Abstract

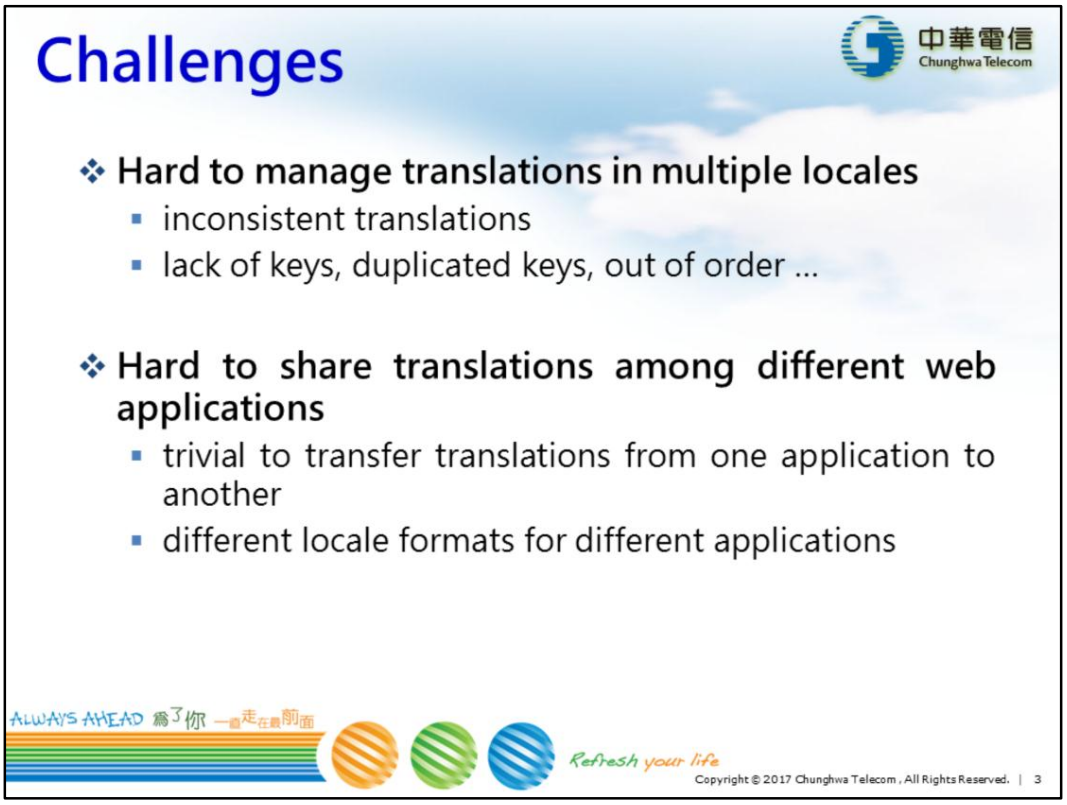
For modern web, it is more and more important to have global websites, which means development teams have to translate their websites to different languages so that they can promote products or services to different cultures, regions, or countries. However, website globalization is never easy, especially when we have already built a lot of complicated web applications. To simplify the process, we have come up with an automatic mechanism which integrates various implementations and procedures.

Keywords: internationalization, localization

1. Introduction

In computing, internationalization (i18n) and localization (l10n) are means of adapting computer software to different languages, regional differences and technical requirements of a target market (locale) [1].

Generally, internationalization means abstracting all of the strings and other locale-specific code (such as date or currency formats) out of applications, and localization means giving abstracted parts translations and localized formats.

