20. Vue3 核心模块源码解析

课程资料

课程目标

- 初级：
  - 掌握 Vue3 中响应式及 AST 的使用；
  - 掌握 最长递增子序列的使用；
- 中级：
  - 掌握 Vue3 的整体源码的核心执行流程；
  - 能够理解 Vue3 diff 算法的优势；
- 高级：
  - 深入理解 Vue3 核心源码，熟练掌握 Vue3 完整的执行逻辑；
  - 熟练掌握 Vue2 和 Vue3 diff 算法实现的异同点；

课程大纲

1. Vue3 模块源码解析；
2. Vue 3 Diff算法；

课程内容

Vue3 模块源码解析

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
```

```
95 │          shapeFlags.ts
96 │          toDisplayString.ts
```

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；

目录结构

```
1  ├─src
2  │   │  ast.ts         // ts类型定义，比如type，enum，interface等
3  │   │  codegen.ts         // 将生成的ast转换成render字符串
4  │   │  compile.ts // compile统一执行逻辑，有一个 baseCompile ，用来编译模板文件的
5  │   │  index.ts // 入口文件
6  │   │  parse.ts // 将模板字符串转换成 AST
7  │   │  runtimeHelpers.ts // 生成code的时候的定义常量对应关系
8  │   │  transform.ts // 处理 AST 中的 vue 特有语法
9  │   │  utils.ts
10 │   │
11 │   └─transforms
12 │          transformElement.ts
13 │          transformExpression.ts
14 │          transformText.ts
15 │
16 └─__tests__                 // 测试用例
17     │  codegen.spec.ts
18     │  parse.spec.ts
19     │  transform.spec.ts
20     │
21     └─__snapshots__
22           codegen.spec.ts.snap
23
```

compile逻辑

```
1 // src/index.ts
2 export { baseCompile } from "./compile";
3
4 // src/compiler.ts
5 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:
```

```
56
57        traverseChildren(node, context);
58        break;
59
60      default:
61        break;
62    }
63
64
65
66    let i = exitFns.length;
67    // i-- 这个很巧妙
68    // 使用 while 是要比 for 快 (可以使用 https://jsbench.me/ 来测试一下)
69    while (i--) {
70      exitFns[i]();
71    }
72  }
73
74  function createRootCodegen(root: any, context: any) {
75    const { children } = root;
76
77    // 只支持有一个根节点
78    // 并且还是一个 single text node
79    const child = children[0];
80
81    // 如果是 element 类型的话 ， 那么我们需要把它的 codegenNode 赋值给 root
82    // root 其实是个空的什么数据都没有的节点
83    // 所以这里需要额外的处理 codegenNode
84    // codegenNode 的目的是专门为了 codegen 准备的  为的就是和 ast 的 node 分离开
85    if (child.type === NodeTypes.ELEMENT && child.codegenNode) {
86      const codegenNode = child.codegenNode;
87      root.codegenNode = codegenNode;
88    } else {
89      root.codegenNode = child;
90    }
91  }
92
```

- generate

```
1  export function generate(ast, options = {}) {
2    // 先生成 context
3    const context = createCodegenContext(ast, options);
4    const { push, mode } = context;
5
```

```
  // 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,
  };
}
```

reactivity

> 负责Vue3中响应式实现的部分

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
```

```
9  |
10 └─__tests__ // 测试用例
11         computed.spec.ts
12         dep.spec.ts
13         effect.spec.ts
14         reactive.spec.ts
15         readonly.spec.ts
16         ref.spec.ts
17         shallowReadonly.spec.ts
```

reactivity逻辑

- index.ts

```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

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

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

```

```ts
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);
}

```

runtime-core

> 运行的核心流程，其中包括初始化流程和更新流程

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
```

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
    // 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.provi
    // 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
    // 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
    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
    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() {
```

```
21    // 如果同时触发了两个组件的更新的话
22    // 这里就会触发两次 then （微任务逻辑）
23    // 但是着是没有必要的
24    // 我们只需要触发一次即可处理完所有的 job 调用
25    // 所以需要判断一下 如果已经触发过 nextTick 了
26    // 那么后面就不需要再次触发一次 nextTick 逻辑了
27    if (isFlushPending) return;
28    isFlushPending = true;
29    nextTick(flushJobs);
30  }
31
32  export function queuePreFlushCb(cb) {
33    queueCb(cb, activePreFlushCbs);
34  }
35
36  function queueCb(cb, activeQueue) {
37    // 直接添加到对应的列表内就ok
38    // todo 这里没有考虑 activeQueue 是否已经存在 cb 的情况
39    // 然后在执行 flushJobs 的时候就可以调用 activeQueue 了
40    activeQueue.push(cb);
41
42    // 然后执行队列里面所有的 job
43    queueFlush()
44  }
45
46  function flushJobs() {
47    isFlushPending = false;
48
49    // 先执行 pre 类型的 job
50    // 所以这里执行的job 是在渲染前的
51    // 也就意味着执行这里的 job 的时候 页面还没有渲染
52    flushPreFlushCbs();
53
54    // 这里是执行 queueJob 的
55    // 比如 render 渲染就是属于这个类型的 job
56    let job;
57    while ((job = queue.shift())) {
58      if (job) {
59        job();
60      }
61    }
62  }
63
64  function flushPreFlushCbs() {
65    // 执行所有的 pre 类型的 job
66    for (let i = 0; i < activePreFlushCbs.length; i++) {
67      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 slo
    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;
}

```

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；

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
    // 存储所有的事件函数
    const invokers = el._vei || (el._vei = {});
```

