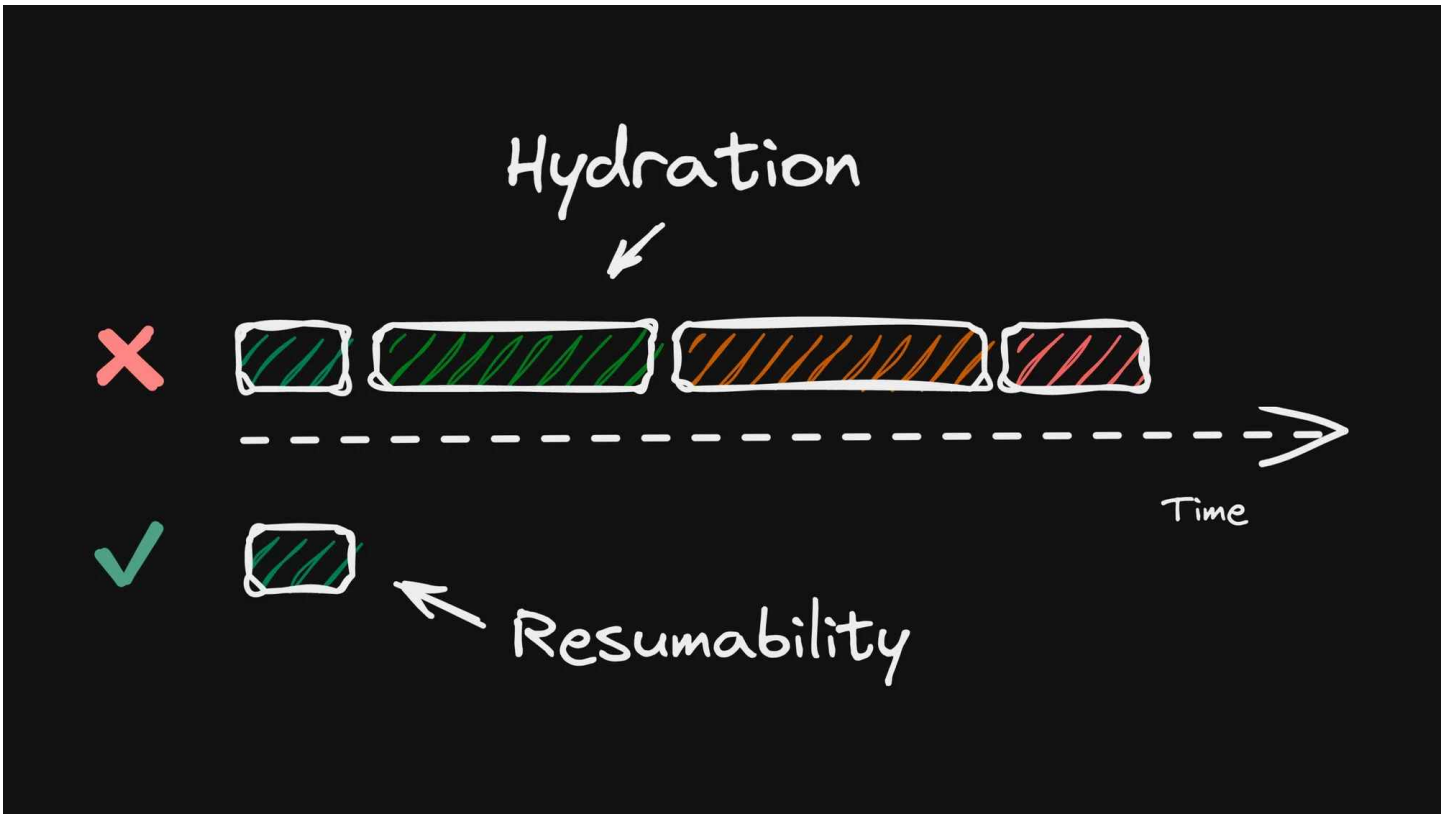
水化（hydration）是无谓的开销

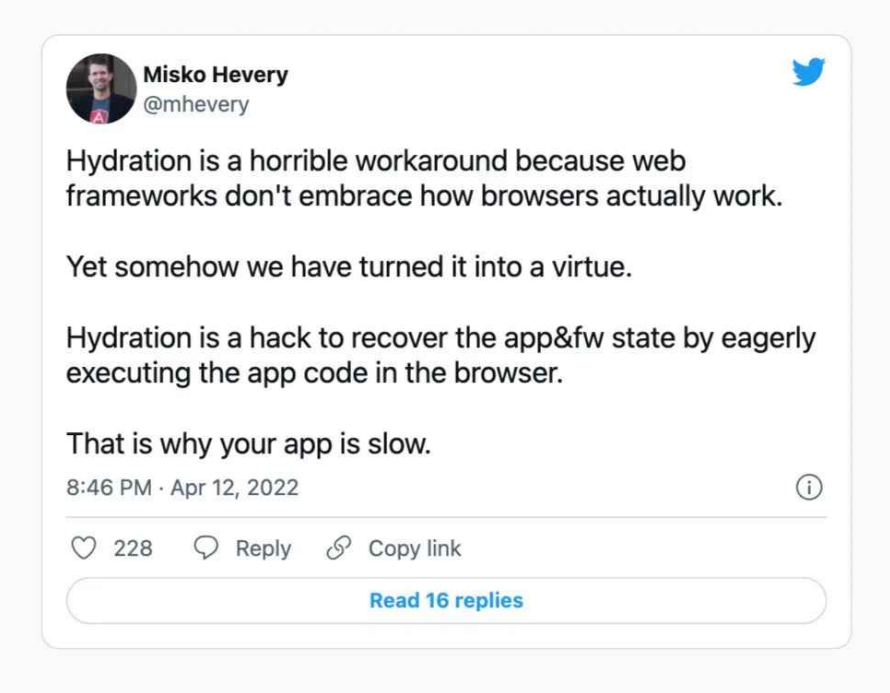
> 本文翻译自Angular作者MIŠKO HEVERY，注意中文里水化、水合、注水通常都是指hydration，本文后面也将不做区分。



水化（hydration）是让SSR渲染的HTML变得可交互的一种解决方案。维基百科是这么定义注水的：

> 在 Web 开发中，水化（hydration）或再水化（rehydration）是一种客户端 JavaScript 通过将事件处理器附加到 HTML（静态托管或由服务器端渲染） 元素从而将静态HTML 网页转换为动态网页的技术。

上面的定义从「将事件处理器（event handler）附加到静态 HTML 」这个角度讨论了hydration。然而，将事件处理器（event handler）附加到DOM并不是hydration中最具挑战或最耗时的步骤，并且它忽略了为什么有人将hydration称为开销的原因。而本文要聊的是，这个这个工作是可以完全避免的，而且即使没有这个工作仍然可以得到相同的最终渲染结果。这么来看的话，那hydration这个工作就是无谓的开销了。



深入理解hydration

hydration的困难部分是了解我们需要什么事件处理器（WHAT）以及它们需要附加在哪里（WHERE）。

- WHAT：事件处理器是一个包含事件处理行为的闭包。如果用户触发此事件，就会执行对应的程序。
- WHERE：需要附加事件处理器的 DOM 元素的位置（包括事件类型。）

