`

关于Vue 3.0

前言：

- 正式版：Vue团队于2020.09.18日发布3.0正式版
- 前置条件：Vue虽然保留了大量的2.0版本api,但由于是使用TypeScript重构，所以想要学习3.0起码要掌握TS的基础使用

Vue3.0中的六大亮点

序号	特性	解析
1	Performance	性能上比Vue2.0快1.3~2倍
2	Tree shaking support	按需编译，体积更加轻量化
3	Composition API	组合API，可参考React hooks理解
4	Better TypeScript support	对 Ts 提供了更好的支持
5	Custom Renderer API	暴露了自定义渲染API
6	Fragment,Teleport(Protal),Suspense	更先进的组件

注：具体可以参考github中Vue3.0的相关源文件https://github.com/vuejs/vue-next/tree/master/packages

---

Vue3.0是基于什么优化，如何做到更轻量，更快的?

- 一、diff 算法优化
  - Vue 2中的虚拟Dom是全量比较
  - Vue 3新增静态标记（PatchFlag）
  - 在与数据变化后，与上次虚拟DOM节点比较时，只比较带有PatchFlag标记的节点
  - 并且可以从flag信息中得知具体需要比较的内容。

> 静态标记就是非全量比较，只会比较那些被标记的变量，比较的数量大大减少因此提升性能
>
> > 这让我想到了JS垃圾回收机制里的标记清除，ORZ 感觉熟悉，但回收机是全标记只是清除具有离开环境的标记变量而已）
> >
> > > **内存垃圾回收机制在我去年的博文中👉点击** 比如下面这个示例

```
    <div>
        <a>土豆哇~ </a>
        <p>静态文本</p>
        <p>{{msg}}</p>
    </div>
//------------在下方编译中可以清晰看到标记--------------------
```

```
export function render(_ctx, _cache, $props, $setup, $data, $options) {
  return (_openBlock(), _createBlock("div", null, [
    _createVNode("a", null, " 土豆哇~ "),
    _createVNode("p", null, "静态文本"),
    _createVNode("p", null, _toDisplayString(_ctx.msg), 1 /* text文本在这里标记为1
*/)
  ]))
}
//编译网址--->  https://vue-next-template-explorer.netlify.app/
```

- 由以上可得知：
  - 在vue2.0中对于数据变化后重新渲染的DOM树，会与上次渲染的DOM树逐个比较节点
  - 在vue3.0的diff中，创建虚拟DOM时，会根据该DOM是否会变化而添加静态标记，数据更新需要生成新的虚拟DOM时，只会与上次渲染的且被标记的节点比较。
  - 不同的动态变化类型，为了便于区分，标记的数值也不同
  - 因此在vue3.0中比较次数更少，效率更高，速度更快。

示例

```
export function render(_ctx, _cache, $props, $setup, $data, $options) {
  return (_openBlock(), _createBlock("div", null, [
    _createVNode("a", { id: _ctx.Poo }, " 土豆哇~ ", 8 /* PROPS */, ["id"]),
    _createVNode("p", { class: _ctx.style }, " 静态文本", 2 /* CLASS */),
    _createVNode("p", null, _toDisplayString(_ctx.msg), 1 /* TEXT */)
  ]))
}
```

标记查询列表

```
  TEXT = 1,// --取值是1---表示具有动态textContent的元素
  CLASS = 1 << 1,  // --取值是2---表示有动态Class的元素
  STYLE = 1 << 2,  // --取值是4---表示动态样式（静态如style="color: pink"，也会提升至
动态）
  PROPS = 1 << 3,  // --取值是8--- 表示具有非类/样式动态道具的元素。
  FULL_PROPS = 1 << 4,  // --取值是16---表示带有动态键的道具的元素，与上面三种相斥
  HYDRATE_EVENTS = 1 << 5,  // --取值是32---表示带有事件监听器的元素
  STABLE_FRAGMENT = 1 << 6,   // --取值是64---表示其子顺序不变，不会改变自顺序的片段。
  KEYED_FRAGMENT = 1 << 7, // --取值是128---表示带有键控或部分键控子元素的片段。
  UNKEYED_FRAGMENT = 1 << 8, // --取值是256---子节点无key绑定的片段（fragment）
  NEED_PATCH = 1 << 9,   // --取值是512---表示只需要非属性补丁的元素，例如ref或hooks
  DYNAMIC_SLOTS = 1 << 10,  // --取值是1024---表示具有动态插槽的元素
```

- 二 、hoistStatic 静态提升
  - vue2.0中，在更新时，元素即使没有变化，也会重新创建进行渲染
  - vue3.0中，不参与更新的元素；会静态提升，只创建一次下次渲染直接复用。

