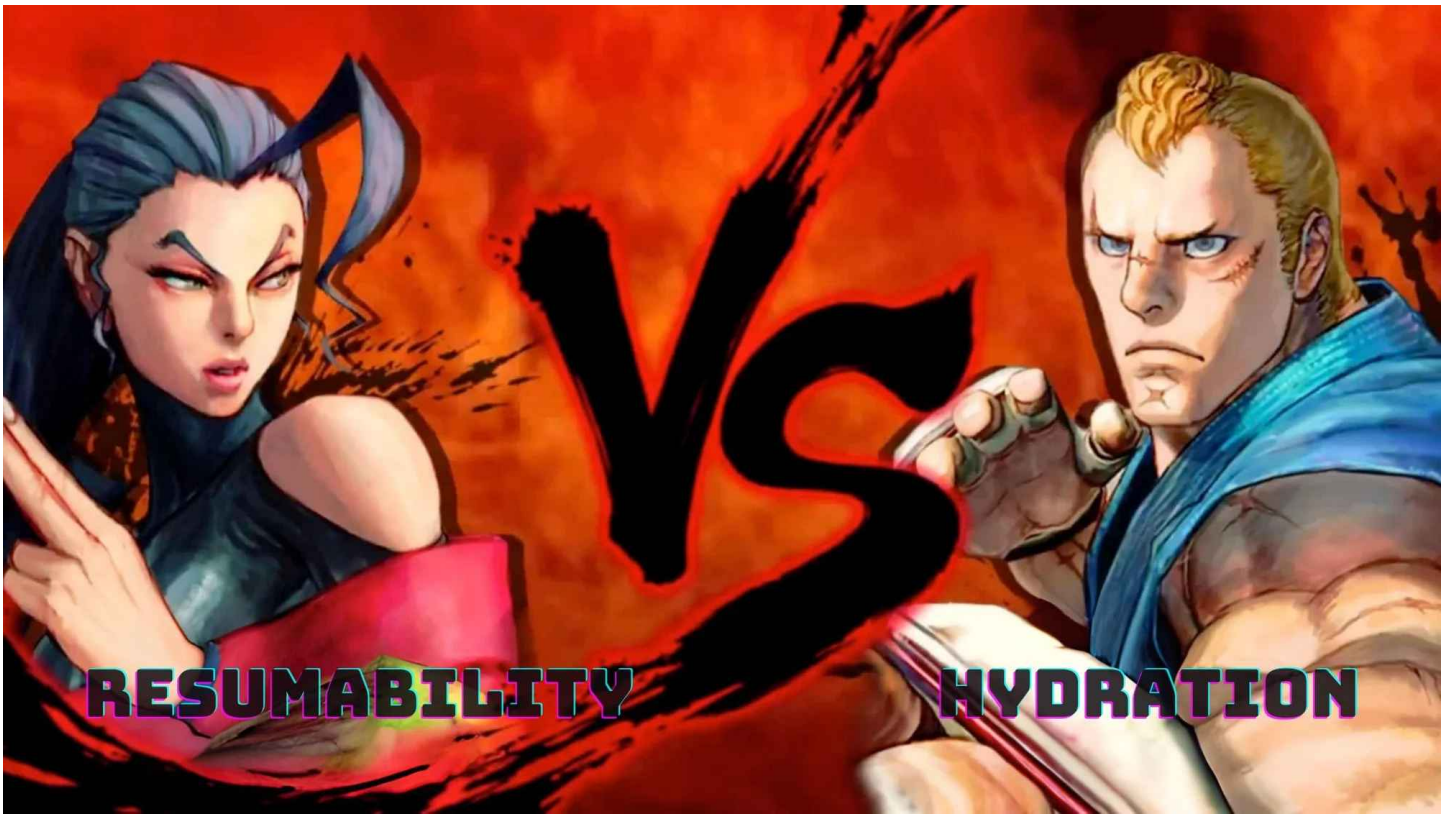
Resumability vs Hydration

> 本文翻译自Angular作者MIŠKO HEVERY，注意中文里水化、水合、注水通常都是指hydration，本文后面也将不做区分。



Resumability是比Hydration更快的替代方案。乍一看，Resumability和Hydration的作用似乎是相同的。毕竟，两者都可以为你的页面带来可交互性。但是如果将 Hydration 仅仅定义为「使页面具有交互性」太过于宽泛了，没有Hydration 页面仍然具有可交互性，比如CSR场景。

