前端框架对比（主要吐槽 React ）

现状对比

构建工具

与 create-vue 相比， create-react-app 的简陋程度很难让人相信前者出自个人开发者（组成的团队），而后者出自国际大厂；以至于尤雨溪亲自下场，建议 React 使用 Vite、Vitest 而不是 CRA 和 jest（：虽然这种在别人坟头蹦迪的行为不太友好，但我觉得至少这个建议挺有道理的；

create-vue 生成的项目，依赖的工具包干净整洁，CRN 则一堆标黄，不清楚是不是一定要崩掉或是再来一次 left-pad 才想起要升级依赖；

生态

Vue 官方开发并维护路由、状态管理、测试、IDE 插件等；React 只有 React，甚至开发文档都不够友好，以至于需要重写（还没写完） beta.reactjs.org/ ; code push 、IDE 插件设置需要友商 MS 来提供；React 源码 flow 编写（曾经是先进性代表，现在使用较少），Vue（Vue 3 之前曾使用 flow）、SolidJS 等现代框架都使用 TS；

当然，上面这些对比可能不是因为 Meta 不厉害，而是因为 Meta 太超前。

> Facebook 是一家技术很厉害的公司，能够超前做一些外界没有的东西，但等外界把这个东西做出来了，Facebook 就发现自己迁移不过去了，被自己过去超前做的技术锁定了，因为迁移成本太高。举个例子，在还没有 webpack 的时候 Facebook 就有自己很好的前端构建流水线，但 webpack 出来后 Facebook 无法迁移到 webpack，甚至无法轻易把 transpiler 迁移到 Babel。
>
> 如果我没记错的话，Babel 作者 Sebastian McKenzie 进入 Facebook 后做过一个项目，就是帮助 Facebook 迁移到 Babel。为什么呢？因为在外界还没有 Babel 的时候，甚至在 Babel 前身 6to5 还没出现的时候，Facebook 内部的流水线已经有自己的 transpiler，能够把一部分 ES6 语法转译为 ES5。当时我们可爽了，在外部根本还没意

> 识到能这样做事情的时候，我们已经可以随手写 ES6 了。但有了 Babel 后，内部流水线根本不兼容中间插入 Babel 这一步，所以就需要专门改造这个流水线才能迁移到 Babel。而且 Facebook 已有的大量代码的 ES6 写法是基于内部 transpiler 写的，谁能保证迁移到 Babel 后 100% 兼容？迁移到 Babel 后如果编译出错了，那还能找出来修复。如果不出错，但实际执行结果略微不一样，导致出现线上事故，那怎么办？
>
> React 在写的时候，是基于上述 Facebook 内部流水线写的，所以自然是内部有什么工具就依赖什么工具。React 一开始写的时候，其实是没有 Babel、TypeScript 和 Flow 的，但有上述内部 transpiler，所以就这样写了。到后来有了 Flow，而且要保证依赖于 React 的代码能够得到正确的 Flow 类型推断，自然就加上了 Flow 注释。此外，内部流水线应该是从来没做过 TypeScript 支持的，所以估计就算想用 TypeScript 来写也做不到，因为这不是加个 webpack 插件就能支持的。

周边

好的一方面是 Meta 一直保持 RN 的迭代更新，而与之相对的阿里 WEEX 项目，活跃程度就差好多，WEEX 2.0 从2021年说到现在，还是神龙见首不见尾；腾讯的 hippy 也是从去年宣传至今，仍未发布。RN 的发展也不是一帆风顺，Airbnb、Notion 因性能问题弃用 RN 转投原生，Flutter 作为后起之秀，因为更好的性能，越来越流行（不支持热更新；Dart 与 JS 相比，学习曲线大；Flutter 包大；嵌套丑）。

React 厉害的地方

JSX、hooks、vdom（fiber）

React 的超前，体现在首次提出或是首次让以上概念流行，并被 Vue、 SolidJS 等框架借鉴；JSX 简单灵活，学习成本几乎没有，也被 Vue、SolidJS 接纳；hooks + function component 的方式，基本能够代替之前的 class component，“启发”了 Vue 的 composition api，hooks 成为前端开发范式；

