Journal : Un petit tour des systèmes de build

Posté par **Julien Jorge** (**page perso**) le 14/06/18 à 23:31. **Licence CC by-sa**
Tags : aucun

Sommaire

- Quelques systèmes de build
    - Make
    - Autotools
    - SCons
    - Premake
    - CMake
    - Ninja
    - Meson
    - FASTbuild
    - Sharpmake
    - Maven
    - Ant
    - Gradle
- Que choisir

Parlons un peu de systèmes de build.

Mon métier consiste à programmer des jeux vidéos destinés aux plates-formes mobiles Android et iOS. Ce qui est commun aux deux plates-formes, c'est-à-dire la plus grosse partie de l'application, est écrit en C++, et ce qui est spécifique à la plate-forme est en Java ou en Objective-C. L'intérêt principal de tout faire en C++ est que les développeurs peuvent lancer l'application directement sur leur poste de travail, sous Linux ou OSX, et tester leurs modifs sans payer le prix d'un émulateur ou du transfert sur un appareil mobile.

L'inconvénient est que l'on se retrouve à gérer des builds sur quatre plates-formes.

Pour compiler une application pour iOS il faut nécessairement un projet Xcode, bien que les bibliothèques puissent être compilées classiquement en ligne de commande. Cela signifie qu'il faut maintenir un fichier de projet Xcode ou alors avoir un outil pour le générer.

Côté Android la compilation du code C++ se fait avec l'outil `ndk-build`, qui est en réalité une interface à GNU Make. Du coup les fichiers de projet sont évidemment des makefiles mais utilisant des variables et fonctions d'une sorte de framework dans le langage de Make. Là encore il faut maintenir des makefiles pour le projet, ou avoir un outil pour les générer.

Pour Linux et OSX la compilation peut se faire avec des outils plus classiques mais évidemment pas avec les fichiers de build iOS ou Android. Et encore une fois il faut maintenir ces fichiers de build.

À moins qu'on ait un outil pour les générer tous…

Quelques systèmes de build

Le problème de la gestion de la compilation d'un projet a été posé bien avant ta naissance ; c'est pour cela qu'il n'existe qu'une seule méthode pour le faire, qui fait consensus depuis des décennies.

Ha ! Ha ! Ha !

Je peux te citer déjà douze outils de tête. Voyons un peu la présentation faite sur leurs sites respectifs et essayons de rester de bonne foi.

Make

D'après le site :

> *GNU Make est un outil qui contrôle la génération d'exécutables et autres fichiers non-source d'un programme à partir des fichiers source du programme.*

J'ai mis la description du site de GNU Make mais la première version de Make date d'avril 1976. Elle précède ainsi le projet GNU de sept ans.

Le principe de base de Make est assez simple, on lui passe un fichier Makefile qui indique « voilà le fichier que je veux créer, il dépend de ces autres fichiers, et voilà les commandes à exécuter pour le créer ». Avec le mécanisme des dépendances l'outil se débrouille pour créer les cibles dans l'ordre.

Il m'arrive encore de créer des petits Makefiles manuellement mais c'est vite pénible. Par exemple, si un binaire `app` est créé à partir d'un fichier `a.cpp` qui inclut un fichier `b.hpp` qui inclut lui-même un fichier `c.hpp`, en toute rigueur je dois lister ces trois fichiers en dépendances de la cible `app` pour que la compilation soit refaite quand l'un d'eux est modifié. On fait ça une fois puis on cherche un truc pour que les dépendances soient gérées automatiquement.

Un autre point difficile à gérer concerne la dépendance au fichier Makefile lui-même. Par exemple, si j'ajoute une option qui concerne l'édition des liens, celle-ci ne sera pas refaite par défaut.

Autotools

> *[Le premier objectif des Autotools] est de simplifier le développement de programmes portables. Le système permet au développeur de se concentrer sur l'écriture du programme, simplifiant de nombreux détails de portabilité entre systèmes Unix et même Windows, et permettant au développeur de décrire comment construire le programme via des règles simples plutôt que des Makefiles complexes.*

Si vous avez déjà utilisé les autotools vous êtes probablement morts de rire en lisant ce paragraphe. Cette suite d'outils qui se présente comme simple et portable est en réalité ce qu'il y a de plus complexe en système de build et n'est portable que sur les systèmes Unix en évitant soigneusement Windows, le plus déployé des systèmes des trente dernières années.

Sans doute révolutionnaire à sa création, cet outil a extrêmement mal vieillit. Pour ma part j'ai eu l'occasion de compiler des logiciels tiers sous Windows avec ça et c'était tout simplement l'enfer. J'ai aussi tenté de l'utiliser pour mes propres projets et je me suis arrêté dans le tuto. Depuis je suis convaincu que ces outils sont faits pour les développeurs qui aiment transpirer.

SCons

> *SCons est un outil Open Source de construction de logiciel—plus précisément, une nouvelle génération d'outil de construction. Voyez SCons comme un substitut amélioré et multi-plateforme pour l'utilitaire Make classique avec des fonctionnalités intégrées similaires à autoconf/automake et aux caches de compilation tels que ccache. Pour faire court, SCons est un moyen plus facile, plus fiable et plus rapide de construire un logiciel.*

Wow, là c'est carrément le futur.

Les fichiers SCons s'écrivent directement en Python, ce qui est assez sympa puisqu'on se retrouve avec un vrai langage et un a accès à un tas de bibliothèques pour décrire le projet.

J'ai compilé quelques projets en SCons il y a une quinzaine d'années et je crois que je n'ai jamais rien vu d'aussi lent. Et pas seulement parce que c'est du Python…

Il faut dire que le comportement par défaut de l'outil est de détecter les modifications des fichiers sources en calculant un md5 des fichiers, pour éviter de recompiler des fichiers dont seule la date a changé. Quand on voit le prix d'un md5 je trouve ce choix très discutable. Ce comportement peut être modifié via une option mais même en passant sur une comparaison de timestamps et en appliquant toutes les astuces connues pour le rendre plus rapide, ça reste hyper lent.

