MiniVue — Vue 3 своими руками

Строим фреймворк с нуля: реактивность, virtual DOM, компилятор, компоненты, роутер, стор и SSR

Серик Мурадов

Содержание

Зачем строить Vue заново **1**
Что мы построим . . . 1
Правила игры: без сборки . . . 2
Как устроена книга и репозиторий . . . 2
Для кого это . . . 2

Минимум JavaScript **3**
Значения и переменные . . . 3
Функции . . . 3
Объекты . . . 4
Массивы . . . 4
Деструктуризация и spread . . . 5
Классы, геттеры и сеттеры . . . 5
Set, Map и WeakMap . . . 6
Symbol . . . 6
Proxy и Reflect . . . 7
Object.is . . . 7
Модули: import и export . . . 8
optional chaining и `!!` . . . 8

Реактивность **9**
Задача в одном предложении . . . 9
Два глагола: track и trigger . . . 9
Эффект — это функция, которую можно перезапустить . . . 10
Где живут связи: targetMap . . . 11
reactive: перехват объекта через Proxy . . . 14
ref: реактивность для одного значения . . . 16
computed: лениво и с кэшем . . . 17
Почему нужен cleanup: задача с ветвлением . . . 18
watch: следить и получать старое/новое . . . 19
Как всё это работает вместе . . . 20
Проверяем себя . . . 20

Virtual DOM и рендерер **22**
Идея: сначала описать, потом сравнить . . . 22
VNode: узел как объект . . . 22
Рендерер не знает про браузер . . . 23
patch: сравнить старое и новое . . . 24

Обновление на месте . . . 25
Сравнение списков по ключам . . . 25
Грабля: unmount обязан обойти всё поддерево . . . 26
Как это связано с реактивностью . . . 26
Проверяем себя . . . 26

Компоненты **28**
Что такое компонент . . . 28
Инстанс: личное дело компонента . . . 29
setup и публичный контекст . . . 29
Связь с реактивностью: render как эффект . . . 30
Планировщик: три изменения — одна перерисовка . . . 31
Грабля: задача, ставящая в очередь саму себя . . . 32
props: данные сверху вниз . . . 32
emit: события снизу вверх . . . 33
slots: разметка снаружи . . . 34
provide / inject: сквозь этажи . . . 34
Жизненный цикл . . . 35
createApp: точка входа . . . 35
Грабля: перемещение keyed-компонента требует его `el` . . . 35
Проверяем себя . . . 36

Компилятор шаблонов **37**
Три шага перевода . . . 37
Парсинг: из текста в дерево . . . 37
Грабля: парсер обязан двигаться вперёд . . . 39
Генерация: из дерева в код . . . 39
Директивы . . . 40
Из строки в функцию: with(ctx) . . . 41
Как компилятор подключается к рантайму . . . 41
Что мы упростили . . . 42
Проверяем себя . . . 42

Router **43**
Идея: адрес → компонент . . . 43
history: где хранится адрес . . . 43
Матчер: разбор пути с параметрами . . . 44
Реактивный текущий маршрут . . . 44
Навигация и guard'ы . . . 45
RouterView и RouterLink . . . 46
Подключение к приложению . . . 47
Что мы упростили . . . 48
Проверяем себя . . . 48

Store **49**
Стор — это снова реактивность . . . 49
createPinia: контейнер сторов . . . 49
defineStore: два стиля . . . 50

Один путь создания . . . 51
proxyRefs: почему store.count, а не store.count.value . . . 51
storeToRefs: деструктуризация без потери реактивности . . . 52
Плагины . . . 52
Что мы упростили . . . 52
Проверяем себя . . . 53

SSR и гидратация **54**
Рендеринг в строку . . . 54
createSSRApp . . . 55
Сервер целиком . . . 56
Проблема: HTML без жизни . . . 56
hydrate: усыновление DOM . . . 56
Грабля: браузер склеивает соседние текстовые узлы . . . 58
Что мы упростили . . . 58
Проверяем себя . . . 58
Итог . . . 58

Формы и привязки **59**
Привязка class и style . . . 59
v-model: двусторонняя связь . . . 60
Модификаторы событий . . . 60
Что мы упростили . . . 62
Проверяем себя . . . 62

Расширения реактивности **63**
watchEffect . . . 63
readonly . . . 64
shallowReactive и shallowRef . . . 65
markRaw . . . 65
Итог . . . 65

Директивы и динамические компоненты **66**
Кастомные директивы . . . 66
Как это устроено . . . 66
Динамические компоненты . . . 67
v-model на компонентах . . . 68
Что мы упростили . . . 68
Проверяем себя . . . 69

Встроенные компоненты **70**
Teleport: рисуем в другом месте . . . 70
KeepAlive: не разрушать, а прятать . . . 71
Побочно найденный баг слотов . . . 72
Асинхронные компоненты . . . 73
Что мы упростили . . . 74
Проверяем себя . . . 74

Расширения стора и итоговое приложение **75**

$patch, $subscribe, $reset 75
Итоговое приложение 76
Ещё два приложения — и баг, который поймал магазин 76
Что мы прошли 77
Что дальше 77
Проверяем себя 78

Зачем строить Vue заново

Есть два способа понять инструмент. Первый — прочитать инструкцию и начать им пользоваться. Второй — разобрать его на части и собрать самому. Эта книга про второй способ. Мы не будем учиться писать приложения на Vue — мы напишем сам Vue.

Звучит страшно, но пугаться нечего. Vue внутри — это не магия, а несколько простых идей, аккуратно сложенных друг на друга. Реактивность («данные изменились — экран обновился сам») — это перехват чтения и записи. Компоненты — это функции, которые возвращают описание разметки. Роутер — это словарь «адрес → что показать». Каждую из этих идей можно уместить в сотню-другую строк, которые читаются как обычный текст. Мы пройдём их все.

Что мы построим

К концу книги у нас будет работающий мини-фреймворк со всей экосистемой настоящего Vue 3:

- **Реактивность** — `ref`, `reactive`, `computed`, `watch`. Ядро, из-за которого Vue вообще существует.
- **Virtual DOM и рендерер** — как описание интерфейса превращается в реальные элементы страницы, и как обновлять их точечно, а не перерисовывать всё.
- **Компоненты** — `setup`, входные параметры (props), события (emit), слоты, хуки жизненного цикла, `createApp`.
- **Компилятор шаблонов** — как `<div>{{ count }}</div>` превращается в функцию.
- **Роутер** — аналог Vue Router: адреса, параметры, `RouterView`, `RouterLink`.
- **Стор** — аналог Pinia: общее состояние приложения.
- **SSR** — рендеринг на сервере и «оживление» (гидратация) на клиенте.

Имена мы намеренно берём такие же, как в настоящем Vue. Наш `ref` называется `ref`, наш `createApp` называется `createApp`. Поэтому всё, что вы поймёте здесь, один в один переносится на реальный фреймворк — только там кода больше, потому что он обрабатывает пограничные случаи, которые мы иногда будем опускать ради ясности.

Правила игры: без сборки

Обычно, чтобы запустить Vue-проект, ставят Node, потом сборщик (Vite или Webpack), и он «магически» превращает ваши файлы в то, что понимает браузер. Для новичка это худшее начало: половина происходящего спрятана внутри инструмента, который вы ещё не понимаете.

Поэтому у нас правило: **никакой сборки**. Мы пишем на чистых ES-модулях — стандарте, который браузер понимает сам. Файл с кодом подключается к странице строчкой `<script type="module">`, а внутри работает `import`. Открыли страницу — код поехал. Всё, что происходит, происходит у вас на глазах, без посредников.

Node.js нам понадобится ровно для двух вещей: запускать тесты и поднимать крошечный локальный сервер, чтобы браузер разрешил загрузку модулей (со схемы `file://` он их из соображений безопасности не грузит). Сам фреймворк от Node не зависит — до главы про SSR, где сервер и есть суть.

Как устроена книга и репозиторий

Каждый слой — это одновременно:

- **глава** здесь, в книге, где идея объясняется словами и картинками;
- **папка в `packages/`** с рабочим кодом, обильно закомментированным на русском;
- **тесты в `test/`**, которые доказывают, что код делает то, что обещано;
- **демо в `playground/`**, которое можно открыть в браузере и потрогать руками.

Советую держать рядом код и главу: прочитали объяснение — открыли файл, нашли те самые строки. Код в книге и код в репозитории — это буквально одни и те же строки.

Для кого это

Для того, кто хочет понимать, а не только применять. Если вы никогда не писали на JavaScript — не страшно: следующая глава даёт ровно тот минимум языка, который нужен, чтобы читать наш код. Если вы уже пишете на Vue, но он для вас «чёрный ящик» — эта книга ящик открывает.

Одно предупреждение: не пытайтесь запомнить всё сразу. Реактивность в первой главе поначалу кажется головоломкой — это нормально. Дойдите до демо, потыкайте кнопки, вернитесь к коду. Идея встанет на место. А когда встанет — всё остальное во Vue станет очевидным следствием.

Поехали.

Минимум JavaScript

Эта глава — не полноценный курс языка, а словарь. Здесь ровно те конструкции, которые встретятся в нашем коде, и ни одной лишней. Если вы уже знаете JavaScript, пролистайте её и переходите к реактивности. Если нет — прочитайте один раз, а дальше возвращайтесь как к справочнику, когда в коде встретится незнакомый значок.

Значения и переменные

Значение — это кусочек данных: число `42`, строка `"привет"`, логическое `true` или `false`, «ничего» — `null` и `undefined`. Переменная — это имя для значения.

```
const name = 'Аня'   // const — имя, которое нельзя переназначить
let count = 0        // let — имя, которое можно менять
count = count + 1    // теперь count равно 1
```

`const` не значит «значение неизменно» — он значит «нельзя присвоить это имя заново». Объект под именем `const` менять можно; нельзя лишь сказать `имя = другой объект`. По умолчанию берите `const`, а `let` — только когда правда нужно переприсваивать.

Функции

Функция — это заготовка действий, которую можно запускать много раз, подставляя разные входные значения (аргументы).

```
function double(x) {
  return x * 2
}
double(21) // 42
```

Есть короткая запись — стрелочная функция. Она нам будет попадаться постоянно.

```
const double = (x) => x * 2       // одно выражение — return подразумевается
const log = (msg) => {            // тело в фигурных скобках — return пишем сами
  console.log(msg)
}
const now = () => Date.now()      // без аргументов — пустые скобки
```

Функцию можно передать другой функции как аргумент — это и есть «колбэк», функция-обработчик. Вся реактивность построена на том, что мы принимаем чужую функцию и решаем, когда её вызвать:

```
effect(() => {
  console.log('перезапусти меня, когда данные изменятся')
})
```

Объекты

Объект — набор пар «ключ: значение». Ключи (свойства) читают и пишут через точку.

```
const user = { name: 'Аня', age: 30 }
user.name       // 'Аня'  — чтение
user.age = 31   // запись
user.city = 'Алматы' // добавление нового свойства
```

Запомните слова **чтение свойства** и **запись свойства** — вокруг них крутится вся первая глава. Реактивность — это перехват именно этих двух операций.

Массивы

Массив — упорядоченный список значений. Индексация с нуля.

```
const items = [10, 20, 30]
items[0]        // 10
items.length    // 3
items.push(40)  // добавить в конец → [10, 20, 30, 40]
items.reduce((sum, x) => sum + x, 0) // 100 — «свернуть» в одно число
```

`reduce` идёт по элементам и накапливает результат: начинаем с `0`, к каждому элементу прибавляем его значение. Так мы будем считать сумму корзины.

Деструктуризация и spread

Можно «разобрать» объект или массив на отдельные переменные:

```
const { name, age } = user       // name = 'Аня', age = 31
const [first, second] = items    // first = 10, second = 20
```

Три точки `...` — «разложи остальное». В нашем коде она встречается как «сделать копию»:

```
const copy = [...items]          // новый массив с теми же элементами
```

Эта строчка спасёт нас в реактивности: перед перебором набора эффектов мы копируем его, чтобы изменения во время перебора не сломали цикл.

Классы, геттеры и сеттеры

Класс — шаблон для создания объектов с общим поведением. `new` создаёт экземпляр, `constructor` задаёт начальное состояние, `this` внутри — «этот конкретный экземпляр».

```
class Counter {
  constructor(start) {
    this.value = start   // поле экземпляра
  }
  increment() {
    this.value++
  }
}
const c = new Counter(5)
c.increment()
c.value // 6
```

Особая пара — **геттер** и **сеттер**. Это методы, которые снаружи выглядят как обычное свойство, но при чтении/записи выполняют код. Именно так устроен `ref`: обращение к `.value` выглядит как поле, а на деле запускает функцию.

```
class Box {
  constructor(v) { this._v = v }
  get value() {           // сработает при ЧТЕНИИ box.value
    console.log('прочитали')
    return this._v
  }
  set value(newV) {       // сработает при ЗАПИСИ box.value = ...
```

```
    console.log('записали')
    this._v = newV
  }
}
const box = new Box(1)
box.value        // печатает «прочитали», возвращает 1
box.value = 2    // печатает «записали»
```

Подчёркивание в `_v` — просто договорённость «это внутреннее поле, снаружи не трогай». Язык его не защищает, но так принято.

Set, Map и WeakMap

Кроме массивов и объектов нам нужны три специальные коллекции.

Set — набор уникальных значений. Добавили дважды — хранится один раз. Идеально для «списка эффектов, подписанных на свойство»: один эффект не должен попасть туда дважды.

```
const s = new Set()
s.add('a'); s.add('a'); s.add('b')
s.size          // 2
s.has('a')      // true
s.delete('b')
```

Map — как объект, но ключом может быть что угодно, включая другой объект. Нам это критично: мы будем связывать реактивный объект с его зависимостями.

```
const m = new Map()
m.set('ключ', 123)
m.get('ключ')   // 123
```

WeakMap — то же, что Map, но со «слабыми» ключами-объектами: если на объект-ключ больше никто не ссылается, сборщик мусора вправе удалить его вместе с записью. Так хранилище зависимостей не мешает освобождать память от объектов, которые больше не нужны.

Symbol

`Symbol('имя')` создаёт уникальное значение-метку, которое гарантированно не совпадёт ни с каким строковым ключом. Мы используем символы как служебные «скрытые» свойства —

например, метку «этот объект уже реактивный», которую не спутать с настоящими данными пользователя.

```
const IS_REACTIVE = Symbol('isReactive')
obj[IS_REACTIVE] // никакое обычное свойство сюда не попадёт случайно
```

Proxy и Reflect

Вот главный инструмент реактивности. **Proxy** оборачивает объект и позволяет перехватить операции над ним — в первую очередь чтение (`get`) и запись (`set`).

```
const raw = { count: 0 }
const proxy = new Proxy(raw, {
  get(target, key) {
    console.log('читают', key)
    return target[key]
  },
  set(target, key, value) {
    console.log('пишут', key, '=', value)
    target[key] = value
    return true   // set обязан вернуть true при успехе
  },
})
proxy.count       // печатает «читают count», возвращает 0
proxy.count = 5   // печатает «пишут count = 5»
```

Из этого крошечного перехвата и вырастает `reactive`: в `get` мы будем вызывать «запомни, кто это прочитал», а в `set` — «разбуди тех, кто это читал».

Reflect — набор функций, повторяющих стандартные операции. `Reflect.get(target, key, receiver)` — то же, что `target[key]`, но корректно передаёт `this` в геттеры и работает с наследованием. Внутри Proxy правильно использовать именно `Reflect`, а не `target[key]` напрямую — иначе на хитрых объектах будут ошибки.

Object.is

Сравнение «изменилось ли значение». Обычный `===` почти везде подходит, но у него есть исторические странности: `NaN === NaN` даёт `false`, хотя это «то же самое». `Object.is` их чинит. Мы будем спрашивать «значение действительно другое?» через `!Object.is(старое, новое)`, чтобы не будить интерфейс на присваивании того же самого.

Модули: import и export

Код разбит на файлы-модули. Файл делится тем, что он `export`, а забирает чужое через `import`.

```
// effect.js
export function track() { /* ... */ }
export let activeEffect = undefined

// reactive.js
import { track, activeEffect } from './effect.js'
```

Путь пишется с расширением `.js` и с `./` для соседних файлов — так требует браузерный стандарт ES-модулей, которым мы пользуемся без сборщика.

optional chaining и `!!`

`a?.b` читает `a.b`, но если `a` — `null` или `undefined`, спокойно возвращает `undefined` вместо ошибки. `!!x` превращает любое значение в честный `true/false` (два «не»: первое инвертирует, второе возвращает обратно). Встретите `return !!(value && value.__isRef)` — это просто «верни true, если у value есть флаг __isRef».

Этого словаря хватит, чтобы прочитать любую строчку нашего фреймворка. Не запоминайте всё — держите главу под рукой и возвращайтесь по мере надобности. А теперь — к сердцу Vue.

Реактивность

Всё во Vue держится на одной способности: данные изменились — то, что от них зависит, обновилось само. Вы меняете `count`, и число на экране становится другим, хотя вы нигде не писали «найди элемент и обнови текст». Эта глава — про то, как это устроено. Мы построим реактивность целиком, и дальше весь фреймворк будет лишь надстройкой над ней.

