第19章 手写Mini-Vue3框架

当我们在使用 Vue.js 3时，或许会好奇，Vue.js 3究竟是如何实现响应式数据的更新、模板编译以及组件化的呢？其实这些都离不开 Vue.js 3的核心代码。如果我们能够自己动手实现一个 Mini-Vue3 框架，不仅能够更深入地理解 Vue.js 3的内部原理，也能够提升我们的编程能力和理解能力。

本章节，我们将一步一步地手写一个 Mini-Vue3 框架，从而深入学习 Vue.js 3的内部实现原理，并从中获取编程经验和知识。在学习之前，先看看本章节源代码的管理方式，目录结构如下：

```
VueCode
├── ......
├── chapter19
|   └── learn_vuesource
```

19.1 Vue.js 3源码概述

在开始手写一个 Mini-Vue3 框架之前，先给大家普及一些与`Vue.js 3`源码相关的知识，比如：

- 什么是真实的`DOM`渲染?
- 什么是`VNode`（虚拟节点）？
- 什么是虚拟`DOM`（Virtual DOM，简称 VDOM）？
- 什么是`diff`算法?
- 虚拟`DOM`是如何转成真实`DOM`的?
- `Vue.js 3`源码包含的三个核心模块：`Compiler`模块、`Runtime`模块、`Reactivity`模块。

下面我们先来学习一下什么是虚拟`DOM`。

19.1.1 认识虚拟DOM

在学习虚拟DOM之前，我们先来看看真实的`DOM`是怎么渲染的。

1.真实的DOM渲染

在传统的前端开发中，我们编写的HTML最终会被渲染到浏览器上，这个过程可以称之为真实DOM渲染。下面我们可以通过一个例子来演示这个过程。

我们在`learn_vuesource`目录下新建`01_真实DOM渲染`文件夹，接着在该文件夹下新建`index.html`文件。然后在`index.html`文件中编写了一个简单的网页。代码如下所示：

```
<html>
  <head>
    <title>My Title</title>
  </head>
  <body>
    <h1>coderwhy</h1>
  </body>
</html>
```

上面的代码会被浏览器解析成一颗DOM树结构，再将DOM树渲染到浏览器上，如图19-1所示，这就是真实的DOM渲染过程。

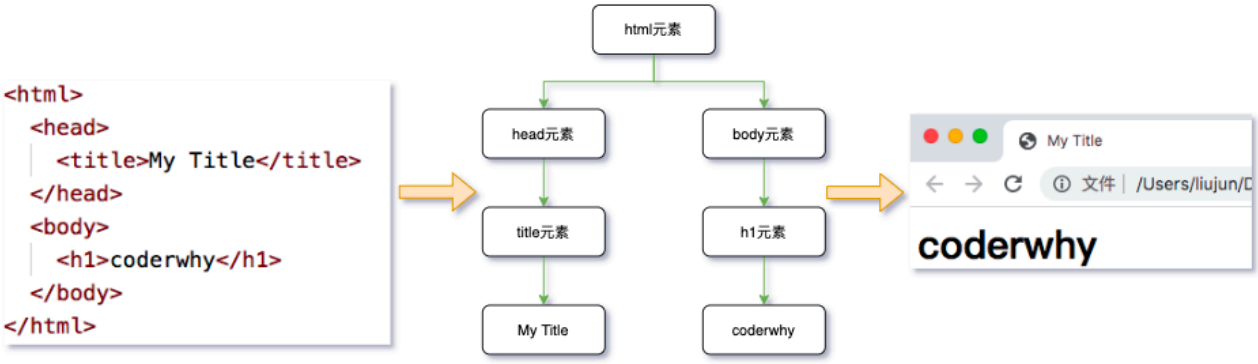

图19-1 真实DOM的渲染

2.虚拟DOM及优势

现代前端框架通常采用虚拟DOM来对真实DOM进行抽象，这种方式带来了许多好处。

首先，将真实的元素节点进行抽象，抽象成VNode（虚拟节点），可以方便后续对其进行各种操作。因为直接操作DOM存在很多的限制，比如diff、clone等。而使用 JavaScript 编程语言来操作 VNode 非常简单，也可以表达更多的逻辑。

其次，VNode可以方便地实现跨平台，将 VNode 节点渲染成任意想要的节点。比如可以渲染在Canvas、WebGL、SSR、Native（iOS、Android）上，如图19-2所示。例如，`<img>`元素在Web端渲染成`<img>`元素，在Android端渲染成`ImageView`控件，在iOS端渲染成`UIImageView`控件。

同时，Vue.js 3还允许开发者开发自己的渲染器（renderer），在其他平台上进行渲染。

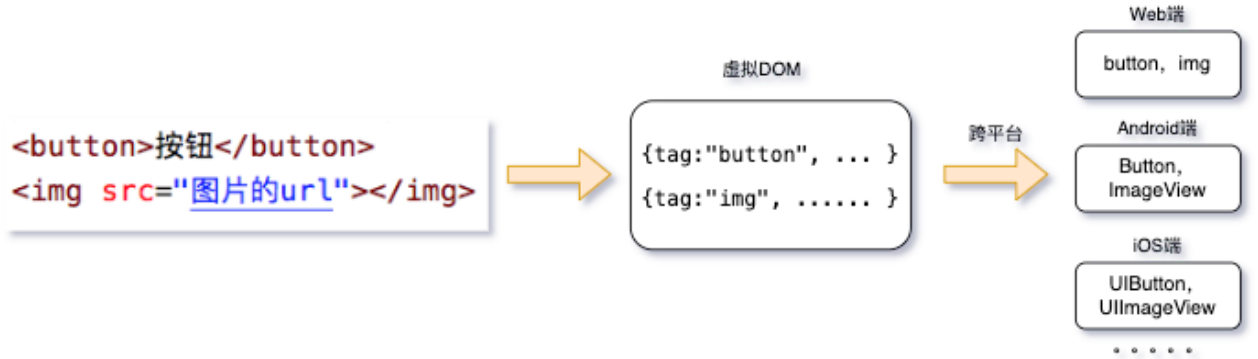

图19-2 虚拟DOM实现跨平台

3.虚拟DOM的渲染过程

虚拟DOM是对真实的DOM进行抽象，把真实的元素节点抽象成VNode，而多个VNode节点便组成了虚拟DOM 树。下面我们来看看虚拟DOM 的渲染过程，如图19-3所示。

1. 首先，编译器将 template 模板编译成 render 函数。
2. 接着，render函数通过调用h函数生成 VNode。
3. 然后，虚拟节点最终转换为真实的DOM元素，并在浏览器中运行代码进行渲染。

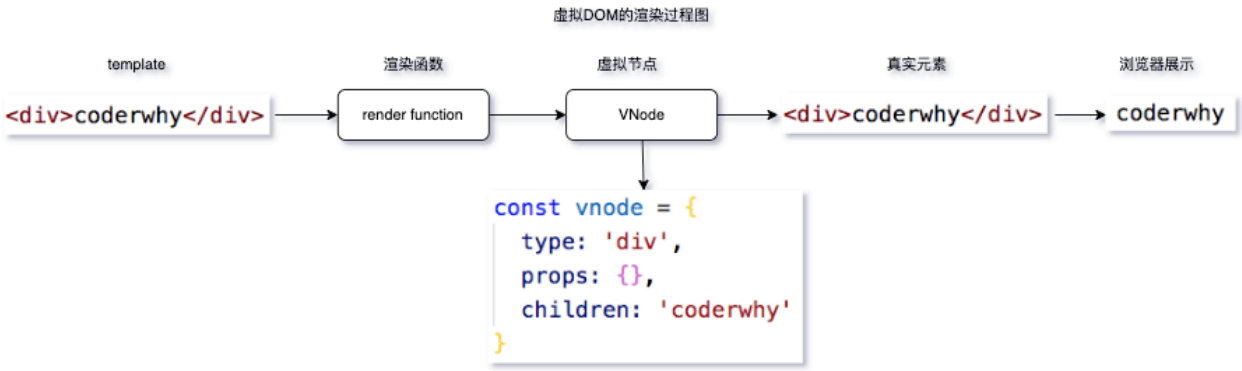

图19-3 虚拟DOM的渲染过程

上面仅仅渲染了一个div元素，如果是整个树结构的话，那就是下面的渲染过程。template编译后变成render函数，而render函数最终会返回虚拟DOM树，虚拟DOM经过mount函数会转成真实DOM并挂载到页面上，