Premake

> *Un système de configuration de construction puissamment simple*
> *Décrivez votre logiciel qu'une seule fois, via la syntaxe simple et facile à lire de Premake, et construisez-le partout.*
> *Générez des fichiers de projets pour Visual Studio, GNU Make, Xcode, Code::Blocks, et plus sous Windows, Mac OS X et Linux. Utilisez l'intégralité du moteur de script Lua pour que la configuration de la construction devienne du gâteau.*

Pfiou, ça claque ! Ce n'est que le quatrième dans la liste et ça promet déjà de tout résoudre

Avec Premake les scripts de builds sont écrits en Lua et, à l'instar de SCons, cela permet de « programmer » son build en profitant de tout un tas de bibliothèques.

Sur le papier c'est très sympa, en pratique ça ne marche pas trop comme attendu. Il s'avère que la syntaxe de Premake est déclarative et ne se mélange pas bien avec le procédural de Lua. Par exemple, si j'écris

```
project "P"

if false then
    files
    {
      "file.cpp"
    }
end
```

On pourrait croire que le fichier `file.cpp` ne fera pas partie du projet `P`, et bien pas du tout, le `if` ne change rien du tout ici. Difficile de programmer dans ces conditions.

Nous utilisons Premake depuis presque quatre ans au boulot. Le choix s'est fait en sa faveur bien que l'outil était en alpha car il permettait de générer des fichiers pour Xcode et ndk-build via deux plugins indépendants, en plus des Makefiles classiques. Aussi, il avait l'air facile à hacker ce qui était rassurant.

Maintenant j'essaye de le remplacer autant que possible.

Parmi les problèmes récurrents le plus pénible est certainement le message `error: (null)`, sans autre info, que l'outil affiche parce qu'une erreur s'est glissée dans un script. Bonne chance pour déboguer ça. J'aime aussi beaucoup le message `Type 'premake5 --help' for help` qui s'affiche quand je fais une faute de frappe sur la ligne de commande. Là encore il n'y a aucune information utile et il faut se débrouiller pour trouver où on s'est trompé. Autre souci : il n'y a pas moyen de mettre des propriétés de link spécifiques à une bibliothèque. C'est embêtant quand on a besoin d'un `-Wl,--whole-archive`.

Le développement de Premake en lui-même a l'air très laborieux. Quatre ans après il est toujours en alpha, avec une release datant d'août 2017. Le module Xcode a été intégré mais ne gère que OSX. Il a fallu réappliquer toutes nos modifs pour iOS. Quant au module pour le NDK il ne fonctionne plus suite à des changements dans Premake (hey, c'est une alpha…). Là encore il a fallu repatcher.

Il y a de nombreux contributeurs, dont des gros, mais chacun a l'air d'aller dans sa propre direction sans qu'il y ait d'objectif commun. Il y a par exemple deux générateurs de Makefiles, `gmake` et `gmake2` (j'attends impatiemment `yagmake`). Il y a des fonctionnalités qui ne marchent qu'avec Visual Studio, d'autres trucs qui fonctionnaient il y a quatre ans et qui ne fonctionnent plus. Ça m'a l'air d'être typiquement le projet qui veut tout faire parfaitement et qui au final ne fait rien de bien. Bref, le produit n'est pas à la hauteur du pitch.

CMake

> *CMake est une suite d'outils open source et multi-plateforme conçus pour construire, tester et empaqueter des logiciels. CMake est utilisé pour contrôler le processus de compilation du logiciel via des fichiers de configuration indépendants du compilateur et de la plate-forme, et il génère des fichiers Makefiles natifs ou des projets qui peuvent être utilisé avec l'environnement de compilation de votre choix.*

CMake est tout simplement mon outil de build préféré de ces quinze dernières années.

Tout est dit.

Bon OK, j'explique. CMake lit des fichiers `CMakeLists.txt` qui décrivent les projets à compiler dans un langage qui lui est propre. À partir de cela il génère des fichiers Makefile ou autres (des projets Xcode ou Visual Studio par exemple) qui permettent de construire le projet.

Ce qui m'a convaincu dans cet outil est qu'il est plutôt rapide (bien qu'il ne soit pas le plus rapide) et qu'il gère parfaitement les règles de reconstruction des cibles. Par exemple, si j'ajoute un paramètre de ligne de commande pour l'édition des liens, alors l'édition des liens va être refaite. Si je modifie un fichier `CMakeLists.txt` et que j'exécute `make` sans relancer CMake, alors CMake se relance tout seul (les Makefiles ont une règle de dépendance vers les `CMakeLists.txt`, pas con!) Je peux aussi simplement définir les répertoires d'entêtes, options de compilation et autres paramètres spécifiques à chaque projet, en précisant s'il le paramètre doit être visible des projets dépendants ou non.

L'outil est assez bien fichu et est très populaire dans le milieu du C++.

Un des points les plus souvent reprochés à CMake est son langage, notamment à l'époque de la version 2 de l'outil qui était excessivement verbeuse et en plus en ALL CAPS. On avait l'impression de se faire crier dessus à longueur de fichier. Aujourd'hui ces soucis sont résolus et le problème semble surtout être que cela fait un langage de plus à apprendre et qui ne sert à rien d'autre (contrairement à SCons et Premake par exemple). Perso je n'y vois pas de difficulté, c'est un bête langage à macros avec des mécanismes bien pratique pour nommer des groupes de paramètres.

Comme d'habitude la qualification « indépendants du compilateur et de la plate-forme » des fichiers de configuration est très discutable dans la mesure où il y a tout ce qu'il faut pour glisser des commandes système dans le build.

Les principaux problèmes que j'ai pu rencontrer avec une version récente de CMake concernent l'export de projet. En effet il y a une commande `install(EXPORTS)` qui permet de créer un fichier de configuration CMake pour inclure la cible en tant que dépendance dans un projet tiers. Malheureusement cette commande exporte par défaut les chemins absolus des dépendances de la cible et il faut bricoler pour exporter les dépendances proprement (en les enrobant dans des cibles importées par exemple).

Un autre souci est que CMake génère plein de fichiers intermédiaires et qu'avec la pratique généralisée de lancer le build à la racine du projet on se retrouve à polluer toute l'arborescence. Idéalement il faudrait que l'outil refuse de faire un build in-source.

