Vue reactivity system 演进

Doctor Wu / 2025-07-12

Doctor Wu

Vue VueUse 核心团队成员

目前就职于 MoeGo

doctorwu.me Doctor-wu Doctorwu666 Doctor___Wu

Challenges in Reactive system

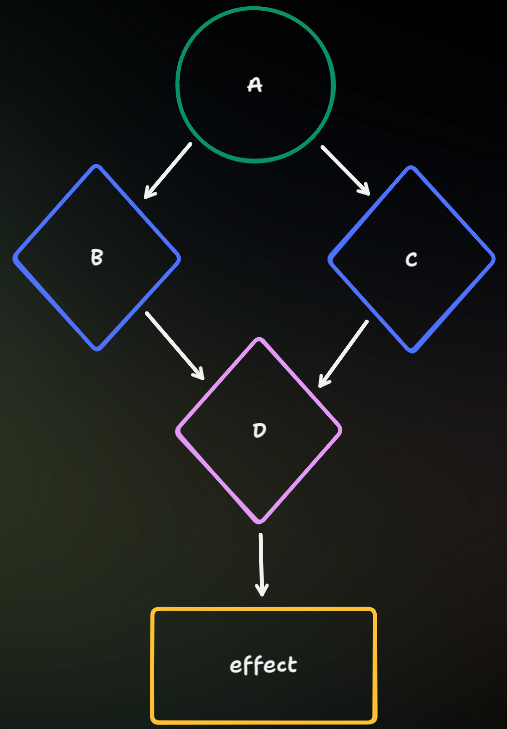
响应式系统的挑战

Glitch

毛刺

经典 Glitch 问题

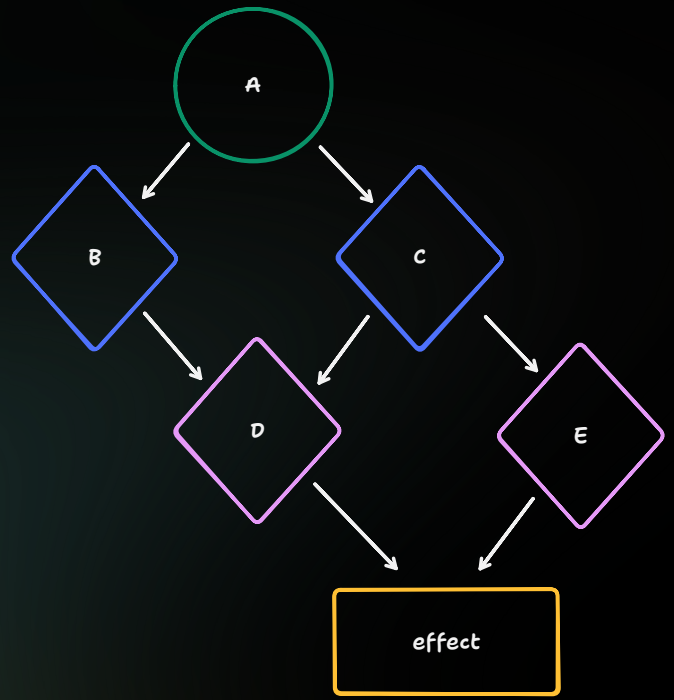
```
const A = ref(1);
const B = computed(() => A.value * 2); // B is now: 2
const C = computed(() => A.value + 1); // C is now: 2
const D = computed(() => B.value + C.value); // D is now: 4

watchEffect(() => {
  console.log('D is now:', D.value); // D is now: 4
});
```



Scheduler

所以, 我们已经安全了吗?

```
const A = ref(1);
const B = computed(() => A.value * 2);
const C = computed(() => A.value + 1);
const D = computed(() => B.value + C.value);

watchEffect(() => {
  console.log('D is now:', D.value);
});
```



TLDRAW

Oops, effect 怎么跑了两次

feat(reactivity): more efficient reactivity system

Effect 支持脏值检查

```ts
const update: SchedulerJob = (instance.update = () => {
  if (effect.dirty) {
    effect.run()
  }
})
```

```ts
public get dirty() {
  if (
    this._dirtyLevel === DirtyLevels.MaybeDirty
  ) {
    // check if effect is dirty
  }
  return this._dirtyLevel >= DirtyLevels.Dirty
}
```

Equality check problem

相等性检查

```
const A = ref(3);
const B = computed(() => A.value * 0); // always 0
const C = computed(() => B.value + 1);

watchEffect(() => {
  console.log('C is now:', C.value);
});
```