这里增加的复杂度在于事件处理器是一个闭包，里面可能会包含APP_STATE和FRAMEWORK_STATE：

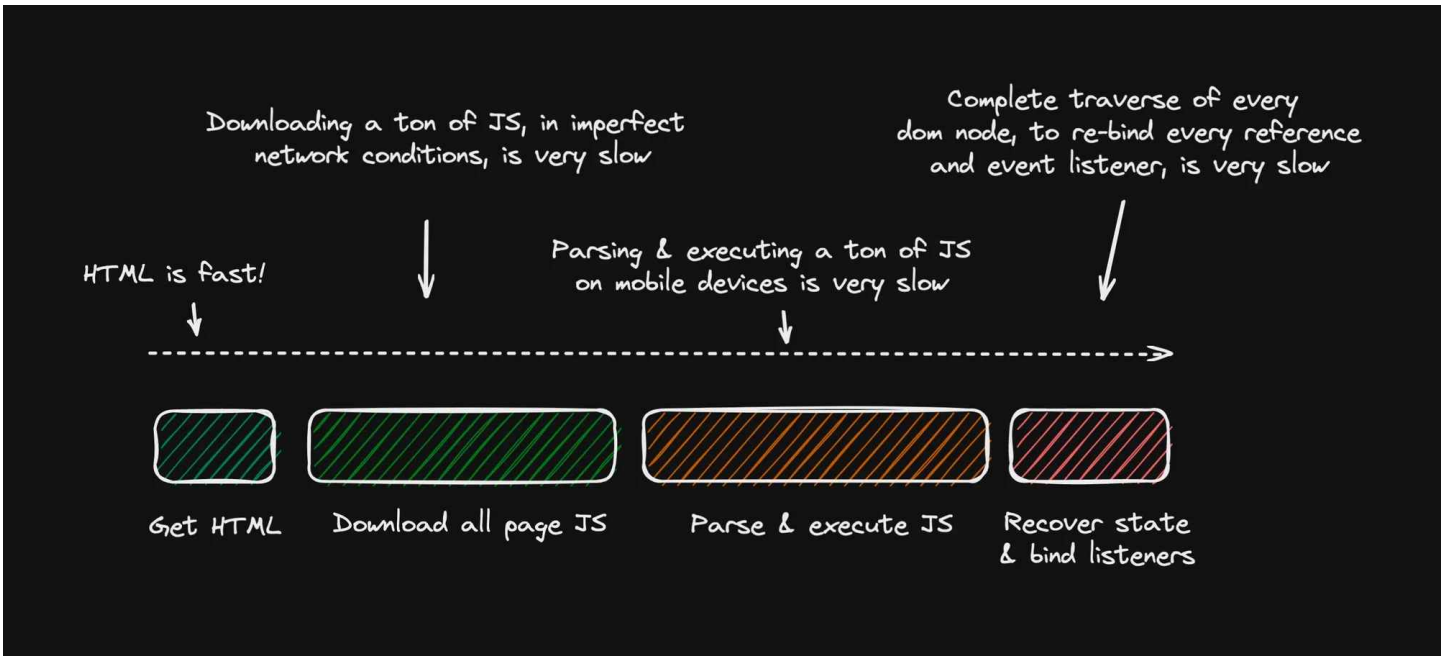
- APP_STATE：APP_STATE是大多数人所认为的状态。如果没有 APP_STATE，你的应用程序就无法向用户显示任何动态的内容。
- FRAMEWORK_STATE：FRAMEWORK_STATE表示框架的内部状态。如果没有 FRAMEWORK_STATE，框架不知道要更新哪些 DOM 节点或框架何时应更新它们。例如组件树和渲染函数的引用。

那么我们如何恢复这里提到的WHAT（APP_STATE + FRAMEWORK_STATE）和WHERE呢?

我们是通过下载并执行当前 HTML 中的组件代码。 然而，HTML 中渲染组件的下载和执行是很耗时的

换句话说，Hydration 是一种通过在浏览器中激进地执行应用程序代码来恢复 APP_STATE 和 FRAMEWORK_STATE 的 一种hack方式，涉及的步骤如下：

- 下载组件代码
- 执行组件代码
- 恢复 WHAT（APP_STATE 和 FRAMEWORK_STATE）和 WHERE 以获取事件处理器闭包
- 将 WHAT（事件处理器闭包）附加到 WHERE（DOM 元素）