上面仅仅渲染了一个`<div>`元素。如果是整个树结构的话，渲染过程如图19-4所示。`template`编译后会变成`render`函数，而`render`函数最终会返回虚拟DOM树。虚拟DOM经过`mount`函数会转换成真实的DOM并挂载到页面上。

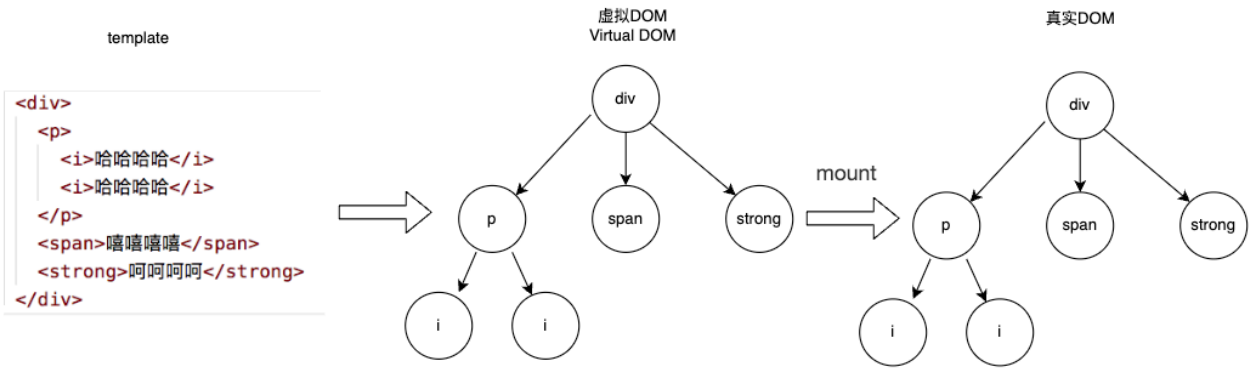

图19-4 虚拟DOM树结构的渲染过程

19.1.2 Vue.js 3三大核心模块

事实上，Vue.js 3的源码包含三大核心模块，如图19-5所示。

1. Compiler模块：编译模板系统。
2. Runtime模块：真正渲染的模块，即渲染系统。也可以称之为 Renderer 模块
3. Reactivity模块：响应式系统。

图19-5 Vue.js 3源码的三大核心

下面让我们来了解一下三个系统是如何协同工作的，如图19-6所示。

- 编译系统：将 template 模板编译成 render 函数。
- 渲染系统：生成 VNode，同时将 VNode 转换为真实的 DOM 并挂载到页面。如果某个 VNode 被修改了，便会调用 diff 算法来比较新旧 VNode，然后将改变的部分转换为真实的 DOM 并挂载到页面。
- 响应式系统：采用 Proxy 进行数据的劫持，并收集依赖。

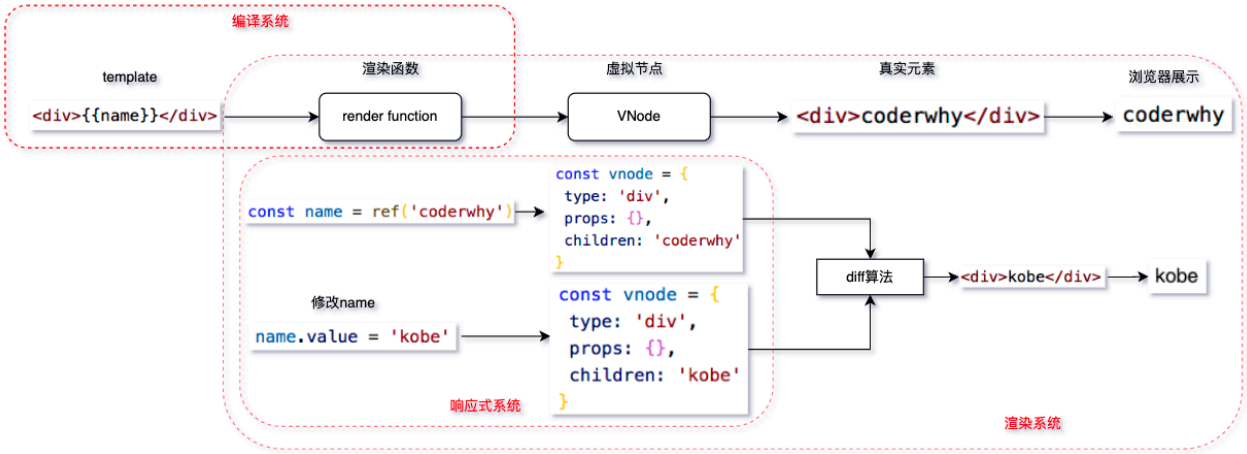

图19-6 三大核心之间协同工作

19.2 Mini-Vue3框架的实现

下面我们来实现一个精简版的 Mini-Vue3 框架，其中包含以下三个模块。

- 渲染系统（Runtime模块）：将 VNode 转换为真实的 DOM。
- 响应式系统（Reactive模块）：负责对数据进行劫持和依赖收集。
- 应用程序入口模块：创建app实例并挂载应用。

需要注意的是，该简洁版的Vue.js 3框架不包含编译系统（Compiler）模块。因为该模块需要编写大量的AST代码和正则表达式，实现起来较为复杂。在真实的开发中，单文件组件的模板编译是交给`@vue/compiler-sfc`插件来实现的。

19.2.1 渲染系统的实现

在 Vue.js 3 框架中，渲染系统主要包含以下三个功能。

1. `h`函数：用于返回一个`VNode`对象。
2. `mount`函数：用于将`VNode`挂载到`DOM`上。
3. `patch`函数：用于对两个`VNode`进行对比，决定如何处理新的`VNode`。

下面将详细介绍这三个功能的实现。

1.h函数的实现

`h`函数，用于返回一个VNode对象。下面我们来看看`h`函数的实现。

我们在`learn_vuesource`项目的`src`目录下新建`02_渲染器实现`文件夹，然后在该文件夹下分别新建`renderer.js`，`index.html`文件。

在`renderer.js`文件中编写h函数的实现，代码如下所示：

```
const h = (tag, props, children) => {
  // 返回 VNode 对象，即返回javascript对象
  return {
    tag,
    props,
    children
  }
}
```