React 不那么厉害的地方

JSX

Vue 虽然也支持 JSX/TSX，但是更推荐使用 template 模板语法；因为 JSX 过于灵活，而相对呆板的 template 能实现大多数功能的同时，在编译阶段就能进行优化。

比如下面这段代码：

```xml
<main>
  <p>{{ count }}</p>
  <!-- cacheHandlers 从 cache 中获取函数 -->
  <button @click="onAddClick">add</button>
  <button @click="onDeleteClick">add</button>
  <!-- 静态数据，编译之后提升为 _hoisted_1，render 时无需重新执行 -->
  <p>static</p>
</main>
```

经过编译后：

```javascript
import { toDisplayString as _toDisplayString, createElementVNode as _createElementVNode, createComme

const _withScopeId = n => (_pushScopeId("scope-id"),n=n(),_popScopeId(),n)
const _hoisted_1 = /*#__PURE__*/ _withScopeId(() => /*#__PURE__*/_createElementVNode("p", null, "sta

export function render(_ctx, _cache, $props, $setup, $data, $options) {
  return (_openBlock(), _createElementBlock("main", null, [
    _createElementVNode("p", null, _toDisplayString(_ctx.count), 1 /* TEXT */),
    _createElementVNode("button", {
      onClick: _cache[0] || (_cache[0] = (...args) => (_ctx.onAddClick && _ctx.onAddClick(...args)))
    }, "add"),
    _createElementVNode("button", {
      onClick: _cache[1] || (_cache[1] = (...args) => (_ctx.onDeleteClick && _ctx.onDeleteClick(...a
    }, "add"),
    _hoisted_1
  ]))
}

// Check the console for the AST
```

SolidJS 在支持 JSX 的同时，也能够在编译阶段进行优化，主要是通过提供特定的流程控制组件：

```javascript
import { render } from 'solid-js/web';
import { createSignal, For } from 'solid-js';

function App() {
```

```
  const [cats, setCats] = createSignal([
    { id: 'J---aiyznGQ', name: 'Keyboard Cat' },
    { id: 'z_AbfPXTKms', name: 'Maru' },
    { id: 'OUtn3pvWmpg', name: 'Henri The Existential Cat' }
  ]);

  return (
    <ul>
      <For each={cats()}>
      {(cat, i) => (
        <li>
          <a target="_blank" href={`https://www.youtube.com/watch?v=${cat.id}`}>
            {i() + 1}: {cat.name}
          </a>
        </li>
      )}
    </For>
    </ul>
  );
}

render(() => <App />, document.getElementById('app'))
```

与 SolidJS 、 Vue 相比，React 在编译阶段所做的事情相对较少:

javascript 复制代码

```
import {useState} from 'react';
const App = ()=>{
  const [count,setCount]=useState();
  const [list]=useState([{name:'wjz'}]);
  return <>
      <p>{count}</p>
      <div>app</div>
    {list.map(v => <p>{v.name}</p>)}
    </>
}
```

编译后结果:

javascript 复制代码

```
"use strict";

var _react = require("react");

const App = () => {
  const [count, setCount] = (0, _react.useState)();
  const [list] = (0, _react.useState)([{
    name: 'wjz'
```

```
  }]);
  return /*#__PURE__*/React.createElement(React.Fragment, null, /*#__PURE__*/React.createElement("p"
};
```

可以看出，与 Vue 和 SolidJS 的编译结果相比，React 只是将 JSX 的预发转换为 JS 可以执行 `createElement` 的语法；变量提升、缓存函数等用于加速执行时的优化，在 React 的编译结果中并没有体现；JSX 语法把灵活体现到极致，但正因为太灵活，导致编译环节进行优化困难重重。

hooks

React 创造性的提出了使用 hooks 来解决代码复用的问题，然后用一个不那么好的实现，引入了其他问题。

1. useMemo/useCallback/useEffect 需要手动处理依赖关系
2. 无生命周期，需要通过 useEffect 模拟；
3. Hooks 通过链表实现，限制了调用顺序，不能出现在条件分支中；
4. Function 组件有可能会执行多次；

Vue 的 composition api 借鉴了 React，但在设计上：

1. Vue 在 0.x 版本开始，就是通过依赖追踪来实现更新（从 defineProperty 到 Proxy）；composition api 的设计一脉相承，自动收集依赖，不需要手动处理依赖关系；
2. 保留了 onMounted、onUpdated 生命周期；
3. Hooks 可以出现在条件分支中；

