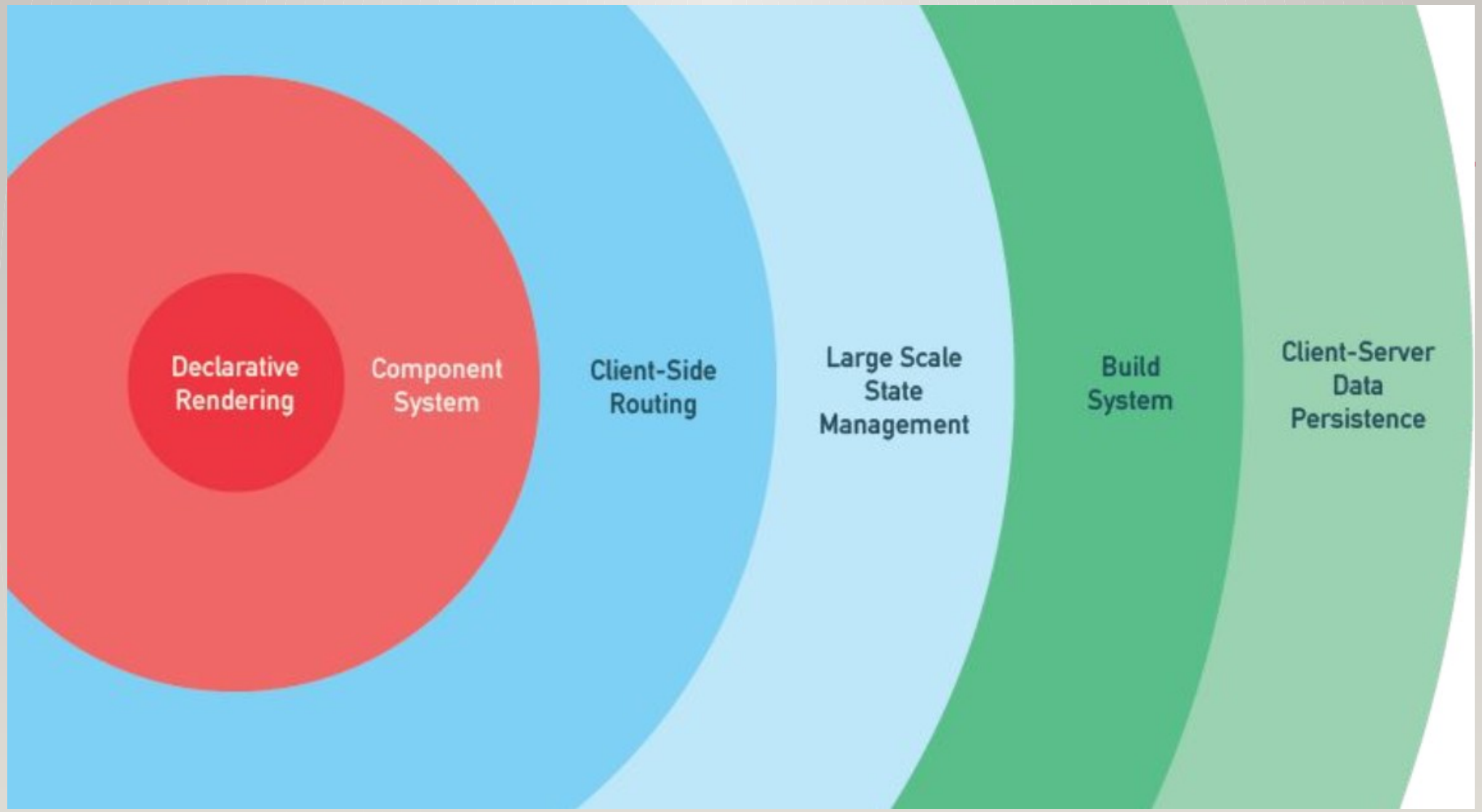
VUE 3.0

大纲

- 背景
- 新特性

背景

- 三大框架之一
- 渐进式框架
- 生态成熟


Declarative Rendering
Component System
Client-Side Routing
Large Scale State Management
Build System
Client-Server Data Persistence