Код этой главы живёт в `packages/reactivity/`. Тесты — в `test/reactivity.test.mjs`, живое демо — `playground/01-reactivity.html`.

Задача в одном предложении

Мы хотим уметь писать так:

```
const count = ref(0)

effect(() => {
  document.title = 'Кликов: ' + count.value
})

count.value = 5   // и заголовок вкладки сам стал «Кликов: 5»
```

Мысль простая. Есть данные (`count`) и есть функция, которая их использует (`effect`). Мы хотим, чтобы при изменении данных функция запустилась заново — сама, без нашего участия. Значит, системе нужно как-то узнать, что эта функция «зависит» от `count`. А узнать это можно ровно в один момент — когда функция читает `count.value`. Вокруг этой мысли строится всё.

Два глагола: track и trigger

Запомните два слова, дальше книга крутится вокруг них.

- **track** («отследить») — происходит при **чтении** данных. Смысл: «сейчас выполняется какая-то функция, и она прочитала вот это значение — запомним связь».
- **trigger** («запустить») — происходит при **записи** данных. Смысл: «значение изменилось — найдём все функции, которые его читали, и запустим их заново».

Читаем — запоминаем связь. Пишем — используем связь. Всё остальное — детали хранения этих связей.

Эффект — это функция, которую можно перезапустить

Функцию, которую надо перезапускать при изменениях, мы называем **эффектом**. Чтобы система могла ей управлять, оборачиваем её в объект. Так к функции получится прицепить служебные данные.

Ключевая переменная — `activeEffect`. Пока выполняется тело эффекта, в ней лежит этот эффект. Любое чтение данных в этот момент знает, «кому» приписать зависимость. Вне эффектов там `undefined`, и чтение ничего не отслеживает.

```
// packages/reactivity/effect.js
export let activeEffect = undefined
const effectStack = []
```

Зачем стек? Эффекты бывают вложенными (компонент, внутри него — вычисляемое значение). Когда внутренний эффект отработал, «активным» снова должен стать внешний. Стек это и обеспечивает: положили при входе, сняли при выходе.

Сам эффект — класс `ReactiveEffect`:

```
export class ReactiveEffect {
  constructor(fn, scheduler = null) {
    this.fn = fn
    this.scheduler = scheduler
    this.deps = []
    this.active = true
    this.onStop = null       // необязательный хук, вызывается раз при остановке
    recordEffectScope(this)  // регистрация в текущем scope (о нём ниже)
  }

  run() {
    if (!this.active) return this.fn()
    cleanup(this)                  // забыть старые зависимости
    try {
```

```
      effectStack.push(this)
      activeEffect = this         // «теперь слушаю я»
      return this.fn()            // выполняем — внутри произойдут track'и
    } finally {
      effectStack.pop()
      activeEffect = effectStack[effectStack.length - 1]
    }
  }
}
```

Магия — в методе `run`. Перед вызовом функции он объявляет себя активным (`activeEffect = this`). Пока функция работает и читает данные, каждое чтение видит этот `activeEffect` и записывает связь. Как функция закончила — снимаем себя со стека и возвращаем активным того, кто был до нас. Обёртка `try/finally` гарантирует, что активный эффект восстановится, даже если внутри случится ошибка.

Две новые строки в конструкторе относятся к `EffectScope` — «корзине» из того же файла, которая собирает все эффекты, созданные, пока она активна (`recordEffectScope`), чтобы потом остановить всю группу одним `scope.stop()`. Слой компонентов воспользуется этим позже: компонент складывает свой render-эффект и вотчеры в scope и на размонтировании отключает их разом.

Про `scheduler` (планировщик) и `cleanup` поговорим чуть ниже — сначала посмотрим, как хранятся связи.

Где живут связи: targetMap

Нам нужно хранить, какие эффекты зависят от какого свойства какого объекта. Структура — три уровня вложенности:

```
targetMap: WeakMap {
  реактивныйОбъект → depsMap: Map {
    'свойство' → dep: Set(эффект1, эффект2, …)
  }
}
```

Читается так: «для этого объекта, для этого его свойства — вот набор эффектов, которые его читали». Верхний уровень — `WeakMap`, чтобы забытые объекты не держались в памяти. Средний — `Map` по именам свойств. Нижний — `Set` эффектов (уникальных, без повторов).

Функция `track` наполняет эту структуру:

```
export function track(target, key) {
  if (!activeEffect || !shouldTrack) return  // никто не слушает (или трекинг на паузе)

  let depsMap = targetMap.get(target)
  if (!depsMap) targetMap.set(target, (depsMap = new Map()))

  let dep = depsMap.get(key)
  if (!dep) depsMap.set(key, (dep = new Set()))

  trackEffects(dep)                  // связать активный эффект с этим dep
}

export function trackEffects(dep) {
  if (!activeEffect || !shouldTrack) return
  if (dep.has(activeEffect)) return  // уже связаны в этом запуске
  dep.add(activeEffect)              // dep знает про эффект
  activeEffect.deps.push(dep)        // эффект знает про dep
}
```

Обратите внимание на двустороннюю связь в `trackEffects`. `dep` запоминает эффект — чтобы потом его разбудить. Но и эффект запоминает `dep` (в своём массиве `deps`) — это понадобится для очистки, о которой ниже. Флаг `shouldTrack` — глобальная «пауза» (`pauseTracking`/`resetTracking` в том же файле): мутирующие методы массива вроде `push()` попутно читают массив, и без паузы эффект, который делает push, подписался бы на собственные чтения.

Обратная операция — `trigger`:

```
export function trigger(target, key, type = 'set', newValue) {
  const depsMap = targetMap.get(target)
  if (!depsMap) return               // объект никто не читал

  // Собираем все dep-наборы, которых касается изменение, и будим их разом.
  const deps = []

  if (type === 'clear') {
    deps.push(...depsMap.values())   // collection.clear(): пропали все ключи
  } else if (key === 'length' && Array.isArray(target)) {
    // … arr.length уменьшился: будим читателей 'length' и удалённых индексов …
  } else {
```

```
    deps.push(depsMap.get(key))
    // … 'add'/'delete' (и map.set) будят ещё и эффекты-перечисления:
    //   depsMap.get(ITERATE_KEY) или 'length' массива …
  }

  // Сливаем в один Set, чтобы эффект с несколькими dep запустился один раз.
  const effects = new Set()
  for (const dep of deps) {
    if (dep) dep.forEach((effect) => effects.add(effect))
  }
  triggerEffects(effects)
}

export function triggerEffects(dep) {
  const effects = [...dep]           // копия! (объясняется ниже)

  // Проход 1: сначала computed-эффекты — их кэш надо сбросить до того,
  // как его прочитают обычные эффекты.
  for (const effect of effects) {
    if (effect.computed) triggerEffect(effect)
  }
  // Проход 2: обычные эффекты.
  // … полный файл ещё пропускает эффекты, которые уже перезапустил проход 1 …
  for (const effect of effects) {
    if (!effect.computed) triggerEffect(effect)
  }
}

function triggerEffect(effect) {
  if (effect === activeEffect) return     // не будим сами себя
  if (effect.scheduler) effect.scheduler()
  else effect.run()
}
```

Две тонкости, каждая из которых иначе стоила бы часов отладки:

1. **Копия набора** (`[...dep]`) перед перебором. Когда эффект перезапускается, он в `cleanup` удаляет себя из `dep` и тут же в `track` добавляется обратно. Менять `Set`, по которому

прямо сейчас идёт цикл, — верный способ получить бесконечный цикл или пропуск. Перебираем копию — оригинал пусть меняется спокойно.

2. **`if (effect === activeEffect) return`** в `triggerEffect` — защита от самоперезапуска. Если внутри эффекта написать `count.value++`, он одновременно и читает, и пишет `count`. Без этой строчки он будил бы сам себя до бесконечности.

В этом коде спрятаны ещё два решения. Аргумент `type` отличает изменение значения от добавления или удаления ключа — второе будит и эффекты-перечисления (`for…in`, `length` массива, `size` у Map/Set), которые подписываются на специальный `ITERATE_KEY`: у перебора нет одного свойства, от которого можно зависеть. А `triggerEffects` делает два прохода: сначала computed-эффекты, потом обычные — эффект, читающий и значение, и построенный на нём computed, никогда не увидит устаревший кэш (классический «глитч»).

reactive: перехват объекта через Proxy

Теперь заставим обычный объект вызывать `track` и `trigger` автоматически. Инструмент — `Proxy`. В перехватчике `get` зовём `track`, в `set` — `trigger`. Настоящий файл строит эти перехватчики фабрикой — одни и те же `createGetter`/`createSetter` обслуживают `reactive`, `shallowReactive` и `readonly` (глава 9), различаясь лишь двумя флагами:

```
// packages/reactivity/reactive.js  (перехватчики get и set, сокращённо)
function createGetter(isReadonly = false, isShallow = false) {
  return function get(obj, key, receiver) {
    // … служебные метки, инструментованные методы массивов, системные символы …

    const result = Reflect.get(obj, key, receiver)

    if (!isReadonly) track(obj, key)   // «меня прочитали»

    if (isShallow) return result

    // ref внутри реактивного объекта разворачивается сам: state.count,
    // а не state.count.value (исключение — массивы).
    if (isRef(result)) {
      return Array.isArray(obj) && isIntegerKey(key) ? result : result.value
    }

    if (isObject(result)) {
      return isReadonly ? readonly(result) : reactive(result) // глубина по требованию
```

```
    }

    return result
  }
}

function createSetter(isShallow = false) {
  return function set(obj, key, value, receiver) {
    let oldValue = obj[key]
    // … ref под этим ключом забирает запись в свой .value;
    //   oldValue и value сравниваются «сырыми», через toRaw …

    const hadKey =
      Array.isArray(obj) && isIntegerKey(key)
        ? Number(key) < obj.length
        : Object.prototype.hasOwnProperty.call(obj, key)

    const result = Reflect.set(obj, key, value, receiver)
    // … защита от цепочки прототипов: trigger зовёт только тот объект,
    //   в который реально писали …

    if (!hadKey) {
      trigger(obj, key, 'add')         // появился новый ключ
    } else if (hasChanged(oldValue, value)) {
      trigger(obj, key, 'set', value)  // существующий ключ получил новое значение
    }

    return result
  }
}

export function reactive(target) {
  return createReactiveObject(target, mutableHandlers, reactiveMap)
}
```

Четыре решения, которые стоит проговорить.

Глубокая реактивность по требованию. Если прочитанное свойство само оказалось объектом, мы оборачиваем его в `reactive` прямо в этот момент — не заранее, рекурсивно

по всему дереву (это дорого и иногда вредно), а лениво, когда до него реально дотянулись. Поэтому `state.user.name` реактивен на всех уровнях, но за это не платится цена, пока к `user` не обратились.

Проверка `hasChanged`. `trigger` зовём только если значение действительно другое. Присваивание того же самого не должно будить интерфейс.

`Reflect` вместо `obj[key]`. На объектах с геттерами и наследованием прямой доступ теряет правильный `this`. `Reflect.get/set` с `receiver` его сохраняют. На простых объектах разницы нет, но привыкать стоит к правильному варианту.

ref разворачивается сам. ref, лежащий внутри реактивного объекта, читается как `state.count`, а не `state.count.value` — геттер отдаёт его `.value`, а сеттер записывает присвоенное не-ref значение внутрь него. Исключение — массивы: в `arr[0]` ref может лежать законно (семантика Vue).

Внизу файла — `createReactiveObject` со всей обвязкой: он кэширует прокси по объекту (`reactive(obj)` дважды вернёт один и тот же Proxy), возвращает без изменений примитивы, объекты с меткой `markRaw` и те, кого проксировать нельзя (Date, RegExp, экземпляры классов с внутренними слотами), а для Map/Set подставляет отдельный набор перехватчиков: их методы работают со скрытыми внутренними слотами, которых у Proxy нет, поэтому файл подменяет их «инструментованными» версиями, зовущими `track`/`trigger` вручную. Массивам достаётся похожее: `includes`/`indexOf`/ `lastIndexOf` ищут ещё и по «сырому» массиву (элементы наружу выходят обёрнутыми, и поиск по идентичности промахнулся бы), а `push`/`pop`/`shift`/`unshift`/`splice` на время работы ставят трекинг на паузу, чтобы пишущий эффект не подписался на собственные попутные чтения. Перечисление — `for…in`, `Object.keys`, `size` и `forEach` коллекций — отслеживается через `ITERATE_KEY`, тот самый ключ из `trigger`.

ref: реактивность для одного значения

`Proxy` перехватывает свойства объекта. А как сделать реактивным простое число? У числа нет свойств, которые можно перехватить. Решение — положить значение в объект, в поле `.value`, и следить за чтением/записью этого поля через геттер и сеттер.

```
// packages/reactivity/ref.js
class RefImpl {
  constructor(value) {
    this._value = convert(value)   // объект внутри ref тоже делаем реактивным
    this._rawValue = value
    this.dep = new Set()           // свой набор эффектов (у ref одно «свойство»)
    this.__isRef = true
```

```
  }
  get value() {
    if (activeEffect) trackEffects(this.dep)  // чтение → track
    return this._value
  }
  set value(newValue) {
    if (hasChanged(toRaw(newValue), this._rawValue)) {
      this._rawValue = toRaw(newValue)
      this._value = convert(newValue)
      triggerEffects(this.dep)                // запись → trigger
    }
  }
}
```

Вот почему у `ref` всегда пишут `.value`: это не каприз, а единственная зацепка, через которую геттер и сеттер могут вклиниться и вызвать `track`/`trigger`. У `ref` свой `dep` прямо в объекте — ему не нужен `targetMap`, ведь «свойство» у него ровно одно.

Есть удобная надстройка `proxyRefs` — она разворачивает `.value` автоматически, чтобы в шаблонах писать `count`, а не `count.value`. Позже слой компонентов применит её к результату `setup`, и обращение к `.value` в разметке исчезнет. Код у неё короткий, загляните в `ref.js`.

computed: лениво и с кэшем

Вычисляемое значение — это формула поверх других реактивных данных, у которой два полезных свойства: она считается **лениво** (только когда результат спросят) и **кэшируется** (пока зависимости не изменились, повторный запрос отдаёт готовое).

Собирается оно из уже готовых деталей: эффект с ленивым запуском и планировщиком плюс флаг «грязно».

```
// packages/reactivity/computed.js
class ComputedRefImpl {
  constructor(getter, setter) {
    this._value = undefined
    this._setter = setter
    this._dirty = true             // надо ли пересчитать
    this.dep = new Set()
    this.__isRef = true
    this.effect = new ReactiveEffect(getter, () => {
```

```
      if (!this._dirty) {          // зависимость изменилась →
        this._dirty = true         // помечаем «грязно»
        triggerEffects(this.dep)   // и будим тех, кто читал computed
      }
    })
    this.effect.computed = true    // «сбрасывать меня раньше обычных эффектов»
  }
  get value() {
    if (activeEffect) trackEffects(this.dep)
    if (this._dirty) {             // пересчитываем только если грязно
      this._value = this.effect.run()
      this._dirty = false
    }
    return this._value
  }
  // … set value(v) передаёт запись сеттеру — computed с записью …
}
```

Здесь впервые заигрывает **планировщик** (второй аргумент `ReactiveEffect`). Обычный эффект при изменении зависимости сразу перезапускается. Но `computed` не должен считаться немедленно — он должен считаться лениво. Поэтому вместо пересчёта его планировщик лишь поднимает флаг `_dirty` и будит читателей. Сам пересчёт случится в `get value`, и только если кто-то действительно спросил результат. Прочитали дважды без изменений — второй раз вернётся из кэша, `getter` не выполнится.

Метка `this.effect.computed = true` — тот самый флаг, по которому `triggerEffects` сбрасывает computed раньше обычных эффектов. А сеттер делает возможным computed с записью: `computed({ get, set })` передаёт присваивание в ваш `set`, computed из одного геттера лишь предупреждает в консоль.

Почему нужен cleanup: задача с ветвлением

Вернёмся к `cleanup`, который `run` зовёт перед каждым запуском. Рассмотрим эффект с условием:

```
effect(() => {
  out.value = show.value ? a.value : b.value
})
```

Когда `show` истинно, эффект читает `show` и `a`, но **не** `b`. Значит, подписываться на `b` он не должен — иначе изменение `b`, которое сейчас ни на что не влияет, зря его перезапустит. Но при следующем запуске `show` может стать ложным, и тогда нужен `b`, а не `a`.

Вывод: перед каждым запуском эффект должен забыть все свои прошлые зависимости и собрать их заново — только те, что реально прочитаны в этот раз. Это и делает `cleanup`:

```
function cleanup(effect) {
  for (const dep of effect.deps) dep.delete(effect)
  effect.deps.length = 0
}
```

Он проходит по всем `dep`, где эффект записан (вот зачем эффект хранил обратные ссылки в `deps`), вычёркивает себя оттуда и очищает свой список. После `cleanup` эффект «ни на что не подписан» — а затем `fn()` подпишет его заново, но уже по факту чтения. В тестах это проверяет случай «ветвление — отписка от неиспользуемой зависимости»: после переключения ветки изменение старой переменной эффект больше не будит.

watch: следить и получать старое/новое

`effect` просто перезапускает функцию. `watch` — надстройка, которая на изменение зовёт колбэк и передаёт в него новое и старое значения. Источником может быть `ref`, геттер-функция, целый `reactive`-объект или массив таких источников.

