SCons 2.5.1

Mode d'emploi

Steven Knight et l'équipe de développement SCons

Steven Knight

Version 2.5.1

Table des matières

Liste des exemples

Préface

Merci d'avoir pris le temps de lire SCons . SCons est un outil de construction de logiciels de nouvelle génération, ou faire outil - c'est un utilitaire logiciel pour la création de logiciels (ou d' autres fichiers) et de garder le logiciel construit à jour chaque fois que les fichiers d'entrée sous - jacentes changent.

La chose la plus distinctive à propos SCons est que ses fichiers de configuration sont en fait des *scripts* , écrits dans le Python langage de programmation. Ceci est en contraste avec la plupart des outils de construction alternatifs, qui inventent généralement une nouvelle langue pour configurer la construction. SCons a encore une courbe d'apprentissage, bien sûr, parce que vous devez savoir quelles fonctions appeler pour configurer votre build correctement, mais la syntaxe sous - jacente utilisée doit être familier à quiconque a déjà regardé un script Python.

Paradoxalement, en utilisant Python comme le format de fichier de configuration permet SCons *plus facile* pour les non-programmeurs d'apprendre que les langues cryptiques d'autres outils de construction, qui sont généralement inventées par les programmeurs pour d' autres programmeurs. Ceci est en grande partie en raison de la cohérence et la lisibilité qui sont caractéristiques de Python. Il se trouve que faire un vrai langage de script en direct la base pour les fichiers de configuration en fait un composant logiciel enfichable pour plus accompli programmeurs de faire des choses plus compliquées avec construit, si nécessaire.

1. SCons Principes

Il y a quelques grands principes que nous essayons de vivre jusqu'à la conception et la mise en œuvre SCons :

exactitude

> Tout d' abord et avant tout, par défaut, SCons garantit une construction correcte , même si cela signifie sacrifier la performance un peu. Nous nous efforçons de garantir la construction est correcte quelle que soit la façon dont le logiciel est en cours de construction structuré, la façon dont il peut avoir été écrit, ou comment les outils que la construction inhabituelle il.

Performance

> Étant donné que la construction est correcte, nous essayons de faire SCons construire le logiciel le plus rapidement possible. En particulier, chaque fois que nous avons besoin de ralentir la valeur par défaut de comportement pour garantir une construction correcte, nous essayons de le rendre facile à accélérer SCons grâce à des options d'optimisation qui vous permettent le commerce hors exactitude garantie dans tous les cas de fin pour une construction plus rapide dans les cas habituels.

Commodité

> SCons essaie de faire autant pour vous sortir de la boîte comme raisonnable, y compris la détection des bons outils sur votre système et de les utiliser correctement pour construire le logiciel.

En un mot, nous essayons de faire SCons juste « faire la bonne chose » et construire correctement le logiciel, avec un minimum de tracas.

2. A propos de caveat Exhaustivité de ce guide

Un mot d'avertissement que vous lisez ce guide: Comme trop des logiciels Open Source là, la SCons documentation ne sont pas toujours tenus à jour avec les fonctionnalités disponibles. Autrement dit, il y a beaucoup de choses que SCons peut faire est pas encore couvert dans le Guide de l'utilisateur. (Venez y penser, qui décrit aussi beaucoup de logiciels propriétaires, non?)

Bien que ce Guide de l' utilisateur ne soit pas aussi complète que nous aimerions qu'il soit, notre processus de développement ne met en vous assurant que la SCons page de manuel est maintenu avec de nouvelles fonctionnalités mises à jour. Donc , si vous essayez de comprendre comment faire quelque chose que SCons supports mais ne peut pas trouver assez (ou tout) d' informations ici, il serait utile de votre temps à regarder la page de manuel pour voir si l'information y est incluse. Et si vous le faites, peut - être vous auriez même envisager de contribuer une section au guide de sorte que la prochaine personne à la recherche de cette information de l'utilisateur ne sera pas passer par la même chose ...?

3. Remerciements

SCons n'existerait pas sans beaucoup d'aide de beaucoup de gens, dont beaucoup peuvent ne pas être même pas conscients qu'ils ont aidé ou servi d'inspiration. Ainsi , dans aucun ordre particulier, et au risque de laisser à quelqu'un:

Tout d'abord, SCons doit une énorme dette à Bob Sidebotham, l'auteur original du classique à base de Perl Cons outil Bob premier sorti au monde en arrière vers 1996. Le travail de Bob sur Cons classique fourni l'architecture sous - jacente et le modèle de spécification d' une construction configuration à l' aide d' un vrai langage de script. Mon expérience dans le monde réel travail sur Cons informé bon nombre des décisions de conception dans SCons, y compris le support de construction améliorée parallèle, la fabrication d' objets Builder facilement définissable par les utilisateurs, et à séparer le moteur de génération de l'interface d'emballage.

Greg Wilson a contribué à faire SCons a commencé comme un véritable projet quand il a lancé le logiciel concours de design de menuiserie en Février 2000. Sans cette nudge, mariant les avantages des inconvénients __gVirt_NP_NN_NNPS<__ une architecture classique avec la lisibilité de Python pourrait y avons séjourné plus d'une belle idée.