```
41      const existingInvoker = invokers[key];
42      if (nextValue && existingInvoker) {
43        // patch
44        // 直接修改函数的值即可
45        existingInvoker.value = nextValue;
46      } else {
47        const eventName = key.slice(2).toLowerCase();
48        if (nextValue) {
49          const invoker = (invokers[key] = nextValue);
50          el.addEventListener(eventName, invoker);
51        } else {
52          el.removeEventListener(eventName, existingInvoker);
53          invokers[key] = undefined;
54        }
55      }
56    } else {
57      if (nextValue === null || nextValue === "") {
58        el.removeAttribute(key);
59      } else {
60        el.setAttribute(key, nextValue);
61      }
62    }
63  }
64
65  function insert(child, parent, anchor = null) {
66    console.log("Insert");
67    parent.insertBefore(child, anchor);
68  }
69
70  function remove(child) {
71    const parent = child.parentNode;
72    if (parent) {
73      parent.removeChild(child);
74    }
75  }
76
77  let renderer;
78
79  function ensureRenderer() {
80    // 如果 renderer 有值的话，那么以后都不会初始化了
81    return (
82      renderer ||
83      (renderer = createRenderer({
84        createElement,
85        createText,
86        setText,
87        setElementText,
```

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

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

export * from "@vue/runtime-core"

```

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；

目录结构

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

```

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
  }>`;
}

```

shared

> 公用逻辑

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);
```

```
23
24 export function hasChanged(value, oldValue) {
25         return !Object.is(value, oldValue);
26 }
27
28 export function hasOwn(val, key) {
29         return Object.prototype.hasOwnProperty.call(val, key);
30 }
31
32 /**
33  * @private
34  * 首字母大写
35  */
36 export const capitalize = (str: string) => str.charAt(0).toUpperCase() + str.sli
37
38 /**
39  * @private
40  * 添加 on 前缀，并且首字母大写
41  */
42 export const toHandlerKey = (str: string) => (str ? `on${capitalize(str)}` : ``)
43
44 // 用来匹配 kebab-case 的情况
45 // 比如 onTest-event 可以匹配到 T
46 // 然后取到 T 在前面加一个 - 就可以
47 // \BT 就可以匹配到 T 前面是字母的位置
48 const hyphenateRE = /\B([A-Z])/g;
49 /**
50  * @private
51  */
52 export const hyphenate = (str: string) => str.replace(hyphenateRE, '-$1').toLowe
53
54
55 // 组件的类型
56 export const enum ShapeFlags {
57   // 最后要渲染的 element 类型
58   ELEMENT = 1,
59   // 组件类型
60   STATEFUL_COMPONENT = 1 << 2,
61   // vnode 的 children 为 string 类型
62   TEXT_CHILDREN = 1 << 3,
63   // vnode 的 children 为数组类型
64   ARRAY_CHILDREN = 1 << 4,
65   // vnode 的 children 为 slots 类型
66   SLOTS_CHILDREN = 1 << 5
67 }
68
69 export const toDisplayString = (val) => {
```

```
70    return String(val);
71 };
72
```

Vue 3 Diff算法

最长递增子序列

Vue3 diff 优化点

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；

实现思路

vue3的diff算法其中有两个理念。第一个是相同的前置与后置元素的预处理；第二个则是最长递增子序列。

前置与后置的预处理

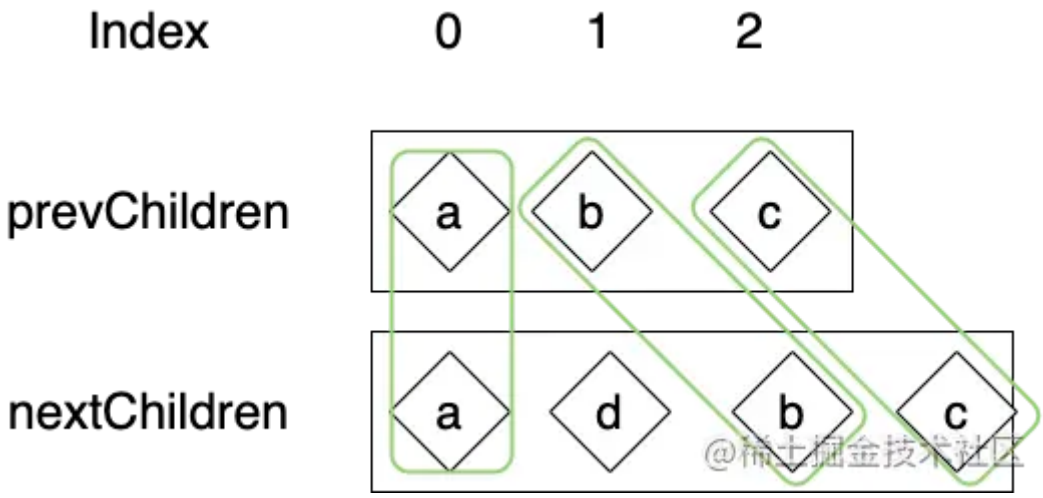
我们看这两段文字

```
1 hello chenghuai
2 hey chenghuai
```

我们会发现，这两段文字是有一部分是相同的，这些文字是不需要修改也不需要移动的，真正需要进行修改中间的几个字母，所以diff就变成以下部分

