       const { createVNode: _createVNode, toDisplayString: _toDisplayString, Fragment: _Fragment, openBlock  
2  
3       return (_openBlock(), _createBlock(_Fragment, null, [  
4         _hoisted_1,  
5         _hoisted_2,  
6         _createVNode("p", null, _toDisplayString(message), 1 /* TEXT */),  
7         _createVNode("button", { onClick: changeMessage }, "修改message", 8 /* PROPS */, ["onClick"])  
8       ], 64 /* STABLE_FRAGMENT */))  
9     }  
0   }  
1 })
```

生命周期回调



template中数据的使用顺序

```
    } else if (setupState !== EMPTY_OBJ && hasOwn(setupState, key)) {  
      accessCache![key] = AccessTypes.SETUP  
      return setupState[key]  
    } else if (data !== EMPTY_OBJ && hasOwn(data, key)) {  
      accessCache![key] = AccessTypes.DATA  
      return data[key]  
    } else if (  
      (normalizedProps = instance.propsOptions[0]) &&  
      hasOwn(normalizedProps, key)  
    ) {  
      accessCache![key] = AccessTypes.PROPS  
      return props![key]  
    } else if (ctx !== EMPTY_OBJ && hasOwn(ctx, key)) {  
      accessCache![key] = AccessTypes.CONTEXT  
      return ctx[key]  
    } else if (!__FEATURE_OPTIONS_API__ || shouldCacheAccess) {  
      accessCache![key] = AccessTypes.OTHER  
    }  
  }  
}
```