L'ensemble SCons équipe ont été absolument merveilleux de travailler avec, et SCons serait loin d'être aussi un outil utile sans l'énergie, les gens d'enthousiasme et de temps ont contribué au cours des dernières années. La "équipe de base" du Chad Austin, Anthony Roach, Bill Deegan, Charles Crain, Steve Leblanc, Greg Noel, Gary Oberbrunner, Greg Spencer et Christoph Wiedemann ont été super à propos de l' examen de mes (et d' autres) changements et attraper les problèmes avant qu'ils obtiennent dans le code de base. De la note technique: un travail exceptionnel et novateur de Anthony sur le moteur tasking a donné SCons un modèle de construction parallèle largement supérieure; Charles a été le maître de l'infrastructure essentielle du noeud; Le travail de Christoph sur l'infrastructure de configuration a ajouté fonctionnalités cruciales comme Autoconf; et Greg a fourni un excellent support pour Microsoft Visual Studio.

Un merci spécial à David Snopek pour contribuer son code « Autoscons » sous - jacent qui a constitué la base du travail de Christoph avec la fonctionnalité Configurer. David était extrêmement généreux à faire ce code à la disposition SCons , étant donné qu'il a publié il d' abord sous la licence GPL et SCons est publié sous une licence de type MIT moins restrictive.

Merci à Peter Miller pour son magnifique système de gestion du changement, Aegis , qui a fourni le SCons projet avec une méthodologie de développement solide dès le premier jour, et qui m'a montré comment on pouvait intégrer des tests de régression supplémentaires dans un cycle de développement pratique (années avant eXtreme Programming arrivée sur la scène).

Et enfin, grâce à Guido van Rossum pour son élégant langage de script, qui est à la base non seulement pour la SCons mise en œuvre, mais pour l'interface elle - même.

4. Contactez

La meilleure façon de contacter les personnes impliquées avec SCons, dont l'auteur, est à travers les listes de diffusion SCons.

Si vous souhaitez poser des questions générales sur la façon d'utiliser SCons envoyez un courriel à `scons-users@scons.org`.

Si vous souhaitez contacter la SCons communauté de développement directement, envoyez un courriel à `scons-dev@scons.org`.

Si vous souhaitez recevoir des annonces sur SCons , rejoindre le faible volume `announce@scons.tigris.org`liste de diffusion.

Chapitre 1. Installation et construction SCons

Ce chapitre vous guidera à travers les étapes de base de l' installation SCons sur votre système, et la construction SCons si vous ne disposez pas d'un package pré-construit disponible (ou simplement préférer la flexibilité de construire vous - même). Avant cela, cependant, ce chapitre décrit également les étapes de base nécessaires à l' installation Python sur votre système, en cas qui est nécessaire. Heureusement, les deux SCons et Python sont très faciles à installer sur presque tous les systèmes, et Python est déjà installé sur de nombreux systèmes.

1.1. Installation de Python

Parce que SCons est écrit en Python, vous devez évidemment avoir Python installé sur votre système pour utiliser SCons . Avant d'essayer d'installer Python, vous devriez vérifier si Python est déjà disponible sur votre système en tapant `python -V` (capitale « V ») ou `python --version` à l'invite de ligne de commande de votre système.

```
$ python -V
Python 2.5.1
```

Et sur un système Windows avec Python installé:

```
C: \>python -V
Python 2.5.1
```

Si Python est pas installé sur votre système, vous verrez un message d'erreur indiquant quelque chose comme « command not found » (sous UNIX ou Linux) ou « « python » est pas reconnu comme une commande interne ou externe, progam exécutable ou un fichier batch » (sous Windows). Dans ce cas, vous devez installer Python avant de pouvoir installer SCons .

L'emplacement standard pour les informations sur le téléchargement et l' installation de Python est http://www.python.org/download/ . Voir cette page pour plus d' informations sur la façon de télécharger et installer Python sur votre système.

SCons fonctionnera avec une version 2.x de Python de 2.7; 3.0 et versions ultérieures ne sont pas encore pris en charge. Si vous devez installer Python et le choix, nous vous recommandons d' utiliser la version la plus récente 2.x Python disponible. Les nouveaux Pythons ont d' importantes améliorations qui contribuent à accélérer la performance des SCons .

1.2. Installation SCons A partir de paquets pré-construits

SCons est livrée pré-emballés pour l' installation sur un certain nombre de systèmes, y compris les systèmes Linux et Windows. Vous n'avez pas besoin de lire toute cette section, vous avez besoin de lire uniquement la section appropriée du type de système que vous utilisez sur.

1.2.1. Installation SCons sur Red Hat (et autres basés sur RPM) Systèmes Linux

SCons est disponible en format RPM (Red Hat Package Manager), pré-construit et prêt à installer sur Red Hat Linux, Fedora, ou toute autre distribution Linux qui utilise RPM. Votre distribution peut déjà avoir un SCons construit RPM spécifiquement pour elle; beaucoup le font, y compris SUSE, Mandrake et Fedora. Vous pouvez vérifier la disponibilité d'un SCons RPM sur les serveurs de téléchargement de votre distribution, ou en consultant un site de recherche RPM commehttp://www.rpmfind.net/ ou http://rpm.pbone.net/ .

Si votre distribution prend en charge l' installation via yum , vous devriez être en mesure d'installer SCons en exécutant:

```
# yum install scons
```

Si votre distribution Linux ne possède pas déjà un spécifique SCons fichier RPM, vous pouvez télécharger et installer à partir du RPM générique fournie par le SCons projet. Cela installera le script SCons (s) `/usr/bin`et les modules de la bibliothèque SCons dans`/usr/lib/scons`.

Pour installer à partir de la ligne de commande, il suffit de télécharger le approprié `.rpm`fichier, puis exécutez:

```
# rpm -Uvh scons-2.5.1-1.noarch.rpm
```

Ou, vous pouvez utiliser un gestionnaire de paquets RPM graphique. Consultez la documentation de l'application de gestionnaire de paquets pour obtenir des instructions précises sur la façon de l'utiliser pour installer un RPM téléchargé.