Идея: любой источник приводим к функции-геттеру, оборачиваем в эффект, а всю «реакцию» кладём в планировщик — он вычислит новое значение и позовёт колбэк.

```
// packages/reactivity/watch.js  (сокращённо)
let oldValue
const job = () => {
  const newValue = effect.run()
  // … выходим, если значение на самом деле не изменилось …
  callback(newValue, oldValue, onCleanup)
  oldValue = newValue
}
// 'sync' срабатывает прямо внутри мутации; по умолчанию ('pre') job попадает
// в микротаск-очередь — три синхронные записи дают один вызов колбэка.
const scheduler = flush === 'sync' ? job : () => queueWatchJob(job)
const effect = new ReactiveEffect(getter, scheduler)
oldValue = effect.run()          // первый прогон: собрать зависимости, запомнить старт
```

Для `reactive`-объекта, чтобы следить «глубоко», используется обход `traverse` — он рекурсивно читает все вложенные свойства, тем самым подписывая эффект на каждый уровень (`deep: true` делает то же для любого источника, а массив источников сводится к геттеру по всем сразу). `immediate: true` заставляет колбэк сработать сразу, `once: true` останавливает наблюдение после первого вызова. Третий аргумент колбэка, `onCleanup`, регистрирует функцию, которая выполнится перед следующим вызовом и при остановке, — стандартный способ отменить запрос, устаревший из-за более нового значения.

Как всё это работает вместе

Соберём картину на примере из демо. Есть `count = ref(0)`, есть `computed` `doubled = count.value * 2`, и есть эффект, который пишет оба числа в DOM.

1. Эффект запускается первый раз. `activeEffect` — это он. Он читает `count.value` (→ `track`: `count.dep` запомнил эффект) и `doubled.value`.
2. Чтение `doubled.value` при `_dirty = true` запускает его внутренний эффект, который читает `count.value` (→ `count.dep` запомнил и computed-эффект). Результат кэшируется, `_dirty = false`.
3. Пользователь жмёт «+1»: `count.value = 1`. Сеттер `ref` зовёт `triggerEffects`.
4. В `count.dep` теперь два «слушателя»: наш DOM-эффект и computed-эффект. У computed есть планировщик — он ставит `_dirty = true` и будит читателей `doubled`. У DOM-эффекта планировщика нет — он просто перезапускается.
5. DOM-эффект читает `doubled.value`. Тот `_dirty`, пересчитывается (`2`), кэшируется. Числа на экране обновились.

Ни одной строчки «найди элемент, поставь текст» мы при клике не писали. Мы лишь описали зависимости, а система сама протянула изменение по цепочке. В этом вся суть Vue — и мы её только что построили.

Проверяем себя

Запустите тесты:

```
npm test
```

Четырнадцать проверок в `test/reactivity.test.mjs` подтверждают каждое свойство: `ref` и `reactive` реагируют, `computed` ленив и кэширует, `cleanup` отписывает от лишнего, `watch` отдаёт старое и новое. Потом откройте демо:

```
npm run serve
# http://localhost:5173/playground/01-reactivity.html
```

Понажимайте кнопки, посмотрите в консоль, как срабатывает эффект. Когда почувствуете, что реактивность «щёлкнула», переходите к следующему слою — virtual DOM. Там мы научимся превращать данные не в `document.title`, а в целые деревья элементов, и обновлять их точечно.

Virtual DOM и рендерер

В прошлой главе реактивность обновляла `document.title` — одну строчку. Настоящий интерфейс — это дерево из сотен элементов, и обновлять его целиком на каждое изменение слишком дорого: браузер тормозит, поля ввода теряют фокус, прокрутка прыгает. Нужен способ менять только то, что реально изменилось. Этим занимается рендерер, а помогает ему **virtual DOM**.

Код главы: `packages/runtime-core/` (платформо-независимое ядро) и `packages/runtime-dom/` (привязка к браузеру). Тесты — `test/renderer.test.mjs`, демо — `playground/02-vdom.html`.

Идея: сначала описать, потом сравнить

Прямо трогать страницу (`document.createElement`, `el.appendChild`) — медленно и многословно. Поэтому мы поступаем хитрее. Сначала описываем, каким интерфейс должен быть, — обычными объектами. Такой объект-описание называется **VNode** (virtual node, виртуальный узел). Потом отдаём это описание рендереру, а он сам решает, какие настоящие операции над страницей нужны.

Выгода в том, что при изменении данных мы строим новое описание и кладём его рядом со старым. Рендерер сравнивает два описания (это называется **diff**) и вносит в реальную страницу только разницу. Никаких «снести всё и нарисовать заново».

VNode: узел как объект

Один VNode выглядит так:

```
{
  type,     // 'div' (тег), либо Text, Fragment, либо компонент
  props,    // { id, class, onClick, ... }
  children, // строка, массив дочерних VNode или null
  key,      // ключ для сопоставления в списках (о нём ниже)
```

```
  el,       // ссылка на настоящий узел — рендерер её проставит при монтировании
}
```

`Text` и `Fragment` — особые типы-метки (`Symbol`), у которых нет тега. `Text` — это просто текст. `Fragment` — группа узлов без общей обёртки (когда нужно вернуть несколько элементов, не заворачивая их в лишний `<div>`). В рендерере фрагмент обрамлён двумя невидимыми пустыми текстовыми узлами — стартовым якорем (он лежит в `vnode.el`) и конечным (`vnode.anchor`), — чтобы diff'у всегда было на что указывать: соседи вставляются перед стартовым якорем, а собственные дети фрагмента живут строго между этими двумя.

Руками такие объекты писать неудобно, поэтому есть функция `h` (от «hyperscript»):

```
h('div', { id: 'app' }, 'привет')
h('ul', [ h('li', 'раз'), h('li', 'два') ])
```

`h` умеет угадывать, передали ли вы `props` или сразу детей, и в любом случае собирает корректный VNode. Именно `h` будет вызывать наш компилятор шаблонов в слое 4 — то есть `<div>{{ x }}</div>` в итоге превратится в вызов `h`.

Рендерер не знает про браузер

Ключевое архитектурное решение — рендерер не привязан к DOM. Все операции над настоящими узлами он получает снаружи, в объекте `options` (мы зовём его `nodeOps`): «создай элемент», «вставь», «удали», «поменяй текст». Сам алгоритм diff к ним не привязан.

```
export function createRenderer(options) {
  const {
    createElement: hostCreateElement,
    // ...
    insert: hostInsert,
    remove: hostRemove,
    patchProp: hostPatchProp,
    // ...
  } = options
  // ... весь diff внутри ...
  return { render, hydrate, createRenderer, patch, __installComponents }
}
```

Кроме `render`, возвращаемый объект несёт `hydrate` и пару служебных хуков — они относятся к более поздним слоям (7 и 3), пока просто не обращайте на них внимания.

Зачем такая абстракция? Затем, что тот же рендерер тогда работает где угодно. Для браузера операции даёт `runtime-dom` (файлы `nodeOps.js` и `patchProp.js`). Для тестов — выдуманное дерево в памяти (`test/helpers/testHost.mjs`), и мы проверяем diff без всякого браузера. Для сервера в слое 7 будет своя реализация. Один алгоритм — три среды. Настоящий Vue устроен ровно так же.

patch: сравнить старое и новое

Центральная функция — `patch(n1, n2, container, anchor)`. Она сравнивает старый узел `n1` и новый `n2`:

- `n1 === null` — узел появился впервые, **монтируем** его;
- `n1.type !== n2.type` — узлы несовместимы (`div` не превратить в `span`), старый убираем, новый монтируем с нуля;
- иначе — типы совпали, **обновляем на месте**: это самый частый и дешёвый путь.

`anchor` («якорь») — это узел, перед которым надо вставлять. Он нужен, чтобы попадать в правильное место среди соседей; `null` означает «в конец».

Дальше `patch` разветвляется по типу нового узла: текст, фрагмент, элемент или компонент. Для элемента при первом появлении работает `mountElement`:

```
function mountElement(vnode, container, anchor) {
  const { type, props, children } = vnode
  const el = (vnode.el = hostCreateElement(type)) // создали и запомнили узел
  for (const key in props) {
    if (key === 'key') continue // key — метаданные diff'а, не DOM-атрибут
    hostPatchProp(el, key, null, props[key])       // проставили атрибуты
  }
  hostInsert(el, container, anchor)                // СНАЧАЛА вставили в родителя
  if (typeof children === 'string' || typeof children === 'number') {
    hostSetElementText(el, String(children))       // текстовое содержимое
  } else if (Array.isArray(children)) {
    mountChildren(children, el, null)              // или рекурсивно детей
  }
  invokeDirectives(vnode, 'mounted')               // пользовательские директивы (слой 8)
}
```

Обратите внимание: `vnode.el = ...`. Мы сохраняем ссылку на настоящий узел прямо в VNode. При следующем обновлении новый VNode унаследует этот `el` — так мы переиспользуем существующий элемент вместо создания нового.

Элемент вставляется в родителя *до* монтирования детей: к моменту, когда сработают mounted-хуки пользовательских директив (слой 8), узел уже прикреплён к документу — `el.focus()` на оторванном узле молча бы не сработал.

Обновление на месте

Если типы совпали, зовётся `patchElement`: он берёт `el` из старого VNode, сравнивает атрибуты (`patchProps`) и детей (`patchChildren`).

`patchProps` проходит по новым props (обновляет изменившиеся, добавляет новые) и по старым (убирает исчезнувшие). Мелочь, которую легко забыть, — удаление: если у узла был `class`, а в новом описании его нет, атрибут надо снять.

`patchChildren` разбирает случаи «было → стало». Дети бывают текстом, массивом или пусты. Большинство сочетаний просты (стало текстом — поставили текст; было текстом, стало массивом — очистили и смонтировали). Один случай сложный и заслуживает отдельного разговора — массив превращается в массив.

Сравнение списков по ключам

Представьте список из тысячи строк, и в начало добавили одну. Наивный подход сравнил бы попарно: первый со первым, второй со вторым — и решил бы, что изменились все тысячи. Тысяча лишних правок вместо одной вставки.

Спасение — **ключи**. Мы просим пометить элементы списка уникальным `key` (обычно это id из данных). Тогда рендерер сопоставляет узлы не по позиции, а по ключу: «узел с ключом 42 был третьим, стал первым — не пересоздаём его, а переставляем».

Алгоритм `patchKeyedChildren` (тот же, что во Vue 3) работает в несколько проходов:

1. **С начала**: пока ключи совпадают, обновляем узлы на месте и двигаемся вперёд.
2. **С конца**: то же самое с хвоста списка. Эти два прохода мгновенно разбираются с самыми частыми случаями — добавили/убрали в начале или в конце.
3. Если после этого остались только новые узлы — монтируем их.
4. Если остались только старые — размонтируем.
5. Если в середине всё перемешано — строим карту «ключ → новый индекс», обновляем совпавшие узлы, удаляем лишние, а оставшиеся двигаем.

Пятый шаг прячет ещё одну оптимизацию. Двигать все узлы не нужно — часть из них уже стоит в правильном относительном порядке. Чтобы найти максимальную такую группу, мы считаем **наибольшую возрастающую подпоследовательность** (LIS, функция `getSequence`). Узлы из неё не трогаем вовсе, двигаем только остальные. Так число перестановок в DOM минимально.

Проверить это можно в тесте «переворот списка сохраняет узлы»: мы запоминаем настоящие объекты-узлы до перестановки и убеждаемся, что после `[a,b,c] → [c,b,a]` это те же самые объекты, просто переставленные. Ни один не пересоздан — значит, фокус, выделение и прочее живое состояние уцелели бы.

Грабля: unmount обязан обойти всё поддерево

Наш первый `unmount` снимал элемент одним `hostRemove(vnode.el)` — и DOM действительно исчезал. Но всё, что было *внутри*, умирало молча: вложенные в этот элемент компоненты (слой 3) не получали своих unmount-хуков, их эффекты продолжали работать, а дети `Teleport` (слой 11) навсегда оставались в целевом контейнере. Починенный `unmount` всегда рекурсивно обходит детей — с `doRemove: false`, ведь DOM потомков уходит вместе с предком, — так что уборка выполняется для всего поддерева, а пропускаются только лишние поштучные удаления.

Как это связано с реактивностью

Рендерер сам по себе ничего не знает про реактивность — ему просто говорят «отрисуй вот это». Соединяет два слоя один `effect`. Посмотрите демо `02-vdom.html`:

```
effect(() => {
  render(view(), app) // view() строит VNode-дерево из реактивного состояния
})
```

`view()` читает реактивные данные (`people.value`), поэтому `effect` подписался на них. Меняем `people` — `effect` перезапускается, строит новое дерево, `render` делает diff и вносит точечную правку. Реактивность решает «когда перерисовать», рендерер — «как перерисовать минимально».

Это, по сути, уже и есть Vue в миниатюре. Не хватает лишь одного — чтобы такой блок «состояние + view + эффект» можно было объявлять как переиспользуемую единицу. Такая единица называется **компонентом**, и ей посвящён следующий слой.

Проверяем себя

```
npm test          # тесты реактивности и рендерера
npm run serve     # http://localhost:5173/playground/02-vdom.html
```

Двенадцать проверок в `renderer.test.mjs` покрывают монтирование, обновление атрибутов, все переходы детей и, главное, keyed-diff во всех режимах: вставка, удаление, переворот,

сложная перестановка. В демо понажимайте «Перемешать» с открытым инспектором — увидите, что `<li>` двигаются, а не мигают заново.

Компоненты

К этому моменту у нас есть реактивность (слой 1) и рендерер (слой 2). В демо слоя 2 мы вручную соединили их одним `effect`: он строил дерево `h(...)` из состояния и звал `render`. Это работало, но такой блок нельзя переиспользовать — состояние, разметка и эффект висели в воздухе. Компонент оформляет ровно эту связку как самостоятельную, повторно используемую единицу.

Код главы: `packages/runtime-core/component.js`, `scheduler.js`, `apiLifecycle.js`, `apiInject.js`, `apiCreateApp.js`, `errorHandling.js`. Тесты — `test/component.test.mjs`, демо — `playground/03-components.html`.

Что такое компонент

Компонент — это обычный объект с описанием: откуда берётся состояние (`setup`) и как из него получается разметка (`render`).

```
const Counter = {
  props: ['start'],
  setup(props) {
    const count = ref(props.start)
    return { count, inc: () => count.value++ }
  },
  render(ctx) {
    return h('button', { onClick: ctx.inc }, 'Кликов: ' + ctx.count)
  },
}
```

`setup` выполняется один раз при создании компонента и возвращает то, чем будет пользоваться разметка. `render` вызывается каждый раз, когда компонент надо нарисовать, и возвращает VNode-дерево. Между ними — `ctx`, через который разметка дотягивается до состояния. Разберём, как это всё оживает.

Инстанс: личное дело компонента

Один и тот же `Counter` можно вставить на страницу десять раз, и каждый будет со своим счётчиком. Значит, у каждого вхождения должно быть своё состояние. Его хранит **инстанс** — объект, который заводится на каждое появление компонента (`createComponentInstance`). В нём лежит всё: разобранные `props`, `slots`, результат `setup` (`setupState`), последнее отрисованное дерево (`subTree`), флаги `isMounted`/`isUnmounted`, effect-scope (`scope`), собирающий всё реактивное, что компонент создаёт, функция обновления `update` и ссылка на родителя.

setup и публичный контекст

`setupComponent` готовит инстанс в три шага: разбирает `props`, раскладывает `slots` и запускает `setup`.

```
let setupResult
try {
  setupResult = instance.scope.run(() => setup(instance.props, setupContext))
} catch (err) {
  handleError(err, instance, 'setup function') // пользовательский код может упасть
} finally {
  setCurrentInstance(null) // сбрасываем «текущий» инстанс всегда, даже после throw
}

if (typeof setupResult === 'function') {
  instance.render = setupResult          // setup вернул саму render-функцию
} else if (setupResult && typeof setupResult === 'object') {
  instance.setupState = proxyRefs(setupResult) // объект состояния
}
```

Здесь две детали. `setup` выполняется внутри *effect-scope* инстанса: каждый watcher и computed, созданный в нём, попадает в `instance.scope`, и при размонтировании все они остановятся одним вызовом. И раз `setup` — пользовательский код, его падение не срывает монтирование: `handleError` ведёт ошибку по цепочке `onErrorCaptured` к `app.config.errorHandler` (см. `errorHandling.js` и раздел о жизненном цикле ниже).

Обратите внимание на `proxyRefs` из слоя 1. Он оборачивает возвращённый объект так, что `.value` у ref'ов разворачивается автоматически. Поэтому в `render` пишут `ctx.count`, а не `ctx.count.value` — это работа `proxyRefs`.