染色算法

```
const A = ref(3);
const B = computed(() => A.value * 0); // always 0
const C = computed(() => B.value + 1);

watchEffect(() => {
  console.log('C is now:', C.value);
});
```

```
let sec_counter = 0
let min_counter = 0
let hour_counter = 0

const ms = ref(0)
const sec = computed(() => { sec_counter++; return Math.floor(ms.value / 1000) })
const min = computed(() => { min_counter++; return Math.floor(sec.value / 60) })
const hour = computed(() => { hour_counter++; return Math.floor(min.value / 60) })

for (ms.value = 0;  ms.value < 10000000; ms.value += 1) {
  hour.value
}
```

Memory usage

内存占用优化

基于 Set 的数据结构

```
// 全局的 WeakMap，存储所有响应式对象的依赖
// target -> depsMap
const targetMap = new WeakMap();

// 当前正在执行的 effect
let activeEffect = null;

// effect 的结构
class ReactiveEffect {
  constructor(fn) {
    this.fn = fn;
    // deps 是一个数组，存储了所有包含此 effect 的 Set (dep)
    this.deps = [];
  }
  run() { /* ... */ }
}
```

track 依赖收集

```
function track(target, key) {
  if (!activeEffect) return;
  let depsMap = targetMap.get(target);

  if (!depsMap) {
    depsMap = new Map();
    targetMap.set(target, depsMap);
  }
  let dep = depsMap.get(key);

  if (!dep) {
    // *** 核心痛点 1: 每次都可能创建一个新的 Set 实例 ***
    dep = new Set();
    depsMap.set(key, dep);
  }

  dep.add(activeEffect);

  activeEffect.deps.push(dep);
}
```

trigger 触发更新

```
function trigger(target, key) {
  const depsMap = targetMap.get(target);
  if (!depsMap) return;

  const dep = depsMap.get(key);
  if (!dep) return;

  // 遍历 Set，执行所有依赖的 effect
  dep.forEach(effect => {
    effect.run(); // 重新执行副作用函数
  });
}
```

cleanupEffect 依赖清理

```
function cleanupEffect(effect) {
  // *** 核心痛点 2: 这是一个 O(N) 操作，N 是依赖项数量 ***
  for (const dep of effect.deps) {
    // 从每一个 dep (Set) 中删除自己
    dep.delete(effect);
  }
  // 清空自己的 deps 数组
  effect.deps.length = 0;
}
```

