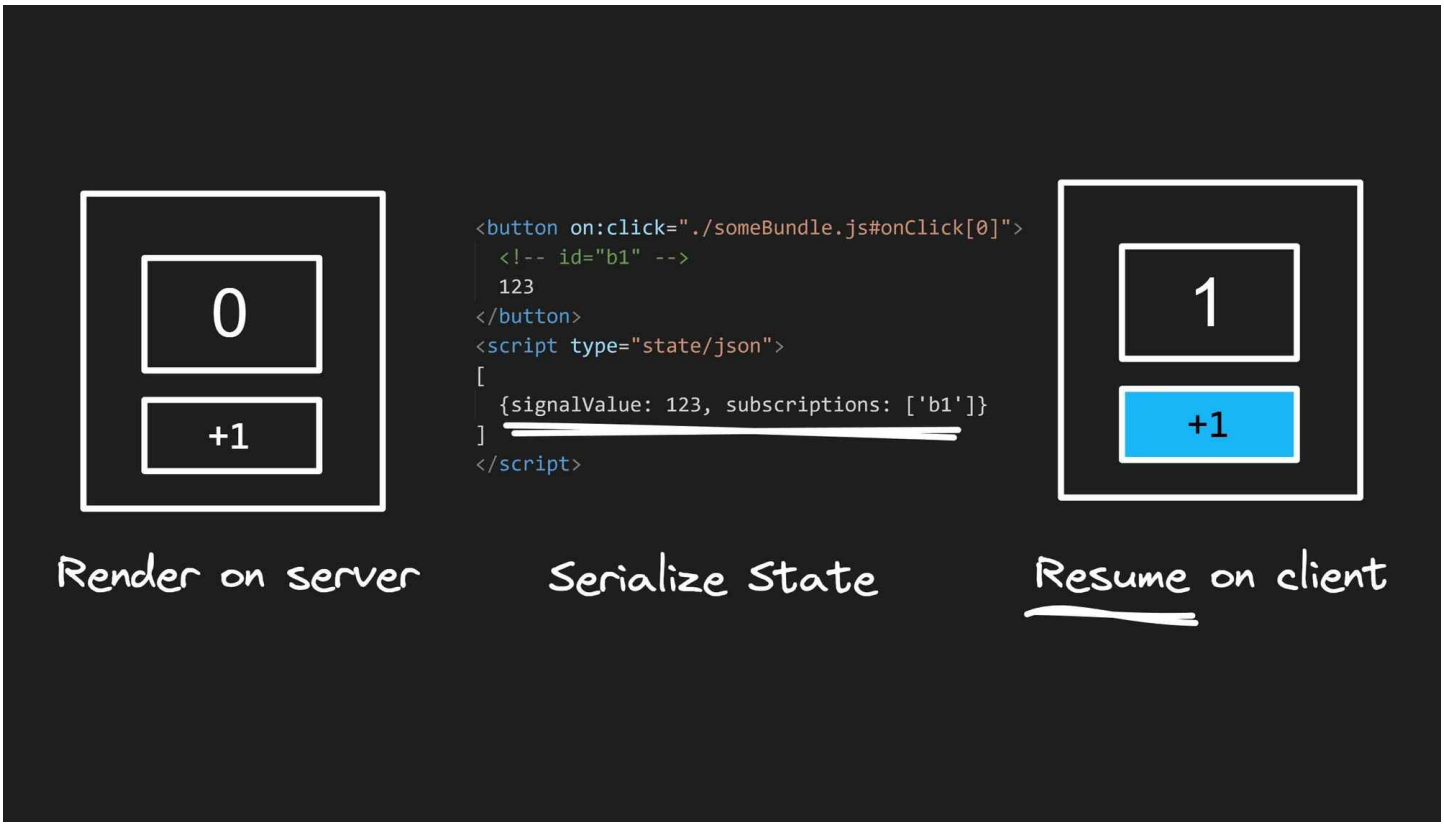
构建自己的Resumability框架

> 本文翻译自Angular、Qwik作者MIŠKO HEVERY，在之前论述Resumability vs Hydration基础之上进一步讲述了如何实现Resumability的细节。