1.2.2. Installation SCons sur Debian Linux Systems

Systèmes Debian Linux utilisent un autre format de gestion des paquets qui le rend également très facile à installer SCons .

Si votre système est connecté à Internet, vous pouvez installer le dernier paquet Debian officiel en cours d'exécution:

```
# apt-get install scons
```

1.2.3. Installation SCons sur les systèmes Windows

SCons fournit un programme d' installation Windows qui rend l' installation extrêmement facile. Téléchargez le `scons-2.5.1.win32.exe`fichier de la SCons page de téléchargement àhttp://www.scons.org/download.php . Ensuite , tout ce que vous devez faire est d' exécuter le fichier (généralement en cliquant sur son icône dans l' Explorateur Windows). Ceux - ci vous mènera à travers une petite séquence de fenêtres qui installeront SCons sur votre système.

1.3. Construction et l' installation SCons sur tout système

Si un pré-construit SCons package ne sont pas disponibles pour votre système, vous pouvez toujours construire et d' installer facilement SCons en utilisant Python natif `distutils`package.

La première étape consiste à télécharger soit le `scons-2.5.1.tar.gz` ou `scons-2.5.1.zip`, qui sont disponibles à partir de la page de téléchargement SCons à http://www.scons.org/download.html .

Décompressez l'archive que vous avez téléchargé, en utilisant un utilitaire comme le goudron sous Linux ou UNIX, ou WinZip sous Windows. Cela va créer un répertoire appelé `scons-2.5.1`, généralement dans votre répertoire local. Ensuite , changer votre répertoire de travail dans ce répertoire et installer SCons en exécutant les commandes suivantes:

```
# cd scons-2.5.1
#python setup.py install
```

Cela va construire SCons , installez le `scons`script dans le python qui est utilisé pour exécuter le répertoire des scripts de setup.py (`/usr/local/bin`ou `C:\Python25\Scripts`), et installera les SCons construire moteur dans le répertoire de la bibliothèque correspondant pour le python utilisé (`/usr/local/lib/scons`ou `C:\Python25\scons`). Privilèges à installer car ceux - ci sont des répertoires système, vous devrez peut - être root (sous Linux ou UNIX) ou Administrateur (sous Windows) SCons comme celui - ci.

1.3.1. Construction et l' installation de plusieurs versions de SCons Side-by-Side

Le SCons `setup.py` script quelques extensions qui prennent en charge une installation facile de plusieurs versions de SCons dans des endroits côte à côte. Cela rend plus facile de télécharger et d' expérimenter avec les différentes versions de SCons avant de passer votre processus de construction officiel à une nouvelle version, par exemple.

Pour installer SCons dans un emplacement spécifique à la version, ajoutez l' `--version-lib`option lorsque vous appelez `setup.py`:

```
# python setup.py install --version-lib
```

Cela installera les SCons construire moteur dans le `/usr/lib/scons-2.5.1` ou `C:\Python25\scons-2.5.1` répertoire, par exemple.

Si vous utilisez l' `--version-lib`option de la première fois que vous installez SCons , vous n'avez pas besoin de le préciser à chaque fois que vous installez une nouvelle version. Le SCons `setup.py` script détecte le nom du répertoire spécifique à la version (s) et supposons que vous voulez installer toutes les versions dans des répertoires spécifiques à la version. Vous pouvez remplacer cette hypothèse à l'avenir en spécifiant explicitement l' `--standalone-lib`option.

1.3.2. Installation SCons dans d' autres endroits

Vous pouvez installer SCons dans des endroits autres que la valeur par défaut en spécifiant l' `--prefix=`option de :

```
# python setup.py install --prefix=/opt/scons
```

Cela installerait le scons script `/opt/scons/bin` et le moteur de construction dans `/opt/scons/lib/scons`,

Notez que vous pouvez spécifier les deux `--prefix=` et les `--version-lib`options à la même type, auquel cas `setup.py` installera le moteur de construction dans un répertoire spécifique à la version par rapport au préfixe spécifié. Ajout `--version-lib`à l'exemple ci - dessus installer le moteur de construction dans `/opt/scons/lib/scons-2.5.1`.

1.3.3. Construction et l' installation SCons sans privilèges d' administration

Si vous ne disposez pas des privilèges droit d'installer SCons dans un emplacement du système, il suffit d' utiliser l' `--prefix=`option pour l'installer dans un emplacement de votre choix. Par exemple, pour installer SCons dans des endroits appropriés par rapport à l'utilisateur `$HOME`du répertoire, le `scons`script `$HOME/bin` et le moteur de construction dans `$HOME/lib/scons`, tapez simplement:

```
$ python setup.py install --prefix=$HOME
```

Vous pouvez, bien sûr, indiquer tout autre endroit que vous préférez, et peut utiliser l' `--version-lib`option si vous souhaitez installer des répertoires spécifiques à la version par rapport au préfixe spécifié.

Cela peut également être utilisé pour expérimenter une nouvelle version de SCons que celui installé dans vos emplacements du système. Bien sûr, l'emplacement dans lequel vous installez la version plus récente du `scons`scénario (`$HOME/bin`dans l'exemple ci - dessus) doit être configuré dans votre `PATH`variable avant le répertoire contenant la version du système installé du `scons`script.

Chapitre 2. Simple Builds

Dans ce chapitre, vous verrez plusieurs exemples de configurations de construction très simples à l' aide SCons , qui démontreront combien il est facile d'utiliser SCons pour construire des programmes de plusieurs langages de programmation sur différents types de systèmes.

