

State of Vue

VueConf Shanghai
2019.06.08

发展现状

发展现状

- Chrome DevTools: ~90万周活跃用户
 - 对比: React ~160万
- npm 下载量: ~400 万次/月
- Jsdelivr CDN: 5亿次引用/月
- GitHub stars: 14万
 - 全 GitHub 第三, 实际代码项目第一

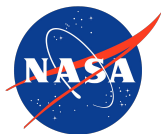
全球化的影响力

- 遍布世界各地的线下聚会 <https://events.vuejs.org/>
- 目前每年在世界各地举办的 Vue 主题会议：
 - VueConf 中国
 - Vue.js Amsterdam (欧洲)
 - VueConf US (美国)
 - Vue Fes Japan (日本)
 - VueConf Toronto (加拿大)
 - VueDay Italy (欧洲)
 - Vue Summit Brazil (南美)

良好的反馈

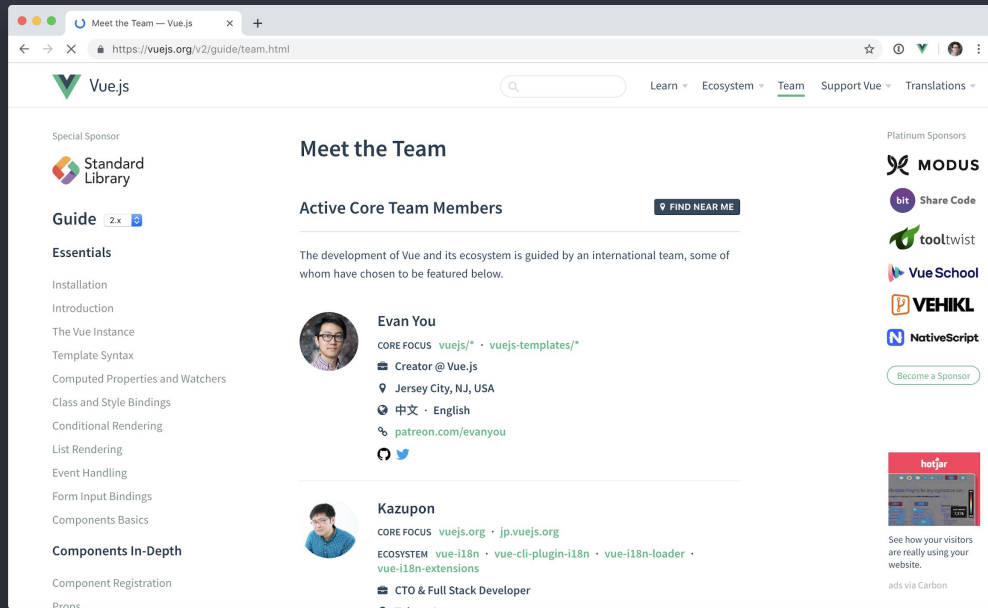
- State of JavaScript 2018 调查
 - 前端框架满意度第一 (91%)
- StackOverflow 2019 年度调查
 - Most Loved Web Frameworks 第二

使用公司遍布全球



团队

- 20 人的活跃核心团队，来自世界各地，大部分日常工作与 Vue 相关
- 独立运营，资金主要来源于赞助商，三年来稳步增长
- 蒋豪群 (@sodatea) 全职维护 CLI 及工具链