Что такое `ctx`? Это `instance.ctx` — Proxy над инстансом с правилами доступа (`PublicInstanceHandlers`). Когда `render` читает `ctx.count`, прокси ищет `count` сначала в `setupState`, потом в `props`, потом

среди служебных `$`-свойств (`$emit`, `$slots`, `$attrs`). Один `ctx` — единая точка доступа ко всему, что видно разметке.

Ещё тонкость: пока выполняется `setup`, мы объявляем инстанс «текущим» (`setCurrentInstance`). Благодаря этому `onMounted`, `provide` и `inject`, вызванные внутри `setup`, знают, к какому компоненту относятся, — без передачи инстанса аргументом.

Связь с реактивностью: render как эффект

Вот главный момент главы. Как сделать, чтобы при изменении состояния компонент сам перерисовался? Обернуть его отрисовку в реактивный эффект из слоя 1.

```
const componentUpdateFn = () => {
  if (instance.isUnmounted) return // залежавшаяся задача не должна трогать мёртвый компонент
  // …
  if (!instance.isMounted) {
    const subTree = (instance.subTree = renderComponentRoot(instance))
    patch(null, subTree, container, anchor) // первый раз — монтируем
    instance.vnode.el = subTree.el
    instance.isMounted = true
    pendingPostHooks = instance.m // onMounted — сработает позже, см. ниже
  } else {
    // …
    const nextTree = renderComponentRoot(instance)
    const prevTree = instance.subTree
    instance.subTree = nextTree
    patch(prevTree, nextTree, container, anchor) // потом — diff со старым деревом
    instance.vnode.el = nextTree.el
    pendingPostHooks = instance.u // onUpdated
  }
}
const effect = instance.scope.run(
  () => new ReactiveEffect(componentUpdateFn, () => queueJob(instance.update)),
)
const update = (instance.update = () => {
  effect.run()
  // … здесь, уже вне эффекта, срабатывают отложенные onMounted/onUpdated
})
update.id = instance.uid
```

```
update.i = instance
update() // первый запуск = монтирование
```

Когда `componentUpdateFn` первый раз выполняет `render`, тот читает реактивное состояние — и эффект на него подписывается (ровно как в слое 1). Меняется состояние — эффект должен перезапуститься. Но не сразу: его планировщик (`scheduler`) кладёт компонент в очередь через `queueJob`. Почему через очередь — следующий раздел.

При первом запуске мы монтируем поддерево (`patch(null, subTree, ...)`), при повторных — сравниваем новое дерево со старым (`patch(prevTree, nextTree, ...)`), и diff из слоя 2 вносит минимальную правку. Так реактивность отвечает на вопрос «когда перерисовать», а рендерер — «как перерисовать дёшево».

На трёх деталях стоит задержаться. Проверка `isUnmounted` — последний рубеж: если обновление встало в очередь в том же тике, в котором родитель убрал компонент, оно не должно трогать DOM. Эффект создаётся внутри `instance.scope`, поэтому размонтирование останавливает его вместе со всеми watcher'ами из `setup` одним вызовом. А `onMounted`/`onUpdated` срабатывают *после* возврата из `effect.run()`: внутри эффекта защита от самозапуска в реактивности молча проглотила бы изменение состояния из хука, и хук, меняющий состояние, никогда не дождался бы повторной перерисовки. `id` у задачи позволяет планировщику держать порядок «родитель раньше ребёнка»; `.i` говорит, чья это задача, — для маршрутизации ошибок.

Планировщик: три изменения — одна перерисовка

Если в обработчике поменять три реактивных значения подряд, наивный эффект перерисовал бы компонент трижды, хотя пользователю важен только итог. Поэтому обновления компонентов идут через очередь (`scheduler.js`).

```
export function queueJob(job) {
  // дедупликация — но только по ещё не выполненной части очереди
  if (!queue.includes(job, isFlushing ? flushIndex + 1 : 0)) {
    if (!isFlushing) {
      queue.push(job)
    } else {
      // … задача, пришедшая посреди flush, вставляется по id сразу за бегущей
    }
    queueFlush()
  }
}
```

```
function queueFlush() {
  if (!isFlushPending && !isFlushing) {
    isFlushPending = true
    currentFlushPromise = resolvedPromise.then(flushJobs)
  }
}
```

`resolvedPromise.then(...)` откладывает разбор очереди на микрозадачу — она выполнится, как только закончится текущий синхронный код. Все изменения к этому моменту уже случились, компонент в очереди один раз — значит, одна перерисовка. Это и есть «батчинг». Тест «несколько изменений подряд = одна перерисовка» это подтверждает. `flushJobs` сортирует очередь по возрастанию id задач (у родителя id меньше, чем у ребёнка, поэтому он обновляется первым — и может убрать ребёнка до его очереди) и запускает каждую задачу в собственном try/catch: компонент, упавший в render, не отменяет остальные обновления тика — ошибка уходит по цепочке `onErrorCaptured` из `errorHandling.js`.

Отсюда же берётся `nextTick`. Раз обновление DOM отложено, иногда нужно дождаться его — например, чтобы измерить уже обновлённый элемент. `await nextTick()` возвращает управление после того, как очередь разобрана и DOM актуален.

Грабля: задача, ставящая в очередь саму себя

Первая версия дедуплицировала простым `queue.includes(job)`. Звучит разумно — пока задача не поставит в очередь *саму себя* посреди flush: `onUpdated` меняет состояние, эффект срабатывает, `queueJob` находит в массиве ещё выполняющуюся задачу и молча выбрасывает повторное обновление — навсегда. Поэтому дедупликация теперь ищет только в ещё не выполненной части очереди — `includes(job, flushIndex + 1)`: бегущая задача может встать снова, а пришедшая посреди flush вставляется по id, чтобы родители по-прежнему обновлялись раньше детей.

props: данные сверху вниз

Родитель передаёт компоненту данные через атрибуты его VNode: `h(Child, { label: 'привет' })`. Компонент объявляет, что из этого — его `props`: либо списком `props: ['label']`, либо объектом с настройками на каждый prop: `props: { step: { type: Number, default: 1, required: false } }` (голый конструктор, `step: Number`, — сокращение для `{ type }`). Всё объявленное попадает в `instance.props`, всё прочее (например, `class`, повешенный снаружи) — в `attrs`.

```
for (const key in raw) {
  if (key === 'key') continue
  if (options.has(key)) props[key] = raw[key]  // объявленный prop
  else attrs[key] = raw[key]                    // «сквозной» атрибут
}
// Object-syntax extras: defaults for absent props, required/type warnings.
for (const [key, opt] of options) {
  if (key in props) {
    validatePropType(key, props[key], opt)
  } else {
    if (opt.required) console.warn(`[minivue] Missing required prop: "${key}"`)
    if ('default' in opt) props[key] = resolvePropDefault(opt)
  }
}
instance.props = reactive(props) // делаем props реактивными
```

Объектная форма проверяется по-девелоперски: неправильный тип или отсутствующий `required`-prop даёт `console.warn`, но никогда не исключение. Дефолт-объект или массив обязан быть функцией-фабрикой (`resolvePropDefault` её вызывает) — иначе все инстансы делили бы один изменяемый объект.

`props` — реактивные, и это важно. Когда родитель перерисуется и передаст новое значение, `updateComponent` вызовет `updateProps`, тот запишет новое значение в реактивный `props`, и любой, кто читал этот prop в `render`, computed или watch, отреагирует. В тесте «родитель передаёт, ребёнок обновляет» именно это и происходит.

`attrs` тоже не лежат в инстансе мёртвым грузом. После каждой отрисовки `renderComponentRoot` сливает их на единственный корневой элемент компонента: `class` и `style` объединяются, обработчики событий срабатывают оба, для остального побеждает внешнее значение. Именно поэтому `h(Button, { class: 'primary' })` работает без объявления `class` в `Button`; компонент может отказаться через `inheritAttrs: false`, а корню-фрагменту не достаётся ничего (их некуда класть).

emit: события снизу вверх

Обратное направление. Компонент не меняет чужие данные напрямую — он «кричит наверх» о событии, а родитель решает, что делать. Компонент зовёт `emit('increment')`, родитель слушает это как `onIncrement`.

```
function emit(instance, event, ...args) {
  const props = instance.vnode.props || {}
  // сначала точное имя, потом camelCase-форма ('value-change' → onValueChange)
  const handler = props[toHandlerKey(event)] || props[toHandlerKey(camelize(event))]
  if (handler) callWithErrorHandling(handler, instance, `"${event}" event handler`, args)
}
```

`emit('ping', 42)` ищет в props компонента функцию `onPing` и вызывает её с `42`. Так ребёнок сообщает, а родитель реагирует — данные текут вниз, события всплывают вверх. Это базовый контракт общения компонентов. Две мелочи повторяют настоящий Vue: kebab-case событие (`emit('value-change')`, стиль шаблонов) камелизуется и всё равно находит `onValueChange`, а обработчик выполняется через `callWithErrorHandling` — упавший обработчик родителя уходит по цепочке ошибок, а не раскручивает стек через `render` ребёнка, вызвавшего `emit`.

slots: разметка снаружи

Иногда родитель хочет положить свою разметку внутрь компонента — как содержимое в карточку. Эти «дырки» называются слотами. Дети компонента и есть его слоты:

```
h(Card, [ h('span', 'внутри карточки') ]) // это слот по умолчанию
```

`normalizeSlots` превращает детей в объект функций: массив/строка становятся слотом `default`, объект `{ header, footer }` — именованными слотами. Компонент выводит их через `ctx.$slots.default()`:

```
const Card = {
  render(ctx) {
    return h('div', { class: 'card' }, ctx.$slots.default())
  },
}
```

Слот — это функция (а не готовый VNode), чтобы содержимое вычислялось в нужный момент отрисовки, а не один раз навсегда.

provide / inject: сквозь этажи

Передавать данные через props удобно между соседними уровнями, но мучительно, если их надо протащить глубоко вниз через много промежуточных компонентов. `provide` и `inject` пробивают «туннель»: предок кладёт значение, любой потомок достаёт его напрямую.

Трюк — в наследовании через прототип. У каждого компонента объект `provides` наследует `provides` родителя (`Object.create`). Поэтому чтение ключа само поднимается по цепочке предков, пока не найдёт значение. Тест «provide/inject: значение доходит до внука» проверяет это через два уровня вложенности.

Жизненный цикл

Компонент проходит стадии: создан → смонтирован → обновляется → размонтирован. На каждую можно повесить свой код хуками `onMounted`, `onUpdated`, `onUnmounted` и другими. Хук просто добавляет вашу функцию в список у текущего инстанса, а рендерер, дойдя до стадии, вызывает весь список. `onMounted` удобен для запроса данных с сервера (элемент уже на странице), `onUnmounted` — для уборки (снять таймер, отписаться).

Кроме них есть `onActivated/onDeactivated` — они срабатывают, когда `KeepAlive` (слой 11) прячет компонент вместо разрушения, — и `onErrorCaptured`, превращающий компонент в границу ошибок: ошибка из `setup`, `render`, хуков или обработчиков потомка поднимается по цепочке `onErrorCaptured`-хуков (вернуть `false` — остановить её), затем попадает в `app.config.errorHandler`, затем в `console.error` — весь маршрут живёт в `errorHandling.js`. Хуки — тоже пользовательский код, поэтому `invokeHooks` оборачивает каждый в `callWithErrorHandling`: упавший `onMounted` не сорвёт монтирование и не помешает соседним хукам.

createApp: точка входа

Осталось собрать всё в приложение. `createApp(RootComponent).mount('#app')` оборачивает корневой компонент в VNode, цепляет к нему контекст приложения (для `provide` уровня приложения и плагинов) и зовёт `render`. Метод `use` подключает плагины — именно через него в следующих слоях встанут роутер и стор:

```
createApp(App)
  .use(router)   // плагин: router.install(app)
  .use(pinia)
  .mount('#app')
```

Грабля: перемещение keyed-компонента требует его `el`

Когда keyed-список пересортируется, diff двигает узлы, вставляя `vnode.el` перед якорем. Перерисовка компонента *откладывается в очередь*, поэтому `updateComponent` обязан скопировать DOM-узел синхронно — `n2.el = n1.el` — в момент получения нового vnode. Без этой строки пересортированный список компонентов (например, сортировка сетки

товаров) падает с `insertBefore ... is not of type Node`: у перемещаемого vnode ещё нет `el`. Обычные элементы сюда не попадают — их `el` проставляется во время patch, — поэтому баг всплывает только на keyed-списке *компонентов*.

Проверяем себя

```
npm test        # среди прочего — 10 тестов компонентов
npm run serve   # http://localhost:5173/playground/03-components.html
```

В демо — список задач: родитель держит состояние, дочерний `TodoItem` получает задачу через props и шлёт `toggle`/`remove` через emit, а список компонентов использует keyed-diff из слоя 2. Всё это без единого шаблона — на чистых `render` и `h`. Шаблоны (`<div>{{ x }}</div>`) — тема следующего слоя: мы напишем компилятор, который превращает привычную разметку ровно в такие вызовы `h`.

Компилятор шаблонов

До сих пор разметку мы писали функцией `render` и вызовами `h`. Это работает, но громоздко: сравните `h('li', { key: t.id }, t.text)` с привычным `<li :key="t.id">{{ t.text }}</li>`. Шаблоны читаются как HTML и потому нагляднее. Компилятор — переводчик: он превращает строку-шаблон в ту самую render-функцию с вызовами `h`. Никакой новой магии в рантайме не появляется — только удобный синтаксис сверху.

Код главы: `packages/compiler/`. Тесты — `test/compiler.test.mjs`, демо — `playground/04-compiler.html`.

Три шага перевода

Компилятор работает в три приёма, и это классическая схема любого компилятора:

1. **Парсинг** (`parse.js`): строка → дерево объектов (AST). Текст «`<p>{{ msg }}</p>`» превращается в структуру «элемент p, внутри вставка msg».
2. **Трансформация**: разбираем «сырые» атрибуты в осмысленные директивы (`v-if`, `v-for`, `:bind`, `@on`). У нас этот шаг совмещён с генерацией ради простоты.
3. **Генерация** (`compile.js`): обходим дерево и собираем строку с кодом `h('p', null, [_s(msg)])`, а затем делаем из неё настоящую функцию.

На выходе — функция, неотличимая от той, что вы написали бы руками. Рантайм из слоёв 1–3 про компилятор ничего не знает: ему всё равно, откуда взялась render-функция.

Парсинг: из текста в дерево

Компьютер не понимает текст структурно — для него «`<div>`» это просто пять символов. Парсер идёт по строке слева направо и, узнавая знакомые формы (тег, текст, `{{`), строит дерево узлов. Мы пишем его «рекурсивным спуском»: на вложенные теги функция вызывает сама себя.

```
function parseChildren(context) {
  const nodes = []
```

```
  while (!isEnd(context)) {
    // Remember how much input is left: every loop iteration MUST consume at
    // least one character, otherwise we would spin forever on the same spot.
    const lengthBefore = context.source.length

    const s = context.source
    let node = null
    if (s.startsWith('<!--')) {
      parseComment(context) // comments produce no output — just skip them
    } else if (s.startsWith('{{')) {
      node = parseInterpolation(context)
    } else if (s[0] === '<' && /[a-zA-Z]/.test(s[1])) {
      node = parseElement(context)
    } else {
      // Everything else is text — including a lone '<' that does not start a
      // tag (e.g. "5 < 10" or "i <3 vue").
      node = parseText(context)
    }
    if (node) nodes.push(node)

    // The progress guarantee. If a branch above ever consumes nothing, we fail
    // loudly instead of hanging the whole program in an infinite loop.
    if (context.source.length === lengthBefore) {
      parseError(context, 'Parser made no progress — cannot understand this input')
    }
  }
  return condenseWhitespace(nodes)
}
```

`context.source` — это оставшаяся, ещё не разобранная часть строки. Каждая функция «откусывает» от её начала распознанный кусок (`advanceBy`). `parseElement` читает тег и атрибуты, а на содержимое рекурсивно зовёт `parseChildren`, пока не встретит закрывающий `</tag>`. Так из плоской строки вырастает дерево.

По краям цикла прячутся две рутинные обязанности. Комментарии `<!-- -->` распознаются и пропускаются — узла из них не рождается вовсе. А `condenseWhitespace` прибирается в конце: текстовый узел из одних пробелов с переводом строки — это форматирование кода, его выбрасываем; намеренный пробел между строчными элементами выживает как одиночный `' '`. Текст и значения атрибутов заодно декодируют HTML-сущности (`&` → `&`) — наши

шаблоны не проходят через браузерный парсер, так что кроме компилятора это сделать некому.

Узлы бывают трёх типов: `Element` (тег с атрибутами и детьми), `Text` (просто текст) и `Interpolation` (вставка `{{ выражение }}`). Атрибуты парсер собирает как «сырые» пары `{ name, value }`, не вникая в их смысл, — разбираться, что `v-if` это условие, а `@click` это обработчик, будет следующий шаг.

