HMR API 及原理：代码改动后，如何进行毫秒级别的局部更新?

发布于 2022-05-09

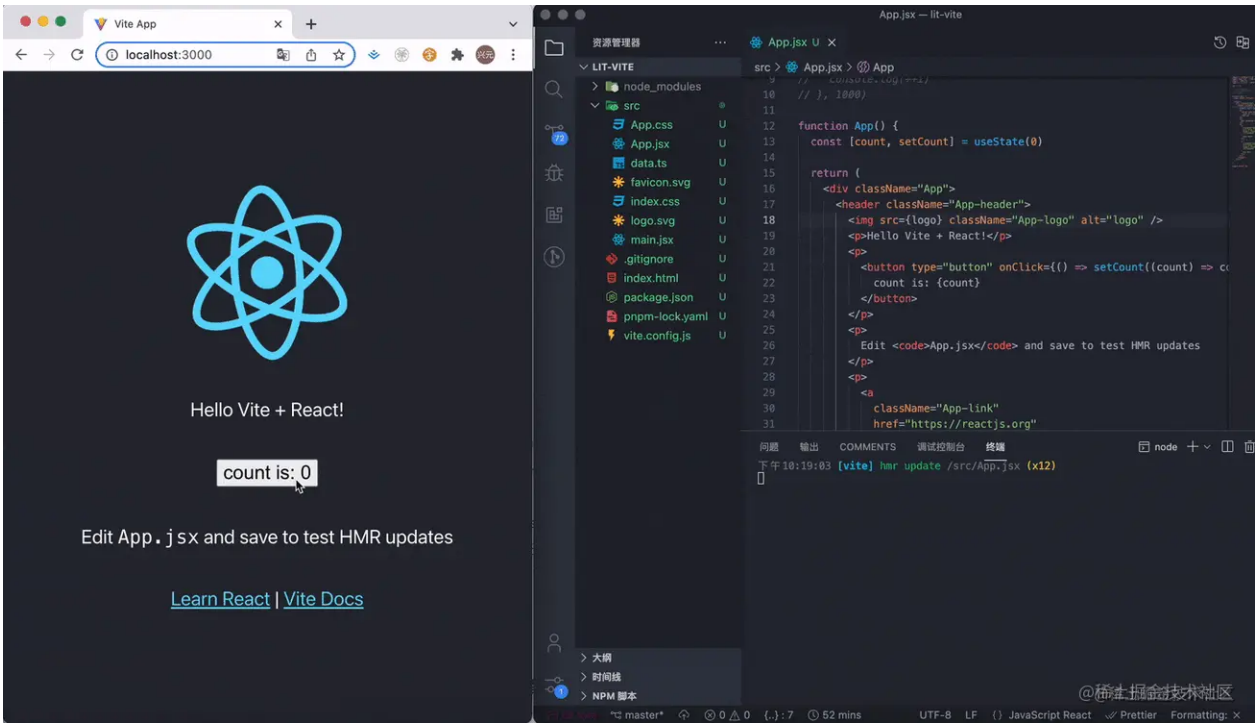
在开始今天的课程之前，我想先问你一个日常开发中的问题：在代码变更之后，如何实时看到更新后的页面效果呢?

很久之前通过 live reload也就是自动刷新页面的方式来解决的。不过随着前端工程的日益庞大，开发场景也越来越复杂，这种 `live reload` 的方式在诸多的场景下却显得十分鸡肋，简单来说就是 模块局部更新 + 状态保存 的需求在 `live reload` 的方案没有得到满足，从而导致开发体验欠佳。当然，针对部分场景也有一些临时的解决方案，比如状态存储到浏览器的本地缓存(localStorage 对象)中，或者直接 mock 一些数据。但这些方式未免过于粗糙，无法满足通用的开发场景，且实现上也不够优雅。

