6. Vue3 核心模块源码解析

1. 课程资料

1202 vue3-sourcecode.zip
2.56MB

2. 课程目标

- 初级：
  - 掌握 Vue3 中响应式及 AST（Abstract Syntax Tree） 的使用；
  - 了解编译器基本原理；
  - 了解简单 diff 算法；
- 中级：
  - 掌握编译器原理；
  - 掌握简单 diff 算法、双端 diff 算法；
  - 了解快速 diff 算法；
- 高级：
  - 深入理解 Vue3 核心源码，熟练掌握 Vue3 完整的执行逻辑；
  - 掌握简单 diff 算法、快速 diff 算法、双端 diff 算法；
  - 熟练掌握 Vue2 和 Vue3 diff 算法实现的异同点；

3. 课程大纲

1. Vue3 模块源码解析；
2. Vue 3 Diff算法；
3. 编译器原理

4. Vue3 模块源码解析（整体了解）

3.3.4

官方地址：https://github.com/vuejs/core/tree/main/packages

基本核心模块目录结构如下：

```
├─compiler-core
│   │  package.json
│   │
│   ├─src
│   │   │  ast.ts
│   │   │  codegen.ts
│   │   │  compile.ts
│   │   │  index.ts
│   │   │  parse.ts
│   │   │  runtimeHelpers.ts
│   │   │  transform.ts
│   │   │  utils.ts
│   │   │
│   │   └─transforms
│   │           transformElement.ts
│   │           transformExpression.ts
│   │           transformText.ts
│   │
│   └─__tests__
│       │  codegen.spec.ts
│       │  parse.spec.ts
│       │  transform.spec.ts
│       │
│       └─__snapshots__
│              codegen.spec.ts.snap
│
├─reactivity
│   │  package.json
│   │
│   ├─src
│   │      baseHandlers.ts
│   │      computed.ts
│   │      dep.ts
│   │      effect.ts
│   │      index.ts
│   │      reactive.ts
│   │      ref.ts
│   │
│   └─__tests__
│          computed.spec.ts
│          dep.spec.ts
```

```
│          effect.spec.ts
│          reactive.spec.ts
│          readonly.spec.ts
│          ref.spec.ts
│          shallowReadonly.spec.ts
│
├─runtime-core
│   │  package.json
│   │
│   ├─src
│   │   │  .pnpm-debug.log
│   │   │  apiInject.ts
│   │   │  apiWatch.ts
│   │   │  component.ts
│   │   │  componentEmits.ts
│   │   │  componentProps.ts
│   │   │  componentPublicInstance.ts
│   │   │  componentRenderUtils.ts
│   │   │  componentSlots.ts
│   │   │  createApp.ts
│   │   │  h.ts
│   │   │  index.ts
│   │   │  renderer.ts
│   │   │  scheduler.ts
│   │   │  vnode.ts
│   │   │
│   │   └─helpers
│   │           renderSlot.ts
│   │
│   └─__tests__
│          apiWatch.spec.ts
│          componentEmits.spec.ts
│          rendererComponent.spec.ts
│          rendererElement.spec.ts
│
├─runtime-dom
│   │  package.json
│   │
│   └─src
│          index.ts
│
├─runtime-test
│   └─src
│          index.ts
│          nodeOps.ts
│          patchProp.ts
│          serialize.ts
```

```
│
├─shared
│   │  package.json
│   │
│   └─src
│          index.ts
│          shapeFlags.ts
│          toDisplayString.ts
```

4.1 compiler-core

> Vue3的编译核心，核心作用就是将字符串转换成 抽象对象语法树AST；

4.1.1 目录结构

```
├─src
│   │  ast.ts         // ts类型定义，比如type，enum，interface等
│   │  codegen.ts         // 将生成的ast转换成render字符串
│   │  compile.ts // compile统一执行逻辑，有一个 baseCompile ，用来编译模板文件的

│   │  index.ts // 入口文件
│   │  parse.ts // 将模板字符串转换成 AST
│   │  runtimeHelpers.ts // 生成code的时候的定义常量对应关系
│   │  transform.ts // 处理 AST 中的 vue 特有语法
│   │  utils.ts
│   │
│   └─transforms
│          transformElement.ts
│          transformExpression.ts
│          transformText.ts
│
└─__tests__                  // 测试用例
    │  codegen.spec.ts
    │  parse.spec.ts
    │  transform.spec.ts
    │
    └─__snapshots__
           codegen.spec.ts.snap
```

4.1.2 compile逻辑

```
// src/index.ts
```

```
export { baseCompile } from "./compile";

// src/compiler.ts
import { generate } from "./codegen";
import { baseParse } from "./parse";
import { transform } from "./transform";
import { transformExpression } from "./transforms/transformExpression";
import { transformElement } from "./transforms/transformElement";
import { transformText } from "./transforms/transformText";

export function baseCompile(template, options) {
  // 1. 先把 template 也就是字符串 parse 成 ast
  const ast = baseParse(template);
  // 2. 给 ast 加点料 (- -#)
  transform(
    ast,
    Object.assign(options, {
      nodeTransforms: [transformElement, transformText, transformExpression],
    })
  );

  // 3. 生成 render 函数代码
  return generate(ast);
}
```

- baseParse

```
export function baseParse(content: string) {
  const context = createParserContext(content);
  return createRoot(parseChildren(context, []));
}

function createParserContext(content) {
  console.log("创建 paserContext");
  return {
    source: content,
  };
}

function createRoot(children) {
  return {
    type: NodeTypes.ROOT,
    children,
    helpers: [],
  };
```

```
}

function parseChildren(context, ancestors) {
  console.log("开始解析 children");
  const nodes: any = [];

  while (!isEnd(context, ancestors)) {
    let node;
    const s = context.source;

    if (startsWith(s, "{{")) {
      // 看看如果是 {{ 开头的话，那么就是一个插值， 那么去解析他
      node = parseInterpolation(context);
    } else if (s[0] === "<") {
      if (s[1] === "/") {
        // 这里属于 edge case 可以不用关心
        // 处理结束标签
        if (/[a-z]/i.test(s[2])) {
          // 匹配 </div>
          // 需要改变 context.source 的值 -> 也就是需要移动光标
          parseTag(context, TagType.End);
          // 结束标签就以为这都已经处理完了，所以就可以跳出本次循环了
          continue;
        }
      } else if (/[a-z]/i.test(s[1])) {
        node = parseElement(context, ancestors);
      }
    }

    if (!node) {
      node = parseText(context);
    }

    nodes.push(node);
  }

  return nodes;
}
```

- transform

```
export function transform(root, options = {}) {
  // 1. 创建 context

  const context = createTransformContext(root, options);
```

```
  // 2. 遍历 node
  traverseNode(root, context);

  createRootCodegen(root, context);

  root.helpers.push(...context.helpers.keys());
}

function createTransformContext(root, options): any {
  const context = {
    root,
    nodeTransforms: options.nodeTransforms || [],
    helpers: new Map(),
    helper(name) {
      // 这里会收集调用的次数
      // 收集次数是为了给删除做处理的，  （当只有 count 为0 的时候才需要真的删除掉）
      // helpers 数据会在后续生成代码的时候用到
      const count = context.helpers.get(name) || 0;
      context.helpers.set(name, count + 1);
    },
  };

  return context;
}

function traverseNode(node: any, context) {
  const type: NodeTypes = node.type;

  // 遍历调用所有的 nodeTransforms
  // 把 node 给到 transform
  // 用户可以对 node 做处理
  const nodeTransforms = context.nodeTransforms;
  const exitFns: any = [];
  for (let i = 0; i < nodeTransforms.length; i++) {
    const transform = nodeTransforms[i];

    const onExit = transform(node, context);
    if (onExit) {
      exitFns.push(onExit);
    }
  }

  switch (type) {
    case NodeTypes.INTERPOLATION:
      // 插值的点，在于后续生成 render 代码的时候是获取变量的值
      context.helper(TO_DISPLAY_STRING);
```

```
        break;

      case NodeTypes.ROOT:
      case NodeTypes.ELEMENT:

        traverseChildren(node, context);
        break;

      default:
        break;
    }



    let i = exitFns.length;
    // i-- 这个很巧妙
    // 使用 while 是要比 for 快 (可以使用 https://jsbench.me/ 来测试一下)
    while (i--) {
      exitFns[i]();
    }
  }

  function createRootCodegen(root: any, context: any) {
    const { children } = root;

    // 只支持有一个根节点
    // 并且还是一个 single text node
    const child = children[0];

    // 如果是 element 类型的话 ， 那么我们需要把它的 codegenNode 赋值给 root
    // root 其实是个空的什么数据都没有的节点
    // 所以这里需要额外的处理 codegenNode
    // codegenNode 的目的是专门为了 codegen 准备的  为的就是和 ast 的 node 分离开
    if (child.type === NodeTypes.ELEMENT && child.codegenNode) {
      const codegenNode = child.codegenNode;
      root.codegenNode = codegenNode;
    } else {
      root.codegenNode = child;
    }
  }
```

- generate

```
  export function generate(ast, options = {}) {
```

```
  // 先生成 context
  const context = createCodegenContext(ast, options);
  const { push, mode } = context;

  // 1. 先生成 preambleContext

  if (mode === "module") {
    genModulePreamble(ast, context);
  } else {
    genFunctionPreamble(ast, context);
  }

  const functionName = "render";

  const args = ["_ctx"];

  // _ctx,aaa,bbb,ccc
  // 需要把 args 处理成 上面的 string
  const signature = args.join(", ");
  push(`function ${functionName}(${signature}) {`);
  // 这里需要生成具体的代码内容
  // 开始生成 vnode tree 的表达式
  push("return ");
  genNode(ast.codegenNode, context);

  push("}");

  return {
    code: context.code,
  };
}
```

4.2 reactivity

> 负责Vue3中响应式实现的部分

4.2.1 目录结构

```
├─src
│      baseHandlers.ts // 基本处理逻辑
│      computed.ts // computed属性处理
│      dep.ts // effect对象存储逻辑
│      effect.ts // 依赖收集机制
│      index.ts // 入口文件
│      reactive.ts // 响应式处理逻辑
```

```
|      ref.ts // ref执行逻辑
|
└─__tests__ // 测试用例
        computed.spec.ts
        dep.spec.ts
        effect.spec.ts
        reactive.spec.ts
        readonly.spec.ts
        ref.spec.ts
        shallowReadonly.spec.ts
```

4.2.2 reactivity逻辑

- index.ts

```
export {
reactive,
  readonly,
  shallowReadonly,
  isReadonly,
  isReactive,
  isProxy,
} from "./reactive";

export { ref, proxyRefs, unRef, isRef } from "./ref";

export { effect, stop, ReactiveEffect } from "./effect";

export { computed } from "./computed";
```

- reactive.ts

```
import {
  mutableHandlers,
  readonlyHandlers,
  shallowReadonlyHandlers,
} from "./baseHandlers";

export const reactiveMap = new WeakMap();
export const readonlyMap = new WeakMap();
export const shallowReadonlyMap = new WeakMap();
```

```
export const enum ReactiveFlags {
  IS_REACTIVE = "__v_isReactive",
  IS_READONLY = "__v_isReadonly",
  RAW = "__v_raw",
}

export function reactive(target) {
  return createReactiveObject(target, reactiveMap, mutableHandlers);
}

export function readonly(target) {
  return createReactiveObject(target, readonlyMap, readonlyHandlers);
}

export function shallowReadonly(target) {
  return createReactiveObject(
    target,
    shallowReadonlyMap,
    shallowReadonlyHandlers
  );
}

export function isProxy(value) {
  return isReactive(value) || isReadonly(value);
}

export function isReadonly(value) {
  return !!value[ReactiveFlags.IS_READONLY];
}

export function isReactive(value) {
  // 如果 value 是 proxy 的话
  // 会触发 get 操作，而在 createGetter 里面会判断
  // 如果 value 是普通对象的话
  // 那么会返回 undefined ，那么就需要转换成布尔值
  return !!value[ReactiveFlags.IS_REACTIVE];
}

export function toRaw(value) {
  // 如果 value 是 proxy 的话 ,那么直接返回就可以了
  // 因为会触发 createGetter 内的逻辑
  // 如果 value 是普通对象的话，
  // 我们就应该返回普通对象
  // 只要不是 proxy ，只要是得到了 undefined 的话，那么就一定是普通对象
  // TODO 这里和源码里面实现的不一样，不确定后面会不会有问题
  if (!value[ReactiveFlags.RAW]) {
    return value;
```

```
  }

  return value[ReactiveFlags.RAW];
}

function createReactiveObject(target, proxyMap, baseHandlers) {
  // 核心就是 proxy
  // 目的是可以侦听到用户 get 或者 set 的动作

  // 如果命中的话就直接返回就好了
  // 使用缓存做的优化点
  const existingProxy = proxyMap.get(target);
  if (existingProxy) {
    return existingProxy;
  }

  const proxy = new Proxy(target, baseHandlers);

  // 把创建好的 proxy 给存起来，
  proxyMap.set(target, proxy);
  return proxy;
}
```

- ref.ts

```
import { trackEffects, triggerEffects, isTracking } from "./effect";
import { createDep } from "./dep";
import { isObject, hasChanged } from "@mini-vue/shared";
import { reactive } from "./reactive";

export class RefImpl {
  private _rawValue: any;
  private _value: any;
  public dep;
  public __v_isRef = true;

  constructor(value) {
    this._rawValue = value;
    // 看看value 是不是一个对象，如果是一个对象的话
    // 那么需要用 reactive 包裹一下
    this._value = convert(value);
    this.dep = createDep();
  }
```

```
  get value() {
    // 收集依赖
    trackRefValue(this);
    return this._value;
  }

  set value(newValue) {
    // 当新的值不等于老的值的话，
    // 那么才需要触发依赖
    if (hasChanged(newValue, this._rawValue)) {
      // 更新值
      this._value = convert(newValue);
      this._rawValue = newValue;
      // 触发依赖
      triggerRefValue(this);
    }
  }
}

export function ref(value) {
  return createRef(value);
}

function convert(value) {
  return isObject(value) ? reactive(value) : value;
}

function createRef(value) {
  const refImpl = new RefImpl(value);

  return refImpl;
}

export function triggerRefValue(ref) {
  triggerEffects(ref.dep);
}

export function trackRefValue(ref) {
  if (isTracking()) {
    trackEffects(ref.dep);
  }
}

// 这个函数的目的是
// 帮助解构 ref
// 比如在 template 中使用 ref 的时候，直接使用就可以了
// 例如： const count = ref(0) -> 在 template 中使用的话 可以直接 count
```

```
// 解决方案就是通过 proxy 来对 ref 做处理

const shallowUnwrapHandlers = {
  get(target, key, receiver) {
    // 如果里面是一个 ref 类型的话，那么就返回 .value
    // 如果不是的话，那么直接返回value 就可以了
    return unRef(Reflect.get(target, key, receiver));
  },
  set(target, key, value, receiver) {
    const oldValue = target[key];
    if (isRef(oldValue) && !isRef(value)) {
      return (target[key].value = value);
    } else {
      return Reflect.set(target, key, value, receiver);
    }
  },
};

// 这里没有处理 objectWithRefs 是 reactive 类型的时候
// TODO reactive 里面如果有 ref 类型的 key 的话， 那么也是不需要调用 ref.value 的
//  (but 这个逻辑在 reactive 里面没有实现)
export function proxyRefs(objectWithRefs) {
  return new Proxy(objectWithRefs, shallowUnwrapHandlers);
}

// 把 ref 里面的值拿到
export function unRef(ref) {
  return isRef(ref) ? ref.value : ref;
}

export function isRef(value) {
  return !!value.__v_isRef;
}
```

- effect

```
export function effect(fn, options = {}) {
  const _effect = new ReactiveEffect(fn);

  // 把用户传过来的值合并到 _effect 对象上去
  // 缺点就是不是显式的，看代码的时候并不知道有什么值
  extend(_effect, options);
  _effect.run();
```

```
  // 把 _effect.run 这个方法返回
  // 让用户可以自行选择调用的时机（调用 fn）
  const runner: any = _effect.run.bind(_effect);
  runner.effect = _effect;
  return runner;
}

export function stop(runner) {
  runner.effect.stop();
}
```

- computed

```
import { createDep } from "./dep";
import { ReactiveEffect } from "./effect";
import { trackRefValue, triggerRefValue } from "./ref";

export class ComputedRefImpl {
  public dep: any;
  public effect: ReactiveEffect;

  private _dirty: boolean;
  private _value

  constructor(getter) {
    this._dirty = true;
    this.dep = createDep();
    this.effect = new ReactiveEffect(getter, () => {
      // scheduler
      // 只要触发了这个函数说明响应式对象的值发生改变了
      // 那么就解锁，后续在调用 get 的时候就会重新执行，所以会得到最新的值
      if (this._dirty) return;

      this._dirty = true;
      triggerRefValue(this);
    });
  }

  get value() {
    // 收集依赖
    trackRefValue(this);
    // 锁上，只可以调用一次
    // 当数据改变的时候才会解锁
    // 这里就是缓存实现的核心
    // 解锁是在 scheduler 里面做的
```

```
    if (this._dirty) {
      this._dirty = false;
      // 这里执行 run 的话，就是执行用户传入的 fn
      this._value = this.effect.run();
    }

    return this._value;
  }
}

export function computed(getter) {
  return new ComputedRefImpl(getter);
}
```

4.3 runtime-core

> 运行的核心流程，其中包括初始化流程和更新流程

4.3.1 目录结构

```
├─src
│   │  apiInject.ts        // 提供provider和inject
│   │  apiWatch.ts        // 提供watch
│   │  component.ts        // 创建组件实例
│   │  componentEmits.ts        // 执行组件props 里面的 onXXX 的函数
│   │  componentProps.ts        // 获取组件props
│   │  componentPublicInstance.ts        // 组件通用实例上的代理,如$el,$emit等
│   │  componentRenderUtils.ts        // 判断组件是否需要重新渲染的工具类
│   │  componentSlots.ts        // 组件的slot
│   │  createApp.ts        // 根据跟组件创建应用
│   │  h.ts        // 创建节点
│   │  index.ts        // 入口文件
│   │  renderer.ts        // 渲染机制,包含diff
│   │  scheduler.ts // 触发更新机制
│   │  vnode.ts        // vnode节点
│   │
│   └─helpers
│           renderSlot.ts        // 插槽渲染实现
│
└─__tests__        // 测试用例
        apiWatch.spec.ts
        componentEmits.spec.ts
        rendererComponent.spec.ts
        rendererElement.spec.ts
```

4.3.2 runtime核心逻辑

- provide/inject

```
import { getCurrentInstance } from "./component";

export function provide(key, value) {
  const currentInstance = getCurrentInstance();

  if (currentInstance) {
    let { provides } = currentInstance;

    const parentProvides = currentInstance.parent?.provides;

    // 这里要解决一个问题
    // 当父级 key 和 爷爷级别的 key 重复的时候，对于子组件来讲，需要取最近的父级别组件的值
    // 那这里的解决方案就是利用原型链来解决
    // provides 初始化的时候是在 createComponent 时处理的，当时是直接把 parent.provides 赋值给组件的 provides 的
    // 所以，如果说这里发现 provides 和 parentProvides 相等的话，那么就说明是第一次做 provide(对于当前组件来讲)
    // 我们就可以把 parent.provides 作为 currentInstance.provides 的原型重新赋值
    // 至于为什么不在 createComponent 的时候做这个处理，可能的好处是在这里初始化的话，是有个懒执行的效果（优化点，只有需要的时候在初始化）
    if (parentProvides === provides) {
      provides = currentInstance.provides = Object.create(parentProvides);
    }

    provides[key] = value;
  }
}

export function inject(key, defaultValue) {
  const currentInstance = getCurrentInstance();

  if (currentInstance) {
    const provides = currentInstance.parent?.provides;

    if (key in provides) {
      return provides[key];
    } else if (defaultValue) {
      if (typeof defaultValue === "function") {
        return defaultValue();
      }
```

```
        return defaultValue;
      }
    }
  }
```