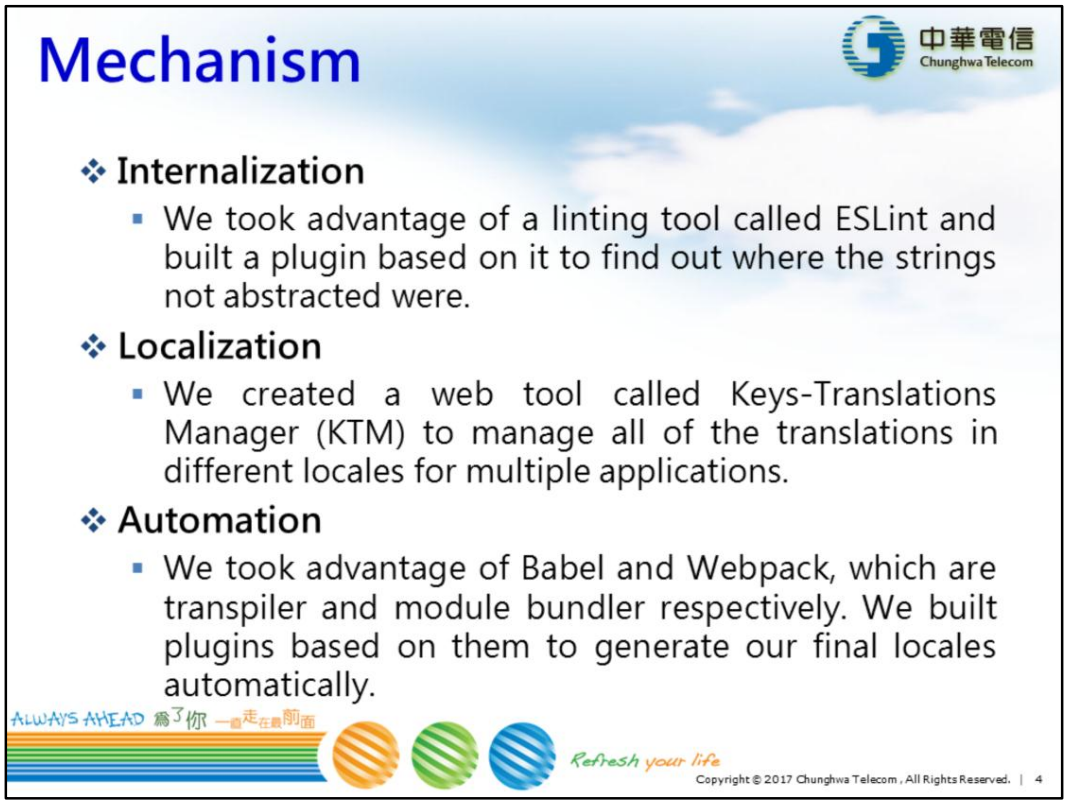


2. Challenges

If our web applications were all small projects, it is would be sufficient to manage locales manually. However, these applications usually grew rapidly, and that made us difficult to keep abstracted keys and translations consistent for all locales. By and large, we confronted a couple of challenges while maintaining locale-related resources without tools:

(1) Managing translations for multiple locales: Sometimes, developers needed to adjust or revise translations to make phrases more readable or meaningful, but they didn't update those for the other locales at that moment, which made our translations inconsistent. Moreover, the number or the sequence of these translations might be diverged sometime. In either case, it's really painful to make it right again.

(2) Sharing translations among different web applications: Developers might develop and maintain more than one application at the same time, and there are definitely numerous translations which can be reused. We could copy these locale translations from one application and paste them to another, but it's obviously not an efficient way. What's worse, the format of locales for different applications might not be the same, like JSON format for one application and Properties format for another one.

