

4-5 直播 CI/CD、Vue 3

模块直播答疑

CI/CD

Github Actions

创建 Access Token

为项目配置 Access Token

设置 Action

Vue 3

通过 Vite 创建项目

安装 Vue 项目

Devtools Extension

不同构建版本

功能

createApp

data

setup

reactive

toRefs

ref

生命周期

结语

内容介绍：

- 模块直播答疑
- CI/CD
- Vue 3

直播代码：📎 [4-5 直播代码汇总.rar](#)

模块直播答疑

1. Vue Router 中，\$router 与 \$route 的区别。

a. \$router 用于对路由进行导航操作，如跳转等功能。

b. \$route 用于访问路由的信息，如路径等。

c. \$router 与 \$route —“动”—“静”，功能性不同，可在操作中多加体会。

CI/CD

CI/CD 指的是持续集成与持续部署。

当我们在项目开发中，往往不可能一次就实现所有的功能，为了尽早上线，我们会先实现基础版本，再不断更新、迭代。而在这个过程中就会存在很多的重复的步骤，例如，每次代码更新后，代码会被推送到 github，后续还需要进行测试、构建、上传服务器、安装依赖，压缩等一系列操作，而测试与部署还存在不同平台的问题。这些操作复杂度不高，但重复量大，十分繁琐。

CI/CD 的概念是对某个仓库的代码进行监控，当收到更新后，自动根据对其进行指定的操作，如测试、构建、部署到服务器等等，这样可以大大减少繁杂的操作，也避免出现因为疏忽导致的问题（忘记测试就发布）。

许多大型的代码托管平台都提供了这样的工具，下面我们以体验更好且使用更多的 Github 为例进行讲解。

Github Actions

官方文档：<https://docs.github.com/cn/actions>

直播示例仓库：<https://github.com/wuyouu/cicd-demo>

创建 Access Token

要使用 Github Actions 进行 CI/CD 服务，需要先配置 Github Access Token

WilliamWu
Your personal account

Account settings

- Profile
- Account
- Appearance New
- Account security
- Billing & plans
- Security log
- Security & analysis
- Emails
- Notifications
- SSH and GPG keys
- Repositories
- Packages
- Organizations
- Saved replies
- Applications
- Developer settings

GitHub Apps

OAuth Apps

Personal access tokens

Personal access tokens

Generate new token

Revoke all

Tokens you have generated that can be used to access the [GitHub API](#).

git: <https://github.com/> on DESKTOP-QHUN6P2 at 25-2月-2020 18:19 — gist, Last used within the last 8 months

Delete

Note

my-test-token

What's this token for?

Select scopes

Scopes define the access for personal tokens. [Read more about OAuth scopes](#).

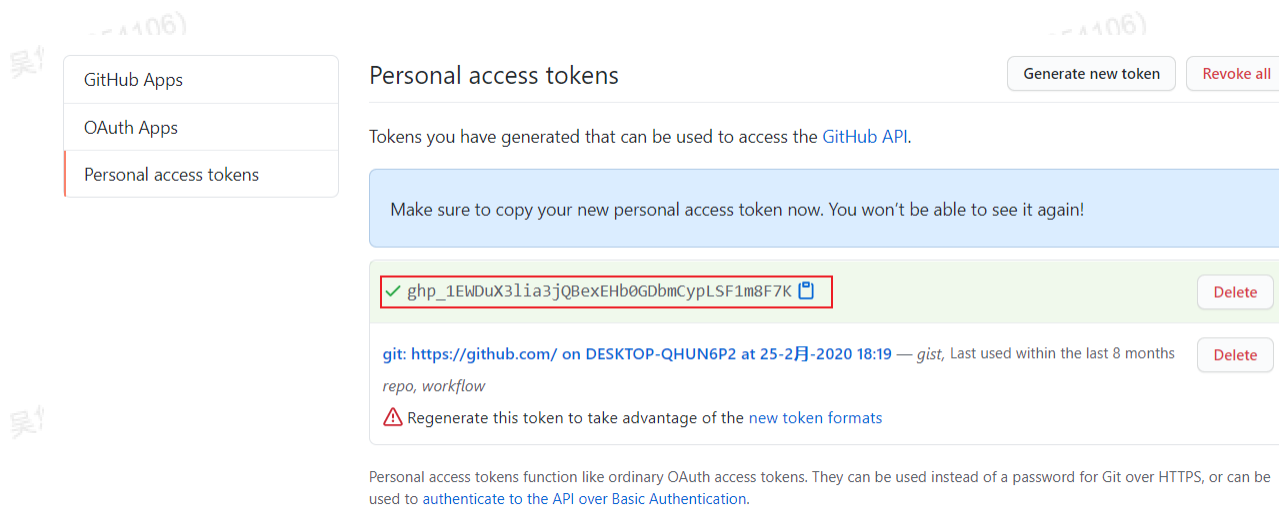
<input checked="" type="checkbox"/> repo	Full control of private repositories
<input checked="" type="checkbox"/> repo:status	Access commit status
<input checked="" type="checkbox"/> repo_deployment	Access deployment status
<input checked="" type="checkbox"/> public_repo	Access public repositories
<input checked="" type="checkbox"/> repo:invite	Access repository invitations
<input checked="" type="checkbox"/> security_events	Read and write security events