理解一个事情最好的方式就是自己实现一遍。所以让我们来一步步的实现Resumability的各个步骤，更好的理解为啥它这么快，一层层地来揭开它的神秘面纱。

我们从一个最简单的应用开始：

```
export function MyApp () {
  console.log('Render: MyApp');
  return <Greeter/>;
}

function Greeter() {
  console.log('Render: Greeter');
  const onClick = () => alert('hello world');
  return (
    <button onClick={onClick}>
      greet
    </button>
  );
}
```

Resumability需要在SSR或者SSG情况下才会有，所以在SSR/SSG场景下，执行上述代码后，服务端会输出：

```
<button>greet</button>
```

如果你看服务端的日志，你会看到我们已经执行了 `MyApp` 和 `Greeter` ，所以会有如下输出：

```
Render: MyApp
Render: Greeter
```

所有这些都很简单明了，所有的框架都是这么干的。

上面的例子很简单。但是假设我们不是只有2个组件，而是有一个复杂的组件树。进一步假设执行所有的组件、提取所有的事件监听器、把它们绑定到DOM上会比较耗时（事实上也确实是这样）——不管是从需要下载的JS量还是需要执行的JS量来看都是如此。Hydration 在旧手机和比较慢的网络条件下可能会耗时 15+秒。

让它变的可交互

我们需要让上面的HTML变的可交互，这意味着我们需要：

- 获取到onClick事件监听器
- 将它附加在button上

获取到事件监听器并且弄清楚需要把它附加到DOM的哪个地方是让应用变成可交互必须解决的最基本的问题。

Hydration

那怎样才能获取到onClick事件监听器呢？好吧，可以注意到，我们唯一导出的只是 `MyApp` 。这意味着，框架必须从根组件开始执行所有的代码才能有机会获取到 `onClick` （以及button的位置）。没有别的路子。应用程序打包后会有一个导出符号，因此框架的唯一选择就是从那里开始执行。

从根组件开始意味着需要下载整个App的代码，按tree shaker的定义来讲，如果你对根组件有引用，那它也做不了啥。

> 注意：懒加载可以用于组件树里当前不需要的渲染的组件中，但那是一个更深入的话题，这里将不会进行讨论

新的入口

所以，如果我们想要在不从 `MyApp` 开始遍历所有树的情况下获取到 `onClick` 事件处理器，我们需要改变代码的打包结构。我们需要一种直接获取到 `onClick` 的方式。这里说的“直接”指的是不需要遍历组件树。那这就意味着 `onClick` 需要被导出。

所以让我们重写示例应用，将 `onClick` 放在顶层导出：

```
export function MyApp () {
  console.log('Render: MyApp');
  return <Counter/>;
}

export const onClick = () => alert('hello world');

function Counter() {
  console.log('Render: Counter');
  return (
    <button onClick={onClick}>
      greet
    </button>
  );
}
```

> 注意：日常中我们像这样编写代码是极不自然的，因此我们稍后将讨论如何自动化做这个事情。

现在 `onClick` 放在顶层导出了，所以框架做下面的事情变成了可能：

```
button.addEventListener('click', onClick);
```

就是这样，我们的应用程序变的可交互了。无需从顶层（`MyApp`）开始执行所有的代码来查找事件监听器。这可以节省大量的（substantial）时间，尤其是在移动设备上。

另外可注意，tree shaker现在可以从bundle中删除 MyApp 和 Greeter 组件。我们不再引用他们！这可以节省时间，因为这两个组件在我们的应用程序中不执行任何操作。这只是死代码。所以现在我们没必要浪费带宽来传输它们。

序列化事件监听器

好吧，这个事情并没有那么简单。框架怎么知道哪个按钮、事件和导出函数必须由 addEventListener 注册呢?

```
button.addEventListener('click', onClick);
```

而且，执行`addEventListener`仍然是代码，并且这个代码需要尽早地被执行从而完成事件绑定。我们可以解决这个问题吗？有没有可能不执行任何代码?