2.1. Bâtiment simple C / C ++ Programmes

Voici le fameux « Bonjour, monde! » programme dans C:

```
int
principale()
{
    printf ( "Bonjour, monde! \ n");
}
```

Et voici comment le construire en utilisant SCons . Entrez ce qui suit dans un fichier nommé `SConstruct`:

```
Programme ( 'hello.c')
```

Ce fichier de configuration minimale donne SCons deux informations: ce que vous voulez construire (un programme exécutable) et le fichier d'entrée à partir de laquelle vous voulez construire (le `hello.c`fichier).`Program`est un *builder_method* , un appel Python qui indique SConsque vous voulez construire un programme exécutable.

C'est tout. Maintenant , exécutez la `scons`commande pour construire le programme. Sur un système compatible POSIX comme Linux ou UNIX, vous verrez quelque chose comme:

```
% scons
scons: Lecture des fichiers SConscript ...
scons: fait la lecture des fichiers SConscript.
scons: objectifs de construction ...
cc -o -c hello.o hello.c
cc -o bonjour hello.o
scons: fait des objectifs de construction.
```

Sur un système Windows avec le compilateur Visual C ++ .NET, vous verrez quelque chose comme:

```
C: \>scons
scons: Lecture des fichiers SConscript ...
scons: fait la lecture des fichiers SConscript.
scons: objectifs de construction ...
cl /Fohello.obj / c hello.c / nologo
lien / nologo /OUT:hello.exe hello.obj
embedManifestExeCheck (cible, source, env)
scons: fait des objectifs de construction.
```

Tout d' abord, notez que vous ne devez spécifier le nom du fichier source, et que SCons DEDUIT correctement les noms de l'objet et les fichiers exécutables à être construit à partir de la base du nom du fichier source.

Deuxièmement, notez que la même entrée `SConstruct`fichier, sans aucune modification, génère les noms de fichiers de sortie correctes sur les deux systèmes: `hello.o`et `hello` sur les systèmes POSIX,`hello.obj`et `hello.exe` sur les systèmes Windows. Ceci est un exemple simple de la façon dont SCons il est extrêmement facile d'écrire des logiciels portables construit.

(Notez que nous ne fournirons pas double côte à côte et la sortie de Windows POSIX pour tous les exemples présentés dans ce guide, il suffit de garder à l'esprit que, sauf indication contraire, l'un des exemples devrait fonctionner aussi bien sur les deux types de systèmes .)

2.2. Fichiers objet de construction

La `Program`méthode constructeur est seulement l' une des nombreuses méthodes de constructeur qui SCons fournit pour construire différents types de fichiers. Une autre est la `Object`méthode constructeur, qui indique SCons de construire un fichier objet à partir du fichier source spécifié:

```
Object ( 'hello.c')
```

Maintenant , lorsque vous exécutez la `scons`commande pour construire le programme, il va construire juste le `hello.o`fichier objet sur un système POSIX:

```
% scons
scons: Lecture des fichiers SConscript ...
scons: fait la lecture des fichiers SConscript.
scons: objectifs de construction ...
cc -o -c hello.o hello.c
scons: fait des objectifs de construction.
```

Et le `hello.obj`fichier objet sur un système Windows (avec le compilateur Microsoft Visual C ++):

```
C: \>scons
scons: Lecture des fichiers SConscript ...
scons: fait la lecture des fichiers SConscript.
scons: objectifs de construction ...
cl /Fohello.obj / c hello.c / nologo
scons: fait des objectifs de construction.
```

2.3. Simple builds Java

SCons fait également construire avec Java extrêmement facile. Contrairement aux `Program`et `Object`méthodes constructeur, cependant, la `Java`méthode constructeur exige que vous spécifiez le nom d'un répertoire de destination dans lequel vous voulez que les fichiers de classe placés, suivi du répertoire source dans lequel les `.java`fichiers en direct:

```
Java ( 'classes', 'src')
```

Si le `src`répertoire contient un seul `hello.java`fichier, puis la sortie de l' exécution de la `scons`commande ressemblerait à quelque chose comme ça (sur un système POSIX):

```
% scons
scons: Lecture des fichiers SConscript ...
scons: fait la lecture des fichiers SConscript.
scons: objectifs de construction ...
javac classes de -sourcepath src src / Hello.java
scons: fait des objectifs de construction.
```

Nous allons couvrir Java construit plus en détail, y compris la construction d' archive Java (`.jar`) et d' autres types de fichiers, dans le chapitre 26, *builds Java* .

2.4. Nettoyage après un contrat de construction

Lors de l' utilisation SCons , il est inutile d'ajouter des commandes spéciales ou les noms de cibles pour nettoyer après une accumulation. Au lieu de cela, vous utilisez simplement la `-c`ou `--clean` option lorsque vous invoquez SCons et SCons supprime les fichiers intégrés de appropriés. Donc , si nous construisons notre exemple ci - dessus, puis invoquons `scons -c` ensuite, la sortie sur Posix ressemble à :

```
% scons
scons: Lecture des fichiers SConscript ...
scons: fait la lecture des fichiers SConscript.
scons: objectifs de construction ...
cc -o -c hello.o hello.c
cc -o bonjour hello.o
scons: fait des objectifs de construction.
% scons -c
scons: Lecture des fichiers SConscript ...
scons: fait la lecture des fichiers SConscript.
scons: objectifs de nettoyage ...
Suppression hello.o
Suppression bonjour
scons: fait des cibles de nettoyage.
```