- 因此在vue3.0中复用更多，创建次数更少，速度更快。见下方示例：

```
    <div>
        <a>土豆哇~ </a>
        <p>静态文本</p>
        <p>{{msg}}</p>
        <a href='https://vue-next-template-explorer.netlify.app/'>vue3.0编译地址
</a>
    </div>
```

```
    /**
     * 在下方编译中(在options中勾选hoistStatic)进行静态提升，
     * 可以清晰看到不更新元素未参与重新创建
    */
const _hoisted_1 = /*#__PURE__*/_createVNode("a", null, "土豆哇~ ", -1 /* HOISTED
*/)

export function render(_ctx, _cache, $props, $setup, $data, $options) {
  return (_openBlock(), _createBlock("div", null, [
    _hoisted_1,
    _createVNode("p", { style: _ctx.myStyle }, "静态文本", 4 /* STYLE */),
    _createVNode("p", null, _toDisplayString(_ctx.msg), 1 /* TEXT */),
    _createVNode("a", {
      style: _ctx.myStyle,
      href: "https://vue-next-template-explorer.netlify.app/"
    }, "vue3.0编译地址", 4 /* STYLE */)
  ]))
}
}
```

- 三、cachehandlers 事件侦听缓存
  - onClick默认视为动态绑定，因此会追踪它的变化
  - 事件绑定的函数为同一个，因此不追踪它的变化，直接缓存后进行复用
  - 同样的，我在编译中进行演示

```
    <div>
        <button @click='Pooo'>按钮</button>
    </div>
```

```
    /**
     * 开启事件侦听缓存前：
     * 下方为常规编译后，可以看到静态标记为8
     * 既然有静态标记，那么它就会进行比较
    */
export function render(_ctx, _cache, $props, $setup, $data, $options) {
```

```
  return (_openBlock(), _createBlock("div", null, [
    _createVNode("button", { onClick: _ctx.Pooo }, "按钮", 8 /* PROPS */,
["onClick"])
  ]))
}
```

然后我在options中打开事件侦听缓存（cachehandlers）

```
/**
 * 可以发现打开侦听缓存后，没有静态标记
 * 在diff算法中，没有静态标记的是不会进行比较和进行追踪的
*/
export function render(_ctx, _cache, $props, $setup, $data, $options) {
  return (_openBlock(), _createBlock("div", null, [
    _createVNode("button", {
      onClick: _cache[1] || (_cache[1] = (...args) => (_ctx.Pooo(...args)))
    }, "按钮")
  ]))
}
```

使用vue3.0提供的Vite快速创建项目

- Vite是Vue作者开发的一款意图取代webpack的工具
- 原理是利用ES6的import发送请求加载文件的特性，进而拦截，然后做预编译，省去webpack冗长的打包
- 使用步骤
  - 安装Vite命令： `npm install -g create-vite-app`
  - 创建Vue3项目： `create-vite-app PoooName`
  - 安装依赖：`cd PoooName / npm install / npm run dev`
  - Vue3.0中兼容2.0的写法，具体代码在此文件同级的PoooName项目文件中

vue3.0中的 reactive 用法

- 在2.0中对于业务实现
- 需要先在data中变更补充数据，然后在`methods`或`watch`中补充业务逻辑
- 这样数据和逻辑是分模块的，查找不便，不利于业务的管理和维护
- 为解决这样的问题，Vue3.0中加入了 `reactive`
- Vue3.0提供了setup 组合API的入口函数，可以把数据和业务逻辑组合在一起

```
import { reactive } from "vue"; //在Vue3.0使用中需要引入reactive
export default {
  name: "App",
  //Vue3.0提供了setup 组合API的入口函数
  setup() {
    /**
     * ref一般用来监听简单类型变化（也可以用来监听复杂类型变化,先不讨论）
     * 通常使用reactive用来监听复杂类型变化（比如数组、函数之类）
     * 以下为一种常规的写法
```

```
     */
    let stus = reactive({ stusList: [****its data****], });
    function removeVeget(index) {
      stus.stusList.splice(index, 1);
    }
    return { stus, removeVeget };// 必须暴露出去，组件中才可以使用
  },
  methods: {},
};
```

- 另一种更加优雅的写法，也是**非常非常**推荐的写法是

```
import { reactive } from "vue";
export default {
  name: "App",
  setup() {
    let {stus, removeVeget }=removeItem();// 三、直接声明、获取
    return { stus, removeVeget };//四、暴露给外界组件使用
  },
  methods: {},
};
    /**
     * 保证数据和业务不分散利于更新维护
     * 也避免了setup中的大量数据函数填充
     * 也不需要使用this指向Vue实例
    */
  function removeItem() {
    let stus = reactive({ stusList: [****its data****], });
    function removeVeget(index) {
      stus.stusList.splice(index, 1);
    }
    return {stus,removeVeget} // 二、暴露给组合API使用
  }
```