我们将前三个步骤称为RECOVERY阶段。 RECOVERY 是指框架尝试重建应用程序时的情况。重建的成本很高，因为它需要下载并执行应用程序代码。

RECOVERY与水合页面的复杂性成正比，在移动设备上很容易花费 10 秒。由于RECOVERY是很耗时的过程，因此大多数应用程序的启动性能都不是最佳的，尤其是在移动设备上。

RECOVERY 也是没必要的开销。「没必要」是指已完成但不直接提供价值的工作。就Hydration而言，RECOVERY 是没必要的，因为它会重建服务器已经收集过的信息作为 SSR/SSG 的一部分。该信息并没有发送给客户端，而是被服务端丢弃。因此，客户端必须执行费时的RECOVERY来重建服务器已有的内容。如果服务器序列化了信息并将其与 HTML 一起发送到客户端，则可以避免 RECOVERY。序列化信息将使客户端免于下载和执行 HTML 中的所有组件。

在客户端上重新执行服务器已经执行的代码作为 SSR/SSG 的一部分，这使得Hydration成为纯粹的开销：也就是说，客户端重复服务器已经完成的工作。该框架本可以通过将信息从服务器传输到客户端来避免额外开销，但它却丢弃了这些信息。

总之，Hydration 是通过下载并重新执行 SSR/SSG 渲染的 HTML 中的所有组件来恢复事件绑定的。网页会被发送到客户端两次，一次作为 HTML，另一次作为 JavaScript。此外，框架必须立即执行 JavaScript 来恢复 WHAT、WHERE、APP_STATE 和 FRAMEWORK_STATE。所有这些工作只是为了获得服务器已经拥有但丢弃的东西！
为了理解为什么水合作用会迫使客户端重复工作，让我们看一个包含一些简单组件的示例。

```
export const Main = () => <>
   <Greeter />
   <Counter value={10}/>
</>

export const Greeter = () => {
  return (
    <button onClick={() => alert('Hello World!'))}>
      Greet
    </button>
  )
}

export const Counter = (props: { value: number }) => {
  const store = useStore({ count: props.number || 0 });
  return (
    <button onClick={() => store.count++)}>
      {count}
    </button>
  )
}
```

在完成SSR/SSG后，上述代码会生成如下HTML：

```
<button>Greet</button>
<button>10</button>
```

上述HTML不包含任何关于事件处理器或者组件边界的在什么位置的信息，也就是说生成的 HTML 不包含 WHAT(APP_STATE, FRAMEWORK_STATE) 或 WHERE。然而，其实在服务器生成 HTML 时，该信息就已存在了，但服务器未对其进行序列化。客户端可以做的使应用程序交互的唯一事情是通过下载和执行代码来恢复这些信息。

这里的要点是，必须先下载并执行代码，然后才能附加任何事件处理器并处理事件。代码执行后会实例化组件并重新创建状态（WHAT(APP_STATE, FRAMEWORK_STATE) 和 WHERE）。