Ninja

> *Ninja est un petit système de construction se concentrant sur la vitesse. Il est différent des autres systèmes de construction sur deux aspects majeurs : il est conçu pour que ses fichiers d'entrées soient générés par un système de construction de plus haut niveau, et il est aussi conçu pour exécuter les builds le plus rapidement possible.*

Ah cool, encore un outil qui se veut rapide, c'est exactement ce qu'il nous manquait. En plus quand on voit SCons et Premake qui se prétendent déjà les plus rapides, on a tout de suite confiance. Cela dit, contrairement à SCons et Premake, Ninja n'est pas un générateur de script de build. Il serait plutôt à comparer à Make.

Je n'ai jamais utilisé Ninja mais si jamais mes builds devenaient très lents je n'hésiterais pas à y jeter un coup d'œil. À moins que je ne passe à Meson.

Meson

> *Meson est un système open source de construction voulant être à la fois extrêmement rapide et, encore plus important, aussi accessible que possible.*

Bon là je désespère. Encore un outil qui veut être le plus rapide et toujours pas d'outil qui prétend fonctionner correctement.

Je n'ai jamais utilisé Meson mais on m'a dit que c'est-nouveau-c'est-bien-tu-devrais-essayer.

Pourquoi pas, enfin moi je cherche surtout un truc qui me génère de quoi faire un build iOS, Android, OSX et Linux. Un truc qui juste marche quoi.

FASTbuild

> *FASTBuild est un système de construction open source et de haute performance supportant une haute montée en charge de la compilation, la mise en cache et la distribution sur le réseau.*

Non mais franchement…

Là encore je n'ai pas utilisé cet outil. La promesse est sympa mais je ne vois pas trop l'intérêt par rapport aux deux mille autres outils du même genre.

Sharpmake

> *Sharpmake est un générateur de projets et solutions pour Visual Studio. Il est similaire à CMake et Premake, mais il est conçu pour être rapide et passer à l'échelle.*

Celui-ci est développé par Ubisoft initialement en interne et libéré en septembre 2017. Apparemment il n'y a plus d'activité dans le dépôt depuis octobre de la même année. Comme son nom l'indique, les scripts de build sont écrits en C#.

D'après la doc il sait aussi générer des projets pour Xcode et des Makefiles. J'ai longtemps considéré l'utiliser en remplaçant de Premake mais d'une part c'est écrit en C#, donc c'est mort pour l'utiliser sous Linux, et d'autre part je sens bien que tout ce qui n'est pas Windows et Visual Studio va être bancal.

Maven

> *Apache Maven un outil de compréhension et de gestion de projet logiciel. Basé sur le concept de modèle d'objet de projet, Maven peut gérer la construction, le compte-rendu et la documentation d'un projet depuis un élément central d'information.*

Je ne sais pas vous mais moi je comprends à peine le descriptif. C'est peut-être ma traduction qui est foireuse.

Maven est un outil du monde Java. J'ai pu l'utiliser un peu via des scripts déjà prêts et il n'y a pas grand-chose à lui reprocher de ce point de vue. Ça m'a l'air assez cool pour la gestion des dépendances.

C'est un outil qui a l'air très professionnel. Ah mais attend… c'est pour ça qu'on comprend rien au descriptif ! La première version date de 2004 et c'est donc tout naturellement que le langage le plus populaire du début du siècle a été choisi pour les scripts de build, je parle bien sûr du 🎉 XML 🎉.

Ant

> *Apache Ant est une bibliothèque Java et un outil en ligne de commande dons la mission est de piloter des processus décrits dans des fichiers de construction en tant que cibles et points d'extensions dépendant les uns des autres.*

Là encore ça sent le professionnalisme. Les fichiers Ant sont des sortes de Makefiles mais écrits en XML. Faut-il le préciser, Ant est lui aussi un outil du monde Java.

Gradle

> *Accélérez la productivité des développeurs*
> *Depuis les applications mobiles aux micro-services, des petites startups aux grandes entreprises, Gradle aide les équipes à construire, automatiser et livrer de meilleurs logiciels, plus rapidement.*

Je te laisse deviner à quel langage est destiné cet outil.

Gradle est l'outil de référence pour les builds des applications Android et c'est donc dans ce cadre que j'ai pu l'utiliser. Les scripts Gradle sont écrits en Groovy, un langage que je n'ai jamais utilisé par ailleurs. Perso je trouve pas ça génial mais c'est peut-être simplement parce ce que c'est loin de ce que l'on fait en C++.

Les trucs qui me fatiguent le plus avec Gradle sont d'abord le temps de compilation. La compilation du projet Java de nos jeux, une partie qui contient pourtant peu de code, prend quasiment une minute. L'autre souci est de trouver de la doc claire et facile à digérer. La doc officielle représente à peu près 24% du web[référence nécessaire], ce qui fait que la moindre interrogation demande déjà plusieurs heures de lectures, et les exemples de StackOverflow et divers blogs sont assez disparates.

Que choisir

Déjà douze outils et rien n'a l'air de dominer le marché :/ Sans doute faudrait-il inventer un nouvel outil pour les remplacer tous, quelque chose d'hyper rapide, évolutif, mais aussi adapté aux processes de déploiement intraprocéduraux sur concentrateur décentralisés pour les plus professionnels d'entre nous, avec des scripts dans un langage populaire, type GOTO++.

Malgré tout ce choix je n'ai rien trouvé qui résolve mon problème : générer un projet pour iOS, Android, Linux et OSX avec un seul script. Si vous avez une idée, ça m'intéresse.

FastBuild

Posté par **abriotde** le 15/06/18 à 02:42. Évalué à 4 (+3/-0).

> Très bon article. Certes, je pense que CMake domine largement et à juste titre
>
> Pour moi FastBuild n'est pas un concurrent direct. Il est destiné aux grandes entreprises avec un voire des serveurs dédiés à la gestion de code : CVS (type GIT), tests automatiques et compilation multiplateformes. Ce qu'apporte alors FastBuild ce n'est pas de générer des makefile, c'est de lancer les compilations en parallèle sur différents serveurs avec lancement des tests automatiques.
>
> Je sais que par exemple Google a système de ce type.