Doubly-linked list

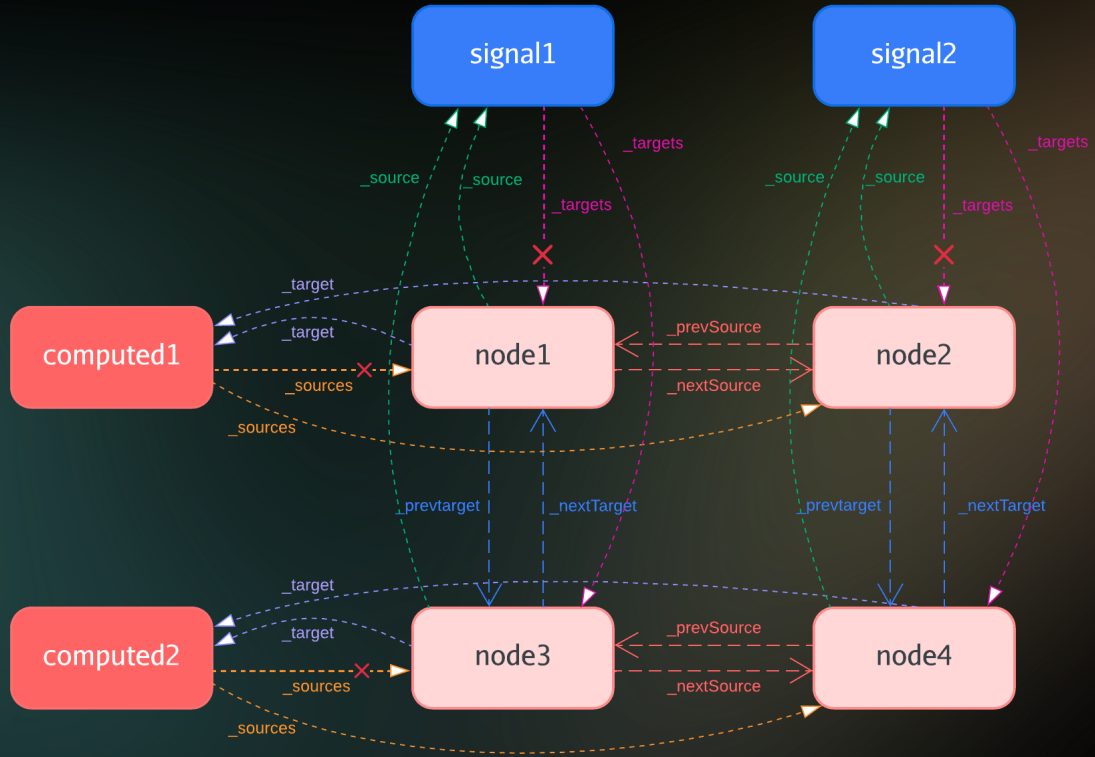
双向链表

基于双向链表的数据结构

```
// targetMap 和 depsMap 结构保持不变
const targetMap = new WeakMap();
let activeEffect = null;

// effect 的结构发生了变化
class ReactiveEffect {
  constructor(fn) {
    this.fn = fn;
    // this.deps 不再需要了

    // *** 新增的链表指针 ***
    // 插入删除时间复杂度更低，O(1)
    this.next = null; // 指向链表中的下一个 effect
    this.prev = null; // 指向链表中的上一个 effect
  }
  run() { /* ... */ }
}
```

track 依赖收集

```
function track(target, key) {
  // ...

  // dep 不再是 Set，而是一个指向链表头的 Link 指针
  // dep 现在可能是 undefined 或一个 Link
  let dep = depsMap.get(key);

  // *** 核心优化点：O(1) 的链表头插入操作 ***
  // 将 activeEffect 链接到 dep 链表的头部
  if (dep !== activeEffect) { // 防止重复链接
      // 新 effect 的 next 指向旧的链表头
      activeEffect.next = dep;
      if (dep) {
          // 旧的链表头 prev 指向新 effect
          dep.prev = activeEffect;
      }
      // 更新 map，让 dep 指向新的链表头 (activeEffect)
      depsMap.set(key, activeEffect);
  }
}
```