一旦水合完成，应用程序就可以运行。单击按钮将按预期更新 UI。

可恢复性：水合的无开销替代方案

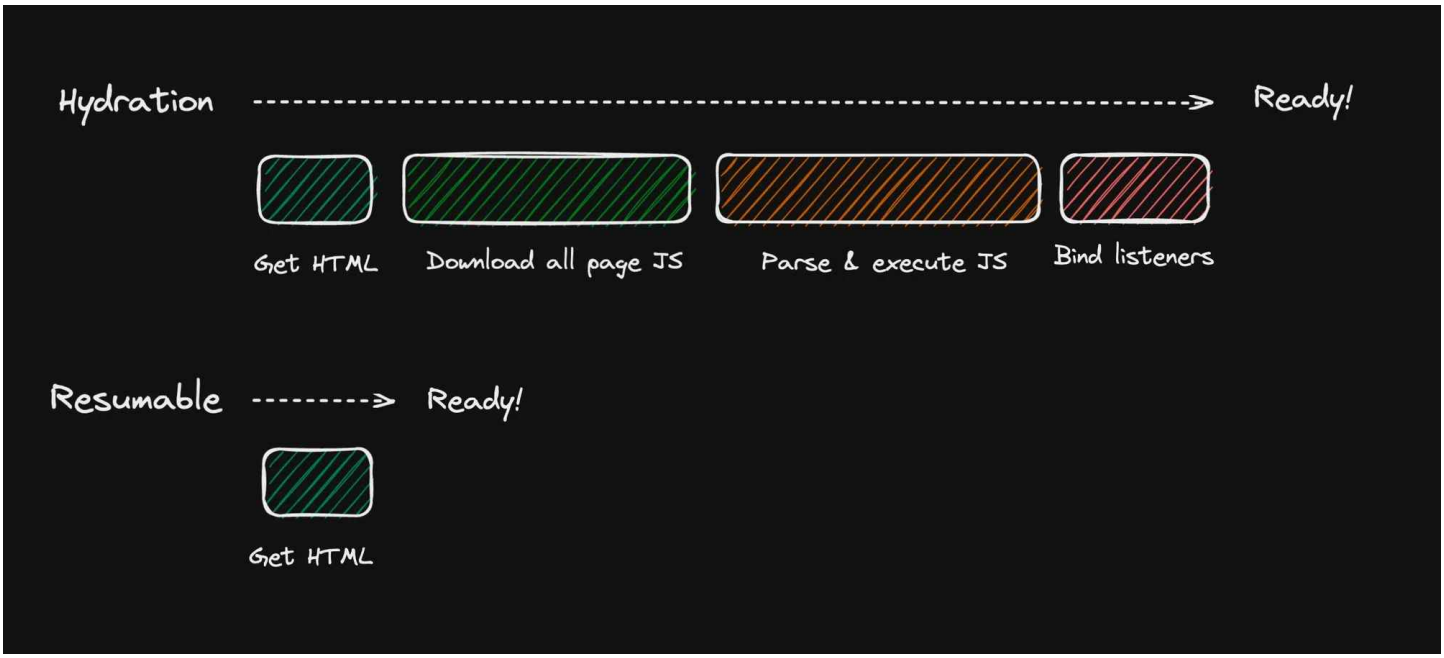
那么如何设计一个没有水合作用，也即没有开销的系统呢?

为了消除开销，框架不仅必须避免 RECOVERY，还必须避免上面的第四步。第四步是将“WHAT”附加到“WHERE”，这是可以避免的成本。

为了避免这个开销，你需要做三件事：

- 将所有必需的信息序列化为 HTML 的一部分。序列化信息需要包括WHAT、WHERE、APP_STATE和FRAMEWORK_STATE。

- 用依赖事件冒泡的全局事件处理器来拦截所有事件。事件处理器需要是全局的，这样我们就不必激进地在特定的 DOM 元素上单独注册所有事件。
- 一个可以延迟恢复事件处理器（WHAT）的工厂函数。



工厂函数是其中的关键，水合作用急切地创造“WHAT”，因为它需要“WHAT”将其附加到“WHERE”。相反，我们可以通过延迟创建 WHAT 作为对用户事件的响应来避免做不必要的工作。

上述设置是可恢复的，因为它可以从服务器停止的地方恢复执行，而无需重做服务器已经完成的任何工作。更重要的是，该设置没有任何开销，因为所有工作都是必要的，并且没有任何工作会重做服务器已经完成的工作。

思考差异的一个好方法是对照Push/Pull系统看：

- Push（Hydration）：急切地下载并执行代码以急切地注册事件处理器，以防用户交互。
- Pull（Resumability）：什么也不做，等待用户触发事件，然后延迟创建处理程序来处理该事件。