Et la sortie sur Windows ressemble à:

```
C: \>scons
scons: Lecture des fichiers SConscript ...
scons: fait la lecture des fichiers SConscript.
scons: objectifs de construction ...
cl /Fohello.obj / c hello.c / nologo
lien / nologo /OUT:hello.exe hello.obj
embedManifestExeCheck (cible, source, env)
scons: fait des objectifs de construction.
C: \>scons -c
scons: Lecture des fichiers SConscript ...
scons: fait la lecture des fichiers SConscript.
scons: objectifs de nettoyage ...
Suppression hello.obj
Suppression hello.exe
scons: fait des cibles de nettoyage.
```

Notez que SCons change de sortie pour vous dire qu'il est `Cleaning targets ...`et `done cleaning targets.`

2.5. le `SConstruct`fichier

Si vous avez l' habitude de construire des systèmes comme Assurez vous avez déjà compris que le `SConstruct`fichier est le SCons équivalent d'un `Makefile`. Autrement dit, le `SConstruct`fichier est le fichier d'entrée qui SCons lit pour contrôler la construction.

2.5.1. `SConstruct`Fichiers sont des scripts Python

Il y a, cependant, une différence importante entre un `SConstruct`fichier et un `Makefile`: le `SConstruct`fichier est en fait un script Python. Si vous n'êtes pas déjà familier avec Python, ne vous inquiétez pas. Ce Guide de l' utilisateur présente étape par étape à la quantité relativement faible de Python , vous aurez besoin de savoir pour être en mesure d'utiliser SCons efficacement. Et Python est très facile à apprendre.

Un aspect de l' utilisation de Python comme langage de script est que vous pouvez mettre dans votre commentaires `SConstruct`fichier en utilisant la convention de commentaires de Python; qui est, tout entre un « # » et la fin de la ligne sera ignorée:

```
# Arrangez-vous pour construire le programme « bonjour ».
Programme ( 'hello.c') # "hello.c" est le fichier source.
```

Vous verrez tout le reste de ce guide qui étant en mesure d'utiliser la puissance d'un véritable langage de script peut simplifier considérablement les solutions aux besoins complexes dans le monde réel construit.

2.5.2. SCons fonctions sont indépendantes de l' ordre

Un moyen important dans lequel le `SConstruct` fichier est pas exactement comme un script Python normal et est plus comme un `Makefile`, est que l'ordre dans lequel les SCons fonctions sont appelées dans le `SConstruct`fichier ne *pas* affecter l'ordre dans lequel SCons construit effectivement les programmes et objets les fichiers que vous voulez pour construire. [1] En d' autres termes, lorsque vous appelez le `Program`constructeur (ou toute autre méthode de constructeur), vous n'êtes pas dire SCons pour construire le programme à l'instant la méthode constructeur est appelé. Au lieu de cela, vous dire SCons pour construire le programme que vous souhaitez, par exemple, un programme construit à partir d' un fichier nommé `hello.c`, et il est à SCons pour construire ce programme (et tout autre fichier) chaque fois qu'il est nécessaire. (Nous allons en apprendre davantage sur la façon dont SCons décide lors de la construction ou la reconstruction d' un fichier est nécessaire dans le chapitre 6, *dépendances* , ci - dessous.)

SCons reflète cette distinction entre l' *appel d' une méthode de constructeur comme* `Program` et la *construction en fait le programme* en imprimant les messages d'état qui indiquent quand il est « en train de lire » le `SConstruct`fichier, et quand il fait construire les fichiers cibles. Est de préciser quand SCons exécute les instructions Python qui composent le `SConstruct`fichier, et quand SCons exécute réellement les commandes ou d' autres actions pour créer les fichiers nécessaires.

Clarifions cela par un exemple. Python a une `print`déclaration qui imprime une chaîne de caractères à l'écran. Si nous mettons des `print`déclarations autour de nos appels à la `Program`méthode constructeur:

```
print "Programme d'appel ( 'hello.c')"
Programme ( 'hello.c')
print "Programme d'appel ( 'goodbye.c')"
Programme ( 'goodbye.c')
imprimer « programme appelant fini () »
```

Puis , quand nous exécutons SCons , nous voyons la sortie des `print`déclarations entre les messages au sujet de la lecture des `SConscript`fichiers, ce qui indique que c'est lorsque les instructions Python sont en cours d' exécution:

```
% scons
scons: Lecture des fichiers SConscript ...
Programme d'appel ( 'hello.c')
Programme d'appel ( 'goodbye.c')
Programme d'appel terminé ()
scons: fait la lecture des fichiers SConscript.
scons: objectifs de construction ...
cc -o -c goodbye.o goodbye.c
cc -o au revoir goodbye.o
cc -o -c hello.o hello.c
cc -o bonjour hello.o
scons: fait des objectifs de construction.
```

Notez également que SCons a construit le `goodbye`premier programme, même si la « lecture `SConscript`» de sortie montre que nous avons appelé d' `Program('hello.c')` abord dans le `SConstruct`fichier.

2.6. Faire la SCons sortie Moins verbeux

Vous avez déjà vu comment SCons imprime des messages sur ce qu'il fait, entourant les commandes réelles utilisées pour construire le logiciel:

```
C: \>scons
scons: Lecture des fichiers SConscript ...
scons: fait la lecture des fichiers SConscript.
scons: objectifs de construction ...
```

```
cl /Fohello.obj / c hello.c / nologo
lien / nologo /OUT:hello.exe hello.obj
embedManifestExeCheck (cible, source, env)
scons: fait des objectifs de construction.
```

Ces messages mettent l' accent sur l'ordre dans lequel SCons fait son travail: tous les fichiers de configuration (génériquement appelés `SConscript`fichiers) sont lues et exécutées en premier lieu , et alors seulement sont les fichiers cibles construits. Entre autres avantages, ces messages permettent de distinguer entre les erreurs qui se produisent lorsque les fichiers de configuration sont lus et les erreurs qui se produisent lorsque les cibles sont en cours de construction.

