开课吧vue3剖析(蹭一下热度)

vue3在国庆假期突然发布，我们讲师们在旅途中看了下新版本的架构和vue2的对比，先做一个初步的解析，毕竟3还没有正式发布，很多功能还没有实现

地址 https://github.com/vuejs/vue-next

现在还处于Pre-Alpha版本，后面至少还有alpha，beta等版本后，估计正式发布得到2020年了

Composition API RFC

不得不说，vue的语法，越来越react了，感觉vue3发布后，会有更多人专项react了

vue2实现计算属性和点击累加的代码就不看了，大家都会，看下vue3的

我们执行npm run dev 调试打个包

```
<!DOCTYPE html>
<html lang="en">
<body>
  <div id='app'></div>
</body>
<script src="./dist/vue.global.js"></script>
<script>
const { createApp, reactive, computed, effect } = Vue;

const RootComponent = {
  template: `
    <button @click="increment">
      Count is: {{ state.count }}, double is: {{ state.double }}
    </button>
  `,
  setup() {
    const state = reactive({
      count: 0,
      double: computed(() => state.count * 2)
    })
```

```
    effect(() => {
      console.log('数字被修改了 ',state.count)
    })
    function increment() {
      state.count++
    }

    return {
      state,
      increment
    }
  }
}

createApp().mount(RootComponent, '#app')
</script>
</html>

```

这个**reactive**和react-hooks越来越像了, 大家可以去Composition API RFC这里看细节

功能

编译器（Compiler）的优化主要在体现在以下几个方面：

- 使用模块化架构
- 优化 "Block tree"
- 更激进的 static tree hoisting 功能
- 支持 Source map
- 内置标识符前缀（又名 "stripWith"）
- 内置整齐打印（pretty-printing）功能
- 移除 source map 和标识符前缀功能后，使用 Brotli 压缩的浏览器版本精简了大约 10KB

运行时（Runtime）的更新主要体现在以下几个方面：

- 速度显著提升
- 同时支持 Composition API 和 Options API，以及 typings
- 基于 Proxy 实现的数据变更检测

- 支持 Fragments
- 支持 Portals
- 支持 Suspense w/ async setup()

最后，还有一些 2.x 的功能尚未移植过来，如下：

- 服务器端渲染
- keep-alive
- transition
- Compiler DOM-specific transforms
- v-on DOM 修饰符 v-model v-text v-pre v-onc v-html v-show

typescript

全部由typescript构建，我们学的TS热度++ 三大库的最终选择，ts乃今年必学技能

大家回顾下

proxy取代deineProperty

除了性能更高以为，还有以下缺陷，也是为啥会有`$set`，`$delete`的原因 1、属性的新加或者删除也无法监听； 2、数组元素的增加和删除也无法监听

reactive模块

看源码

```
export function reactive(target: object) {
  // if trying to observe a readonly proxy, return the readonly version.
```

```
  if (readonlyToRaw.has(target)) {
    return target
  }
  // target is explicitly marked as readonly by user
  if (readonlyValues.has(target)) {
    return readonly(target)
  }
  return createReactiveObject(
    target,
    rawToReactive,
    reactiveToRaw,
    mutableHandlers,
    mutableCollectionHandlers
  )
}

function createReactiveObject(
  target: any,
  toProxy: WeakMap<any, any>,
  toRaw: WeakMap<any, any>,
  baseHandlers: ProxyHandler<any>,
  collectionHandlers: ProxyHandler<any>
) {
  if (!isObject(target)) {
    if (__DEV__) {
      console.warn(`value cannot be made reactive: ${String(target)}`)
    }
    return target
  }
  // target already has corresponding Proxy
  let observed = toProxy.get(target)
  if (observed !== void 0) {
    return observed
  }
  // target is already a Proxy
  if (toRaw.has(target)) {
    return target
  }
  // only a whitelist of value types can be observed.
  if (!canObserve(target)) {
    return target
```

```
  }
  const handlers = collectionTypes.has(target.constructor)
    ? collectionHandlers
    : baseHandlers
  observed = new Proxy(target, handlers)
  toProxy.set(target, observed)
  toRaw.set(observed, target)
  if (!targetMap.has(target)) {
    targetMap.set(target, new Map())
  }
  return observed
}