在 Hydration 中，事件处理器的创建发生在事件触发之前，因此是急切的。 Hydration 还要求创建并注册所有可能的事件处理器，以防用户触发事件（可能是不必要的工作）。因此事件处理器的创建是推测性的。这是可能不必要的额外工作。 （事件处理器也是通过重做服务器已经完成的相同工作来创建的；因此它是无谓的开销）。

而在可恢复系统中，事件处理器的创建是惰性的。因此，创建是在事件触发后发生的，并且严格按照需要进行。该框架通过反序列化事件处理器来创建事件处理器，因此客户端不会重做服务器已经完成的任何工作。

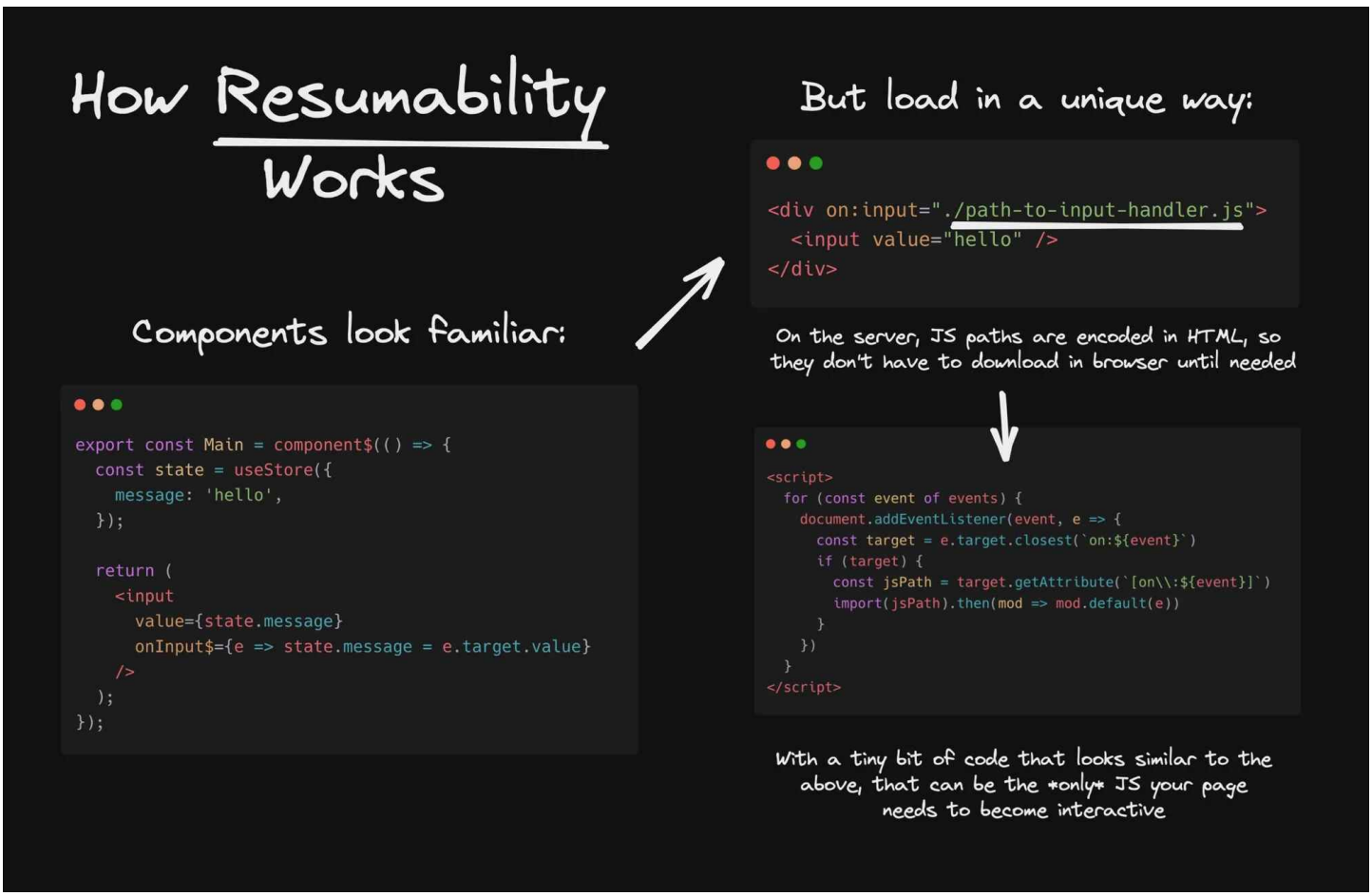
事件处理器的延迟创建是 Qwik 的工作原理，这使得它能够创建快速启动的应用程序。

