复习

作业

属性更新是如何实现的?

```
//patch.js
const hooks = ['create', 'activate', 'update', 'remove', 'destroy']
export function createPatchFunction (backend) {
    // 传递进来的扩展模块和节点操作对象
    const { modules, nodeOps } = backend
    for (i = 0; i < hooks.length; ++i) {
        // cbs['update'] = []
        cbs[hooks[i]] = []
        //modules: [ attrs, klass, events, domProps, style, transition]
        for (j = 0; j < modules.length; ++j) {
            // modules[0]['update'] 是创建属性执行函数，其他hook以此类推
            if (isDef(modules[j][hooks[i]])) {
                cbs[hooks[i]].push(modules[j][hooks[i]])
            }
        }
        // cbs['update']: [fn,fn,fn....]
    }

    function patchVnode (...) {
        if (isDef(data) && isPatchable(vnode)) {
          // 每次patch时先对属性做更新
          for (i = 0; i < cbs.update.length; ++i) cbs.update[i](oldVnode,
vnode)
          if (isDef(i = data.hook) && isDef(i = i.update)) i(oldVnode, vnode)
        }
    }
}
```

知识点

模板编译

模板编译的主要目标是**将模板(template)转换为渲染函数(render)**

Vue 2.0需要用到VNode描述视图以及各种交互，手写显然不切实际，因此用户只需编写类似HTML代码的Vue模板，通过编译器将模板转换为可返回VNode的render函数。

体验模板编译

带编译器的版本中，可以使用template或el的方式声明模板

```
<div id="demo">
  <h1>Vue.js测试</h1>
    <p>{{foo}}</p>
</div>
<script>
    // 使用el方式
    new Vue({
        data: { foo: 'foo' },
        el: "#demo",
    });
</script>
```

然后输出渲染函数

```
<script>
    const app = new Vue({});
    // 输出render函数
    console.log(app.$options.render);
</script>
```

> 输出结果大致如下：
>
> ```
> function anonymous() {
> with (this) {
>   return _c('div', { attrs: { "id": "demo" } }, [
>       _c('h1', [_v("Vue.js测试")]),
>       _v(" "),
>       _c('p', [_v(_s(foo))])
>   ])
> }
> }
> ```
>
> 元素节点使用createElement创建，别名_c
>
> 本文节点使用createTextVNode创建，别名_v
>
> 表达式先使用toString格式化，别名_s

模板编译过程

实现模板编译共有三个阶段：解析、优化和生成

解析 - parse

解析器将模板解析为抽象语法树AST，只有将模板解析成AST后，才能基于它做优化或者生成代码字符串。

调试查看得到的AST，/src/compiler/parser/index.js，结构如下：

```
▼root: Object
  ▶attrs: [{…}]
  ▶attrsList: [{…}]
  ▶attrsMap: {id: "demo"}
  ▼children: Array(3)
    ▶0: {type: 1, tag: "h1", attrsList: Array(0), attrsMap: {…},
    ▶1: {type: 3, text: " ", start: 37, end: 42}
    ▶2: {type: 1, tag: "p", attrsList: Array(0), attrsMap: {…},
     length: 3
    ▶__proto__: Array(0)
   end: 65
   parent: undefined
   plain: false
  ▶rawAttrsMap: {id: {…}}
   start: 0
   tag: "div"
   type: 1
```

解析器内部分了**HTML解析器**、**文本解析器**和**过滤器解析器**，最主要是HTML解析器，核心算法说明：

```
//src/compiler/parser/index.js
parseHTML(tempalte, {
    start(tag, attrs, unary){}, // 遇到开始标签的处理
    end(){},// 遇到结束标签的处理
    chars(text){},// 遇到文本标签的处理
    comment(text){}// 遇到注释标签的处理
})
```