Xcode

Posté par **groumly** le 15/06/18 à 05:02. Évalué à 2 (+1/-1).

> *Pour compiler une application pour iOS il faut nécessairement un projet Xcode*

Ca simplifie très largement la tâche, mais Xcode se contente essentiellement de lancer des commandes à droite à gauche.
La compilation en soi est pas plus compliquée que pour des libs, les assets catalogs doivent pouvoir se gérer sans trop de problèmes. La ou ça se corse, comme on dit à Bastia, c'est pour construire le reste, à savoir les entitlements, les on demand assets, et ce genre de merdier. L'ipa se génère facilement (c'est juste un zip avec une structure de dossiers spécifiques), et de ce que je m'en rappelle, il est possible de re-signer un ipa avec codesign.

Après, oui, c'est clairement une énorme perte de temps, surtout que Xcode est plutôt pas mauvais du tout sur la gestion de projets. Et se scripte aussi très bien.

--
Linuxfr, le portail francais du logiciel libre et du neo nazisme.

Re: Xcode

Posté par **abriotde** le 15/06/18 à 08:11. Évalué à 1 (+0/-0).

Oui enfin comme dit dans l'article c'est la même chose que GNU Make pour Mac. Le but est là de chercher qqchose qui va nous épargner cette écriture fastidieuse et qui sera multiplateformes, pas la même étape...

Analyse très subjective!

Posté par **Ely** le 15/06/18 à 06:06. Évalué à 9 (+7/-0).

> *Je n'ai jamais utilisé Ninja mais si jamais mes builds devenaient très lents je n'hésiterais pas à y jeter un coup d'œil. À moins que je ne passe à Meson.*

Ninja n'est pas un remplacement de Meson ou de CMake. Par contre, ces deux derniers peuvent générer des fichiers ninja plutôt que des Makefile.

Tu aurais pu aussi parler des build system utilisés par chromium, GN et anciennement GYP (les deux génèrent des fichiers de build ninja).

En ce qui concerne les projets open-source C/C++, beaucoup effectuent la transition vers Meson+Ninja ces temps-ci.

Re: Analyse très subjective!

Posté par **EmmanuelP** le 15/06/18 à 08:09. Évalué à 3 (+3/-0).

Effectivement, c'est dommage de passer à côté de ninja et meson. Surtout pour un utilisateur de CMake en ce qui concerne ninja. Il suffit d'ajouter '-G Ninja' à l'appel de cmake pour profiter de la rapidité de ninja.

Au sujet de meson, les développeur ont pris pour principe de ne pas faire de meson un langage de programmation. La conséquence est de faciliter la prise en main, et de rendre plus clair les fichiers de configuration de compilation. Ajouté à ça, la documentation est complète et bien fichu.

Re: Analyse très subjective!

Posté par **Julien Jorge** (**page perso**) le 15/06/18 à 08:47. Évalué à 3 (+1/-0).

Alors suite à ton commentaire j'ai fait le test sur iscool::core et le gain n'est pas flagrant.

```
$ time make -j 4
real    0m29,490s
user    1m36,778s
sys 0m5,436s

$ time ninja -j 4
real    0m28,451s
user    1m39,555s
sys 0m4,701s
```

C'est peut-être un projet trop simple, pas assez gros. D'expérience le changement le plus efficace pour accélérer les builds a été de passer aux *single compilation units* (SCU), où on inclut plusieurs `.cpp` dans un seul pour les compiler en une seule fois et d'utiliser des instanciations explicites de templates. Après ça je n'ai pas vu de gain radical.

Quelqu'un a-t-il vu une comparaison de temps de build de projets utilisant des SCU avec Ninja, Make ou autre ? Il y a bien ce comparatif sur le site de Meson mais on ne parle pas de SCU ici.

Re: Analyse très subjective!

Posté par **serge_sans_paille** (**page perso**) le 15/06/18 à 09:07. Évalué à 4 (+2/-0).

C'est surtout lors des rebuilds que ninja brille (curieux pour une profession de l'ombre, n'est ce pas ?). Comme le dit cet article c'est surtout valable pour les softs de grosse taille, mais à ce niveau ça devient significatif.

Une présentation par le dev de meson qui complète bien ce journal.

Re: Analyse très subjective!

Posté par **pulkomandy** (**page perso**) le 15/06/18 à 10:07. Évalué à 5 (+3/-0).

Je plussoie complètement l'utilisation de ninja. Juste pour le plaisir de le lancer une deuxième fois quand le build est fini et de le voir dire *tout de suite* "nothing to do". Sur un projet comme WebKit, make dans la même situation mets chez moi presque une minute pour arriver à la même conclusion.

Autres avantages:
- Pas besoin de choisir un -j à la main comme avec make, ninja surveille le system load et s'ajuste tout seul (en tenant compte de la charge CPU effective, donc il lancera plus de jobs en parallèle si la compilation est limitée par les accès disques, et moins si elle est limitée par le CPU ou si on fait autre chose sur la machine pendant ce temps).
- Affichage correct de la sortie de la compilation, sans mélanger la sortie de différent threads
- Affichage du nombre de cibles à compiler, ce qui permet de savoir où on en est dans le build du projet

Re: Analyse très subjective!

Posté par **liberforce** (**page perso**) le 15/06/18 à 11:20. Évalué à 3 (+1/-0).

Les SCU sont appelées unity builds dans Meson.

Re: Analyse très subjective!

Posté par **liberforce** (**page perso**) le 15/06/18 à 12:19. Évalué à 4 (+2/-0). Dernière modification le 15/06/18 à 12:19.

Je te conseille aussi de lire les objectifs de Meson et ceux de ninja.