- watch

```
import { ReactiveEffect } from "@mini-vue/reactivity";
import { queuePreFlushCb } from "./scheduler";

// Simple effect.
export function watchEffect(effect) {
  doWatch(effect);
}

function doWatch(source) {
  // 把 job 添加到 pre flush 里面
  // 也就是在视图更新完成之前进行渲染（待确认？）
  // 当逻辑执行到这里的时候 就已经触发了 watchEffect
  const job = () => {
    effect.run();
  };

  // 这里用 scheduler 的目的就是在更新的时候
  // 让回调可以在 render 前执行 变成一个异步的行为（这里也可以通过 flush 来改变）
  const scheduler = () => queuePreFlushCb(job);

  const getter = () => {
    source();
  };

  const effect = new ReactiveEffect(getter, scheduler);

  // 这里执行的就是 getter
  effect.run();
}
```

- component创建

```
export function createComponentInstance(vnode, parent) {
  const instance = {
```

```
    type: vnode.type,
    vnode,
    next: null, // 需要更新的 vnode，用于更新 component 类型的组件
    props: {},
    parent,
    provides: parent ? parent.provides : {}, //  获取 parent 的 provides 作为当前
组件的初始化值 这样就可以继承 parent.provides 的属性了
    proxy: null,
    isMounted: false,
    attrs: {}, // 存放 attrs 的数据
    slots: {}, // 存放插槽的数据
    ctx: {}, // context 对象
    setupState: {}, // 存储 setup 的返回值
    emit: () => {},
  };

  // 在 prod 环境下的 ctx 只是下面简单的结构
  // 在 dev 环境下会更复杂
  instance.ctx = {
    _: instance,
  };

  // 赋值 emit
  // 这里使用 bind 把 instance 进行绑定
  // 后面用户使用的时候只需要给 event 和参数即可
  instance.emit = emit.bind(null, instance) as any;

  return instance;
}
```

- createApp

```
import { createVNode } from "./vnode";

export function createAppAPI(render) {
  return function createApp(rootComponent) {
    const app = {
      _component: rootComponent,
      mount(rootContainer) {
        console.log("基于根组件创建 vnode");
        const vnode = createVNode(rootComponent);
        console.log("调用 render，基于 vnode 进行开箱");
        render(vnode, rootContainer);
      },
```

```
    };

    return app;
  };
}
```

- 创建Vnode节点

```
import { createVNode } from "./vnode";
export const h = (type: any , props: any = null, children: string | Array<any>
= []) => {
  return createVNode(type, props, children);
};
```

- 入口文件

```
export * from "./h";
export * from "./createApp";
export { getCurrentInstance, registerRuntimeCompiler } from "./component";
export { inject, provide } from "./apiInject";
export { renderSlot } from "./helpers/renderSlot";
export { createTextVNode, createElementVNode } from "./vnode";
export { createRenderer } from "./renderer";
export { toDisplayString } from "@mini-vue/shared";
export {
  // core
  reactive,
  ref,
  readonly,
  // utilities
  unRef,
  proxyRefs,
  isReadonly,
  isReactive,
  isProxy,
  isRef,
  // advanced
  shallowReadonly,
  // effect
  effect,
  stop,
  computed,
```

```
} from "@mini-vue/reactivity";
```

- render

```
// 具体update的Diff见下节课内容;
function updateElement(n1, n2, container, anchor, parentComponent) {
    const oldProps = (n1 && n1.props) || {};
  const newProps = n2.props || {};
  // 应该更新 element
  console.log("应该更新 element");
  console.log("旧的 vnode", n1);
  console.log("新的 vnode", n2);

  // 需要把 el 挂载到新的 vnode
  const el = (n2.el = n1.el);

  // 对比 props
  patchProps(el, oldProps, newProps);

  // 对比 children
  patchChildren(n1, n2, el, anchor, parentComponent);
}
```

- scheduler

```
// 具体的调度机制见下节课内容
const queue: any[] = [];
const activePreFlushCbs: any = [];

const p = Promise.resolve();
let isFlushPending = false;

export function nextTick(fn?) {
  return fn ? p.then(fn) : p;
}

export function queueJob(job) {
  if (!queue.includes(job)) {
    queue.push(job);
    // 执行所有的 job
    queueFlush();
  }
}
```

```
function queueFlush() {
  // 如果同时触发了两个组件的更新的话
  // 这里就会触发两次 then （微任务逻辑）
  // 但是着是没有必要的
  // 我们只需要触发一次即可处理完所有的 job 调用
  // 所以需要判断一下 如果已经触发过 nextTick 了
  // 那么后面就不需要再次触发一次 nextTick 逻辑了
  if (isFlushPending) return;
  isFlushPending = true;
  nextTick(flushJobs);
}

export function queuePreFlushCb(cb) {
  queueCb(cb, activePreFlushCbs);
}

function queueCb(cb, activeQueue) {
  // 直接添加到对应的列表内就ok
  // todo 这里没有考虑 activeQueue 是否已经存在 cb 的情况
  // 然后在执行 flushJobs 的时候就可以调用 activeQueue 了
  activeQueue.push(cb);

  // 然后执行队列里面所有的 job
  queueFlush()
}

function flushJobs() {
  isFlushPending = false;

  // 先执行 pre 类型的 job
  // 所以这里执行的job 是在渲染前的
  // 也就意味着执行这里的 job 的时候 页面还没有渲染
  flushPreFlushCbs();

  // 这里是执行 queueJob 的
  // 比如 render 渲染就是属于这个类型的 job
  let job;
  while ((job = queue.shift())) {
    if (job) {
      job();
    }
  }
}

function flushPreFlushCbs() {
  // 执行所有的 pre 类型的 job
```

```
  for (let i = 0; i < activePreFlushCbs.length; i++) {
    activePreFlushCbs[i]();
  }
}
```

- vnode类型定义及格式规范

```
import { ShapeFlags } from "@mini-vue/shared";

export { createVNode as createElementVNode }

export const createVNode = function (
  type: any,
  props?: any,
  children?: string | Array<any>
) {
  // 注意 type 有可能是 string 也有可能是对象
  // 如果是对象的话，那么就是用户设置的 options
  // type 为 string 的时候
  // createVNode("div")
  // type 为组件对象的时候
  // createVNode(App)
  const vnode = {
    el: null,
    component: null,
    key: props?.key,
    type,
    props: props || {},
    children,
    shapeFlag: getShapeFlag(type),
  };

  // 基于 children 再次设置 shapeFlag
  if (Array.isArray(children)) {
    vnode.shapeFlag |= ShapeFlags.ARRAY_CHILDREN;
  } else if (typeof children === "string") {
    vnode.shapeFlag |= ShapeFlags.TEXT_CHILDREN;
  }

  normalizeChildren(vnode, children);

  return vnode;
};
```

```
export function normalizeChildren(vnode, children) {
  if (typeof children === "object") {
    // 暂时主要是为了标识出 slots_children 这个类型来
    // 暂时我们只有 element 类型和 component 类型的组件
    // 所以我们这里除了 element ，那么只要是 component 的话，那么children 肯定就是
slots 了
    if (vnode.shapeFlag & ShapeFlags.ELEMENT) {
      // 如果是 element 类型的话，那么 children 肯定不是 slots
    } else {
      // 这里就必然是 component 了，
      vnode.shapeFlag |= ShapeFlags.SLOTS_CHILDREN;
    }
  }
}
// 用 symbol 作为唯一标识
export const Text = Symbol("Text");
export const Fragment = Symbol("Fragment");

/**
 * @private
 */
export function createTextVNode(text: string = " ") {
  return createVNode(Text, {}, text);
}

// 标准化 vnode 的格式
// 其目的是为了让 child 支持多种格式
export function normalizeVNode(child) {
  // 暂时只支持处理 child 为 string 和 number 的情况
  if (typeof child === "string" || typeof child === "number") {
    return createVNode(Text, null, String(child));
  } else {
    return child;
  }
}

// 基于 type 来判断是什么类型的组件
function getShapeFlag(type: any) {
  return typeof type === "string"
    ? ShapeFlags.ELEMENT
    : ShapeFlags.STATEFUL_COMPONENT;
}
```

4.4 runtime-dom

> Vue3靠虚拟dom，实现跨平台的能力，runtime-dom提供一个渲染器，这个渲染器可以渲染虚拟dom节点到指定的容器中；

4.4.1 主要功能

```
// 源码里面这些接口是由 runtime-dom 来实现
// 这里先简单实现

import { isOn } from "@mini-vue/shared";
import { createRenderer } from "@mini-vue/runtime-core";

// 后面也修改成和源码一样的实现
function createElement(type) {
  console.log("CreateElement", type);
  const element = document.createElement(type);
  return element;
}

function createText(text) {
  return document.createTextNode(text);
}

function setText(node, text) {
  node.nodeValue = text;
}

function setElementText(el, text) {
  console.log("SetElementText", el, text);
  el.textContent = text;
}

function patchProp(el, key, preValue, nextValue) {
  // preValue 之前的值
  // 为了之后 update 做准备的值
  // nextValue 当前的值
  console.log(`PatchProp 设置属性:${key} 值:${nextValue}`);
  console.log(`key: ${key} 之前的值是:${preValue}`);

  if (isOn(key)) {
    // 添加事件处理函数的时候需要注意一下
    // 1. 添加的和删除的必须是一个函数，不然的话 删除不掉
    //    那么就需要把之前 add 的函数给存起来，后面删除的时候需要用到
    // 2. nextValue 有可能是匿名函数，当对比发现不一样的时候也可以通过缓存的机制来避免注册多次
    // 存储所有的事件函数
    const invokers = el._vei || (el._vei = {});
```

```
      const existingInvoker = invokers[key];
      if (nextValue && existingInvoker) {
        // patch
        // 直接修改函数的值即可
        existingInvoker.value = nextValue;
      } else {
        const eventName = key.slice(2).toLowerCase();
        if (nextValue) {
          const invoker = (invokers[key] = nextValue);
          el.addEventListener(eventName, invoker);
        } else {
          el.removeEventListener(eventName, existingInvoker);
          invokers[key] = undefined;
        }
      }
    } else {
      if (nextValue === null || nextValue === "") {
        el.removeAttribute(key);
      } else {
        el.setAttribute(key, nextValue);
      }
    }
  }

  function insert(child, parent, anchor = null) {
    console.log("Insert");
    parent.insertBefore(child, anchor);
  }

  function remove(child) {
    const parent = child.parentNode;
    if (parent) {
      parent.removeChild(child);
    }
  }

  let renderer;

  function ensureRenderer() {
    // 如果 renderer 有值的话，那么以后都不会初始化了
    return (
      renderer ||
      (renderer = createRenderer({
        createElement,
        createText,
        setText,
        setElementText,
```

```
      patchProp,
      insert,
      remove,
    }))
  );
}

export const createApp = (...args) => {
  return ensureRenderer().createApp(...args);
};

export * from "@vue/runtime-core"
```

4.5 runtime-test

> 可以理解成runtime-dom的延伸，,因为runtime-test对外提供的确实是dom环境的测试，方便用于runtime-core的测试；

4.5.1 目录结构

```
——src
index.ts
nodeOps.ts
patchProp.ts
serialize.ts
```

4.5.2 runtime-test核心逻辑

- index.ts

```
// 实现 render 的渲染接口
// 实现序列化
import { createRenderer } from "@mini-vue/runtime-core";
import { extend } from "@vue/shared";
import { nodeOps } from "./nodeOps";
import { patchProp } from "./patchProp";

export const { render } = createRenderer(extend({ patchProp }, nodeOps));

export * from "./nodeOps";
export * from "./serialize"
```

```
export * from '@mini-vue/runtime-core'
```

- nodeOps，节点定义及操作再runtime-core中的映射

```
export const enum NodeTypes {
  ELEMENT = "element",
  TEXT = "TEXT",
}

let nodeId = 0;
// 这个函数会在 runtime-core 初始化 element 的时候调用
function createElement(tag: string) {
  // 如果是基于 dom 的话 那么这里会返回 dom 元素
  // 这里是为了测试 所以只需要反正一个对象就可以了
  // 后面的话 通过这个对象来做测试
  const node = {
    tag,
    id: nodeId++,
    type: NodeTypes.ELEMENT,
    props: {},
    children: [],
    parentNode: null,
  };

  return node;
}

function insert(child, parent) {
  parent.children.push(child);
  child.parentNode = parent;
}

function parentNode(node) {
  return node.parentNode;
}

function setElementText(el, text) {
  el.children = [
    {
      id: nodeId++,
      type: NodeTypes.TEXT,
      text,
      parentNode: el,
    },
  ];
```

```
}

export const nodeOps = { createElement, insert, parentNode, setElementText };
```

- serialize，序列化： 把Vnode处理成 string

```
// 把 node 给序列化
// 测试的时候好对比

import { NodeTypes } from "./nodeOps";

// 序列化： 把一个对象给处理成 string （进行流化）
export function serialize(node) {
  if (node.type === NodeTypes.ELEMENT) {
    return serializeElement(node);
  } else {
    return serializeText(node);
  }
}

function serializeText(node) {
  return node.text;
}

export function serializeInner(node) {
  // 把所有节点变成一个string
  return node.children.map((c) => serialize(c)).join(``);
}

function serializeElement(node) {
  // 把 props 处理成字符串
  // 规则:
  // 如果 value 是 null 的话 那么直接返回 ``
  // 如果 value 是 `` 的话，那么返回 key
  // 不然的话返回 key = value（这里的值需要字符串化）
  const props = Object.keys(node.props)
    .map((key) => {
      const value = node.props[key];
      return value == null
        ? ``
        : value === ``
        ? key
        : `${key}=${JSON.stringify(value)}`;
    })
```

```
    .filter(Boolean)
    .join(" ");

  console.log("node---------", node.children);
  return `<${node.tag}${props ? ` ${props}` : ``}>${serializeInner(node)}</${
    node.tag
  }>`;
}
```

4.6 shared

> 公用逻辑

4.6.1 具体逻辑

```
export * from '../src/shapeFlags';
export * from '../src/toDisplayString';

export const isObject = val => {
        return val !== null && typeof val === 'object';
};

export const isString = val => typeof val === 'string';

const camelizeRE = /-(\w)/g;
/**
 * @private
 * 把中划线命名方式转换成驼峰命名方式
 */
export const camelize = (str: string): string => {
        return str.replace(camelizeRE, (_, c) => (c ? c.toUpperCase() : ''));
};

export const extend = Object.assign;

// 必须是 on+一个大写字母的格式开头
export const isOn = key => /^on[A-Z]/.test(key);

export function hasChanged(value, oldValue) {
        return !Object.is(value, oldValue);
}

export function hasOwn(val, key) {
        return Object.prototype.hasOwnProperty.call(val, key);
```

```
}

/**
 * @private
 * 首字母大写
 */
export const capitalize = (str: string) => str.charAt(0).toUpperCase() +
str.slice(1);

/**
 * @private
 * 添加 on 前缀，并且首字母大写
 */
export const toHandlerKey = (str: string) => (str ? `on${capitalize(str)}` :
``);

// 用来匹配 kebab-case 的情况
// 比如 onTest-event 可以匹配到 T
// 然后取到 T 在前面加一个 - 就可以
// \BT 就可以匹配到 T 前面是字母的位置
const hyphenateRE = /\B([A-Z])/g;
/**
 * @private
 */
export const hyphenate = (str: string) => str.replace(hyphenateRE,
'-$1').toLowerCase();


// 组件的类型
export const enum ShapeFlags {
  // 最后要渲染的 element 类型
  ELEMENT = 1,
  // 组件类型
  STATEFUL_COMPONENT = 1 << 2,
  // vnode 的 children 为 string 类型
  TEXT_CHILDREN = 1 << 3,
  // vnode 的 children 为数组类型
  ARRAY_CHILDREN = 1 << 4,
  // vnode 的 children 为 slots 类型
  SLOTS_CHILDREN = 1 << 5
}

export const toDisplayString = (val) => {
  return String(val);
};
```

我们需要关注的核心包

compiler-core

- compiler-core

- parse，模板字符串转为基础 ast

- transform 进行转换

- codegen 生成目标代码

模板什么时候是字符串，什么时候可能不是字符串

h('div', null)

编译器核心代码实现

```
// 初步处理，通过 parser 将模板字符串转为 ast
const ast = isString(template) ? baseParse(template, options) : template

// 通过转换器将 ast 转为目标代码 ast
transform(
  ast,
  extend({})
)

// 通过目标代码 ast 生成目标代码
generate(
    ast,
    extend({})
  )
```

补充：如果你以后需要自定义编译器实现低代码特性语言的支持（DSL）

Vue3 编译器的核心代码

reactivity

核心，针对于不同的响应式对象，vue3 reactivity 源码做了一些什么处理

通过 `createReactiveObject` 创建响应式对象，但是创建响应式有很多规则的

- 基础类型，不需要通过我们这个响应式对象进行代理的

```
if (!isObject(target)) {
    return target
  }
```

Q：那针对于基础类型的，难道不能用 reactive 了吗？

```
ref('ffff')
reactive({value: 'ffff'})

const count = ref(0)

console.log(count) // { value: 0 }
```

- weakMap 用来存储响应式对象的引用关系

```
  const proxy = new Proxy(
    target,
    targetType === TargetType.COLLECTION ? collectionHandlers : baseHandlers
  )
  proxyMap.set(target, proxy)
```

runtime-core

runtime 包代表着什么？

js 解释型语言，所以本身其实不存在编译过程的，在处理源码的时候，我们认为分了两个时机：

- 编译时
- 运行时

预先将源码中 `<div>hello</div> -> h('div', 'hello')` 就像是编译，项目跑起来的时候，就直接运行了。

Taro、uni-app。

一套代码，你在写的时候就那一套，然后你在写完以后会打个包，这时候打包的过程我们就当做为编译时。

你针对不同端，将项目跑起来，这是就是运行时。

运行时主要就是将编译后的内容进行进一步加工，比如针对于生命周期的处理、watch、directives 等相关处理

```
export const onBeforeMount = createHook(LifecycleHooks.BEFORE_MOUNT)
export const onMounted = createHook(LifecycleHooks.MOUNTED)
export const onBeforeUpdate = createHook(LifecycleHooks.BEFORE_UPDATE)
export const onUpdated = createHook(LifecycleHooks.UPDATED)
export const onBeforeUnmount = createHook(LifecycleHooks.BEFORE_UNMOUNT)
export const onUnmounted = createHook(LifecycleHooks.UNMOUNTED)
export const onServerPrefetch = createHook(LifecycleHooks.SERVER_PREFETCH)
```

这里重点认识一下两个编程思路

1. 对外统一口径，设计一致化的 API
2. 策略模式处理多场景

runtime-dom

针对 dom （浏览器环境）的运行时

它是为了补充 runtime-core 针对于浏览器环境运行时的一些特殊处理，比如我可以针对 dom 操作相关（浏览器环境）定义一些内置指令——vModel、vOn、vShow 特殊处理

> *这里重点提示一下，vIf 指令呢，属于编译时处理的*

补充说明

Vue3 中的编译时和运行时

- 编译时，就是将 xxx.vue 进行解析，解析生成对应 js 可运行代码
- 运行时，就是将编译时生成的代码进一步处理，关于生命周期的声明、watch

5. Vue 3 Diff算法

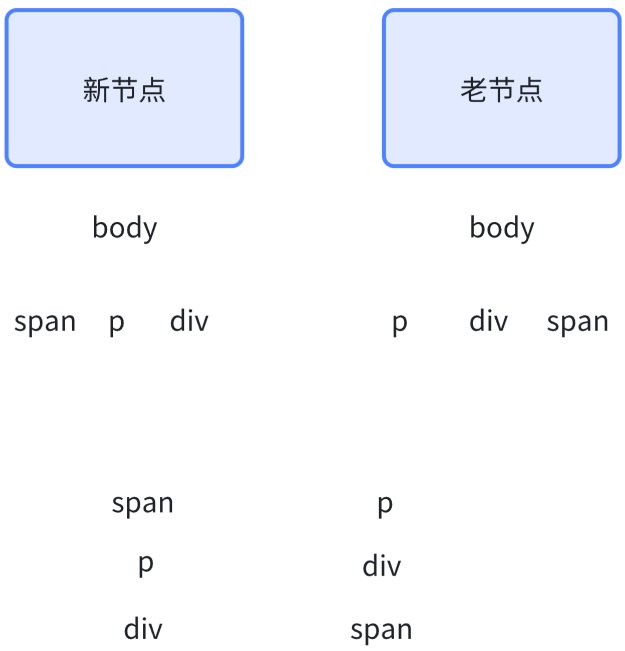
简单 diff（只是教学）

patch

在上一节中，我们通过减少 DOM 操作的次数，提升了更新性能。 但这种方式仍然存在可优化的空间。举个例子，假设新旧两组子节点 的内容如下：

```
const oldChildren = [
    { type: 'p' }, { type: 'div' }, { type: 'span' }
]
const newChildren = [
    { type: 'span' }, { type: 'p' }, { type: 'div' }
]
```

```
const oldChildren = [
  { type: 'p' }, { type: 'div' }, { type: 'span' }
]
const newChildren = [
  { type: 'span' }, { type: 'p' }, { type: 'div' }
]
```