那么，如果在改动代码后，想要进行模块级别的局部更新该怎么做呢？业界一般使用 HMR 技术来解决这个问题，像 Webpack、Parcel 这些传统的打包工具底层都实现了一套 HMR API，而我们今天要讲的就是 Vite 自己所实现的 HMR API，相比于传统的打包工具，Vite 的 HMR API 基于 ESM 模块规范来实现，可以达到毫秒级别的更新速度，性能非常强悍。接下来，让我们一起来谈谈在 Vite 当中，这一套 HMR 相关的 API 是如何设计的，以及我们可以通过这些 API 实现哪些功能。

HMR 简介

HMR 的全称叫做 `Hot Module Replacement` ，即 模块热替换 或者 模块热更新 。在计算机领域当中也有一个类似的概念叫 热插拔 ，我们经常使用的 USB 设备就是一个典型的代表，当我们插入 U 盘的时候，系统驱动会加载在新增的 U 盘内容，不会重启系统，也不会修改系统其它模块的内容。HMR 的作用其实一样，就是在页面模块更新的时候，直接把**页面中发生变化的模块替换为新的模块**，同时不会影响其它模块的正常运作。具体来说，你可以观察下面这个实现 HMR 的例子。



在这里，我改变了页面的一个状态 `count` ，当我对页面再次进行调整的时候，比如把最上面的 Logo 图片去掉，这个时候大家可以实时地看到图片消失了，但其他的部分并没有发生改变，包括组件此时的一些数据。

如此一来，通过 HMR 的技术我们就可以实现 局部刷新 和 状态保存 ，从而解决之前提到的种种问题。

深入 HMR API

Vite 作为一个完整的构建工具，本身实现了一套 HMR 系统，值得注意的是，这套 HMR 系统基于原生的 ESM 模块规范来实现，在文件发生改变时 Vite 会侦测到相应 ES 模块的变化，从而触发相应的 API，实现局部的更新。

Vite 的 HMR API 设计也并非空穴来风，它基于一套完整的 ESM HMR 规范来实现，这个规范由同时期的 no-bundle 构建工具 Snowpack、WMR 与 Vite 一起制定，是一个比较通用的规范。

我们可以直观地来看一看 HMR API 的类型定义:

```
interface ImportMeta {
  readonly hot?: {
    readonly data: any
    accept(): void
    accept(cb: (mod: any) => void): void
```

```
    accept(dep: string, cb: (mod: any) => void): void
    accept(deps: string[], cb: (mods: any[]) => void): void
    prune(cb: () => void): void
    dispose(cb: (data: any) => void): void
    decline(): void
    invalidate(): void
    on(event: string, cb: (...args: any[]) => void): void
  }
}
```

这里稍微解释一下， `import.meta` 对象为现代浏览器原生的一个内置对象，Vite 所做的事情就是在这个对象上的 `hot` 属性中定义了一套完整的属性和方法。因此，在 Vite 当中，你就可以通过 `import.meta.hot` 来访问关于 HMR 的这些属性和方法，比如 `import.meta.hot.accept()` 。接下来，我们就来一一熟悉这些 API 的使用方式。

模块更新时逻辑: hot.accept

在 `import.meta.hot` 对象上有一个非常关键的方法 `accept` ，因为它决定了 Vite 进行热更新的边界，那么如何来理解这个 `accept` 的含义呢?