- 功能分离：
  - 乍一看上方把函数整合到下方，然后在`setup`中引用是很简洁
  - 若需要的业务功能多了呢，比如增加个`updateItem`,`addItem`
  - 虽然数据和逻辑代码还是在一块，但是各种功能聚集在一块还是显得文件臃肿
  - 那么还要继续优化，分离各个功能
    1. 新建一个单独的JS文件，如remove.js
    2. 在APP文件中引入这个JS文件 3.这样就可以在单独的JS文件中对某个功能进行维护了

```
import { reactive } from "vue"; //引入依赖
function removeItem() {//定义函数，实现功能
  let stus = reactive({
    stusList: [
      { id: 1, Name: "potato", price: "2.5" },
      { id: 2, Name: "tomato", price: "3.5" },
```

```
      { id: 3, Name: "cucumber", price: "4.5" },
    ],
  });
  function removeVeget(index) {
    stus.stusList.splice(index, 1);
  }
  return {stus,removeVeget}
}
export  {removeItem};//暴露给外界使用
```

```
/*那么主文件就变成了如下形式（单独JS文件中已经引入reactive）*/
import { removeItem } from "./remove"; //导入删除的业务逻辑模块
export default {
name: "App",
setup() {
  let { stus, removeVeget } = removeItem();
  return { stus, removeVeget };
},
methods: {},
};
```

vue3.0中的 Composition API本质

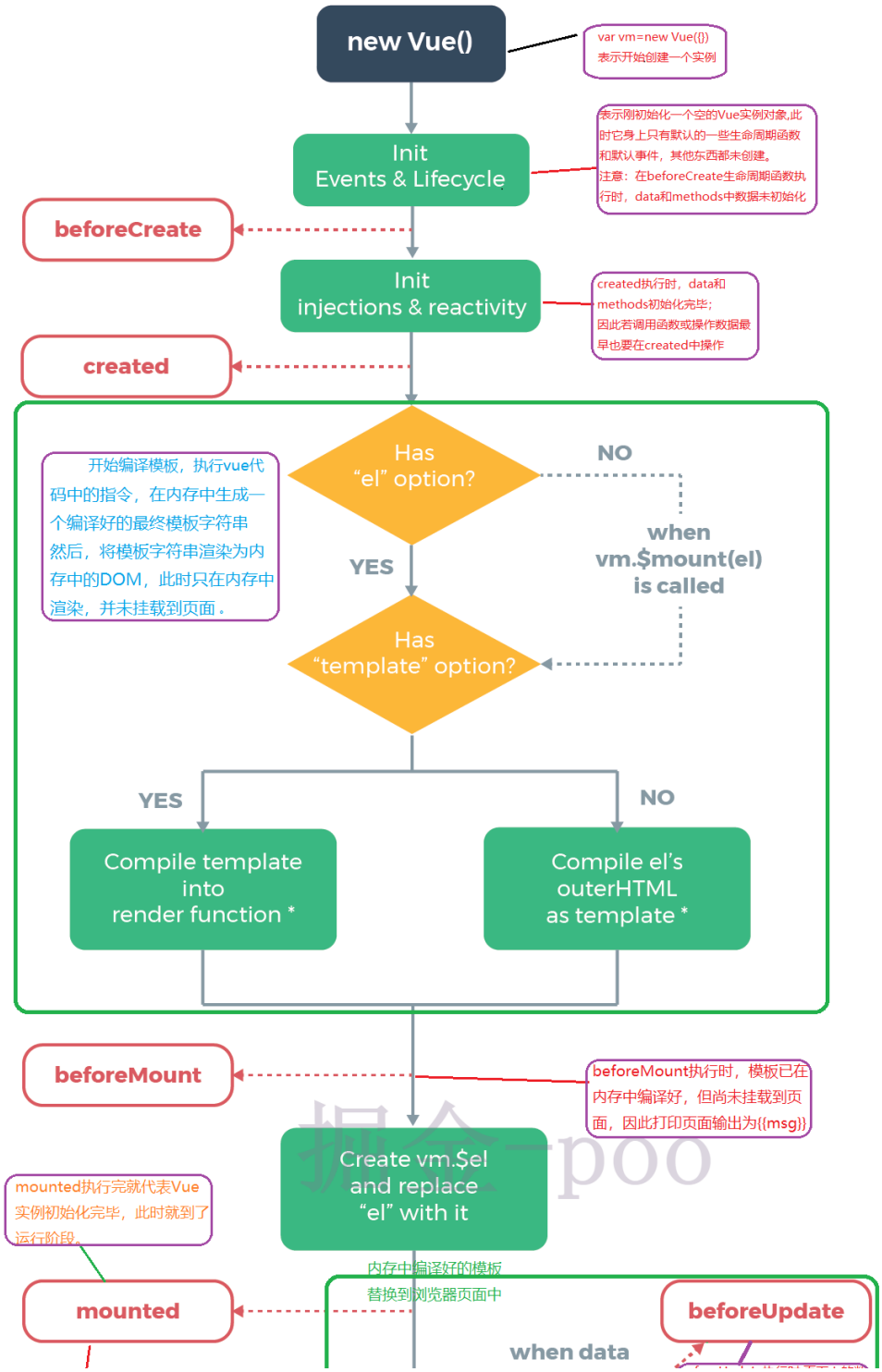
- Option API：即在APP中为实现业务逻辑进行的配置
  - 在2.0中比如你要实现一个点击按钮，弹出提示语功能，你需要
    1. 利用 `Opaction API`
    2. 在 `data` 中配置数据
    3. 在 `methods` 中配置相应函数
  - 在3.0中通过上方 `reactive` 的知识点我们知道，实现这个功能，你需要
    1. 利用 `Composition API`
    2. 在 `setup` 中定义数据，编写函数
    3. 通过 `return{ 数据,方法}`暴露出去
  - 其实 `Composition` (也叫注入API)本质是在运行时
    1. 把暴露出来的数据注入到 `opaction` 中的 `data`
    2. 把暴露出来的函数注入到 `opaction` 中的 `methods`