如果使用上一节介绍的算法来完成上述两组子节点的更新，则需 要 6 次 DOM 操作。

- 调用 patch 函数在旧子节点 `{ type: 'p'}` 与新子节点 `{ type: "span"}`，了之间打补丁，由于两者是不同的标签，所以patch 函数会卸载 `{type: 'p'}` 了，然后再挂载 `{type: 'span'}` 了，这需要执行 2 次 DOM 操作。
- 与第 1 步类似，卸载旧子节点 `{ type: 'div'}` 了，然后再挂载新子节点 `{type: 'p'}` 了，这也需要执行 2 次 DOM 操作。
- 与第 1 步类似，卸载旧子节点 `{type: 'span'}`，然后再挂载新子节点 `{ type: 'div'}` 了，同样需要执行 2 次 DOM 操作。

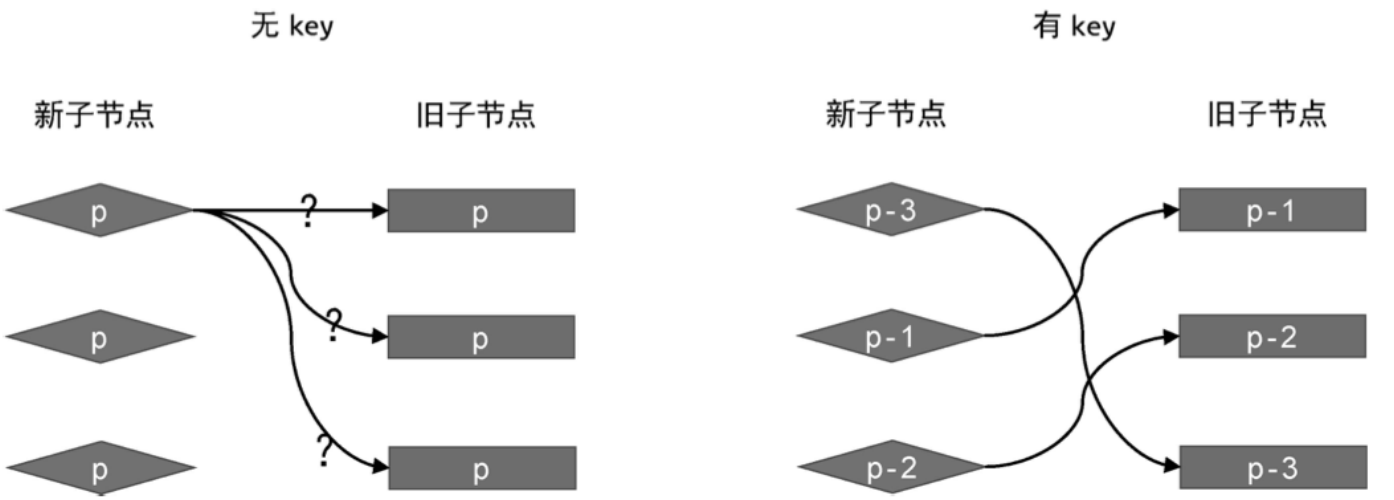
因此，一共进行 6 次 DOM 操作才能完成上述案例的更新。但是， 观察新旧两组子节点，很容易发现，二者只是顺序不同。所以最优的 处理方式是，通过 DOM 的移动来完成子节点的更新，这要比不断地执 行子节点的卸载和挂载性能更好。但是，想要通过 DOM 的移动来完成 更新，必须要保证一个前提：新旧两组子节点中的确存在可复用的节 点。这个很好理解，如果新的子节点没有在旧的一组子节点中出现， 就无法通过移动节点的方式完成更新。所以现在问题变成了：应该如 何确定新的子节点是否出现在旧的一组子节点中呢？拿上面的例子来 说，怎么确定新的一组子节点中第 1 个子节点 ^ type 'span' ` 与旧的一组子节点中第 3 个子节点相同呢？一种解决方案是，通过 vnode.type 来判断，只要 vnode.type 的值相同，我们就认为两者 是相同的节点。但这种方式并不可靠，思考如下例子：

```
const oldChildren = [
    { type: 'p', children: '1' },
    { type: 'p', children: '2' },
    { type: 'p', children: '3' }
]
const newChildren = [
    { type: 'p', children: '3' },
    { type: 'p', children: '1' },
    { type: 'p', children: '2' }
]
```

观察上面两组子节点，我们发现，这个案例可以通过移动 DOM 的 方式来完成更新。但是所有节点的 vnode.type 属性值都相同，这导 致我们无法确定新旧两组子节点中节点的对应关系，也就无法得知应 该进行怎样的 DOM 移动才能完成更新。这时，我们就需要引入额外的 key 来作为 vnode 的标识，如下面的代码所示：

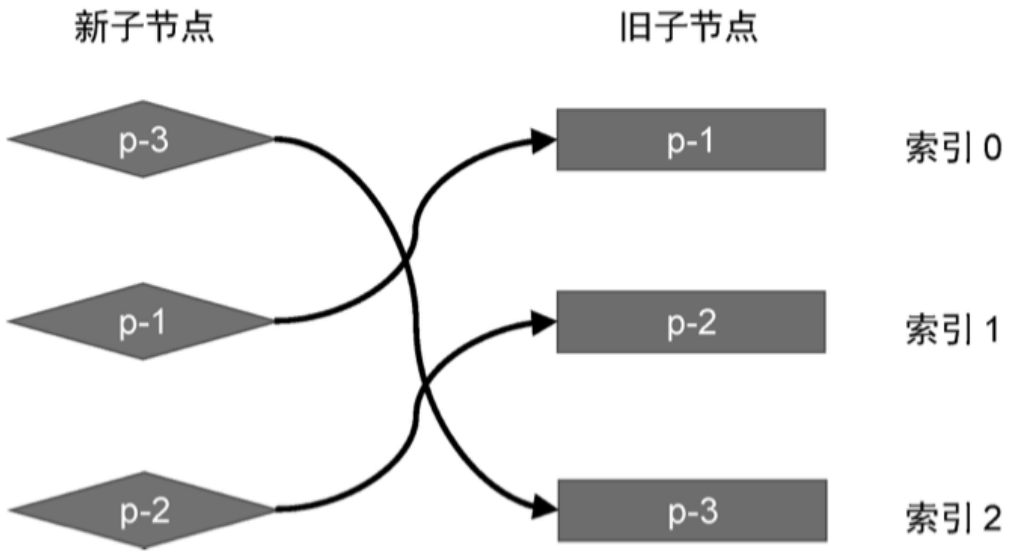
```
const oldChildren = [
    { type: 'p', children: '1', key: '1' },
    { type: 'p', children: '2', key: '2' },
    { type: 'p', children: '3', key: '3' }
]
const newChildren = [
    { type: 'p', children: '3', key: '3' },
    { type: 'p', children: '1', key: '1' },
    { type: 'p', children: '2', key: '2' }
]
```

key 属性就像虚拟节点的“身份证”号，只要两个虚拟节点的 type 属性值和 key 属性值都相同，那么我们就认为它们是相同的，即可以 进行 DOM 的复用。图 9-4 展示了有 key 和 无 key 时新旧两组子节点 的映射情况。



接下来会进行 patch 操作，patch 理解为打补丁，这个操作是不需要移动元素的，而只需要更新元素即可，不改变顺序。

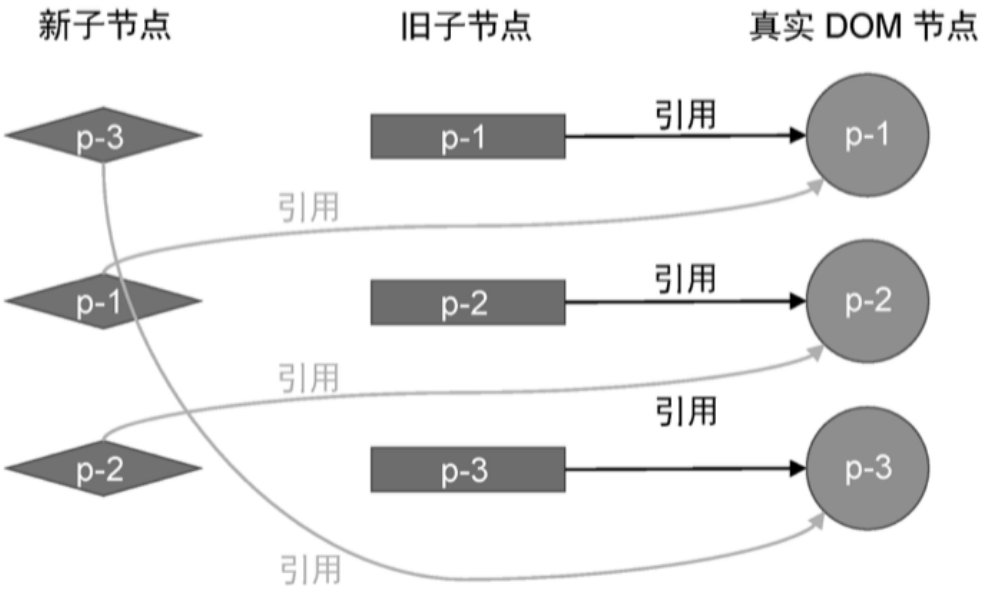
接下来找寻需要移动的元素，进行元素移动



详述整个过程

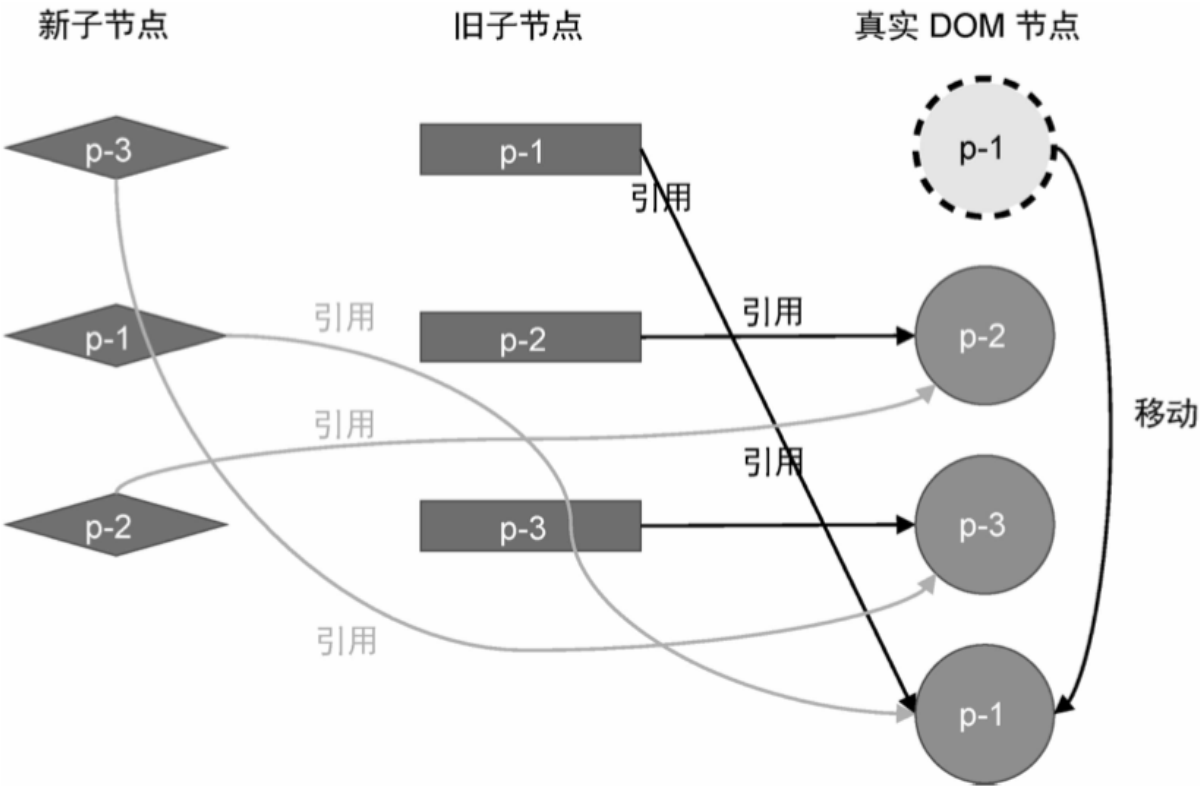
- 第一步：取新的一组子节点中的第一个节点 p-3，它的 key 为 3。尝试在旧的一组子节点中找到具有相同 key 值的可复用节点， 发现能够找到，并且该节点在旧的一组子节点中的索引为 2。
- 第二步：取新的一组子节点中的第二个节点 p-1，它的 key 为 1。尝试在旧的一组子节点中找到具有相同 key 值的可复用节点， 发现能够找到，并且该节点在旧的一组子节点中的索引为 0。
  到了这一步我们发现，索引值递增的顺序被打破了。节点p-1在旧 children 中的素引是 0，它小于节点p-3在日 children 中的索引2。这说明节点p-1 在旧children 中排在节点p-3前面，但在新的 children 中，它排在节点p-3后面。因此，我们能够得出一个结论：节点p-1对应的真实 DOM 需要移动。
- 第三步：取新的一组子节点中的第三个节点 p-2，它的 key 为 2。尝试在旧的一组子节点中找到具有相同 key 值的可复用节点， 发现能够找到，并且该节点在旧的一组子节点中的索引为 1。
  到了这一步我们发现，节点p-2 在旧children 中的素引1要小于节点口-3在日children 中的索引2。这说明，节点p-2在旧 children 中排在节点p-3前面，但在新的 children 中，它排在节点口-3后面。因此，节点口-2对应的真实 DOM 也需要移动。

复用

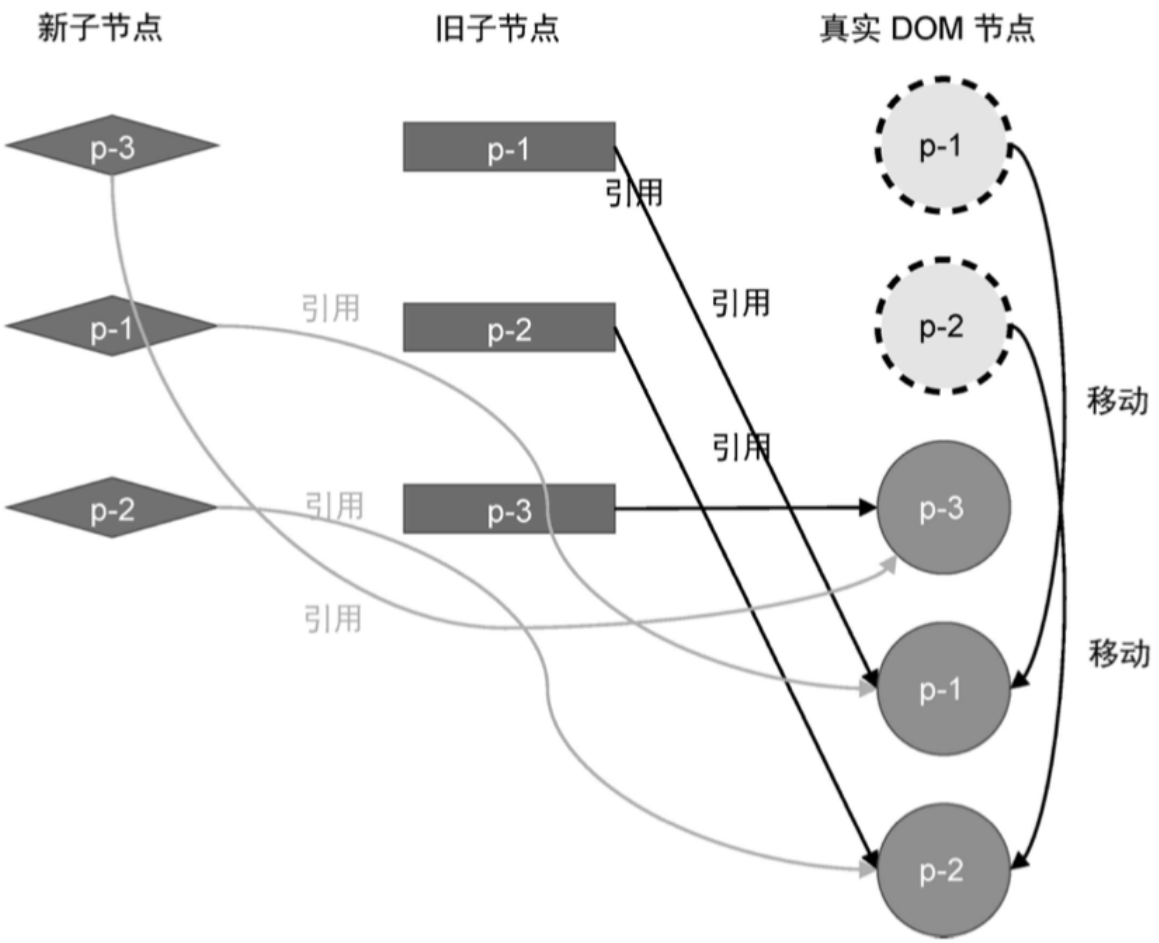


移动

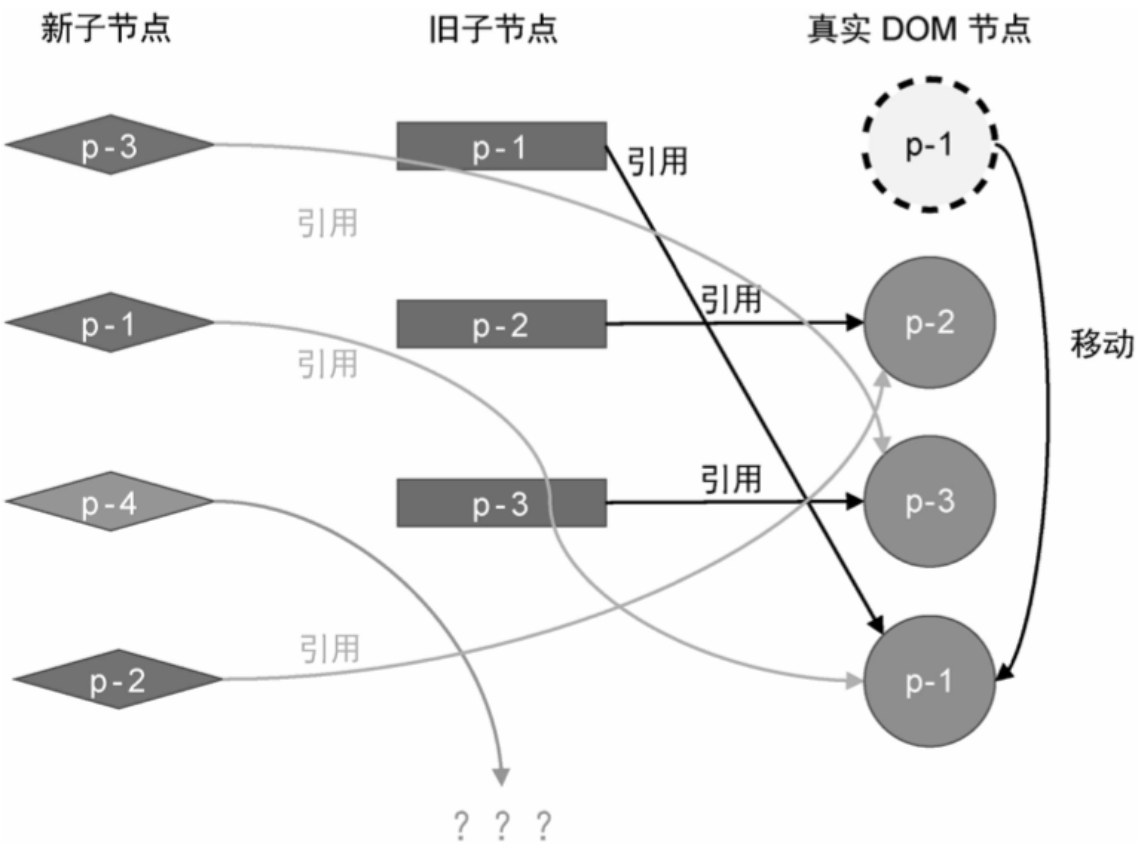
- 第一步：取新的一组子节点中第一个节点 p-3，它的 key 为 3，尝试在旧的一组子节点中找到具有相同 key 值的可复用节点。发现能够找到，并且该节点在旧的一组子节点中的索引为 2。此时变量 lastIndex 的值为0，索引 2 不小于 0，所以节点 p-3 对应的真实 DOM 不需要移动，但需要更新变量 lastIndex 的值为 2。
- 第二步：取新的一组子节点中第二个节点 p-1，它的 key 为 1，尝试在旧的一组子节点中找到具有相同key值的可复用节点。发现能够找到，并且该节点在旧日的一组子节点中的素引为 0。此时变量 lastIndex 的值为 2，索引 0 小于2，所以节点 p-1 对应的真实 DOM 需要移动。
  到了这一步，我们发现，节点 p-1 对应的真实 DOM 需要移动，但应该移动到哪里呢？我们知道，新 children 的顺序其实就是更新后真实 DOM 节点应有的顺序。所以节点 p-1 在新 children 中的位置就代表了真实 DOM 更新后的位置。由于节点 p-1在新children 中排在节点p-3 后面，所以我们应该把节点 p-1 所对应的真实 DOM 移动到节点p-3所对应的真实 DOM 后面。
  可以看到，这样操作之后，此时真实 DOM 的顺序为 **p-2、p-3、p-1**。

