АЛГОРИТМЫ ОПТИМИЗАЦИИ СБОРКИ WEB-ПРИЛОЖЕНИЙ

Мазура А. А., Гуринович А. Б.
Кафедра информационных технологий автоматизированных систем, Белорусский государственный университет информатики и радиоэлектроники
Минск, Республика Беларусь
E-mail: irynamazura22@gmail.com, gurinovich@bsuir.by

Для разработчиков проектов с большими объемами программного кода является основной проблема оптимизации сборки web-приложений. Поэтому актуальны исследования методов сборки web-приложений и разработки алгоритмов оптимизации данных методов.

Введение

JS-приложения, сайты и другие ресурсы становятся сложнее и инструменты сборки – это реальность web-разработки. Сборщики помогают упаковывать, компилировать и организовывать приложения. По мере развития и роста приложения увеличивается и время его сборки – от нескольких минут при пересборке в development-режиме до десятков минут при «холодной» production-сборке. Это оказывает влияние на продуктивность работы и в дальнейшем может значительно увеличить время разработки приложения.

Большой объем сборки замедляет загрузку приложения, что в перспективе негативно влияет на отношение пользователей. По статистике Google, если сайт загружается дольше трех секунд, 53% пользователей покинут его. Каждый второй ожидает, что страница загрузится менее чем за 2 секунды. 46% людей говорят, что ожидание загрузки страниц - это то, что им больше всего не нравится при просмотре веб-страниц [1].

I. Анализ сборки web-приложения

Первым шагом к оптимизации процесса сборки web-приложения является анализ профиля его сборки. В случае с Webpack существует инструмент [2], позволяющий визуализировать содержимое сборки и оценить размеры включенных файлов:

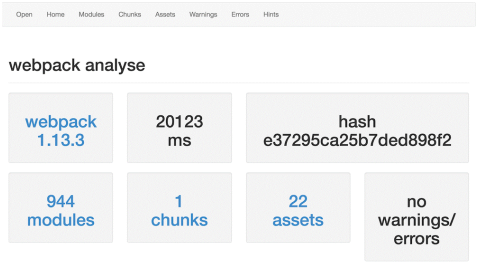

Рис. 1 – Визуализация результатов анализа профиля сборки

В данном примере на процесс сборки было затрачено около 20 секунд и скомпилировано 944 модуля в 1 файл фрагмента. Это означает, что Webpack должен обрабатывать 944 модуля при каждой компиляции. При переходе в раздел модулей появляется график, представляющий отношения между необходимыми модулями и исходным кодом.

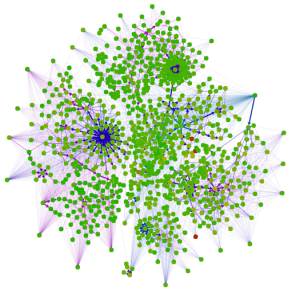

Рис. 2 – Визуализация результатов анализа профиля сборки

Webpack необходимо обработать все это дерево, чтобы собрать проект. На изображении можно идентифицировать несколько узлов с большим количеством связанных зависимостей. В большинстве случаев эти узлы являются точками входа в библиотеки, которые используются в проекте. Но зачем включать их в процесс сборки, если они не меняются? Это один из важных моментов для оптимизации сборки: нет необходимости перекомпилировать библиотеки.

II. Оптимизация сборки

Кеширование позволит сохранять результаты вычислений для дальнейшего переиспользования. Первая сборка может быть немного медленнее обычной из-за накладных расходов на кеширование, однако последующие будут гораздо быстрее за счет переиспользования результатов компиляции неизменившихся модулей. Причина, по которой использование исключительно кэширования не помогает, заключается в том, что кеш веб-пакетов предназначен только для повышения производительности инкрементных сборок. Его цель - ускорить обновления в режиме просмотра, а не быстро запускаться. Так что проблему оптимизации сборки полностью это не решит.