Meson ce n'est pas que les performances (pour beaucoup dues à ninja):
- lisibilité du code
- disparition des problèmes de quoting de CMake
- activation des warnings par défaut (parce que les débutants ne savent pas qu'il faut les activer, et ne le font donc pas, alors que ce sont ceux qui en ont le plus besoin)
- obligation de fait du build out-of-tree, parce que les développeurs peuvent casser le out-of tree en ne testant que le in-tree
- unique backend ninja: le choix a été fait de ne pas utiliser make, mais ninja qui est plus restreint et plus spécialisé. ninja ne gère que les dépendances, quitte à faire des fichiers verbeux, mais qui ne contiennent pas de logique à évaluer (pas de scripts bash, etc.). En contrepartie, on utilise un générateur de fichier de build ninja, comme Meson ou CMake.
- génération de fichiers pour Xcode ou Visual Studio
- gestion de la cross compilation
- utilisation massive de pkg-config pour la gestion de dépendances
- gestion de sous projets, ce qui permet de télécharger et builder GTK+ et toutes ses dépendances en une passe.
- gestion des SCU ou « unity builds »

Ensuite ce n'est pas une solution magique. Il y a encore des soucis, et j'avais entendu que la gestion d'Android n'était pas son fort, donc ce n'est peut être pas adapté à ton besoin. Mais cela a été assez pour convaincre de nombreux projets de l'utiliser (en migration ou compltément de leur ancien build system):
- GStreamer
- Tracker
- GNOME
- Shotwell
- Mesa

Mais aussi Xorg, libinput, et d'autres…

Et les outils spécialisés pour un langage spécifique

Posté par **Michaël** (**page perso**) le 15/06/18 à 07:17. Évalué à 4 (+2/-0).

Il y a aussi plein d'outils de build non génériques (du type qui compile tout ton projet pour le projet X avec un minimum de configuraiton mais qui est tellement chiant è étendre qu'on doit faire de efforts incroyables pour ajouter quoique ce soit à la procédure).

Pour les amateurs de Makefile il y a BSD Make (hachement mieux que GNU Make)et mes macros BSD Owl (projet hobby mais bien portable)

Les autotools sont surtout intéressants si on déploie sur des systèmes très hétérogènes – même si la chance de tomber sur un UNIX exotiques doit avoir sensiblement diminué dans les 20 dernières années. Ce qu'il faut comprendre avec les autotools est que GNU Make est surtout un backend pour automake – mais à programmer à la main, c'est infernal – et en ce qui concerne autoconf, que M4 est un langage assez bizarre (ça ressemble à TeX au niveau de l'évaluation) mais qu'on peut aussi écrire du shell script normal dans son fichier autoconf!

Meson et Ninja

Posté par **gnumdk** (**page perso**) le 15/06/18 à 08:35. Évalué à 4 (+2/-0).

Ici, j'utilise Meson et Ninja, pour moi Meson c'est la partie autotools et Ninja la partie Makefile, non?

Et franchement, c'est vraiment top.

build sous android

Posté par **serge_sans_paille** (**page perso**) le 15/06/18 à 08:59. Évalué à 3 (+1/-0).

> *Côté Android la compilation du code C++ se fait avec l'outil ndk-build, qui est en réalité une interface à GNU Make*

On notera que depuis quelques temps, ils offrent aussi une intégration à CMake.

Cmake non ?

Posté par **Frédéric Heulin** le 15/06/18 à 08:59. Évalué à 1 (+0/-0).

Il semble qu'il existe :
- un generator xcode pour cmake
- une intégration cmake prévue dans gradle.

Donc un moyen de gérer avec CMake en somme ?

Re: Cmake non ?

Posté par **Julien Jorge** (**page perso**) le 15/06/18 à 09:37. Évalué à 2 (+0/-0).

En dehors de l'excuse « pas-le-temps-après-tout-ça-marche » nous n'avons pas encore remplacé Premake par CMake car il y a des subtilités dans la génération du projet XCode dont j'ignore si elles sont gérables par CMake. Je parle de patches que nous avons dû appliquer à la génération du pbxproj dans Premake.

Tout d'abord il y a le fait que nous utilisons encore la sélection manuelle du provisioning profile. Il faut mettre la propriété `ProvisioningStyle = Manual` dans les `attributes.TargetAttributes` des *targets* dans le pbxproj.

Ensuite il y a les *capabilities* (*push notifications* et autres). Il faut les lister dans `SystemCapabilities`.

Nous avons aussi dû ajouter une option `SKIP_INSTALL` sur les libs pour éviter qu'elles soient embarquées dans l'ipa.

Enfin nous avons dû ajouter un moyen de marquer les frameworks comme optionnels afin de pouvoir lancer l'app sur de vieux appareils qui n'ont pas l'OS nécessaire pour la fonctionnalité (par exemple quand ReplayKit est arrivé nous voulions que l'application puisse se toujours se lancer sur les versions précédentes d'iOS. Nous testons alors à l'exécution si le framework est disponible).

Tous ces problèmes ont été trouvés au fil de l'eau et le système de plugins en Lua de Premake nous a permis de trouver des solutions assez rapidement. Je ne sais pas si cela aurait été aussi simple avec CMake.

Re: Cmake non ?

Posté par **pulkomandy** (**page perso**) le 15/06/18 à 10:10. Évalué à 3 (+1/-0).

Je précise que la communauté cmake (et son canal IRC) est pas mal réactive et disponible pour faire les modifications quand c'est nécessaire. Si tu n'es pas forcé de supporter de vieilles versions de cmake, tu pourras donc obtenir de l'aide pour régler ces problèmes, et s'il y a lieu, ce sera intégré dans cmake directement.

Je mentionne aussi au passage le très pratique cpack, qui permet de générer un .deb, un .rpm et/ou un installeur windows sans trop s'embêter. Mais peut être pas encore les .apk (ce n'est pas trop compliqué d'ajouter le support d'un nouveau format de distribution à cpack, cela dit…)

Make mon ami

Posté par **Sygne** (**page perso**) le 15/06/18 à 09:26. Évalué à 3 (+3/-2).

J'avoue que les deux critiques de make exposées dans le journal me laissent coi…

> *Il m'arrive encore de créer des petits Makefiles manuellement mais c'est vite pénible.*

C'est pas faux ! Mais d'un autre côté, je n'ai pas l'impression que l'on ait à modifier le makefile bien souvent, sinon pour ajouter une dépendance, ce qui prend trois secondes. Donc ce temps perdu peut vite ressembler à du temps gagné si, par ailleurs, (BSD)make fait le boulot.