<https://vuejs.org/v2/guide/team.html>

3.0 进展

回顾 3.0 设计目标

- 更小
- 更快
- 加强 TypeScript 支持
- 加强 API 设计一致性
- 提高自身可维护性
- 开放更多底层功能

回顾 3.0 设计目标

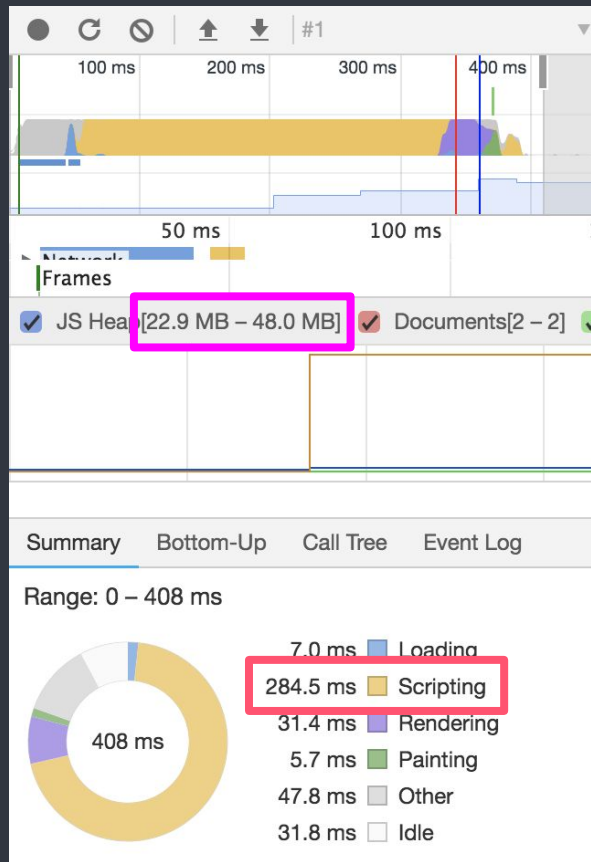
- 更小
- 更快
- 加强 TypeScript 支持
- 加强 API 设计一致性
- 提高自身可维护性
- 开放更多底层功能

更快

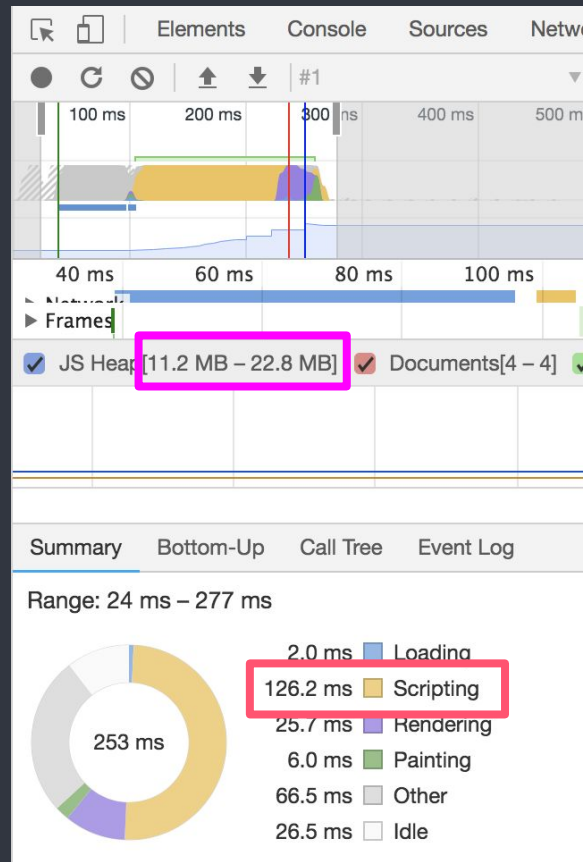
如何更快？

- Object.defineProperty -> Proxy
- Virtual DOM 重构
- 更多编译时优化
 - Slot 默认编译为函数
 - Monomorphic vnode factory
 - Compiler-generated flags for vnode/children types