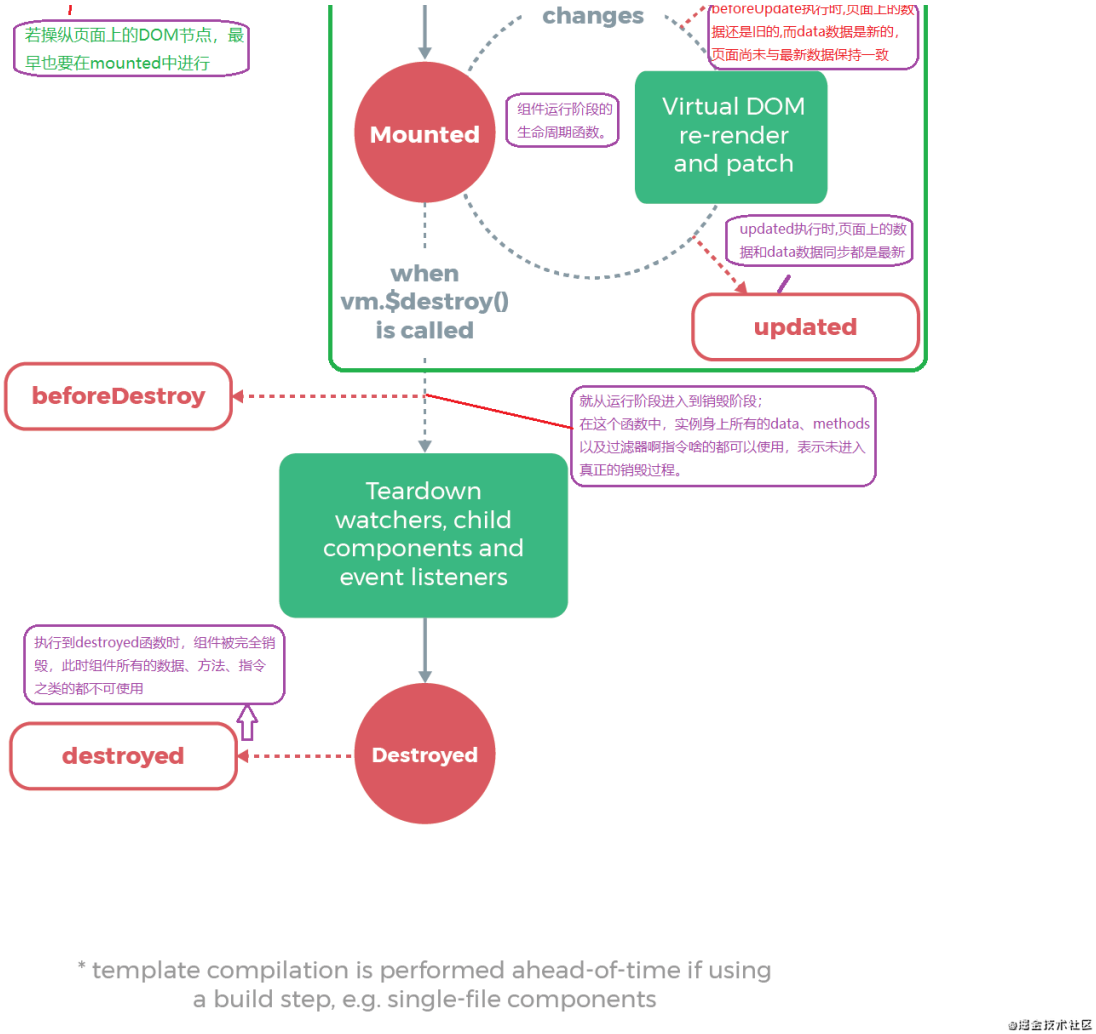
> 注: 具体它咋区分数据还是函数用以注入到相应配置中的，我也不知道（**~~标志位or传参顺序?~~**）

- 小结：
  - `Opaction API`中对配置项都进行了规定，比如：
    - 在data中配置数据，methods中编写方法， `watch` 中进行监听。
    - 保姆式的分配较为清晰，但也对代码层层分割，维护要扒拉半天跳来跳
  - `Composition`中更加自由，比如：
    - 不用担心各种this指向
    - 随意进行模块分割导出，维护时查找固定模块文件

生命周期中的 setup

- **setup的执行时机在`beforeCreate`和`created`之前，前前前**
  - 这其实根据上方总结的内容，用屁股也能想出来原因
  - 在Vue生命周期中我们知道：
    1. `beforeCreate` 时，刚初始化一个空 `Vue` 实例对象， `data` 和 `methods` 中数据 **未初始化**
    2. `created` 执行时，data和methods已经**初始化完毕**
    3. `Composition` 需要把setup中的数据对应注入到 `data` 和 `methods` 中去
    4. 很显然setup必须要在 `created` 之前执行
  - 也因此，若你在Vue3.0中进行混合开发，不可以在 `setup` 中使用 `data` 中的数据和 `methods` 中的方法
  - 在3.0中`setup` 里的this也被修改为了`undefiend`
  - 在3.0中`setup` 里也不可以使用异步
  - （下方贴的图我也加入了旁释，可帮助你回想下生命周期）