谈论Hydration的话，页面就一定有一个在服务端预渲染（如SSR或SSG）的步骤，如果没有这个步骤，那页面就仅仅是在客户端渲染，也就没有所谓的Hydration过程（这是一个重要的区别，因为Hydration和客户端渲染在大多数情况下做的事情是相同的）。

那么，如果 Hydration 和 Resumabality 做的事情是一样的，那么他俩有什么区别呢？我认为理解他俩之间差别的最佳方法是创建一个小示例，展示Hydration 和 Resumabality，并观察哪些代码在哪里（服务器与客户端）执行以及在什么时候（交互之前与之后）执行。

我们将选择三个框架来展示他们在Hydration 和 Resumabality上的不同能力：

- **React**: 一个很流行的框架，它会在Hydration和用户交互的时候执行组件代码(playground)。
- **SolidJS**: 一个细粒度响应式框架，会在Hydration的时候执行组件代码，在用户交互的时候不会做这个事情 (playground)

- **Qwik**: 一个细粒度响应式框架，压根儿就不会有Hydration这个过程，所以自然也不会存在「在Hydration的时候执行组件代码」(playground)

客户端渲染

在我们开始讨论 Hydration 和Resumability之前，回顾一下客户端渲染 (CSR) 的工作原理会很有帮助。要真正理解Hydration 和Resumability带来的好处，用一个把状态提升到最上层的层次化组件来做示例非常重要。

```tsx
import * as React from 'react';
import { useState } from 'react';
import './style.css';

export default function Counter() {
  console.log('Render: <Counter/>');
  const [count, setCount] = useState(0);
  return (
    <div>
      <Display count={count} />
      <Incrementor setCount={setCount} count={count} />
    </div>
  );
}

function Display(props: { count: number }) {
  console.log('Render: <Display/>');
  return <div>Current count: {props.count}</div>;
}

function Incrementor(props: {
  count: number;
  setCount: (count: number) => void;
}) {
  console.log('Render: <Incrementor/>');
  return (
    <button
      onClick={() => {
        console.log('Interaction: +1');
        props.setCount(props.count + 1);
      }}
    >
      +1
    </button>
  );
}
```

我们已经用三个不同的框架实现了上述代码。请注意，我们已将 console.log() 放置在关键位置以显示正在发生的情况。

		React	Solid	Qwik
		Blank HTML sent to client with app script only		
CSR	Render	<Counter/>	<Counter/>	<Counter/>
		<Display/>	<Display/>	<Display/>
		<Incremontor/>	<Incremontor/>	<Incremontor/>

如你所见，在客户端渲染情况下，三个框架没什么区别，他们都会执行全部的代码，所以有着相同的输出。这是有道理的，因为应用程序第一次执行和渲染时，框架别无选择，只能从根组件开始执行，然后下钻到子组件并一路执行每个组件。

现在让我们与组件进行交互（点击+按钮）并观察日志有何不同。

		React	Solid	Qwik
		Blank HTML sent to client with app script only		
Client Side Rendering	Render	<Counter/>	<Counter/>	<Counter/>
		<Display/>	<Display/>	<Display/>
		<Incremontor/>	<Incremontor/>	<Incremontor/>
	Interaction	+1	+1	+1
		<Counter/>		
		<Display/>		
		<Incremontor/>		

从这个时候开始，我们可以观察到三个框架的不同之处。因为点击了一下+1，所以所有三个框架都会输出+1，但是只有React会导致组件被重新执行，并且会导出打印出对应的日志。那为啥会有这样的区别呢?