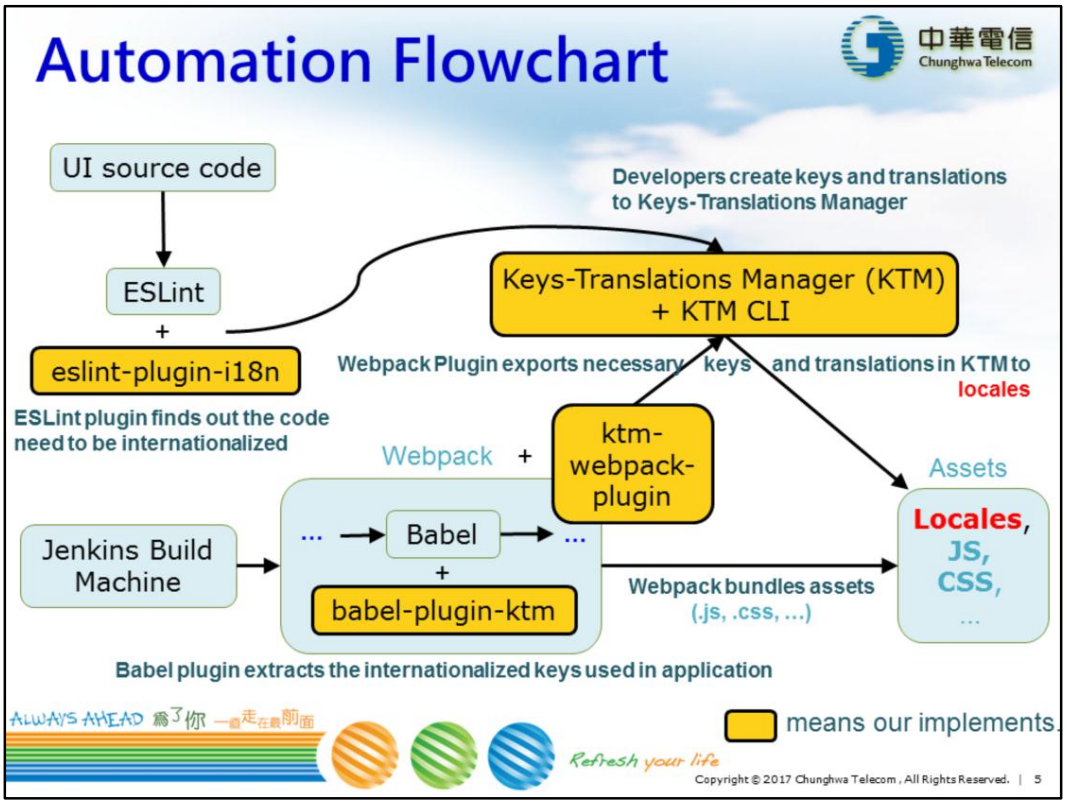


3. Mechanism

To tackle the aforementioned problems, we have taken a few measures. First of all, we created a plugin based on a linting tool called ESLint [2] to help developers find out where the strings needed to be abstracted were. The ESLint plugin helped us do internalization.

In the next place, we built a web tool, called Keys-Translations Manager (KTM), to manage all of the translations in different locales for multiple applications, and allow the translations to be shared among these applications. KTM helped us do localization.

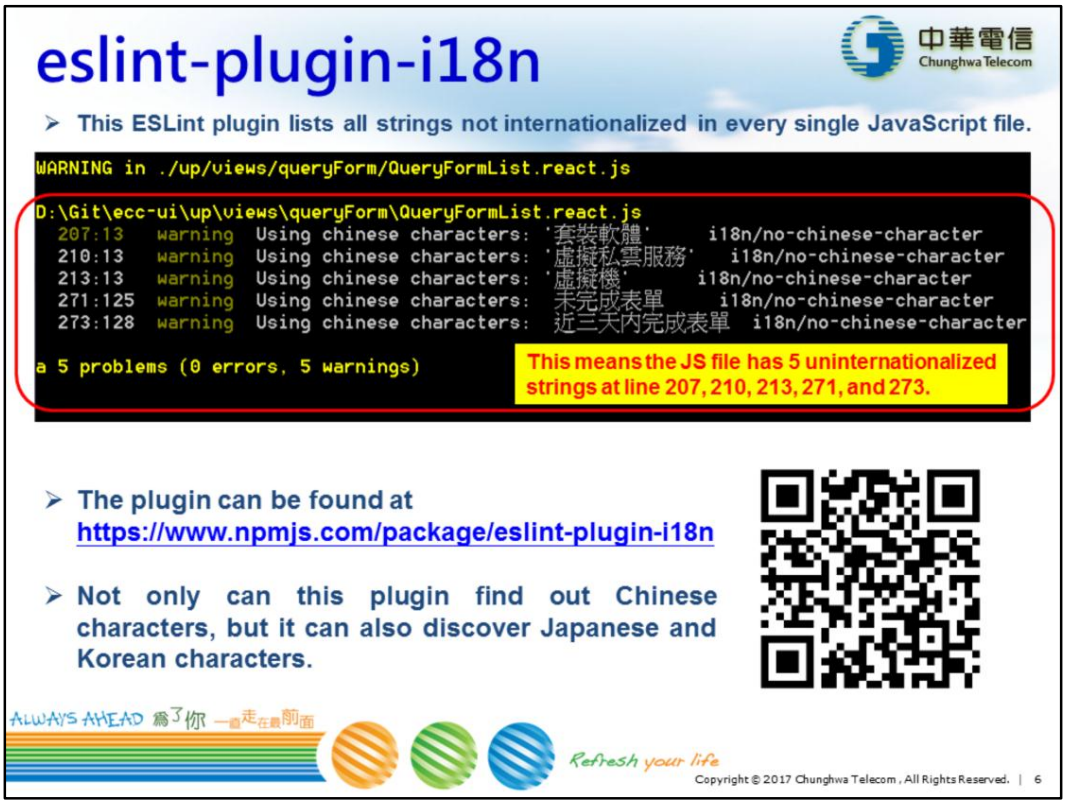
Finally, we implemented Babel [3] and Webpack [4] plugins to facilitate our workflow. Originally, Babel is a transpiler which assists us transpile the code; Webpack is a module bundler which assists us bundle our assets. Based on these, we built our own plugins to automatically export locales. These plugins helped us do automation and reduced manual works.



3.1 Automation Flowchart

As the above figure shows, we applied ESLint to analyze our UI source code, and the plugin we built based on ESLint would show us where the uninternationalized strings were. After analyzing, we abstracted and translated these strings, and then stored them to Keys-Translations Manager to decrease the complexity of managing a great deal of translations.