   // 在 Vue 中，这是可以的 if (n <= 3) { watchEffect(() => { console.log("time changed", time.value); }); }

4.Setup 函数只执行一次；

React 需手动处理依赖关系，相当于把优化手段交给开发者。虽然有 eslint 工具辅助，但开发体验不佳；

SolideJS 语法类似 React，实现类 Vue，和 Vue 致敬 React 的同时，一起教 React 团队实现 hooks 的正确方式：

```javascript 复制代码
import { render } from "solid-js/web";
import { createSignal, createEffect } from "solid-js";

function Counter() {
  console.log('render only once');
  const [count, setCount] = createSignal(0);
  const increment = () => setCount(count() + 1);
  // 无需手动处理依赖
  createEffect(() => console.log(count()));
  return (
    <button type="button" onClick={increment}>
      {count()}
    </button>
  );
}

render(() => <Counter />, document.getElementById("app")!);
```

好消息是 React 团队也意识到自家 hooks 在使用上的问题，也就有了 useEvent 提案 和正在实现中的 React forget；

vdom（fiber）

React 顶层组件 state 更新，如果不进行任何优化，则所有子组件都会重新 render 一遍；

```javascript 复制代码
import React, { useState } from 'react';
function Foo(props) {
  console.log('foo render')
  return <>
    {props.children}
    <div>Foo</div>
  </>
}
function Bar() {
  console.log('bar render');
  return <div>Bar</div>;
}

function App() {
  console.log('app render');
  // count 每次更新，app render、 foo render、bar render
```

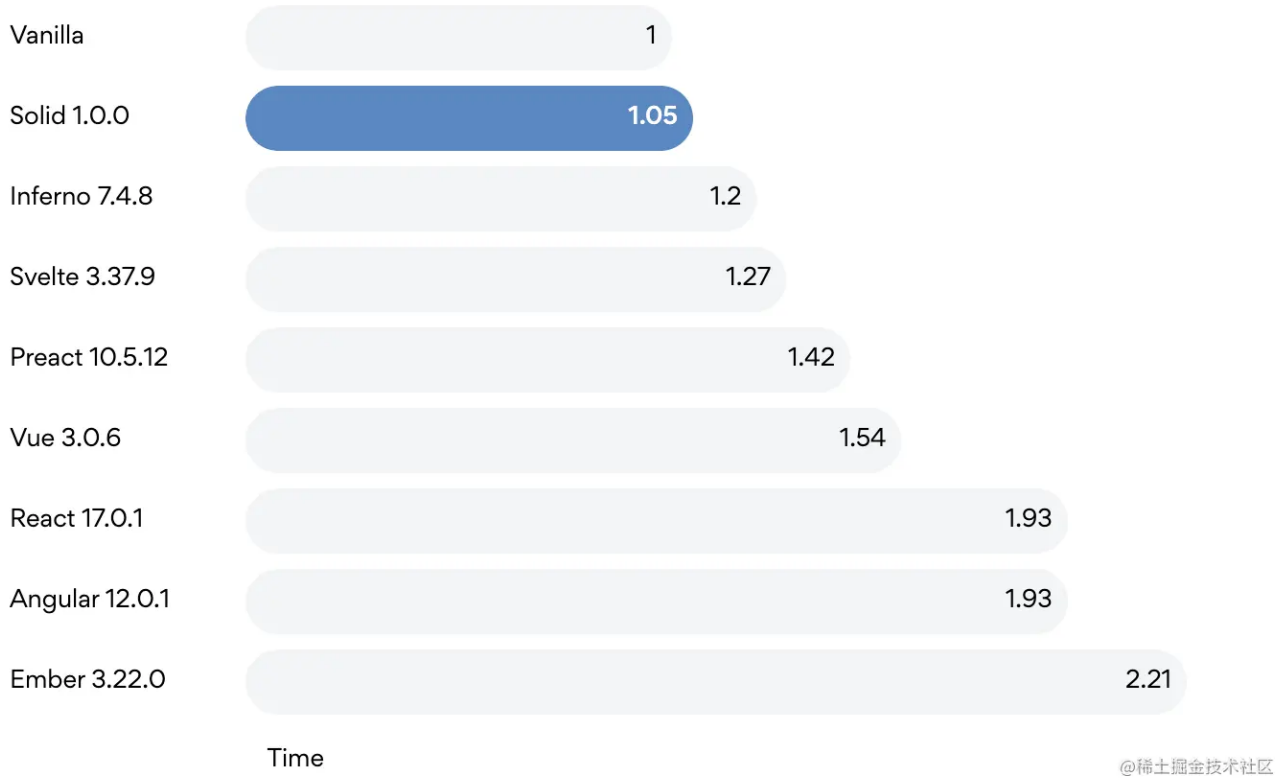
```
  const [count, setCount] = useState(0);
  return (
    <div className="App">
      <button onClick={()=>setCount((v)=>v+1)}>{count}</button>
      <Foo>
        <Bar />
      </Foo>
    </div>
  );
}

export default App;
```