设置名称，并勾选对仓库的读写权限即可，其他权限可根据实际需求设置。

Generate token Cancel

创建完成，复制 token。

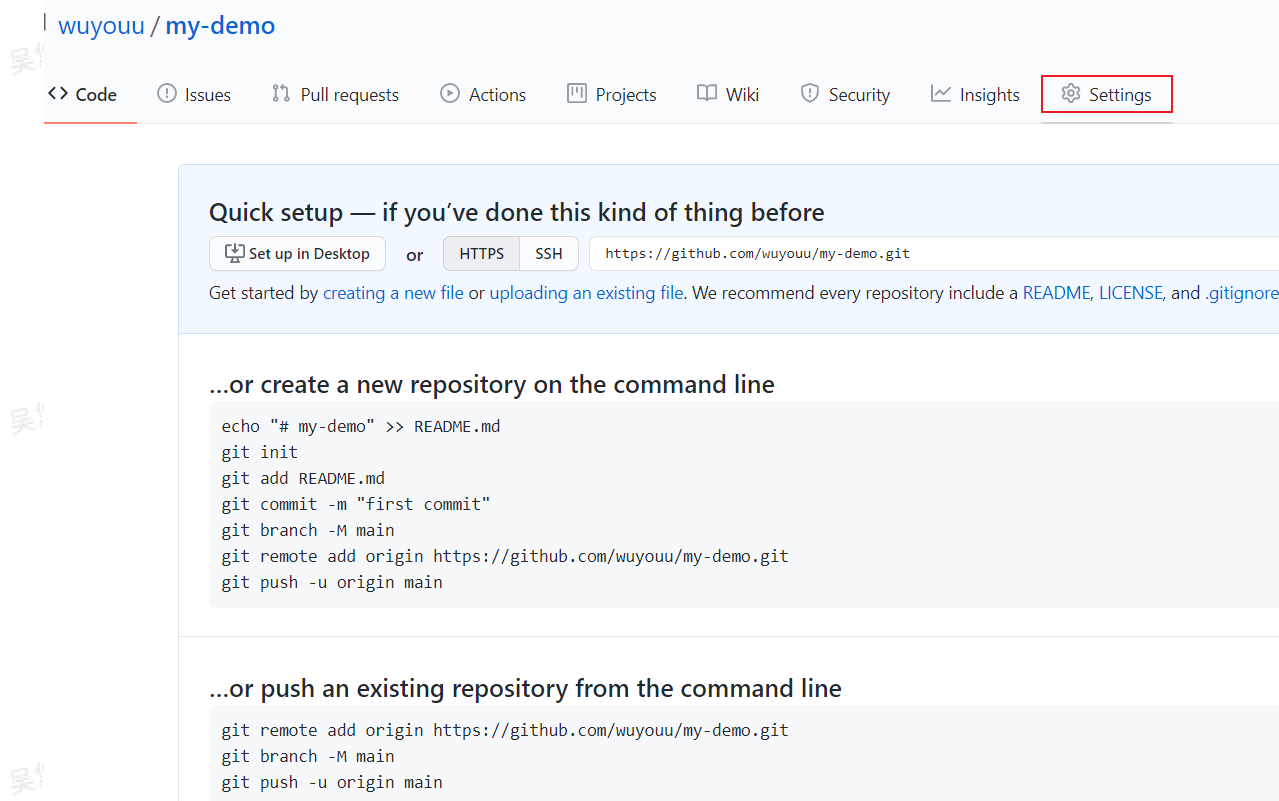
- 注意，token 只显示一次，创建后复制并保存起来，如果找不到了，再创建新的即可。