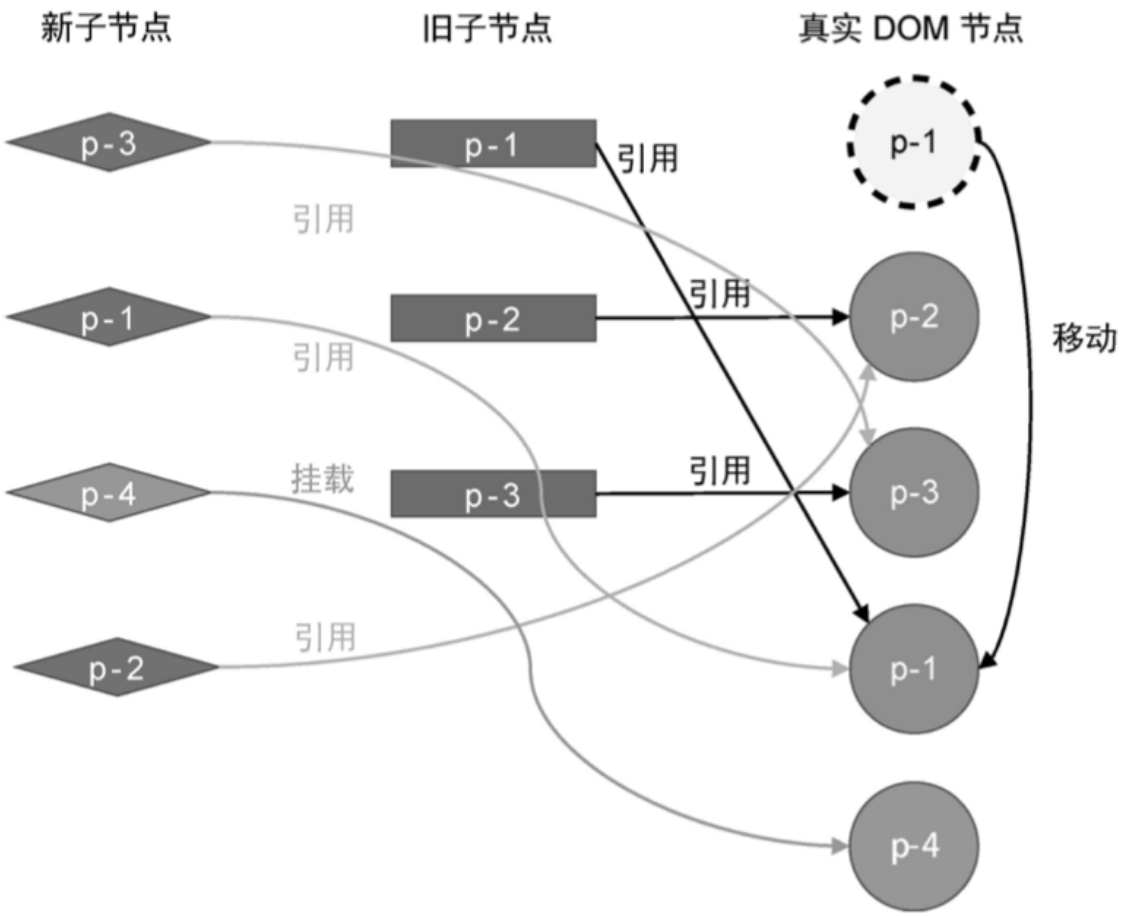


- 第三步：取新的一组子节点中第三个节点 p-2，它的 key 为 2。
  尝试在旧的一组子节点中找到具有相同 key 值的可复用节点。发现能够找到，并且该节点在旧的一组子节点中的索引为 1。此时变量 lastIndex 的值为 2，索引 1 小于 2，所以节点 p-2 对应的真实 DOM 需要移动。原理与第二步相似。


新子节点
旧子节点
真实 DOM 节点
p-3
p-1
p-2
p-1
p-2
p-3
p-1
p-2
引用
引用
引用
引用
引用
引用
移动
移动

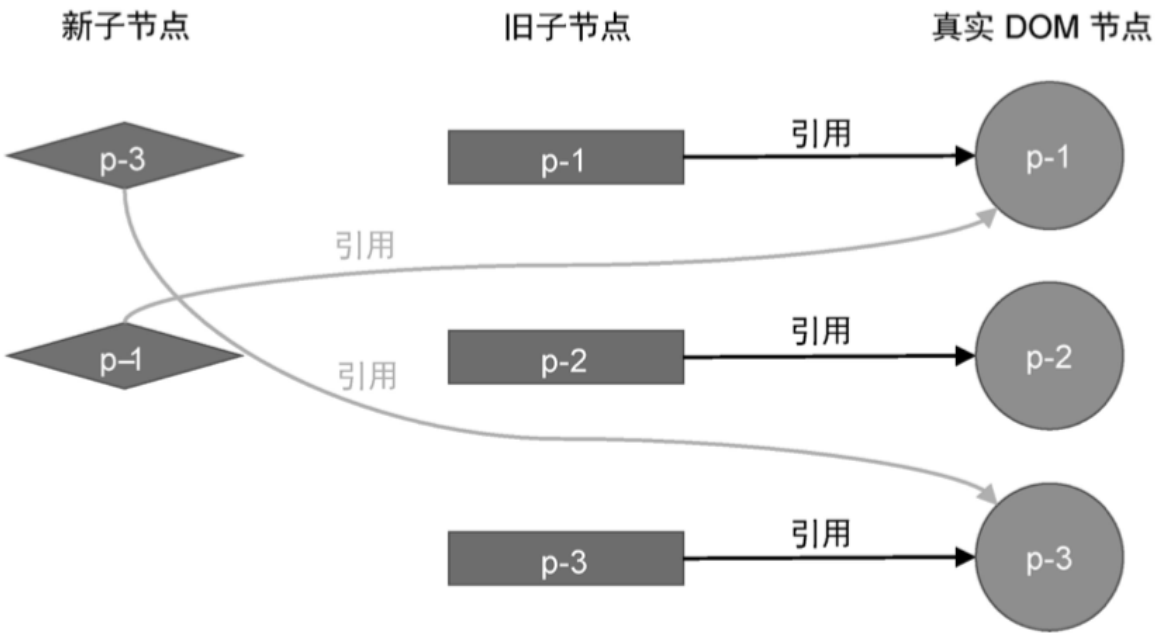
新增


新子节点
旧子节点
真实 DOM 节点
p-3
p-1
p-4
p-2
p-1
p-2
p-3
p-1
引用
引用
引用
引用
引用
移动
? ? ?



两次移动，一次新增

删除



一次移动，一次删除。

节点 p-2 对应的真实 DOM 仍然存 在，所以需要增加额外的逻辑来删除遗留节点。思路很简单，当基本 的更新结束时，我们需要遍历旧的一组子节点，然后去新的一组子节 点中寻找具有相同 key 值的节点。如果找不到，则说明应该删除该节点。

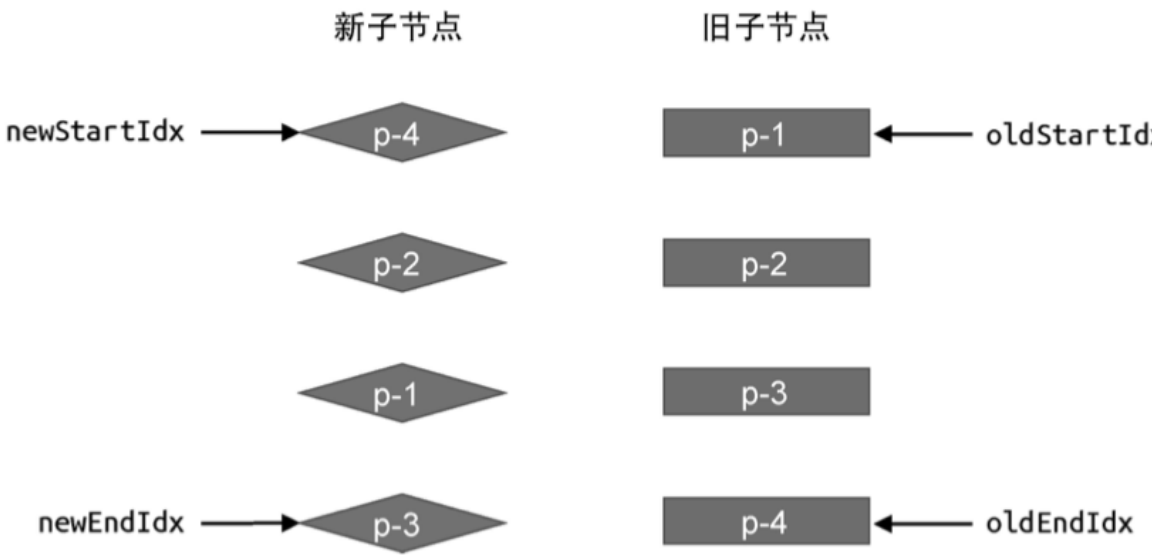
https://github.com/vuejs/vue/blob/49b6bd4264c25ea41408f066a1835f38bf6fe9f1/src/core/vdom/patch.ts#L413

简单 Diff 算法的核心逻辑是，拿新的一组子节点中的节点去旧的一组 子节点中寻找可复用的节点。如果找到了，则记录该节点的位置索 引。我们把这个位置索引称为最大索引。在整个更新过程中，如果一 个节点的索引值小于最大索引，则说明该节点对应的真实 DOM 元素需要移动。

最后我们整体介绍了渲染器如何移动、添加、删除 虚拟节点所对应的 DOM 元素。

双端 diff（）

双端 diff 顾名思义，对比的逻辑分别从两端收敛进行。双端 Diff 算法是一种同时对新旧两组子节点的两个端 点进行比较的算法。因此，我们需要四个索引值，分别指向新旧两组 子节点的端点。



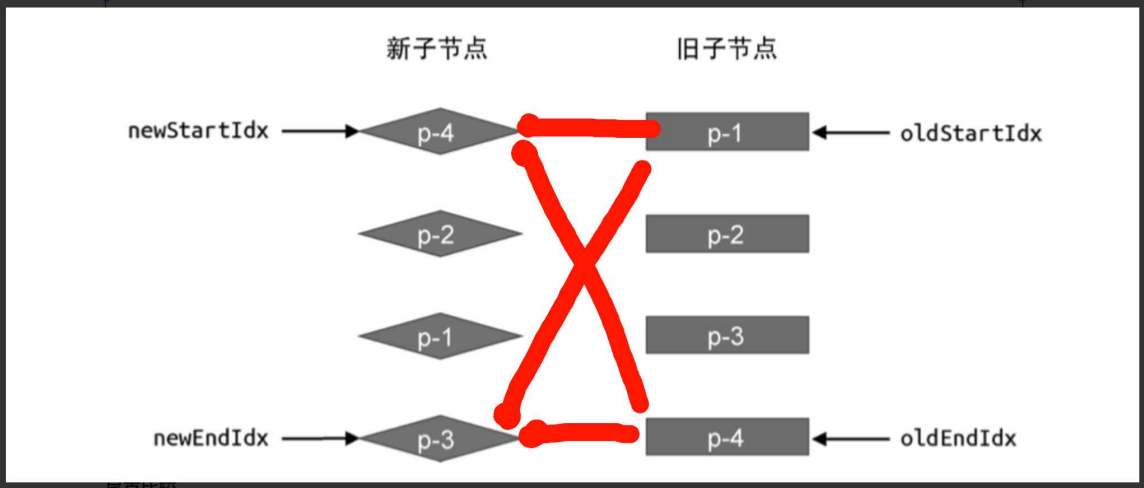
首首比较

尾尾比较

尾首比较

首尾比较

双端 diff 顾名思义，对比的逻辑分别从两端收敛进行。双端 Diff 算法是一种同时对新旧两组子节点的两个端点进行比较的算法。因此，我们需要四个索引值，分别指向新旧两组 子节点的端点。