官方工具链

- 构建工具脚手架 :vue-cli
- 开发者工具 :vue-devtools
- IDE 支持 :VSCode + Vetur
- 静态检查 :ESLint + eslint-plugin-vue
- 单元测试 :Jest + vue-jest + vue-test-utils
- 文档 / 静态站生成 :VuePress

目标

- 更小
- 更快
- 加强 api 一致性设计
- 加强 typescript 支持
- 提高自身可维护性
- 开放更多底层功能

更小

- 全局 api 和内置组件 / 功能特性将支持 tree-shaking
- 常驻代码将永远保持在 10kb gzipped 之下

更快

- 基于 Proxy 的数据监测，性能整体优于 Object.defineproperty
- Virtual DOM 重构
- 架构重构，更多的优化

提高自身可维护性

- 代码采用 monorepo 结构，内部分层更清晰
- TypeScript 使得外部贡献者更有信心做改动

开放更多底层功能

```
import { createRenderer } from '@vue/runtime-core'

const { render } = createRenderer({
 nodeOps,
 patchData
})
```

虚拟 DOM 重写

- 传统 **vdom** 的性能瓶颈

- 动静结合突破瓶颈

传统 VDOM 的性能瓶颈

- 虽然 Vue 能够保证触发更新的组件最小化，但在单个组件内部依然需要遍历该组件的整个 vdom 树
- 在一些组件整个模版内只有少量动态节点的情况下，这些遍历都是性能的浪费
- 传统 vdom 的性能跟模版大小正相关，跟动态节点的数量无关

动静结合突破瓶颈

- 通过模版静态分析生成更优化的 vdom 渲染函数
- 将模版切分为 block(if, for, slot)，每个 block 内部动态节点位置是固定的
- 每个 block 的根节点会记录自己所包含的动态节点 (包含子 block 的根节点)
- 更新时只需要直接遍历动态节点
- 新策略将 vdom 更新性能与模版大小解耦，变为与动态节点的数量相关

基于 PROXY 的 OBSERVATION

- 目前， Vue 的响应式系统是使用带有 Object.defineProperty 的 getter 和 setter
- Vue 3 将使用 ES6 Proxy 作为其观察机制。
- 速度 / 内存

基于 PROXY 的 OBSERVATION

- 3.0 将带来一个基于 Proxy 的 observer 实现，它可以提供覆盖 javascript 全范围的响应式能力，消除了当前 Vue 2 系列中基于 Object.defineProperty 所存在的一些局限，这些局限包括：
- 对属性的添加、删除动作的监测；
- 对数组基于下标的修改、对于 length 修改的监测；
- 对 Map 、 Set 、 WeakMap 和 WeakSet 的支持；

FUNCTION-BASED API

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

FUNCTION-BASED API

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

关于逻辑复用

- Mixin
  - 混入的属性来源不清晰
  - 命名空间冲突
- 高阶组件 (HOC)
  - Props 来源不清晰
  - Props 命名空间冲突
  - 多余的组件实例造成的性能浪费
- Scoped Slots
  - 来源清晰
  - 无命名空间冲突
  - 多余的组件实例造成的性能浪费

关于逻辑复用

- Composition Functions
  - 就是简单的函数组合
  - 无额外的组件实例开销
  - 以“功能” 而不是“选项”或“生命周期”切分代码

COMPOSITION FUNCTIONS

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

```
function useMousePosition() {
  const x = value(0)
  const y = value(0)

  const update = e => {
    x.value = e.pageX
    y.value = e.pageY
  }

  onMounted(() => {
    window.addEventListener('mousemove', update)
  })

  onUnmounted(() => {
    window.removeEventListener('mousemove', update)
  })

  return { x, y }
}
```

优化 SLOT 的生成

- 所有由编译器生成的 slot 都将是函数形式
- slot 的内容发生变动时，只有子组件会被重新渲染

TYPESCRIPT

- 3.0 本身用 TypeScript 重写，内置 typing
- TSX 支持
- 不会影响不使用 **TS** 的用户

更好的调试能力

- 通过使用新增的 renderTracked 和 renderTriggered 钩子，我们可以精确地追踪到一个组件发生重渲染的触发时机和完成时机及其原因

兼容 IE 11

- 基于老式的 Object.defineProperty API

核心改动

- https://github.com/vuejs/rfcs

THANKS