可恢复性要求我们序列化 WHAT(APP_STATE, FRAMEWORK_STATE) 和 WHERE。可恢复系统可以生成以下 HTML 作为存储 WHAT(APP_STATE, FRAMEWORK_STATE) 和 WHERE 的可能解决方案。确切的实现细节并不重要，重要的是所有信息都存在。

```
<div q:host>
  <div q:host>
    <button on:click="./chunk-a.js#greet">Greet</button>
  </div>
  <div q:host>
    <button q:obj="1" on:click="./chunk-b.js#count[0]">10</button>
  </div>
</div>
<script>/* code that sets up global listeners */</script>
<script type="text/qwik">/* JSON representing APP_STATE, FRAMEWORK_STATE */</script>
```

当上述 HTML 在浏览器中加载时，它将立即执行设置全局侦听器的内联脚本。应用程序已准备好接受事件，但浏览器尚未执行任何应用程序代码。这是你能得到的最接近零 JS 的结果。

HTML 包含编码为元素属性的 WHERE。当用户触发事件时，框架可以使用 DOM 中的信息来延迟创建事件处理器。创建过程涉及 APP_STATE 和 FRAMEWORK_STATE 的延迟反序列化以完成“WHAT”。一旦框架延迟创建事件处理器，事件处理器就可以处理该事件。请注意，客户端不会重做服务器已完成的任何工作。