Грабля: парсер обязан двигаться вперёд

Проверка прогресса в `parseChildren` — шрам от настоящего бага. В первой версии цикла такой гарантии не было, и шаблон с безобидным `<!-- -->` замораживал страницу: ветки для комментариев не существовало, `parseText` останавливается на каждом `<`, итерация не съедала ни одного символа — и `while` вечно топтался на одном месте. Теперь правило жёсткое: каждая ветка обязана поглотить хотя бы один символ, а итерация, не съевшая ничего, — громкая ошибка компиляции, а не тихое зависание.

Генерация: из дерева в код

Теперь по дереву собираем строку кода. Для каждого типа узла — своё правило:

```
function genNode(node) {
  switch (node.type) {
    case 'Element':       return genElement(node)
    case 'Text':          return JSON.stringify(node.content)  // "привет"
    case 'Interpolation': return `_s(${node.content})`         // _s(msg)
  }
}
```

Текст становится строковым литералом (`JSON.stringify` заодно экранирует кавычки). Вставка `{{ msg }}` превращается в `_s(msg)` — вызов помощника `_s`, который приводит значение к строке. А элемент — в `h('tag', props, children)`, где `props` и `children` рекурсивно генерируются из атрибутов и вложенных узлов.

Для `<div>привет</div>` получается ровно `h("div", null, ["привет"])` — то, что вы написали бы сами. Убедиться можно в тесте `codegen: элемент с текстом` или прямо в демо, где под приложением показан сгенерированный код его шаблона.

Директивы

Самое интересное — перевод директив. При генерации `props` мы классифицируем каждый атрибут:

- обычный `class="btn"` → статическая пара `"class": "btn"`;
- `:id="bid"` (сокращение `v-bind`) → `"id": (bid)` — значение вычисляется как выражение;
- `@click="inc"` (сокращение `v-on`) → `"onClick": (inc)` — имя события получает приставку `on` и заглавную букву, а рантайм из слоя 2 уже умеет вешать такие обработчики.

С обработчиками есть тонкость. `@click="inc"` — это ссылка на метод, её оставляем как есть: `(inc)`. А `@click="count++"` — это выражение-действие, его надо обернуть, иначе `count++` выполнится один раз при генерации, а не по клику:

```
function genHandler(on) {
  const exp = on.exp
  // ...
  const isMethodPath = /^[A-Za-z_$][\w$.]*$/.test(exp.trim())

  if (mods.length === 0) {
    return isMethodPath ? `(${exp})` : `$event => (${exp})`
  }
  // ... модификаторы событий (@click.stop) — эта история ждёт в главе о формах
}
```

`v-if` и `v-for` — «структурные» директивы, они управляют самим появлением узлов, а не их атрибутами. `v-for="item in list"` оборачивает узел в помощник `_l` (render list), который проходит по списку и возвращает массив узлов:

```
// <li v-for="item in items">{{ item }}</li>  превращается в:
h(Fragment, null, [..._l(items, (item) => h("li", null, [_s(item)]))])
```

Массив — не vnode, а `render` обязан вернуть ровно один. Поэтому когда `v-for` стоит на корневом элементе (это вполне законно, как здесь), генератор разворачивает массив в невидимый `Fragment`. Внутри родительского элемента вызов `_l(...)` точно так же разворачивается спредом в его детей.

`v-if` генерируется на уровне списка детей, потому что ему нужен доступ к соседнему `v-else`. Цепочка `v-if / v-else-if / v-else` собирается в тернарный оператор:

```
// <span v-if="ok">да</span><span v-else>нет</span>  →
(ok) ? h("span", null, ["да"]) : h("span", null, ["нет"])
```

Нет `v-else` — ветка становится `: null`, то есть «ничего не рисуем».

Из строки в функцию: with(ctx)

Финальный фокус — превратить сгенерированную строку в настоящую функцию. Здесь всплывает вопрос: в коде `h("p", null, [_s(msg)])` откуда возьмётся `msg`? Он должен прийти из состояния компонента, из `ctx`. Тащить `ctx.` перед каждым именем в генераторе — морока. Вместо этого оборачиваем тело в `with(ctx)`:

```
factory = new Function(
  'h',
  'Fragment',
  '_s',
  '_l',
  // ... плюс помощники следующих глав: _c, _key, _cd, _wd, _dir, _m, _th
  `return function render(ctx){ with(ctx){ return ${code} } }`,
)
```

`with(ctx)` заставляет каждое имя внутри сначала искаться в `ctx`. Поэтому `msg` найдётся как `ctx.msg`, а `count` — как `ctx.count`, без единого `ctx.` в коде. А помощники — `h`, `_s`, `_l` и, по мере роста движка, ещё семь — приходят параметрами внешней функции-фабрики, минуя `ctx`. Ещё одна житейская деталь: вызов `new Function` обёрнут в `try/catch` — синтаксическая ошибка здесь значит, что сломано выражение *из шаблона* (скажем, `@click="count +"`), и переброшенная ошибка цитирует сгенерированный код, а не тычет пальцем в пустоту.

`with` — редкий и обычно нежелательный оператор JavaScript, но именно здесь он к месту, и настоящий рантайм-компилятор Vue пользуется тем же приёмом (`with(this)`). Чтобы `with` правильно решал, какое имя брать из `ctx`, а какое — из внешней области, мы добавили в прокси контекста компонента ловушку `has` (слой 3). Она отвечает «да» для состояния и props, «нет» — для помощников компилятора и короткого белого списка безопасных глобалей (`Math`, `Date`, `JSON`...), а любое *другое* незнакомое имя забирает себе: шаблон получает `undefined` и однократное предупреждение — вместо опечатки, проваливающейся к `window` или взрывающейся голым `ReferenceError` посреди рендера.

Как компилятор подключается к рантайму

Рантайм не зависит от компилятора напрямую — иначе нельзя было бы собрать «лёгкую» версию без него. Связь устанавливается через регистрацию: импорт `packages/compiler/index.js` вызывает `registerRuntimeCompiler(compile)`, и с этого момента компонент со свойством `template` автоматически компилируется при инициализации (`finishComponentSetup` в слое 3). Полная сборка `packages/minivue.js` делает этот импорт за вас — как главный пакет `vue` включает и рантайм, и компилятор. Компиляция не бесплатна, поэтому `finishComponentSetup`

кэширует результат по определению компонента в `WeakMap`: список из тысячи одинаковых компонентов компилирует свой шаблон один раз, а не тысячу.

Что мы упростили

Настоящий компилятор Vue делает куда больше: статический анализ и подъём неизменных узлов, флаги патчинга для ускорения diff, кэширование обработчиков, слоты, компиляцию в отдельном шаге сборки. (`v-model` и модификаторы событий вроде `@click.stop`, которых пока нет, появятся в главе о формах.) Одно упрощение — архитектурное: `compile.js` импортирует `h` и `Fragment` прямо из runtime-core. Это инверсия слоёв по сравнению с настоящим Vue, где компилятор порождает код, который лишь *называет* своих помощников, а привязывает их рантайм при регистрации компилятора. Привязка на месте держит всю историю «шаблон → функция» в одном читаемом файле — ценой компилятора, который не живёт без runtime-core; Vue платит противоположную цену, лишний слой косвенности, ради runtime-сборки без компилятора и компилятора, работающего на этапе сборки без рантайма. Мы взяли ядро идеи — парсинг, генерацию, `with(ctx)` — и самые нужные директивы. Этого достаточно, чтобы понять, как из `<div>{{ x }}</div>` получается работающий интерфейс, и чтобы писать на шаблонах настоящие приложения.

Проверяем себя

```
npm test        # среди прочего — 10 тестов компилятора
npm run serve   # http://localhost:5173/playground/04-compiler.html
```

Тесты проверяют и генерируемый код (`compileToString`), и полный цикл «шаблон → DOM»: интерполяцию, реактивные обновления, `@click`, `v-if/v-else`, `v-for`. В демо todo-список теперь написан шаблоном, а внизу страницы виден код, в который его превратил компилятор. Дальше — роутер: научим приложение показывать разные страницы по адресу в браузере.

Router

Обычный сайт на каждый переход запрашивает у сервера новую страницу и перезагружает её. Одностраничное приложение (SPA) так не делает: оно один раз загружается, а дальше само подменяет содержимое, глядя на адрес в строке браузера. За это отвечает роутер. Наш — уменьшенная копия Vue Router, и держится он целиком на реактивности из слоя 1.

Код главы: `packages/router/`. Тесты — `test/router.test.mjs`, демо — `playground/05-router.html`.

Идея: адрес → компонент

Роутер решает одну задачу: по текущему адресу выбрать, какой компонент показать. `/` — компонент `Home`, `/about` — `About`, `/user/42` — `User` с параметром `id = 42`. Показывается выбранный компонент в специальном месте — `<RouterView>`. Ссылки `<RouterLink>` меняют адрес, не перезагружая страницу.

Весь фокус в том, что «текущий маршрут» — это реактивный объект. `<RouterView>` читает его в своей render-функции, а значит, по правилам слоя 1 подписывается на него. Меняется адрес — меняется маршрут — `<RouterView>` перерисовывается с новым компонентом. Никакого особого механизма реактивности для роутера не нужно, он переиспользует наш.

history: где хранится адрес

Адрес можно хранить по-разному, и это зависит от окружения. Поэтому способ спрятан за объектом `history` с единым интерфейсом: `location` (текущий путь), `push` и `replace` (сменить), `go` (ходить по стеку истории), `listen` (подписаться на изменения), `destroy` (снять слушателей). Реализаций три:

- **`createWebHistory`** — обычные адреса `/about` через History API (`pushState`). Красиво, но сервер должен на любой путь отдавать `index.html`.
- **`createWebHashHistory`** — адреса с решёткой `/#/about`. Часть после `#` серверу не уходит, поэтому работает на любом статическом хостинге без настройки.

- **`createMemoryHistory`** — адрес в обычной переменной, без `window`. Нужен для тестов и для сервера (слой 7), где браузера нет.

Роутеру всё равно, какой из них подключён, — он общается через один и тот же интерфейс. Тесты пользуются `createMemoryHistory`, демо — `createWebHashHistory`.

Матчер: разбор пути с параметрами

Маршрут `/user/:id` должен совпасть с адресом `/user/42` и выдать `id = 42`. Для этого путь маршрута мы превращаем в регулярное выражение, а `:параметры` — в группы захвата:

```
function compileRoute(record) {
  const keys = []
  let score = 0 // очко за каждый статический сегмент — победит самый конкретный маршрут
  const segments = record.path.split('/').filter(Boolean)
  const pattern = segments
    .map((segment) => {
      if (segment[0] === ':') {
        keys.push(segment.slice(1)) // запоминаем имя параметра
        return '/([^/]+)'           // и матчим «любой сегмент без слэша»
      }
      score++
      return '/' + escapeRegExp(segment) // статический сегмент, спецсимволы экранированы
    })
    .join('')
  const regex = new RegExp('^' + (pattern || '/') + '/?$') // '/?$' прощает хвостовой слэш
  return { ...record, regex, keys, score }
}
```

`resolve(to)` сначала отрезает от адреса хэш и строку запроса (`?q=vue` через `parseQuery` превращается в объект `{ q: 'vue' }`), затем прогоняет путь по всем записям и оставляет *самое конкретное* совпадение — именно `score` не даёт `/user/:id` заслонить `/user/new` только потому, что тот объявлен раньше. У победителя `params` собираются из групп захвата по сохранённым именам. Не совпало ничего — возвращается пустой маршрут, и `<RouterView>` покажет пусто.

Реактивный текущий маршрут

Сердце роутера — реактивный объект `currentRoute`:

```
const currentRoute = reactive({
  path: '/',
  fullPath: '/',
  query: {},
  hash: '',
  params: {},
  matched: [], // запись(и) подходящего маршрута
})

function applyRoute(route) {
  currentRoute.path = route.path
  currentRoute.fullPath = route.fullPath
  currentRoute.query = route.query
  currentRoute.hash = route.hash
  currentRoute.params = route.params
  currentRoute.matched = route.matched
}
```

`applyRoute` записывает разобранный маршрут в реактивный объект — и это единственное, что нужно, чтобы обновить интерфейс. Всё, что читало `currentRoute`, отреагирует. Роутер также подписывается на историю: внешняя смена адреса (кнопки «назад/вперёд» браузера) приходит как `navigate(path, 'pop')` — тот же конвейер, что и у нашего `push`, вместе с guard'ами — и заканчивается той же перерисовкой.

Навигация и guard'ы

`push` не сразу меняет адрес — сначала прогоняет навигационные хуки (`beforeEach`). Хук получает, куда идём и откуда, и может разрешить переход, отменить его (вернув `false`), перенаправить (вернув другой путь) — или вернуть Promise с любым из этих значений: асинхронные guard'ы дожидаются через `await`, а если за время ожидания началась более новая навигация, устаревшую бросают:

```
async function navigate(to, mode, redirectDepth = 0, id = ++navigationId) {
  const toRoute = resolve(to)
  // …
  for (const guard of guards) {
    let result = guard(toRoute, currentRoute)
    if (result && typeof result.then === 'function') {
      result = await result // асинхронный guard
```

```
      if (id !== navigationId) return false // победила более новая навигация — бросаем эту
    }
    if (result === false) {
      // На «назад/вперёд» браузер уже сменил адрес — возвращаем его на место.
      if (mode === 'pop') history.replace(currentRoute.fullPath)
      // …
      return false
    }
    if (typeof result === 'string') {
      // Редирект — с потолком, чтобы guard'ы, перенаправляющие друг на друга, не зациклились.
      if (redirectDepth >= MAX_REDIRECTS) {
        // …
        return false
      }
      // …
      const nextMode = mode === 'push' ? 'push' : 'replace'
      return navigate(result, nextMode, redirectDepth + 1, id)
    }
  }
  applyRoute(toRoute) // сначала маршрут — и только теперь адрес
  if (mode === 'push' || mode === 'replace') history[mode](toRoute.fullPath)
  return true
}
```

`mode` говорит, кто начал навигацию: `push`/`replace` приходят из кода и при успехе записывают адрес; `pop` означает, что адрес уже сменился (кнопки «назад/вперёд») и его нужно лишь проверить — с откатом при отмене; `init` проверяет стартовый адрес при подключении к приложению, чтобы auth-guard сработал и на закладке с приватной страницей. `push` возвращает Promise с `true` (перешли) или `false` (отменено, вытеснено или слишком много редиректов — цепочка ограничена `MAX_REDIRECTS = 10`, как в vue-router).

Так делают защиту страниц: «не пускать неавторизованного на `/admin`, отправлять на `/login`». Тесты проверяют обе ветки — и отмену, и редирект.

RouterView и RouterLink

`<RouterView>` — компонент из нескольких строк. Он инжектит текущий маршрут и в render отдаёт компонент из `matched`:

```
const RouterView = {
  setup() {
    const route = inject(ROUTE_KEY)
    return () => {
      const matched = route.matched[0]
      return matched ? h(matched.component) : null
    }
  },
}
```

Ключевой момент — `return () => ...`. Мы возвращаем из `setup` render-функцию, которая читает `route.matched` при каждом вызове. Значит, эффект компонента подписан на маршрут, и при навигации `<RouterView>` перерисуется сам.

`<RouterLink>` рисует обычную `<a>`, но перехватывает клик: отменяет стандартный переход браузера (`preventDefault`) и вызывает `router.push` — так адрес меняется без перезагрузки. Ещё он сравнивает свою цель с текущим маршрутом и вешает на `<a>` классы, как во vue-router: `router-link-exact-active` — при точном совпадении пути, `router-link-active` — когда текущий путь лежит «под» ссылкой (`/user` активна, пока вы на `/user/42`).

Подключение к приложению

Роутер — плагин. `app.use(router)` вызывает его `install`, где роутер раздаёт себя и маршрут через `provide`, регистрирует `<RouterView>` / `<RouterLink>` как глобальные компоненты и кладёт `$router` / `$route` в глобальные свойства:

```
install(app) {
  app.provide(ROUTER_KEY, router)
  app.provide(ROUTE_KEY, currentRoute)
  app.component('RouterView', RouterView)
  app.component('router-view', RouterView)
  app.component('RouterLink', RouterLink)
  app.component('router-link', RouterLink)
  app.config.globalProperties.$router = router
  app.config.globalProperties.$route = currentRoute
  validateInitialNavigation() // стартовый адрес тоже обязан пройти guard'ы
}
```

Чтобы `<RouterView>` в шаблоне превращался в компонент, а не в неизвестный HTML-тег, компилятору (слой 4) добавлена одна способность: теги с большой буквы или с

дефисом он генерирует через `_c('RouterView')` — разрешение компонента по имени среди зарегистрированных. В `setup` тот же роутер и маршрут достают хуками `useRouter()` и `useRoute()`.

Что мы упростили

Настоящий Vue Router умеет куда больше: вложенные маршруты (несколько `<RouterView>` в глубину), именованные маршруты и представления, ленивую загрузку компонентов, guard'ы отдельных маршрутов (`beforeEnter`) и хуки `afterEach`, стиль guard'ов с колбэком `next()`, повторяющиеся query-ключи, собираемые в массив, скролл-поведение, мета-поля и переходы-анимации. Мы взяли костяк — история, матчер с параметрами и конкретностью, реактивный маршрут с `query` и `hash`, `RouterView`/`RouterLink` с активными классами и guard'ы `beforeEach` (синхронные и асинхронные, с потолком на цепочку редиректов) — которого достаточно, чтобы понять принцип и собрать рабочее многостраничное приложение.