Next, our continuous integration and continuous delivery (CI/CD) workflow would take over the automation process. It triggered the build process with Webpack, and used Babel to parse the entire code. At this time, the Babel plugin we built would reveal the internationalized keys which exactly used in the application so that we would not put outdated keys and translations to our locales. After that, the keys discovered by the Babel plugin would be passed to Webpack, and then the Webpack plugin we built would take these parameters to query necessary translations from KTM, and exported them to our production environment eventually.

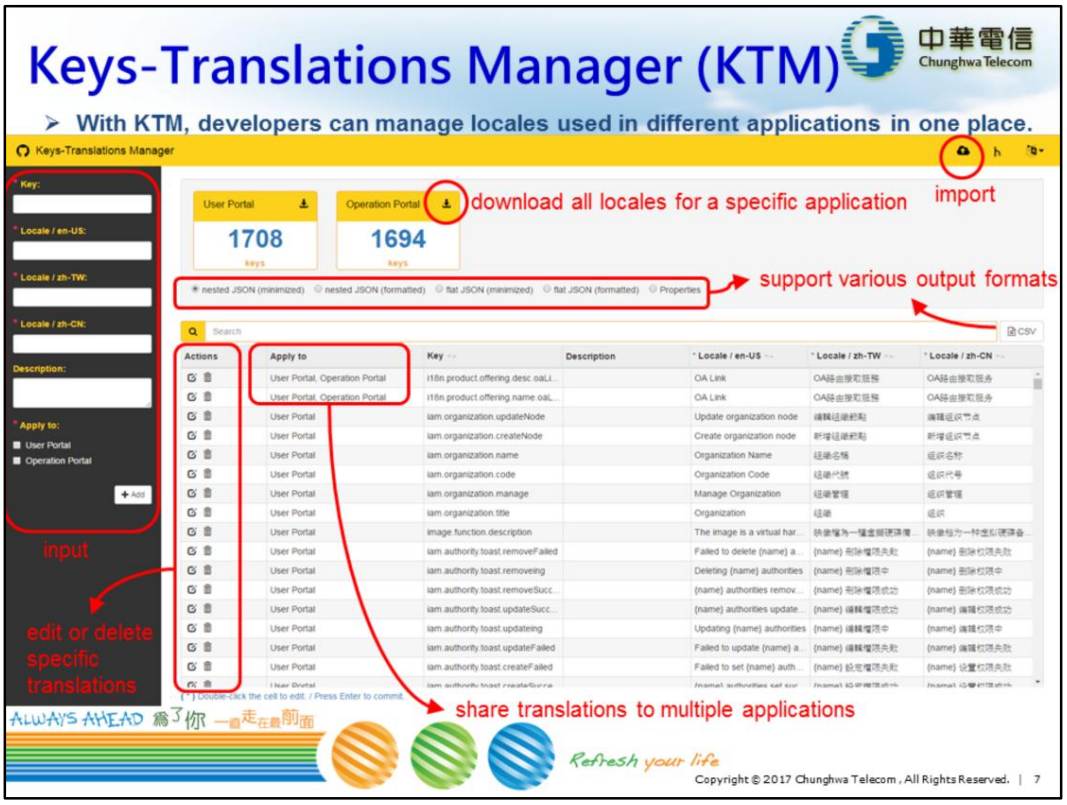


3.2 eslint-plugin-i18n

The first step of internationalization is that we have to abstract the strings in our applications. We have developed many UI modules and the strings which needed to be internationalized are scattered around these modules, and it is difficult to find out all of them manually. However, we have overcome this obstacle by means of ESLint and its plugin.