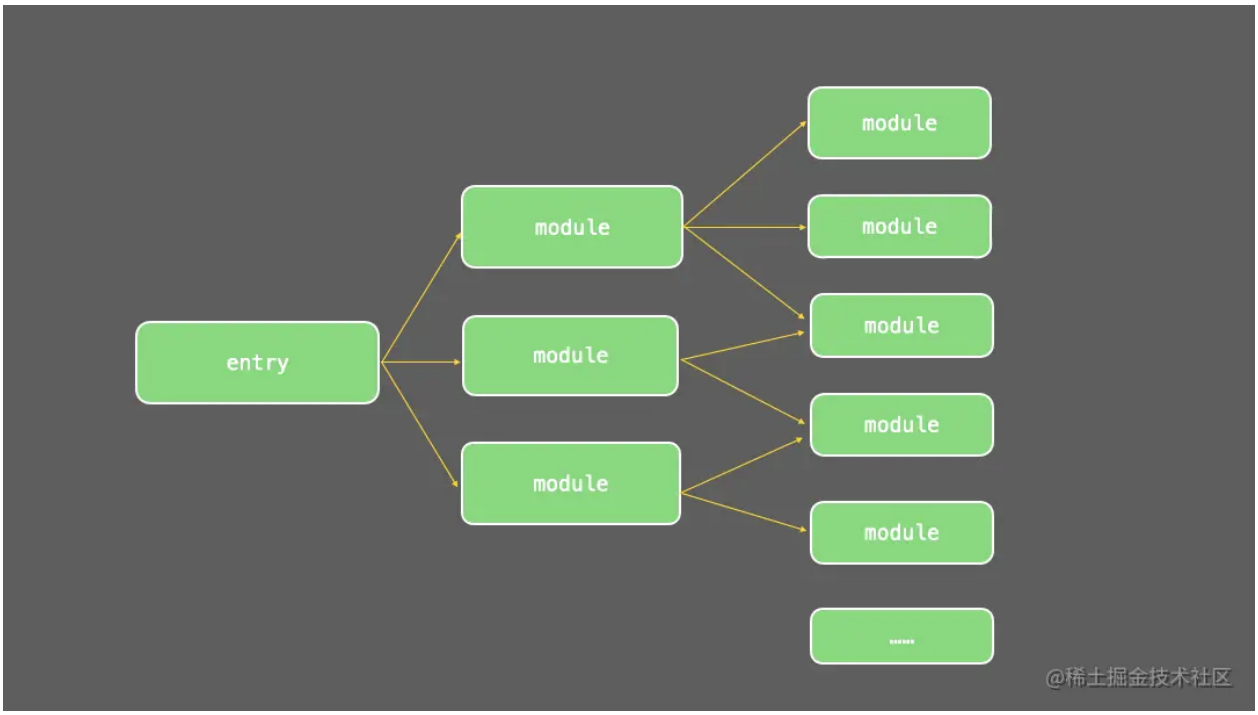
从字面上来看，它表示接受的意思。没错，它就是用来**接受模块更新**的。 一旦 Vite 接受了这个更新，当前模块就会被认为是 HMR 的边界。那么，Vite 接受谁的更新呢？这里会有三种情况：

- 接受**自身模块**的更新
- 接受**某个子模块**的更新
- 接受**多个子模块**的更新

这三种情况分别对应 accept 方法三种不同的使用方式，下面我们就一起来分析一下。

1. 接受自身更新

当模块接受自身的更新时，则当前模块会被认为 HMR 的边界。也就是说，除了当前模块，其他的模块均未受到任何影响。下面是我准备的一张示例图，你可以参考一下：



为了加深你的理解，这里我们以一个实际的例子来操练一下。这个例子已经放到了 Github 仓库 中，你可以把这个链接克隆到本地，然后跟着我一步步添加内容。首先展示一下整体的目录结构:

```
.
├── favicon.svg
├── index.html
├── node_modules
│   └── ...
├── package.json
├── src
│   ├── main.ts
│   ├── render.ts
│   ├── state.ts
│   ├── style.css
│   └── vite-env.d.ts
└── tsconfig.json
```

这里我放出一些关键文件的内容，如下面的 `index.html` ：

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <link rel="icon" type="image/svg+xml" href="favicon.svg" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Vite App</title>
  </head>
  <body>
```