Проверяем себя

```
npm test        # среди прочего — 6 тестов роутера
npm run serve   # http://localhost:5173/playground/05-router.html
```

Тесты гоняют навигацию на истории «в памяти»: стартовый маршрут, `push`, параметры `:id`, пустой маршрут и оба вида guard'ов. В демо — приложение с меню из `RouterLink` и страницей пользователя, читающей `:id`. Дальше — стор: общее состояние приложения, доступное любому компоненту без передачи через props.

Store

Состояние компонента живёт внутри него, и это правильно: счётчик кнопки не должен торчать наружу. Но часть состояния общая для всего приложения — корзина, текущий пользователь, тема оформления. Передавать её через props и события по всему дереву компонентов утомительно и хрупко. Стор — отдельное реактивное хранилище, к которому любой компонент обращается напрямую. Наш стор — уменьшенная копия Pinia.

Код главы: `packages/store/`. Тесты — `test/store.test.mjs`, демо — `playground/06-store.html`.

Стор — это снова реактивность

Главная мысль: в сторе нет никакого нового механизма. Его состояние — это `reactive`/`ref` из слоя 1, вычисляемые поля — `computed`, действия — обычные функции, меняющие состояние. Компонент, читающий стор в `render`, подписывается на него так же, как на собственный `ref`. Меняется стор — перерисовываются все компоненты, которые его читали, где бы в дереве они ни находились. Стор просто даёт этому общему реактивному состоянию имя и удобный доступ.

createPinia: контейнер сторов

`createPinia()` создаёт контейнер, который подключается к приложению как плагин (`app.use(createPinia())`). Внутри — карта готовых сторов, общее реактивное состояние и список плагинов:

```
export function createPinia() {
  const pinia = {
    _stores: new Map(), // id → готовый стор (создаётся один раз, лениво)
    _plugins: [], // расширения (см. use ниже)
    state: reactive({}), // общее состояние: state[id] = состояние стора id

    use(plugin) {
```

```
      pinia._plugins.push(plugin)
      return pinia
    },

    install(app) {
      setActivePinia(pinia)
      app.provide(PINIA_KEY, pinia)
      app.config.globalProperties.$pinia = pinia
    },
  }
  return pinia
}
```

`install` запоминает контейнер как «активный» и раздаёт его через `provide` — так любой компонент сможет до него дотянуться.

defineStore: два стиля

`defineStore(id, ...)` объявляет стор. Как и во Vue, поддерживаются два стиля — Options и Setup, оба сводятся к одному внутри.

Options — привычно и декларативно:

```
const useCounter = defineStore('counter', {
  state: () => ({ count: 0 }),
  getters: { double: (s) => s.count * 2 },
  actions: { inc() { this.count++ } },
})
```

Setup — гибко, на Composition API:

```
const useCart = defineStore('cart', () => {
  const items = ref([])
  const total = computed(() => items.value.reduce((s, p) => s + p.price, 0))
  const add = (p) => (items.value = [...items.value, p])
  return { items, total, add }
})
```

`defineStore` возвращает не сам стор, а функцию `useStore()`. При первом вызове она создаёт стор, при последующих — отдаёт тот же экземпляр. Стор — одиночка на приложение:

```
function useStore() {
  const pinia = getActivePinia()
  // …
  if (!pinia._stores.has(id)) {
    createStore(id, setupOrOptions, pinia)
  }
  return pinia._stores.get(id)
}
```

Тест «стор — одиночка» это проверяет: два вызова `useCounter()` дают один и тот же объект, и изменение через один видно в другом.

Один путь создания

Чтобы не писать логику дважды, options-стор мы превращаем в setup-функцию (`optionsToSetup`) — и дальше оба стиля идут одной дорогой:

```
function setup() {
  const state = reactive(options.state ? options.state() : {})
  pinia.state[id] = state
  const parts = {}
  for (const key in state) parts[key] = toRef(state, key)          // state → refs
  for (const name in options.getters) parts[name] = computed(...)  // getters → computed
  for (const name in options.actions) parts[name] = (...a) => ...  // actions → функции
  return parts
}
```

Результат — объект `parts` из ref, computed и функций. Осталось сделать его удобным.

proxyRefs: почему store.count, а не store.count.value

`parts` содержит ref'ы, значит снаружи пришлось бы писать `store.count.value`. Некрасиво. Спасает `proxyRefs` из слоя 1 — он оборачивает объект так, что при чтении ref разворачивается автоматически, а при записи значение уходит в `.value`:

```
store = proxyRefs(parts)
```

Теперь `store.count` возвращает число, `store.count = 5` пишет в состояние, а `store.inc()` вызывает функцию. Функции `proxyRefs` не трогает — они проходят как есть. Одна строчка превращает набор ref'ов в эргономичный стор.

Внутри options-действий `this` указывает на стор, поэтому `this.count++` читает и пишет состояние через тот же `proxyRefs`, оставаясь реактивным. Тест «effect ловит изменения» подтверждает: подписка на `store.count` срабатывает при каждом `inc()`.

storeToRefs: деструктуризация без потери реактивности

Соблазнительно написать `const { count, double } = store`, но так реактивность теряется — `count` станет обычным числом на момент деструктуризации. Для состояния и getter'ов есть `storeToRefs`: он отдаёт их как ref'ы, сохраняющие связь со стором. Действия при этом берут прямо из стора (их деструктурировать безопасно — это функции):

```
const { count, double } = storeToRefs(store) // реактивные ref'ы
const { inc } = store                        // действия — как есть
```

Работает это через `toRef(store, key)` из слоя 1: полученный ref читает и пишет `store[key]`, а значит, остаётся частью реактивной системы.

Плагины

`pinia.use(plugin)` добавляет расширение, которое выполняется при создании каждого стора и получает `{ store, pinia, id }`. Что плагин вернёт — домешивается в стор. Так делают сквозные возможности: сохранение стора в `localStorage`, логирование действий, добавление общих полей. У нас это несколько строк в `createStore`.

Что мы упростили

Настоящая Pinia умеет больше: `$onAction` для перехвата действий, горячую замену модулей при разработке, интеграцию с devtools и типобезопасность через TypeScript. А вот служебные методы все на месте: `$patch` применяет групповое изменение (объект или функцию), `$subscribe` вызывает один сгруппированный колбэк на изменение с информацией о мутации (`{ type, storeId }`), `$state` у options-стора можно читать и даже присваивать — присваивание вливается в существующий реактивный объект, а не подменяет его, а `$reset` возвращает options-стор к начальному состоянию (setup-стор кидает ошибку, ровно как Pinia). Мы взяли суть — реактивное состояние-одиночку, getters на `computed`, actions, оба стиля объявления, `storeToRefs` и плагины. Этого хватает, чтобы понять идею и разделять состояние между любыми компонентами.

Проверяем себя

```
npm test        # среди прочего — 7 тестов стора
npm run serve   # http://localhost:5173/playground/06-store.html
```

Тесты проверяют оба стиля, реактивность состояния и getter'ов, одиночность, `storeToRefs` и работу стора внутри компонента. В демо — мини-магазин: «шапка» и «каталог» — два независимых компонента, но корзину они делят через стор, без единого props между ними. Остался последний слой — SSR: научимся рисовать приложение в HTML на сервере и «оживлять» его в браузере.

SSR и гидратация

До сих пор приложение рождалось в браузере: загружался JavaScript, он строил DOM, и только тогда пользователь видел содержимое. У этого две беды. Первая — пустой экран, пока грузится и выполняется код. Вторая — поисковые роботы часто видят пустую страницу. Решение — нарисовать приложение заранее, на сервере, и отдать готовый HTML. Это server-side rendering (SSR). А чтобы отданный HTML стал живым и интерактивным, его на клиенте «оживляют» — это гидратация. Оба механизма — тема последнего слоя.

Код главы: `packages/server-renderer/` (сервер) и функция `hydrate` в `packages/runtime-core/renderer.js` (клиент). Тесты — `test/ssr.test.mjs`, живой пример — `playground/07-ssr/` (Node-сервер).

Рендеринг в строку

Ключевое наблюдение: VNode-дерево не обязано превращаться именно в DOM. Его так же легко превратить в текст — HTML-строку. Никакого браузера для этого не нужно, только обход дерева и склейка строк. Этим занимается `renderToString`:

```
function renderVNode(vnode, parentComponent) {
  vnode = normalizeVNode(vnode)
  const { type } = vnode

  if (type === Text) {
    return escapeHtml(vnode.children)
  }
  if (type === Fragment) {
    return renderChildren(vnode.children, parentComponent)
  }
  if (typeof type === 'string') {
    return renderElement(vnode, parentComponent)
  }
```

```
  if (typeof type === 'object' || typeof type === 'function') {
    // компонент: создаём инстанс, выполняем setup, сериализуем поддерево
    const { instance, subTree } = createSSRComponent(vnode, parentComponent)
    // async setup() возвращает Promise — синхронный рендерер обязан упасть громко
    if (instance.setupState && typeof instance.setupState.then === 'function') {
      throw new Error('async setup() is not supported by this renderToString')
    }
    return renderVNode(subTree, instance)
  }
  return ''
}
```

Текстовый узел становится экранированным текстом, элемент — строкой `<tag атрибуты>дети</tag>` (void-теги вроде `<img>` или `<input>` — полный HTML-набор из четырнадцати штук — остаются без закрывающего тега), компонент — результатом своего `render`. Для компонента на сервере мы создаём инстанс и выполняем `setup` (`createSSRComponent`), но реактивный эффект не заводим: серверу не нужно ничего перерисовывать, нужен один снимок. А если `setup` вернул Promise (асинхронный `setup`), наш синхронный рендерер дождаться его не может — и вместо того, чтобы молча сериализовать `undefined` на месте состояния, громко падает с понятной ошибкой.

Три детали важны для правильного HTML. Первая — **обработчики событий не сериализуются**: `onClick` в HTML не имеет смысла, обработчик навесится на клиенте. Вторая — **экранирование**: данные пользователя проходят через `escapeHtml`, иначе строка `<script>` в тексте превратилась бы в исполняемый код (XSS). Тест «экранирование против XSS» это проверяет. Третья — **имена атрибутов**: значение экранировать можно, имя — нет, в HTML для этого просто нет синтаксиса, поэтому имя с пробелами, кавычками, `<`, `>`, `/` или `=` вырвалось бы из своей позиции и вписало бы в разметку что-то своё. Как `isSSRSafeAttrName` во Vue, такие имена мы просто отказываемся рендерить.

createSSRApp

Как и в браузере есть `createApp`, для сервера есть `createSSRApp`: тот же интерфейс с `use`, `provide`, `component`, но вместо `mount` — метод `renderToString`, возвращающий HTML. Плагины (роутер, стор) подключаются так же через `use`, поэтому серверный рендер полноценного приложения выглядит почти как клиентский — в этом смысл «изоморфности»: один код на две среды.

Сервер целиком

Сервер (`playground/07-ssr/server.js`) на каждый запрос страницы делает три вещи:

```
const appHtml = renderToString(createVNode(App)) // 1. рисуем приложение в строку
res.end(page(appHtml))                           // 2. вшиваем её в HTML-каркас
// 3. каркас содержит <script> клиентской гидратации
```

Каркас — обычная HTML-страница, где в `<div id="app">` уже лежит готовая разметка, а следом подключён клиентский скрипт. Пользователь видит содержимое мгновенно, ещё до того, как выполнится хоть строка JavaScript. Открыв «исходный код страницы» (View Source), вы увидите там счётчик уже нарисованным — это и есть SSR.

Обратите внимание: `App` — один и тот же модуль для сервера и клиента. Сервер импортирует его в Node, клиент — в браузере. Это изоморфное приложение.

Проблема: HTML без жизни

Отданный сервером HTML статичен. Кнопки нарисованы, но на клики не реагируют — на них нет обработчиков (мы их не сериализовали). Наивное решение — на клиенте просто запустить обычный `mount`. Но он снёс бы весь серверный HTML и создал точно такой же заново: лишняя работа, мигание, потеря состояния (позиция курсора в поле, прокрутка). Правильный путь — не создавать, а «усыновить» уже готовые узлы. Это и есть гидратация.

hydrate: усыновление DOM

`hydrate` идёт по VNode-дереву параллельно с существующим DOM. Для каждого узла он не создаёт новый, а связывает VNode с найденным реальным узлом (`vnode.el = node`) и довешивает то, чего в HTML нет, — прежде всего обработчики событий:

```
function hydrateNode(node, vnode, container) {
  vnode = normalizeVNode(vnode)
  const { type } = vnode
  const parent = (node && hostParentNode(node)) || container

  if (type === Text) {
    // … проверки расхождений; при нужде — разрезаем склеенный браузером текст (см. ниже)
    vnode.el = node
    return hostNextSibling(node)
  }
```

```
  // … (Fragment: вставляем невидимые якоря и гидрируем детей на месте)
  if (typeof type === 'string') {
    // … проверка тега → при расхождении предупреждаем и рендерим поддерево на клиенте
    vnode.el = node                              // усыновляем существующий элемент
    for (const key in vnode.props) {
      if (key !== 'key') hostPatchProp(node, key, null, vnode.props[key]) // навешиваем события
    }
    if (Array.isArray(vnode.children)) {
      let cur = node.firstChild
      for (let i = 0; i < vnode.children.length; i++) {
        const child = (vnode.children[i] = normalizeVNode(vnode.children[i]))
        cur = hydrateNode(cur, child, node)      // рекурсивно детей
      }
    }
    return hostNextSibling(node)
  }
  // компонент — усыновляется «поверх» узла, с заведением реактивного эффекта
  hydrateComponentImpl(vnode, node, parent)
  return getNextHostNode(vnode)
}
```

Для компонента гидратация делает то же, что монтирование, но первый проход рендера не создаёт поддерево через `patch(null, ...)`, а усыновляет существующий DOM через `hydrateNode`. При этом заводится обычный реактивный эффект — поэтому дальше компонент живёт как всегда: меняется состояние, работает `patch`, но уже по усыновлённым узлам.

А когда серверный DOM и клиентский vnode расходятся по-настоящему — другой тег, другой текст, — гидратация не падает: `handleHydrationMismatch` предупреждает и откатывается на клиентский рендер этого поддерева, монтируя vnode заново на месте устаревшего узла (текст же просто перезаписывается — клиенту, с его живым состоянием, доверия больше).

Тест «hydrate усыновляет DOM, навешивает события и обновляет» проверяет всё это строго: серверный узел кнопки и клиентский после гидратации — это один и тот же объект (`Object.is`), значит, узел не пересоздан; на нём появился обработчик; а клик меняет состояние и патчит ту же самую кнопку. Именно так после гидратации статичный серверный HTML становится полноценным SPA без единого лишнего пересоздания.

Грабля: браузер склеивает соседние текстовые узлы

Сервер пишет `<button>Clicks: 0</button>` двумя кусками текста — статическим `Clicks:` и динамическим `0`, — но HTML-парсер склеивает их в ОДИН DOM-узел, а в клиентском vdom по-прежнему два текстовых vnode. Наивная гидратация усыновляла склеенный узел под первый vnode, спрашивала `nextSibling` — и сходила с реального DOM: всё, что дальше, переставало совпадать. Лечение в `hydrateNode`: если текст в DOM *начинается* с текста vnode, `splitText` отрезает усыновлённую половину, а остаток оставляет следующему vnode.

Что мы упростили

Настоящий SSR во Vue сложнее в деталях: он вставляет узлы-комментарии как якоря фрагментов и порталов, сериализует и переносит на клиент состояние стора (`window.__INITIAL_STATE__`), поддерживает потоковый рендер (streaming), асинхронные `setup` и `Suspense` (наш вместо молчаливой сериализации Promise кидает понятную ошибку), кэширование фрагментов; его обработка расхождений тоже куда тоньше нашего «предупредить и перерисовать». Мы собрали костяк — рендер дерева в строку с экранированием, изоморфный сервер и гидратацию, которая усыновляет узлы, навешивает события, переживает склеенные браузером текстовые узлы и восстанавливается после расхождений, — которого достаточно, чтобы понять, как работает SSR, и собрать рабочий пример.

Проверяем себя

```
npm test                          # среди прочего — 7 тестов SSR и гидратации
node playground/07-ssr/server.js  # затем открыть http://localhost:5174/
```

Тесты покрывают `renderToString` (атрибуты, отсутствие событий, экранирование, вложенные компоненты, void-теги, `createSSRApp`) и полный цикл гидратации. В Node-примере сервер отдаёт готовый HTML, а клиент его оживляет — откройте исходный код страницы, чтобы увидеть серверную разметку, и понажимайте кнопки, чтобы убедиться, что она ожила.