优化 - optimize

优化器的作用是在AST中找出静态子树并打上标记。静态子树是在AST中永远不变的节点，如纯文本节点。

标记静态子树的好处：

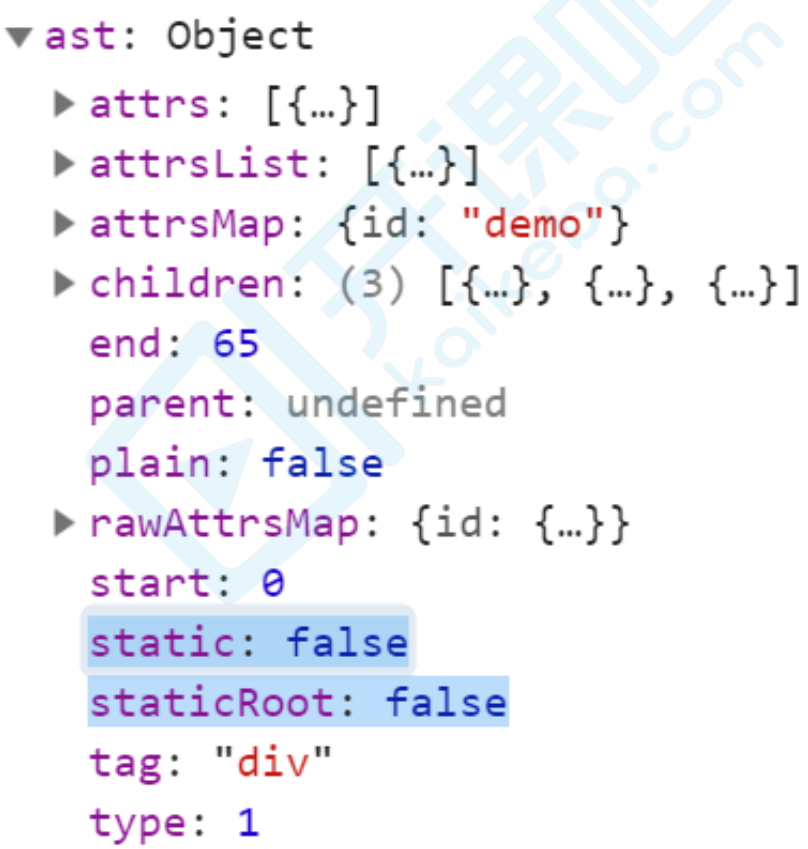
- 每次重新渲染，不需要为静态子树创建新节点

- 虚拟DOM中patch时，可以跳过静态子树

代码实现，src/compiler/optimizer.js - optimize

```
export function optimize (root: ?ASTElement, options: CompilerOptions) {
  if (!root) return
  isStaticKey = genStaticKeysCached(options.staticKeys || '')
  isPlatformReservedTag = options.isReservedTag || no
  // 找出静态节点并标记
  markStatic(root)
  // 找出静态根节点并标记
  markStaticRoots(root, false)
}
```

标记结束



代码生成 - generate

将AST转换成渲染函数中的内容，即代码字符串。

generate方法生成渲染函数代码，src/compiler/codegen/index.js

```
export function generate (
  ast: ASTElement | void,
```

```
  options: CompilerOptions
): CodegenResult {
  const state = new CodegenState(options)
  const code = ast ? genElement(ast, state) : '_c("div")'
  return {
    render: `with(this){return ${code}}`,
    staticRenderFns: state.staticRenderFns
  }
}

//生成的code长这样
`_c('div',{attrs:{"id":"demo"}},[
  _c('h1',[_v("Vue.js测试")]),
  _c('p',[_v(_s(foo))])
])`
```