事件委托

上面提到的几个问题都可以用事件委托来解决。

1. 首先，我们在根DOM上创建一个`addEventListener`，它会依赖事件冒泡拦截所有事件（这个单一的全局事件监听器将会保留。这意味着，无论我们在DOM中有多少个事件监听实例，都只会有一个全局监听器。）
2. 然后，我们将事件监听器的位置、类型进行序列化，并且在HTML中引入对应的URL地址。

在这种情况下，我们的服务端输出的会是下面这种：

```
<button on:click="./someBundle.js#onClick">greet</button>
```

而不是之前的这种：

```
<button>greet</button>
```

注意到，所有我们需要的信息都埋藏在HTML中：

1. on:click的属性告诉我们事件监听器是需要放在button上
2. 属性名on:click则告诉我们它是一个click事件
3. 属性值（"./someBundle.js#onClick"）则告诉我们需要引入哪个bundle（./someBundle.js）以及事件触发后需要执行bundle中的哪块代码（onClick）。

当button被点击时，浏览器会进行事件冒泡。全局事件监听器捕获冒泡事件。然后，全局侦听器查找 on:click 属性并获取事件处理器代码并执行它。这种策略会延迟获取事件处理程序，而不是急切地执行 addEventListener，后者要求事件监听器在执行时就已经存在。

在这种策略下，浏览器只会执行最少量的必要代码。一旦服务器完成其工作，浏览器就会从服务器中断的地方恢复。这种可恢复性实现了「最少量的代码传递」和「代码的延迟执行」，最终带来更快的启动时间、更少的网络使用和更长的电池寿命。

状态处理

上面的示例缺少让应用之所以称为应用的一些必要特性——状态和数据绑定。如果没有状态和数据绑定，应用只不过是静态页面。所以，接下来我们重新审视所有的步骤，但这次要考虑到状态和数据绑定。

我们把之前的实例改改，让它具有状态，这里把 `Greeter` 换成了 `Counter`：

```
export function MyApp () {
  console.log('Render: MyApp');
  return <Counter/>;
}

function Counter() {
  console.log('Render: Counter');
  const count = useSignal(123);
  const onClick = () => count.value++;
  return (
    <button onClick={onClick}>
      {count.value}
    </button>
  );
}
```

导出闭包

在这里我们遇到了问题：

```
export const onClick = () => {
  count.value++; // ERROR: `count` is not declared
}

function Counter() {
  console.log('Render: Counter');
  const count = useSignal(123);
  return (
    <button onClick={onClick}>
      {count.value}
    </button>
  );
}
```

我们需要解决的根本问题是导出的函数无法将组件的状态（count）进行闭合。我们如何将count从Counter 传递到 onClick 侦听器呢?

重写捕获的变量

让我们从写代码，将onClick从捕获count改成用另外一种方式获取count：

```
export const onClick = () => {
  const [count] = __closedOverVars__;
  count.value++;
}

function Counter() {
  console.log('Render: Counter');
  const count = useSignal(123);
  return (
    <button onClick={withClosedOverVars(onClick, [count]}>
      {count.value}
    </button>
  );
}

let __closedOverVars__;
function withClosedOverVars(eventHandler, consts) {
  return (...args) => {
    __closedOverVars__ = consts;
    try {
      return eventHandler(...args);
    } finally {
      __closedOverVars__ = null;
    }
  }
}
```

通过改写onClick事件处理器代码，我们从全局变量上获取事件处理器原本需要的闭合的变量（count），这样我们的onClick就可以作为顶层进行导出了。并且它的效果和之前是一样的。

状态序列化

上面解决了我们能够获取现有组件深处的闭包的问题。但是，如果我们想通过全局侦听器调用事件处理程序，则需要在调用它之前恢复 closeOverVars 初始的状态。那么我们如何恢复count的状态呢？