Un inconvénient, bien sûr, est que ces messages encombrent la sortie. Heureusement, ils sont facilement désactivés en utilisant l' `-Q`option lors de l' invocation SCons :

```
C: \>scons -Q
cl /Fohello.obj / c hello.c / nologo
lien / nologo /OUT:hello.exe hello.obj
embedManifestExeCheck (cible, source, env)
```

Parce que nous voulons que ce Guide de l' utilisateur de se concentrer sur ce que SCons est en train de faire, nous allons utiliser l' `-Q`option pour supprimer ces messages de la sortie de tous les exemples de ce guide.

---

[1] Dans le langage de programmation, le `SConstruct`fichier est*déclaratif* , ce qui signifie que vous dites SCons ce que vous voulez faire et le laisser comprendre l'ordre dans lequel de le faire, plutôt que strictement *impératif* , vous permet de spécifier explicitement l'ordre dans lequel faire les choses.

Chapitre 3. Moins simples choses à faire avec Builds

Dans ce chapitre, vous verrez plusieurs exemples de configurations de construction très simples à l' aide SCons , qui démontreront combien il est facile d'utiliser SCons pour construire des programmes de plusieurs langages de programmation sur différents types de systèmes.

3.1. Spécification du nom du fichier cible (sortie)

Vous avez vu que lorsque vous appelez la `Program`méthode constructeur, il construit le programme résultant avec le même nom que le fichier source. Autrement dit, l'appel suivant pour construire un programme exécutable à partir du `hello.c`fichier source construira un programme exécutable nommé `hello`sur les systèmes POSIX, et un programme exécutable nommé `hello.exe`sur les systèmes Windows:

```
Programme ( 'hello.c')
```

Si vous voulez construire un programme avec un nom différent de celui de la base du nom du fichier source, vous mettez simplement le nom du fichier cible à la gauche du nom du fichier source:

```
Programme ( 'new_hello', 'hello.c')
```

(SCons nécessite le nom du fichier cible en premier lieu , suivi du nom du fichier source, de sorte que l'ordre d'imitateurs que l'instruction d'affectation dans la plupart des langages de programmation, y compris Python: `"program = source files"`.)

Maintenant SCons va construire un programme exécutable nommé `new_hello`lorsqu'il est exécuté sur un système POSIX:

```
% scons -Q
cc -o -c hello.o hello.c
cc -o new_hello hello.o
```

Et SCons construiront un programme exécutable nommé `new_hello.exe`lorsqu'il est exécuté sur un système Windows:

```
C: \>scons -Q
cl /Fohello.obj / c hello.c / nologo
lien / nologo /OUT:new_hello.exe hello.obj
embedManifestExeCheck (cible, source, env)
```

3.2. Plusieurs fichiers source Compiler

Vous venez de voir comment configurer SCons pour compiler un programme à partir d' un seul fichier source. Il est plus courant, bien sûr, que vous aurez besoin de construire un programme à partir de nombreux fichiers source d'entrée, pas seulement un. Pour ce faire, vous devez mettre les fichiers source dans une liste Python (entre crochets), comme suit:

```
Programme ([ 'prog.c', 'fichier1.c', 'fichier2.c'])
```

Une construction de l'exemple ci-dessus ressemblerait à ceci:

```
% scons -Q
cc -o -c file1.o file1.c
cc -o -c file2.o file2.c
cc -o -c prog.o prog.c
cc -o prog prog.o file1.o file2.o
```

Notez que SCons déduit le nom du programme de sortie du premier fichier source spécifié dans la liste - qui est, parce que le premier fichier source est `prog.c`, SCons sera Programmez un nom résultant `prog` (ou `prog.exe`sur un système Windows). Si vous souhaitez spécifier un autre nom de programme, puis (comme nous l' avons vu dans la section précédente) vous faites glisser la liste des fichiers source vers la droite pour faire de la place pour le nom du fichier du programme de sortie. (SCons met le nom du fichier de sortie à gauche des noms de fichier source afin que les imitateurs d'ordre que d'une instruction d'affectation. « Programme = fichiers source ») Cela rend notre exemple:

```
Programme ( 'programme', [ 'prog.c', 'fichier1.c', 'fichier2.c'])
```

Sous Linux, une version de cet exemple ressemblerait à ceci:

```
% scons -Q
cc -o -c file1.o file1.c
cc -o -c file2.o file2.c
cc -o -c prog.o prog.c
cc programme -o prog.o file1.o file2.o
```

Ou sous Windows:

```
C: \>scons -Q
cl /Fofile1.obj / c file1.c / nologo
cl /Fofile2.obj / c file2.c / nologo
cl /Foprog.obj / c prog.c / nologo
lien / nologo /OUT:program.exe prog.obj file1.obj file2.obj
embedManifestExeCheck (cible, source, env)
```

3.3. Faire une liste des fichiers avec`Glob`

Vous pouvez également utiliser la `Glob`fonction pour trouver tous les fichiers correspondant à un certain modèle, en utilisant le modèle standard du shell caractères correspondant `*`, `?` et `[abc]`pour correspondre à l' un `a`, `b`ou `c`. `[!abc]`est également pris en charge, pour correspondre à tout caractère *sauf* `a` , `b`ou `c`. Cela fait beaucoup de multi-sources fichier construit assez facile:

```
Programme ( 'programme', Glob ( '*. C'))
```

La page de manuel SCons a plus de détails sur l' utilisation `Glob` des répertoires de variantes (voir chapitre 16, la *variante Builds* , ci - dessous) et des dépôts (voir chapitre 22, *bâtiment du code Référentiels* , ci - dessous), à l' exception des fichiers et des chaînes de retour plutôt que des nœuds.