上面代码编写了一个`h`函数，该函数直接返回一个`VNode`对象即可。

index.html文件，代码如下所示：

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
  <div id="app"></div>
  <script src="./renderer.js"></script>
  <script>
    // 1.通过h函数来创建一个 VNode 对象
    const vnode = h('div', {class: "why", id: "aaa"}, [
      h("h2", null, "当前计数: 100"),
      h("button", {onClick: function() {}}, "+1")
    ]);
    console.log(vnode) // 2.打印h函数返回的VNode(虚拟DOM)
  </script>
</body>
</html>
```

可以看到，我们在`index.html`文件中导入了`renderer.js`，即渲染系统模块。接着，调用该系统中的`h`函数来编写计数器案例页面。该函数最终会生成虚拟DOM并返回（注意：这里直接使用`h`函数来编写页面，并没有使用`template`模板）。

然后，我们在`index.html`文件上单击右键，选择“Open In Default Browser”。在浏览器中显示的效果如图`19-7`所示。这时，控制台可以看到`h`函数返回的`VNode`（也可称为虚拟DOM）。

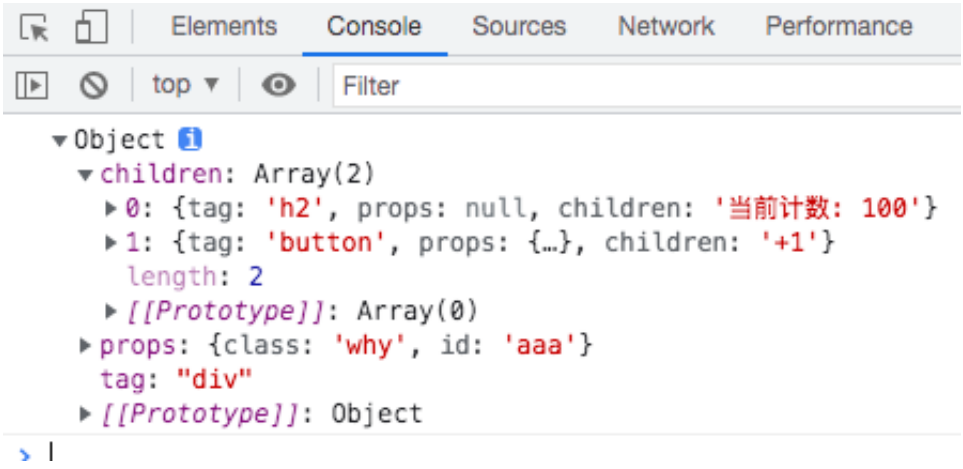

图19-7 h函数的实现

2.mount函数的实现

`mount`函数是用于将`VNode`转换成真实的`DOM`，并将其挂载到页面的`DOM`上的。

下面看看`mount`函数的实现，我们继续在`renderer.js`文件中添加一个`mount`函数，代码如下所示：

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

// 将 VNode 转成 Element，并挂载到页面上
const mount = (vnode, container) => {
  // 1.创建出真实的原生DOM对象，并且在 vnode 对象上保存 el 对象
  const el = vnode.el = document.createElement(vnode.tag);

  // 2.处理props
  if (vnode.props) {
    for (const key in vnode.props) {
      const value = vnode.props[key];

      if (key.startsWith("on")) { // 对事件监听的判断
        el.addEventListener(key.slice(2).toLowerCase(), value)
      } else {
        el.setAttribute(key, value);
      }
    }
  }

  // 3.处理children
```

```
  if (vnode.children) {
    if (typeof vnode.children === "string") {
      el.textContent = vnode.children;
    } else {
      vnode.children.forEach(item => {
        mount(item, el);
      })
    }
  }

  // 4.将el挂载到container元素上
  container.appendChild(el);
}
```

可以看到，这里的`mount`函数需要接收两个参数：`vnode`和`container`（需要挂载到目标的DOM对象）。该函数负责将`vnode`挂载到`container`上，主要分为以下四个步骤实现：

1. 根据参数`vnode`的`tag`属性，创建出真实的原生对象，并将该对象存在`vnode`的`el`属性中。
2. 处理`vnode`中的`props`属性，直接遍历`vnode`中的`props`。
    - 如果是`on`开头的属性，那么调用原生对象的`addEventListener`函数来添加该属性的事件监听。
    - 如果不是`on`开头的属性，那直接调用生对象的`setAttribute`方法来添加。
3. 处理`vnode`中的`children`属性。先判断`children`的类型是否是一个字符串。
    - 如果是字符串，就给原生对象的`textContent`属性重新赋值来更新。
    - 如果`children`是一个数组，遍历`children`数组，拿到数组中的每一个`item`（`vnode`对象）之后，递归调用`mount`函数来将数组中的每个子节点挂载到`el`原生对象上（注意：`el`是`children`的父节点）。
4. 将创建出来的`el`原生对象挂载到`container`上（例如：`<div id='app'></div>`元素上）。

接着，我们在 index.html 文件中使用 `mount` 函数，代码如下所示：

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
  <div id="app"></div>
  <script src="./renderer.js"></script>
  <script>
    // 1.通过h函数来创建一个VNode对象
    const vnode = h('div', {class: "why", id: "aaa"}, [
      h("h2", null, "当前计数: 100"),
      h("button", {onClick: function() {}}, "+1")
```

```
    ]); // 返回 VNode ( 也可称虚拟DOM )
    console.log(vnode)
    // 2.通过mount函数，将vnode挂载到 <div id="app"> 元素上
    mount(vnode, document.querySelector("#app"))
  </script>
</body>
</html>
```

可以看到，我们先调用`h`函数来创建`vnode`，接着调用`mount`函数来将`vnode`挂载到`<div id='app'>`元素上。

最后，在浏览器中显示的效果如图19-8所示。这时，控制台可以看到`h`函数返回的虚拟DOM。

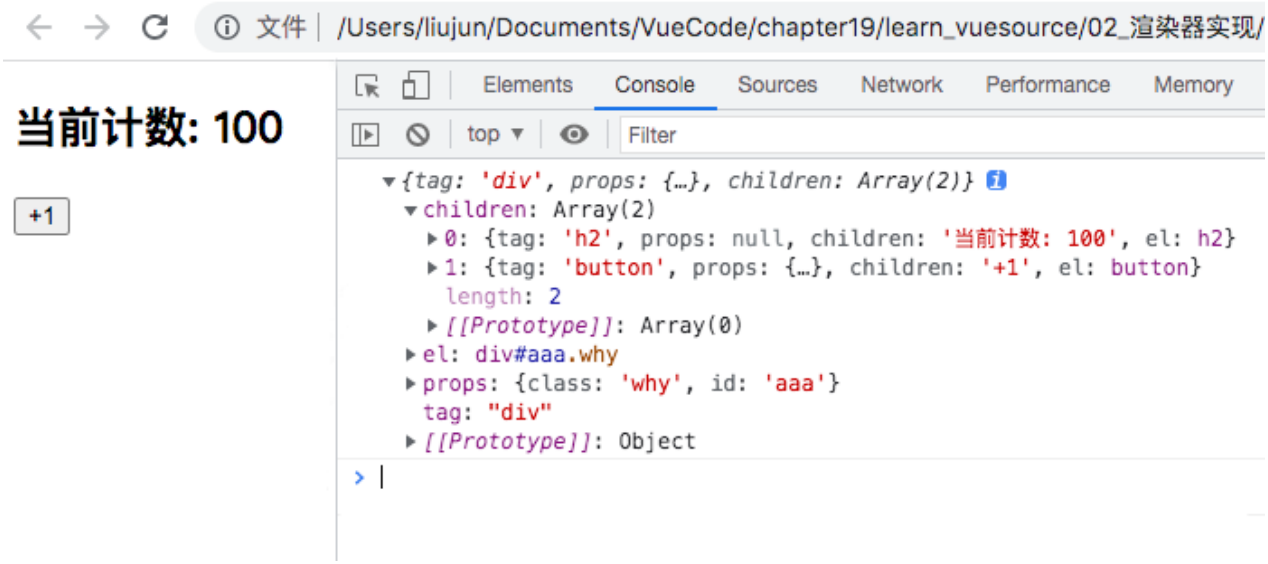

图19-7 mount函数的实现

3.patch函数的实现

`patch`函数是用于对两个VNode进行对比，决定如何处理新旧的VNode。其实，patch函数的实现就是diff算法的实现。下面我们来看看`patch`函数具体的实现。

我们继续在`renderer.js`文件中添加`patch`函数，代码如下所示：

```
const h = (tag, props, children) => {  ......  }
const mount = (vnode, container) => { ......  }