The screenshot shows the GitHub 'Personal access tokens' interface. On the left, a sidebar lists 'GitHub Apps', 'OAuth Apps', and 'Personal access tokens' (which is selected). The main area is titled 'Personal access tokens' and includes buttons for 'Generate new token' and 'Revoke all'. Below this, a message states: 'Tokens you have generated that can be used to access the GitHub API.' A blue box contains the instruction: 'Make sure to copy your new personal access token now. You won't be able to see it again!'. A green box displays a newly generated token: 'ghp_1EWDuX31ia3jQBexEHb0GDbmCypLSF1m8F7K', with a 'Delete' button next to it. Below the token, it shows the token's scope: 'git: https://github.com/ on DESKTOP-QHUN6P2 at 25-2月-2020 18:19 — gist, Last used within the last 8 months' and a 'Delete' button. A warning icon and text advise: 'Regenerate this token to take advantage of the new token formats'. At the bottom, a note explains that personal access tokens function like ordinary OAuth access tokens and can be used instead of a password for Git over HTTPS or to authenticate to the API over Basic Authentication.

为项目配置 Access Token

进入某个仓库的设置，此处演示为空仓库。

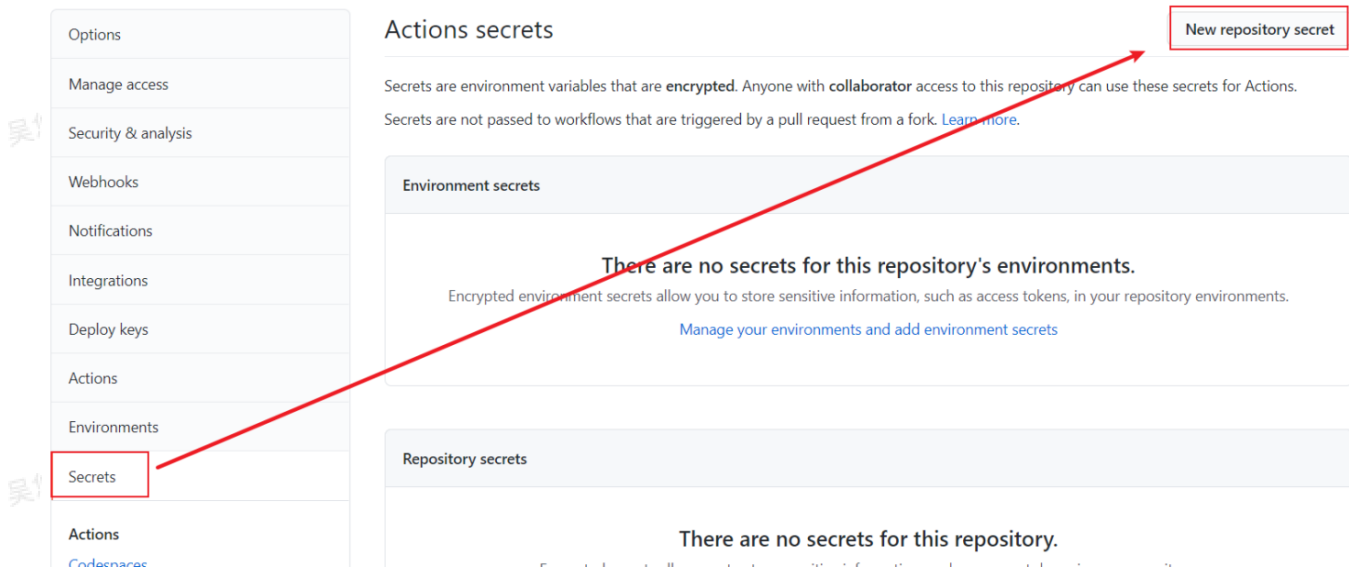


The screenshot shows the GitHub repository settings page for 'wuyouu / my-demo'. The top navigation bar includes links for 'Code', 'Issues', 'Pull requests', 'Actions', 'Projects', 'Wiki', 'Security', 'Insights', and 'Settings' (which is highlighted with a red box). Below the navigation bar, the 'Quick setup' section offers options to 'Set up in Desktop', 'HTTPS', or 'SSH' with the repository URL 'https://github.com/wuyouu/my-demo.git'. A message states: 'Get started by creating a new file or uploading an existing file. We recommend every repository include a README, LICENSE, and .gitignore.' The next section, '...or create a new repository on the command line', provides a series of terminal commands:

```
echo "# my-demo" >> README.md
git init
git add README.md
git commit -m "first commit"
git branch -M main
git remote add origin https://github.com/wuyouu/my-demo.git
git push -u origin main
```