v2.5



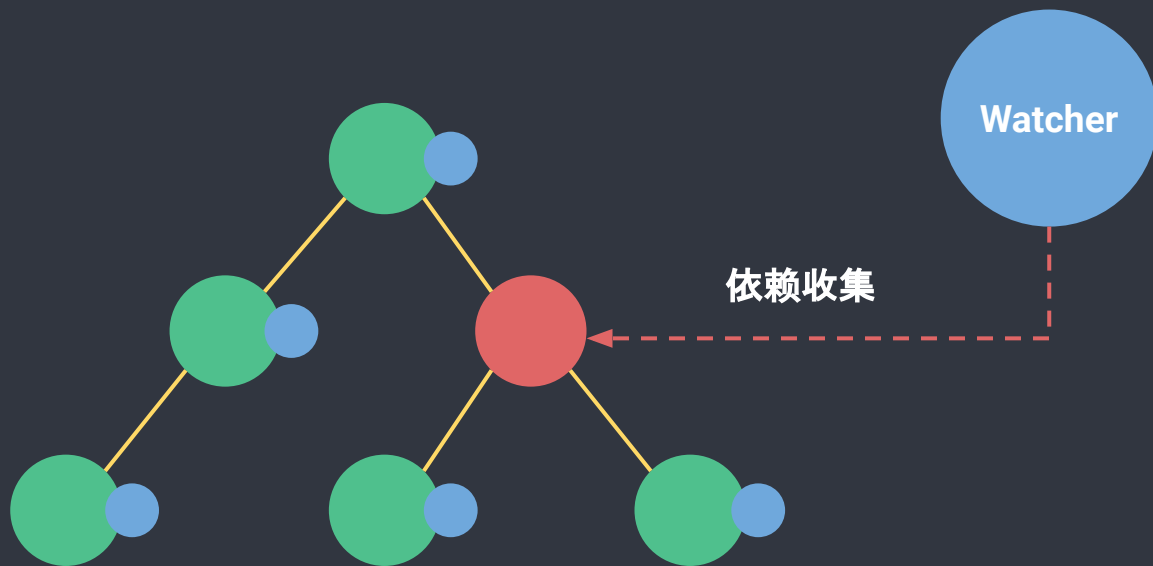
v3.0-proto



- 渲染 3000 个带状态的组件实例

还不够！

传统 vdom 的性能瓶颈



- 虽然 Vue 能够保证触发更新的组件最小化, 但在单个组件内部依然需要遍历该组件的整个 vdom 树

传统 vdom 的性能瓶颈

```
<template>
  <div id="content">
    <p class="text">Lorem ipsum</p>
    <p class="text">Lorem ipsum</p>
    <p class="text">{{ message }}</p>
    <p class="text">Lorem ipsum</p>
    <p class="text">Lorem ipsum</p>
  </div>
</template>
```

- Diff <div>
 - Diff props of <div>
 - Diff children of <div>
 - Diff <p>
 - Diff props of <p>
 - Diff children of <p>
 - Repeat n times...

- 传统 vdom 的性能跟模版大小正相关, 跟动态节点的数量无关。在一些组件整个模版内只有少量动态节点的情况下, 这些遍历都是性能浪费

传统 vdom 的性能瓶颈

```
function render() {  
  const children = []  
  for (let i = 0; i < 5; i++) {  
    children.push(h('p', {  
      class: 'text'  
    }, i === 2 ? this.message : 'Lorum ipsum'))  
  }  
  return h('div', { id: 'content' }, children)  
}
```

- 根本原因: JSX 和手写的 render function 是完全动态的, 过度的灵活性导致运行时可以用于优化的信息不足

限制越多，静态编译能够优化得就越多



```
<h1>Hello {name}!</h1>
```



```
p(changed, ctx) {  
  if (changed.name) {  
    set_data(t1, ctx.name);  
  }  
}
```

为什么不抛弃 Virtual DOM?