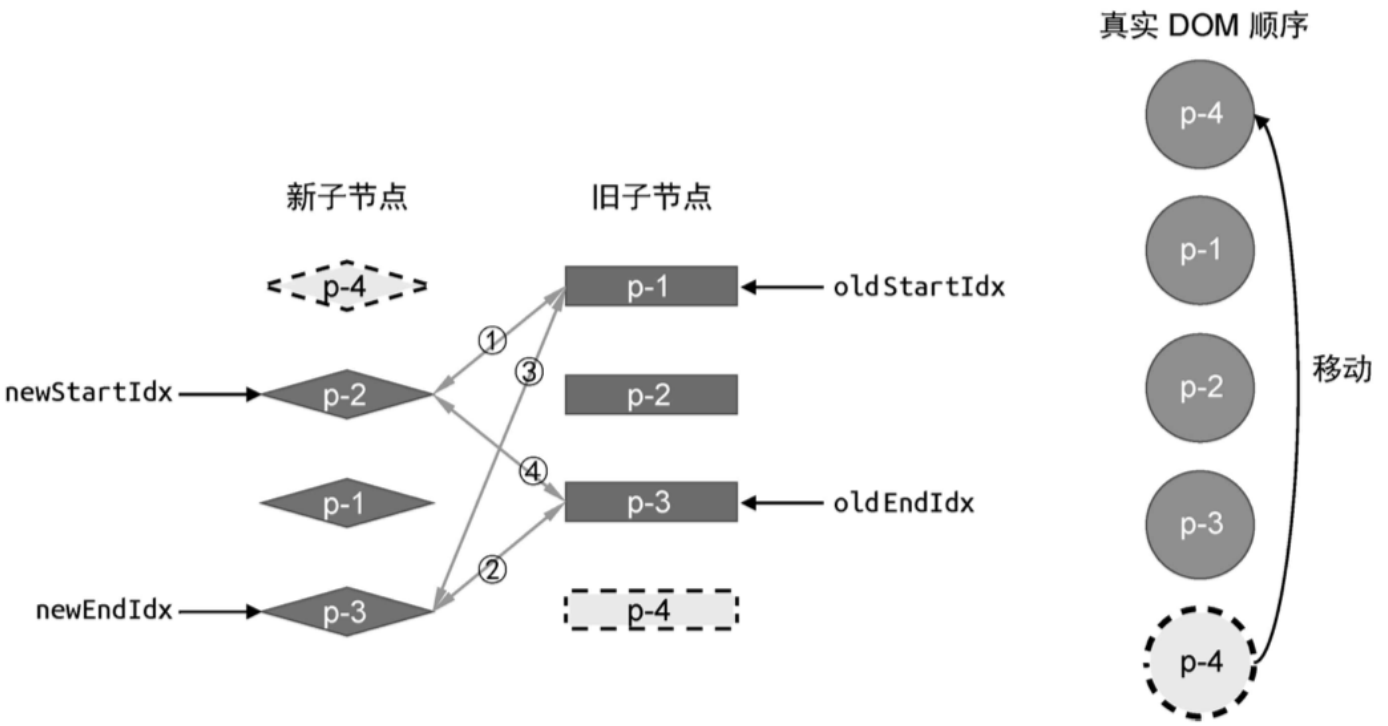


尾首比较

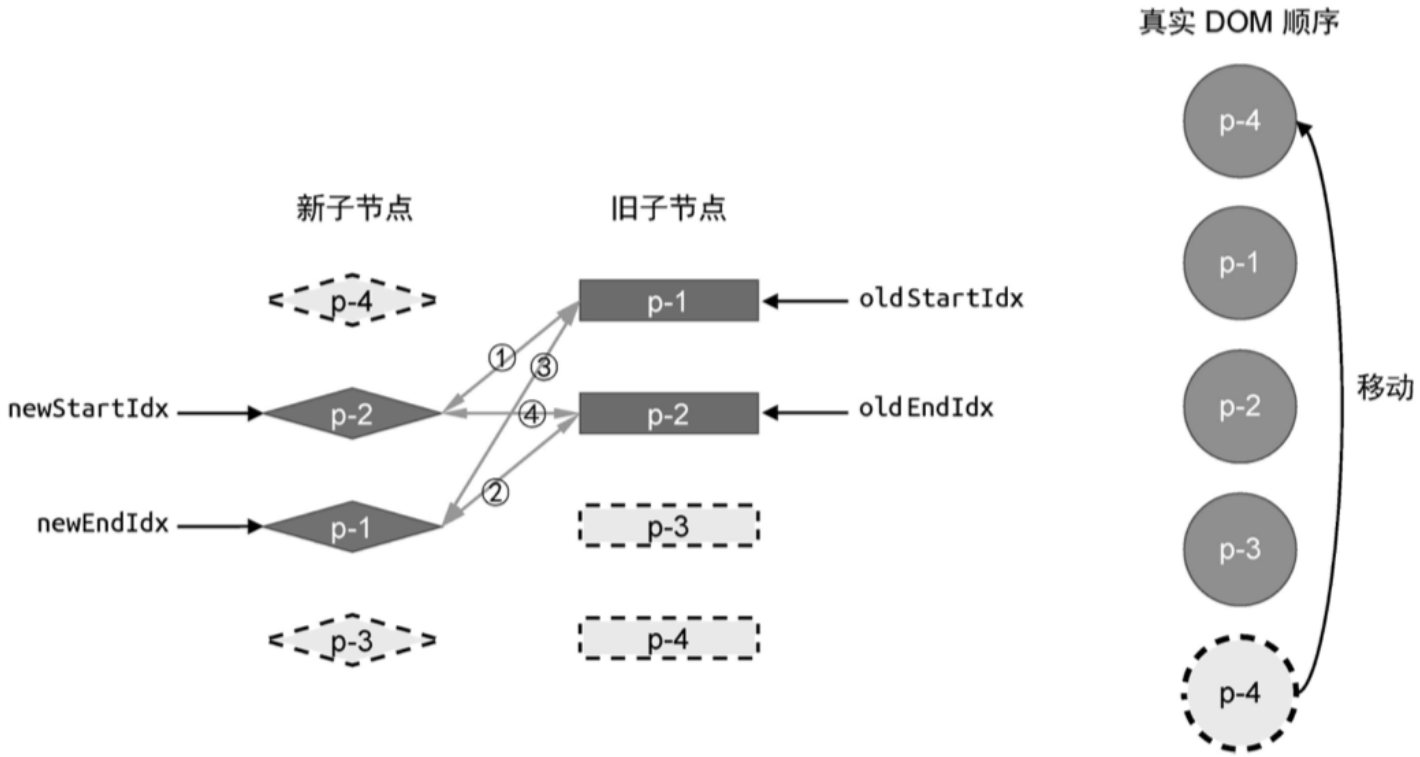
首尾比较

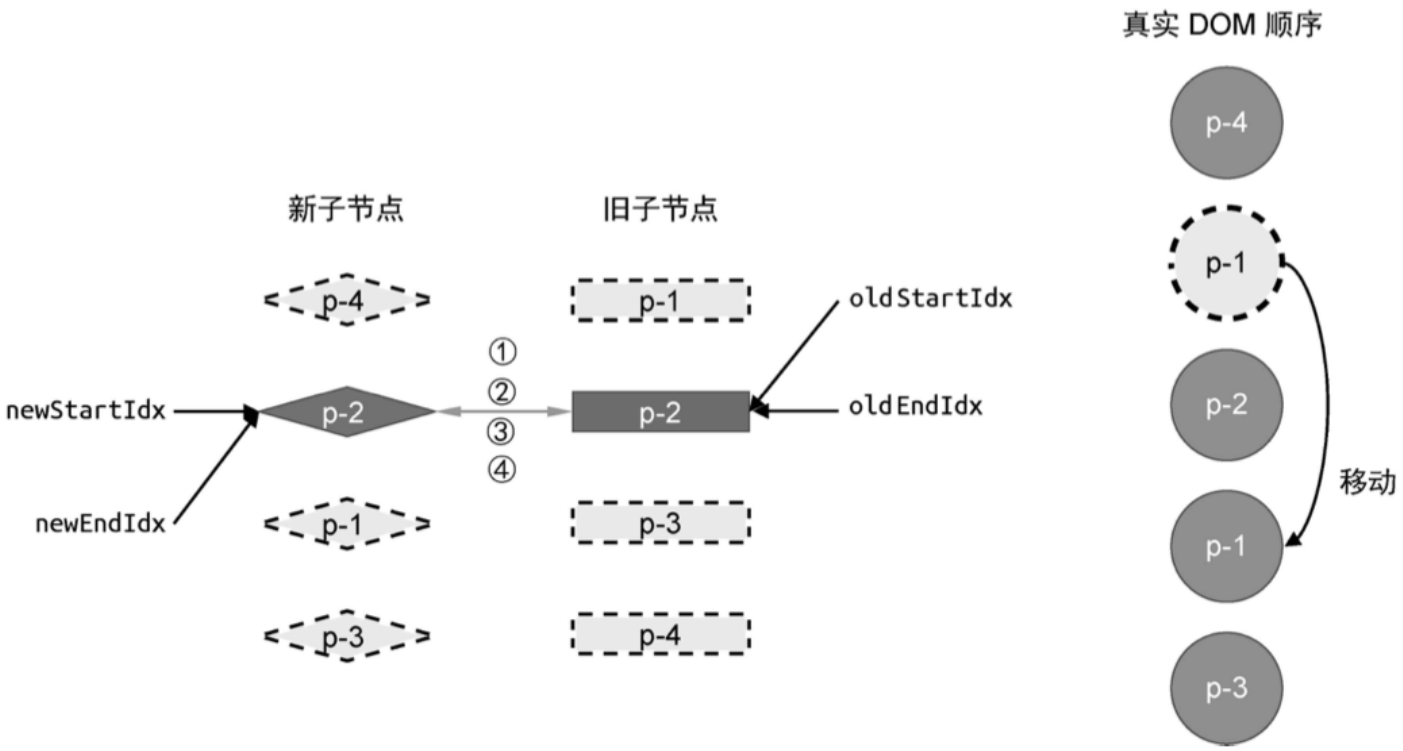
详述一下执行步骤：

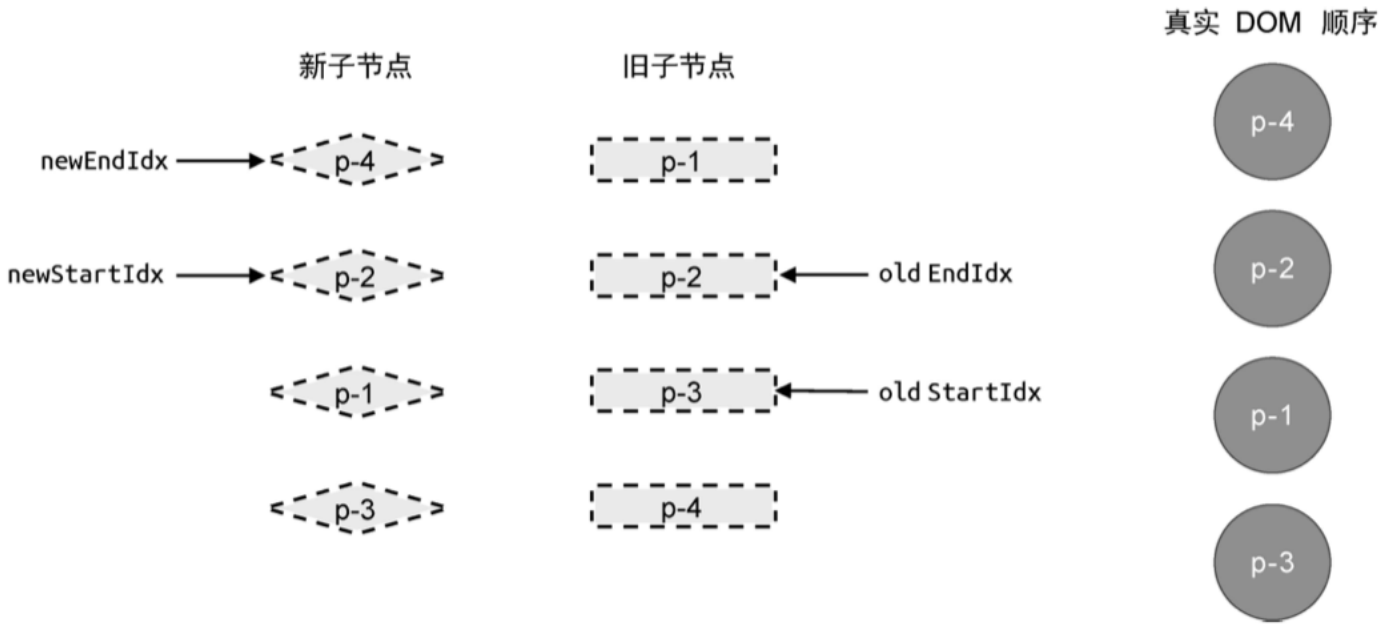
- 第一步：比较旧的一组子节点中的第一个子节点 p-1 与新的一组子节点中的第一个子节点 p-4，看看它们是否相同。由于两者的 key 值不同，因此不相同，不可复用，于是什么都不做。
- 第二步：比较旧的一组子节点中的最后一个子节点 p-4 与新的一组子节点中的最后一个子节点 p-3，看看它们是否相同。由于两者的 key 值不同，因此不相同，不可复用，于是什么都不做。
- 第三步：比较旧的一组子节点中的第一个子节点 p-1 与新的一组子节点中的最后一个子节点 p-3，看看它们是否相同。由于两者的 key 值不同，因此不相同，不可复用，于是什么都不做。
- 第四步：比较旧的一组子节点中的最后一个子节点 p-4 与新的一组子节点中的第一个子节点 p-4。由于它们的 key 值相同，因此可以进行 DOM 复用。