> *Un autre point difficile à gérer concerne la dépendance au fichier Makefile lui-même.*

Trop gros, passera pas.

Mon expérience à deux balles

Posté par **Miguel Moquillon** (**page perso**) le 15/06/18 à 09:50. Évalué à 3 (+2/-0).

Ouep, l'enfer des outils de build s'ouvre dès que l'on veut construire un logiciel destiné à plusieurs plate-formes d'exécution. Il y a eu des langages mutli plate-formes qui ont été développées (Lisp, Smalltalk, Java, …) pour éviter, entre autre, ce genre de problèmes, mais ça n'a jamais vraiment marché parce qu'il y a toujours une plate-forme, le vent en poupe, qui ne joue pas leur jeu. Ou alors ça ne rempli pas les contraintes du logiciel (le plus souvent la performance) sur une ou plusieurs plate-formes cibles (mais ce critère relève peu ou prou aussi du premier, à savoir de jouer le jeu ; voir l'exemple raté des machines Lisp).

Dans mon cas, après avoir développé en C++ jusqu'en 2003 où on utilisait massivement des Makefiles, je me suis mis à Java jusqu'à encore aujourd'hui. Au début, on utilisait Ant (qui, comme tu l'as si bien écris, est du Makefile en XML). Maintenant on utilise Maven et bien que non exempte de défauts, je trouve que c'est un outil formidable surtout sur des gros, très gros projets ou lorsque les projets sont composés de plusieurs langages et ressources (Java, Groovy, Javascript, CSS, …). On utilise aussi du Gradle mais là c'est plus comme base pour concevoir des installateurs et configurateurs d'applications sur serveurs que comme système de build. A côté de ça je fais aussi un peu d'Haskell. Ce dernier a son propre outil de build (pardon ses propres outils de build) et ce n'est pas plus mal (j'adore slack). Les pires systèmes de build que j'ai pu utiliser restent quand même ceux du monde Javascript … Le meilleur que j'ai trouvé est celui du monde Smalltalk : il n'y en a pas ! Ou plus exactement, comme il ne s'appuie pas sur une approche de programmation et de construction orientée fichier, les problèmes adressés ne sont pas les mêmes et, au regard de ma piètre expérience avec, j'ai m'y suis pris beaucoup moins la tête. Mais je m'égare …

Pour répondre à ta question, dans mon ancienne boite, j'ai vu des équipes qui développaient des projets multi plate-formes (desktop, mobile Androïd, …) utiliser Maven comme outil de build centralisé. Par contre, ne m'étant jamais vraiment intéressé à ce problème, je ne leur ai jamais demandé leur retour là dessus. En tout cas, il semble qu'il existe des plugins Maven pour construire des projets iOS : j'ai trouvé ça https://github.com/letsdev/maven-ios-plugin

Re: Mon expérience à deux balles

Posté par **totof2000** le 15/06/18 à 12:54. Évalué à 2 (+0/-0).

Pour moi, le gros défaut der Maven, c'est le XML : ça rend le truc difficile à lire et à comprendre.

Re: Mon expérience à deux balles

Posté par **Yves** (**page perso**) le 15/06/18 à 13:57. Évalué à 1 (+1/-1).

Bon, je réponds ici, mais c'est une réponse générale, en fait… (rien de personnel)

Eh, les gars, faut dépoussiérer le cerveau, si la lecture d'un `pom.xml` vous incommode sous prétexte que c'est du XML !

C'est comme si vous organisiez un entretien pour un développeur full-stack multi-plateforme et tout, et là, le candidat vous sort « Alors, euh, en fait, euh, HTML je comprends pas trop, c'est trop dur à lire… ». Bonne change pour lui mettre un hexdump sous les yeux ou lui faire débuguer un problème d'encoding !

En revanche, oui, se plonger dans Maven n'est pas forcément simple, mais c'est le prix à payer pour un outil très complet et… professionnel en effet.

[Sorry for the rant… -->[]

Re: Mon expérience à deux balles

Posté par **barmic** le 15/06/18 à 14:12. Évalué à 1 (+0/-0). Dernière modification le 15/06/18 à 14:16.

> *Gradle est le plus intéressant des 3 a mon avis. Déjà il a abandonner le XML au profit du JSON.*

C'est du groovy pas du tout du json. Par contre il a un très gros défaut c'est que la syntaxe n'est pas complètement validée et on se retrouve avec des erreurs au runtime (runtime du build). C'est pour ça qu'ils regardent pour utilise kotlin à la place, mais ça fait un an et demi et c'est toujours pas vraiment la méthode conseillée.

qbs

Posté par **Fabien PRORIOL** le 15/06/18 à 10:07. Évalué à 1 (+0/-0).

Perso, j'utilise aussi CMake; parfois avec Ninja sur les gros projets…
Par contre, j'ai pas encore eu le temps de trop tester, mais je pense que QBS peut être un projet intéressant a surveiller…
Le but est de se passer de Make, QBS lance lui même GCC au lieu de CMake -> Makefile -> GCC
De plus, sa syntaxe déclarative a la QML devrait permettre une meilleur intégration dans les IDE, et cerise sur le gâteau, si on a besoin de faire des truc tordus, le javascript peut être utilisé directement dans le qbs…
Y'a une bonne équipe de développeur derrière (projet Qt, a la prétention de remplacer QMake/CMake pour Qt 6)

A suivre

Ant > Maven > Gradle

Posté par **Mimoza** le 15/06/18 à 10:39. Évalué à 3 (+1/-0).

Historiquement c'est dans cet ordre qu'il faut les présenter : Ant > Maven > Gradle

Ant, comme tu le dit, est une sorte de Makefiles en XML et très orienté pour les build Java, mais pas que, normalement. De base il n'a pas de gestion de dépendance, c'est le plug-in Ivy qui le fait pour lui. Un de ses gros défaut (outre la syntaxe XML) est que ses tâches/cibles sont très verbeuses a écrire et difficilement mutualisable/réutilisable. Il faut sans cesse réinventer la roue, mais permet de personnaliser finement ses script.