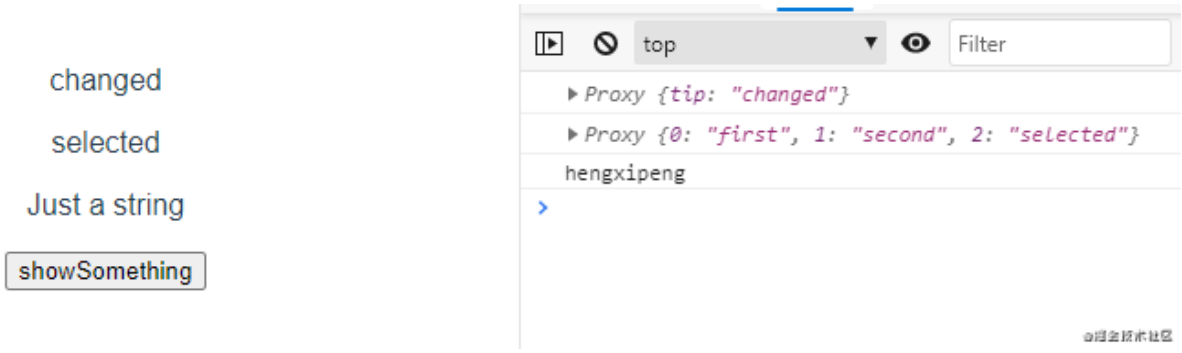
什么是 reactive

- reactive是VUE3.0中提供的实现响应式数据的方法
- 在Vue2.0中使用的是defineProperty来实现的（我自己也手动实现过👉 点击）
- 而VUE3.0中使用的是ES6里的proxy实现的
- reactive中需要注意的点：
    - 传递给它的类型必须是对象（JSON或者arr数组）
    - 并且它会自动把传递进来条件再赋值给Proxy对象
    - 若传递的为上述以外的对象
        1. 在方法中直接修改它，界面上它也不会自动更新
        2. 若想更新只能通过重新赋值的方式

```
/*示例如下*/
setup() {
  let testJson=reactive({
    tip:'its a Json! '
  })
  let testArray=reactive(['first','second','third'])
  let testString=reactive('Just a string')
  function showProxyPar(){
    testJson.tip='changed';
    testArray[2]='selected';
```

```
    testString='hengxipeng';//由于不是对象，所以即使更改视图也不会同步更新
    console.log(testJson);// Proxy {tip: "changed"}
    console.log(testArray);// Proxy {0: "first", 1: "second", 2: "selected"}
    console.log(testString);// hengxipeng
  }
  return { testJson,testArray,testString,showProxyPar };
},
```

- 效果正如下图所示，符合传递条件的参数会再赋值给Proxy，并且修改它也会直接影响视图

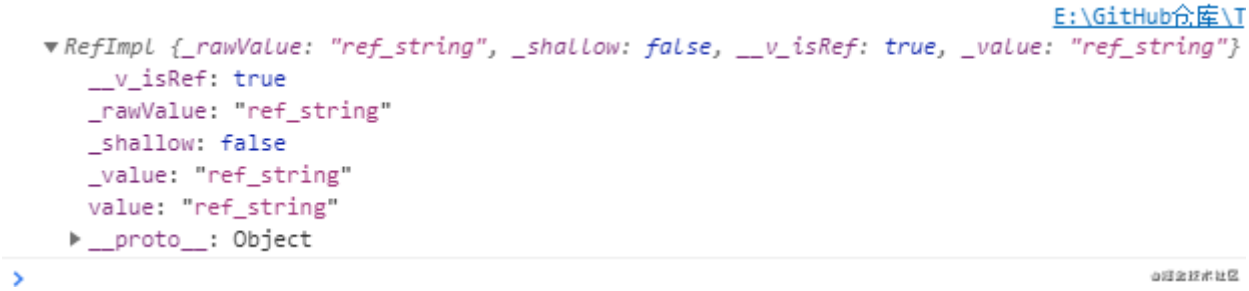


什么是 ref

- 它也是实现响应式数据的方法
- reactivce向来都是进行传递对象，实际开发中若只想更改某简单变量则会显得大材小用
- 所以vue3提供了ref方法，来实现对简单值的监听
- ref本质也是使用reactive，给ref的赋值，它底层会自动转化

```
/**
 * 实质是 ref('its a string')==>reactive({value:'its a string'})
 * 也因此更改时应该 testRef.value=XX 才能更改
 *  (使用的时候不必再加 value，直接使用即可)
 */
setup() {
  let testRef = ref('its a string');
  function showProxyPar() {
    testRef.value='ref_string'
    console.log(testRef);
  }
  return { testRef, showProxyPar };
},
```