v-if、v-for

着重观察几个结构性指令的解析过程

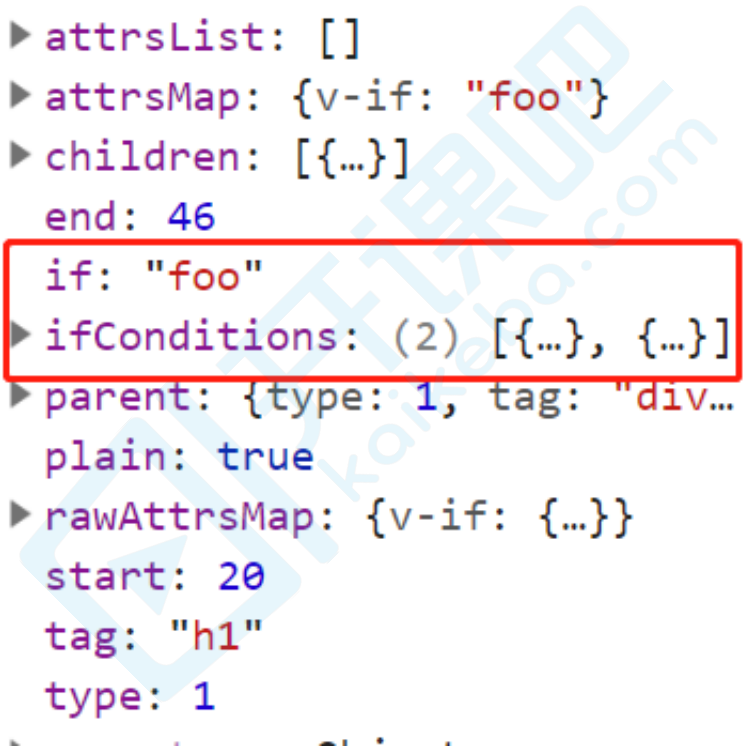
```
// 解析v-if，parser/index.js
function processIf (el) {
  const exp = getAndRemoveAttr(el, 'v-if') // 获取v-if="exp"中exp并删除v-if属性
  if (exp) {
    el.if = exp // 为ast添加if表示条件
    addIfCondition(el, { // 为ast添加ifConditions表示各种情况对应结果
      exp: exp,
      block: el
    })
  } else { // 其他情况处理
    if (getAndRemoveAttr(el, 'v-else') != null) {
      el.else = true
    }
    const elseif = getAndRemoveAttr(el, 'v-else-if')
    if (elseif) {
      el.elseif = elseif
    }
  }
}

// 代码生成，codegen/index.js
function genIfConditions (
  conditions: ASTIfConditions,
  state: CodegenState,
  altGen?: Function,
  altEmpty?: string
): string {
```

```
  const condition = conditions.shift() // 每次处理一个条件
  if (condition.exp) { // 每种条件生成一个三元表达式
    return `(${condition.exp})?${
      genTernaryExp(condition.block)
    }:${
      genIfConditions(conditions, state, altGen, altEmpty)
    }`
  } else {
    return `${genTernaryExp(condition.block)}`
  }

  // v-if with v-once should generate code like (a)?_m(0):_m(1)
  function genTernaryExp (el) {}
}
```

解析结果：



生成结果：

```
"with(this){return _c('div',{attrs:{"id":"demo"}},[
  (foo) ? _c('h1',[_v(_s(foo))]) : _c('h1',[_v("no title")]),
    _v(" "),_c('abc')],1)}"
```

插槽

组件编译的顺序是**先编译父组件，再编译子组件**。

普通插槽是在父组件编译和渲染阶段生成 `vnodes`，**数据的作用域是父组件**，子组件渲染的时候直接拿到这些渲染好的 `vnodes`。

作用域插槽，父组件在编译和渲染阶段并不会直接生成 `vnodes`，而是在父节点保留一个 `scopedSlots` 对象，**存储着不同名称的插槽以及它们对应的渲染函数**，只有在编译和**渲染子组件阶段才会执行**这个渲染函数生成 `vnodes`，由于是在子组件环境执行的，所以对应的**数据作用域是子组件实例**。

解析相关代码：

```
// processSlotContent：处理<template v-slot:xxx="yyy">
const slotBinding = getAndRemoveAttrByRegex(el, slotRE)  // 查找v-slot:xxx
if (slotBinding) {
    const { name, dynamic } = getSlotName(slotBinding) // name是xxx
    el.slotTarget = name // xxx赋值到slotTarget
    el.slotTargetDynamic = dynamic
    el.slotScope = slotBinding.value || emptySlotScopeToken // yyy赋值到
slotScope
}

// processSlotOutlet：处理<slot>
if (el.tag === 'slot') {
    el.slotName = getBindingAttr(el, 'name') // 获取slot的name并赋值到slotName
}
```