```
  <body>
    <div id="app"></div>
    <p>
      count: <span id="count">0</span>
    </p>
    <script type="module" src="/src/main.ts"></script>
  </body>
</html>
```

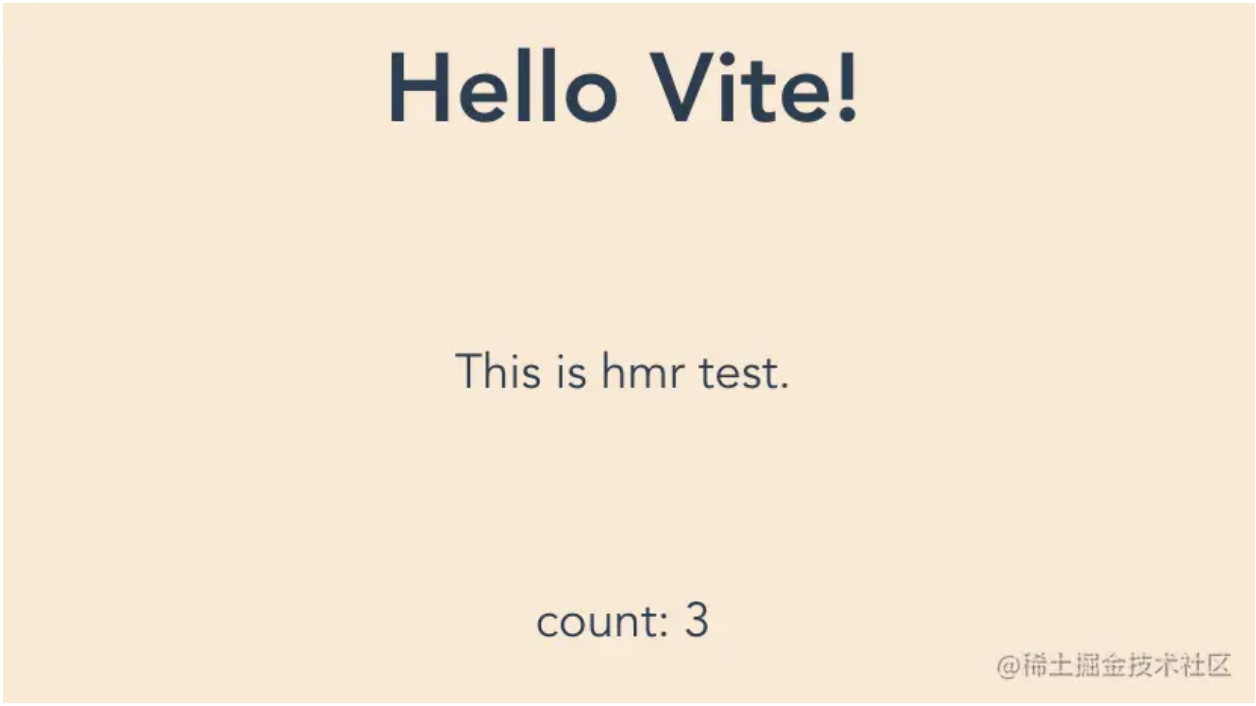
里面的 DOM 结构比较简单，同时引入了 `/src/main.ts` 这个文件，内容如下:

```
import { render } from './render';
import { initState } from './state';
render();
initState();
```

文件依赖了 `render.ts` 和 `state.ts` ，前者负责渲染文本内容，而后者负责记录当前的页面状态:

```
// src/render.ts
// 负责渲染文本内容
import './style.css'
export const render = () => {
  const app = document.querySelector<HTMLDivElement>('#app')!
  app.innerHTML = `
    <h1>Hello Vite!</h1>
    <p target="_blank">This is hmr test.123</p>
  `
}
// src/state.ts
// 负责记录当前的页面状态
export function initState() {
  let count = 0;
  setInterval(() => {
    let countEle = document.getElementById('count');
    countEle!.innerText =  ++count + '';
  }, 1000);
}
```

好了，仓库当中关键的代码就目前这些了。现在，你可以执行 `pnpm i` 安装依赖，然后 `npm run dev` 启动项目，在浏览器访问可以看到这样的内容:



同时，每隔一秒钟，你可以看到这里的 `count` 值会加一。OK，现在你可以试着改动一下 `render.ts` 的渲染内容，比如增加一些文本:

```
// render.ts
export const render = () => {
  const app = document.querySelector<HTMLDivElement>('#app')!
  app.innerHTML = `
    <h1>Hello Vite!</h1>
+   <p target="_blank">This is hmr test.123 这是增加的文本</p>
  `
}
```

效果如下所示:

页面的渲染内容是更新了，但不知道你有没有注意到最下面的 `count` 值瞬间被置零了，并且查看控制台，也有这样的 log：

```
[vite] page reload src/render.ts
```

很明显，当 `render.ts` 模块发生变更时，Vite 发现并没有 HMR 相关的处理，然后直接刷新页面了。

现在让我们在 `render.ts` 中加上如下的代码：

```
// 条件守卫
+ if (import.meta.hot) {
+  import.meta.hot.accept((mod) => mod.render())
+ }
```

`import.meta.hot` 对象只有在开发阶段才会被注入到全局，生产环境是访问不到的，另外增加条件守卫之后，打包时识别到 if 条件不成立，会自动把这部分代码从打包产物中移除，来优化资源体积。因此，我们需要增加这个条件守卫语句。

接下来，可以注意到我们对于 `import.meta.hot.accept` 的使用：

```
import.meta.hot.accept((mod) => mod.render())
```

这里我们传入了一个回调函数作为参数，入参即为 Vite 给我们提供的更新后的模块内容，在浏览器中打印 `mod` 内容如下，正好是 `render` 模块最新的内容:

我们在回调中调用了一下 `mod.render` 方法，也就是当模块变动后，每次都重新渲染一遍内容。这时你可以试着改动一下渲染的内容，然后到浏览器中注意一下 `count` 的情况，并没有被重新置零，而是保留了原有的状态:

没错，现在 `render` 模块更新后，只会重新渲染这个模块的内容，而对于 state 模块的内容并没有影响，并且控制台的 log 也发生了变化:

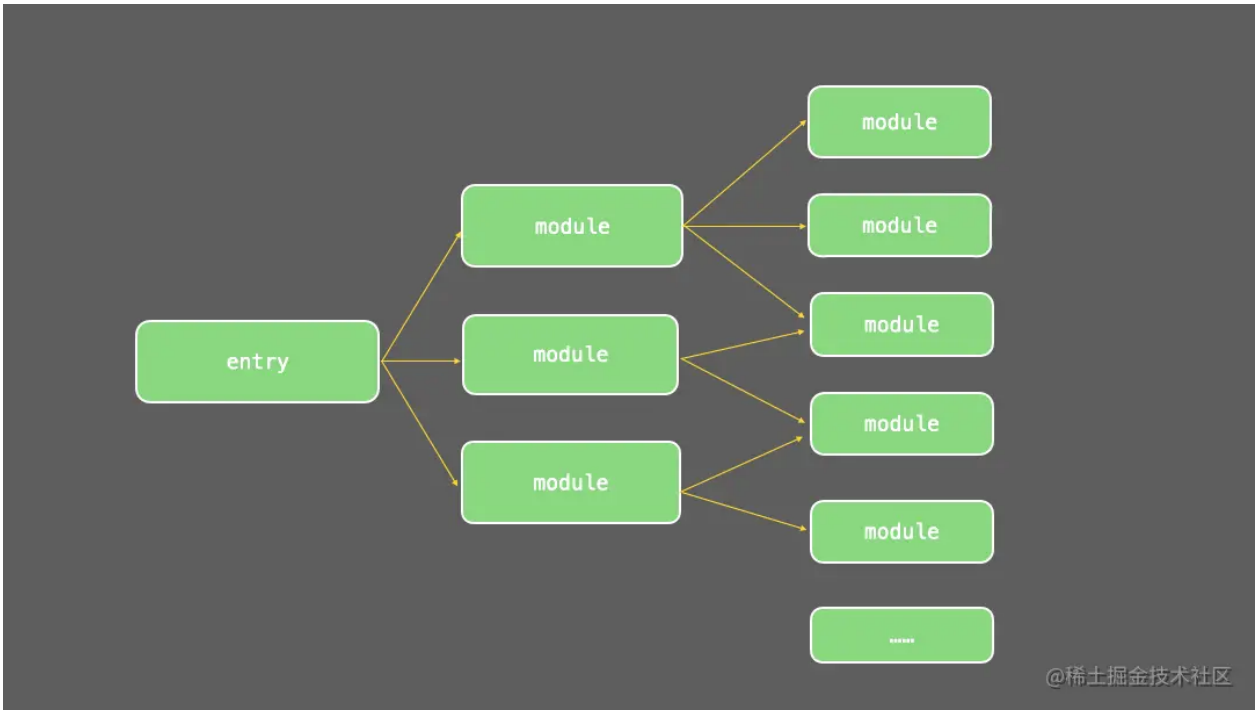
```
[vite] hmr update /src/render.ts
```

现在我们算是实现了初步的 HMR，也在实际的代码中体会到了 accept 方法的用途。当然，在这个例子中我们传入了一个回调函数来手动调用 render 逻辑，但事实上你也可以什么参数都不传，这样 Vite 只会把 `render` 模块的最新内容执行一遍，但 `render` 模块

什么参数都不传，这样 Vite 大会把 render 模块的最新内容执行一遍，但 render 模块内部只声明了一个函数，因此直接调用 import.meta.hot.accept() 并不会重新渲染页面。

2. 接受依赖模块的更新

上面介绍了 接受自身模块更新 的情况，现在来分析一下 接受依赖模块更新 是如何做到的。先给大家放一张原理图，直观地感受一下:



还是拿示例项目来举例， main 模块依赖 render 模块，也就是说， main 模块是 render 父模块，那么我们也可以在 main 模块中接受 render 模块的更新，此时 HMR 边界就是 main 模块了。

我们将 render 模块的 accept 相关代码先删除:

```
// render.ts
- if (import.meta.hot) {
-   import.meta.hot.accept((mod) => mod.render())
- }
```

然后再 main 模块增加如下代码:

```
// main.ts
import { render } from './render';
import './state';
render();
```

```
+if (import.meta.hot) {
+  import.meta.hot.accept('./render.ts', (newModule) => {
+    newModule.render();
+  })
+}
```

在这里我们同样是调用 accept 方法，与之前不同的是，第一个参数传入一个依赖的路径，也就是 `render` 模块的路径，这就相当于告诉 Vite: 我监听了 `render` 模块的更新，当它的内容更新的时候，请把最新的内容传给我。同样的，第二个参数中定义了模块变化后的回调函数，这里拿到了 `render` 模块最新的内容，然后执行其中的渲染逻辑，让页面展示最新的内容。

通过接受一个依赖模块的更新，我们同样又实现了 HMR 功能，你可以试着改动 `render` 模块的内容，可以发现页面内容正常更新，并且状态依然保持着原样。

3. 接受多个子模块的更新

接下来是最后一种 accept 的情况——接受多个子模块的更新。有了上面两种情况的铺垫，这里再来理解第三种情况就容易多了，我依然先给出原理示意图:

这里的意思是**父模块可以接受多个子模块的更新，当其中任何一个子模块更新之后，父模块会成为 HMR 边界。**还是拿之前的例子来演示，现在我们更改 `main` 模块代码:

```
// main.ts
import { render } from './render';
import { initState } from './state';
render();
initState();
+if (import.meta.hot) {
+  import.meta.hot.accept(['./render.ts', './state.ts'], (modules) => {
+    console.log(modules);
+  })
+}
```

在代码中我们通过 accept 方法接受了 `render` 和 `state` 两个模块的更新，接着让我们手动改动一下某一个模块的代码，观察一下回调中 `modules` 的打印内容。例如当我改动 `state` 模块的内容时，回调中拿到的 modules 是这样的:

```
▼(2) [undefined, Module] i                                    main.ts:9
  0: undefined
  ▶1: Module {Symbol(Symbol.toStringTag): 'Module'}
  length: 2
  ▶[[Prototype]]: Array(0)                          @稀土掘金技术社区
```

可以看到 Vite 给我们的回调传来的参数 `modules` 其实是一个数组，和我们第一个参数声明的子模块数组一一对应。因此 `modules` 数组第一个元素是 `undefined`，表示 `render` 模块并没有发生变化，第二个元素为一个 Module 对象，也就是经过变动后 `state` 模块的最新内容。于是在这里，我们根据 `modules` 进行自定义的更新，修改 `main.ts`：

```
// main.ts
import { render } from './render';
import { initState } from './state';
render();
initState();
if (import.meta.hot) {
  import.meta.hot.accept(['./render.ts', './state.ts'], (modules) => {
    // 自定义更新
    const [renderModule, stateModule] = modules;
    if (renderModule) {
      renderModule.render();
    }
    if (stateModule) {
      stateModule.initState();
    }
  })
}
```

现在，你可以改动两个模块的内容，可以发现，页面的相应模块会更新，并且对其它的模块没有影响。但实际上你会发现另外一个问题，当改动了 `state` 模块的内容之后，页面的内容会变得错乱:

这是为什么呢?

我们快速回顾一下 `state` 模块的内容:

```
// state.ts
export function initState() {
  let count = 0;
  setInterval(() => {
    let countEle = document.getElementById('count');
    countEle!.innerText =  ++count + '';
  }, 1000);
}
```

其中设置了一个定时器，但当模块更改之后，这个定时器并没有被销毁，紧接着我们在 accept 方法调用 `initState` 方法又创建了一个新的定时器，导致 count 的值错乱。那如何来解决这个问题呢？这就涉及到新的 HMR 方法—— `dispose` 方法了。

模块销毁时逻辑: hot.dispose

这个方法相较而言就好理解多了，代表在模块更新、旧模块需要销毁时需要做的一些事情，拿刚刚的场景来说，我们可以通过在 `state` 模块中调用 dispose 方法来轻松解决定时器共存的问题，代码改动如下:

```
// state.ts
```

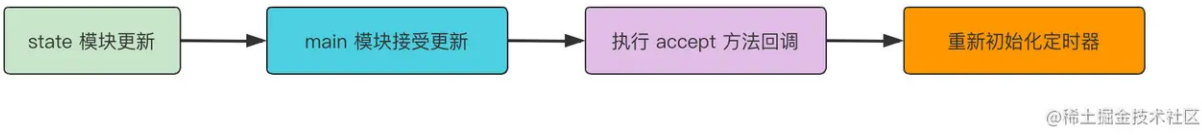
```
let timer: number | undefined;

if (import.meta.hot) {
  import.meta.hot.dispose(() => {
    if (timer) {
      clearInterval(timer);
    }
  })
}
export function initState() {
  let count = 0;
  timer = setInterval(() => {
    let countEle = document.getElementById('count');
    countEle!.innerText =  ++count + '';
  }, 1000);
}
```

此时，我们再来到浏览器观察一下 HMR 的效果:

可以看到，当我稍稍改动一下 `state` 模块的内容(比如加个空格)，页面确实会更新，而且也没有状态错乱的问题，说明我们在模块销毁前清除定时器的操作是生效的。但你又可以很明显地看到一个新的问题: 原来的状态丢失了， `count` 的内容从 `64` 突然变成 `1` 。这又是为什么呢?

让我们来重新梳理一遍热更新的逻辑:



当我们改动了 `state` 模块的代码， `main` 模块接受更新，执行 accept 方法中的回调，接着会执行 `state` 模块的 `initState` 方法。注意了，此时新建的 `initState` 方法的确会

首会执行 `state` 模块的 `initState` 方法。注意了，此时新建的 `initState` 方法的就会

初始化定时器，但同时也会初始化 count 变量，也就是 `count` 从 0 开始计数了！

这显然是不符合预期的，我们期望的是每次改动 `state` 模块，之前的状态都保存下来。怎么来实现呢？

共享数据: hot.data 属性

这就不得不提到 hot 对象上的 data 属性了，这个属性用来在不同的模块实例间共享一些数据。使用上也非常简单，让我们来重构一下 `state` 模块:

```
let timer: number | undefined;
if (import.meta.hot) {
+  // 初始化 count
+  if (!import.meta.hot.data.count) {
+    import.meta.hot.data.count = 0;
+  }
  import.meta.hot.dispose(() => {
    if (timer) {
      clearInterval(timer);
    }
  })
}
export function initState() {
+  const getAndIncCount = () => {
+    const data = import.meta.hot?.data || {
+      count: 0
+    };
+    data.count = data.count + 1;
+    return data.count;
+  };
  timer = setInterval(() => {
    let countEle = document.getElementById('count');
+    countEle!.innerText =  getAndIncCount() + '';
  }, 1000);
}
```

我们在 `import.meta.hot.data` 对象上挂载了一个 `count` 属性，在二次执行 `initState` 的时候便会复用 `import.meta.hot.data` 上记录的 count 值，从而实现状态的保存。

此时，我们终于大功告成，基本实现了这个示例应用的 HMR 的功能。在这个过程中，我们用到了核心的 `accept` 、 `dispose` 和 `data` 属性和方法。当然还有一些方法将会给大家进行介绍，但相较而言就比较简单了，而且用的也不多，大家只需要留下初步的印象，知道这些方法的用途是什么，需要用到的时候再来查阅即可。

其它方法

1. import.meta.hot.decline()

这个方法调用之后，相当于表示此模块不可热更新，当模块更新时会强制进行页面刷新。感兴趣的同学可以继续拿上面的例子来尝试一下。

2. import.meta.hot.invalidate()

这个方法就更简单了，只是用来强制刷新页面。

3. 自定义事件

你还可以通过 `import.meta.hot.on` 来监听 HMR 的自定义事件，内部有这么几个事件会自动触发:

- `vite:beforeUpdate` 当模块更新时触发；
- `vite:beforeFullReload` 当即将重新刷新页面时触发；
- `vite:beforePrune` 当不再需要的模块即将被剔除时触发；
- `vite:error` 当发生错误时（例如，语法错误）触发。

如果你想自定义事件可以通过上节中提到的 `handleHotUpdate` 这个插件 Hook 来进行触发:

```
// 插件 Hook
handleHotUpdate({ server }) {
  server.ws.send({
    type: 'custom',
    event: 'custom-update',
    data: {}
  })
  return []
}
// 前端代码
import.meta.hot.on('custom-update', (data) => {
  // 自定义更新逻辑
})
```

小结

本篇的正文内容到这里就接近尾声了，在这一节中，你需要重点掌握 **HMR 的概念**、**Vite HMR API 的使用**以及**HMR 的更新原理**。

我们首先认识了 HMR 这个概念，了解它相比于传统的 live reload 所解决的问题：模块局部更新 和 状态保存 。然后我带你熟悉了 Vite HMR 中的各种 API，尤其是 accept 方法，根据 accept 的不同用法，我们分了三种情况来讨论 Vite 接受更新的策略：接受自身更新 、 接受依赖模块的更新 和 接受多个子模块的更新 ，并通过具体的示例来进行这三种情况的代码演示，可以看到在代码发生变动的时候，Vite 会定位到发生变化的局部模块，也就是找到对应的 HMR 边界，然后基于这个边界进行更新，其他的模块并没有受到影响，这也是 Vite 中的热更新的时间也到达毫秒级别的重要原因。

在 Vite 中，HMR 是一套比较复杂的系统，不过一旦理解了本文提到的 HMR 边界 的作用原理，那么在后面解读 Vite HMR 源码的时候将会倍感轻松。大家加油吧！

上一篇： 插件开发与实战: 如何开发一个完整的 Vite 插件?

下一篇：代码分割：打包完产物体积太大，怎么拆包?