请注意，count包含2部分信息：

- 状态：在本例中是123。
- 数据绑定：本例中对应的是状态绑定到button的文本节点

因此，让我们看看是如何通过除了将事件的位置序列化之外，华江关联的状态和数据绑定信息进行序列化，从而恢复所有这些信息的。新的 HTML 可以存储这样的附加信息：

```
<button on:click="./someBundle.js#onClick[0]">
  <!-- id="b1" -->
  123
</button>
<script type="state/json">
[
  {signalValue: 123, subscriptions: ['b1']}
]
</script>
```

上面对HTML进行了一些扩展，增加了关于状态和绑定相关的额外信息：

1. 注意，on:click处理器现在包含了额外的信息 `[0]` 。它表明在第0个位置的状态需要被反序列化用于函数执行（可以往下看）
2. `<!-- id="b1" -->` 表示它后面的文本节点是数据绑定，并为其分配唯一且任意的ID： b1。
3. 而 `<script>` 标签包含一个编码好的JSON数据：包含一个初始值为123的 Signal ，并且这个数据绑定到b1节点（更新这个Signal需要相应地去更新相关的DOM节点）

有了上述所有信息后，现在可以在不下载或执行 MyApp 或 Counter 的情况下调用 onClick 方法。现在，应用程序可以执行 onClick 处理函数，而无需执行应用程序的任何初始化代码。应用程序从服务器中断的地方完成了恢复。