```
1 text1: llo
2 text2: y
```

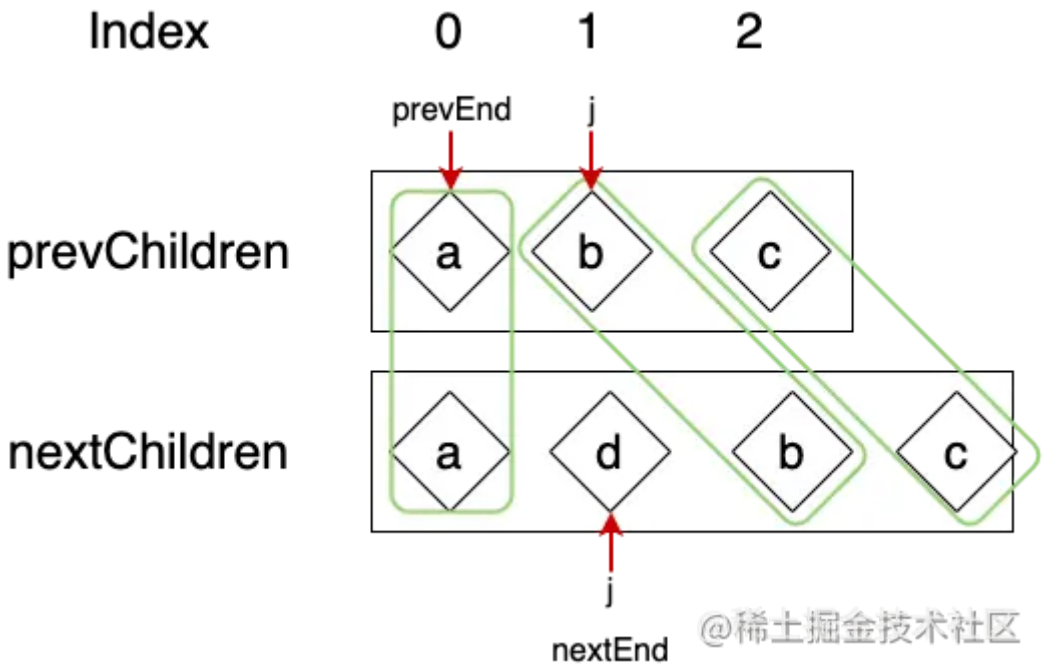
接下来换成 `vnode`：



图中的被绿色框起来的节点，他们是不需要移动的，只需要进行打补丁 `patch` 就可以了。我们把该逻辑写成代码。

```
function vue3Diff(prevChildren, nextChildren, parent) {
  let j = 0,
    prevEnd = prevChildren.length - 1,
    nextEnd = nextChildren.length - 1,
    prevNode = prevChildren[j],
    nextNode = nextChildren[j];
  while (prevNode.key === nextNode.key) {
    patch(prevNode, nextNode, parent)
    j++
    prevNode = prevChildren[j]
    nextNode = nextChildren[j]
  }

  prevNode = prevChildren[prevEnd]
  nextNode = prevChildren[nextEnd]

  while (prevNode.key === nextNode.key) {
    patch(prevNode, nextNode, parent)
    prevEnd--
    nextEnd--
    prevNode = prevChildren[prevEnd]
    nextNode = prevChildren[nextEnd]
  }
}
```

这时候，我们需要考虑边界情况，一种是j > prevEnd；另一种是j > nextEnd。



在上图中，此时j > prevEnd且j <= nextEnd，只需要把新列表中 j 到nextEnd之间剩下的节点插入进去就可以了。相反， 如果j > nextEnd时，把旧列表中 j 到prevEnd之间的节点删除就可以了。

```
function vue3Diff(prevChildren, nextChildren, parent) {
  // ...
  if (j > prevEnd && j <= nextEnd) {
    let nextpos = nextEnd + 1,
      refNode = nextpos >= nextChildren.length
                ? null
                : nextChildren[nextpos].el;
    while(j <= nextEnd) mount(nextChildren[j++], parent, refNode)

  } else if (j > nextEnd && j <= prevEnd) {
    while(j <= prevEnd) parent.removeChild(prevChildren[j++].el)
  }
}
```

在while循环时，指针是从两端向内逐渐靠拢的，所以我们应该在循环中就应该去判断边界情况，我们使用label语法，当我们触发边界情况时，退出全部的循环，直接进入判断：

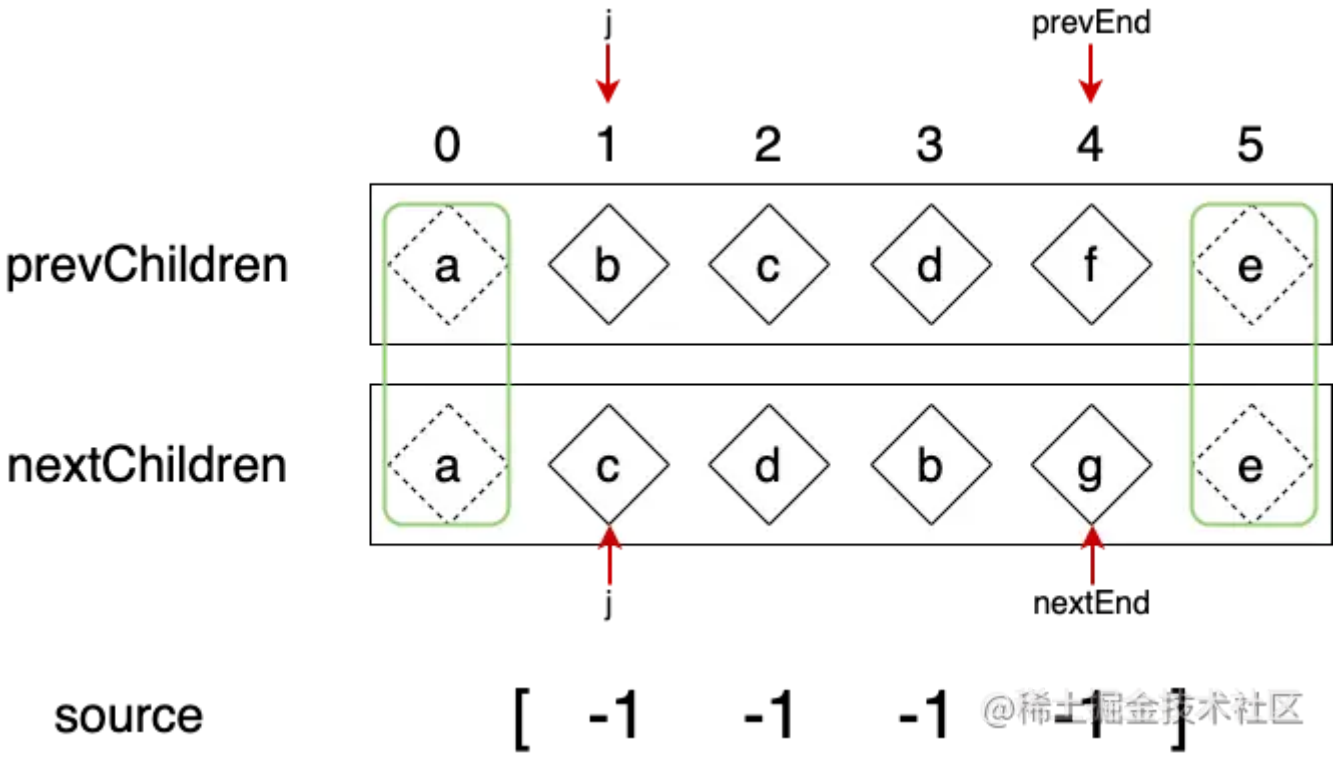
```
function vue3Diff(prevChildren, nextChildren, parent) {
  let j = 0,
    prevEnd = prevChildren.length - 1,
    nextEnd = nextChildren.length - 1,
    prevNode = prevChildren[j],
    nextNode = nextChildren[j];
  // label语法
  outer: {
```