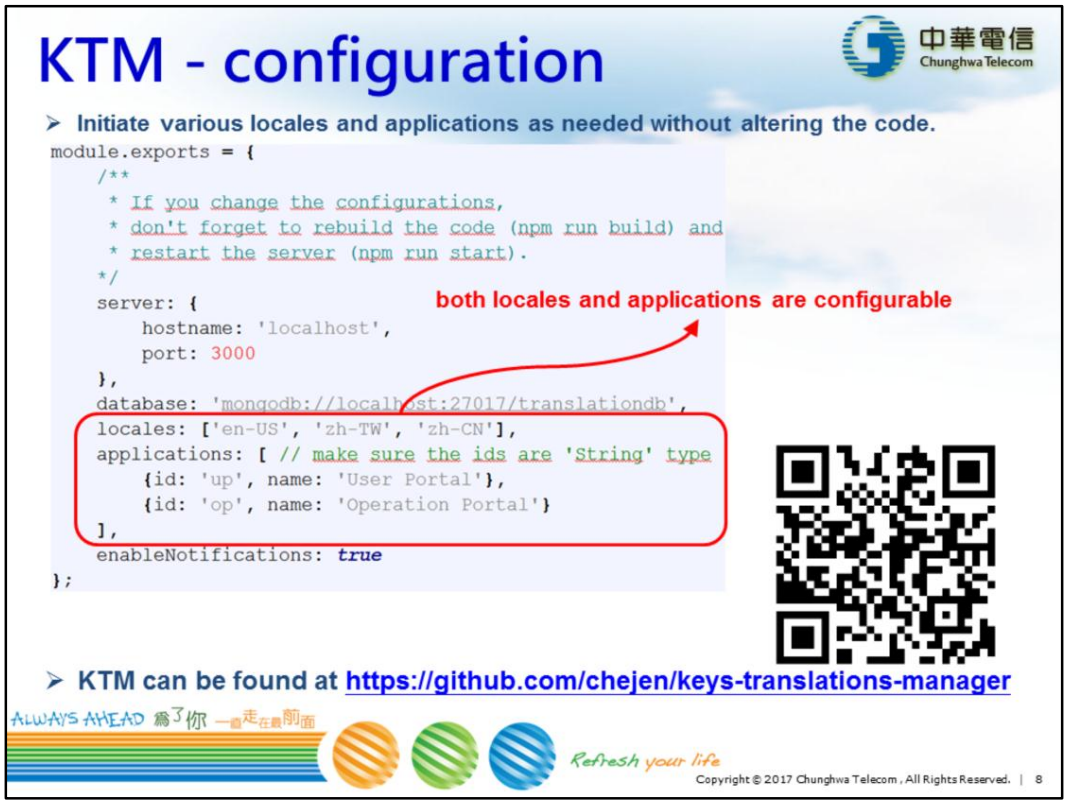
ESLint is a linting tool for JavaScript aiming to make code consistent and avoid bugs. It is a static analysis tool, which means we can use it to analyze our JavaScript code without executing it. Based on ESLint, we develop a plugin called eslint-plugin-i18n to locate the code snippets which have not internationalized yet. In our case, which means there are still some Chinese characters left in our application. As the screenshot shows, eslint-plugin-i18n lists all of the strings which are not internationalized in every JavaScript file.

This plugin supports not only Chinese characters digging, but also Japanese, Korean, and Thai characters. We have made it an open source tool and have published it as an NPM package [5], and thus those developers who have the same need do not have to reinvent the wheel.