生成相关代码：

```
// genScopedSlot：这里把slotScope作为形参转换为工厂函数返回内容
const fn = `function(${slotScope}){` +
    `return ${el.tag === 'template'
      ? el.if && isLegacySyntax
        ? `(${el.if})?${genChildren(el, state) || 'undefined'}:undefined`
        : genChildren(el, state) || 'undefined'
      : genElement(el, state)
    }}`
// reverse proxy v-slot without scope on this.$slots
const reverseProxy = slotScope ? `` : `,proxy:true`
return `{key:${el.slotTarget || `"default"`},fn:${fn}${reverseProxy}}`
```

vue.js项目中一些最佳实践

- 项目配置

  npm run eject

  vue.config.js

做一些基础配置：指定应用上下文、端口号、主页title

```
// vue.config.js
const port = 7070;
const title = "vue项目最佳实践";

module.exports = {
  publicPath: '/best-practice', // 部署应用包时的基本 URL
  devServer: {
    port: port,
  },
  configureWebpack: {
    // 向index.html注入标题
    name: title
  }
};

// index.html
<title><%= webpackConfig.name %></title>
```

- 链式操作：演示webpack规则配置

范例：项目要使用icon，传统方案是图标字体(字体文件+样式文件)，不便维护；svg方案采用svg-sprite-loader自动加载打包，方便维护。

使用icon前先安装依赖：svg-sprite-loader

```
npm i svg-sprite-loader -D
```

下载图标，存入src/icons/svg中

修改规则和新增规则，vue.config.js

```
// resolve定义一个绝对路径获取函数
const path = require('path')

function resolve(dir) {
  return path.join(__dirname, dir)
}
//...
chainWebpack(config) {
    // 配置svg规则排除icons目录中svg文件处理
    config.module
        .rule("svg")
        .exclude.add(resolve("src/icons"));

    // 新增icons规则，设置svg-sprite-loader处理icons目录中的svg
    config.module
```

```
        .rule("icons") //新增icons规则
          .test(/\.svg$/) //test选项
          .include.add(resolve("src/icons")) // include选项是数组
            .end() //add完上下文是数组不是icons规则，使用end()回退
          .use("svg-sprite-loader") //添加use选项
            .loader("svg-sprite-loader") //切换上下文为svg-sprite-loader
            .options({ symbolId: "icon-[name]" }) // 为svg-sprite-loader新
增选项
          .end();// 回退
}
```

图标自动导入

```
// icons/index.js
// 指定require上下文
const req = require.context('./svg', false, /\.svg$/)
// 遍历加载上下文中所有项
req.keys().map(req);

// main.js
import './icons'
```

创建SvgIcon组件，./components/SvgIcon.vue

```
<template>
  <svg :class="svgClass" aria-hidden="true" v-on="$listeners">
    <use :xlink:href="iconName" />
  </svg>
</template>

<script>
export default {
  name: 'SvgIcon',
  props: {
    iconClass: {
      type: String,
      required: true
    },
    className: {
      type: String,
      default: ''
    }
  },
  computed: {
    iconName() {
      return `#icon-${this.iconClass}`
    },
```

```
    svgClass() {
      if (this.className) {
        return 'svg-icon ' + this.className
      } else {
        return 'svg-icon'
      }
    }
  }
}
</script>

<style scoped>
.svg-icon {
  width: 1em;
  height: 1em;
  vertical-align: -0.15em;
  fill: currentColor;
  overflow: hidden;
}
</style>
```

注册，icons/index.js

```
import Vue from 'vue'
import SvgIcon from '@/components/SvgIcon.vue'

Vue.component('svg-icon', SvgIcon)
```