Итог

Мы прошли все семь слоёв и собрали работающий аналог Vue с нуля: реактивность, virtual DOM с точечным diff, компоненты, компилятор шаблонов, роутер, стор и SSR. Ни одного из этих механизмов больше не должно быть для вас чёрным ящиком — вы написали каждый сами. Теперь, открывая настоящий Vue, вы будете видеть не магию, а знакомые идеи, просто доведённые до промышленного лоска. А это и было целью: понять, как оно устроено, изнутри.

Формы и привязки

Ядро фреймворка готово, но на первой же настоящей форме новичок упрётся в три неудобства. Поле ввода приходится вручную и читать, и записывать. Классы хочется включать по условию, а не клеить строку. А обработчику события постоянно нужны `preventDefault` и проверка нажатой клавиши. Vue закрывает это тремя удобствами: `v-model`, привязкой `class/style` объектом и модификаторами событий. Все три — надстройки над тем, что уже есть, и добавляются малой кровью.

Код главы: `packages/shared.js` (нормализация class/style), изменения в `packages/compiler/compile.js` (`v-model`, модификаторы) и `packages/runtime-dom/patchProp.js`. Тесты — `test/forms.test.mjs`, демо — `playground/08-forms.html`.

Привязка class и style

В слое 2 `class` можно было задать только строкой. Но удобнее — объектом «класс → включён ли» или массивом:

```
<span :class="{ active: isActive, done: isDone }" />
<span :class="['btn', isPrimary && 'btn-primary']" />
<div :style="{ color: 'red', fontSize: size + 'px' }" />
```

Чтобы и браузер, и SSR понимали любую форму одинаково, мы приводим её к каноничной в `shared.js`. `normalizeClass` превращает объект/массив в строку, `normalizeStyle` — в объект, `styleToString` — в строку с kebab-case-свойствами:

```
normalizeClass({ active: true, off: false })  // 'active'
normalizeClass(['a', false, 'b'])              // 'a b'
styleToString({ color: 'red', fontSize: '14px' }) // 'color:red;font-size:14px'
```

Эти помощники вызываются в двух местах: в браузерном `patchProp` (при установке `class/style` на реальный элемент) и в серверном `renderAttrs`. Одна логика — оба пути. `fontSize` превращается в `font-size` автоматически, поэтому в объекте стиля можно писать привычные JS-имена свойств.

v-model: двусторонняя связь

Поле ввода — улица с двусторонним движением: состояние должно показываться в поле, а ввод пользователя — обновлять состояние. Вручную это две вещи сразу:

```
<input :value="name" @input="name = $event.target.value" />
```

`v-model` — сахар ровно над этой парой. `<input v-model="name">` компилятор разворачивает в привязку значения плюс обработчик. Смотрим на реализацию в `applyVModel`: она выбирает свойство и событие по типу поля, потому что поля разные:

- текстовый `input` и `textarea` → `:value` + `@input`;
- `checkbox` → `:checked` + `@change` (и берём `$event.target.checked`);
- `radio` → `:checked` + `@change`, где «включён» значит «модель равна значению именно *этой* кнопки» — писать сюда `:value`, как для текста, нельзя: затёрли бы значение, по которому кнопка себя узнаёт;
- `select` → `:value` + `@change`.

```
// The value from the event, honoring the .trim / .number modifiers.
// Both may be present at once: trim first, then convert (like Vue).
let valueExpr = `$event.target.${field}`
if (mods.includes('trim')) valueExpr = `${valueExpr}.trim()`
if (mods.includes('number')) valueExpr = `Number(${valueExpr})`

directives.binds.push({ arg: prop, exp })
directives.ons.push({ event, exp: `${exp} = ${valueExpr}`, modifiers: [] })
```

Дальше это идёт по обычному пути генерации props — никакого особого узла. Модификаторы видны прямо в отрывке: `.trim` и `.number` преобразуют значение, прочитанное из события, а `.lazy` (строкой выше этого отрывка) подменяет событие на `change` — синхронизация по blur/Enter, а не на каждое нажатие. Тест «ввод обновляет состояние» проверяет обе стороны: и что состояние показывается в поле, и что ввод его меняет.

Тонкость на стороне рантайма: для `value` и `checked` `patchProp` пишет не в атрибут, а в свойство элемента (`el.value`, `el.checked`). Атрибут задаёт лишь начальное значение, а обновить уже отрисованное поле можно только через свойство — без этого `v-model` «залипал» бы.

Модификаторы событий

Обработчики почти всегда начинаются с шаблонных строк: «не перезагружай страницу», «не всплывай выше», «реагируй только на Enter». Vue выносит их в модификаторы после имени события:

```
<form @submit.prevent="save" />        <!-- $event.preventDefault() -->
<div @click.stop="onClick" />           <!-- $event.stopPropagation() -->
<input @keyup.enter="submit" />         <!-- только если нажат Enter -->
```

Компилятор парсит имя `@submit.prevent` на событие и список модификаторов (`parseEvent`), а `genHandler` строит обёртку с нужными проверками:

```
function genHandler(on) {
  const exp = on.exp
  // Listener options (.once/.capture/.passive) live in the prop name, not in
  // the handler body — ignore them here.
  const mods = (on.modifiers || []).filter((m) => !OPTION_MODS.includes(m))
  const isMethodPath = /^[A-Za-z_$][\w$.]*$/.test(exp.trim())

  // No modifiers — same as before (short code, backwards compatible).
  if (mods.length === 0) {
    return isMethodPath ? `(${exp})` : `$event => (${exp})`
  }

  // With modifiers — wrap in an arrow function with the checks.
  const guards = []
  const keyMods = mods.filter((m) => KEY_MODS.includes(m))
  if (keyMods.length) guards.push(`if(!_key($event,${JSON.stringify(keyMods)}))return;`)
  if (mods.includes('stop')) guards.push('$event.stopPropagation();')
  if (mods.includes('prevent')) guards.push('$event.preventDefault();')
  if (mods.includes('self')) guards.push('if($event.target!==$event.currentTarget)return;')

  const call = isMethodPath ? `${exp}($event)` : `(${exp})`
  return `$event => { ${guards.join('')} ${call} }`
}
```

Клавишные модификаторы (`.enter`, `.esc`, `.up` и другие) сверяются через помощник `_key`, который сопоставляет имя модификатора значению `event.key` (`enter → 'Enter'`). Тесты проверяют и сгенерированный код, и поведение: `@click.prevent` действительно зовёт `preventDefault`, а `@keyup.enter` срабатывает только на Enter.

Три модификатора стоят особняком. `.once`, `.capture` и `.passive` — не проверки внутри обработчика, а опции самого `addEventListener`. Компилятор кодирует их суффиксами в имени пропа (`@click.once → onClickOnce`, см. `genProps`), а на стороне DOM `patchProp` снимает суффиксы

обратно и передаёт опциями слушателя. Поэтому `genHandler` первым делом отфильтровывает их (`OPTION_MODS`) и строит проверки только из остальных.

Важно, что путь «без модификаторов» мы оставили прежним — короткая ссылка `(inc)` или `$event => (count++)`. Так старый код и тесты из слоя 4 продолжают работать без изменений, а обёртка появляется только там, где реально есть модификаторы.

Что мы упростили

Настоящий Vue поддерживает больше: несколько моделей на одном компоненте (`v-model:title` — модель с аргументом), `v-model` на множественном `select`, динамический `:type` (для него нужна runtime-директива — наш компилятор выбирает тип поля только по статическому `type`), системные модификаторы клавиш (`.ctrl`, `.shift`, `.exact`) и кнопки мыши. Мы взяли самое ходовое — текстовые поля, чекбокс, radio и select, объектный/массивный `class`/`style`, `.stop`/`.prevent`/`.self` и клавиши, модификаторы значения `.lazy`/`.trim`/`.number` и опции слушателя `.once`/`.capture`/`.passive`. Этого хватает, чтобы собрать полноценную форму, а принцип у остального ровно тот же. `v-model` на собственных компонентах появится в главе о директивах.

Проверяем себя

```
npm test        # среди прочего — 11 тестов форм и привязок
npm run serve   # http://localhost:5173/playground/08-forms.html
```

В демо — форма с текстовым полем, `select` и чекбоксом на `v-model`, значком, класс которого включается объектом `:class`, отправкой по `@submit.prevent` и по `@keyup.enter`. «Живое состояние» под формой всегда совпадает с полями — это и есть двусторонняя связь в действии.

Расширения реактивности

Ядро реактивности из слоя 1 закрывает 90% нужд, но в реальном коде всплывают ещё несколько инструментов. Они маленькие и целиком строятся на уже готовых `effect`, `track` и `trigger`. Эта глава короткая — просто добираем удобные примитивы.

Код главы: `packages/reactivity/` (дополнения в `reactive.js`, `ref.js`, `watch.js`). Тесты — `test/reactivity-extras.test.mjs`.

watchEffect

`watch` требует явный источник и даёт старое/новое значение. Часто это лишнее — хочется просто «выполняй вот это и повторяй, когда что-то изменится». Для этого есть `watchEffect`:

```
const stop = watchEffect(() => {
  console.log('count =', count.value) // читает count → подписывается на него
})
```

Он сразу выполняет функцию (и заодно собирает зависимости из того, что она прочитала), а потом перезапускает при их изменении. Реализация — эффект, планировщик которого ставит перезапуск в ту же микротаск-очередь, что и у `watch` (`flush: 'pre'`):

```
export function watchEffect(fn, options = {}) {
  // … обвязка onCleanup — такая же, как в watch …
  const job = () => {
    if (!effect.active) return
    runCleanup()
    effect.run()
  }
  const scheduler = options.flush === 'sync' ? job : () => queueWatchJob(job)
  const effect = new ReactiveEffect(() => fn(onCleanup), scheduler)
  effect.onStop = runCleanup
```

```
  effect.run() // первый запуск: и выполняет, и собирает зависимости
  return () => effect.stop()
}
```

Возвращает функцию остановки — вызвали её, и слежка прекращается (`effect.stop` делает `cleanup` и помечает эффект неактивным, см. слой 1). Сама функция получает `onCleanup` — контракт тот же, что у `watch`: зарегистрированное через него выполняется перед каждым перезапуском и при остановке.

readonly

Иногда данные надо отдать «только на просмотр» — например, `provide` значения, которое потомки не должны менять. `readonly(obj)` возвращает Proxy, где чтение работает (в том числе вложенное — тоже становится `readonly`), а запись и удаление молча запрещены с предупреждением:

```
const readonlyHandlers = {
  get: createGetter(true, false), // общая фабрика геттеров, флаг isReadonly включён
  set(obj, key) {
    console.warn(`readonly: cannot modify property "${String(key)}"`)
    return true // запись «проглатывается», но не выполняется
  },
  deleteProperty(obj, key) {
    console.warn(`readonly: cannot delete property "${String(key)}"`)
    return true
  },
}

export function readonly(target) {
  return createReactiveObject(target, readonlyHandlers, readonlyMap, true)
}
```

Отслеживать здесь нечего — значение через этот прокси не меняется, поэтому геттер (тот же `createGetter` из слоя 1, только с включённым флагом `isReadonly`) пропускает `track`; вложенные объекты наружу выходят тоже в `readonly`. Одно честное упрощение: Map или Set, переданные в `readonly`, проваливаются в общие перехватчики коллекций и read-only-представления не получают — запись в них не блокируется (во Vue такие представления есть; мы держим слой маленьким).

shallowReactive и shallowRef

Глубокая реактивность удобна, но иногда избыточна. Если большой объект вы меняете только заменой целиком (а не мутацией внутренних полей), оборачивать каждый уровень незачем. «Мелкие» версии реагируют только на верхний уровень:

- `shallowReactive(obj)` — реактивен `state.count`, но не `state.nested.x`. В `get` мы просто не оборачиваем вложенные объекты.
- `shallowRef(v)` — реагирует на замену `.value` целиком, но не на изменение поля внутри. Если поле всё-таки поменяли и хотите оповестить подписчиков вручную — есть `triggerRef(ref)`.

Тест «shallowRef реагирует на замену, но не на мутацию» показывает разницу: `s.value.count = 5` эффект не будит, а `s.value = { count: 10 }` — будит.

markRaw

Обратная задача: пометить объект, чтобы он **никогда** не становился реактивным. Так поступают с тяжёлыми сторонними объектами (экземпляр карты, класс из библиотеки), которым реактивность не нужна и даже вредит производительности:

```
const map = markRaw(new ExpensiveThing())
const state = reactive({ map }) // state.map останется обычным объектом
```

Реализация — невидимая метка-`Symbol` на объекте, которую `reactive` проверяет перед тем, как оборачивать вложенное: увидел метку — вернул как есть.

Итог

Все пять добавок — тонкие надстройки над механизмом слоя 1, без нового «движка». Это хорошая проверка того, насколько чистой вышла реактивность: раз новые инструменты пишутся в несколько строк каждый, значит, фундамент правильный. Дальше — директивы и динамические компоненты, там снова будет настоящая новая механика.

Директивы и динамические компоненты

Три возможности, без которых реальные приложения обходятся редко: свои директивы (`v-focus`, `v-tooltip`), переключение компонента по данным (`<component :is>`) и `v-model` на собственных компонентах. Все три докручивают уже готовую механику компонентов и компилятора.

Код главы: изменения в `packages/compiler/compile.js`, `runtime-core/renderer.js`, `runtime-core/vnode.js`, `runtime-core/component.js`. Тесты — `test/directives.test.mjs`, демо — `playground/10-directives.html`.

Кастомные директивы

Директива — способ навесить на обычный элемент низкоуровневое поведение: поставить фокус, подключить стороннюю библиотеку, следить за появлением на экране. Директива — это объект с хуками жизненного цикла элемента:

```
app.directive('focus', {
  mounted(el) { el.focus() }, // элемент появился в DOM — фокусируем
})
```

Хуки получают реальный элемент `el` и объект `binding` с данными использования: `value` (текущее значение выражения), `oldValue` (прежнее, в `updated`), `arg` (аргумент после двоеточия) и `modifiers`. Так `v-color:bg.important="c"` даст хуку `binding.value = c`, `binding.arg = 'bg'`, `binding.modifiers = { important: true }`.

Как это устроено

Компилятор, встретив неизвестную `v-*`, кладёт её в отдельный список (`parseDirective`) и оборачивает элемент в вызов `_wd` (withDirectives), передавая для каждой директивы кортеж «директива, значение, аргумент, модификаторы»:

```
// <div v-focus></div> компилируется в:
_wd(h("div", null, null), [[_dir("focus"), void 0, void 0, {}]])
```

`_dir` (`resolveDirective`) находит директиву по имени — сначала в локальной опции `directives` компонента, затем среди глобальных `app.directive`. `withDirectives` просто прикрепляет разобранные привязки к `vnode.dirs`.

Дальше в дело вступает рендерер. После вставки элемента он зовёт хук `mounted`, после обновления — `updated` (предварительно сложив прошлое значение в `oldValue`), перед удалением — `beforeUnmount` и `unmounted`:

```
function invokeDirectives(vnode, name) {
  const dirs = vnode.dirs
  if (!dirs) return
  for (const binding of dirs) {
    const hook = binding.dir && binding.dir[name]
    if (hook) hook(vnode.el, binding, vnode)
  }
}
```

То есть директивы — это просто дополнительные вызовы в тех же точках жизненного цикла элемента, где рендерер уже и так работал. Тест «mounted/updated/unmounted» проверяет все три момента и правильность `binding`.

Динамические компоненты

Иногда какой компонент показать — решают данные: вкладки, шаги мастера, список разнородных блоков. Для этого есть особый тег `<component :is="...">`, где `:is` — выражение с компонентом (объектом) или его именем (строкой):

```
<component :is="currentTab" />
```

Компилятор узнаёт тег `component`, забирает у него `:is` и генерирует тип узла через помощник `_cd`, а не строковый тег:

```
const _cd = (is) => (typeof is === 'string' ? resolveComponent(is) : is)
// <component :is="cur"> → h(_cd(cur), null, [...])
```

Если `is` — строка, `_cd` находит компонент по имени; если объект — использует напрямую. Дальше всё как с обычным компонентом: меняется `is` — `patch` видит другой `type`, размонтирует старый компонент и монтирует новый.

Тонкость из практики видна в демо и тесте: компонент кладут в `shallowRef`, а не обычный `ref`. Иначе `ref` попытался бы сделать определение компонента реактивным (обернуть в Proxy), что бессмысленно и вредно. `shallowRef` из слоя 9 хранит значение как есть — ровно для таких случаев он и нужен.

v-model на компонентах

`v-model` мы уже разобрали для полей ввода. На компоненте он работает по тому же принципу «значение вниз, событие вверх», только через prop `modelValue` и событие `update:modelValue`:

```
<MoneyInput v-model="amount" />
<!-- компилируется в: -->
<MoneyInput :modelValue="amount" @update:modelValue="amount = $event" />
```

Компилятор в `applyVModel` видит, что тег — компонент, и раскрывает `v-model` именно в эту пару (а не в `:value/@input`, как для `<input>`). Сам компонент принимает `modelValue` как prop и сообщает об изменении через `emit`:

```
const MoneyInput = {
  props: ['modelValue'],
  setup(props, { emit }) {
    return { onIn: (e) => emit('update:modelValue', Number(e.target.value)) }
  },
  template: `<input :value="modelValue" @input="onIn" />`,
}
```

Так родитель и компонент связываются двусторонне, оставаясь при этом честными: данные текут вниз через prop, изменения всплывают вверх через событие. Тест «v-model на компоненте» проверяет полный круг: ввод в дочернем поле обновляет состояние родителя.