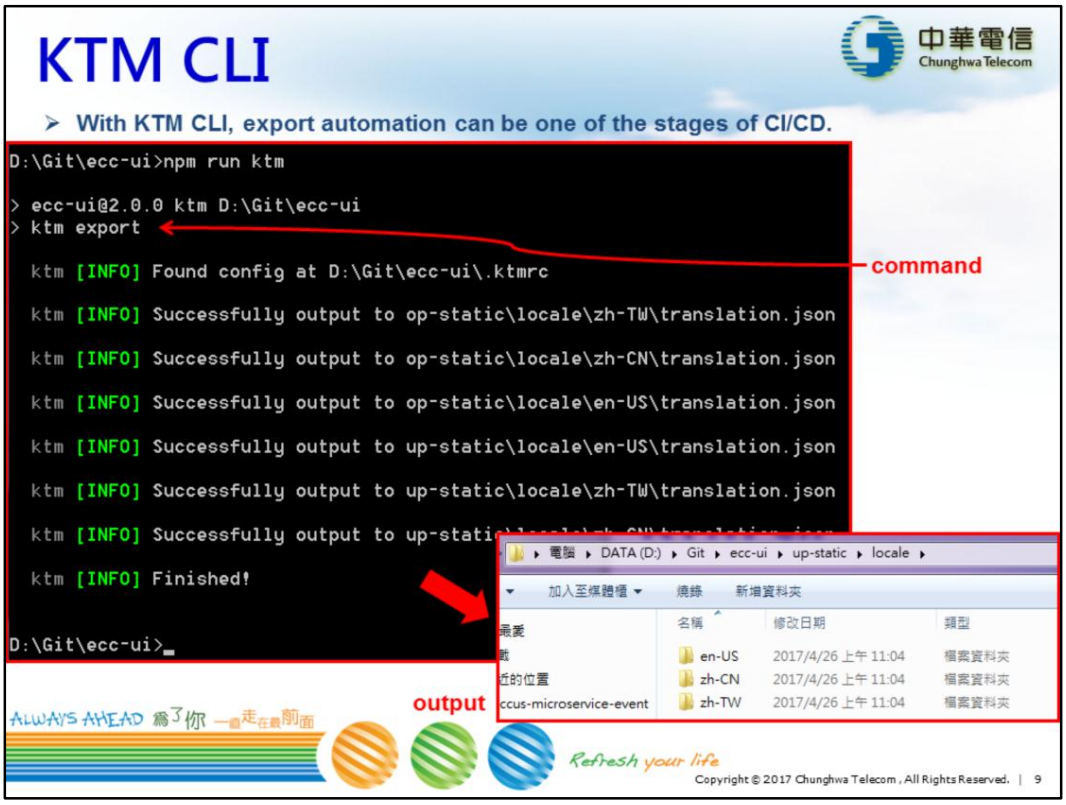


3.3 Keys-Translations Manager (KTM)

Keys-Translations Manager (KTM) is a locale management tool, which lets us manage locales in one place so that we can control multiple locales in different applications at the same time. With KTM, we can simply input, import, and export translations, and we can add, edit and delete translations with effortless ease. KTM supports different kinds of locale formats including nested JSON, flat JSON and Properties. In addition, you can choose whether to minimize these files or not. Normally, we minimize the files for production environment for better performance. KTM also provides basic data validations to keep our data from being inconsistent, which makes it painless to prevent duplicated keys and missing translations.



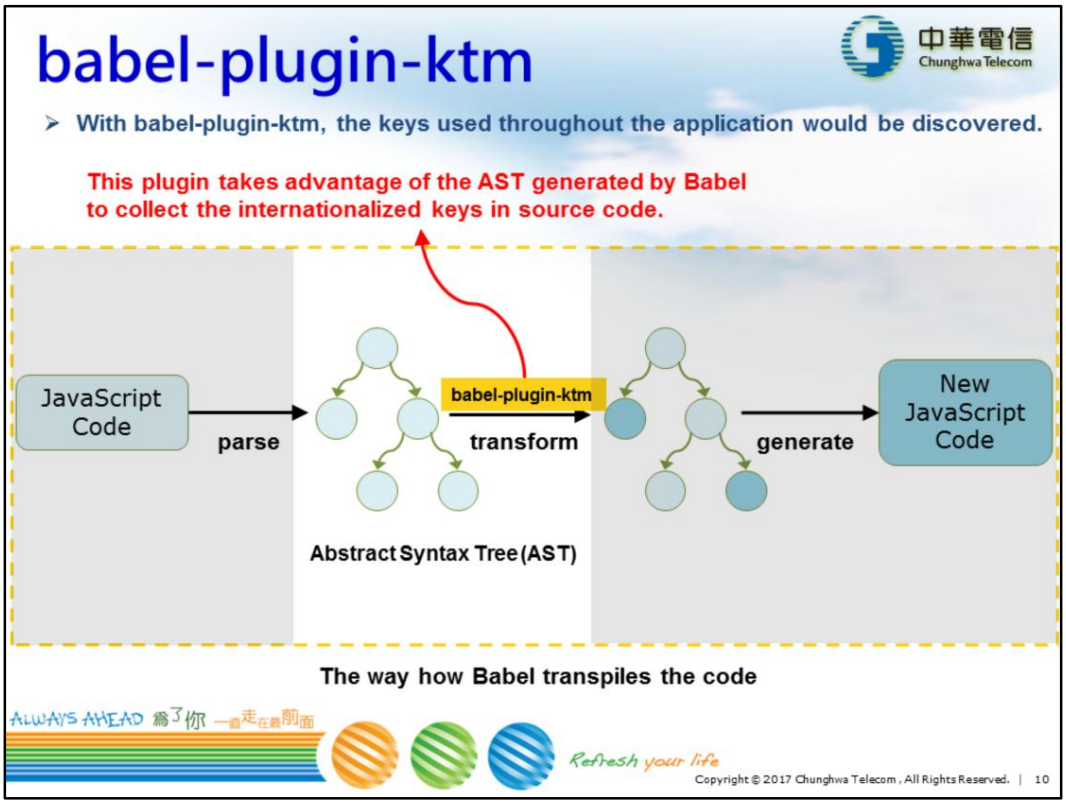
KTM is the second step to improve development processes and is released as an open source project, which can be found at Github [6]. KTM is flexible and configurable, and developers can initiate various locales and applications according to their needs through different configurations without changing a line of the code.



3.4 KTM CLI

We already have a tool (eslint-plugin-i18n) to find out the code we missed to internationalize and an application (KTM) to help us implement localization. However, it is still annoying that we have to download these translations as locale files manually every time when we need to deploy.

This is where keys-translations-manager-cli (KTM CLI) comes in. KTM CLI is a command line tool written in Node.JS. With KTM CLI, we can export locales managed by KTM without navigating to the web. It is useful as we adopt CI/CD workflow in our development process. We have made KTM CLI an open source tool as well [7].