```

稍微精简下

```
function reactive(target) {
    const handlers = collectionTypes.has(target.constructor)
        ? collectionHandlers
        : baseHandlers
    observed = new Proxy(target, handlers)
    return observed
}

```

基本上除了set map weakset 和weakmap，都是baseHandlers，下面重点关注一下，Proxy的语法 大家需要复习下es6

```
export const readonlyHandlers: ProxyHandler<any> = {
  get: createGetter(true),

  set(target: any, key: string | symbol, value: any, receiver:
any): boolean {
    if (LOCKED) {
      if (__DEV__) {
        console.warn(
```

```
          `Set operation on key "${key as any}" failed: target is readonly.`,
          target
        )
      }
      return true
    } else {
      return set(target, key, value, receiver)
    }
  },

  deleteProperty(target: any, key: string | symbol): boolean {
    if (LOCKED) {
      if (__DEV__) {
        console.warn(
          `Delete operation on key "${key as any}" failed: target is readonly.`,
          target
        )
      }
      return true
    } else {
      return deleteProperty(target, key)
    }
  },

  has,
  ownKeys
}
```

关于proxy

```
let data = [1,2,3]
let p = new Proxy(data, {
  get(target, key) {
    console.log('获取值:', key)
    return target[key]
  },
  set(target, key, value) {
    console.log('修改值:', key, value)
```

```
    target[key] = value
    return true
  }
})

p.push(4)


获取值: push
获取值: length
修改值: 3 4
修改值: length 4
```

比defineproperty优秀的 就是数组和对象都可以直接触发getter和setter， 但是数组会触发两次，因为获取push和修改length的时候也会触发

我们还可以用**Reflect**

```
let data = [1,2,3]
let p = new Proxy(data, {
  get(target, key) {
    console.log('获取值:', key)
    return Reflect.get(target,key)
  },
  set(target, key, value) {
    console.log('修改值:', key, value)
    return Reflect.set(target,key, value)
  }
})

p.push(4)


```

多次触发和深层嵌套问题，一会我们看vue3是怎么解决的

```
let data = {name:{ title:'kkb'}}
let p = new Proxy(data, {
```

```
  get(target, key) {
    console.log('获取值:', key)
    return Reflect.get(target,key)
  },
  set(target, key, value) {
    console.log('修改值:', key, value)
    return Reflect.set(target,key, value)
  }
})

p.name.title = 'xx'

```

vue3深度检测

baseHander

```

function createGetter(isReadonly: boolean) {
  return function get(target: any, key: string | symbol, receiver: any) {
    const res = Reflect.get(target, key, receiver)
    if (typeof key === 'symbol' && builtInSymbols.has(key)) {
      return res
    }
    if (isRef(res)) {
      return res.value
    }
    track(target, OperationTypes.GET, key)
    return isObject(res)
      ? isReadonly
        ? // need to lazy access readonly and reactive here to avoid
          // circular dependency
          readonly(res)
        : reactive(res)
      : res
  }
}
```