- 如下图



ref 和 reactive 之间的不同

- 通过以上得知，使用ref其实相当于使用reactive，只是省略了手动创建对象的步骤
- ref中底层会添加一个value的键，并且在视图中可省略调用value
  - 经过我自己测试
    1. 使用reactive，创建一个键值为value的Json对象，验证是否可省略value调用（**不可以**）
    2. 得知，只有使用ref传递参数时，视图才允许省略value调用

```
/**
 * Vue在解析时，通过  __v_isRef 来判定当前参数是否由 ref 传递出来的
 * 是的话，则会自动在调用当前参数时添加 value
 */
    __v_isRef: true
    _rawValue: "its a string"
    _shallow: false
    _value: "its a string"
    value: "its a string"
```

- 其中Vue3.0中提供了两个方法，isReactive和isRef用来判定数据来源

```
import {isRef,isReactive } from "vue";
  setup() {
    let testReactive = reactive({value:'its a string'});
    let testRef = ref('its a string');
    function showProxyPar() {
      console.log('检测是否是Ref',isRef(testReactive));// false
      console.log('检测是否是Ref',isRef(testRef));// true
    }
    return { testRef,testReactive, showProxyPar };
  }
```

递归监听

- 通常情况下ref和reactive都会监听数据变化

> 验证如下，点击按钮触发recursion 页面显示都会改变

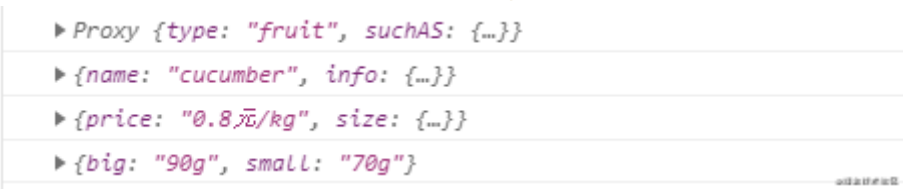
```
//验证ref 只需添加value即可，如:  parse.value.type='fruit';
  setup() {
    let parse = reactive({
      type: "vegetables",
      suchAS: {
        name: "tomato",
        info: {
          price: "0.4元/kg",
          size: {
```

```
            big: "50g",
            small: "20g",
          },
        },
      },
    });
    function recursion() {
      parse.type='fruit';
      parse.suchAS.name='cucumber';
      parse.suchAS.info.price='0.8元/kg';
      parse.suchAS.info.size.small='70g';
      parse.suchAS.info.size.big='90g';
    }
    return { parse,recursion };
  },
```

- 当数据量庞大，需要考虑性能时
  - 在< **什么是** reactive >中总结知道：
    - reactive和ref通过递归取出参数中所有值，包装为proxy对象
    - 递归的优与劣我总结过，涉及内存中的压栈和栈顶弹出等，建议回顾下👉点击

非递归监听

- 上面知道了递归监听上的种种劣势，而Vue3.0也提供了解决方案
  - 非递归监听，即：只监听数据的第一层。方案如下：
    1. 引入Vue3.0中提供的shallowReactive
    2. 改用 shallowReactive({})传递参数
    3. 经过调试发现，只有第一层包装成了proxy对象，如下图



  - 而对于ref对应的shallowRef非递归监听则比较特殊
    1. 首先试引入Vue3.0中官方提供的shallowRef
    2. 原理上与reactive相同，只是它并不会监听JSON第一层数据
    3. 而是要直接修改value的值，这样视图才会同步更新

```
    function recursion() {
      /** * shallowRef 对第一层修改不会监听，所以视图不变 */
      parse.value.type='fruit';
      parse.value.suchAS.name='cucumber';
      parse.value.suchAS.info.price='0.8元/kg';
      parse.value.suchAS.info.size.small='70g';
      parse.value.suchAS.info.size.big='90g';
      /** * 正确做法应该是整个修改 value */
        parse.value = {
        type: "fruit",
        suchAS: {
```

```
        name: "cucumber",
        info: {
          price: "0.8元/kg",
          size: {
            big: "70g",
            small: "90g",
          },},},};}
```

> 注意点：虽然他们只对第一层进行了监听，但若恰巧每次都更改了第一层数据，则也会引起下方数据和视图的同步更新，此时shallowReactive或者shallowRef就和reactive、Ref效果一模一样！

数据监听补充

- 通过以上这些知识点可知：
  - ref和reactive监听每一层数据，响应好但递归取值性能差。
  - shallowReactive和shallowRef监听第一层（或value），性能好但更新值较麻烦
  - shallowRef中，为了数据和视图一致，更新值要更新整个parse.value太繁琐
  - 场景：若我更新数据的第三层，不整个更新value行不行？
    1. 这就用到了Vue3.0为ref准备的triggerRef(不用查啦 就一个)
    2. 作用：根据传入的数据，主动去更新视图
       - 老规矩，import {shallowRef, triggerRef } from "vue"
       - 改完非首层的数据，而你使用的是shallowRef还不想整个更新value
       - 使用triggerRef大法，传入整个对象，就好啦
       - （使用reactive传入的数据，无法触发triggerRef）

```
function recursion() {
  /**
   *  方法一、手动更新
  parse.value = {
    type: "fruit",
    suchAS: {
      name: "cucumber",
      info: {
        price: "0.8元/kg",
        size: {
          big: "70g",
          small: "90g",
      },},},};
   */
  /** * 方法而、使用 triggerRef */
  parse.value.suchAS.info.price='0.8元/kg';
  triggerRef(parse)
}
```