 The final section, '...or push an existing repository from the command line', provides another set of terminal commands:

```
git remote add origin https://github.com/wuyouu/my-demo.git
git branch -M main
git push -u origin main
```



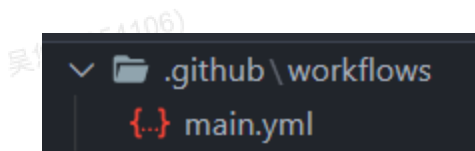
注意名称的拼写格式，参考示例格式，且记住此名称，后续会使用。

Actions secrets / New secret

设置 Action

在项目中创建 `.github/workflows` 不要拼错，将 `main.yml` 放入，这个文件中包含了 Actions 的处理规则。

文件：📎 `main.yml`



注意：下面代码中的 `MY_TEST_TOKEN` 部分需要根据实际设置的 Secrets 名称进行设置。

```
1 name: my demo
```

```

2 # 执行时机
3 on:
4   # push 时执行
5   push
6 # 要执行的工作
7 jobs:
8   build:
9     # Github 托管的运行环境: 可为 linux、window、macOS 的不同版本
10    runs-on: ubuntu-latest
11    # 策略
12    strategy:
13      # 矩阵: 可以配置多种规则
14      matrix:
15        # 指定 node 版本 (会根据不同环境执行多次工作)
16        node-version: [12, 14, 15]
17    # 执行步骤
18    steps:
19      # 下载源码
20      - uses: actions/checkout@v2
21
22      # 构建
23      - uses: actions/setup-node@v2
24        with:
25          node-version: ${ matrix.node-version }
26      - run: npm install
27      - run: npm run lint
28      - run: npm run build
29      - run: tar -zcvf release.tgz dist
30
31      # 发布 Release
32      - name: Create Release
33        id: create-release
34        uses: actions/create-release@master
35        env:
36          GITHUB_TOKEN: ${ secrets.MY_TEST_TOKEN }
37        with:
38          # github.ref 为触发工作流的分支或标记的地址
39          tag_name: ${ github.ref }
40          release_name: Release ${ github.ref }
41          draft: false

```

```

42         prerelease: false
43
44     # 上传构建结果
45     - name: Upload Release
46       id: upload-release-asset
47       uses: actions/upload-release-asset@master
48       env:
49         GITHUB_TOKEN: ${ secrets.MY_TEST_TOKEN }
50       with:
51         upload_url: ${ steps.create-release.outputs.upload_url
52       }}
53         asset_path: ./release.tgz
54         asset_name: release.tgz
55         asset_content_type: application/x-tgz

```

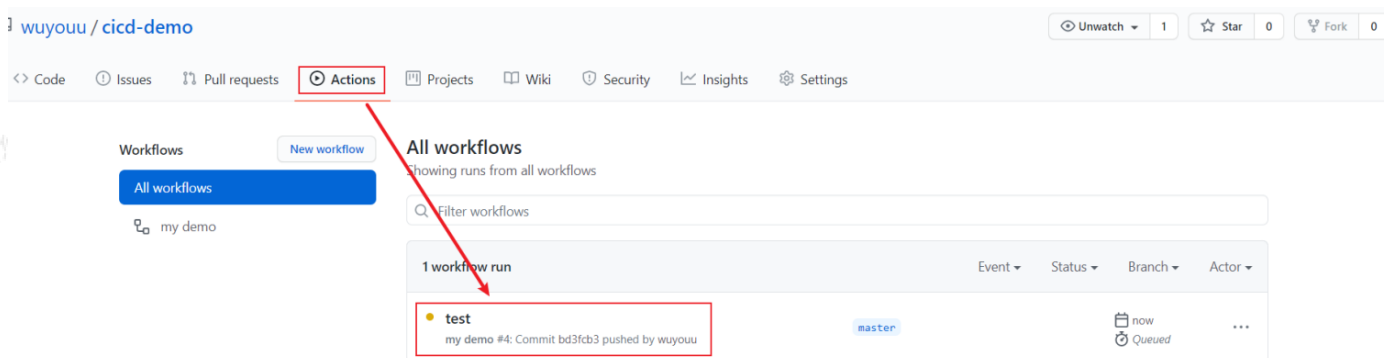
给项目添加内容，如教育后台项目（注意不含 node_modules），并推送到仓库。

```

1 git add .
2 git commit -m "描述"
3 git remote add origin https://github.com/xxxxxxxxxx # 仓库地址自行填写
4 git push -u origin master