对比完以后，接下来反复如此，直到不满足 `oldStartIdx <= oldEndIdx && newStartIdx <= newEndIdx` 这一条件

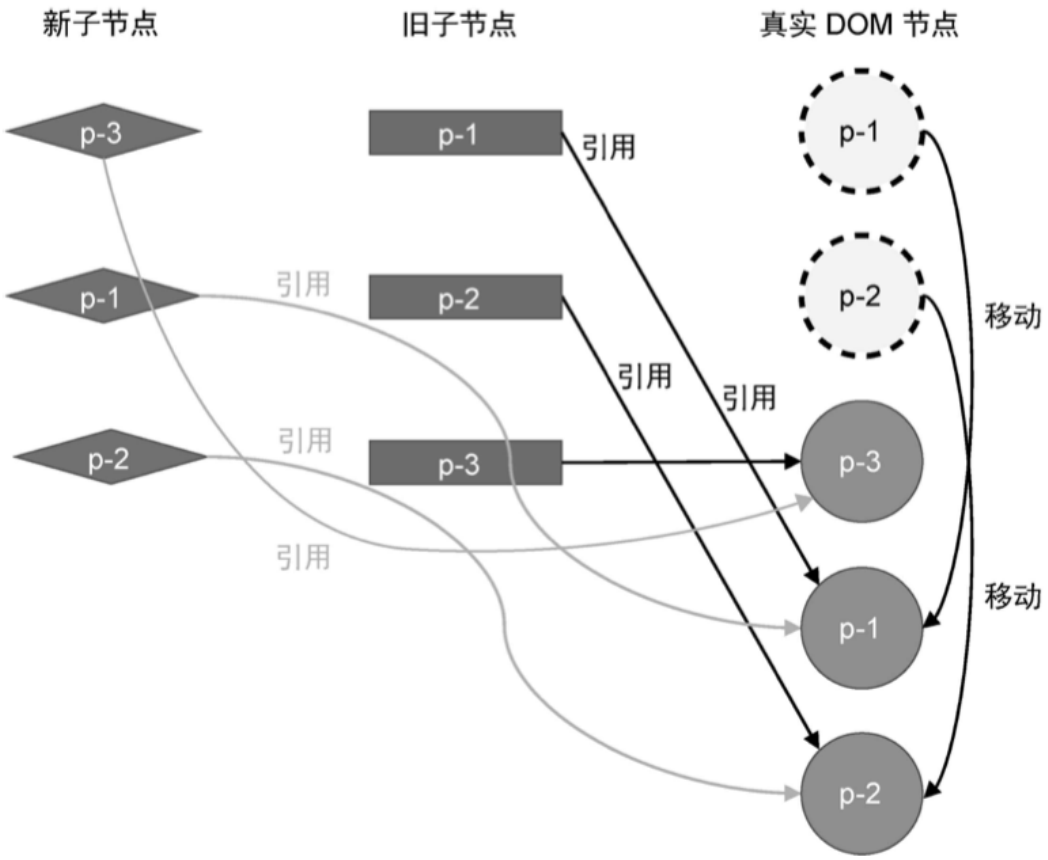






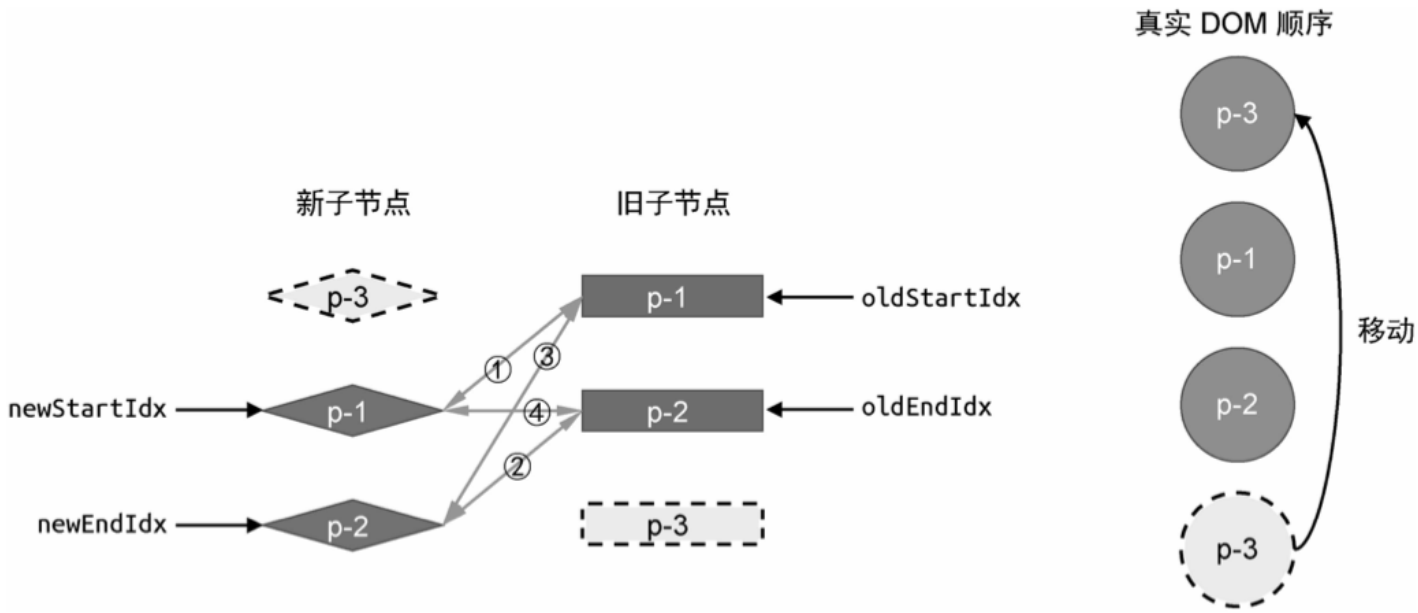
双端 diff vs 简单 diff

简单 diff



我们发现，这样的一次 diff，共需要进行两次 dom 移动。

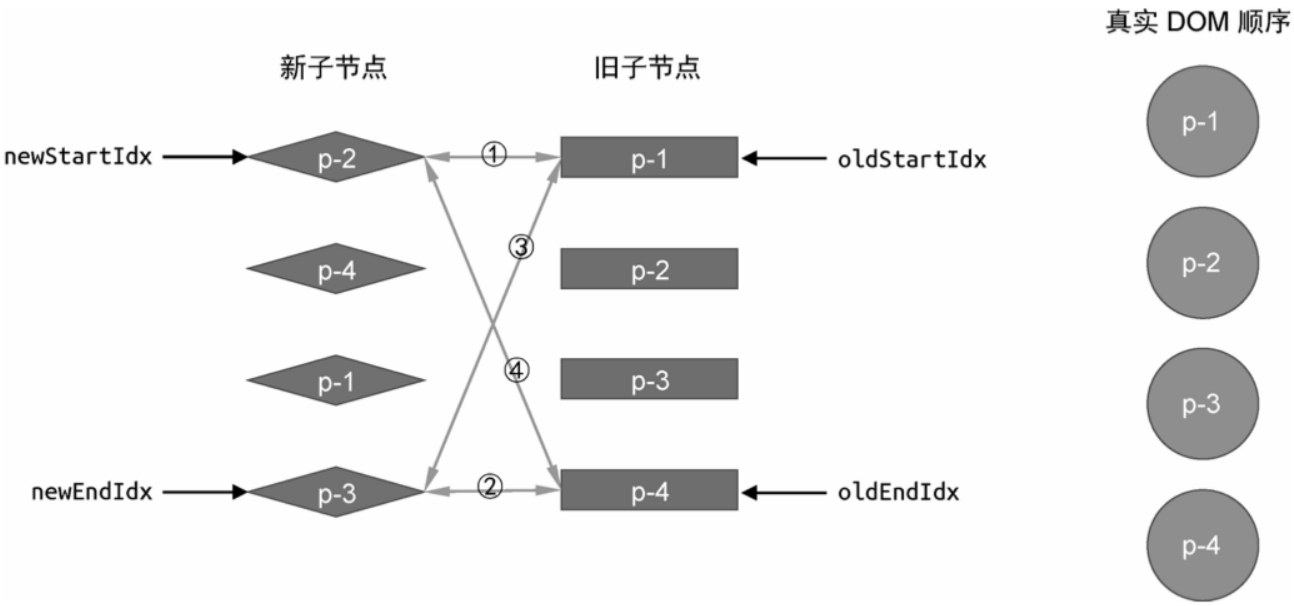
双端 diff



而双端 diff 只需要一次。

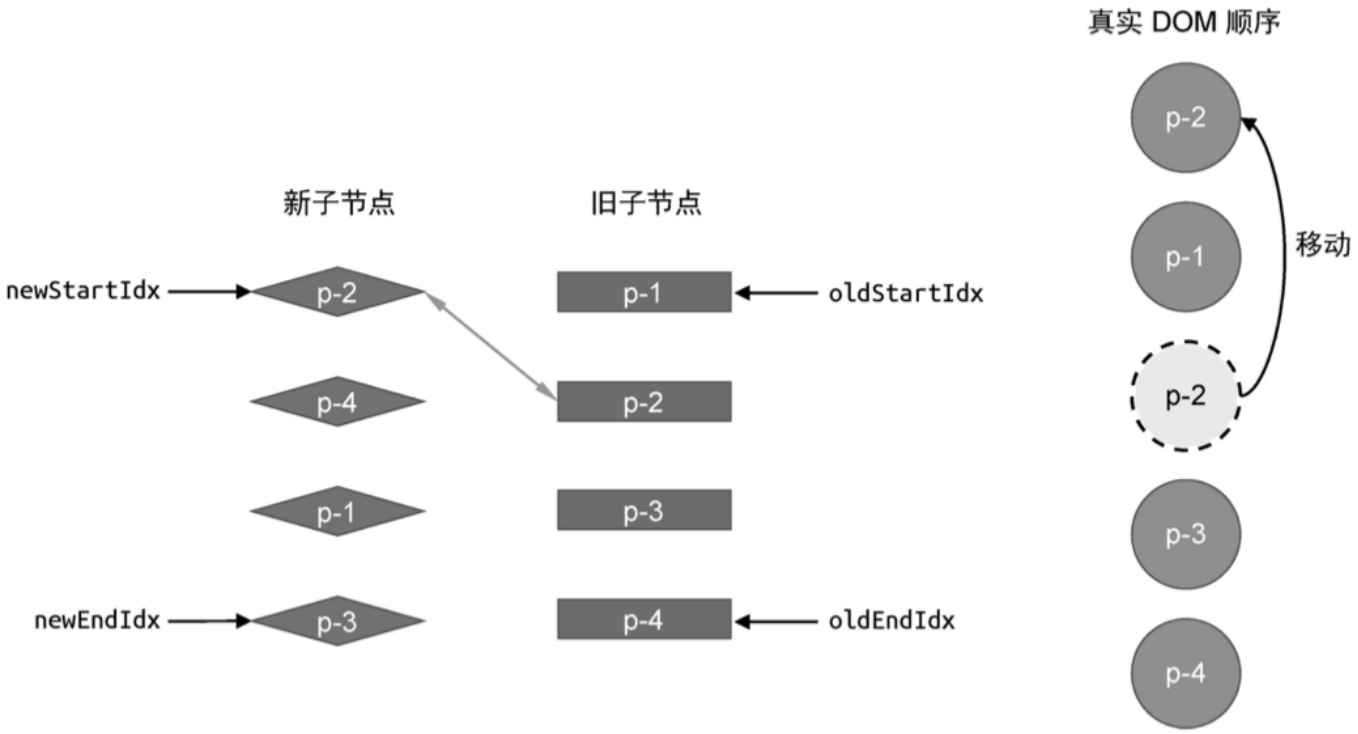
双端 diff 的非理想情况处理

大家想想，如果一轮对比发现根本就没有可以复用的内容，该怎么处理？

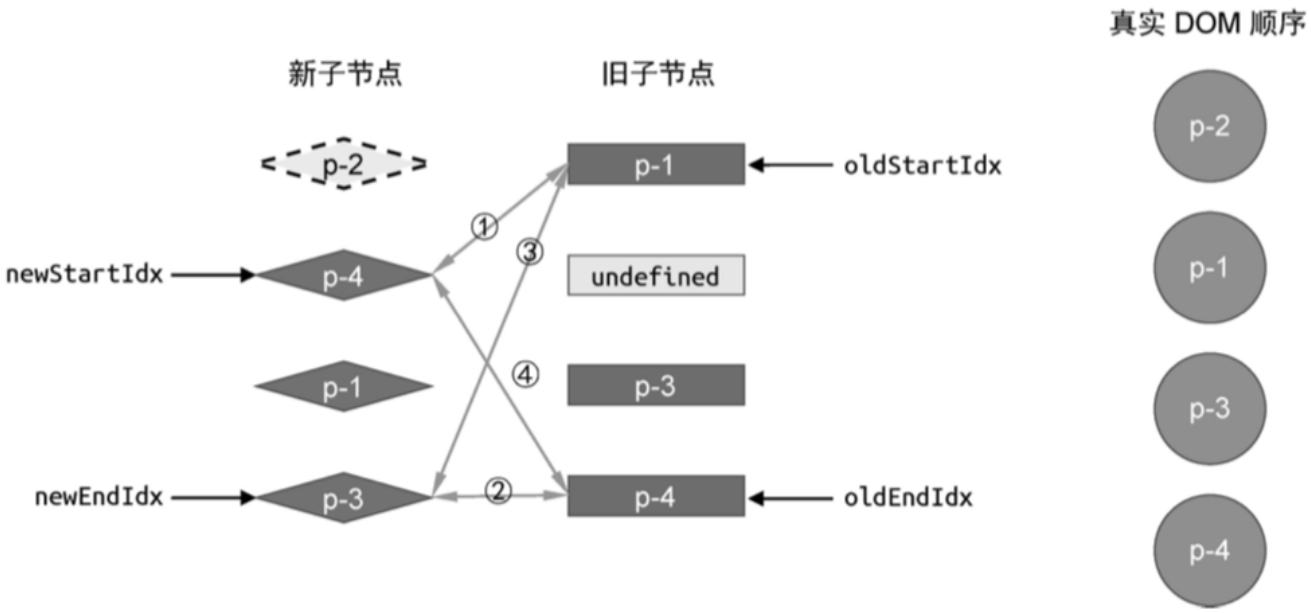


我们从上图可以发现，第一轮对比都没有命中可复用节点，怎么办呢？

我们就只能拿新子节点中第一个节点 p-2 去旧子节点中找

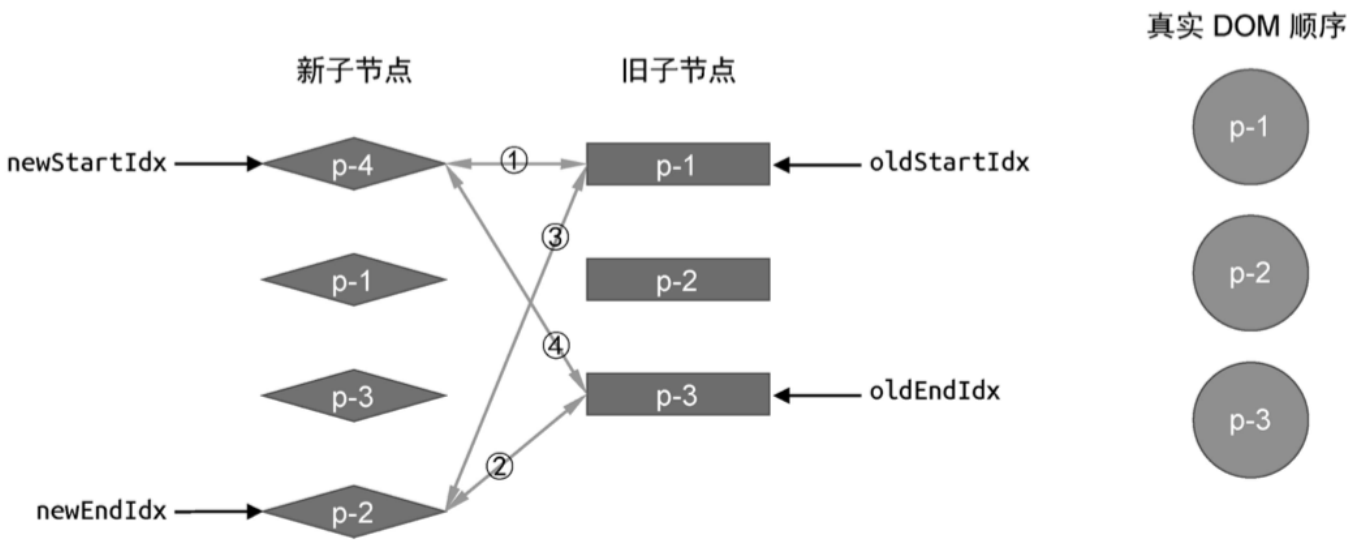


找到啦，那就移动，且将旧子节点 p-2 赋为 undefined。

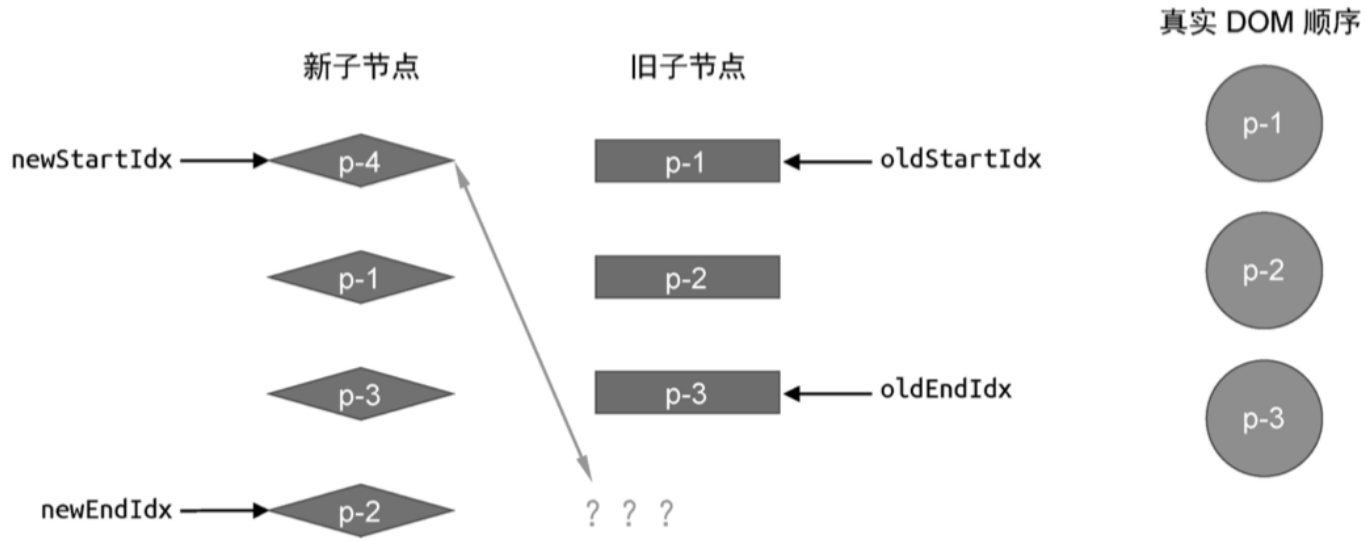


以此，完成后续比对。

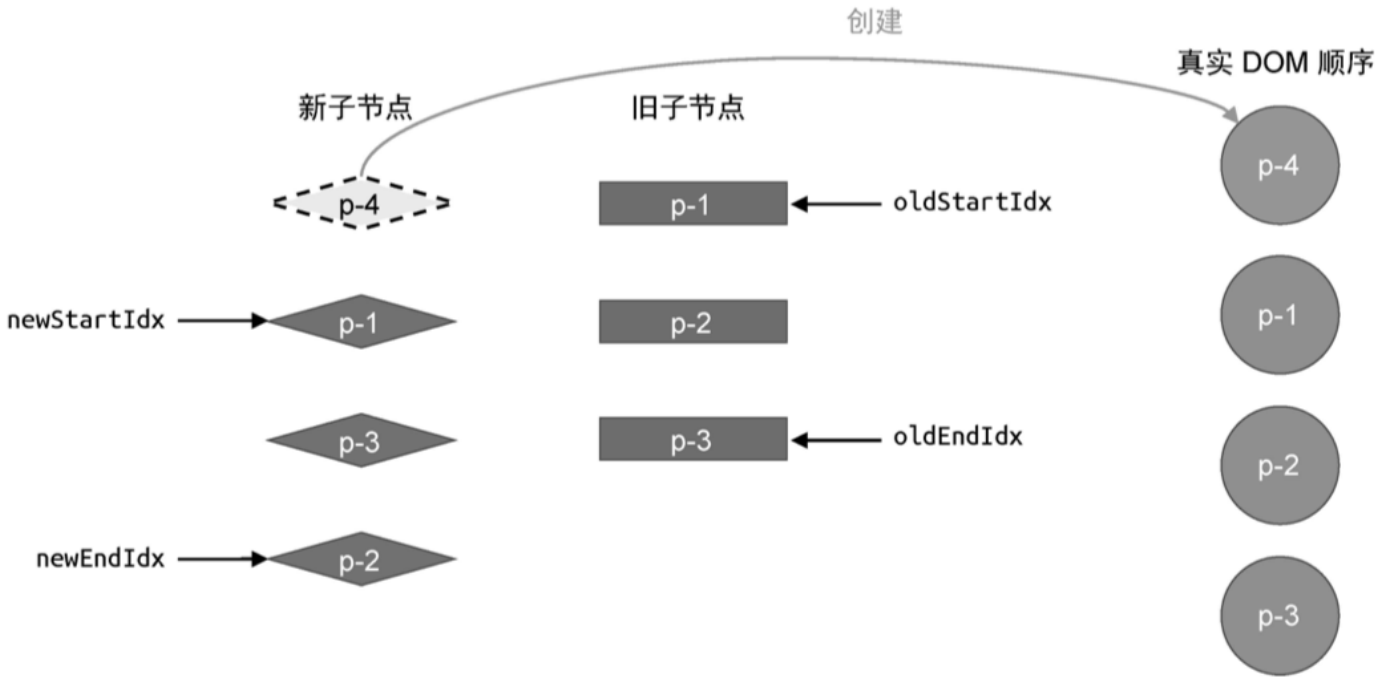
添加新元素



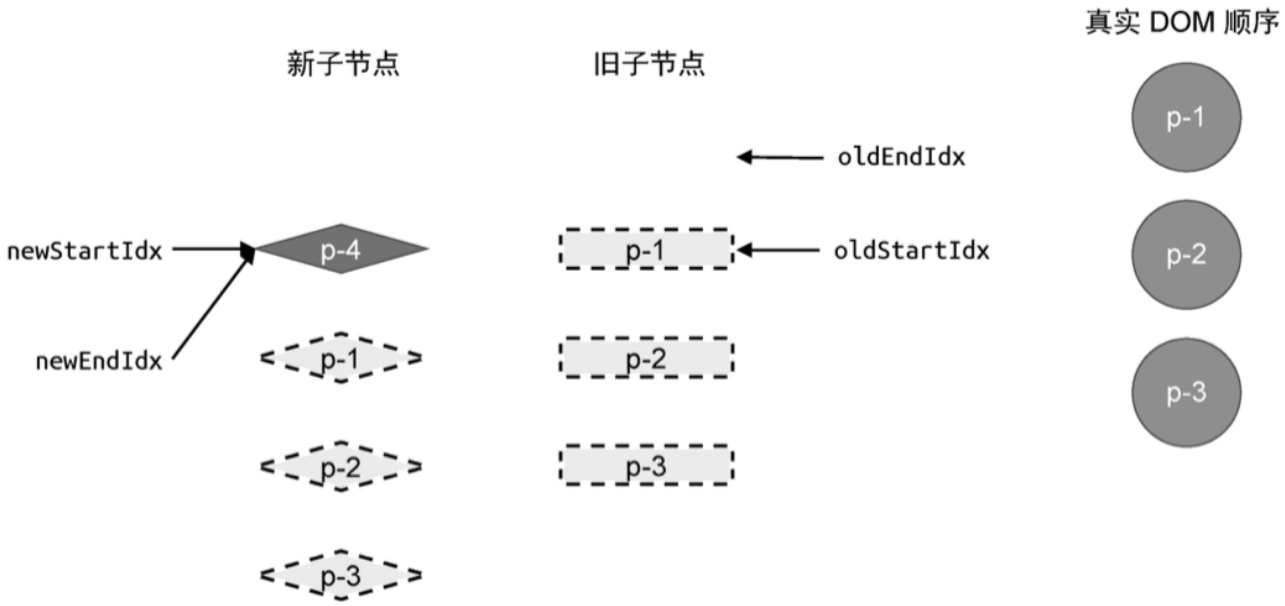
对比完，发现，糟糕，没有任何可复用的节点，那么接下来尝试拿出新节点中第一个节点去旧节点找



新子节点第一个节点 p-4 在旧子节点也没有找到，情况不妙，说明这个 p-4 是新成员，那么我们就需要创建 dom，然后 newStartIdx 后移。

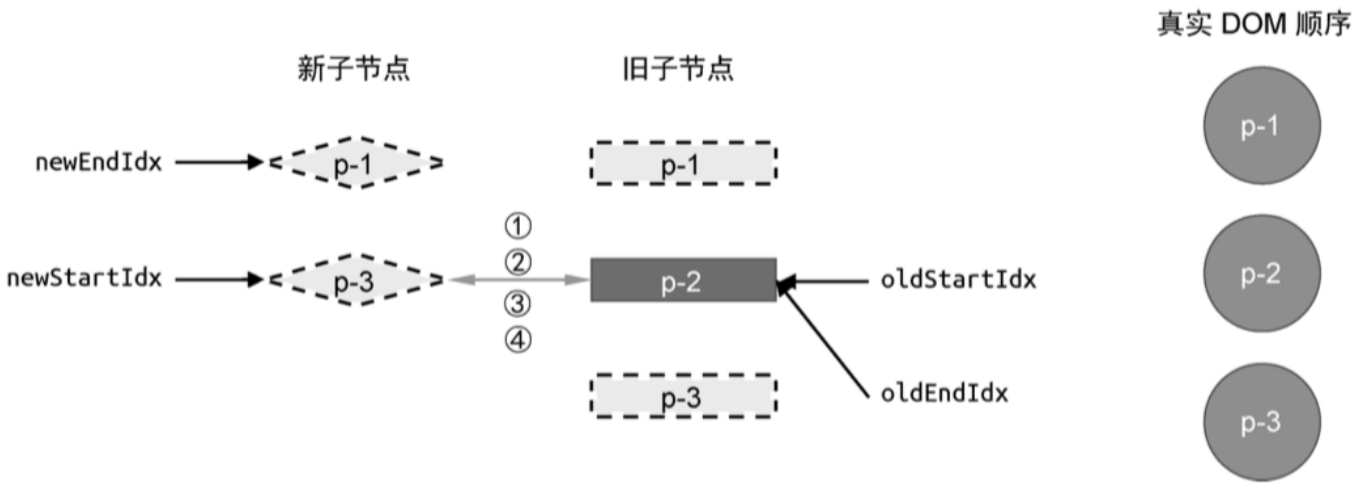


还有一种情况，就是比对时，尾部有可复用元素，直到新子节点前端发现无法复用。



删除元素

删除元素其实很简单，就是在 diff 过程完成后，如果 newEndIdx < newStartIdx 了，但是此时旧子节点的 oldStartIdx <= oldEndIdx，则需要将 oldStartIdx ~ oldEndIdx 的所有元素删除。



快速 diff（重点关注）

快速 diff 这一算法借鉴了 ivi 或者 inferno 这两个库的实现。
https://github.com/infernojs/inferno/blob/a59d1fbe296b9468823a59f34f6591d8591d2759/packages/inferno/src/DOM/patching.ts#L1223

最长递增子序列

这是快速 diff 算法的核心，我们非常有必要将其单独抽取出来进行讲解。

对应 LeetCode，https://leetcode.cn/problems/longest-increasing-subsequence/description/

比如我现在有一个数组，[4,6,7]，那么它的递增子序列有以下几个：

- [4,6]
- [4,6,7]
- [4,7]
- [6,7]
- [7]

那么可以看出，最长递增子序列是 [4,6,7]

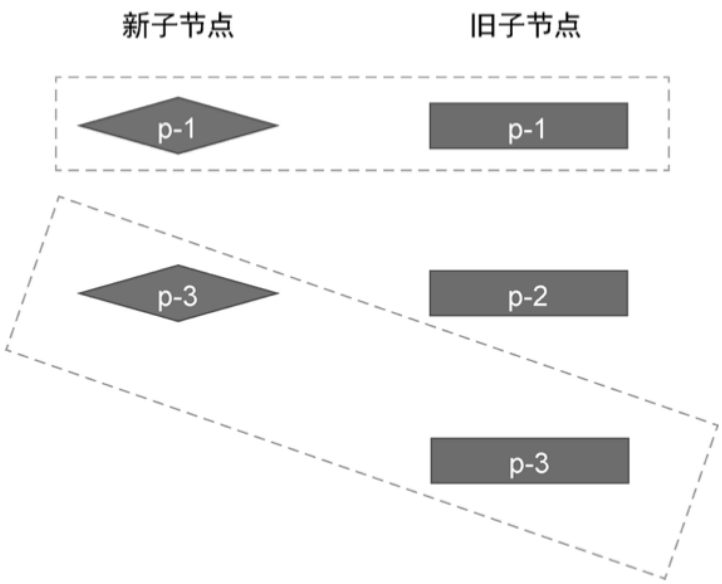
预处理

Vue3 在 diff 时会预先进行优化处理，怎么做呢？我们可以看看如下示例：

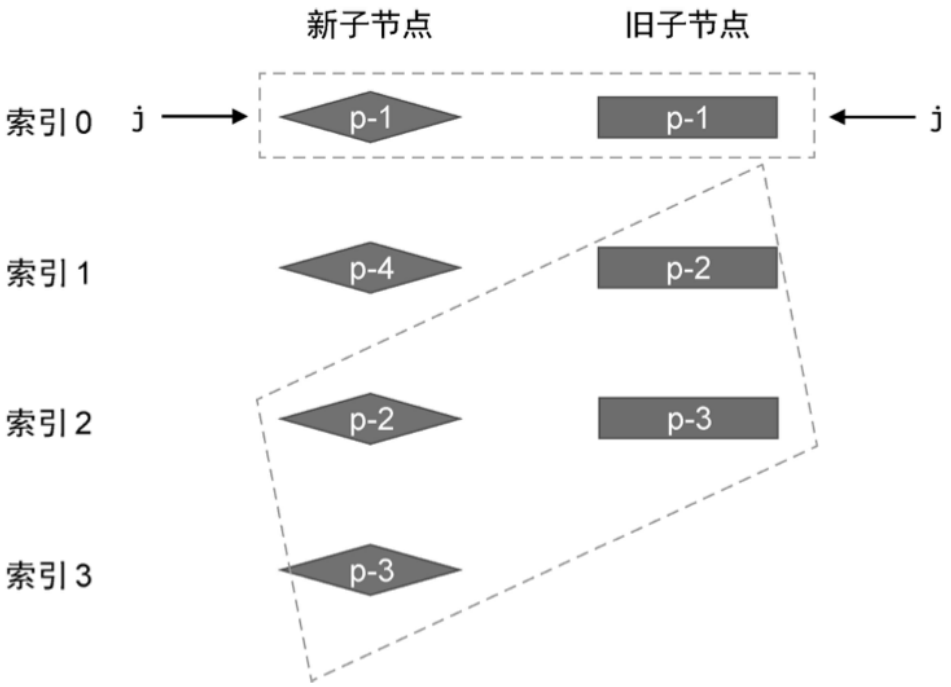
```
const text1 = 'Hello World'
const text2 = 'Hello'
```

