15. 性能优化

缓存数据

useState - 执行一次

1. 优化原理：useState 参数为**普通变量**时，组件的**每次渲染**都会执行该 Hook；参数为**函数**时，组件的**首次渲染**才会执行该 Hook。因此，**当状态数据结构复杂、计算成本高时，可以使用函数作为 useState 的参数**，从而实现性能优化。
2. 代码演示
   a. useState 参数为普通变量

```
import { FC, useState } from 'react';

const genInitArr = () => {
  console.log('genInitArr 执行了~');
  return ['happy', 'sad', 'angry'];
};

const App: FC = () => {
  /* 每次 arr 变化时，genInitArr 函数都会被执行，尽管该函数仅仅用于初始渲染
      ==> 这说明 useState 参数为普通变量时，每次组件渲染都会执行 */
  const [arr, setArr] = useState(genInitArr());

  return (
    <>
      <p>length of arr: {arr.length}</p>
      <p>
        <button onClick={() => setArr([...arr,
'delighted'])}>add</button>
      </p>
    </>
  );
};

export default App;
```

   b. useState 参数为函数

```tsx
import { FC, useState } from 'react';

const genInitArr = () => {
  console.log('genInitArr 执行了~');
  return ['happy', 'sad', 'angry'];
};

const App: FC = () => {
  /* 只有组件首次渲染时，genInitArr 函数才会被执行
      ==> 这说明 useState 参数为函数时，只有在组件初次渲染时才会被执行 */
  const [arr, setArr] = useState(genInitArr);

  return (
    <>
      <p>length of arr: {arr.length}</p>
      <p>
        <button onClick={() => setArr([...arr, 'delighted'])}>add</button>
      </p>
    </>
  );
};

export default App;
```

useMemo - 缓存数据

1. 优化原理：useMemo 用于**缓存数据**，该 Hook 接收两个参数 calculateValue、dependencies，前者用于计算数据，后者表示依赖项数组。该 Hook 返回值为最新的缓存数据。该 Hook 适用于计算量较大的场景，缓存数据用于提高性能。

   > 如果依赖项经常变化 or 缓存的数据创建成本不高，则不要去缓存。

   - calculateValue：用于计算数据的函数，是一个**没有任何参数的纯函数**，可以返回任意类型（包括 JSX 组件）。该函数的执行时机为 ①首次渲染时调用 ②之后的渲染中，如果 dependencies 中的依赖项没有发生变化，则使用之前调用该函数所计算的数据；否则调用该函数计算最新数据。
   - dependencies：在 calculateValue 中使用的**响应式变量**（props、state、组件中定义的变量和函数）所构成的依赖数组。

2. 代码演示

```tsx
import { useMemo } from 'react';

function TodoList({ todos, tab, theme }) {
```

```
  // 除了首次渲染，只有 todos 或 tab 变化时 visibleTodos 对应的 calculateValue 函数才会重新执行，计算最新数据；否则使用缓存数据
  const visibleTodos = useMemo(() => filterTodos(todos, tab), [todos, tab]);
  // ...
}
```

useCallback - 缓存函数

1. 优化原理：useCallback 用于**缓存函数**，该 Hook 接收两个参数 fn、dependencies，前者表示想要缓存的函数，后者表示依赖项数组。该 Hook 返回值为最新的缓存函数。
   a. fn：想要缓存的函数，该函数**可以接收任何参数**，并返回任意类型。React 不会调用该函数，而是会返回该函数。①首次渲染时，useCallback 返回 fn ②之后的渲染中，如果 dependencies 中的依赖项没有发生变化，则返回相同的 fn；否则返回最新的 fn 并将其缓存。
   b. dependencies：与 useMemo 的 dependencies 类似，是控制是否更新 fn 的**响应式变量**所构成的数组。
2. 代码演示

```
import { useCallback } from 'react';

function ProductPage({ productId, referrer, theme }) {
  const handleSubmit = useCallback((orderDetails) => {
    post('/product/' + productId + '/buy', {
      referrer,
      orderDetails,
    });
  }, [productId, referrer]);
  // ...
```