С помощью подключаемого модуля DLL webpack можно собрать node_modules только один раз и забыть о нем, пока снова что-то не

изменится в node_modules . В следующих сборках Webpack будет повторно использовать модули из скомпилированного dll-пакета. Процесс немного сложный, но оно того стоит. Сначала создаётся свой dll-bundle и manifest.json с отдельной конфигурацией веб-пакета и DllPlugin. А затем применяется DLLReferencePlugin для повторного использования dll-bundle в любой другой сборке веб-пакета. Значительный плюс в том, что он работает не только с node_modules. Можно переместить любую часть приложения в dll-bundle. Единственная особенность при использовании DllPlugin заключается в том, что при импорте из пакета npm, он не знает package.json: main и сохраняет полный путь к импортированному файлу. Для разработчика это означает, что после сборки dll-bundle в manifest.json вместо путей импорта, таких как redux, будут найдены ./redux/es/index.js. Это нарушит пути импорта приложения. Чтобы решить эту проблему, нужно передать измененное содержимое manifest.json в DLLReferencePlugin [3].

Также добиться оптимизации процесса сборки можно с помощью минификации JS-кода, к примеру, библиотекой UglifyJS. Minify (также известный как uglify) в программировании - это процесс удаления всех ненужных символов из исходного кода без изменения его функциональности. Эти ненужные символы обычно включают в себя символы пробела, символы новой строки, комментарии и иногда разделители блоков, которые используются для повышения читабельности кода, но не требуются для его выполнения. Минифицированный исходный код особенно полезен для интерпретируемых языков и форматов обмена данными, развертываемых и передаваемых в Интернете (например, JavaScript), поскольку он уменьшает объем данных, которые необходимо передать.

Рис. 3 – Пример минифицированного кода

Минифицированный исходный код может быть отменен с помощью процесса beautify (также известного как prettify), который применяет к нему различные стилистические соглашения о форматировании. Эти соглашения о форматировании обычно регулируют позиционирование, интервалы и аналогичные изменения, предназначенные для облегчения просмотра, чтения и понимания содержимого [4].

Для CSS и изображений необходимо использовать специальные плагины и загрузчики, а версионирование сборки позволит пользователю перезагружать минимально возможную часть кода.

В данной работе исследуется эффективность совместного использования алгоритмов кэширования, минификации и параллелизации процесса сборки web-приложения.

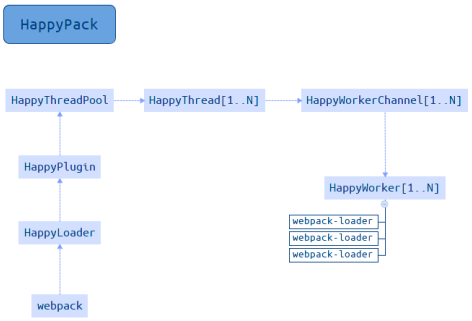

Рис. 4 – Алгоритм параллелизации потоков сборки

III. Заключение

Результаты анализа сборки web-приложения до и после оптимизации показали, что использование алгоритма параллелизации потоков сборки совместно с кэшированием и минификацией позволит в некоторых случаях ускорить процесс в несколько раз и значительно уменьшить размер сборки, что в дальнейшем позволит ускорить загрузку приложения на сервере.

Список литературы

1. The need for mobile speed. / Mode of access: https://blog.google/products/admanager/the-need-for-mobile-speed/ – Date of access: 17.10.2020.
2. Webpack bundle analyzer. / Mode of access: https://www.npmjs.com/package/webpack-bundle-analyzer – Date of access: 17.10.2020.
3. DLL Plugin. / Mode of access: https://github.com/webpack/docs/wiki/list-of-plugins#dllplugin – Date of access: 17.10.2020.
4. JS Minify. / Mode of access: https://www.uglifyjs.net/ – Date of access: 17.10.2020.