// patch函数用于对两个VNode进行对比。n1是旧节点，n2是新节点
const patch = (n1, n2) => {
  // 如果n1和n2的类型不一样，n2直接替换n1
  if (n1.tag !== n2.tag) {
    const n1ElParent = n1.el.parentElement;
    n1ElParent.removeChild(n1.el);
    mount(n2, n1ElParent);
  // 如果n1和n2的类型一样。先处理props，在处理children
  } else {
```

```
// 1.取出el对象，并且在n2中进行保存
const el = n2.el = n1.el;

// 2.处理props的情况
const oldProps = n1.props || {};
const newProps = n2.props || {};
// 2.1.获取所有的newProps添加到el
for (const key in newProps) {
  const oldValue = oldProps[key];
  const newValue = newProps[key];
  if (newValue !== oldValue) {
    if (key.startsWith("on")) { // 对事件监听的判断
      el.addEventListener(key.slice(2).toLowerCase(), newValue)
    } else {
      el.setAttribute(key, newValue);
    }
  }
}

// 2.2.删除旧的props
for (const key in oldProps) {
  if (key.startsWith("on")) { // 对事件监听的判断
    const value = oldProps[key];
    el.removeEventListener(key.slice(2).toLowerCase(), value)
  }
  if (!(key in newProps)) {
    el.removeAttribute(key);
  }
}

// 3.处理children的情况
const oldChildren = n1.children || [];
const newChidlren = n2.children || [];

if (typeof newChidlren === "string") { // 情况一: newChildren本身是一个string
  // 边界情况 (edge case)
  if (typeof oldChildren === "string") {
    if (newChidlren !== oldChildren) {
      el.textContent = newChidlren
    }
  } else {
    el.innerHTML = newChidlren;
  }
} else { // 情况二: newChildren 新节点是一个数组
  if (typeof oldChildren === "string") { // 旧节点是一个字符串类型
    el.innerHTML = "";
    // 遍历新节点，将每一个新节点挂载到el上
    newChidlren.forEach(item => {
      mount(item, el);
```

```
        })
      } else {
        // 如 oldChildren 为: [v1, v2, v3, v8, v9]
        // 如 newChildren 为: [v1, v5, v6]
        // 3.1.取出n1、n2中children数组的最小长度
        const commonLength = Math.min(oldChildren.length, newChidlren.length);
        for (let i = 0; i < commonLength; i++) {
          // 前面相同索引的VNode，进行patch操作。patch(v1,v1)，patch(v2,v5)，
patch(v3,v6)
          patch(oldChildren[i], newChidlren[i]);
        }

        // 如 oldChildren 为: [v1, v2, v3]
        // 如 newChildren 为: [v1, v5, v6, v8, v9]
        // 3.2.newChildren.length > oldChildren.length
        if (newChidlren.length > oldChildren.length) {
          // 新节点children多出的v8，v9，将mount挂载到el
          newChidlren.slice(oldChildren.length).forEach(item => {
            mount(item, el);
          })
        }

        // 如 oldChildren 为: [v1, v2, v3, v8, v9]
        // 如 newChildren 为: [v1, v5, v6]
        // 3.3.newChildren.length < oldChildren.length
        if (newChidlren.length < oldChildren.length) {
          // 旧节点children多出的v8，v9，将从el中移除
          oldChildren.slice(newChidlren.length).forEach(item => {
            el.removeChild(item.el);
          })
        }
      }
    }
  }
}
```

可以看到，`patch`函数需要接收`n1`和`n2`两个参数，其中`n1`是旧节点，`n2`是新节点。`patch`函数主要分为以下两种情况来实现。

第一种情况：如果n1和n2是不同类型的节点，如图19-9所示。

1. 找到 n1 的父节点，删除原来的 n1 节点。
2. 将 n2 节点挂载到 n1 的父节点上。

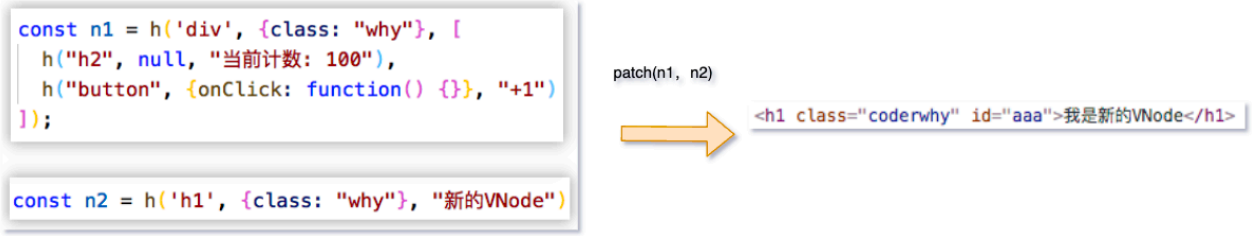

图19-9 patch处理不同类型的n1和n2

第二种情况：如果n1和n2节点是相同的节点，需要处理以下情况。

（1）处理props的情况，如图19-10所示。

- 首先，将新节点的props全部挂载到el上。
- 然后，判断旧节点的props是否需要在新节点上，如果不需要，那么删除对应的属性。

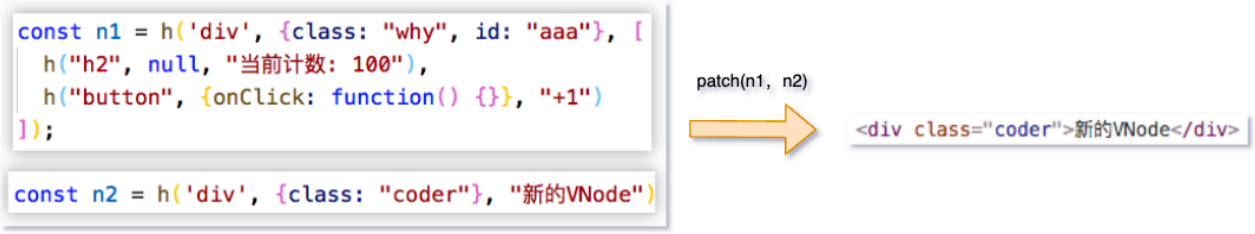

图19-10 patch处理同类型n1和n2的props

（2）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`，如上图19-10所示。
- 如果新节点是数组类型，需要判断旧节点的类型，例如。
  - 旧节点是一个字符串类型，如图19-11所示。
    - 首先，将`el`的`textContent`设置为空字符串。
    - 然后，直接遍历新节点，将每一个新节点挂载（`mount`）到`el`上。

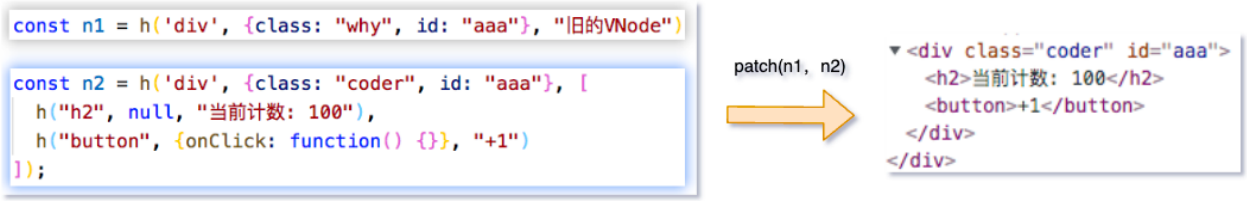

图19-11 n1的children为字符串

- 旧节点也是一个数组类型（不考虑key），如图19-12所示。
  - 取出n1、n2中children数组的最小长度。
  - 遍历数组在最小长度范围内的所有节点，对新节点和旧节点进行path操作。
  - 如果新节点的length更长，则对剩余的新节点进行挂载操作。
  - 如果旧节点的length更长，则对剩余的旧节点进行卸载操作。

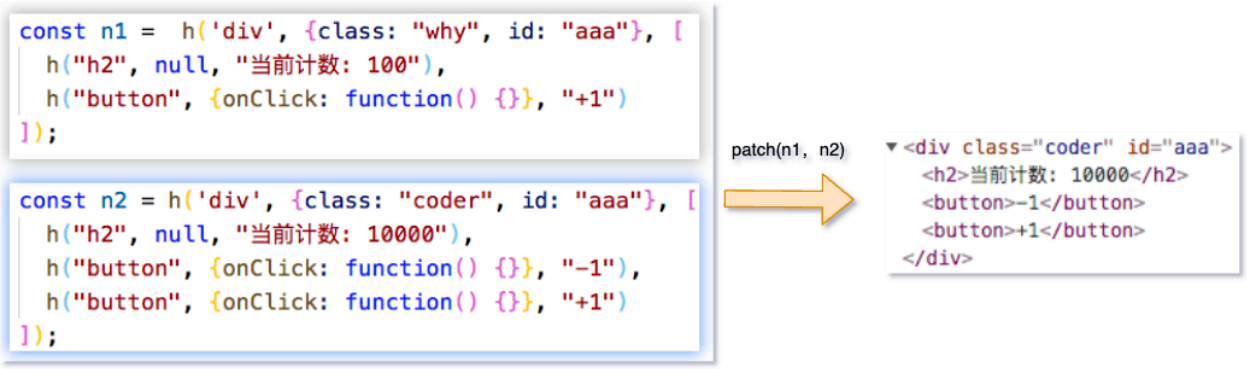

图19-12 n1和n2的children都为数组

实现完`patch`函数之后，接着我们在 index.html 文件中使用`patch`函数，代码如下所示：