React是一个粗粒度（coarse-grain）响应式系统，这意味着，React会从拥有状态的组件位置开始重新渲染，然后沿着组件树往下渲染。而Solid 和 Qwik是细粒度（fine-grain）响应式系统，这意味着状态的改变并不是与组件关联到一起，而是直接与需要更新的DOM节点关联在一起。最终的结果就是，React会重新执行组件，而Solid 和 Qwik只会更新对应的DOM节点。缩短组件的重新执行和重新创建 vDOM 是 Solid 成为最快框架之一的原因。

尽管在更新时执行更少的代码很重要，但如果我们可以在初始渲染时也避免执行，那岂不是更好？那么就让我们看看当涉及到 HTML 预渲染时会发生什么。

HTML预渲染

在CSR情况下，用户打开页面可能会看到很长时间的白屏，这是因为CSR在客户端执行JS，这个过程可能会比较耗时。 HTML 预渲染就是为了解决这个特定问题而引入的。预渲染的目标是向用户展示静态页面，并在背地里使页面具有交互性。

那如何预渲染 HTML呢？嗯，这很简单。你只需在服务端而不是在客户端上执行应用程序（SSR/SSG）。应用程序构建 HTML 后，你可以将 HTML 发送到客户端（或将其缓存在 CDN 中以供以后发送）。

		React	Solid	Qwik
Server	SSR SSG	<Counter/>	<Counter/>	<Counter/>
		<Display/>	<Display/>	<Display/>
		<Incremontor/>	<Incremontor/>	<Incremontor/>

请注意，就像 CSR 一样，SSR/SSG情况下所有框架都必须从根开始执行所有组件，只是这个过程是在服务端完成的。这是有意义的，因为执行应用程序代码是框架了解应用程序的方式。

让HTML变的可交互

一旦我们将预渲染结果发送到客户端，我们就需要使其具有交互性。最初，当引入 HTML 预渲染时，框架仍然会执行 CSR，只是用客户端渲染的内容替换静态内容。从这个意义上说，“Hydration”只是重建的过程。从那时起，CSR 变得更加聪明，现在它尝试尽可能重用 HTML，但从本质上讲，Hydration 仍然只是 CSR。

		React	Solid	Qwik
Server	SSR SSG	<Counter/>	<Counter/>	<Counter/>
		<Display/>	<Display/>	<Display/>
		<Incremontor/>	<Incremontor/>	<Incremontor/>
		HTML sent to client		
Client	Hydration	<Counter/>	<Counter/>	
		<Display/>	<Display/>	
		<Incremontor/>	<Incremontor/>	

现在我们开始看到了差异。请注意，**React 和 Solid 都必须在客户端上重新执行应用程序代码，而 Qwik 则不需要这样做**。也就是说**SSR情况下，React 和 Solid 都需要Hydration，而Qwik则压根儿不需要Hydration**。那么发生了什么？好吧，请记住，Hydration 只是换了个名字的 CSR。所以从这个角度上来说，这是有道理的。框架需要从根开始并向下到子级重新执行应用程序。此执行的目的是收集有关应用程序的信息，例如组件边界、事件监听器和响应式图（reactivity graphs）。

Solid 的优点是不做不必要的工作，但它仍然需要执行组件来重建细粒度的响应式图（reactivity graphs）。 Qwik 能够在服务器上将响应式图（reactivity graphs）序列化，因此在客户端上它可以跳过该步骤。

那在应用启动时执行代码有什么问题吗？嗯，这取决于您的应用程序的大小。在我们的演示DEMO中，这几乎没有什么区别。问题是这个成本与应用程序的复杂性成正比。当你开始一个项目时，Hydration不会是一个问题，但随着项目的发展，它就会成为一个致命的（an issue as a death）问题。