使用，App.vue

```
<svg-icon icon-class="qq"></svg-icon>
```

- 权限控制

  路由分为两种，`constantRoutes` 和 `asyncRoutes`。

  定义路由，router/index.js

```
import Vue from "vue";
import Router from "vue-router";
import Layout from '@/layout'; // 布局页

Vue.use(Router);

// 通用页面
export const constRoutes = [
  {
    path: "/login",
```

```
    component: () => import("@/views/Login"),
    hidden: true // 导航菜单忽略该项
  },
  {
    path: "/",
    component: Layout,// 应用布局
    redirect: "/home",
    children: [
      {
        path: "home",
        component: () =>
          import(/* webpackChunkName: "home" */ "@/views/Home.vue"),
        name: "home",
        meta: {
            title: "Home", // 导航菜单项标题
            icon: "qq" // 导航菜单项图标
        }
      }
    ]
  }
];
// 权限页面
export const asyncRoutes = [
  {
    path: "/about",
    component: Layout,
    redirect: "/about/index",
    children: [
      {
        path: "index",
        component: () =>
          import(/* webpackChunkName: "home" */ "@/views/About.vue"),
        name: "about",
        meta: {
            title: "About",
            icon: "qq",
            roles: ['admin', 'editor']
        },
      }
    ]
  }
];
export default new Router({
  mode: "history",
  base: process.env.BASE_URL,
  routes: constRoutes
});
```

路由守卫，创建./src/permission.js，并在main.js中引入

```
import router from './router'
import store from './store'
import { Message } from 'element-ui'
import { getToken } from '@/utils/auth' // 从cookie获取令牌

const whiteList = ['/login'] // 无需令牌白名单

router.beforeEach(async (to, from, next) => {

  // 获取令牌判断用户是否登录
  const hasToken = getToken()

  if (hasToken) {
    if (to.path === '/login') {
      // 若已登录重定向至首页
      next({ path: '/' })
    } else {
      // 若用户角色已附加则说明动态路由已添加
      const hasRoles = store.getters.roles && store.getters.roles.length >
0
      if (hasRoles) {
        next() // 继续即可
      } else {
        try {
          // 先请求获取用户信息
          const { roles } = await store.dispatch('user/getInfo')

          // 根据当前用户角色动态生成路由
          const accessRoutes = await
store.dispatch('permission/generateRoutes', roles)

          // 添加这些路由至路由器
          router.addRoutes(accessRoutes)

          // 继续路由切换，确保addRoutes完成
          next({ ...to, replace: true })
        } catch (error) {
          // 出错需重置令牌并重新登录（令牌过期、网络错误等原因）
          await store.dispatch('user/resetToken')
          Message.error(error || 'Has Error')
          next(`/login?redirect=${to.path}`)
        }
      }
    }
  } else {
    // 用户无令牌
    if (whiteList.indexOf(to.path) !== -1) {
      // 白名单路由放过
      next()
```

```
    } else {
      // 重定向至登录页
      next(`/login?redirect=${to.path}`)
    }
  }
})
```

utils/auth.js

```
import Cookies from "js-cookie";

const Token = "token";

export function getToken() {
  return Cookies.get(Token);
}

export function setToken(token) {
  return Cookies.set(Token, token);
}

export function removeToken() {
  return Cookies.remove(Token);
}
```

vuex相关模块实现，创建store/index.js

```
import Vue from 'vue'
import Vuex from 'vuex'
import permission from './modules/permission'
import user from './modules/user'

Vue.use(Vuex)

const store = new Vuex.Store({
  modules: {permission, user}
})

export default store
```