那其实，我们真正需要 diff 的只有 'World'，为什么，因为字符串前后我们可以先剔除掉相同子串。

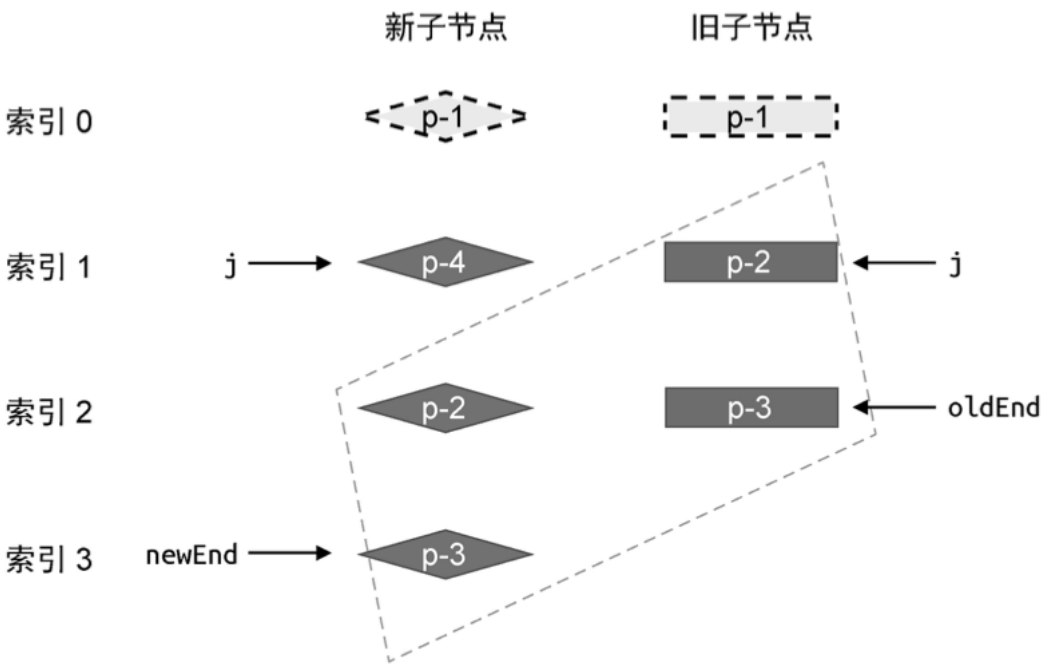
那在 Vue diff 时，也一样



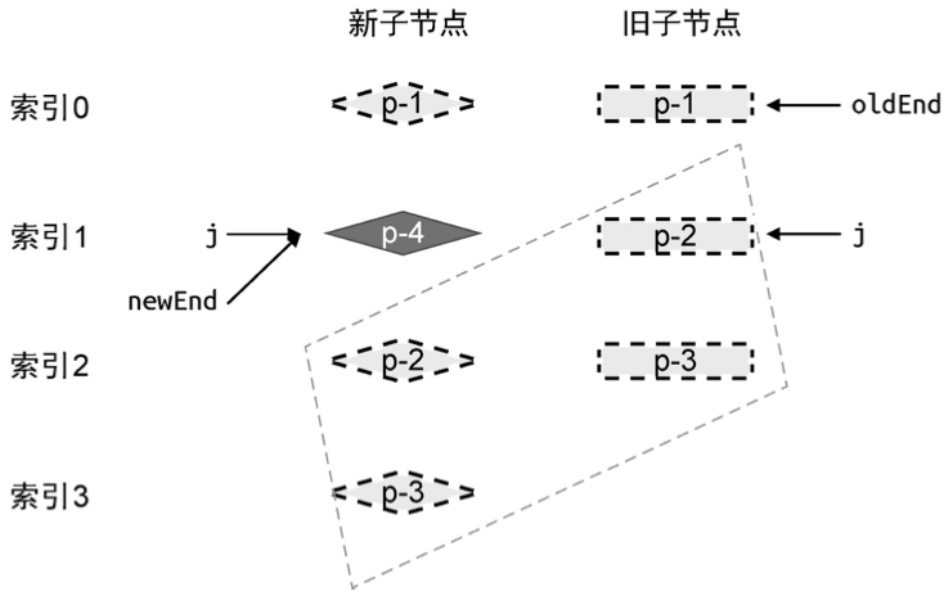
我们先将首尾相同节点 diff 并在发现相同元素时进行 patch 操作。



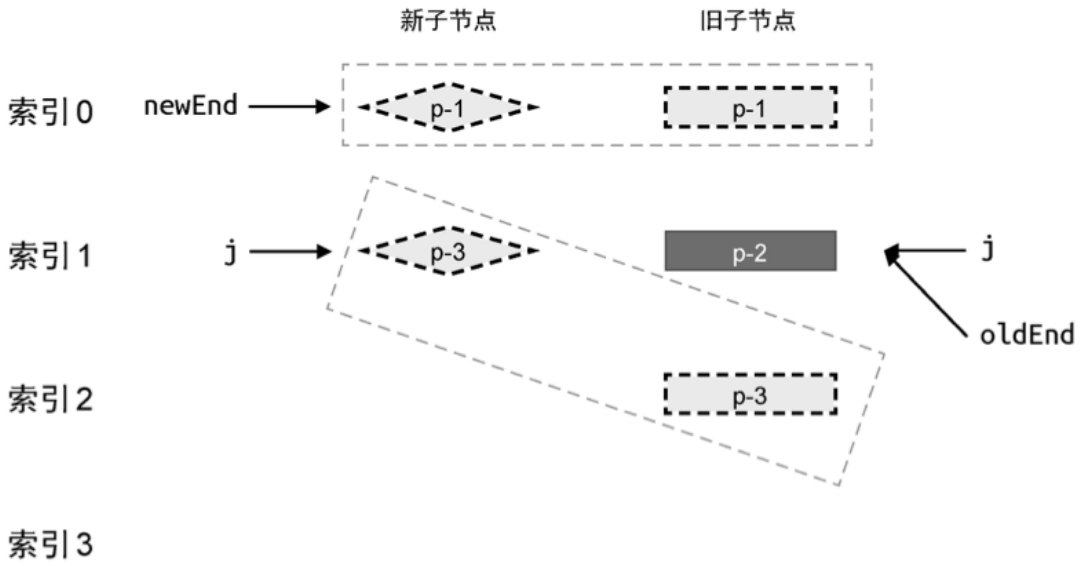
先定义一个索引 j，逐步递增，对比新旧子节点如果相同则进行 patch 操作，直至不相同，当不相同时，我们转变处理方法，从尾部开始进行对比，如下：



很显然，这里执行完以后，只剩下 p-4，所以这个作为新节点进行挂载，需要创建新 dom。



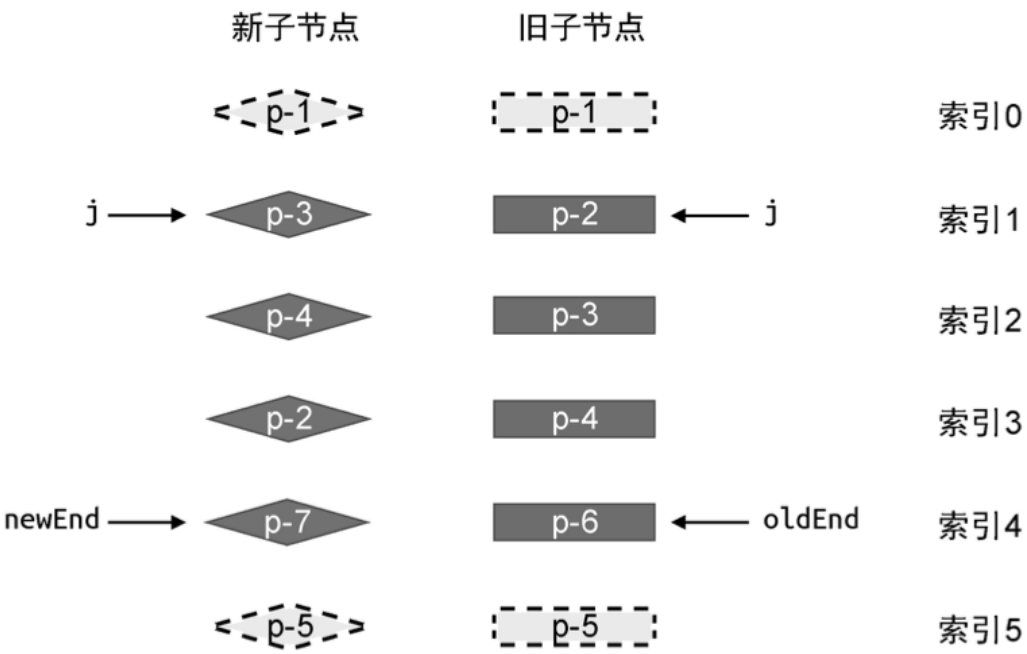
还存在另外一种情况，就是最终剩下旧子节点，那么旧子节点对应 dom 会被移除。如图：



总结下来就是：

- 遍历完后，如果新子节点 newEnd > j，则 newEnd~j 的子元素全部作为新元素挂载，需创建 dom
- 遍历完后，如果旧子节点 j > oldEnd，则 j~oldEnd 的子元素全部移除，需删除 dom

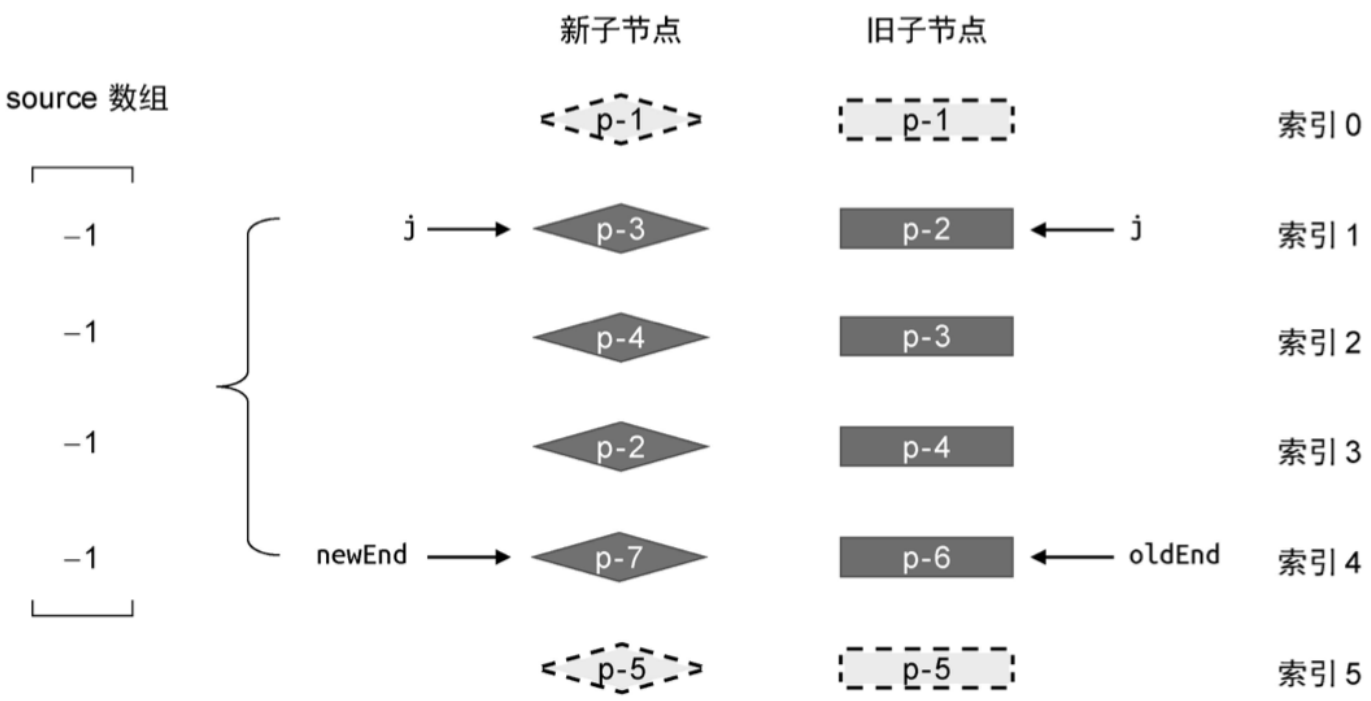
DOM 移动



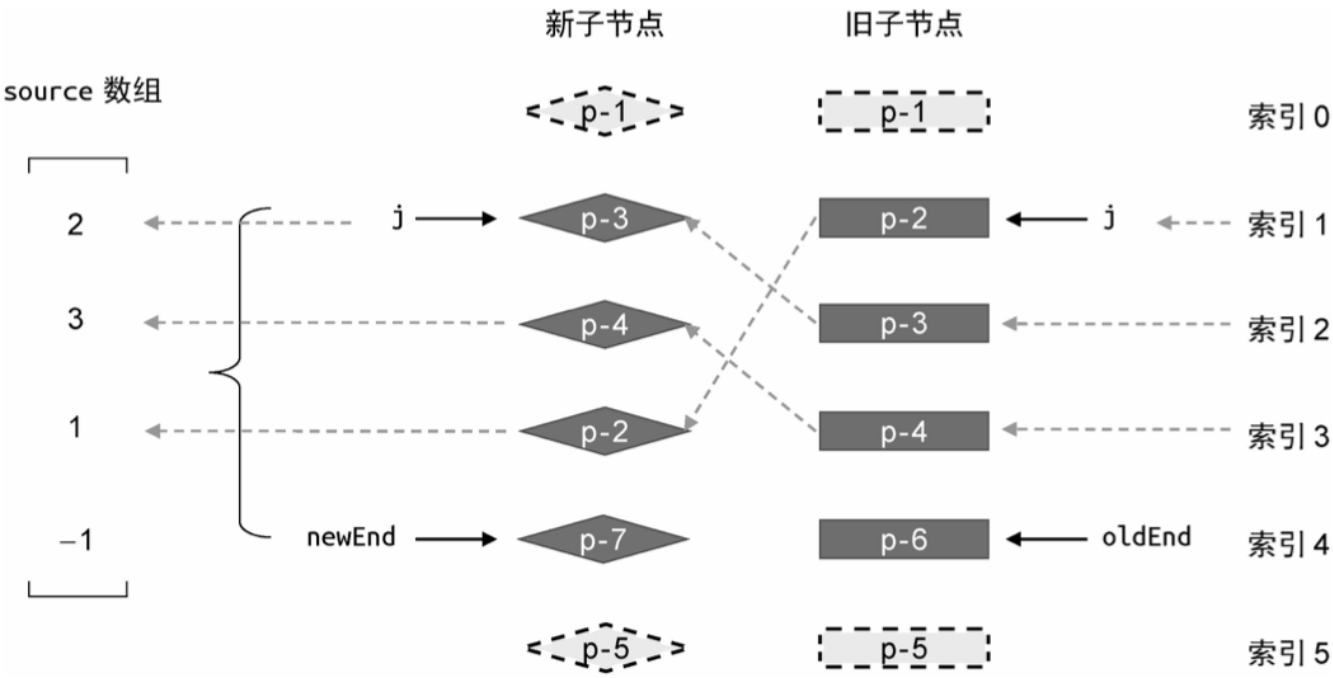
接下来构造 source 数组，这就是为我们后面计算**最长递增子序列**做准备的。

初始化的逻辑简化代码如下：

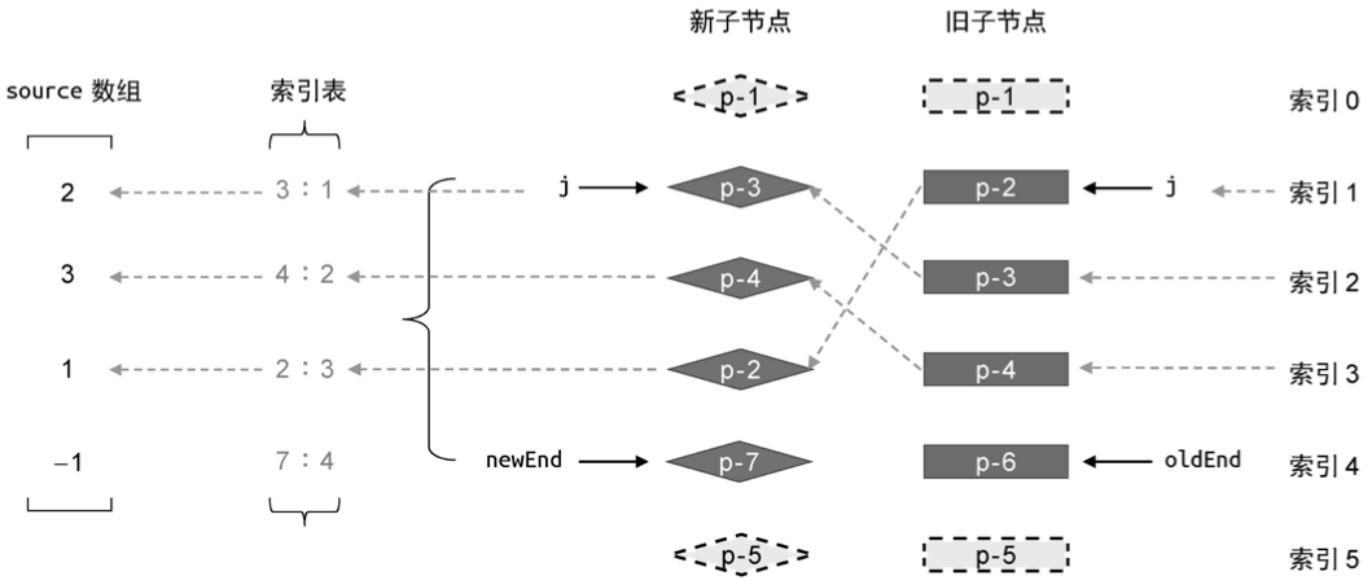
```
const count = newEnd - j + 1;
const source = new Array(count);
source.fill(-1)

// 3.3.4
// core-main/packages/runtime-core/src/renderer.ts 1905 行
const newIndexToOldIndexMap = new Array(toBePatched)
for (i = 0; i < toBePatched; i++) newIndexToOldIndexMap[i] = 0
```



然后将每一位的值设置为该节点在旧子节点中的索引，填充后，source 数组更新为：`[2,3,1,-1]`，其中 -1 表示在旧子节点中没有对应节点。



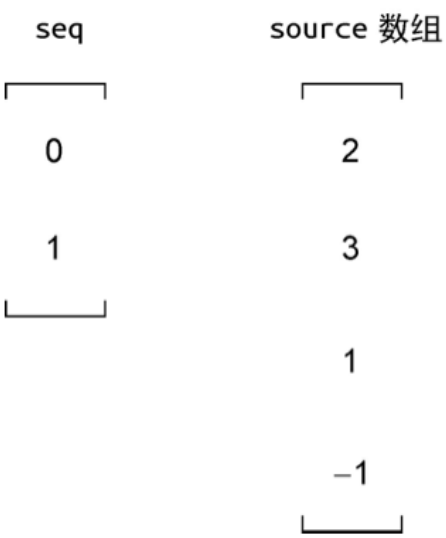
接下来为了性能优化考虑，需要额外新建一张索引表，用于标志新旧子节点的对照关系，构建为：



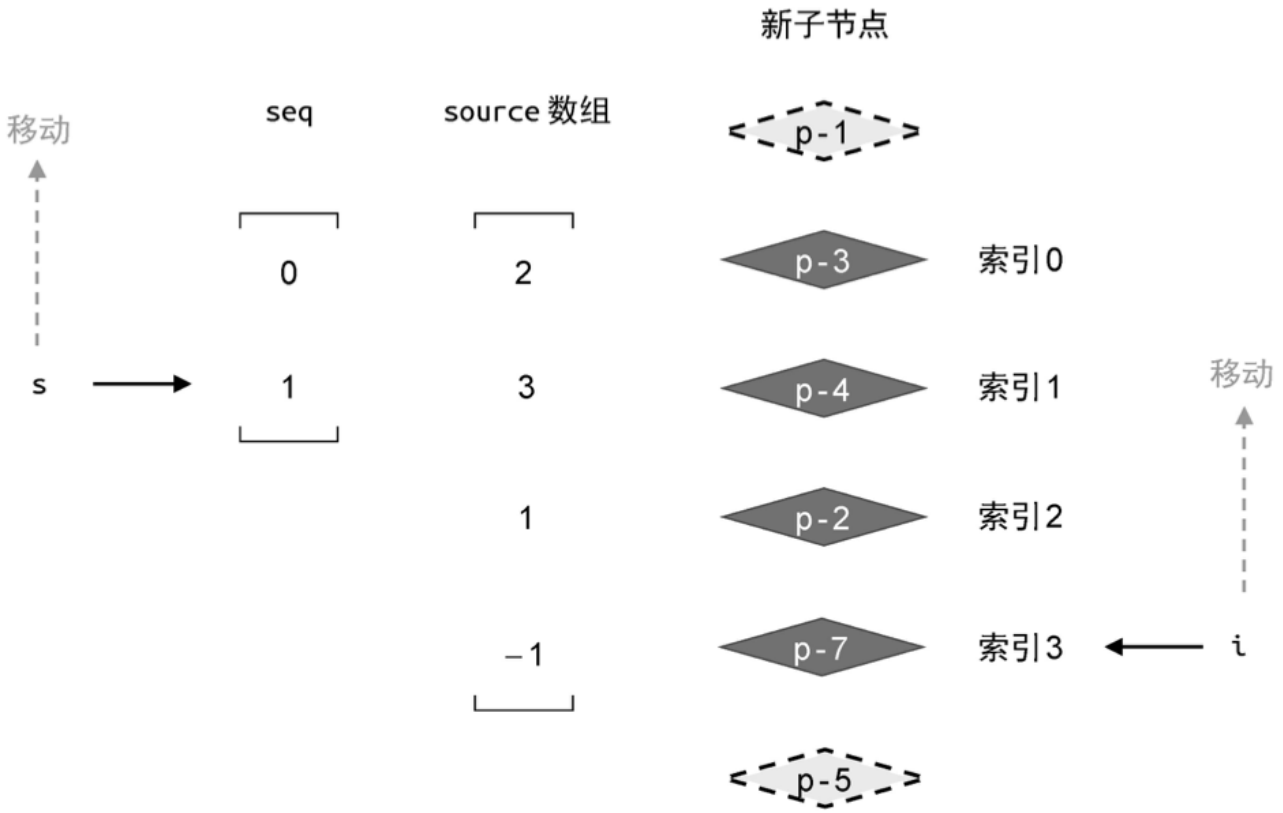
我们来看看，这个 source 数组的最长递增子序列是？考下大家 `[2, 3, 1, -1]`