请记住，Hydration 本质上只是换了个名字的 CSR。 而因为CSR 速度是一个问题，所以我们引入了 HTML 预渲染。预渲染 HTML 有助于解决首次打开页面时页面空白时间较长的问题，但对解决交互没有任何作用。 （相反，可以提出这样的论点：添加预渲染的 HTML 会导致浏览器承担更多工作，因此延迟了可交互时间。）

预渲染完HTML之后再进行交互

让我们看看，服务端预渲染完，客户端完成事件绑定（Hydration）之后，再对页面进行交互时的情况：

		React	Solid	Qwik
Server	SSR SSG	<Counter/>	<Counter/>	<Counter/>
		<Display/>	<Display/>	<Display/>
		<Incremontor/>	<Incremontor/>	<Incremontor/>
		HTML sent to client		
Client	Hydration	<Counter/>	<Counter/>	
		<Display/>	<Display/>	
		<Incremontor/>	<Incremontor/>	
	Interaction	+1	+1	+1
		<Counter/>		
		<Display/>		
		<Incremontor/>		

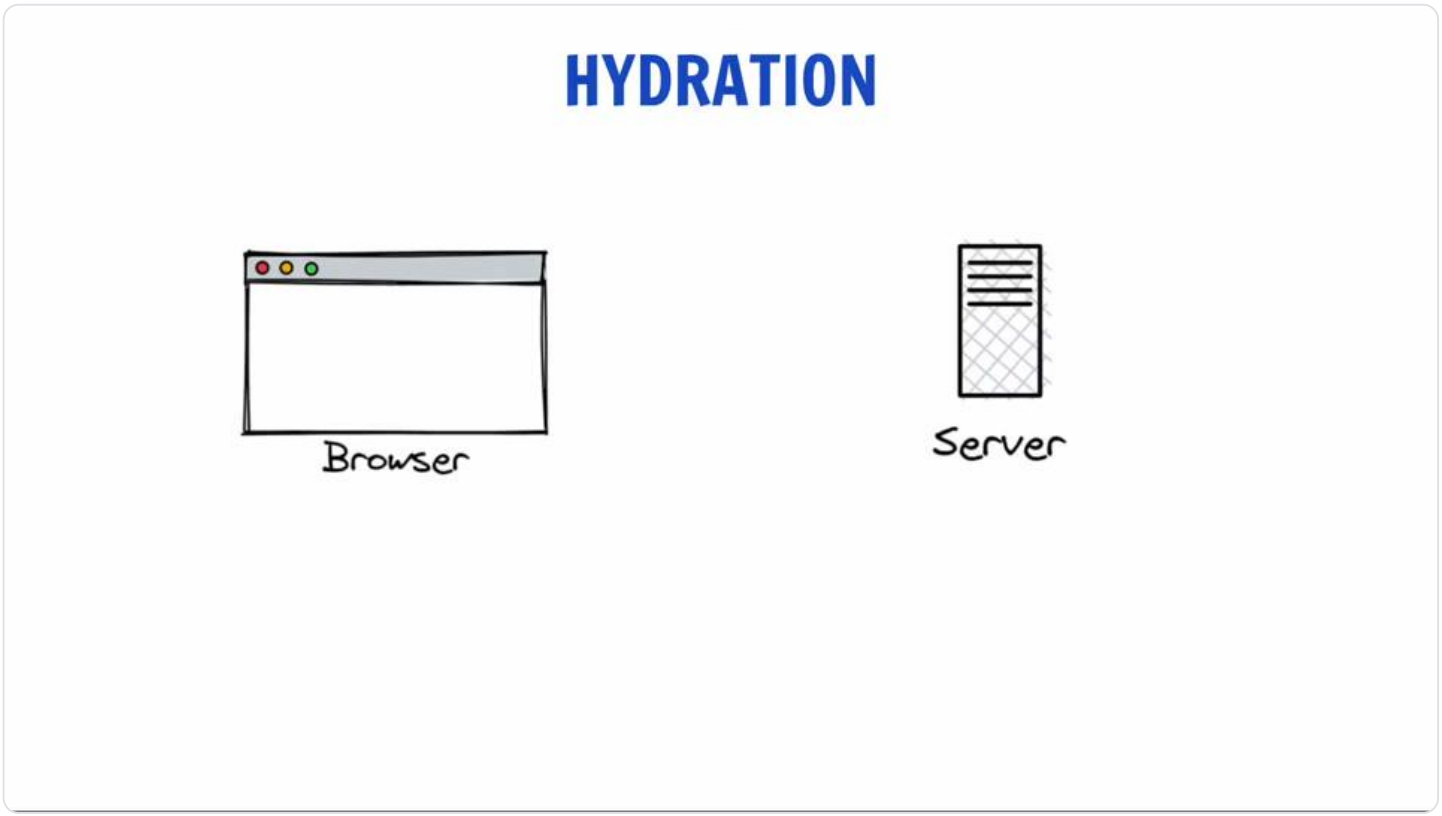
这个交互过程与纯CSR情况非常相似。 React 会导致所有组件重新执行，但 SolidJS 和 Qwik 只执行事件处理程序，不会重新渲染组件。