user模块：用户数据、用户登录等，store/modules/user.js

```
import { getToken, setToken, removeToken } from "@/utils/auth";

const state = {
  token: getToken(),
  roles: []
  // 其他用户信息
```

```
};

const mutations = {
  SET_TOKEN: (state, token) => {
    state.token = token;
  },
  SET_ROLES: (state, roles) => {
    state.roles = roles;
  }
};

const actions = {
  // user login
  login({ commit }, userInfo) {
    const { username } = userInfo;
    return new Promise((resolve, reject) => {
      setTimeout(() => {
        if (username === "admin" || username === "jerry") {
          commit("SET_TOKEN", username);
          setToken(username);
          resolve();
        } else {
          reject("用户名、密码错误");
        }
      }, 1000);
    });
  },

  // get user info
  getInfo({ commit, state }) {
    return new Promise((resolve) => {
      setTimeout(() => {
        const roles = state.token === 'admin' ? ['admin'] : ['editor']
        commit("SET_ROLES", roles);
        resolve({roles});
      }, 1000);
    });
  },

  // remove token
  resetToken({ commit }) {
    return new Promise(resolve => {
      commit("SET_TOKEN", "");
      commit("SET_ROLES", []);
      removeToken();
      resolve();
    });
  }
};
```

```
export default {
  namespaced: true,
  state,
  mutations,
  actions
};
```

permission模块：路由配置信息、路由生成逻辑, store/modules/permission.js

```
import { asyncRoutes, constRoutes } from "@/router";

/**
 * 根据路由meta.role确定是否当前用户拥有访问权限
 * @roles 用户拥有角色
 * @route 待判定路由
 */
function hasPermission(roles, route) {
  // 如果当前路由有roles字段则需判断用户访问权限
  if (route.meta && route.meta.roles) {
    // 若用户拥有的角色中有被包含在待判定路由角色表中的则拥有访问权
    return roles.some(role => route.meta.roles.includes(role));
  } else {
    // 没有设置roles则无需判定即可访问
    return true;
  }
}

/**
 * 递归过滤AsyncRoutes路由表
 * @routes 待过滤路由表，首次传入的就是AsyncRoutes
 * @roles 用户拥有角色
 */
export function filterAsyncRoutes(routes, roles) {
  const res = [];

  routes.forEach(route => {
    // 复制一份
    const tmp = { ...route };
    // 如果用户有访问权则加入结果路由表
    if (hasPermission(roles, tmp)) {
      // 如果存在子路由则递归过滤之
      if (tmp.children) {
        tmp.children = filterAsyncRoutes(tmp.children, roles);
      }
      res.push(tmp);
    }
  });
```

```
  return res;
}

const state = {
  routes: [], // 完整路由表
  addRoutes: [] // 用户可访问路由表
};

const mutations = {
  SET_ROUTES: (state, routes) => {
    state.addRoutes = routes;
    state.routes = constRoutes.concat(routes);
  }
};

const actions = {
  // 路由生成：在得到用户角色后会第一时间调用
  generateRoutes({ commit }, roles) {
    return new Promise(resolve => {
      let accessedRoutes;
      //   用户是管理员则拥有完整访问权限
      if (roles.includes("admin")) {
        accessedRoutes = asyncRoutes || [];
      } else {
        //   否则需要根据角色做过滤处理
        accessedRoutes = filterAsyncRoutes(asyncRoutes, roles);
      }
      commit("SET_ROUTES", accessedRoutes);
      resolve(accessedRoutes);
    });
  }
};

export default {
  namespaced: true,
  state,
  mutations,
  actions
};
```

getters编写，store/index.js

```
const store = new Vuex.Store({
  modules: { permission, user },
  // 全局定义getters便于访问
  getters: {
    roles: state => state.user.roles,
  }
});
```

布局页面，layout/index.vue

```
<template>
  <div class="app-wrapper">
    <!-- <sidebar class="sidebar-container" /> -->
    <div class="main-container">
      <router-view />
    </div>
  </div>
</template>
```

用户登录页面，views/Login.vue

```
<template>
  <div>
    <h2>用户登录</h2>
    <div>
      <input type="text" v-model="username">
      <button @click="login">登录</button>
    </div>
  </div>
</template>
<script>
export default {
  data() {
    return {
      username: "admin"
    };
  },
  methods: {
    login() {
      this.$store
        .dispatch("user/login", { username: this.username })
        .then(() => {
          this.$router.push({
            path: this.$route.query.redirect || "/"
          });
        })
```

```
        .catch(error => {
            alert(error)
        });
    }
  }
};
</script>
```