3.4. Spécification Vs. fichiers individuels Listes de fichiers

Nous vous avons maintenant montré deux façons de spécifier la source d'un programme, l'une avec une liste de fichiers:

```
Programme ( 'bonjour', [ 'fichier1.c', 'fichier2.c'])
```

Et l'un avec un seul fichier:

```
Programme ( 'bonjour', 'hello.c')
```

Vous pouvez effectivement mettre un nom de fichier dans une liste, aussi, que vous préférez peut-être juste pour un souci de cohérence:

```
Programme ( 'bonjour', [ 'hello.c'])
```

SCons fonctions acceptent un nom de fichier dans la forme. En fait, en interne, SCons traite toutes les entrées sous forme de listes de fichiers, mais vous permet d'omettre les crochets pour couper un peu sur le frappe quand il n'y a qu'un seul nom de fichier.

Important

Bien que SCons fonctions pardonnent si oui ou non vous utilisez une chaîne par rapport à une liste pour un seul nom de fichier, Python lui - même est plus stricte sur le traitement des listes et des chaînes différentes. Alors , où SCons permet soit une chaîne ou d'une liste:

```
# Ce qui suit correctement deux appels à la fois de travail:
Programme ( 'Program1', 'program1.c')
Programme ( 'program2', [ 'program2.c'])
```

Essayer de faire les choses « Python » que les chaînes de mix et des listes provoqueront des erreurs ou entraîner des résultats incorrects:

```
common_sources = [ 'file1.c', 'file2.c']

# VOICI INCORRECT ET CRÉE UNE ERREUR PYTHON
# CAR IL TENTE AJOUTER UN STRING À UNE LISTE:
Programme ( 'Program1', common_sources + 'program1.c')

# Ce qui suit fonctionne correctement, car il est l'ajout de deux
# les listes ensemble pour faire une autre liste.
Programme ( 'program2', common_sources + [ 'program2.c'])
```

3.5. Faire des listes de fichiers plus facile à lire

Un inconvénient de l'utilisation d'une liste Python pour les fichiers source est que chaque nom de fichier doit être entre guillemets (guillemets simples ou doubles). Cela peut devenir lourd et difficile à lire lorsque la liste des noms de fichiers est longue. Heureusement, SCons et Python fournissent un certain nombre de façons de vous assurer que le `SConstruct`fichier reste facile à lire.

Pour faire de longues listes de noms de fichiers plus facile à traiter, SCons fournit une `Split`fonction qui prend une liste cité des noms de fichiers, les noms séparés par des espaces ou d' autres caractères d'espace blanc, et il se transforme en une liste de noms de fichiers séparés. En utilisant la `Split`fonction transforme l'exemple précédent en:

```
Programme ( 'programme', Split ( 'main.c fichier1.c fichier2.c'))
```

(Si vous êtes déjà familier avec Python, vous aurez compris que ce qui est similaire à la `split()`méthode dans le standard de Python `string`module. Contrairement à la `split()`fonction membre de chaînes, cependant, la `Split`fonction ne nécessite pas une chaîne en entrée et se terminera un seul objet non-chaîne dans une liste, ou retourner son argument intacte si elle est déjà une liste. Cela est très pratique comme un moyen de vous assurer que les valeurs arbitraires peuvent être transmises à sCons fonctions sans avoir à vérifier le type de la variable à la main .)

Mettre l'appel à la `Split`fonction à l' intérieur de la `Program`communication peut aussi être un peu difficile à manier. Une alternative plus lisible est d'attribuer la sortie de l' `Split`appel à un nom de variable, puis utiliser la variable lorsque vous appelez la`Program`fonction:

```
src_files = split ( 'main.c fichier1.c fichier2.c')
Programme ( 'programme', src_files)
```

Enfin, la `Split`fonction ne se soucie pas de combien d' espace blanc sépare les noms de fichiers dans la chaîne entre guillemets. Cela vous permet de créer des listes de noms de fichiers sur plusieurs lignes, ce qui rend souvent plus facile pour l' édition:

```
src_files = de Split ( "" "main.c
                    fichier1.c
                    fichier2.c « » ")
Programme ( 'programme', src_files)
```

(Notez dans cet exemple que nous avons utilisé le Python « triple citation » syntaxe, ce qui permet une chaîne de contenir plusieurs lignes. Les trois citations peuvent être des guillemets simples ou doubles.)

3.6. Arguments de mots-clés

SCons vous permet également d'identifier les fichiers sources de fichiers de sortie et d' entrée en utilisant des arguments de mot - clé Python. Le fichier de sortie est connue comme la *cible* , et sont connus le fichier source (s) (logiquement) comme la *source* . La syntaxe Python pour cela est:

```
src_files = split ( 'main.c fichier1.c fichier2.c')
Programme (target = 'programme', source = src_files)
```

Parce que les mots-clés identifient explicitement ce que chaque argument est, vous pouvez réellement inverser l'ordre si vous préférez:

```
src_files = split ( 'main.c fichier1.c fichier2.c')
Programme (source = src_files, target = 'programme')
```

Que ce soit ou non vous choisissez d'utiliser des arguments de mots clés pour identifier les fichiers source et cible, et l'ordre dans lequel vous les spécifier lors de l' utilisation des mots - clés, sont des choix purement personnels; SCons fonctionne de la même quel que soit.