那么为什么这个很重要呢？请注意，Qwik 仅在服务器上执行组件，而从不在客户端上执行。无论是在初始的交互步骤还是后续交互步骤中。这意味着在这种情况下，Qwik 不需要将组件代码下载到客户端。 （在其他情况下，Qwik 可能必须下载组件代码，但这些情况并不常见。）

虽然 SolidJS 也能做到比较高效，即在交互时不执行组件代码，但它确实必须下载代码并在一开始就执行它以使应用程序具有交互性。尽管 SolidJS 永远不必再次执行代码，但下载和执行的代价已经付出了。

可恢复性（Resumability）

可恢复性是框架既可以恢复其状态而又无需在客户端上重新执行应用程序组件的一种方法。这是通过在 HTML 预渲染期间将应用程序状态和框架状态进行序列化来完成的。通过序列化组件边界、事件监听器和响应图，具有Resumability能力的框架可以在服务端停止的地方继续执行。客户端可以从服务器中断的地方恢复执行，从而节省了重新执行组件树（Hydration）以实现交互的成本。将其与细粒度响应式（fine-grained Reactivity）相结合后，框架在大多数情况下都不需要下载组件。



动画示例

即使我们想让Qwik 做 Hydration，它也无法做，因为 Hydration 代码（组件）并没有下载到客户端。

还有一点值得一提。Hydration必须在应用程序变得可交互之前执行。是的，尽管这个执行可能是惰性的，但是在 Hydration 执行之前按钮是不会处理事件的。而Resumability则是直接的。按钮在任何代码执行之前都是可交互的。

Resumability利用用户的交互来恢复框架的状态，而Hydration必须在交互之前运行。

代码重复

Hydration 是指应用程序被下载并执行两次，一次作为 HTML，另一次作为 JavaScript。Resumability则不会有这个重复。客户端会下载特定部分的 HTML 或 JavaScript，但很少同时下载两者。

延迟加载没有帮助

Hydration 要求框架执行组件来恢复事件侦听器和响应图。在组件树中插入延迟加载边界无助于防止代码被下载。在 Hydration 期间，框架将被迫尽早地下载并执行代码。

结论

我们构建了一个简单的应用程序，其状态、渲染和交互性分布在组件之间，以模拟框架如何处理组件执行。在 CSR/SSR/SSG 的初始渲染期间，不同框架的行为是没啥区分的。但交互性展示了细粒度响应式框架的优势，这使得框架能够显著减少客户端上需要呈现的代码量。将细粒度响应与可恢复性（Resumability）相结合，允许框架跳过大多数组件的下载和执行，从而实现即时启动。虽然 Hydration 和Resumability都使应用程序具有交互性，但它们的实现方式却是截然不同的。