重新 render 是为了对前后生成的 vdom 进行对比，但实际上，上面代码中的 Foo 组件和 Bar 组件都是无需更新的；在 React 16 版本之前，vdom 的对比是通过递归实现，如果组件树嵌套很深，那性能势必降低；React 16 之后，推出 fiber 架构，虽然省不掉必要的 render，但把递归 diff 改为可打断的循环，并且花费精力解决任务优先级调度问题，优化了用户体验。

但 React 花大力气解决的问题，在 Vue 中从来不是问题；Vue 在初始版本，都没有引入 vdom，通过 Object.defineProperty 的方式，直接绑定了数据更新和 dom 操作，设置省略了 diff 过程；在 Vue 2中，为了减少内存占用和实现跨平台，引入了 vdom，但依然保持了细粒度的更新逻辑，仅执行组件级别的 diff，而不会去执行整棵 dom 树的 diff；在 Vue 3 中，并没有跟进 React 实现 fiber，因为这是一件可以（花费大经理）但没必要（收益相对较小）的事情。Vue 在设计之初，因为 `Object.defineProperty` 的原因，无法通过 polyfill 的方式来进行兼容非主流浏览器。在 Vue 3 中，Proxy 的使用也意味着 IE11 以下版本浏览器无法使用。不过 2022 年了，微软自己都在跟 IE 告别~

性能



讲道理，随着硬件性能的提升，大多数框架在大多数场景下，一般是不会出现性能问题的。虽然跑分可能并不重要，但 React 的数据甚至比不过类 React 框架 Inferno、Preact 有些让人迷惑；再加上代码体积，Preact 为 3kb， React 为 40 +kb，纯 web 端有不少业务可以尝试进行迁移；不过，列举的这俩类 React 框架，并没有百分百实现 React 的 API，所以考虑到 React 的生态，迁移的成本和收益需要多做考量。

为什么类 React 框架这么多，而且体积更小、跑得比 React 还要快呢？而类 Vue 框架则相对较少呢？

> 这其实涉及一个难度问题，在angular里面自带了一个HTML parser与JS parser。 而在React里面， JSX parser已经被babel 插件外置了。因此少了最难的parser部分。
>
> 其次是 先行者的研究，我们可以做大量的裁减。
>
> 虽然React的库很大，但是它一半的代码都在事件系统中，而我们可以一个简单的事件系统（不用支持旧式IE，就不用管focus, blur, select,mouseenter,mouserleave, wheel,input 这些令人头痛的兼容问题）。代码量变成1.2~1.5万行了。
>
> 然后我们发现React还有四分之一是各种友好的错误提示，删掉这些用于生产环境的代码后，大概不到1万行。
>
> React仓库把每个模块划分得很细，因此这些我们不想用的东西可以轻松忽略掉。

> 然后我们看它最核心的虚拟DOM diff算法，打从最开始的virtual-dom库开始，人们就发现更快的diff优化方案，原来官方做的东西是不最优的（fb方面其实更关注于其平台覆盖度，让它可以在各种平台上运行，而不是DOM环境）。于是大家在virtual-dom上面的改动才会更来劲。diff算法基本上把字符串算法那套挪过来了，比如最短编辑长度距离，最长上升序列。。。
>
> 作者：司徒正美 链接：www.zhihu.com/question/34... 来源：知乎 著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

作者：@建忠