- 高级场景下手写 render function 获得更强的表达力
- 生成的代码更简洁
- 兼容 2.x

Vue 所特有的:

- 底层是 Virtual DOM
- 上层有包含大量静态信息的模版

Vue 需要做的:

- 兼容手写 render function
- 最大化利用模版静态信息

动静结合

最简单的情况

```
<template>
  <div id="content">
    <p class="text">Lorem ipsum</p>
    <p class="text">Lorem ipsum</p>
    <p class="text">{{ message }}</p>
    <p class="text">Lorem ipsum</p>
    <p class="text">Lorem ipsum</p>
  </div>
</template>
```

- Diff `<p> textContent`

- 节点结构完全不会改变
- 只有一个动态节点

节点结构变化:v-if

```
<template>
  <div>
    <p class="text">Lorem ipsum</p>
    <p v-if="ok">
      <span>Lorem ipsum</span>
      <span>{{ message }}</span>
    </p>
  </div>
</template>
```

- Check `<p v-if="ok">`
 - Diff `` `textContent`

- v-if 外部: 只有 v-if 是动态节点
- v-if 内部: 只有 `{{ message }}` 是动态节点

节点结构变化:v-if

```
<template>
<div>
  <p class="text">Lorem ipsum</p>
  <p v-if="ok">
    <span>Lorem ipsum</span>
    <span>{{ message }}</span>
  </p>
</div>
</template>
```

- Check `<p v-if="ok">`
 - Diff `` `textContent`

- v-if 外部: 只有 v-if 是动态节点
- v-if 内部: 只有 `{{ message }}` 是动态节点

节点结构变化: v-for

```
<template>
  <div>
    <p class="text">Lorem ipsum</p>
    <p v-for="item of list">
      <span>Lorem ipsum</span>
      <span>{{ item.message }}</span>
    </p>
  </div>
</template>
```

- Diff <p v-for> children
 - Diff textContent
 - Repeat n times...

- v-for 外部: 只有 v-for 是动态节点 (fragment)
- 每个 v-for 循环内部: 只有 {{ item.message }} 是动态节点

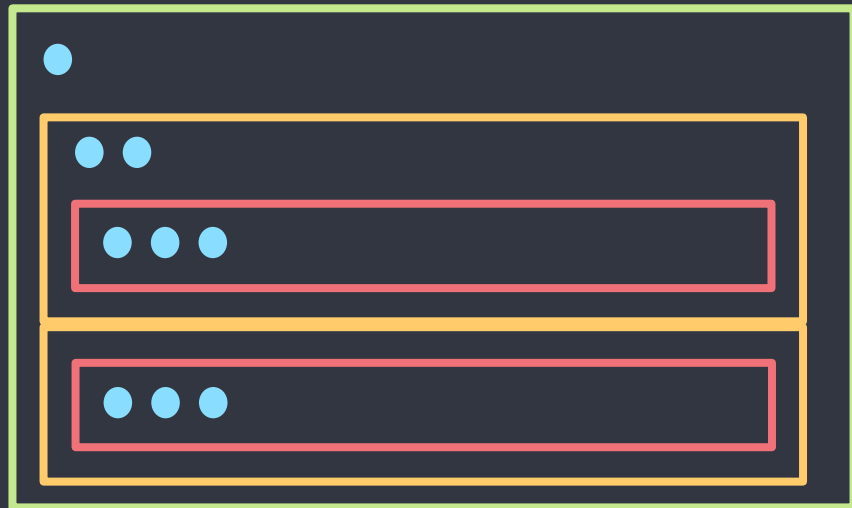
节点结构变化: v-for

```
<template>
<div>
  <p class="text">Lorem ipsum</p>
  <p v-for="item of list">
    <span>Lorem ipsum</span>
    <span>{{ item.message }}</span>
  </p>
</div>
</template>
```

- Diff <p v-for> children
 - Diff textContent
 - Repeat n times...

- v-for 外部: 只有 v-for 是动态节点 (fragment)
- 每个 v-for 循环内部: 只有 {{ item.message }} 是动态节点

Block tree “区块树”



- 将模版基于动态节点指令切割为嵌套的区块
- 每个区块内部的节点结构是固定的
- 每个区块只需要以一个 Array 追踪自身包含的动态节点

Before

```
<template>
  <div id="content">
    <p class="text">Lorem ipsum</p>
    <p class="text">Lorem ipsum</p>
    <p class="text">{{ message }}</p>
    <p class="text">Lorem ipsum</p>
    <p class="text">Lorem ipsum</p>
  </div>
</template>
```

- Diff <div>
 - Diff props of <div>
 - Diff children of <div>
 - Diff <p>
 - Diff props of <p>
 - Diff children of <p>
 - Repeat n times...