```
    while (prevNode.key === nextNode.key) {
      patch(prevNode, nextNode, parent)
      j++
      // 循环中如果触发边界情况，直接break，执行outer之后的判断
      if (j > prevEnd || j > nextEnd) break outer
      prevNode = prevChildren[j]
      nextNode = nextChildren[j]
    }

    prevNode = prevChildren[prevEnd]
    nextNode = prevChildren[nextEnd]

    while (prevNode.key === nextNode.key) {
      patch(prevNode, nextNode, parent)
      prevEnd--
      nextEnd--
      // 循环中如果触发边界情况，直接break，执行outer之后的判断
      if (j > prevEnd || j > nextEnd) break outer
      prevNode = prevChildren[prevEnd]
      nextNode = prevChildren[nextEnd]
    }
  }

  // 边界情况的判断
  if (j > prevEnd && j <= nextEnd) {
    let nextpos = nextEnd + 1,
      refNode = nextpos >= nextChildren.length
                ? null
                : nextChildren[nextpos].el;
    while(j <= nextEnd) mount(nextChildren[j++], parent, refNode)

  } else if (j > nextEnd && j <= prevEnd) {
    while(j <= prevEnd) parent.removeChild(prevChildren[j++].el)
  }
}
```

判断是否需要移动

接下来，就是找到移动的节点，然后给他移动到正确的位置

当前/后置的预处理结束后，我们进入真正的diff环节。首先，我们先根据新列表剩余的节点数量，创建一个source数组，并将数组填满-1。

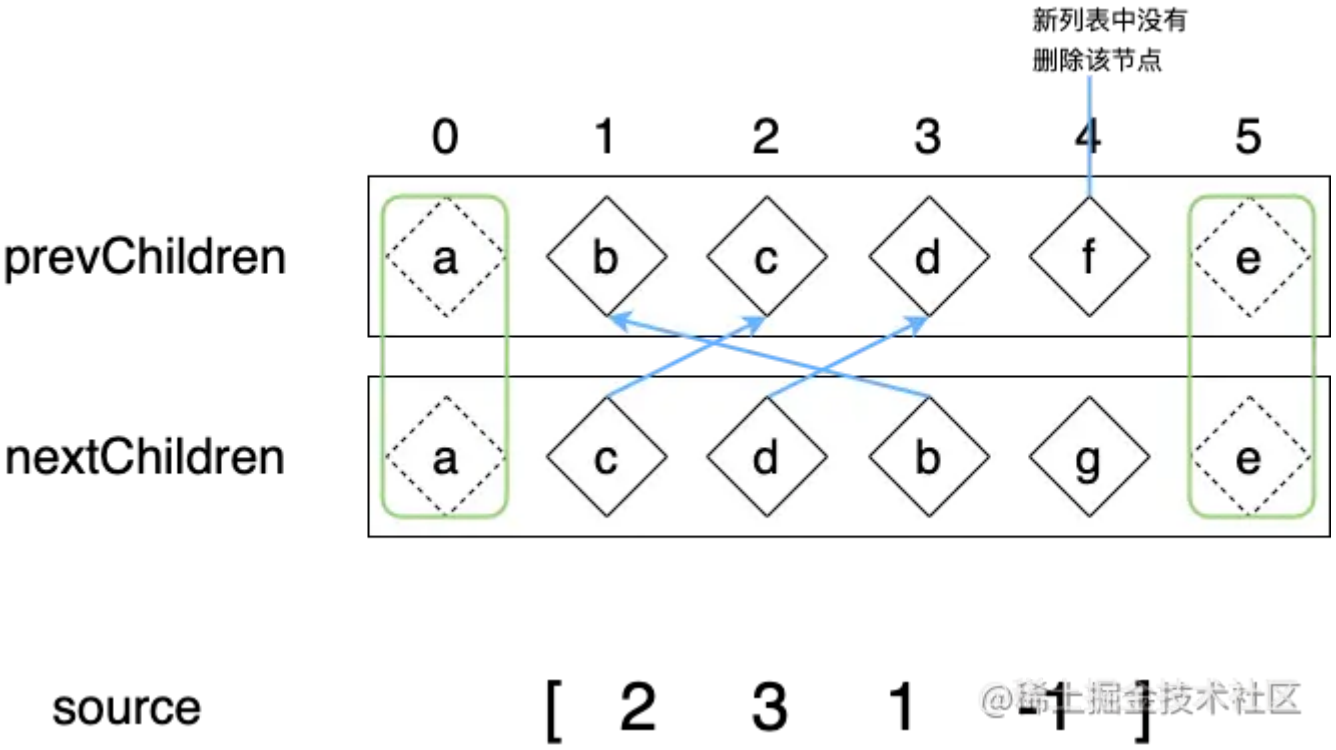