```
<!DOCTYPE html>
<html lang="en">
......
<body>
  <div id="app"></div>
  <script src="./renderer.js"></script>
  <script>
    const n1 = h('div', {class: "why", id: "aaa"}, [
      h("h2", null, "当前计数: 100"),
      h("button", {onClick: function() {}}, "+1")
    ]);
    mount(n1, document.querySelector("#app"))

    // 模拟2秒后更新布局
    setTimeout(() => {
      // 1.创建新的 n2
      const n2 = h('div', {class: "coder", id: "aaa"}, [
        h("h2", null, "当前计数: 10000"),
        h("button", {onClick: function() {}}, "-1"),
        h("button", {onClick: function() {}}, "+1")
      ]);
      // 2.旧，新VNode对比，决定如何处理新的VNode
      patch(n1, n2);
    }, 2000)
  </script>
</body>
</html>
```

可以看到，我们先将 n1 节点挂载到`<div id="app">`的元素上，接着过了2秒我们创建了n2新节点，该节点将旧节点的`class`属性的值更新为`coder`，将`<h2>`元素的内容更新为当前计数：1000，将`<button>`的值更新为-1。同时，我们还新增加一个+1的`<button>`。

然后，我们调用`patch`函数（即`diff`算法）来对比`n1`和`n2`节点，从而决定如何处理旧节点的更新。

最后，我们将`index.html`在浏览器中运行代码的效果如图19-13所示。默认先显示`n1`节点的计数器，过了两秒之后显示更新后的计数器。

图19-11 patch函数的使用

19.2.2 响应式系统的实现

前面我们已经介绍了如何将VNode转换为真实的DOM节点，并将其挂载到DOM上。这一步仅仅实现了Vue.js 3的渲染系统。除此之外，Vue.js 3还有一个非常重要的模块：响应式系统。

下面将详细介绍Vue.js的响应式系统原理，包括响应式思想、依赖收集系统以及Vue.js 2和Vue.js 3的响应式系统。

1.响应式思想

在介绍Vue.js 3的响应式系统之前，我们先来了解一下响应式的思想。假设我们有一个info对象，并在doubleCounter函数中依赖于info的counter属性。

其中，响应式的思想就是：当info中的counter属性发生改变时，会自动触发doubleCounter函数的回调。因为该函数依赖于info中的counter属性。

如果要实现该功能，最简单的实现方式是在修改info中的counter属性时，手动调用一次doubleCounter函数来实现响应式。代码如下所示：

```
const info = {counter: 100};
function doubleCounter() {
  console.log(info.counter * 2);
}
doubleCounter();

// 1.修改info
info.counter++;
// 2.手动调用 doubleCounter 函数
doubleCounter();
```

2.依赖收集系统

前面只是响应式最简单的实现，Vue.js 3中的响应式系统要复杂得多。接下来，让我们来探究一下Vue.js 3是如何实现响应式系统的。

我们在`learn_vuesource`项目的`src`目录下新建`03_响应式系统`文件夹，然后在该文件夹下分别新建`index.html，reactive.js`文件。

我们在`reactive.js`文件中编写响应式系统，代码如下所示：

```
class Dep {
    constructor() {
      this.subscribers = new Set(); // 存放收集的依赖，即副作用函数
    }
    // 添加订阅
    addEffect(effect) {
      this.subscribers.add(effect);
    }
    // 发布通知
    notify() {
      // 遍历所有收集的依赖，并执行
      this.subscribers.forEach(effect => {
        effect();
      })
    }
}


const info = {counter: 100};
const dep = new Dep(); // Dep里面会新建一个Set集合存依赖

function doubleCounter() {
  console.log(info.counter * 2); // 该函数持有info中counter的依赖
}

function powerCounter() {
  console.log(info.counter * info.counter); // 该函数持有info中counter的依赖
}
// 手动进行依赖收集，即收集doubleCounter和powerCounter副作用函数
dep.addEffect(doubleCounter);
dep.addEffect(powerCounter);

// 修改counter
info.counter++;
// 手动通知更新
dep.notify();
```

可以看到，这里我们封装了一个可以手动收集依赖（函数）的系统：

1. 使用Dep类创建依赖收集的dep对象。
2. 调用dep的addEffect方法来进行依赖收集。
3. 当info中的counter属性被修改后，手动调用dep的notify方法来实现响应式（即通知所有依赖执行更新）。

最后，在index.html文件上右击，选择“Open In Default Browser”。在浏览器中显示的效果如图19-14所示。

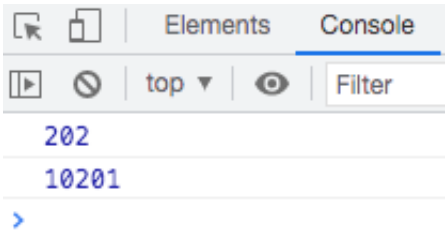

图19-14 手动收集依赖和更新

大家可以发现，前面写的响应式系统有一个明显的缺陷：需要手动收集依赖和手动触发更新。

为了解决这个问题，我们需要对代码进行优化，让程序可以自动进行依赖收集。接下来，我们将继续优化reactive.js中的响应式系统。代码如下所示：

```
class Dep {
    constructor() {
      this.subscribers = new Set(); // 存放收集的依赖
    }
    // 自动收集依赖
    depend() {
      if (activeEffect) { // 被收集的全局依赖，即副作用函数
        this.subscribers.add(activeEffect);
      }
    }
    // 发布通知
    notify() {
      this.subscribers.forEach(effect => {
        effect();
      })
    }
}

const dep = new Dep(); // Dep里面会新建一个Set集合存依赖
let activeEffect = null;
function watchEffect(effect) {
  activeEffect = effect; // 将需收集的依赖赋给全局activeEffect变量
  dep.depend(); // 自动依赖收集
  effect(); // 收集完依赖后执行一下该依赖函数
  activeEffect = null;
}

const info = {counter: 100, name:'why'};
console.log('==自动依赖收集==')
// 自动依赖收集，该回调函数也称副作用函数
watchEffect(function () {
  console.log(info.counter * 2);
})
watchEffect(function () {
```