数据监听方式选择

- 正常数据量时，通常使用ref和reactive（递归监听）即可满足业务需要
- 当数据量庞大且注重性能时，就需考虑shallowReactive和shallowRef了（非递归监听）

shallowRef底层原理

- 在看 ref 时，我们知道它的本质其实是 reactive({value:XX})
- 那么 shallowRef 其实是 shallowReactive({value:XX})
    - 因为通过shallowRef 创建的数据，它监听的是 .value 的变化

```
let state1=shallowRef({
  a:'a',
  b:{
    b_1:'b_1',
    b_2:'b_2'
  }
})
//--其实是如下所示
let state2=shallowReactive({
  value:{
      a:'a',
      b:{
        b_1:'b_1',
        b_2:'b_2'
      }
  }
})
```

toRaw

- 在之前的知识体系中我们知道
    - setup 中定义参数对象，在函数中直接修改页面是不会同步更新。
    - 需要利用 Ref 或者 reactive 进行包装，这样修改才生效

```
let obj={ name:'花花',age:'3'}
let test=reactive(obj);
function myFun() {test.name='乐乐';}
```

-
    - obj 和 test是引用关系
    - reactive 会把传进来的参数包装为一个 porxy 对象并返回
    - 例子中 **test** 本质是一个porxy对象，而这个对象也引用了 obj
        - 那么请注意：
            - 直接修改obj或引用的 test 都会引起内存中数据变化
            - 但是修改 obj因为没有 proxy监听，所以视图不会更新
- 说了那么多，再绕回来说toRaw
    - 作用：返回由 reactive 或 readonly 等方法转换成响应式代理的普通对象
    - 特点：toRaw 拿到的数据不会被监听变化，节省性能
    - 场景：数据更改不需更新视图，为提高性能，通过 toRaw 拿到数据修改
    - 提示：因为是原始数据，风险较大，一般不建议使用、
    - 注意：若想拿到的是Ref 创建的对象，记得加 **value**

```
    let obj={
      name:'花花',
      age:'3'
    }
    let testReactive=reactive(obj);
    let testRef=ref(obj);
    let rawReac=toRaw(testReactive);
    let rawRef=toRaw(testRef.value);
    console.log(rawReac===obj); //true
    console.log(rawRef===obj); //true
```

markRaw

- 在之前的知识体系中我们知道
- 作用：固定某数据，不追踪它值的变化,同时视图也不会更新
- 通过控制台查看，使用markRaw的对象参数，被赋予v_skip监听跳过标识符

```
let obj={
    name:'poo',
    age:'3'
}
console.log(obj);//{name: "poo", age: "3"}
 obj=markRaw(obj)//使其值的改变，不会被监听，视图不会发生变化

let testReactive=reactive(obj);
function myFun() {
  testReactive.name='地瓜';
  console.log(obj);//{name: "地瓜", age: "3", __v_skip: true}
}
```

toRef

- toRef和ref一样，同样也是创建响应式数据的
- 先说结论：
  - 1.ref 将对象中某属性变为响应式，修改时原数据不受影响
  - 2.toRef 会改变原数据
  - 3.并且 toRef 创建的数据，改变时界面不会自动更新
- 应用场景：性能优化
  - 想使创建的响应式数据与元数据关联起来
  - 更新响应式数据后，不想更新UI

```
setup() {
    /** * toRef */
    let obj={ name:'poo' }
    let obj2={name:'boo'}
    //-注意：这里是让 toRef 知道是让 obj里的 name变成响应式
```

```
    let test_toRef=toRef(obj,'name');
    let test_ref=ref(obj2.name);
    console.log(test_toRef);
    function myFun() {
      test_toRef.value='土豆';
      test_ref.value='地瓜';
      console.log(obj,);// {name: "土豆"}
      console.log(obj2);// {name: "boo"}
    }
    return {obj,obj2, myFun };
  }
```

toRefs

- toRef只能接受两个参数，当传递某对象多个属性值时会很麻烦
- 结论：
  - 1.toRefs 是避免 toRef 对多个属性操作繁琐
  - 2.toRefs 底层原理是使用 toRef 方法遍历对象属性值

```
  setup() {
      let obj={
        name:'poo',
        age:'3'
      }
    let test_toRefs=toRefs(obj);
    /**
     * 在 toRefs 底层中其实执行了以下遍历方法
     * let par1=toRef(obj,'name')
     * let par2=toRef(obj,'age')
     */
    function myFun() {
      test_toRefs.name.value='HAHA';
      test_toRefs.age.value='13';
    }
    return {test_toRefs, myFun };
  }
```