```
function vue3Diff(prevChildren, nextChildren, parent) {
  //...
  outer: {
  // ...
  }

  // 边界情况的判断
  if (j > prevEnd && j <= nextEnd) {
    // ...
  } else if (j > nextEnd && j <= prevEnd) {
    // ...
  } else {
    let prevStart = j,
      nextStart = j,
      nextLeft = nextEnd - nextStart + 1,      // 新列表中剩余的节点长度
      source = new Array(nextLeft).fill(-1);  // 创建数组，填满-1

  }
}
```

source是用来做新旧节点的对应关系的，我们将新节点在旧列表的位置存储在该数组中，我们在根据source计算出它的最长递增子序列用于移动DOM节点。为此，先建立一个对象存储当前新列表中的节点与index的关系，再去旧列表中去找位置。

注意：如果旧节点在新列表中没有的话，直接删除就好。除此之外，我们还需要一个数量表示记录我们已经patch过的节点，如果数量已经与新列表剩余的节点数量一样，那么剩下的旧节点我们就直接删除了就可以了

```
function vue3Diff(prevChildren, nextChildren, parent) {
  //...
  outer: {
  // ...
  }

  // 边界情况的判断
  if (j > prevEnd && j <= nextEnd) {
    // ...
  } else if (j > nextEnd && j <= prevEnd) {
    // ...
  } else {
    let prevStart = j,
      nextStart = j,
      nextLeft = nextEnd - nextStart + 1,      // 新列表中剩余的节点长度
      source = new Array(nextLeft).fill(-1),  // 创建数组，填满-1
      nextIndexMap = {},                       // 新列表节点与index的映射
      patched = 0;                             // 已更新过的节点的数量

    // 保存映射关系
    for (let i = nextStart; i <= nextEnd; i++) {
      let key = nextChildren[i].key
      nextIndexMap[key] = i
    }

    // 去旧列表找位置
    for (let i = prevStart; i <= prevEnd; i++) {
      let prevNode = prevChildren[i],
              prevKey = prevNode.key,
        nextIndex = nextIndexMap[prevKey];
      // 新列表中没有该节点 或者 已经更新了全部的新节点，直接删除旧节点
      if (nextIndex === undefind || patched >= nextLeft) {
        parent.removeChild(prevNode.el)
        continue
      }
      // 找到对应的节点
      let nextNode = nextChildren[nextIndex];
      patch(prevNode, nextNode, parent);
      // 给source赋值
      source[nextIndex - nextStart] = i
      patched++
    }
  }
}
```



找到位置后，我们观察这个重新赋值后的source，我们可以看出，如果是全新的节点的话，其在source数组中对应的值就是初始的-1，通过这一步我们可以区分出来哪个为全新的节点，哪个是可复用的。

其次，我们要判断是否需要移动，如果我们找到的index是一直递增的，说明不需要移动任何节点。我们通过设置一个变量来保存是否需要移动的状态。

```
function vue3Diff(prevChildren, nextChildren, parent) {
  //...
  outer: {
  // ...
  }

  // 边界情况的判断
  if (j > prevEnd && j <= nextEnd) {
    // ...
  } else if (j > nextEnd && j <= prevEnd) {
    // ...
  } else {
    let prevStart = j,
      nextStart = j,
      nextLeft = nextEnd - nextStart + 1,      // 新列表中剩余的节点长度
      source = new Array(nextLeft).fill(-1),   // 创建数组，填满-1
      nextIndexMap = {},                       // 新列表节点与index的映射
      patched = 0,
      move = false,                            // 是否移动
      lastIndex = 0;                           // 记录上一次的位置

```

```
    // 保存映射关系
    for (let i = nextStart; i <= nextEnd; i++) {
      let key = nextChildren[i].key
      nextIndexMap[key] = i
    }

    // 去旧列表找位置
    for (let i = prevStart; i <= prevEnd; i++) {
      let prevNode = prevChildren[i],
              prevKey = prevNode.key,
        nextIndex = nextIndexMap[prevKey];
      // 新列表中没有该节点 或者 已经更新了全部的新节点，直接删除旧节点
      if (nextIndex === undefind || patched >= nextLeft) {
        parent.removeChild(prevNode.el)
        continue
      }
      // 找到对应的节点
      let nextNode = nextChildren[nextIndex];
      patch(prevNode, nextNode, parent);
      // 给source赋值
      source[nextIndex - nextStart] = i
      patched++