Maven est l'outil le plus populaire/répandu actuellement, surtout grâce a son écosystème très étendu. Tu as un plug-in pour quasiment tout ce que tu souhaite. Là encore les fichiers (pom.xml) sont en … XML. Il essaye de palier au défaut de Ant en ne permettant pas de scripter dans ses fichiers de description. Il faut obligatoirement passer par un plug-in, donc très grosse réutilisabilité du code … normalement. Sa gestion des dépendance est assez efficace et propose quelques outils pour résoudre les conflits

Gradle est le plus intéressant des 3 a mon avis. Déjà il a abandonner le XML au profit du JSON. Il ne profite malheureusement pas d'un écosystème aussi étendue que Maven, mais permet l'écriture de scipt en Groovy dans les fichiers projets en plus de l'utilisation de tâche/plug-in. Groovy n'est peut être pas le meilleur choix, mais vue que la population qui l'utilise est Javaïste en majorité le choix de ce langage de scripting est logique. Un peu le meilleur des 2 outils précédents a mon avis. Bien sûr gère les dépendance de manière assez efficace. Enfin point de vu perf il a un mode résident et compilation incrémentale des plus intéressant (https://guides.gradle.org/performance/).

maven et gradle

Posté par **barmic** le 15/06/18 à 10:42. Évalué à 3 (+2/-0).

Pour maven :

> *C'est un outil qui a l'air très professionnel. Ah mais attend… c'est pour ça qu'on comprend rien au descriptif ! La première version date de 2004 et c'est donc tout naturellement que le langage le plus populaire du début du siècle a été choisi pour les scripts de build, je parle bien sûr du 🎉 XML 🎉.*

C'est marrant de reprocher à tous les outils de juste chercher à être rapide plutôt que fonctionnelle et de se foutre de la gueule de celui qui est juste fonctionnel ;)

Pour gradle :

> *Les trucs qui me fatiguent le plus avec Gradle sont d'abord le temps de compilation. La compilation du projet Java de nos jeux, une partie qui contient pourtant peu de code, prend quasiment une minute.*

Je suis vraiment surpris, gradle est vraiment très efficace. Il faut par contre d'une part laisser le deamon se lancer et d'autre part ne pas utiliser de clean à tout bout de champ. Il prend très bien en compte les rebuilds (une étape de build est identifiée par les fichiers sources, la configuration de l'outil de build et éventuellement des éléments de ton système (l'architecture matériel par exemple).

Re: maven et gradle

Posté par **Julien Jorge** (**page perso**) le 15/06/18 à 11:17. Évalué à 4 (+2/-0).

Pour être tout à fait honnête je pense que le temps de build un peu long avec Gradle est surtout lié au fait que la cible soit Android. Il y a des étapes d'empaquetage (assets, apk) et de fusion de dex qui sont bien coûteuses et que l'on ne trouve pas dans un projet Java classique.

Re: maven et gradle

Posté par **SpaceFox** (**page perso**) le 15/06/18 à 11:26. Évalué à 2 (+0/-0).

Mon expérience confirme ça. J'ajoute qu'Android utilise assez massivement des étapes supplémentaires type génération de code ou obscurcissement/optimisation du code. Résultat : c'est plus long de compiler un projet gros-mais-raisonnable en Android que de générer un .war démesurément immense (parce qu'il embarque n'importe quoi) en Java EE, les deux projets étant sous Gradle – alors que le PC de développement Android est plus puissant.

Autre effet kiss cool : avec les étapes supplémentaires, un build Android d'un gros projet devient très rapidement extrêmement couteux en CPU et en RAM, puis en accès disque. D'ailleurs, un build Android sous Linux est bien plus rapide (facteur > 2 il y a environ deux ans sur un bon PC avec SSD) que le même sur le même matériel mais sous Windows – à tel point que la VM Linux dans un hôte Windows builde plus rapidement que Windows natif !

--

La connaissance libre : https://zestedesavoir.com

Build2

Posté par **rewind** le 15/06/18 à 11:05. Évalué à 3 (+1/-0).

Tu peux ajouter build2 à ta liste.

Une défense des autotools

Posté par **gouttegd** (**page perso**) le 15/06/18 à 13:59. Évalué à 2 (+0/-0).

Je sais que beaucoup de gens aiment bien dire pis que pendre des autootools, mais j’aimerais quand même les défendre un peu, à la fois du point de vue de *l’utilisateur* (ie, celui qui veut compiler et installer un programme utilisant les autotools) et du point de vue du *développeur* (celui qui utilise les autotools pour gérer son projet).

Du point de vue de l’utilisateur, d’abord un peu de contexte. J’utilise Slackware, qui comme vous le savez peut-être est fournie avec assez peu de logiciels, de sorte qu’il m’arrive fréquemment de compiler moi-même les programmes que j’utilise (certainement plus souvent qu’un utilisateur de Debian ou Fedora par exemple). À titre indicatif, sur les 1534 paquets installés présentement sur mon système, 387 (25%) sont étrangers à la distribution et ont été compilés par mes soins. Une conséquence de ça, c’est que je suis pas mal confronté aux systèmes de build (non pas que ça me fasse grand plaisir, mais quand on choisit Slackware on sait à quoi s’attendre).

Encore un peu de contexte, avec quelques chiffres : sur les 387 paquets que j’ai construit moi-même et que j’utilise actuellement (je ne compte pas les paquets que j’ai construit par le passé mais que je n’utilise plus), j’enlève 149 paquets pour des modules Python qui ne sont pas vraiment pertinents ici (ils se construisent tous de la même manière, `python setup.py build` — et bon sang comme c’est agréable). Reste 238 paquets non-Python qui se répartissent ainsi :

- 173 utilisant les autotools (73%) ;
- 22 utilisant un « simple » (ou moins simple) Makefile (9%) ;
- 19 utilisant CMake (8%) ;
- 9 utilisant Waf (4%) ;
- 9 utilisant QMake (4%) ;
- 3 utilisant un système maison (1%) ;
- 2 utilisant Meson (1%) ;
- 1 utilisant Ant (< 1%).