- 新策略将 vdom 更新性能由与模版整体大小相关提升为与动态内容的数量相关

After

```
<template>
  <div id="content">
    <p class="text">Lorem ipsum</p>
    <p class="text">Lorem ipsum</p>
    <p class="text">{{ message }}</p>
    <p class="text">Lorem ipsum</p>
    <p class="text">Lorem ipsum</p>
  </div>
</template>
```

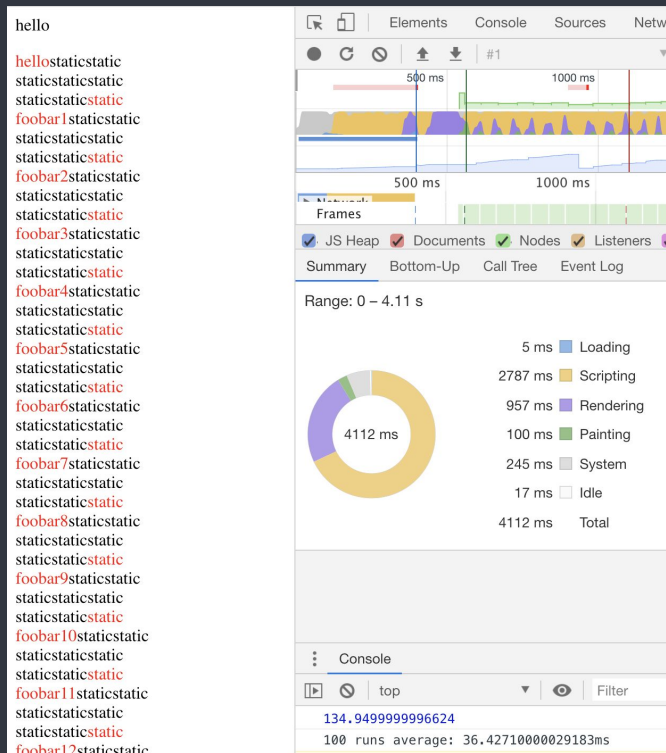
- Diff `<p> textContent`

- 新策略将 vdom 更新性能由与模版整体大小相关提升为与动态内容的数量相关

Update performance benchmark

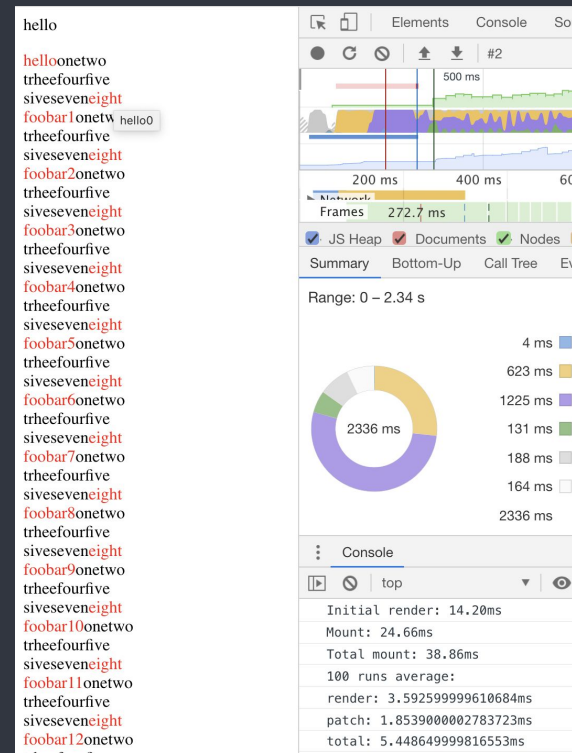
- 1000 个循环的 v-for
- 每个循环包含
 - 三层嵌套共 12 个 DOM 元素
 - 2 个动态 class 绑定
 - 1 个动态文字绑定
 - 1 个动态 id 属性绑定
- 动态更新所有绑定 100 次取平均值