      // 递增方法，判断是否需要移动
      if (nextIndex < lastIndex) {
              move = true
      } else {
              lastIndex = nextIndex
      }
    }

    if (move) {

    // 需要移动
    } else {

    //不需要移动
    }
  }
}
```

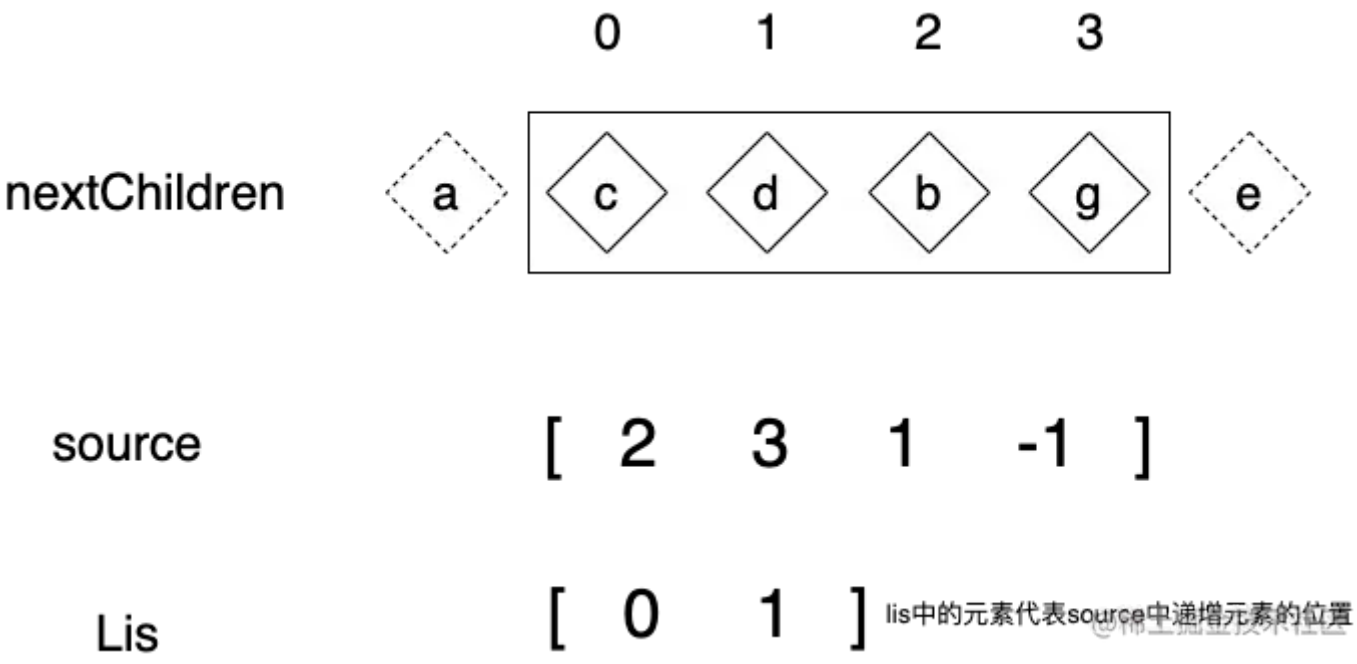
DOM如何移动

判断完是否需要移动后，我们就需要考虑如何移动了。一旦需要进行DOM移动，我们首先要做的就是找到source的最长递增子序列。

```
function vue3Diff(prevChildren, nextChildren, parent) {
  //...
  if (move) {
        const seq = lis(source); // [0, 1]
  // 需要移动
  } else {