memo - 记忆组件

1. 优化原理：memo **允许组件在 props 没有改变的情况下跳过重新渲染**。该函数接受两个参数 Component、arePropsEqual（可选），前者表示要记忆的组件，后者是一个可选的函数参数，用于进行新旧 props 是否变化的比较。该函数返回值为一个新的 React 组件，其行为与 Component 相同，但是当它的父组件重新渲染时，如果其 props 没有变化，则可以跳过重新渲染，提高性能。
   - Component：要进行记忆化的组件。memo 不会修改该组件，而是返回一个新的、记忆化的组件。
   - arePropsEqual?：函数，接收两个参数，prevProps、curProps，分别表示前一个 props 和新的 props。如果新旧 props 相等，则应该返回 true，否则返回 false。该参数一般不需指定。

2. 代码演示

```
const Greeting = memo(function Greeting({ name }) {
  return <h1>Hello, {name}!</h1>;
});

export default Greeting;
```

代码体积分析

1. 代码体积分析工具：`vite-bundle-visualizer`
2. 使用步骤
   a. 安装 `npm i vite-bundle-visualizer -D`
   b. 在 package.json 中配置 npm scripts

```
{
    "scripts": {
        "visualize-treemap":"vite-bundle-visualizer",
        "visualize-sunburst":"vite-bundle-visualizer -t sunburst",
        "visualize-network":"vite-bundle-visualizer -t network",
        "visualize-rawdata":"vite-bundle-visualizer -t raw-data"
    },
}
```

   c. 执行不同的 npm scripts 得到不同类型的代码分析

```
npm run visualize-treemap
npm run visualize-sunburst
npm run visualize-network
npm run visualize-rawdata
```

代码组织优化

优化 lodash 体积

loadash 包体积优化 - 使用 lodash-es

1. 安装 `npm i lodash-es`

2. 引入 `import {cloneDeep} from "lodash-es"`

减少首页加载数据的体积

拆分首页要加载数据 - 使用路由懒加载（避免一上来就加载所有路由组件）

> 路由懒加载允许将代码拆分为更小的模块，并按需加载

```
const Edit = lazy(() => import(/* webpackChunkName: "editPage" */ '@pages/Question/Edit'));
const Stat = lazy(() => import(/* webpackChunkName: "statPage" */ '@pages/Question/Stat'));
```

> 注意：为了在 Vite 中使用 Webpack 的基于 webpackChunkName 注释设置 chunk 名称的特性，需要借助 vite-plugin-webpackchunkname
>
> Step1. 安装 `npm install --save-dev vite-plugin-webpackchunkname`
>
> Step2. 配置 vite.config.js 使用插件
>
> ```
> // vite.config.ts
> import { manualChunksPlugin } from 'vite-plugin-webpackchunkname'
> // Other dependencies...
>
> export default defineConfig({plugins: [manualChunksPlugin(),]})
> ```
>
> Step3. 路由组件的懒加载
>
> ```
> import(/* webpackChunkName: "detail" */ "@/detail/somepage.vue")
> ```

抽离公共代码，合理使用缓存

1. 优化原理：Vite 将项目代码打包为多个 chunk。项目中的 `src` 代码经常变化，而依赖项版本则固定。因此，将内容固定的包与 `src` 分开打包，浏览器在多次加载时只需频繁加载较小的 `src` chunk，而内容固定的包可以利用 304 缓存，从而减少不必要的加载，提升性能。
2. 优化方法：在 vite.config.ts 中，通过 build.rollupOptions.output.mannualChunks 来自定义哪些模块应该被分到同一个 chunk 中，从而优化代码分块的策略。

```
export default defineConfig({
  build: {
```

```
    rollupOptions: {
      output: {
        manualChunks(id) {
          // 1. 优先级最高：将 antd 相关的代码打包到 antd-chunk
          if (id.includes('/antd/')) {
            return 'antd-chunk';
          }
          // 2. 次高优先级：将 react-dom 相关的代码打包到 reactDom
          if (id.includes('/react-dom/')) {
            return 'reactDom';
          }
          // 3. 最低优先级：将所有其他 node_modules 相关的代码打包到 vendors-chunk
          if (id.includes('/node_modules/')) {
            return 'vendors-chunk';
          }
        },
      },
    },
  },
});
```