图片来自 soonwang

Change propagation algorithms

变化传播算法

Push-based (Eager) propagation

基于 "推" 的传播算法, 当值变化时, 立即通知消费者。
这些通知一般包含所有必要信息, 消费者不需要再查询额外信息。

Pull-based (Lazy) propagation

基于 "拉" 的传播算法, 消费者定期 (或按需) 查询源以获取值。

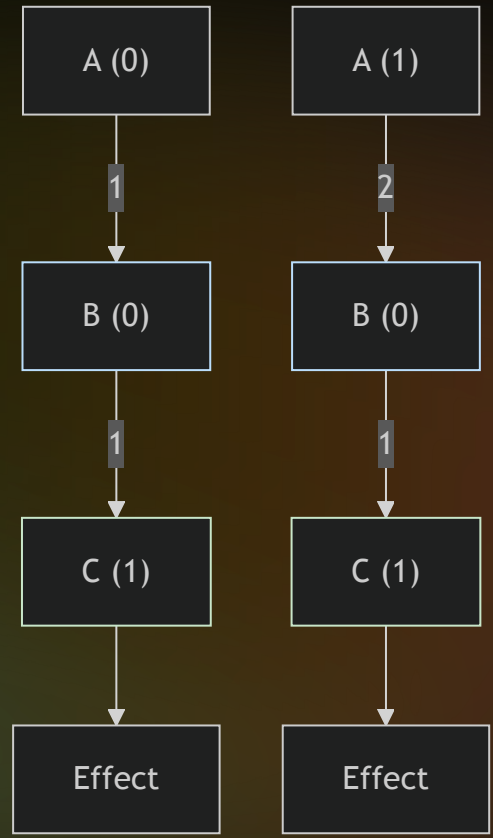
Push-pull-based propagation

结合 "推" 和 "拉" 的传播算法。
当值变化时, 先发送轻量级的通知 然后消费者根据通知请求具体信息。

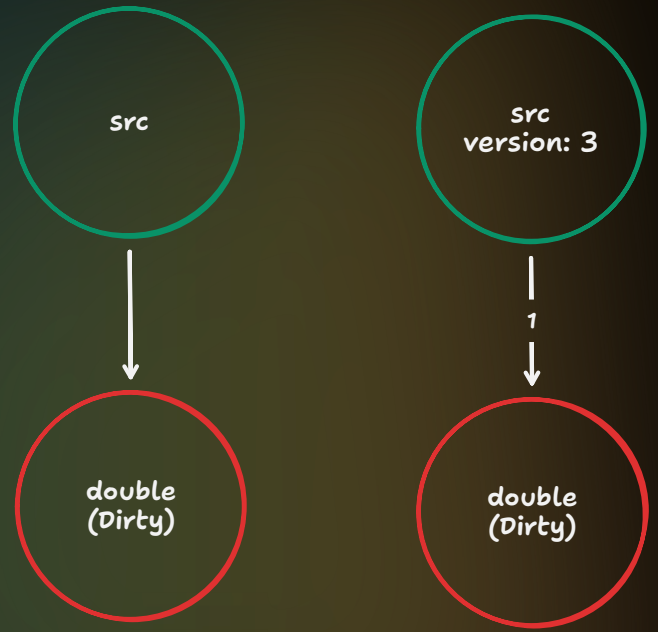
Version counting

版本计数

```
const A = ref(0);
const B = computed(() => A.value * 0); // always 0
const C = computed(() => B.value + 1);

watchEffect(() => {
  console.log('C is now:', C.value);
});
```

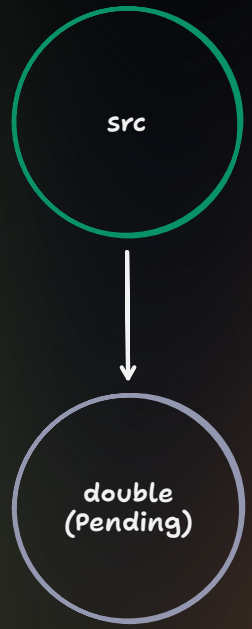


```
const src = signal(10);
const double = computed(() => src() * 2);

double(); // -> 20

src(999); // double.flags -> Dirty
src(10);  // double.flags -> Dirty

double(); // propagate (unnecessary)
```

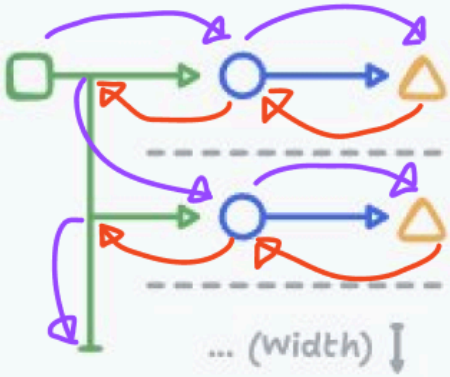


更加极致的染色算法

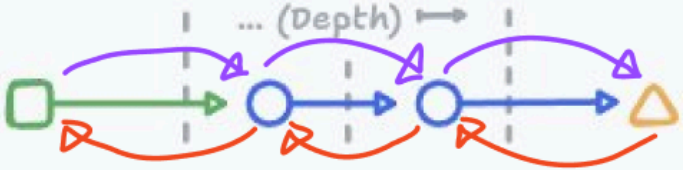
```
const src = signal(10);
const double = computed(() => src() * 2);

double(); // -> 20

src(999); // double.flags -> Dirty
src(10);  // double.flags -> Dirty

double(); // propagate (unnecessary)
```