  //不需要移动
  }
}
```

> 最长递增子序列：给定一个数值序列，找到它的一个子序列，并且子序列中的值是递增的，子序列中的元素在原序列中不一定连续。
>
> 例如给定数值序列为：[0, 8, 4, 12]。
>
> 那么它的最长递增子序列就是：[0, 8, 12]。
>
> 当然答案可能有多种情况，例如：[0, 4, 12] 也是可以的。

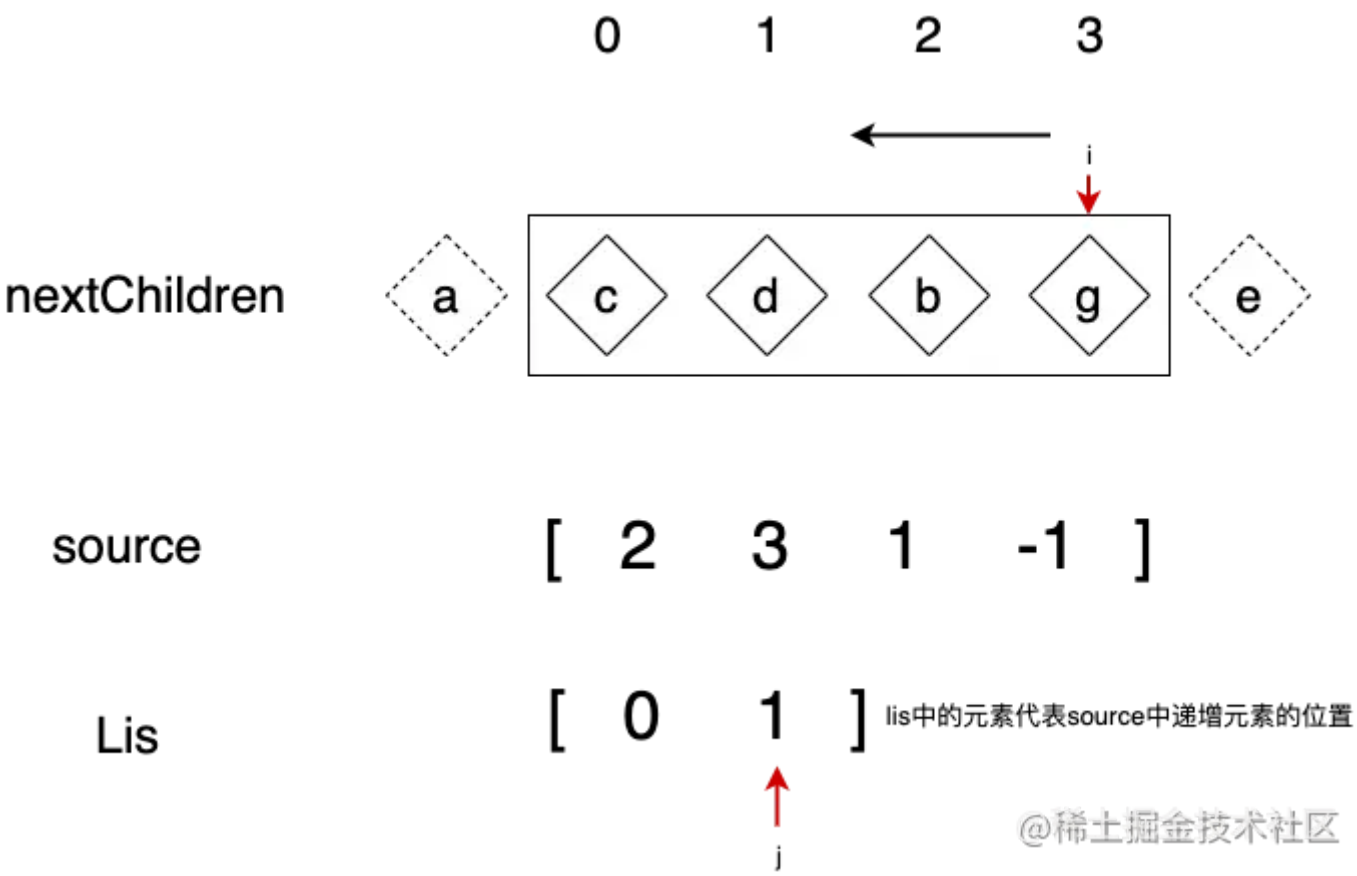
上面的代码中，我们调用lis 函数求出数组source的最长递增子序列为[0, 1]。我们知道 source 数组的值为 [2, 3, 1, -1]，很显然最长递增子序列应该是[2, 3]，计算出的结果是[0, 1]代表的是最长递增子序列中的各个元素在source数组中的位置索引，如下图所示：



我们根据source，对新列表进行重新编号，并找出了最长递增子序列。

我们从后向前进行遍历source每一项。此时会出现三种情况：

1. 当前的值为-1，这说明该节点是全新的节点，又由于我们是从后向前遍历，我们直接创建好DOM节点插入到队尾就可以了；
2. 当前的索引为最长递增子序列中的值，也就是i === seq[j]，这说说明该节点不需要移动；
3. 当前的索引不是最长递增子序列中的值，那么说明该DOM节点需要移动，这里也很好理解，我们也是直接将DOM节点插入到队尾就可以了，因为队尾是排好序的；




```
function vue3Diff(prevChildren, nextChildren, parent) {
  //...
  if (move) {
   // 需要移动
        const seq = lis(source); // [0, 1]
    let j = seq.length - 1;  // 最长子序列的指针
    // 从后向前遍历
    for (let i = nextLeft - 1;  i >= 0; i--) {
      let pos = nextStart + i, // 对应新列表的index
        nextNode = nextChildren[pos],        // 找到vnode
              nextPos = pos + 1,     // 下一个节点的位置，用于移动DOM
        refNode = nextPos >= nextChildren.length ? null : nextChildren[nextPos].
        cur = source[i];  // 当前source的值，用来判断节点是否需要移动

      if (cur === -1) {
        // 情况1，该节点是全新节点
              mount(nextNode, parent, refNode)
      } else if (cur === seq[j]) {
```

```
        // 情况2，是递增子序列，该节点不需要移动
        // 让j指向下一个
        j--
      } else {
        // 情况3，不是递增子序列，该节点需要移动
        parent.insetBefore(nextNode.el, refNode)
      }
    }

  } else {
  //不需要移动

  }
}
```

说完了需要移动的情况，再说说不需要移动的情况。如果不需要移动的话，我们只需要判断是否有全新的节点给他添加进去就可以了。具体代码如下：

```

function vue3Diff(prevChildren, nextChildren, parent) {
  //...
  if (move) {
        const seq = lis(source); // [0, 1]
    let j = seq.length - 1;  // 最长子序列的指针
    // 从后向前遍历
    for (let i = nextLeft - 1;  i >= 0; i--) {
      let pos = nextStart + i, // 对应新列表的index
        nextNode = nextChildren[pos],        // 找到vnode
              nextPos = pos + 1,     // 下一个节点的位置，用于移动DOM
        refNode = nextPos >= nextChildren.length ? null : nextChildren[nextPos].
        cur = source[i];  // 当前source的值，用来判断节点是否需要移动

      if (cur === -1) {
        // 情况1，该节点是全新节点
              mount(nextNode, parent, refNode)
      } else if (cur === seq[j]) {
        // 情况2，是递增子序列，该节点不需要移动
        // 让j指向下一个
        j--
      } else {
        // 情况3，不是递增子序列，该节点需要移动
        parent.insetBefore(nextNode.el, refNode)
      }
    }
  } else {
```

```
    //不需要移动
    for (let i = nextLeft - 1;  i >= 0; i--) {
      let cur = source[i];  // 当前source的值，用来判断节点是否需要移动

      if (cur === -1) {
       let pos = nextStart + i, // 对应新列表的index
          nextNode = nextChildren[pos],        // 找到vnode
          nextPos = pos + 1,     // 下一个节点的位置，用于移动DOM
          refNode = nextPos >= nextChildren.length ? null : nextChildren[nextPos
              mount(nextNode, parent, refNode)
      }
    }
  }
}
```

最长递增子序列

强烈建议看leetcode原题解法：https://leetcode.cn/problems/longest-increasing-subsequence/

我们以该数组为例