```
  console.log(info.counter * info.counter);
})
watchEffect(function () {
  console.log(info.name);
})
console.log('======更新======')

// 修改counter
info.counter++;
// 通知更新
dep.notify();
```

可以看到，我们这次封装了一个可以自动收集依赖的系统：

- 首先，在Dep类中添加了一个depend方法，用于自动收集全局的副作用函数（也称为activeEffect）的依赖。
- 接着，我们编写了watchEffect函数，实现了自动收集依赖的功能。该函数需要接收一个副作用函数。
  - 在该函数中，我们首先将effect函数赋给全局的activeEffect变量。
  - 接着，调用dep的depend方法来实现对activeEffect依赖的自动收集。
  - 自动收集完成后，再执行一次effect函数（Vue.js 3中的watchEffect函数默认会先执行一次）
  - 最后，将activeEffect函数赋为null。
- 然后，我们调用watchEffect函数来实现自动收集依赖。
- 最后，当info中的counter修改时，我们手动调用dep的notify方法来通知依赖执行更新。

保存代码，在浏览器中运行`index.html`文件。控制台的输出如图19-15所示。

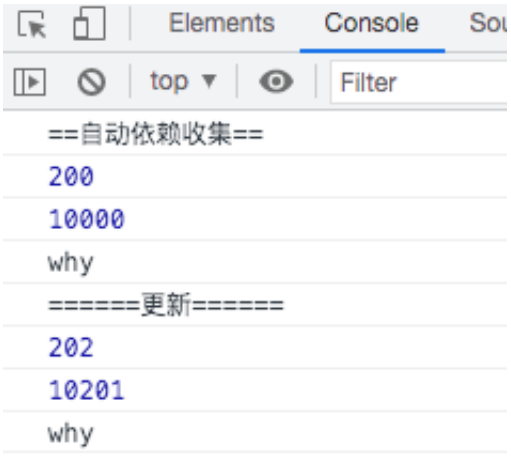

图19-15 自动收集依赖

3.Vue.js 2响应式系统的实现

大家可能会注意到，刚才我们更新了counter，但依赖于name的副作用函数也被执行了。

如果我希望在修改counter时，只执行与counter有关的副作用函数；在修改name时，只执行与name有关的副作用函数。这时，就需要进一步优化 reactive.js 中的响应式系统。代码如下所示：

```
class Dep {
   ......(和上面一样，这省略)
```

```
}

// const dep = new Dep(); // 已被getDep函数代替
let activeEffect = null;
function watchEffect(effect) {
  activeEffect = effect;
//   dep.depend(); // 自动依赖收集，已经移到get函数中实现
  effect();
  activeEffect = null;
}

// Map({key: value}): key是一个字符串
// WeakMap({key(对象): value}): key是一个对象，弱引用
// 1.创建一个WeakMap对象来存放所有的依赖
const targetMap = new WeakMap();
// 2.获取某一个属性对应的依赖Set集合
function getDep(target, key) {
  // 2.1根据对象(target)取出对应的Map对象
  let depsMap = targetMap.get(target);
  if (!depsMap) {
    depsMap = new Map();
    targetMap.set(target, depsMap);
  }

  // 2.2取出具体的dep对象
  let dep = depsMap.get(key);
  if (!dep) {
    dep = new Dep();
    depsMap.set(key, dep);
  }
  return dep;
}

// 3.Vue.js 2对原生（raw）数据进行劫持
function reactive(raw) {
  Object.keys(raw).forEach(key => {
    const dep = getDep(raw, key);
    let value = raw[key];
    Object.defineProperty(raw, key, {
      get() {
        dep.depend(); // 自动依赖收集
        return value;
      },
      set(newValue) {
        if (value !== newValue) {
          value = newValue;
          dep.notify(); // 通知更新
        }
      }
```

```
    })
  })
  return raw;
}

// 注意：这里以下的代码是属于测试响应式系统用的代码
// 4.reactive函数对数据劫持。
const info = reactive({counter: 100, name:'why'});
console.log('==自动依赖收集==')
// 自动依赖收集
watchEffect(function () {
  console.log(info.counter * 2);
})
watchEffect(function () {
  console.log(info.counter * info.counter);
})
watchEffect(function () {
    console.log(info.name);
})
console.log('======更新======')
// 修改counter
info.counter++;
// dep.notify();  // 已经移到set函数中实现自动通知更新
```

可以看到，这里我们增加了`getDep`和`reactive`函数的实现。

- `getDep`函数的作用就是拿到某个属性指定的`dep`对象。因为我们这里将所有的依赖收集到一个`WeakMap`集合中，该集合的`key`是一个对象，`value`又是一个`Map`集合。`value`中`Map`集合的`key`是对象的某个属性，值是一个`dep`对象（`dep`对象中用`Set`集合存放副作用函数）。这样我们就可以实现将同一个属性的所有副作用函数存到单独的一个`dep`对象中，而不像前面案例将所有属性的副作用函数都存到同一个`dep`对象中，如图19-16所示。
- `reactive`函数的作用就是采用`Object.defineProperty`（`Vue.js 2`的实现）来对数据进行劫持。在该函数中，我们先遍历`raw`原始对象，接着对原始对象每一个属性的`get`和`set`进行劫持。在`get`函数中我们调用了`Dep`对象的`depend`来自动进行依赖收集，在`set`函数中调用`Dep`对象的`notify`函数来通知收集的依赖函数再次执行，从而实现了响应式。

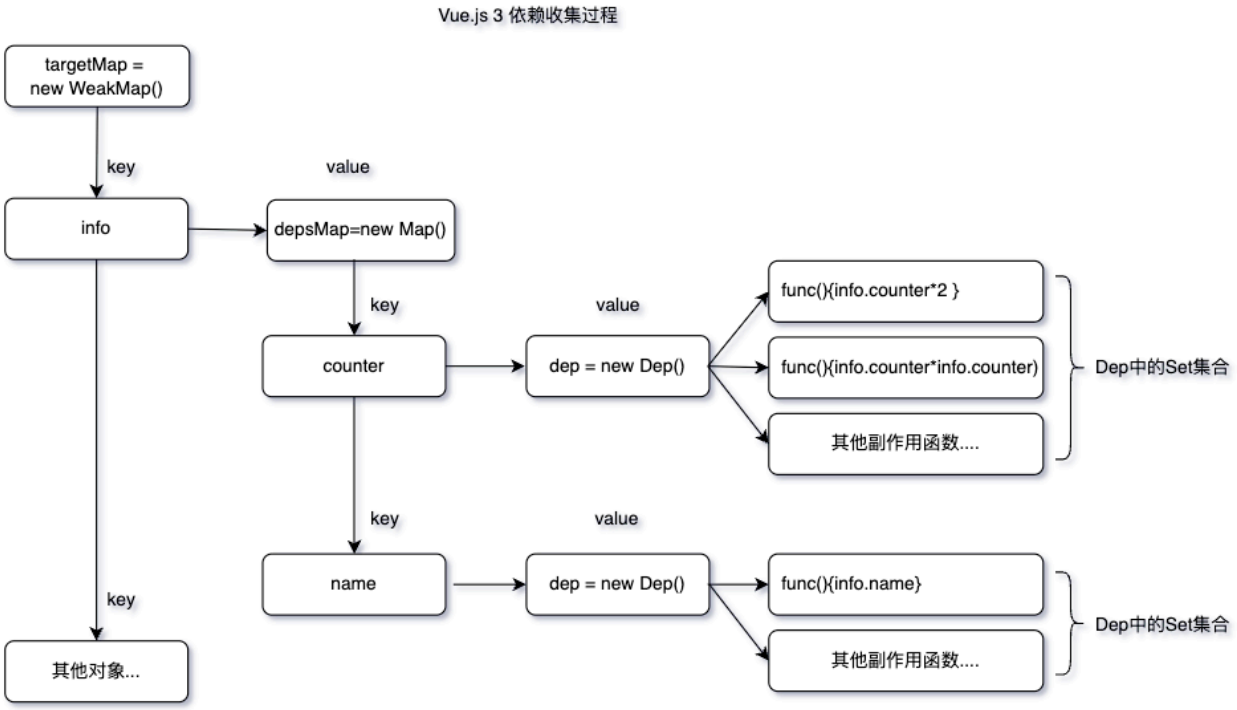

图19-16 Vue.js 3依赖收集过程

完成了响应式系统开发后，在浏览器中运行`index.html`文件，控制台的输出如图19-17所示。

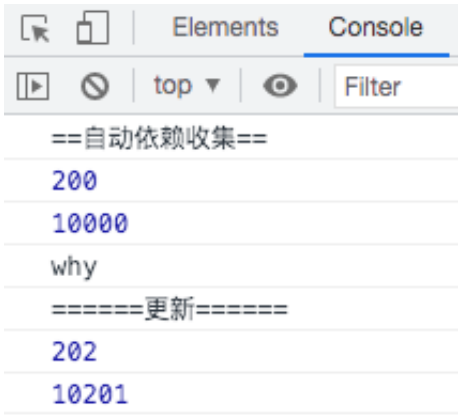

图19-17 自动收集依赖和更新

最后，我们来总结一下以上代码的执行过程：

- 首先，使用 reactive 函数来包裹原生的 info 对象。
- 接着，调用 watchEffect 函数来自动进行依赖收集。该函数需要传递一个副作用函数，该副作用函数会被立即执行。但是，在执行前会把该副作用函数先赋给全局的activeEffect变量。
- 然后，在执行副作用函数时，会发现引用了info中的counter属性。这时便会触发该属性的get函数，get函数中会调用指定dep对象的depend函数，将全局的activeEffect函数收集到dep中。这样就实现自动收集依赖。同样的原理，下面调用的watchEffect函数也会这样自动收集依赖。
- 最后，当我们修改info中的counter时，会触发该属性的set函数，该函数中会调用该属性对应dep对象的notify函数来通知更新，从而实现了响应式更新。

4.Vue.js 3响应式系统的实现

在Vue.js 3中，响应式系统采用了Proxy来进行数据劫持，下面我们来看看这两种实现方式有什么区别。

以下是 Object.defineProperty 和 Proxy 实现响应式系统的区别。

1. Object.defineProperty在劫持对象的属性时，如果新增元素，需要再次调用defineProperty。例如Vue.js 2提供的$set API。而Proxy劫持的是整个对象，不需要做特殊处理。
2. Proxy不会修改原始的对象。
    - 在使用Object.defineProperty时，只要修改原始对象就可以触发拦截。
    - 而使用Proxy，必须修改代理对象，即修改Proxy的实例才会触发拦截。
3. Proxy能观察的类型比defineProperty更丰富，例如：
    - has方法：捕获 in 操作符。
    - deleteProperty方法：捕获 delete 操作符。
    - 等等
4. 作为新标准，Proxy将受到浏览器厂商的重点性能优化。
5. Proxy也有缺点，它不兼容IE，也没有polyfill。而Object.defineProperty支持到 IE9。

下面我们修改 reactive.js 文件，使用 Proxy 来实现响应式系统，代码如下所示：