```

在 github 上打开仓库，查看 Actions。



左侧会显示执行的工作，12、14、15 为配置的 3 个 node 版本。

● test my demo #4

Summary

Jobs

● build (12)

● build (14)

● build (15)

build (12)

Started 20s ago

- > ☒ Set up job
- > ☒ Run actions/checkout@v2
- > ☒ Run actions/setup-node@v2
- Run npm install
- ☐ Run npm run lint
- ☐ Run npm run build
- ☐ Run tar -zcvf release.tgz dist
- ☐ Create Release
- ☐ Upload Release
- ☐ Post Run actions/checkout@v2

右侧为工作中的步骤执行情况，可以看到具体信息。

Summary

Jobs

● build (12)

● build (14)

● build (15)

build (12)

Started 1m 24s ago

- > ☒ Set up job
- > ☒ Run actions/checkout@v2
- > ☒ Run actions/setup-node@v2
- > ☒ Run npm install
- > ☒ Run npm run lint
- > ☒ Run npm run build
- > ☒ Run tar -zcvf release.tgz dist

● Create Release

```
1 ▼ Run actions/create-release@master
2   with:
3     tag_name: refs/heads/master
4     release_name: Release refs/heads/master
5     draft: false
6     prerelease: false
7   env:
8     GITHUB_TOKEN: ***
```

- ☐ Upload Release
- ☐ Post Run actions/checkout@v2

左侧会显示工作的执行进度，绿色图标为成功，黄色为执行中，红色为失败。

Summary

Jobs

- build (12)
- build (14)
- build (15)

build (12)
succeeded now in 1m 38s

- > Set up job
- > Run actions/checkout@v2
- > Run actions/setup-node@v2
- > Run npm install
- > Run npm run lint
- > Run npm run build
- > Run tar -zcvf release.tgz dist
- > Create Release
- > Upload Release
- > Post Run actions/checkout@v2
 - 1 Post job cleanup.
 - 2 /usr/bin/git version
 - 3 git version 2.31.1
 - 4 /usr/bin/git config --local --name-only
 - 5 /usr/bin/git submodule foreach --recurse-submodules 'core.sshCommand' || :
 - 6 /usr/bin/git config --local --name-only
 - 7 http.https://github.com/.extraheader
 - 8 /usr/bin/git config --local --unset-all
 - 9 /usr/bin/git submodule foreach --recurse-submodules --unset-all 'http.https://github.com/
- > Complete job

失败时可以从右侧看到失败的步骤以及原因，后期我们可以根据情况进行代码修改。

Summary

Jobs

- build (12)
- build (14)
- build (15)

build (15)
failed 1 minute ago in 1m 21s

- > Set up job
- > Run actions/checkout@v2
- > Run actions/setup-node@v2
- > Run npm install
- > Run npm run lint
- > Run npm run build
- > Run tar -zcvf release.tgz dist
- > Create Release
 - 1 ▶ Run actions/create-release@master
 - 9 Error: Validation Failed: {"resource":"Release","code":"already_exists","field":"tag_name"}
- > Upload Release
- > Post Run actions/checkout@v2
- > Complete job

Vue 3

文档: <https://www.vue3js.cn/docs/zh/>

突出的新特性:

- 源码组织方式改变
 - 源码采用 TS 编写, 具有更可靠的类型系统。
- 性能提升
 - 通过 Proxy 重写了响应式系统, 由于 Proxy 在浏览器的性能更好, 使得 Vue 3 响应式系统的效率更高。
 - 不仅提升了性能, 在功能方面也有提升, 具体见 4-5 Proxy 的部分。
 - 不仅可以监听属性的获取与设置, 还可监听添加与删除操作
 - 直接监听对象而非属性。
 - 可监听数组的索引与 length 操作。
 - 虚拟 DOM 更新, 提升了首次渲染与更新的性能。
 - Vue 2 中会在数据变化时通过 diff 对比节点的区别, 并进行有变化的部分进行更新。
 - Vue 3 进一步优化, 在每次编译后对静态节点进行“提升”, 提升的节点由于不会变化, 所以 diff 时无法比较; 动态节点在每次编译后会动态的部分进行标记, 如内容为动态或属性为动态等, 在 diff 时检测对应的部分即可, 这样可以进一步减少比较的节点数量, 提升性能。
 - 动态静态取决于内部是否存在绑定的数据。
- Composition API 组合式
 - 使用 Options API 处理复杂业务时, 不同的功能代码在书写上会交叉在一起, 不利于复用与维护。
- Vite
 - Vue CLI 开发模式下需要对项目进行打包才能预览, 而通过 Vite 在开发模式下可以不打包直接执行, 启动速度非常快。

Vue 3 中还处于新生阶段, 我们可关注文档以学习新的功能。

通过 Vite 创建项目

Vite 官网: <https://cn.vitejs.dev/>

安装 Vue 项目

Vue 3 文档中的步骤: 通过 Vite 安装的的项目默认为 Vue 项目

```
1 npm init vite-app <project-name>
2 cd <project-name>
3 npm install
4 npm run dev
```

Vite 2 更新后改变了这一特点，不存在默认的类型，而需要手动选择。（将 Vite 打造成框架无关的工具）

```
1 # npm 6.x
2 npm init @vitejs/app my-vue-app --template vue
3
4 # npm 7+, 需要额外的双横线:
5 npm init @vitejs/app my-vue-app -- --template vue
6
7 # yarn
8 yarn create @vitejs/app my-vue-app --template vue
```

也可以直接书写以下命令并跟随引导操作：

```
1 npm init @vitejs/app
```

创建完毕，可以查看一下通过 Vite 创建的 Vue 项目结构。

- index.html 中通过 ES Module 的方式引入了入口文件 `main.js`。
 - 组件内部引入了 App.vue，观察后可以发现，组件内的写法与 Vue CLI 创建的项目相同。
- main.js 中的 Vue 操作与 Vue CLI 创建的 Vue 项目在书写上有所不同，后面会讲解具体用法。

接下来 cd 进入项目目录，通过 npm run dev 来感受一下 Vite 飞快的启动速度吧！

Devtools Extension

crx: [extension_6_0_0_7.rar](#)

创建项目后可以发现，浏览器中的 devtools 并没有正常显示。官方正在开发新版本的 devtools 以支持多个版本的 Vue，目前处于测试中，暂只支持 Vue3，官方下载方式见[文档](#)。

不同构建版本

[文档](#)

- .cjs 指的是 commonjs 的版本
- .esm 指的是 esmodule 的版本
- .runtime 指的是只包含运行时的版本，不含编译器
- .global 指的是全局版本，安装后添加一个全局变量
- .bundler 指的是需要配合打包工具一起使用的版本

如果希望在浏览器中书写 demo 练习功能，可以引入 `vue.esm-browser.js`。

功能

createApp

create App 是 Vue 3 提供的，且不兼容 Vue 2 的新 API。

createApp 是从 'vue' 中 import 得到，说明 createApp 是 Vue 对象的属性，相当于 `Vue.createApp`。

通过观察可以发现，createApp 函数其实是充当了 Vue 2 中 Vue 类的功能；如字面含义所示，createApp 用于创建 App（实例），并对实例进行操作（如挂载）。

- 参数：组件的选项对象，用于设置 data、methods 等等。
- 返回值：应用实例

```
1 // Vite 创建方式中的 App.vue
2 import { createApp } from 'vue'
3 import App from './App.vue'
4
5 createApp(App).mount('#app')
6
7 /*
8     // 等价于上面代码
9     const app = createApp(App)
10    app.mount('#app')
11 */
```

这种操作变化，使得多个实例间的界限更加分明。当我们在书写时语义也更清晰。

- 例如：将 `Vue.use(xxx)` 更改为 `app1.use(xxx)` `app2.use(xxx)` 等等

我们可以在代码中书写以下示例：

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <meta http-equiv="X-UA-Compatible" content="IE=edge">
6   <meta name="viewport" content="width=device-width, initial-scale=1.0">
7   <title>Document</title>
8 </head>
9 <body>
10  <div id="app"></div>
11  <script type="module">
12    import { createApp } from './node_modules/vue/dist/vue.esm-browser.js'
```