Что мы упростили

Настоящий Vue поддерживает больше нюансов: директивные хуки `created/beforeMount/beforeUpdate`, директивы на компонентах, сокращённую форму (функция = mounted + updated), несколько `v-model` с аргументом (`v-model:title`), `keep-alive` для динамических компонентов (следующая глава). Мы взяли главное — полноценные хуки элементных директив, `<component :is>` для объектов и имён, `v-model` на компоненте — чего достаточно для подавляющего большинства задач.

Проверяем себя

```
npm test        # среди прочего — 5 тестов директив и динамики
npm run serve   # http://localhost:5173/playground/10-directives.html
```

В демо: `v-focus` ставит фокус в поле при появлении, вкладки переключают компонент через `<component :is>`, а `MoneyInput` — свой компонент с `v-model`. Дальше — встроенные компоненты Vue: `Teleport`, `KeepAlive` и асинхронные компоненты.

Встроенные компоненты

У Vue есть несколько «системных» компонентов, которые ведут себя не как обычные: `Teleport` рисует детей в другом месте страницы, `KeepAlive` сохраняет неактивные компоненты живыми, а `defineAsyncComponent` подгружает код по требованию. Они особые — рендерер и система компонентов узнают их по меткам и обрабатывают иначе.

Код главы: `packages/runtime-core/builtins.js` и правки в `renderer.js` и `component.js`. Тесты — `test/builtins.test.mjs`, демо — `playground/11-builtins.html`.

Teleport: рисуем в другом месте

Модалка логически принадлежит компоненту, который её открыл, но физически должна быть в конце `<body>` — иначе её обрежет `overflow: hidden` родителя или задавит чужой `z-index`. `Teleport` разрывает эту связь: в разметке дети внутри компонента, а в DOM — в указанном контейнере.

```
<Teleport to="#modals">
  <div class="modal">...</div>
</Teleport>
```

Сам компонент — просто метка `{ __isTeleport: true }`. Вся работа — в рендерере, который, увидев эту метку, направляет детей не в текущий контейнер, а в целевой:

```
function processTeleport(n1, n2, container, anchor) {
  if (n1 == null) {
    n2.el = hostCreateText('')           // пустой якорь на исходном месте
    hostInsert(n2.el, container, anchor)
    const target = (n2.target = resolveTeleportTarget(n2.props))
    if (target && Array.isArray(n2.children)) mountChildren(n2.children, target, null)
  } else {
    n2.el = n1.el
    const prevTarget = n1.target
```

```
    const nextTarget = resolveTeleportTarget(n2.props)
    if (prevTarget) {
      n2.target = prevTarget
      patchChildren(n1, n2, prevTarget, null) // сначала diff в СТАРОМ target
      if (nextTarget && nextTarget !== prevTarget) {
        n2.target = nextTarget
        for (const child of n2.children) move(child, nextTarget, null) // `to` сменился
      }
    }
    // … target нашёлся только сейчас → монтируем детей с нуля
  }
}
```

Пустой текстовый якорь остаётся на исходном месте, чтобы соседние узлы не сбивались, а содержимое живёт в целевом контейнере. Тест «дети рендерятся в целевом контейнере» проверяет ровно это: в исходном месте пусто, всё содержимое — в target. При обновлении дети сравниваются в *старом* target (его якоря стабильны), и только потом, если `to` сменился, переезжают в новый контейнер — игнорирование свежего target здесь было настоящим багом: вычисленное значение выбрасывалось, и дети навсегда оставались на месте.

KeepAlive: не разрушать, а прятать

Обычно при переключении динамического компонента старый уничтожается со всем состоянием: ввели текст в форму, ушли на другую вкладку, вернулись — поле пустое. `KeepAlive` это чинит: неактивный компонент не разрушается, а прячется, сохраняя инстанс и состояние.

```
<KeepAlive><component :is="currentTab" /></KeepAlive>
```

Реализация — связка «метки на vnode + особая обработка в системе компонентов». Сам `KeepAlive` ведёт кэш «ключ → vnode с живым инстансом» и расставляет метки:

```
if (cache.has(key)) {
  vnode.component = cache.get(key).component // переиспользовать живой инстанс
  vnode.__keptAlive = true // система компонентов «оживит», а не смонтирует
}
cache.set(key, vnode) // всегда кладём СВЕЖИЙ vnode — в нём актуальные props
vnode.__shouldKeepAlive = true  // при уходе — спрятать, а не разрушить
vnode.__keepAliveOwner = instance
```

Система компонентов (`component.js`) реагирует на метки в двух точках. При размонтировании вместо разрушения `unmountComponent` уносит DOM компонента в off-screen хранилище, не трогая инстанс, — и зовёт `onDeactivated`:

```
if (vnode.__shouldKeepAlive && owner && !owner.__keepAliveTearingDown) {
  move(instance.subTree, keepAliveStorage(), null) // спрятали
  instance.isDeactivated = true
  invokeHooks(instance.da, instance, 'deactivated hook') // onDeactivated
  return
}
```

А при «монтировании» кэшированного компонента `activateComponent` не создаёт его заново — возвращает спрятанный DOM, потом прогоняет обычное обновление по новому vnode (пока компонент спал, его props и слоты могли поменяться) и зовёт `onActivated`:

```
function activateComponent(vnode, container, anchor) {
  const instance = vnode.component // положен рендер-функцией KeepAlive из кэша
  move(instance.subTree, container, anchor)
  updateComponent(instance.vnode, vnode)
  vnode.el = instance.subTree.el
  instance.isDeactivated = false
  invokeHooks(instance.a, instance, 'activated hook') // onActivated
}
```

Инстанс всё это время жив, его реактивный эффект не остановлен — поэтому состояние на месте. Тест «состояние сохраняется при переключении» доказывает это: счётчик, увеличенный на вкладке A, после ухода на B и возврата остаётся прежним, а не сбрасывается.

Метка `__keepAliveOwner` важна в самом конце. Когда сам `KeepAlive` размонтируется по-настоящему, `unmountComponent` помечает его как сворачивающийся: детей больше не прячут, а всё, что ещё сидит в кэше, получает настоящий unmount — хуки срабатывают, эффекты останавливаются, DOM хранилища освобождается. Без владельца «размонтирование» прятало бы вечно, и каждый закэшированный компонент утекал бы до конца жизни страницы.

Побочно найденный баг слотов

`KeepAlive` вскрыл тонкую ошибку. Его `setup` возвращает `() => slots.default()`, захватывая `slots` в замыкание. А наш код при обновлении компонента **переприсваивал** `instance.slots` новым объектом — и замыкание продолжало видеть старый. Из-за этого `KeepAlive` всегда показывал первую вкладку. Починка: обновлять содержимое того же объекта `slots`, а не заменять ссылку (`updateSlots` в `component.js`). Хороший урок: стабильность ссылок важна везде, где что-то захватывается в замыкание.

Асинхронные компоненты

Большое приложение незачем грузить целиком сразу — редкие экраны можно подтянуть позже. `defineAsyncComponent` оборачивает загрузчик (обычно динамический `import`) в компонент, который показывает «загрузку», пока код едет, а потом — настоящий компонент:

```
const Chart = defineAsyncComponent(() => import('./Chart.js'))
```

Реализация — снова просто реактивность. Обёртка в `setup` запускает загрузчик и держит `ref`'ы состояния; когда промис разрешится, `ref` переключается — и обёртка перерисовывается уже с настоящим компонентом:

```
setup() {
  const instance = getCurrentInstance()
  const loaded = ref(false)
  const error = ref(null)

  load() // один общий запрос на все инстансы обёртки
    .then(() => {
      if (!instance || !instance.isUnmounted) loaded.value = true
    })
    .catch((err) => {
      if (!instance || !instance.isUnmounted) error.value = err
    })

  return () => {
    if (loaded.value && resolvedComponent) return h(resolvedComponent)
    if (error.value) {
      return options.errorComponent ? h(options.errorComponent) : h('span', 'Loading failed')
    }
    return options.loadingComponent ? h(options.loadingComponent) : h('span', 'Loading…')
  }
}
```

Никакой особой машинерии — реактивность сама переключает вид. Две страховки заслуживают внимания: `load()` кэширует единственный промис загрузки, так что десять асинхронных компонентов в списке дают один запрос, а не десять (проваленная загрузка очищает слот и разрешает повтор); а проверки `isUnmounted` не дают обёртке, снятой до прихода чанка, переключить состояние и смонтировать опоздавший компонент в уже исчезнувший контейнер. Тесты проверяют оба исхода: успешную загрузку и ошибку (тогда показывается `errorComponent`).

Что мы упростили

За кадром остался `Suspense` — координатор нескольких асинхронных зависимостей с общим фоллбэком и асинхронным `setup`. Он завязан на более глубокую работу с промисами внутри рендера и заслуживал бы отдельной большой главы; в учебных целях достаточно `defineAsyncComponent`, показывающего суть «загрузка → готово». Также в настоящем Vue `KeepAlive` умеет `include`/`exclude` и лимит кэша (`max`), а `Teleport` — `disabled`. Мы взяли ядро каждого — но хуки `activated`/`deactivated` всё-таки вошли: `onActivated`/`onDeactivated` срабатывают ровно там, где компонент прячут и возвращают.

Проверяем себя

```
npm test        # среди прочего — 5 тестов встроенных компонентов
npm run serve   # http://localhost:5173/playground/11-builtins.html
```

В демо: модалка через `Teleport` уезжает в `#modals`, вкладки под `KeepAlive` хранят введённый текст, а асинхронный блок появляется с задержкой. Остался последний слой — расширения стора и большое итоговое приложение, где сойдётся всё.

Расширения стора и итоговое приложение

Последний слой добирает удобства стора и сводит всё, что мы построили, в одно живое приложение. Здесь новой глубокой механики почти нет — есть завершение картины.

Код главы: дополнения в `packages/store/index.js`, итоговое приложение — `playground/12-capstone.html`.

$patch, $subscribe, $reset

Три служебных метода стора, которые в реальной работе нужны постоянно.

`$patch` меняет несколько полей разом — объектом или функцией. Объект удобен для простых замен, функция — для вычислений на основе текущего состояния:

```
store.$patch({ count: 5, name: 'y' })      // слить объект
store.$patch((s) => { s.count++; s.done = true }) // мутация функцией
```

Реализация выбирает цель изменения: для options-стора это его реактивное состояние `pinia.state[id]`, для setup-стора — сам стор (запись через `proxyRefs` попадёт в `.value` нужного ref). Одно изменение — один способ.

`$subscribe` вызывает колбэк на любое изменение состояния. Это глубокий `watch` поверх нашей реактивности, и он батчит вызовы, как Pinia: сколько угодно синхронных мутаций (скажем, один `$patch` из трёх полей) сворачиваются в один колбэк на следующей микрозадаче. Колбэк получает информацию о мутации — `{ type, storeId }`, где `type` — это `'direct'`, `'patch object'` или `'patch function'`:

```
store.$subscribe((mutation, state) => console.log(mutation.type, state))
```

Тонкость, которую вскрыл тест: для setup-стора нельзя следить и за состоянием, и за getter'ами сразу. Иначе одно изменение `items` сработало бы дважды — от самого ref и от зависящего от него `computed`. Поэтому мы отфильтровываем computed (у него есть `.effect`) и

глубоко следим только за ref'ами состояния — `{ deep: true }` заставляет наблюдатель заходить внутрь значений ref, так что `items.value.push(...)` тоже видно. Мелочь, а показывает, как связаны слои: подписка стора опирается на устройство `computed` и `watch` из слоя 1.

`$reset` возвращает состояние к начальному. Он работает для options-сторов, где известна функция `state()`, дающая свежие значения. Для setup-стора начального снимка нет, поэтому `$reset` бросает ошибку — ровно как Pinia (setup-стор при желании заводит собственный action для сброса).

Итоговое приложение

`playground/12-capstone.html` — приложение «Заметки», в котором работает сразу всё:

- **Роутер** — три маршрута: список `/`, редактирование `/note/:id` с параметром, страница «О программе». Навигация через `RouterLink`, отрисовка через `RouterView`.
- **Стор** — `useNotes` в setup-стиле хранит заметки; `storeToRefs` даёт реактивные ссылки в компонент; `$subscribe` логирует изменения в консоль.
- **Формы** — добавление и редактирование через `v-model`, отправка по `@keyup.enter`.
- **Teleport** — подтверждение удаления вынесено в модалку поверх страницы.
- **Компоненты** — список и деталь общаются только через стор и роутер, без проброса props между собой.

Ни одной сторонней строчки: и реактивность, и рендеринг, и компиляция шаблонов, и роутер, и стор — всё наше, собранное за двенадцать слоёв. Если это приложение работает — значит, работает весь фреймворк.

Ещё два приложения — и баг, который поймал магазин

В папке `examples/` лежат два приложения покрупнее, целиком на нашем фреймворке:

- **MiniTrello** (`examples/kanban/`) — kanban-доска: роутер, стор с персистентностью, формы на `v-model`, Teleport-модалка, KeepAlive-вкладки, async-панель и кастомные директивы. Именно её прогон в настоящем браузере впервые выловил несколько багов рендерера.
- **MiniShop** (`examples/shop/`) — витрина на фейковом REST API: асинхронная загрузка с состояниями ошибки, страница `/product/:id` с рефетчем через `watchEffect`, корзина с персистентностью в Teleport-drawer, плюс живой поиск, фильтр по категориям и сортировка.

MiniShop окупил себя сразу. Его сетка товаров — это **keyed-список компонентов** (`<ProductCard :key="id">`), и сортировка пересортировывает эти компоненты. Такое перемещение падало с `insertBefore ... is not of type Node` — та самая грабля из главы 3. Перерисовка

компонента откладывается в очередь, поэтому `updateComponent` обязан синхронно перенести DOM-узел (`n2.el = n1.el`), чтобы keyed-diff смог его подвинуть. Прежние приложения пересортировывали только keyed-*элементы*, у которых `el` ставится во время `patch`, — поэтому баг прятался, пока настоящей сетке товаров не понадобилась сортировка. Ещё одно напоминание: вернее всего дыры во фреймворке находятся, когда строишь на нём что-то настоящее.

Что мы прошли

Двенадцать слоёв, каждый — поверх предыдущих:

1. **Реактивность** — `track`/`trigger`, `ref`, `reactive`, `computed`, `watch`.
2. **Virtual DOM** — `h`, рендерер, keyed-diff через наибольшую возрастающую подпоследовательность.
3. **Компоненты** — `setup`, реактивный render-эффект, планировщик, props/emit, слоты, provide/inject, жизненный цикл, `createApp`.
4. **Компилятор** — из шаблона в render-функцию через `with(ctx)`.
5. **Router** — реактивный маршрут, история, параметры, guard'ы.
6. **Store** — `defineStore`, getters на `computed`, actions, плагины.
7. **SSR** — `renderToString` и гидратация с усыновлением DOM.
8. **Формы** — `v-model`, привязка class/style, модификаторы событий.
9. **Расширения реактивности** — `watchEffect`, `readonly`, `shallow*`, `markRaw`.
10. **Директивы и динамика** — кастомные директивы, `<component :is>`, `v-model` на компонентах.
11. **Встроенные компоненты** — `Teleport`, `KeepAlive`, асинхронные компоненты.
12. **Расширения стора** — `$patch`, `$subscribe`, `$reset` — и это приложение.

Все они выросли из одной идеи первой главы: «данные изменились — то, что от них зависит, обновилось само». Реактивность оказалась не одной из возможностей, а фундаментом, на который встало всё остальное — рендеринг, компоненты, роутер, стор, даже SSR. Это и есть главный вывод книги: большие системы редко держатся на многих сложных механизмах. Чаще — на одной правильной идее, аккуратно доведённой до всех уголков.

Что дальше

Настоящий Vue отличается от нашего не идеями, а отделкой: флаги патчинга для ускорения diff, статический подъём узлов в компиляторе, `Suspense` и потоковый SSR, devtools, строгая типизация, тысячи проверенных пограничных случаев. Всё это — инженерный лоск поверх ровно тех механизмов, что вы теперь написали сами. Откройте исходники Vue: там будут

знакомые `track`, `trigger`, `patchKeyedChildren`, `setupComponent` — только больше и осторожнее. Читать их вы теперь будете не как магию, а как расширенную версию собственного кода.

Спасибо, что прошли весь путь. Вы не выучили Vue — вы его написали.

Проверяем себя

```
npm test        # все 247 тестов двенадцати слоёв
npm run serve   # http://localhost:5173/playground/12-capstone.html
```

Приложение «Заметки» — финальная проверка: добавьте заметку, откройте, отредактируйте, удалите через модалку, полистайте маршруты. Всё это крутится на фреймворке, который ещё двенадцать глав назад был пустой папкой.