TREE SHAKING КАК МЕТОД ОПТИМИЗАЦИИ СБОРКИ WEB-ПРИЛОЖЕНИЯ

Мазура А. А., Гуринович А. Б.
Кафедра информационных технологий автоматизированных систем,
Белорусский государственный университет информатики и радиоэлектроники
Минск, Республика Беларусь
E-mail: mazura.anastasiya@gmail.com, gurinovich@bsuir.by

В процессе разработки программного обеспечения использование сборщиков приложений, имеющих по умолчанию установленный tree shaking, не дает заметного результата для оптимизации. Поэтому исследования принципов для эффективной оптимизации web-приложения данным методом актуальны.

Введение

Tree shaking — это термин, обычно используемый в контексте JavaScript для устранения «мертвого» кода. Он опирается на статическую структуру из ES2015 модуля синтаксиса, то есть import и export. Название и концепция были популяризированы благодаря накопительному пакету модулей ES2015.

Как работает tree shaking:

- Объявляются импорты и экспорты в каждом модуле.
- Сборщик (Webpack, Rollup или другой) во время сборки анализирует дерево зависимостей.
- Неиспользуемый код исключается из итоговой сборки.

К сожалению, для правильной работы tree shaking одной настройки сборщика недостаточно. Чтобы достичь лучшего результата, необходимо учесть множество деталей, а также удостовериться, что модули не были пропущены при оптимизации.

I. Использование синтаксиса ES6 для импортов и экспортов

Использование ES6 импортов и экспортов — первый и важнейший шаг к работающему tree shaking.

Большинство других реализаций паттерна «модуль», включая common.js и require.js, являются недетерминированными в процессе сборки. Эта особенность не позволяет сборщикам типа Webpack точно определить, что импортируется, что экспортируется и, как следствие, какой код может быть безболезненно удалён.

```
module.exports.foo = function foo() {}

module.exports[someVariable] = function foo() {}

function exportFoo() {
  module.exports.foo = function foo() {}
}

function importFoo() {
  const { foo } = require(baseDirectory + '/foo.js');
}
```

Рис. 1 – Варианты, возможные при использовании common.js

При использовании ES6 модулей возможности импорта и экспорта более ограничены:

- импорт и экспорт только на уровне модуля, а не внутри функции;
- имя модуля может быть только статичной строкой, не переменной;
- всё, что импортируется, обязательно должно быть где-то экспортировано.

```
export function foo() {};

import { foo } from './foo';
```

Рис. 2 – Возможности ES6 модулей

Упрощённые правила позволяют сборщикам точно понимать, что было импортировано и экспортировано, и, как следствие, определять, какой код не используется вовсе.

II. Атомарные экспорты

Webpack, как правило, оставляет экспорты нетронутыми в следующих случаях:

- экспортируется объект с большим количеством свойств и методов;
- экспортируется класс с большим количеством методов;
- экспорт объекта по умолчанию используется для экспорта множества разных функций (Рисунок 3).

```
export default {
    add(a, b) {
        return a + b;
    },
    subtract(a, b) {
        return a - b;
    }
}
```

Рис. 3 – Обе функции будут включены в сборку, даже если используется только одна

Такие экспорты будут либо полностью включаться в сборку, либо полностью удаляться. Значит, в итоге может получиться сборка, содержащая код, который никогда не будет использоваться.

Необходимо сохранять экспорты настолько маленькими и простыми, насколько это возможно. Данный подход позволяет сборщику выкидывать больше кода благодаря тому, что в процессе сборки можно отследить, какая из функций была импортирована и использована, а какая не была.

Практика атомарных экспортов также помогает писать код в более функциональном и направленном на переиспользование стиле, а также избегать использования классов там, где это не оправдано.

III. Побочные эффекты на уровне модуля

При написании модулей многие разработчики упускают важный фактор — влияние побочных эффектов.

```
export function add(a, b) {
    return a + b;
}

export const memoizedAdd = window.memoize(add);
```

Рис. 4 – Webpack не понимает, что делает window.memoize, и поэтому не может удалить эту функцию

В примере выше (Рисунок 4) window.memoize будет вызвана в момент импорта модуля.

Как это видит Webpack:

- здесь создаётся и экспортируется чистая функция add — может быть можно удалить её, если она не будет использоваться позже;
- далее вызывается window.memoize, в которую передаётся add;
- сборщик не знает, что делает memoize, но он точно знает, что memoize, возможно, вызовет add и создаст побочный эффект;
- поэтому для сохранности сборщик оставит функцию add в сборке, даже если её больше никто не использует.

В реальности мы уверены, что window.memoize — чистая функция, которая не создаёт никаких побочных эффектов и вызывает add, если кто-то использует memoizedAdd.

Но Webpack этого не знает и для спокойствия добавляет функцию add в итоговую сборку.

```
import { memoize } from './util';

export function add(a, b) {
    return a + b;
}

export const memoizedAdd = memoize(add);
```

Рис. 5 – Исправленная версия кода

Теперь сборщику хватит информации для анализа (Рисунок 5):

- здесь вызывается memoize на уровне модуля, это может повлечь проблемы;
- но функция memoize пришла из ES6 импорта, нужно взглянуть на функцию в util.js;
- действительно, memoize выглядит как чистая функция, здесь нет побочных эффектов;
- если никто не использует функцию add, можно безопасно исключить её из итоговой сборки.

Важно помнить, что когда сборщик не получает достаточно информации для принятия решения, он пойдёт по безопасному пути и оставит функцию.

Без запуска довольно трудно определить, как сборщик будет оптимизировать конкретный модуль.

Поэтому очень удобно для проверки запускать билд и исследовать итоговую сборку. Обязательно необходимо просмотреть JavaScript-код и убедиться, что в нём не осталось ничего лишнего, что должно было быть удалено с помощью tree shaking.

IV. Заключение

Использование исследованных подходов и алгоритмов в процессе разработки web-приложения позволит эффективно применять tree shaking, являющийся неотъемлемой частью сборщика. В свою очередь tree shaking эффективно оптимизирует код, уменьшая размер сборки.

V. Список литературы

1. Frank Zammetti. Modern Full-Stack Development. Using TypeScript, React, Node.js, Webpack, and Docker. – 2020. – с. 396.
2. Juho Vepsäläinen. SurviveJS - Webpack and React. – 2016. – с. 40-43.
3. Webpack bundle analyzer. / Mode of access: https://www.npmjs.com/package/webpack-bundle-analyzer – Date of access: 15.10.2021.
4. DLL Plugin. / Mode of access: https://github.com/webpack/docs/wiki/list-of-plugins#dllplugin – Date of access: 15.10.2021.
5. Thee shaking / Mode of access: https://webpack.js.org/guides/tree-shaking/ – Date of access: 16.10.2021.