3.5 babel-plugin-ktm

KTM CLI is good, but not good enough. The keys in Keys-Translations Manager grow as time goes by, and some of them might be outdated and would not be used whether temporarily or permanently. If we export all of the keys to locales, we get some unnecessary outdated keys, which might slow down the load time of our web page. Therefore, we utilize Babel to solve this problem.

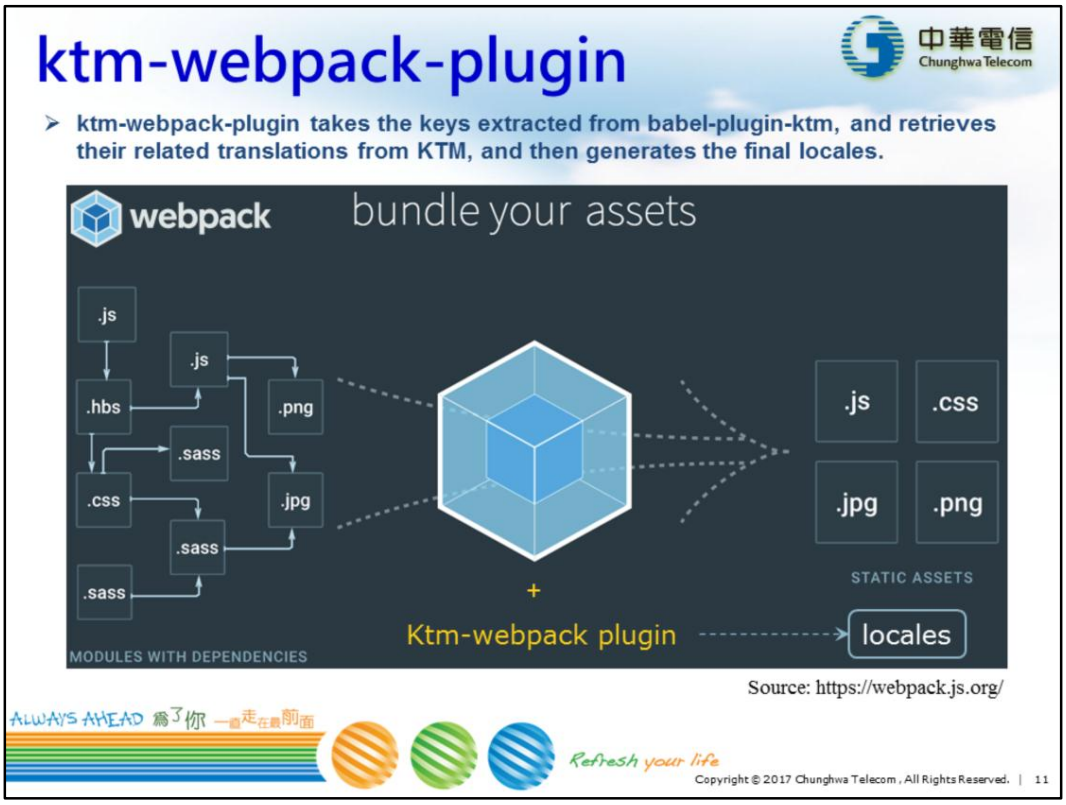
Similar to ESLint, Babel is another static analysis tool. Typically, people use it as a transpiler. This means that we give Babel some code, and Babel modifies the code, and then it gives us the new code. As the above figure shows, Babel takes three stages to process the code:

(1) In **parse** stage, Babel turns JavaScript code into a computer friendly representation called an Abstract Syntax Tree (AST).

(2) In **transform** stage, Babel traverses through the AST and allows us to explore, analyze and modify it. Most importantly, this is where our plugin operates.

(3) In **generate** stage, Babel replaces the original code with the new content generated based on the transformed AST.