```
......
// Vue.js 3对原生（raw）数据进行劫持
function reactive(raw) {
  return new Proxy(raw, {
    get(target, key) { // 这里 target === raw
      const dep = getDep(target, key);
      dep.depend();
      return target[key];
    },
    set(target, key, newValue) {
      const dep = getDep(target, key);
      target[key] = newValue;
      dep.notify();
    }
  })
}
......
```

这次仅仅是修改了Vue.js 3中`reactive`函数的实现，对于`row`数据的劫持改成了`Proxy`实现，其他的代码保持不变。接着，在`reactive`函数中，我们新建一个`Proxy`对象，该构造器函数的第一个参数是我们需要代理的原生`raw`对象，第二个参数是带有`get`和`set`函数的对象。

然后，在`get`函数中调用`dep`对象的`depend`函数来自动收集依赖，在`set`函数中调用`dep`对象的`notify`函数来通知依赖的执行，从而实现数据的响应式。

最后，我们在浏览器中运行`index.html`文件，效果和前面案例一样。

19.2.3 应用程序入口模块

前面我们介绍过，一个精简版的 Mini-Vue3 框架，它需包含渲染系统、响应式系统和应用程序入口这三个模块。现在，渲染系统和响应式系统已经开发完了，接下来我们需要编写应用程序入口模块，该模块需包含以下功能：

1. 使用createApp方法创建一个app对象。

2. app对象具有一个mount方法，可以将根组件挂载到指定的DOM元素上。

我们在`learn_vuesource`项目的`src`目录下新建`04_Mini-Vue3`文件夹，然后在该文件夹下分别新建`index.js、reactive.js、renderer.js、index.html`文件。

在 index.js 文件中编写应用程序的入口模块，代码如下所示：

```
// 1.createApp函数，需要接收一个根组件
function createApp(rootComponent) {
  return {
    // 2.mount函数，用于将组件挂载到指定的DOM上
    mount(selector) {
      const container = document.querySelector(selector);
      let isMounted = false;
      let oldVNode = null;

      // 自动依赖收集。第一次挂载 或 页面数据发生改变都会触发该副作用函数回调
      watchEffect(function() {
        if (!isMounted) {
          // 第一次挂载。例如，初始化计数器
          oldVNode = rootComponent.render(); // 获取VNode，render函数对data的响应式数据有依赖
          mount(oldVNode, container);
          isMounted = true;
        } else {
          // 页面发生更新 。例如，计数器+1操作
          const newVNode = rootComponent.render(); // 获取VNode
          patch(oldVNode, newVNode);
          oldVNode = newVNode;
        }
      })
    }
  }
}
```

可以看到，我们定义了一个`createApp`函数，该函数需要接收一个根组件，用于创建一个`app`对象。该`app`对象有一个`mount`方法。以下是`mount`方法的具体实现。

- 首先，在`mount`方法中通过`selector`参数来获取将要挂载根组件的`container`元素。
- 接着，调用响应式系统的`watchEffect`函数来自动收集依赖，并给该函数传递了一个副作用函数。
- 然后，在副作用函数中，我们做了以下的操作。
  - 如果根组件没有挂载过，那么先调用`rootComponent`组件的`render`函数，由于`render`函数对`data`的`counter`有依赖，那当调用`render`函数的时候会将该副作用函数收集到`dep`中去。当拿到`vnode`之后再调用渲染系统的`mount`函数来将根组件挂载到`container`元素上。
  - 如果根组件已经挂载，那么在拿到新的`vnode`之后，调用渲染系统的`patch`函数来对比新旧

`vnode`，从而决定如何处理旧节点的更新。

接着，我们来看看 index.html 文件的实现，代码如下所示：