```

13     const app = createApp({})
14     app.mount('#app')
15   </script>
16 </body>
17 </html>

```

data

传递给 createApp 的配置对象中的 data 不能为对象，必须为函数形式。

```

1 ...
2 <body>
3   <div id="app">
4     <button @click="handleInc">+</button>
5     <span style="margin: 0 10px;">{{ count }}</span>
6     <button @click="handleDec">-</button>
7   </div>
8   <script type="module">
9     import { createApp } from './node_modules/vue/dist/vue.esm-br
    owser.js'
10    const app = createApp({
11      data () {
12        return {
13          count: 0
14        }
15      },
16      methods: {
17        handleInc () {
18          this.count++
19        },
20        handleDec () {
21          this.count--
22        }
23      }
24    })
25    app.mount('#app')
26  </script>
27 </body>
28 </html>

```

setup

Composition API 称为组合式 API，是 Vue 3 中提供的新功能，用于替代 Vue 2 中使用的 Options API，但 Vue 3 中同样保留了对 Options API 的支持。

通过组合式 API，可以让 Vue 在构建大型应用时的代码灵活度更高，可维护性更好。

要设置组合式 API，需要使用一个新的选项 setup。

- setup 会在组件接收 props 后、组件创建前执行，内部无法通过 this 获取组件实例，也就无法访问 data、methods 等属性。
- setup 接收参数 1 为 props，既父组件传入的数据。
- setup 可以返回一个对象，当组件创建完毕后，此对象可以在模板、methods、mounted 等功能中被访问。
 - 这个对象的属性会被合并给 this，后续可通过 this 访问。
 - 注意，这个对象默认不是响应式的，通过点击增减按钮可以进行测试。
 - 测试后，this.count 会变化，但视图不会更新。

```
1 ...
2 <body>
3   <div id="app">
4     <p>这是内容</p>
5     <button @click="handleInc">+</button>
6     <span style="margin: 0 10px;">{{ count }}</span>
7     <button @click="handleDec">-</button>
8   </div>
9   <script type="module">
10     import { createApp } from './node_modules/vue/dist/vue.esm-br
11     owser.js'
12     const app = createApp({
13       setup () {
14         let count = 0
15         // 设置完毕，点击增减按钮测试发现视图不会更新
16         return { count }
17       },
18       methods: {
19         handleInc () {
20           this.count++
21         },
22         handleDec () {
23           this.count--
```

```

24     }
25   })
26   app.mount('#app')
27 </script>
28 </body>
29 </html>

```

reactive

reactive 用于将对象转换为响应式数据，返回一个 Proxy 对象。功能相当于 Vue 2 中的 Vue.observable()。

在示例中引入并通过 reactive() 包裹 setup 返回的对象即可。

```

1 ...
2 <script type="module">
3   // 引入 reactive 函数
4   import { createApp, reactive } from './node_modules/vue/dist/
vue.esm-browser.js'
5   const app = createApp({
6     setup () {
7       let count = 0
8       // 设置完毕，点击增减按钮测试
9       return reactive({ count })
10    },
11 ...

```

可以发现，handleInc、与 handleDec 这些与 count 相关的功能封装起来，以增强关联性。

```

1 ...
2 function useCount () {
3   let count = reactive(0)
4   return {
5     count,
6     handleInc () {
7       count++
8     },
9     handleDec () {
10      count--


```

```

11     }
12   }
13 }
14 const app = createApp({
15   setup () {
16     // 调用 useCount() 引入 count 相关功能
17     const { count, handleInc, handleDec } = useCount()
18     // 通过 return 方式将功能设置给组件
19     return {
20       count,
21       handleInc,
22       handleDec
23     }
24   }
25 })
26 app.mount('#app')

```

由于 reactive 只能将对象转换为响应式，而 0 为原始类型，所以此时控制台出现如下提示。

 value cannot be made reactive: 0

这时，我们可以为 count 包裹一层对象例如 data，以通过 reactive() 进行响应式处理。

```

1 ...
2 function useCount () {
3   let data = reactive({
4     count: 0
5   })
6   return {
7     data,
8     handleInc () {
9       data.count++
10    },
11    handleDec () {
12      data.count--
13    }
14  }
15 }

```


接下来在 setup 中调用并引入功能，为了避免在模板中写成 data.count，所以通过解构方式取得 count 并返回。