2.6.10



100 runs average: **36ms**

3.0-proto-2019-06



100 runs average: **5.44ms**

36ms → **5.44ms**

More than 6x faster
(in this benchmark)

TypeScript

Class API

~~Class API~~

CANCELLED

为什么撤销 Class API

- 原本目的是更好的 TS 支持
 - Props 和其它需要注入到 this 的属性导致类型声明依然存在问题
 - Decorators 提案的严重不稳定使得依赖它的方案具有重大风险
- 除了类型支持以外 Class API 并不带来任何新的优势
 - 其实我们有更好的选择...

Function-based API

```
const App = {  
  setup() {  
    // data  
    const count = value(0)  
    // computed  
    const plusOne = computed(() => count.value + 1)  
    // method  
    const increment = () => { count.value++ }  
    // watch  
    watch(() => count.value * 2, v => console.log(v))  
    // lifecycle  
    onMounted(() => console.log('mounted!'))  
    // 暴露给模版或渲染函数  
    return { count }  
  }  
}
```

Function-based API



- 对比 Class API
 - 更好的 TypeScript 类型推导支持
 - 更灵活的逻辑复用能力
 - Tree-shaking 友好
 - 代码更容易被压缩

逻辑复用案例：鼠标位置侦听

Mixins

```
const mousePositionMixin = {  
  data() {  
    return {  
      x: 0,  
      y: 0  
    }  
  },  
  mounted() {  
    window.addEventListener('mousemove', this.update)  
  },  
  destroyed() {  
    window.removeEventListener('mousemove', this.update)  
  },  
  methods: {  
    update(e) {  
      this.x = e.pageX  
      this.y = e.pageY  
    }  
  }  
}
```

Mixins

- 当大量使用时:
 -  命名空间冲突
 -  模版数据来源不清晰

Higher-order Components

高阶组件




```
const Demo = withMousePosition({  
  props: ['x', 'y'],  
  template: `

Mouse position: x {{ x }} / y {{ y }}

`  
})
```

Higher-order Components

高阶组件

- 大量使用时:
 -  props 命名空间冲突
 -  props 来源不清晰
 -  额外的组件实例性能消耗

Renderless Components

作用域插槽

```
<mouse v-slot="{ x, y }">  
  Mouse position: x {{ x }} / y {{ y }}  
</mouse>
```

Renderless Components

作用域插槽

- ✓ 没有命名空间冲突
- ✓ 数据来源清晰
- ✗ 额外的组件实例性能消耗

With new API

```
function useMousePosition() {  
  const x = value(0)  
  const y = value(0)  
  
  const update = e => {  
    x.value = e.pageX  
    y.value = e.pageY  
  }  
  
  onMounted(() => {  
    window.addEventListener('mousemove', update)  
  })  
  
  onUnmounted(() => {  
    window.removeEventListener('mousemove', update)  
  })  
  
  return { x, y }  
}
```

With new API

```
new Vue({
  template: `
    <div>
      Mouse position: x {{ x }} / y {{ y }}
    </div>
  `,
  data() {
    const { x, y } = useMousePosition()
    return {
      x,
      y,
      // ... other data
    }
  }
})
```


- ✓ 没有命名空间冲突
- ✓ 数据来源清晰
- ✓ 没有额外的组件性能消耗

对比 React Hooks

- 同样的逻辑组合、复用能力
- 只调用一次
 - 符合 JS 直觉
 - 没有闭包变量问题
 - 没有内存/GC 压力
 - 不存在内联回调导致子组件永远更新的问题

更多详情请参见 RFC

<https://github.com/vuejs/rfcs>