```
[10,9,2,5,3,8,7,13][10,9,2,5,3,8,7,13]
```

我们可以使用动态规划的思想考虑这个问题。动态规划的思想是将一个大的问题分解成多个小的子问题，并尝试得到这些子问题的最优解，子问题的最优解有可能会在更大的问题中被利用，这样通过小问题的最优解最终求得大问题的最优解。

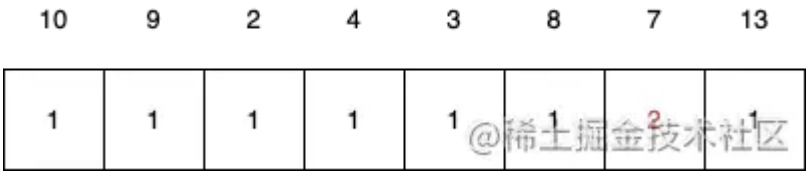
我们先假设只有一个值的数组[13]，那么该数组的最长递增子序列就是[13]自己本身，其长度为1。那么我们认为每一项的递增序列的长度值均为1

那么我们这次给数组增加一个值[7, 13], 由于7 < 13，所以该数组的最长递增子序列是[7, 13]，那么该长度为2。那么我们是否可以认为，当[7]小于[13]时，以[7]为头的递增序列的长度是，[7]的长度和[13]的长度的和，即1 + 1 = 2。

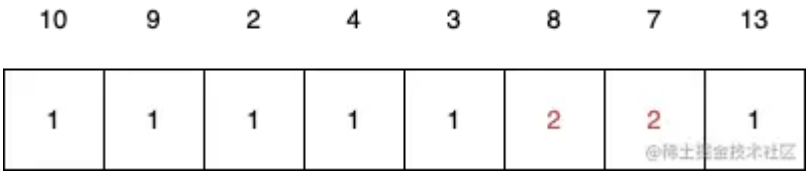
ok，我们基于这种思想来给计算一下该数组。我们先将每个值的初始赋值为1

10	9	2	4	3	8	7	13
1	1	1	1	1	1	1	1

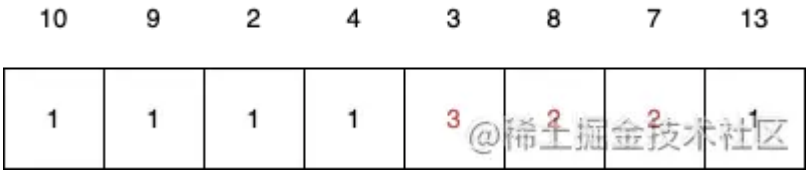
首先 7 < 13 那么7对应的长度就是13的长度再加1，1 + 1 = 2

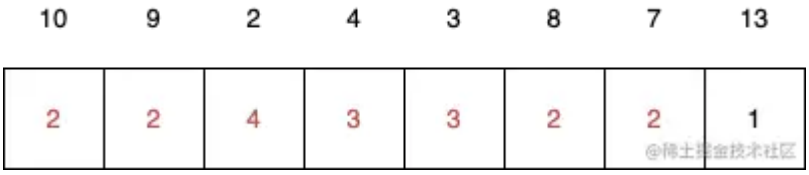


继续，我们对比8。我们首先和7比，发现不满足递增，但是没关系我们还可以继续和13比，8 < 13满足递增，那么8的长度也是13的长度在加一，长度为2



我们再对比3，我们先让其与8进行对比，3 < 8，那么3的长度是8的长度加一，此时3的长度为3。但是还没结束，我们还需要让3与7对比。同样3 < 7，此时我们需要在计算出一个长度是7的长度加一同样是3，我们对比两个长度，如果原本的长度没有本次计算出的长度值大的话，我们进行替换，反之则我们保留原本的值。由于3 === 3，我们选择不替换。最后，我们让3与13进行对比，同样的3 < 13，此时计算出的长度为2，比原本的长度3要小，我们选择保留原本的值。





我们从中取最大的值4，该值代表的最长递增子序列的个数。代码如下：

```
function lis(arr) {
  let len = arr.length,
    dp = new Array(len).fill(1); // 用于保存长度
  for (let i = len - 1; i >= 0; i--) {
    let cur = arr[i]
    for(let j = i + 1; j < len; j++) {
      let next = arr[j]
      // 如果是递增 取更大的长度值
      if (cur < next) dp[i] = Math.max(dp[j]+1, dp[i])
    }
  }
```

```
  return Math.max(...dp)
}
```

在vue3.0中，我们需要的是最长递增子序列在原本数组中的索引。所以我们还需要在创建一个数组用于保存每个值的最长子序列所对应在数组中的index。具体代码如下：

```
function lis(arr) {
  let len = arr.length,
    res = [],
    dp = new Array(len).fill(1);
  // 存默认index
  for (let i = 0; i < len; i++) {
    res.push([i])
  }
  for (let i = len - 1; i >= 0; i--) {
    let cur = arr[i],
      nextIndex = undefined;
    // 如果为-1 直接跳过，因为-1代表的是新节点，不需要进行排序
    if (cur === -1) continue
    for (let j = i + 1; j < len; j++) {
      let next = arr[j]
      // 满足递增条件
      if (cur < next) {
        let max = dp[j] + 1
        // 当前长度是否比原本的长度要大
        if (max > dp[i]) {
          dp[i] = max
          nextIndex = j
        }
      }
    }
    // 记录满足条件的值，对应在数组中的index
    if (nextIndex !== undefined) res[i].push(...res[nextIndex])
  }
  let index = dp.reduce((prev, cur, i, arr) => cur > arr[prev] ? i : prev, dp.le
  // 返回最长的递增子序列的index
  return result[index]
}
```