关于内存使用的注意事项

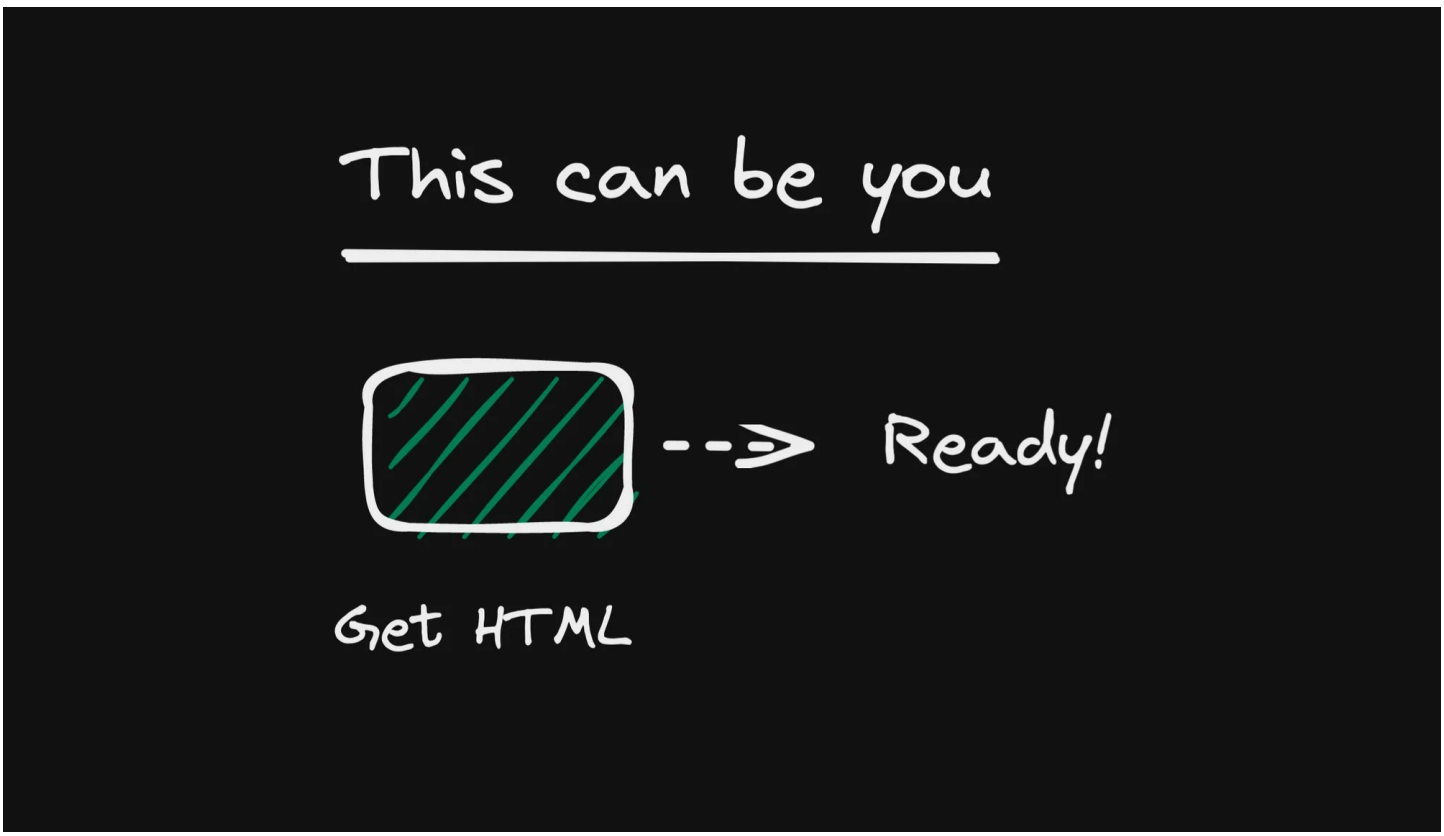
在Hydration中，为了保留DOM元素的事件，会尽快地创建所有的事件监听器。所以Hydration需要在启动时占用一些内存。

而Resumability在事件触发之前不会创建事件处理器。因此，Resumability比 Hydration 消耗更少的内存。此外，Resumability在执行后不会保留事件处理器。事件处理器在执行后被释放，返回内存。

在某种程度上，释放内容是与Hydration相反的过程。就好像框架懒洋洋地水合特定的“WHAT”，执行它，然后将其脱水。处理程序的第一次和第n次执行之间没有太大区别。事件处理器的惰性创建和释放不符合水合心智模型。

结论

Hydration是一项开销，因为它做的是重复性工作。服务器构建 WHERE 和 WHAT（APP_STATE 和 FRAMEWORK_STATE），但信息会被丢弃，而不是为客户端进行序列化。然后，客户端收到的 HTML 没有足够的信息来重建应用程序。信息的缺乏迫使客户端急切地下载应用程序并执行它以恢复 WHERE 和 WHAT（APP_STATE 和 FRAMEWORK_STATE）。



另一种实现类似Hydration效果的方法是Resumability。Resumability侧重于将所有信息从服务器传输到客户端。该信息包含 WHERE 和 WHAT（APP_STATE 和 FRAMEWORK_STATE）。附加信息允许客户端推断应用程序，而无需急切地下载应用程序代码。只有用户交互才会强制客户端下载代码来处理该特定交互。客户端不会重复服务器做过的任何工作；因此，Resumability没有任何开销。

为了将这个想法付诸实践，我们构建了 Qwik，一个围绕Resumability设计的框架，并实现了出色的启动性能。我们也很高兴收到您的来信！让我们继续对话，并作为一个社区更好地为我们的用户构建更快的 Web 应用程序。