自动代码重写和 $() 标记

没有人愿意用如此尴尬的语法编写应用程序。我们需要对语法做一些处理，使其更加自然。

让我们创建一个代码预处理器（编译器），它可以自动将我们的自然应用程序代码重写为resumability所需的格式。该转换是程式化的并且是易于实现的。

我们需要一种方法来注释源代码，以便预处理器（和其开发人员）知道在什么时候对源代码应用这种转换。

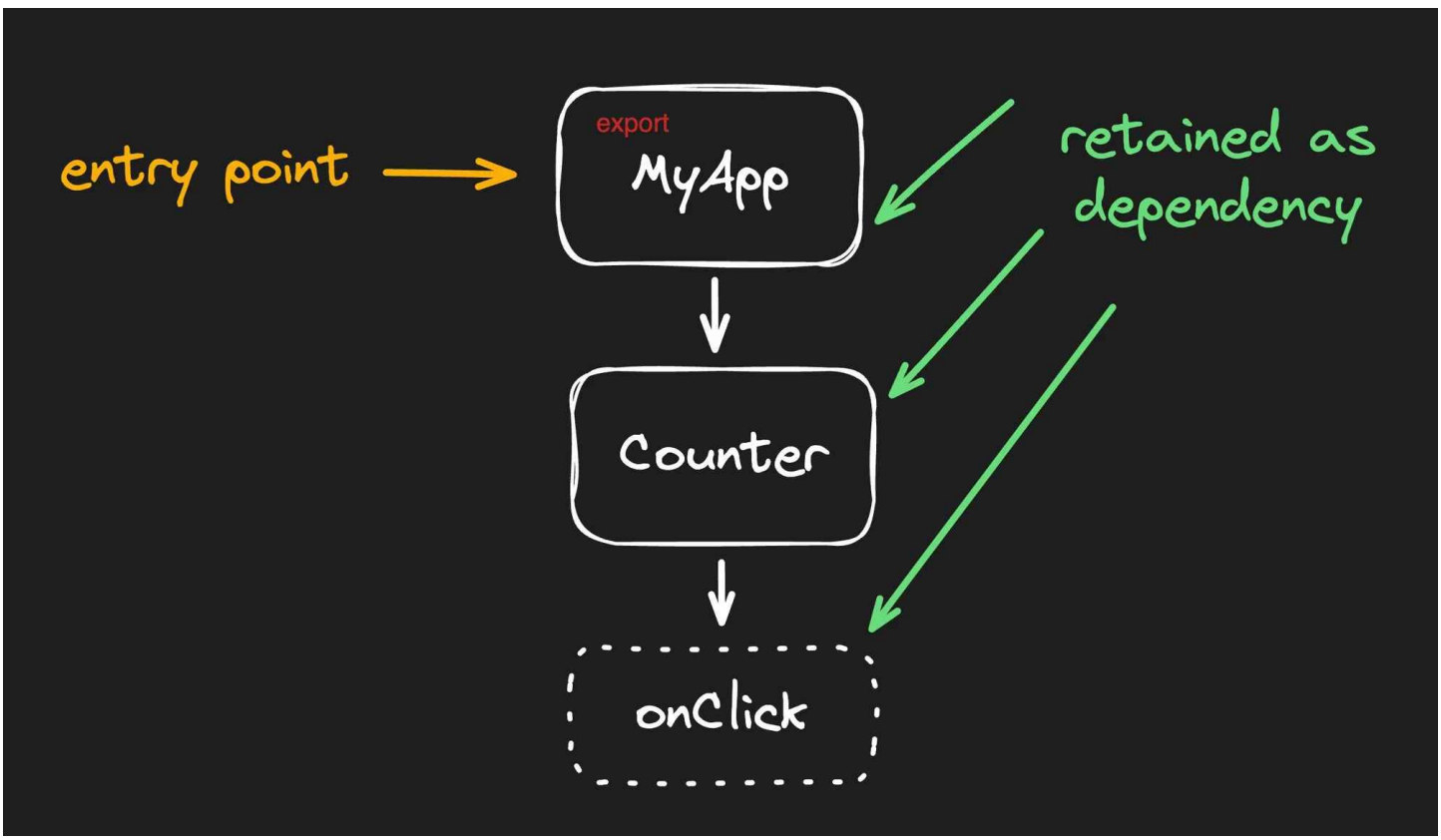
```
export const MyApp = component$(() => {
  console.log('Render: MyApp');
  return <Counter/>;
});

export const Counter = component$(() {
  console.log('Render: Counter');
  const count = useSignal(123);
  return (
    <button onClick$={() => count.value++}>
      {count.value}
    </button>
  );
})
```