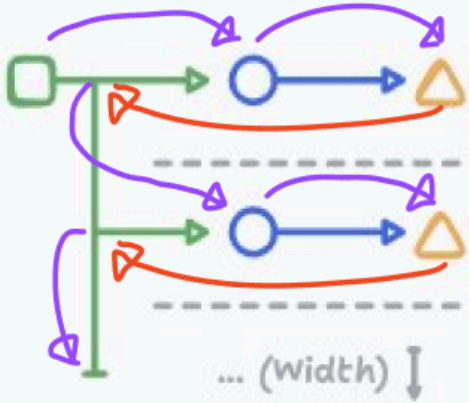
Broad



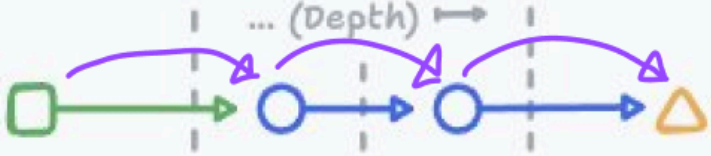
Deep



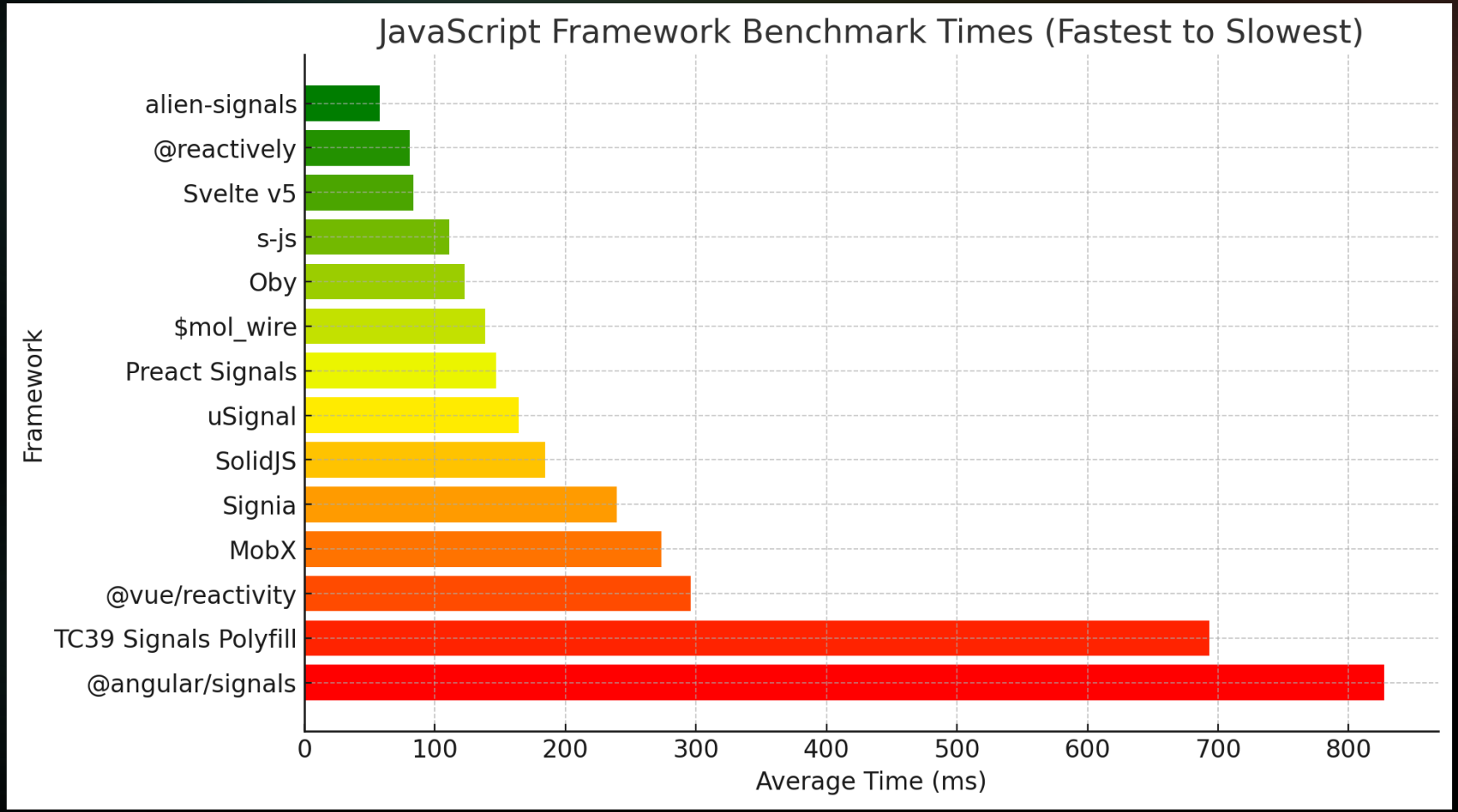
Broad



Deep



```
const foo = { a: 1, foo: 1 }
const bar = { a: 1, bar: 1}

funcA(foo);
funcA(bar);
funcB(foo, foo.a);
funcB(bar, bar.a);

function funcA(obj) {
    const a = obj.a;
    ...
}
function funcB(obj, a) {
    ...
}
```


JavaScript Framework Benchmark Times (Fastest to Slowest)
Framework
alien-signals
@reactively
Svelte v5
s-js
Oby
$mol_wire
Preact Signals
uSignal
SolidJS
Signia
MobX
@vue/reactivity
TC39 Signals Polyfill
@angular/signals
0
100
200
300
400
500
600
700
800
Average Time (ms)

Huge thanks to

Thank you!

Created using  Slidev