返回值如果是object，就再走一次reactive，实现深度

vue3处理重复trigger

很简单，用的hasOwProperty, set肯定会出发多次，但是通知只出去一次， 比如数组修改length的时候，hasOwProperty是true， 那就不触发

```
function set(
  target: any,
  key: string | symbol,
  value: any,
  receiver: any
): boolean {
  value = toRaw(value)
  const hadKey = hasOwn(target, key)
  const oldValue = target[key]
  if (isRef(oldValue) && !isRef(value)) {
    oldValue.value = value
    return true
  }
  const result = Reflect.set(target, key, value, receiver)
  // don't trigger if target is something up in the prototype chain of original
  if (target === toRaw(receiver)) {
    /* istanbul ignore else */
    if (__DEV__) {
      const extraInfo = { oldValue, newValue: value }
      if (!hadKey) {
        trigger(target, OperationTypes.ADD, key, extraInfo)
      } else if (value !== oldValue) {
        trigger(target, OperationTypes.SET, key, extraInfo)
      }
    } else {
      if (!hadKey) {
        trigger(target, OperationTypes.ADD, key)
      } else if (value !== oldValue) {
        trigger(target, OperationTypes.SET, key)
      }
    }
  }
  return result
}
```

手写vue3的reactive

刚才说的细节，我们手写一下

effect

computed

```
// 用这个方法来模式视图更新
 function updateView() {
  console.log('触发视图更新啦')
}
function isObject(t) {
  return typeof t === 'object' && t !== null
}

// 把原目标对象 转变 为响应式的对象
const options = {
  set(target, key, value, reciver) {
    // console.log(key,target.hasOwnProperty(key))
      if(!target.hasOwnProperty(key)){
          updateView()
      }
      return Reflect.set(target, key, value, reciver)
  },
  get(target, key, reciver) {
      const res = Reflect.get(target, key, reciver)
      if(isObject(target[key])){
          return reactive(res)
      }
      return res
  },
  deleteProperty(target, key) {
      return Reflect.deleteProperty(target, key)
  }
}
// 用来做缓存
const toProxy = new WeakMap()

function reactive(target) {
```

```
  if(!isObject(target)){
      return target
  }
  // 如果已经代理过了这个对象，则直接返回代理后的结果即可
  if(toProxy.get(target)){
      return toProxy.get(target)
  }
  let proxyed = new Proxy(target, options)
  toProxy.set(target, proxyed)
  return proxyed
}

// 测试数据
let obj = {
  name: 'Ace7523',
  array: ['a', 'b', 'c']
}

// 把原数据转变响应式的数据
let reactivedObj = reactive(obj)

// 改变数据，期望会触发updateView() 方法 从而更新视图
// reactivedObj.name = 'change'

// reactivedObj.array.unshift(4)
```

其他细节 track收集依赖，trigger触发更新

vue3其他模块细节

代码仓库中有个 packages 目录，里面主要是 Vue 3.0 的相关源码功能实现，具体内容如下所示。

compiler-core

平台无关的编译器，它既包含可扩展的基础功能，也包含所有平台无关的插件。暴露了 AST 和 baseCompile 相关的 API，它能把一个字符串变成一棵 AST

compiler-dom

针对浏览器的编译器。

runtime-core

与平台无关的运行时环境。支持实现的功能有虚拟 DOM 渲染器、Vue 组件和 Vue 的各种 API, 可以用来自定义 renderer ， vue2中也有 ， 入口代码看起来

runtime-dom

针对浏览器的 runtime。其功能包括处理原生 DOM API、DOM 事件和 DOM 属性等， 暴露了重要的render和createApp方法

```
const { render, createApp } = createRenderer<Node, Element>({
  patchProp,
  ...nodeOps
})

export { render, createApp }
```

runtime-test

一个专门为了测试而写的轻量级 runtime。比如对外暴露了**renderToString**方法，在此感慨和react越来越像了

server-renderer

用于 SSR，尚未实现。

shared

没有暴露任何 API，主要包含了一些平台无关的内部帮助方法。

vue

用于构建「完整」版本，引用了上面提到的 runtime 和 compiler目录。入口文件代码如下

```
'use strict'

if (process.env.NODE_ENV === 'production') {
  module.exports = require('./dist/vue.cjs.prod.js')
} else {
  module.exports = require('./dist/vue.cjs.js')
}

```

所以想阅读源码，还是要看构建流程，这个和vue2也是一致的

未完待续

开课吧

https://kaikeba.com/vipcourse/web

https://mp.weixin.qq.com/s/ug86masjqVOJKjc7Ua1dxQ