Et bien, pour moi, le bilan est sans appel : je préfère largement construire un paquet pour un projet utilisant les autotools. Presque tous les autres systèmes m’ont posé des casses-têtes que je n’ai jamais eu avec les autotools qui sont pourtant beaucoup plus fréquents.

Pêle-mêle, quelques trucs qui m’irritent presque chaque fois que je dois compiler un projet sans les autotools :

- Pas toujours d’option standard pour indiquer où placer les bibliothèques. J’ai toujours besoin d’une option de ce genre, parce que Slackware a choisi, pour sa version 64 bits, d’installer les bibliothèques dans `/usr/lib64` au lieu de `/usr/lib`. Oui, c’est un choix qui va à l’encontre de la plupart des autres distributions, mais il n’empêche : le système de build est supposé permettre de s’adapter au système cible. S’il ne le permet pas, c’est un système de build de merde, dont je maudis les auteurs sur cinq générations.

Mention très défavorable à CMake, qui est quand même là depuis suffisamment longtemps pour qu’on puisse se dire qu’une telle option devrait être disponible en standard, mais non : avec CMake, chaque projet fait sa propre sauce. Parfois il y a une option `LIB_SUFFIX`, parfois une option `INSTALL_LIB_DIR`, parfois une option `WANT_LIB64`, parfois une option `MULTIARCH_SUFFIX` — j’adore parcourir les CMakeLists.txt à la recherche de la bonne option, c’est ma joie —, et parfois… il n’y a pas d’option : le chemin d’installation des bibliothèques est codé en dur et il faut aller le modifier soi-même dans le CMakeLists.

- Pas toujours d’option standard pour indiquer où placer les pages de manuel ou d’info. Idem que ci-dessus. Avec les autools, c’est `--mandir` et `--infodir`. Avec CMake, parfois c’est `MANDIR`, le plus souvent c’est rien du tout, encore une fois il faut changer le chemin codé en dur dans le CMakeLists (ou aller déplacer les pages de manuel dans le bon dossier après installation).
- Plus généralement, le fait que tous ces projets ne sont pas fichus de s’arranger pour avoir une interface raisonnablement commune. Pensez ce que vous voulez des autotools, mais ils sont là depuis plus de vingt ans et les développeurs y sont habitués. Si vous voulez coder le nouveau build system du futur, au moins *essayez* de coller autant que possible à la syntaxe du classique `configure` (un bon point à Waf et Meson sur ce point — CMake, va te jeter dans les flammes de la Montagne du Destin où tu as été forgé, merci).
- Le simple fait de devoir installer *yet-another-build-system* juste pour pouvoir compiler un logiciel. Les autotools ont ceci de pratique qu’ils ne doivent être installés que sur la machine du développeur. Côté utilisateur, un shell (même pas besoin que ce soit GNU Bash) et un make (même pas besoin que ce soit GNU Make) suffisent. Mention spéciale aux projets qui ne se compilent qu’avec la dernière version de leur build-system, celle qui est sortie avant-hier.

Du point de vue du développeur maintenant : les autotools ne sont certes pas parfait (notamment, ils sont *lents*, c’est indéniable), mais il y a quand même quelques petites (ou pas si petites) choses que j’apprécie, comme :

> - le support de la compilation croisée, y compris vers Windows (`--build=x86_64-slackware-linux --host=i686-mingw32`) ;
> - `make distcheck`, pour non seulement générer une *tarball* mais vérifier automatiquement qu'elle est complète et autosuffisante ;
> - le support automatique de toutes les options qu'on attend d'un système de build conforme aux GNU Coding Standards (`--prefix`, `--libdir`, `--mandir`, plus généralement tous les `--machinsdir`, `--disable-static`, `--program-prefix`, etc.) ;
> - la prise en charge de gettext (alors là, je suis d'accord, c'est géré de façon *très moche* — mais au moins c'est géré, ce qui n'est pas le cas avec tous les autres systèmes…).
>
> Les autotools ont leur défaut, certes. Mais demandez-vous aussi pourquoi ils sont encore massivement utilisés.
>
> Pour ma part, franchement, je n'ai vu passer aucun système de build que je puisse considérer comme un successeur *crédible* aux autotools (non, CMake n'est pas un successeur crédible, voir ci-dessus), à part Meson récemment qui semble prometteur.

SBT

Posté par **flan** (**page perso**) le 15/06/18 à 14:03. Évalué à 2 (+0/-0).

> Il manque *Simple Build Tool* dans ce panorama, qui a été renommé en *Scala Build Tool*. Bon, ok, ce n'est pas forcément une mauvaise idée de l'oublier, après tout.
> J'ai eu le malheur d'utiliser SBT il y a quelques années. Bon, il fallait 2 min montre en main pour le lancer (je parle bien du temps de chargement de SBT avec un fichier .sbt de 20 ou 30 lignes, hein, pas de l'exécution elle-même), certes sur une machine de 2010.
> Il réussissait aussi à mourir par manque de RAM (seulement 4 Go de dispo, j'avoue) lors du packaging d'un projet de 10 Mo (concrètement, il s'agit juste de copier quelques fichiers dans un zip, rien de plus).
> Bien sûr, il avait aussi besoin de télécharger internet à chaque build (même s'il l'avait téléchargé 5 min avant). Pas pratique quand on n'a pas internet (ce qui est souvent mon cas)…
>
> Ah, et comme les développeurs se sont dit que les classiques points-virgules pour séparer les instructions coûtaient vraiment cher, ils se sont dit qu'il valait mieux les remplacer par une ligne vide, histoire de doubler l'espace vertical occupé à l'écran.
>
> Spark donne (donnait ?) un bon exemple de gestion de versions avec sbt : un fichier .sbt était fourni pour Scala 2.10 mais permettait également de compiler en 2.11, en cherchant d'abord tous les fichiers pom.xml dans le projet pour y appliquer un sed s/2.10/2.11/g (méthode fiable et propre s'il en est).

Note : les commentaires appartiennent à ceux qui les ont postés. Nous n'en sommes pas responsables.