```
1 const app = createApp({
2   setup () {
3     // 调用 useCount() 引入 count 相关功能
4     const { data: { count }, handleInc, handleDec } = useCount()
5     // 通过 return 方式将功能设置给组件
6     return {
7       count,
8       handleInc,
9       handleDec
10    }
11  }
12 })
13 app.mount('#app')
```

此时会发现 count 操作后无法更新视图，仿佛不再是响应式数据了，这是因为代码中的解构操作实际是声明了新变量存储了数据，然后将变量的值作为了 setup 的返回值使用，而这个值与之前设置的响应式数据无关。

toRefs

toRefs 是 Vue 提供的响应式数据函数，用于将 reactive() 返回的响应式对象的属性也设置为响应式数据。

```
1 import { createApp, reactive, toRefs } from './node_modules/vue/dist/vue.esm-browser.js'
2 let data = reactive({
3   count: 0
4 })
5 return {
6   // 将 data 的属性也设置为响应式数据
7   data: toRefs(data),
8   handleInc () {
9     data.count++
10  },
11  handleDec () {
12     data.count--
13  }
```

reactive 与 toRefs 的组合常用于多个数据设置为响应式的场景。

ref

除了上述功能外，Vue 3 中还提供了 ref 函数，用于对单个原始类型数据进行响应式设置。

实现原理为，在原始类型外包裹一层对象，对象存在属性 value 用于存储原始数据值，通过给属性设置 getter、setter 来实现响应式数据。

注意，在后续操作中，需要通过 value 属性进行数据操作，但模板中无需书写 value。

- toRefs 对对象原始类型属性的实现原理与 ref 相同，如果调用 toRefs 后需要操作属性，同样要使用 value。

```

1 ...
2 <!-- 模板中依然为 count，无需书写 value -->
3 <span style="margin: 0 10px;">{{ count }}</span>
4 ...
5 // 引入 ref
6 import { createApp, reactive, ref } from './node_modules/vue/dist/vue.esm-browser.js'
7 function useCount () {
8   // 通过 ref 将 count 变为响应式数据
9   let count = ref(0)
10  console.log(count)
11  return {
12    // 后续可直接操作
13    count,
14    handleInc () {
15      // 使用时通过 value 操作
16      count.value++
17    },
18    handleDec () {
19      count.value--
20    }
21  }
22 }
23 const app = createApp({
24   setup () {
25     // 调用 useCount() 引入 count 相关功能
26     // const { data: { count }, handleInc, handleDec } = useCount

```

```

    ()
27     const { count, handleDec, handleInc } = useCount()
28
29     // 通过 return 方式将功能设置给组件
30     return {
31         count,
32         handleInc,
33         handleDec
34     }
35 }
36 }
37 app.mount('#app')

```

生命周期

[文档](#)

选项 API 生命周期选项和组合式 API 之间的映射

- `beforeCreate` → USE `setup()`
- `created` → USE `setup()`
- `beforeMount` → `onBeforeMount`
- `mounted` → `onMounted`
- `beforeUpdate` → `onBeforeUpdate`
- `updated` → `onUpdated`
- `beforeUnmount` → `onBeforeUnmount`
- `unmounted` → `onUnmounted`
- `errorCaptured` → `onErrorCaptured`
- `renderTracked` → `onRenderTracked`
- `renderTriggered` → `onRenderTriggered`

`created` 与 `beforeCreated` 与 `setup` 是同期执行的（`setup` 略早与 `beforeCreated`），所以无需使用这两个钩子，Vue 也没有提供。

如果希望使用其他的钩子，需要先引入，再到 setup 中调用。

- 钩子可以重复调用多次，这样可以在多个组合功能中使用。

```
1 <div id="app">
2   <p>内容</p>
3 </div>
4 <script type="module">
5   import { createApp, onMounted } from './node_modules/vue/dist/v
  ue.esm-browser.js'
6 createApp({
7   setup () {
8     onMounted(() => {
9       console.log('mounted1')
10    })
11    onMounted(() => {
12      console.log('mounted2')
13    })
14  }
15 }).mount('#app')
16
17 </script>
```

结语

Vue.js 阶段告一段落，但技术的发展是永不停歇的，这一点从本次直播的内容就可略见一斑。望诸君不忘学习的初心，敢于拼搏，愈战愈勇，早日登上事业巅峰！

Good Hunting!