```
<!DOCTYPE html>
<html lang="en">
......
<body>
  <div id="app"></div>
  <!-- 1.渲染系统。这里是直接拷贝前面编写的 -->
  <script src="./renderer.js"></script>
  <!-- 2.响应式系统。这里是直接拷贝前面编写的，记住要删里面测试的代码） -->
  <script src="./reactive.js"></script>
  <!-- 3.createApp入口 -->
  <script src="./index.js"></script>
  <script>
    // 1.创建根组件
    const App = {
      data: reactive({
        counter: 0
      }),
      render() {
        return h("div", null, [
          h("h2", null, `当前计数: ${this.data.counter}`),
          h("button", {
            onClick: () => {
              this.data.counter++
              console.log(this.data.counter);
            }
          }, "+1")
        ])
      }
    }
    // 2.挂载根组件
    const app = createApp(App);
    app.mount("#app");
  </script>
</body>
</html>
```

可以看到，这里分别导入了渲染系统、响应式系统和`createApp`函数。接着，我们在`<script>`标签中创建了一个带有`data`属性和`render`函数的`App`根组件。

其中，`data`属性的值是一个`reactive`返回的响应式对象，`render`函数中编写的是经常用到的计数器布局。在`render`函数中，`<h2>`引用了`data`中的`counter`属性来显示当前计数，当单击"+1"按钮时，修改`data`中的`counter`来触发页面响应式刷新。

然后，我们调用`createApp`函数来创建`app`对象，接着调用`app`对象的`mount`方法将`App`组件挂载到`id`为`app`的元素上。需要注意的是，`mount`方法中的`watchEffect`函数会自动去收集依赖。

最后，我们再来看看渲染系统（`renderer.js`）和响应式系统（`reactive.js`）的完整代码。

renderer.js 文件，由于是直接拷贝前面编写好的代码，这里就不再阐述了，代码如下所示：

```
const h = (tag, props, children) => {
  // VNode -> Javascript对象 -> {}
  return {
    tag,
    props,
    children
  }
}

const mount = (vnode, container) => {
  // VNode -> Element
  // 1.创建出真实的原生对象，并且在VNode上保存el
  const el = vnode.el = document.createElement(vnode.tag);

  // 2.处理props
  if (vnode.props) {
    for (const key in vnode.props) {
      const value = vnode.props[key];

      if (key.startsWith("on")) { // 对事件监听的判断
        el.addEventListener(key.slice(2).toLowerCase(), value)
      } else {
        el.setAttribute(key, value);
      }
    }
  }

  // 3.处理children
  if (vnode.children) {
    if (typeof vnode.children === "string") {
      el.textContent = vnode.children;
    } else {
      vnode.children.forEach(item => {
        mount(item, el);
      })
    }
  }

  // 4.将el挂载到container元素上
  container.appendChild(el);
}

const patch = (n1, n2) => {
  if (n1.tag !== n2.tag) {
    const n1ElParent = n1.el.parentElement;
```

```
    n1ElParent.removeChild(n1.el);
    mount(n2, n1ElParent);
  } else {
    // 1.取出el对象，并且在n2中进行保存
    const el = n2.el = n1.el;

    // 2.处理props
    const oldProps = n1.props || {};
    const newProps = n2.props || {};
    // 2.1.获取所有的newProps添加到el
    for (const key in newProps) {
      const oldValue = oldProps[key];
      const newValue = newProps[key];
      if (newValue !== oldValue) {
        if (key.startsWith("on")) { // 对事件监听的判断
          el.addEventListener(key.slice(2).toLowerCase(), newValue)
        } else {
          el.setAttribute(key, newValue);
        }
      }
    }

    // 2.2.删除旧的props
    for (const key in oldProps) {
      if (key.startsWith("on")) { // 对事件监听的判断
        const value = oldProps[key];
        el.removeEventListener(key.slice(2).toLowerCase(), value)
      }
      if (!(key in newProps)) {
        el.removeAttribute(key);
      }
    }

    // 3.处理children
    const oldChildren = n1.children || [];
    const newChidlren = n2.children || [];

    if (typeof newChidlren === "string") { // 情况一: newChildren本身是一个string
      // 边界情况 (edge case)
      if (typeof oldChildren === "string") {
        if (newChidlren !== oldChildren) {
          el.textContent = newChidlren
        }
      } else {
        el.innerHTML = newChidlren;
      }
    } else { // 情况二: newChildren本身是一个数组
      if (typeof oldChildren === "string") {
        el.innerHTML = "";
```

```
        newChidlren.forEach(item => {
          mount(item, el);
        })
      } else {
        // 如 oldChildren 为: [v1, v2, v3, v8, v9]
        // 如 newChildren 为: [v1, v5, v6]
        // 3.1.前面有相同节点的原生进行patch操作
        const commonLength = Math.min(oldChildren.length, newChidlren.length);
        for (let i = 0; i < commonLength; i++) {
          patch(oldChildren[i], newChidlren[i]);
        }

        // 3.2.newChildren.length > oldChildren.length
        if (newChidlren.length > oldChildren.length) {
          newChidlren.slice(oldChildren.length).forEach(item => {
            mount(item, el);
          })
        }

        // 3.3.newChildren.length < oldChildren.length
        if (newChidlren.length < oldChildren.length) {
          oldChildren.slice(newChidlren.length).forEach(item => {
            el.removeChild(item.el);
          })
        }
      }
    }
  }
}
```

reactive.js文件，也是直接拷贝前面编写的好的代码，这里只是删除了之前编写的测试代码，因此这里也不再阐述，代码如下所示：

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

```
// const dep = new Dep(); // 已被getDep函数代替
let activeEffect = null;
function watchEffect(effect) {
  activeEffect = effect;
//   dep.depend(); // 自动依赖收集，已经移到get函数中实现
  effect();
  activeEffect = null;
}

// Map({key: value}): key是一个字符串
// WeakMap({key(对象): value}): key是一个对象，弱引用
// 1.创建一个WeakMap对象来存放所有的依赖
const targetMap = new WeakMap();
// 2.获取某一个属性对应的依赖set集合
function getDep(target, key) {
  // 2.1根据对象(target)取出对应的Map对象
  let depsMap = targetMap.get(target);
  if (!depsMap) {
    depsMap = new Map();
    targetMap.set(target, depsMap);
  }

  // 2.2取出具体的dep对象
  let dep = depsMap.get(key);
  if (!dep) {
    dep = new Dep();
    depsMap.set(key, dep);
  }
  return dep;
}

// Vue.js 3对原生（raw）数据进行劫持
function reactive(raw) {
  return new Proxy(raw, {
    get(target, key) { // 这里 target === raw
      const dep = getDep(target, key);
      dep.depend();
      return target[key];
    },
    set(target, key, newValue) {
      const dep = getDep(target, key);
      target[key] = newValue;
      dep.notify();
    }
  })
}
```

最后，在`index.html`文件上右击，选择"Open In Default Browser"。在浏览器中显示的效果如图19-18所示。可以看到，App组件正常挂载，当单击"+1"时，页面也会响应式的刷新。至此，一个精简版的Mini-Vue3 框架就实现了。

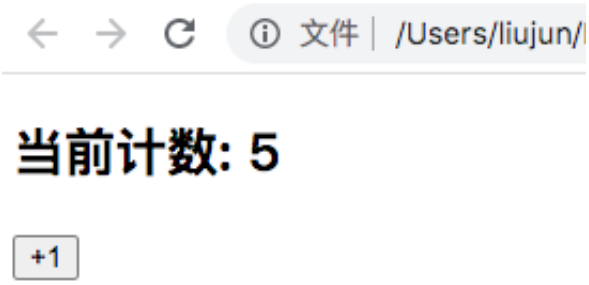

图19-18 Mini-Vue3的实现

19.3 本章小结

本章内容如下。

- VNode：Vue.js 3框架会对真实的元素节点进行抽象，抽象成VNode（虚拟节点）。
- 虚拟DOM：多个VNode节点组成一棵树结构时，便形成了虚拟DOM（Virtual DOM，简称VDOM）。
- Runtime模块：渲染系统，负责将 VNode 转换成真实DOM，并将其挂载到DOM上。
- Reactivity模块：响应式系统，负责数据的劫持和依赖收集。
- 应用程序入口模块：负责创建app实例和挂载应用到页面的DOM上。
- Mini-Vue3：实现了一个精简版的 Mini-Vue3框架，其中包含了渲染系统、响应式系统和应用程序入口模块。