Zeste de savoir

Rust et WebAssembly

22 mars 2019

Table des matières

1. Introduction

Ce 25 juin 2018, l'équipe *Rust and WebAssembly* , chargée du développement des outils nécessaires à la compilation (ainsi qu'à l'interaction avec les divers composants et l'intégration de) Rust vers WebAssembly, a publié un post présentant la philosophie censée guider les décisions qui seront prises, concernant le support du Wasm.

> L'insertion chirurgicale de sources Rust compilées vers WebAssembly devrait être le meilleur moyen de garantir une haute vélocité lorsque JavaScript couvre des pans de code dont le besoin en performance est critique. Inutile d'abandonner votre code base existante car Rust fonctionne avec. Que vous programmiez en Rust ou sur le Web, votre manière de travailler ne devrait pas changer puisque Rust[1] s'intègre parfaitement à vos outils préférés.

Dans ce billet, nous synthétiserons les principaux points de cette vision pour avoir une vue d'ensemble de ce à quoi pourrait ressembler l'écosystème web et ce que nous serions capables de faire avec, une fois une certaine maturité atteinte.

Pour plus d'informations relatives aux implémentations sous-jacentes, je vous recommande vivement de vous rendre au bas du billet.

2. Pourquoi s'axer sur la performance ?

Dans les cas les plus critiques, JavaScript pose plus de problèmes qu'il n'en résout[2]. Son typage dynamique et ses cycles non-déterministes de garbage collection peuvent être relativement gênants. Même des modifications mineures du code peuvent causer de lourdes régressions de performance, si vous omettez accidentellement de rester dans les bonnes pratiques du compilateur JIT.

1. Sources Rust compilées en wasm.

D'autre part, Rust fournit aux programmeurs des outils leur offrant un contrôle bas-niveau, des performances fiables et le code est libre de toutes pauses non-déterministes imposées par le garbage collector. Les programmeurs disposent également d'un contrôle sur l'indirection , la monomorphization ainsi que l'agencement de la mémoire.

Avec Rust, vous n'avez pas besoin d'être un féru d'optimisations extrêmement familier avec les implémentations sous-jacentes de chaque compilateur JIT JavaScript. Vous pouvez obtenir quelque chose de rapide sans sorcellerie .

3. Ne réécrivez pas, intégrez

Le résultat de la compilation d'une source Rust vers WebAssembly ne dispose pas de runtime . Il en résulte seulement un fichier binaire `.wasm` dont la taille est proportionnelle à la quantité de code Rust, l'intégration ne coûtant alors pas plus cher que ce dont vous avez réellement besoin. Par conséquent, cela simplifie, depuis une code base JavaScript existante, l'adoption incrémentale et partielle de Rust.

4. Préservez votre manière de travailler

Si vous êtes passionné par JavaScript et voulez utiliser une bibliothèque écrite en Rust et compilée en Wasm, vous ne devriez pas avoir à changer votre manière de travailler. Il est possible de publier des packages `.wasm` directement sur npm pour que vous puissiez les ajouter en tant que dépendance dans votre `package.json`, comme vous le feriez pour n'importe quelle bibliothèque JavaScript.

Ces derniers peuvent être importés :

— en tant que modules ECMAScript,
— par le biais de `require` de CommonJS,
— ou ajoutés en tant qu'objets globaux.

Les bundlers comprendront Rust et Wasm comme ils comprennent actuellement JavaScript.

À l'inverse, si vous êtes passionné par Rust et voulez compiler votre crate vers un fichier `.wasm` et la partager sur npm, vous ne devriez pas non plus avoir à changer votre manière de travailler. En réalité, vous ne devriez même pas avoir à installer npm, Node.js ou même un environnement de développement dédié à JavaScript. `wasm-pack` compilera, optimisera et générera les intégrations à JavaScript nécessaires pour votre crate. Une fois fait, il se chargera de la publier également, pour vous, sur npm !

5. Conclusion

On en a fini pour cette fois-ci !

2. Nous ne parlons ici que dans le cadre des **performances**.

6. Source

— Le blog de l'équipe *Rust and WebAssembly*

7. Pour aller plus loin

Je vous conseille vivement de consulter les sections dédiées à `wasm-bindgen` et `wasm-pack` ainsi que le post *Making WebAssembly better for Rust & for all languages* qui explique plus en détails les processus de conversion jusqu'à l'intégration d'un module.

Vous pouvez également lire mon billet concernant `wasm-bindgen`, j'y présente une intégration très basique mais qui peut vous donner une idée de ce à quoi c'est censé ressembler.

Bonne lecture !

Liste des abréviations

JIT Just In Time. 1, 2