在 Vue3.0 中如何通过 ref 获取元素？

- 在 Vue2.0版本内，通常使用 this.$refs.XX 获取元素
- 在Vue3.0中，废除了类似$的很多符号，如何获取指定元素？
- 根据Vue生命周期图中可知，要操作DOM，最早也要在mounted中
- 结论：
  - 1.setup 是在beforeCreate之前执行
  - 2.在生命周期中 onMounted最先准备好 DOM元素
  - 3.setup中想操纵 DOM 就在函数中引用 onMounted
  - 4.Vue3.0中生命周期函数被抽离，可根据需要引入相应周期函数

```
  setup() {
    let btn=ref(null);
    console.log(btn.value);
    // 回调函数和它在函数中顺序无关，根据 Vue 生命周期顺序执行
    onMounted(()=>{
      console.log(btn.value);//-  <button>clickMe</button>
    })
    return {btn};
  },
```

readonly

- Vue3.0中提供的这个API，使得数据被保护，只读不可修改
- 默认所有层数据都只读，若只限制第一层只读，可使用`shallowReadonly`
- `isReadonly`用来检测数据创建来源是否是 `readonly`
- 若进行修改，浏览器会提示操作失败，目标只读

```
  setup() {
    let obj={
      name:'poo',
      age:'13'
    }
    let only=readonly(obj)
    function myFun() {
      only.name='HAHA';//  failed: target is readonly
    }
    return {only, myFun };
  }
```

**** Vue3.0响应式数据本质****

- 2.0中使用的 `Object.defineProperty` 实现响应式数据
- 3.0中使用的 `Proxy` 来实现,如下

```
    let obj={
      name:'poo',
      age:'13'
    }
    let objProxy=new Proxy(obj,{
      //数据读会触发
      get(obj,key){
        console.log(obj);//{name: "poo", age: "13"}
        return obj[key]
      },
      //监听的数据被修改会触发
      set(obj,key,value){
      // 操作的对象，操作的属性，赋予的新值
```

```
        obj[key]=value //把外界赋予的新值更新到该对象
        console.log('进行UI之类的操作');
        //-补充，有时会多次操作，此时必须return true才不会影响下次操作
        return true;
        }
      })
     objProxy.name;
```

实现shallowReactive和shallowRef

- 它们二者也是通过参数传递，包装成 `proxy` 对象进行监听
- 在 `Proxy` 的 `set` 监听中，同样只监听第一层
- `shallowRef` 只是在 `shallowReactive` 基础上默认添加 `value` 键名

```
function shallowReactive(obj){
  return new Proxy(obj,{
    get(obj,key){
      return obj[key]
    },
    set(obj,key,value){
      obj[key]=value
      console.log('更新');
      return true;
    }
  })
}
let obj={
  A:'A',
  B:{
    b1:'b1',
    b2:'b2',
    b3:{
      b3_1:'b3-1',
      b3_2:'b3-2'
    } } }
let test=shallowReactive(obj)
//-这里同样只会监听第一层
test.A='apple';
test.B.b2='banana';
function shallowRef(obj){
  return shallowReactive(obj,{value:vl})
}
let state=shallowRef(obj);
```

实现 reactive 和 ref

- 它们与上方区别在于递归监听
- 上方因为直接传递参数对象，所以只监听第一层
- 为了递归监听，那么要把数据的每一层都给包装成 `Proxy`对象

```
function reactive(obj) {
  if (typeof obj === "object") {
    if (obj instanceof Array) {
      //当前参数为数组类型，则循环取出每一项
      obj.forEach((item, index) => {
        if (typeof item === "object") {
          //分析数组每一项，是对象则递归
          obj[index] = reactive(item);
        }
      });
    } else {
      // 当前参数是对象且不是数组，则取属性值并进行分析是否是多层对象
      for (let key in obj) {
        if (typeof obj[key] === "object") {
          obj[key] = reactive(item);
        }
      }
    }
  } else {
    console.log("当前传入为非对象参数");
  }
  //-正常情况下就进行 Proxy对象包装
  return new Proxy(obj, {
    get(obj, key) {
      return obj[key];
    },
    set(obj, key, value) {
      obj[key] = value;
      console.log("更新");
      return true;
    },
  });
}
```

实现 shallowReadonly 和 readonly

- 二者区别只在于首层监听，只读拒绝修改和数据全层修改
- 下方实现的是 `shallowReadonly`
- `readonly` 实现是在 `shallowReadonly` 基础上移除`set` 中的`return true`

```
function shallowReadonly(obj) {
  return new Proxy(obj, {
    get(obj, key) {
      return obj[key];
    },
    set(obj, key, value) {
      // obj[key] = value;
      console.error(`${key}为只读，不可修改-`);
      return true;//此行移除，则就是 readonly 全层数据只读
    },
```

```
  });
}
let parse = {
  type: "fruit",
  suchAS: {
    name: "cucumber",
  },
};
let fakeShowRe=shallowReadonly(parse);
fakeShowRe.type='HAHA';// 此时修改不会生效
fakeShowRe.suchAS.name='HAHA';// 非首层修改会生效
```