答案是：`[2, 3]`

但是我们会发现，在 Vue 源码中并不是，Vue 源码计算出的结果是 `[0, 1]`，为什么呢？因为我们需要得到的是索引，因为索引才是后续我们该如何移动的关键。



接下来我们重点关注需要更新的节点序列，并重新编号



对比思路如下：**i --**

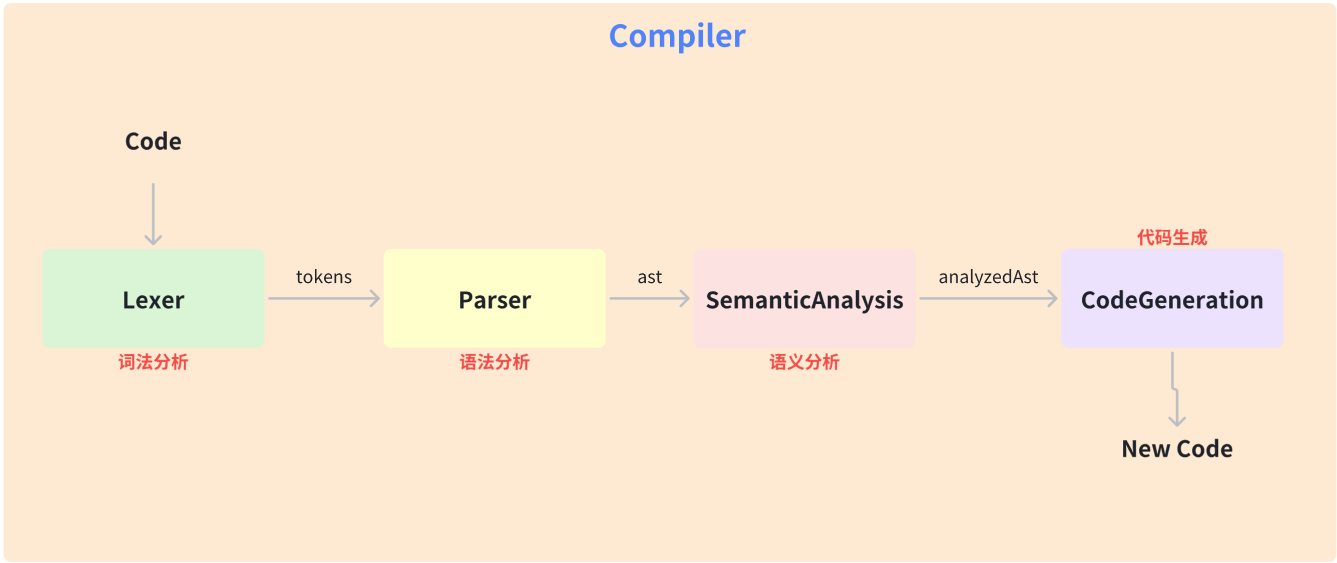
1. 首先看 source[i] 是否为 -1，如果是则表示需要**新增 dom**，否则走下一步判断逻辑
2. 再看 seq[s] 是否等于 i，如果是，则不需要处理，否则需要**移动 dom**
3. 如果 seq[s] 等于 i，则 **s--**

直至结束

Vue3 diff 优化

1. **静态标记** + 非全量 Diff：（Vue 3在创建虚拟DOM树的时候，会根据DOM中的内容会不会发生变化，添加一个静态标记。之后在与上次虚拟节点进行对比的时候，就只会对比这些带有静态标记的节点。）；
2. 使用最长递增子序列（快速 diff ）优化对比流程，可以最大程度的减少 DOM 的移动，达到最少的 DOM 操作；

6. 编译器



- Parser
- Transformer
- Generator

以上是编译器基本原理，在 Vue 中，我们需要实现模板到函数的转换，同理在 react 中，我们需要实现 jsx 到 React.createElement 的转换。

```
<div>
    <p>Vue</p>
    <p>Template</p>
</div>
```

假设我们有如上模板，很显然我们函数是无法使用它的，我们必须把它转换为函数 `h` 形式，如下：

```
function render() {
  return h('div', [
    h('p', 'Vue'),
    h('p', 'Template')
  ])
}
```

从零实现

parser

首先我们实现 parser，将模板转为初始 ast

```
// 定义状态机的状态
const State = {
  initial: 1, // 初始状态
  tagOpen: 2, // 标签开始状态
  tagName: 3, // 标签名称状态
  text: 4, // 文本状态
  tagEnd: 5, // 结束标签状态
  tagEndName: 6, // 结束标签名称状态
}

// 辅助函数，用于判断是否是字母
function isAlpha(char) {
  return char >= 'a' && char <= 'z' || char >= 'A' && char <= 'Z'
}

// 接收模板字符串作为参数，并将模板切割为 Token 返回
function tokenzie(str) {
  // 状态机的当前状态：初始状态
  let currentState = State.initial
  // 缓存字符
  const chars = []
  // 生成的 Token 会存储到 tokens 数组中，并作为函数的返回值返回
  const tokens = []

  // 使用 while 循环开启自动机
  while (str) {
    const char = str[0]

    // switch 匹配当前状态
    switch (currentState) {
      case State.initial: // 初始状态
        // 遇到字符 <
        if (char === '<') {
          // 1. 状态机切换到标签开启状态
          currentState = State.tagOpen
          // 2. 消费字符 <
          str = str.slice(1)
        } else if (isAlpha(char)) {
          // 1. 遇到字母，切换到文本状态
          currentState = State.text
          // 2. 将当前字母缓存到 chars 数组
          chars.push(char)
```

```
      // 3. 消费当前字符
      str = str.slice(1)
    }
    break;
  case State.tagOpen: // 标签开始状态
    if (isAlpha(char)) {
      // 1. 遇到字母，切换到标签名称状态
      currentState = State.tagName
      // 2. 将当前字符缓存到 chars 数组
      chars.push(char)
      // 3. 消费当前字符
      str = str.slice(1)
    } else if (char === '/') {
      // 1. 遇到字符 /，切换到结束标签状态
      currentState = State.tagEnd
      // 2. 消费字符 /
      str = str.slice(1)
    }
    break;
  case State.tagName: // 标签名称状态
    if (isAlpha(char)) {
      // 1. 遇到字母，保持状态不变，缓存当前字符到 chars 数组
      chars.push(char)
      // 2. 消费当前字符
      str = str.slice(1)
    } else if (char === '>') {
      // 1. 遇到字符 >，切换到初始状态
      currentState = State.initial
      // 2. 创建一个标签 Token，并添加到 tokens 数组中
      // tip: cahrs 数组中缓存的字符就是标签名称
      tokens.push({
        type: 'tag',
        name: chars.join('')
      })
      // 3. 清空已消费的 chars 数组
      chars.length = 0
      // 4. 消费当前字符 >
      str = str.slice(1)
    }
    break;
  case State.text: // 文本状态
    if (isAlpha(char)) {
      // 1. 遇到字母，保持状态不变，缓存当前字符到 chars 数组
      chars.push(char)
      // 2. 消费当前字符
      str = str.slice(1)
    } else if (char === '<') {
```

```
            // 1. 遇到字符 < ，切换到标签开始状态
            currentState = State.tagOpen
            // 2. 从文本状态 --> 标签开始状态，此时应该创建文本 Token，并添加到 Token 数
    组中
            // tip: cahrs 数组中缓存的字符就是文本内容
            tokens.push({
              type: 'text',
              content: chars.join('')
            })
            // 3. 清空已消费的 chars 数组
            chars.length = 0
            // 4. 消费当前字符 <
            str = str.slice(1)
          }
          break;
        case State.tagEnd: // 标签结束状态
          if (isAlpha(char)) {
            // 1. 遇到字母，切换到结束标签名称状态
            currentState = State.tagEndName
            // 2. 将当前字符缓存到 chars 数组
            chars.push(char)
            // 3. 消费当前字符
            str = str.slice(1)
          }
          break;
        case State.tagEndName: // 结束表明名称状态
          if (isAlpha(char)) {
            // 1. 遇到字母，保持状态不变，缓存当前字符到 chars 数组
            chars.push(char)
            // 2. 消费当前字符
            str = str.slice(1)
          } else if (char === '>') {
            // 1. 遇到字母 >，切换到初始状态
            currentState = State.initial
            // 2. 从结束标签名称状态 --> 初始状态，应该保存结束标签名称 Token
            // tip: cahrs 数组中缓存的字符就是标签名称
            tokens.push({
              type: 'tagEnd',
              name: chars.join('')
            })
            // 3. 清空已消费的 chars 数组
            chars.length = 0
            // 4. 消费当前字符 >
            str = str.slice(1)
          }
          break;
      }
```

```
    }

    // 返回 tokens
    return tokens
  }

  // parse 函数接收模板作为参数
  function parse(str) {
    // 首先对模板进行标记化，得到 tokens
    const tokens = tokenzie(str)
    //  创建 Root 根节点
    const root = {
      type: 'Root',
      children: []
    }
    // 创建 elementStack 栈
    const elementStack = [root]

    // 开启 while 循环扫描 tokens
    while (tokens.length) {
      // 获取当前栈顶节点作为父节点 parent
      const parent = elementStack[elementStack.length - 1]
      // 当前扫描的 Token
      const t = tokens[0]

      switch (t.type) {
        case 'tag':
          // 如果当前 Token 是开始标签，创建 Element 类型的 AST 节点
          const elementNode = {
            type: 'Element',
            tag: t.name,
            children: []
          }
          // 将其添加到父级节点的 children 中
          parent.children.push(elementNode)
          // 将当前节点压入栈
          elementStack.push(elementNode)
          break;
        case 'text':
          // 如果当前 Token 是文本，创建 Text 类型的 AST 节点
          const textNode = {
            type: 'Text',
            content: t.content
          }
          // 将其添加到父节点的 children 中
          parent.children.push(textNode)
          break;
```

```
      case 'tagEnd':
        // 遇到结束标签，将栈顶节点弹出
        elementStack.pop()
        break;
    }

    // 消费已扫描过的 token
    tokens.shift()
  }

  // 返回 ast
  return root
}

module.exports = {
  parse
}
```

transformer

```
function dump(node, indent = 0) {
  // 节点类型
  const type = node.type
  // 节点的描述，如果是根节点，则没有描述
  // 如果是 Element 类型的节点，则使用 node.tag 作为节点的描述
  // 如果是 Text 类型的节点，则使用 ndoe.content 作为节点的描述
  const desc = node.type === 'Root'
    ? ''
    : node.type === 'Element'
      ? node.tag
      : node.content
  // 打印节点的类型和描述信息
  console.log(`${'-'.repeat(indent)}${type}: ${desc}`)
  // 递归地打印子节点
  if (node.children) {
    node.children.forEach(n => dump(n, indent + 2))
  }
}

// 用来创建 StringLiteral 节点
function createStringLiteral(value) {
  return {
    type: 'StringLiteral',
    value
  }
```

```
}
// 用来创建 Identifier
function createIdentifier(name) {
  return {
    type: 'Identifier',
    name
  }
}
// 用来创建 ArrayExpression 节点
function createArrayExpression(elements) {
  return {
    type: 'ArrayExpression',
    elements
  }
}
// 用来创建 CallExpression 节点
function createCallExpression(callee, arguments) {
  return {
    type: 'CallExpression',
    callee: createIdentifier(callee),
    arguments
  }
}

// 转换文本节点
function transformText(node) {
  // 如果不是文本节点，则什么都不做
  if (node.type !== 'Text') return
  // 文本节点对应的 JavaScript AST 节点其实就是一个字符串字面量，
  // 因此只需要使用 node.content 创建一个 StringLiteral 类型的节点即可
  // 最后将文本节点对应的 JavaScript AST 节点添加到 node.jsNode 属性下
  node.jsNode = createStringLiteral(node.content)
}
// 转换标签节点
function transformElement(node) {
  // 将转换代码编写在退出阶段的回调函数中
  // 这样可以保证该标签节点的子节点全部被处理完毕
  return () => {
    // 如果被转换的节点不是元素节点，则什么都不做
    if (node.type !== 'Element') {
      return
    }

    // 1. 创建 h 函数调用语句，
    // h 函数调用的第一个参数是标签名称，因此我们以 node.tag 来创建一个字符串字面量节点
作为第一个参数
    const callExp = createCallExpression('h', [
```

```
      createStringLiteral(node.tag)
    ])
    // 2. 处理 h 函数调用的参数
    node.children.length === 1
      // 如果当前标签节点只有一个子节点，则直接使用子节点的 jsNode 作为参数
      ? callExp.arguments.push(node.children[0].jsNode)
      // 如果当前标签节点有多个子节点，则创建一个 ArrayExpression 节点作为参数
      : callExp.arguments.push(
        // 数组的每个元素都是子节点的 jsNode
        createArrayExpression(node.children.map(c => c.jsNode))
      )
    // 3. 将当前标签节点对应的 JavaScript AST 添加到 jsNode 属性下
    node.jsNode = callExp
  }
}
// 转换 Root 根节点
function transformRoot(node) {
  // 将逻辑编写在退出阶段的回调函数中，保证子节点全部被处理完毕
  return () => {
    // 如果不是根节点，则什么都不做
    if (node.type !== 'Root') return
    // node 是根节点，根节点的第一个子节点就是模板的根节点
    // 当然，这里我们暂时不考虑模板存在多个根节点的情况
    const vnodeJSAST = node.children[0].jsNode
    // 创建 render 函数的声明语句节点，将 vnodeJSAST 作为 render 函数体的返回语句
    node.jsNode = {
      type: 'FunctionDecl',
      id: { type: 'Identifier', name: 'render' },
      params: [],
      body: [
        {
          type: 'ReturnStatement',
          return: vnodeJSAST
        }
      ]
    }
  }
}

function traverseNode(ast, context) {
  // 当前节点，ast 本身就是 Root 节点
  context.currentNode = ast
  // 增加退出阶段的回调函数数组
  const exitFns = []

  // context.nodeTransforms 是一个数组，其中每一个元素都是一个函数
  const transforms = context.nodeTransforms || []
```

```
    for (let i = 0; i < transforms.length; i++) {
      // 转换函数可以返回另外一个函数，该函数作为退出阶段的回调函数
      const onExit = transforms[i](context.currentNode, context)

      if (onExit) {
        // 将退出阶段的回调函数添加到 exitFns 数组中
        exitFns.push(onExit)
      }

      // 由于任何转换函数都可能移除当前节点，因此每个转换函数执行完毕后，
      // 都应该检查当前节点是否以经被移除，如果被移除，直接返回即可
      if (!context.currentNode) return
    }

    // 如果有子节点，则递归地调用 traverseNode 函数进行遍历
    const children = context.currentNode.children
    if (children) {
      children.forEach((cur, i) => {
        // 设置父节点
        context.parent = context.currentNode
        // 设置位置索引
        context.childIndex = i
        // 递归调用
        traverseNode(cur, context)
      })
    }

    // 节点处理的最后阶段执行缓存到 exitFns 中的回调函数
    // tip: 这里我们要反序执行
    let i = exitFns.length
    while (i--) {
      exitFns[i]()
    }
  }

  function transform(ast) {
    // 在 transform 函数内创建 context 对象
    const context = {
      // 增加 currentNode，存储当前正在转换的节点
      currentNode: null,
      // 增加 childIndex，存储当前节点在父节点的 children 中的位置索引
      childIndex: 0,
      // 增加 parent，存储当前转换节点的父节点
      parent: null,
      // 用于替换节点的函数，接收新节点作为参数
      replaceNode(node) {
        // 为了替换节点，我们需要修改 AST
```

```
      // 找到当前节点在父节点的 children 中的位置：context.childIndex
      // 然后使用新节点替换即可
      context.parent.children[context.childIndex] = node
      // 由于当前新节点已经被新节点替换掉，因此我们需要将 currentNode 更新为新节点
      context.currentNode = node
    },
    // 删除当前节点
    removeNode() {
      if (context.parent) {
        // 调用数组的 splice 方法，根据当前节点的索引删除当前节点
        context.parent.children.splice(context.childIndex, 1)
        // 将 context.currentNode 置空
        context.currentNode = null
      }
    },
    // 注册 nodeTransforms 数组
    nodeTransforms: [
      transformElement,
      transformText,
      transformRoot,
    ]
  }

  // 调用 traverseNode 完成转换
  traverseNode(ast, context)
  // 打印 AST 信息
  dump(ast)
}

module.exports = {
  transform
}
```

generator

```
function genNodeList(nodes, context) {
  const { push } = context
  for (let i = 0; i < nodes.length; i++) {
    const node = nodes[i]
    genNode(node, context)
    if (i < nodes.length - 1) {
      push(', ')
    }
  }
}
```

```
function genFunctionDecl(node, context) {
  // 从 context 对象中取出工具函数
  const { push, indent, deIndent } = context
  // node.id 是一个标识符，用来描述函数的名称，即 node.id.name
  push(`function ${node.id.name} `)
  push(`(`)
  // 调用 genNodeList 为函数的参数生成代码
  genNodeList(node.params, context)
  push(`) `)
  push(`{`)
  // 缩进
  indent()
  // 为函数体生成代码，递归地调用 genNode 函数
  node.body.forEach(n => genNode(n, context))
  // 取消缩进
  deIndent()
  push(`}`)
}

function genArrayExpression(node, context) {
  const { push } = context
  // 追加方括号
  push('[')
  // 调用 genNodeList 为数组元素生成代码
  genNodeList(node.elements, context)
  // 补全方括号
  push(']')
}

function genReturnStateMent(node, context) {
  const { push } = context
  // 追加 return 关键字和空格
  push(`return `)
  // 调用 genNode 函数递归生成返回值代码
  genNode(node.return, context)
}

function genStringLiteral(node, context) {
  const { push } = context
  // 对于字符串字面量，只需要追加与 node.value 对应的字符串即可
  push(`'${node.value}'`)
}

function genCallExpression(node, context) {
  const { push } = context
  // 取得被调用函数名称和参数列表
```

```
  const { callee, arguments: args } = node
  // 生成函数调用代码
  push(`${callee.name}(`)
  // 调用 genNodeList 生成参数代码
  genNodeList(args, context)
  // 补全括号
  push(`)`)
}

function genNode(node, context) {
  switch (node.type) {
    case 'FunctionDecl':
      genFunctionDecl(node, context)
      break;
    case 'ReturnStatement':
      genReturnStateMent(node, context)
      break;
    case 'CallExpression':
      genCallExpression(node, context)
      break;
    case 'StringLiteral':
      genStringLiteral(node, context)
      break;
    case 'ArrayExpression':
      genArrayExpression(node, context)
      break;
  }
}

function genertae(node) {
  const context = {
    // 存储最终生成的渲染函数代码
    code: '',
    // 生成代码时，通过调用 push 函数完成代码拼接
    push(code) {
      context.code += code
    },
    // 当前缩进级别，初始值为 0，即没有缩进
    currentIndent: 0,
    // 该函数用来换行，即在代码字符串的后买你追加 \n 字符
    // 另外，换行时应该保留缩进，所以我们还要追加 currentIdent * 2 个空格字符
    newLine() {
      context.code += '\n' + `  `.repeat(context.currentIndent)
    },
    // 用来缩进，即让 currentIdent 自增后，调用换行函数
    indent() {
      context.currentIndent++
```

```
      context.newLine()
    },
    // 取消缩进，即让 currentIdent 自减后，调用换行函数
    deIndent() {
      context.currentIndent--
      context.newLine()
    }
  }

  // 调用 genNode 函数完成代码生成工作
  genNode(node, context)

  // 返回渲染函数代码
  return context.code
}

module.exports = {
  genertae
}
```

使用编译器

```
const { parse } = require('./compiler/parse')
const { transform } = require('./compiler/transform')
const { generate } = require('./compiler/generate')

function compiler(template) {
  // 模板 AST
  const ast = parse(template)
  // 将模板 AST 转换为 javaScript AST
  transform(ast)
  // 代码生成
  const code = generate(ast.jsNode)
  return code
}

const targetAST = compiler('<div><p>Vue</p><p>Template</p></div>')

console.log(targetAST)
```

```
<div>
```

```
    <p>Vue</p>
    <p>Template</p>
</div>
```

假设我们有如上模板，很显然我们函数是无法使用它的，我们必须把它转换为函数 `h` 形式，如下：

```
function render() {
  return h('div', [
    h('p', 'Vue'),
    h('p', 'Template')
  ])
}
```