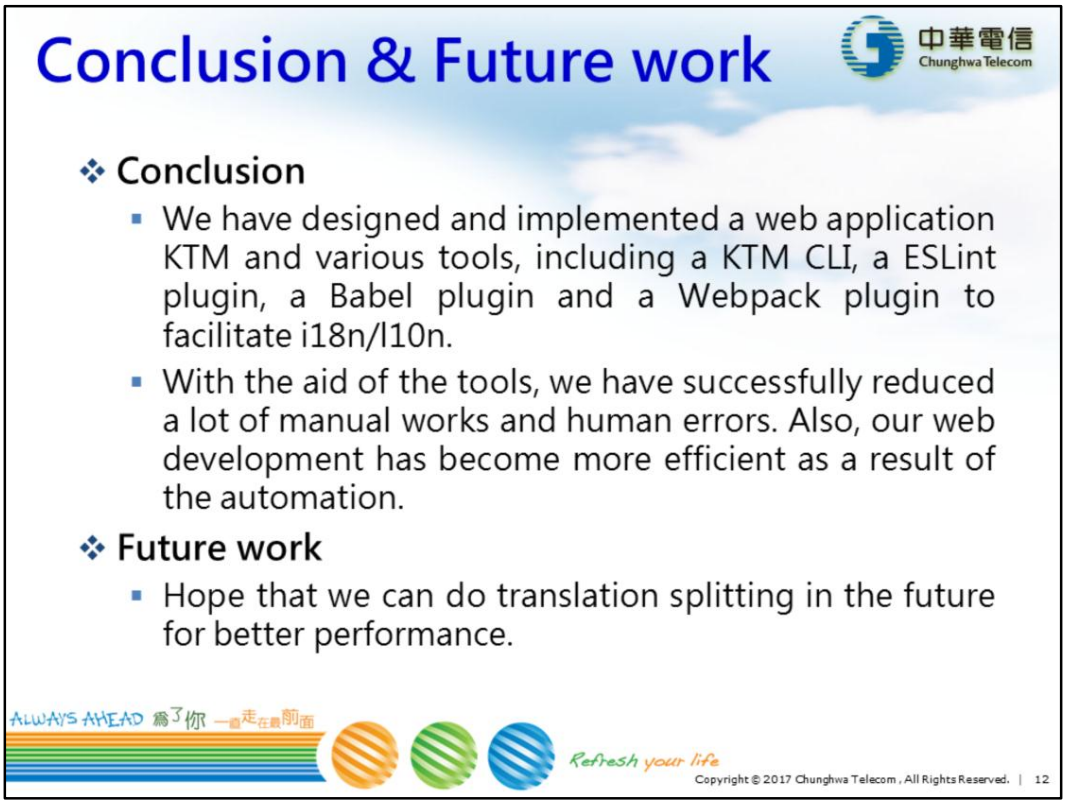
Based on Babel, we have developed a plugin called babel-plugin-ktm to find out the keys actually used throughout the application. This plugin traverses and analyzes the AST in the transform stage after Babel parses the source code, and then collects the internationalized keys according to the configurable string pattern which identifies i18n functions. When three stages are all done, Babel would gather the keys analyzed and extracted from every file, and generate the final result.



3.6 ktm-webpack-plugin

Webpack is a module bundler for modern JavaScript applications. Webpack calculates module dependencies and uses loaders and plugins to process applications, and then packages the complied modules into bundles.

One of the Webpack processes is to transform new syntax to the syntax browsers support through Babel. In other words, Webpack executes babel-loader to process our code before it adds the code to bundles. Hence, we can get the keys extracted via the babel-plugin-ktm we have built, and create our own Webpack plugin — ktm-webpack-plugin to map these keys to their translations managed by KTM, and then export them to locales. Because of these plugins, we will not get redundant data anymore.



4. Conclusion and Future Work

To automate and accelerate our development process, we have designed and implemented a number of tools, including KTM and its command line tool (KTM CLI), an ESLint plugin (eslint-plugin-i18n), a Babel plugin (babel-plugin-ktm) and a Webpack plugin (ktm-webpack-plugin). These tools not only help us to manage locales with ease, but also let us decrease a lot of manual work and increase efficiency.

Although our development process has been improved, there is still some work which can be done. Take big web apps for example, because it is not efficient to put all code into a single file, we usually separate our code into various bundles to load them on demand. However, the translations for a locale cannot be divided currently in most i18n architecture for modern web applications, and have to be downloaded in the beginning when users navigate to the web site. It might be an issue if the size of translations keeps increasing continually, and this is what we want to solve to get better performance in the future.

Reference

[1] Internationalization and localization. In WIKIPEDIA. Retrieved May 5, 2017, from https://en.wikipedia.org/wiki/Internationalization_and_localization
[2] ESLint. Retrieved May 5, 2017, from http://eslint.org/
[3] Babel. Retrieved May 5, 2017, from https://babeljs.io/
[4] webpack. Retrieved May 5, 2017, from https://webpack.js.org/
[5] eslint-plugin-i18n. Retrieved May 5, 2017, from https://www.npmjs.com/package/eslint-plugin-i18n
[6] Keys-Translations Manager (KTM). Retrieved May 5, 2017, from https://github.com/chejen/keys-translations-manager
[7] keys-translations-manager-cli (KTM CLI). Retrieved May 5, 2017, from https://github.com/chejen/keys-translations-manager/tree/master/packages/keys-translations-manager-cli