这样预处理器就会知道 onClick 需要被提取到顶级函数中并进行导出。

> 注意：我们也将函数用 `component$()` 进行了包裹，这样他们也可以被独立的加载（后面将会讨论）

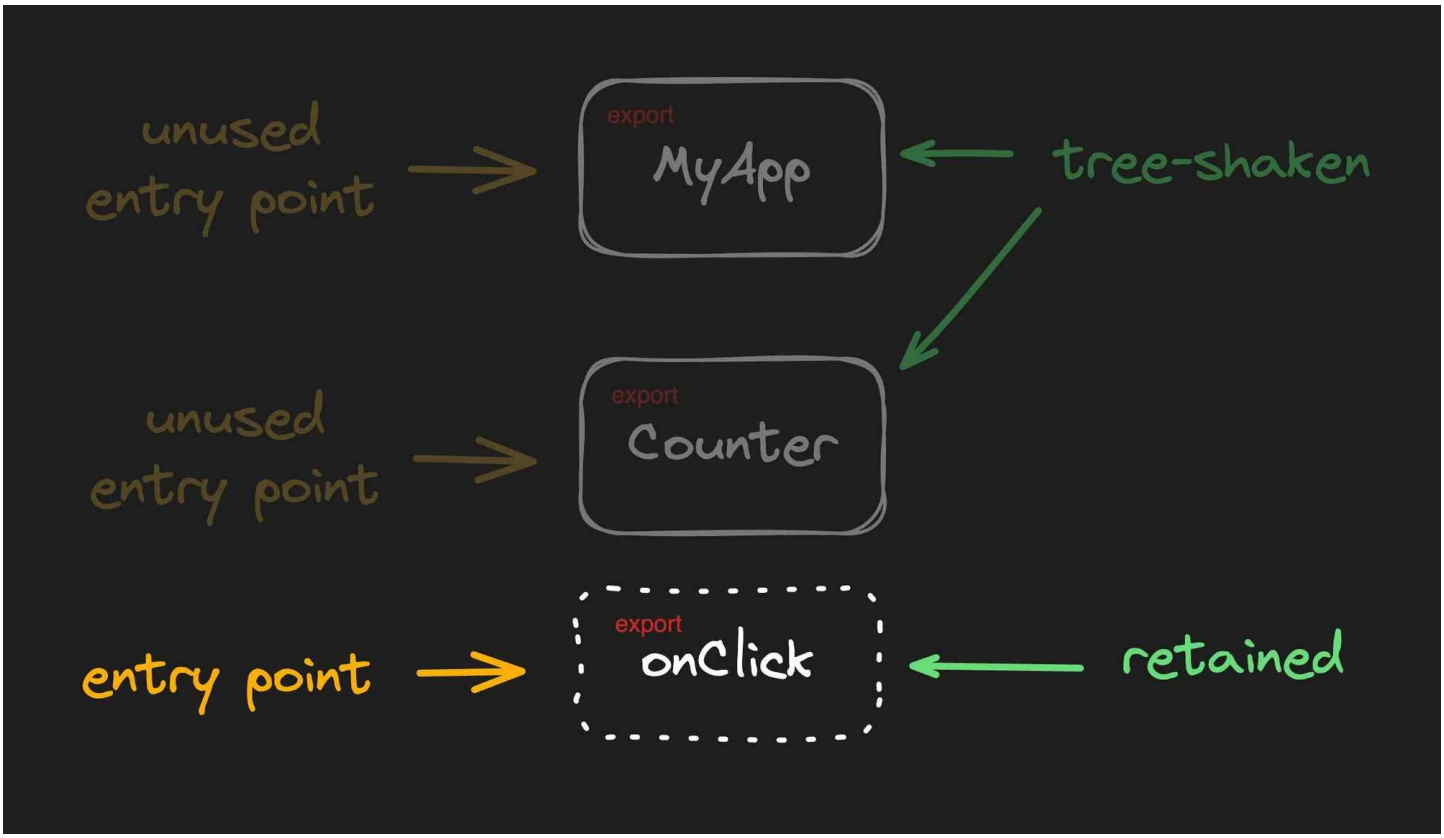
可恢复性意味着创建多个入口点

我们的应用程序是组件组成的组件树。通常，我们只从树中导出根组件，并且只将根组件传递给框架的启动方法。



如果你的应用程序只有一个入口点，那么除了在根组件处启动应用程序并递归遍历组件树以了解应用程序的状态、绑定和事件监听器之外，很难做任何其他的操作。

而Resumability是指从服务器中断的地方继续。Resumability要求每个事件监听器都可以直接import，并且应用程序状态可以序列化。



可恢复性是将应用程序从单个入口点转换为多个入口点。并且入口点需要是事件监听器，因为浏览器总是作为对事件的响应来恢复执行的。

可恢复性意味着打破依赖关系

可恢复性是指不下载不需要在浏览器上重新执行的代码。这意味着可恢复性需要为打包器提供选择，以便打包器可以对仅需要水合作用的代码进行摇树（tree-shake）。

如果我们只将顶级组件导出的话，打包器无法对任何内容进行tree-shake。

这就是为什么每个组件都包装在 component$() 中——用于打破父组件和子组件之间的依赖关系。有时某个组件必须下载并执行，但我们不想强制下载并执行它的子组件。

可恢复性意味着包体积优化

可恢复性要求很多东西都有顶级的导出。我们还确保父组件和子组件之间没有直接依赖关系。这意味着顶级内容往往只具有较浅的依赖图。

这使得在包之间移动导出的某个内容变得容易，并且可以在不更改源代码的情况下拥有单个或多个包。打包变成了为打包器提供配置信息，并且可以在不重构源代码的情况下进行微调。

可恢复性有很多好处，不仅局限于在慢速网络或移动设备上快速启动应用程序。