3.7. Compilent des programmes multiples

Pour compiler plusieurs programmes dans le même `SConstruct`fichier, il suffit d' appeler la `Program`méthode plusieurs fois, une fois pour chaque programme que vous devez construire:

```
Programme ( 'foo.c')
Programme ( 'bar', [ 'bar1.c', 'bar2.c'])
```

SCons alors construire les programmes comme suit:

```
% scons -Q
cc -o -c bar1.o bar1.c
cc -o -c bar2.o bar2.c
cc -o bar bar1.o bar2.o
cc -o -c foo.o foo.c
cc -o foo foo.o
```

Notez que SCons ne construit pas nécessairement les programmes dans le même ordre dans lequel vous les spécifiez dans le `SConstruct`fichier. SCons ne reconnaît cependant que les fichiers objets individuels doivent être construits avant que le programme résultant peut être construit. Nous en reparlerons plus en détail dans la section « Dépendances », ci - dessous.

3.8. Partage de fichiers source entre les programmes multiples

Il est courant de code de réutilisation par le partage de fichiers source entre plusieurs programmes. Une façon de le faire est de créer une bibliothèque à partir des fichiers source commune, qui peuvent ensuite être liés à des programmes qui en découlent. (Création de bibliothèques est abordée dans le chapitre 4, du *bâtiment et du chargement des bibliothèques* , ci - dessous.)

Un plus simple, mais peut-être moins pratique, moyen de partager des fichiers source entre plusieurs programmes est simplement d'inclure les fichiers communs dans les listes de fichiers source pour chaque programme:

```
Programme (Split ( 'foo.c common1.c common2.c'))
Programme ( 'bar', Split ( 'bar1.c bar2.c common1.c common2.c'))
```

SCons reconnaît que les fichiers d'objets pour les `common1.c`et `common2.c`fichiers source chaque besoin d'être construit qu'une seule fois, même si les fichiers objets résultants sont chacun liés à à la fois des programmes exécutables résultant:

```
% scons -Q
cc -o -c bar1.o bar1.c
cc -o -c bar2.o bar2.c
cc -o -c common1.o common1.c
cc -o -c common2.o common2.c
cc -o bar bar1.o bar2.o common1.o common2.o
cc -o -c foo.o foo.c
cc -o foo foo.o common1.o common2.o
```

Si deux ou plusieurs programmes partagent un grand nombre de fichiers source commune, en répétant les fichiers communs dans la liste pour chaque programme peut être un problème d'entretien lorsque vous devez modifier la liste des fichiers communs. Vous pouvez simplifier cela en créant une liste Python séparée pour contenir les noms de fichiers communs et concaténer avec d' autres listes en utilisant Python `+`opérateur:

```
commun = [ 'common1.c', 'common2.c']
foo_files = [ 'foo.c'] + commun
bar_files = [ 'bar1.c', 'bar2.c'] + commun
```

```
Programme ( 'foo', foo_files)
Programme ( 'bar', bar_files)
```

Ceci est fonctionnellement équivalent à l'exemple précédent.

Chapitre 4. Construction et Établir des liens avec les bibliothèques

Il est souvent utile d'organiser de grands projets logiciels en collectant des parties du logiciel dans une ou plusieurs bibliothèques. SCons il est facile de créer des bibliothèques et de les utiliser dans les programmes.

4.1. Bibliothèques de construction

Vous construisez vos propres bibliothèques en spécifiant au `Library`lieu de `Program`:

```
Bibliothèque ( 'foo', [ 'f1.c', 'f2.c', 'f3.c'])
```

SCons utilise le préfixe de la bibliothèque appropriée et suffixe pour votre système. Donc , sur les systèmes POSIX ou Linux, l'exemple ci - dessus construirait comme suit (bien que ranlib ne peuvent pas être appelés sur tous les systèmes):

```
% scons -Q
cc -o -c f1.o f1.c
cc -o -c f2.o f2.c
cc -o -c f3.o f3.c
ar rc libtruc.a f1.o f2.o f3.o
ranlib libtruc.a
```

Sur un système Windows, une version de l'exemple ci-dessus ressemblerait à ceci:

```
C: \>scons -Q
cl /Fof1.obj / c f1.c / nologo
cl /Fof2.obj / c f2.c / nologo
cl /Fof3.obj / c f3.c / nologo
lib / nologo /OUT:foo.lib f1.obj f2.obj f3.obj
```

Les règles pour le nom cible de la bibliothèque sont similaires à ceux des programmes: si vous ne spécifiez pas explicitement un nom de bibliothèque cible, SCons déduirez un à partir du nom du premier fichier source spécifié et SCons ajouterez un préfixe de fichier approprié et le suffixe si vous les lâchez.

4.1.1. La construction de bibliothèques à partir du code source ou objet Fichiers

L'exemple précédent montre la construction d' une bibliothèque à partir d' une liste de fichiers source. Vous pouvez, toutefois également les `Library`fichiers d'objet d'appel, et il correctement se rendre compte qu'ils sont des fichiers objet. En fait, vous pouvez arbitrairement mixer des fichiers de code source et les fichiers d'objet dans la liste des sources:

```
Bibliothèque ( 'foo', [ 'f1.c', 'f2.o', 'f3.c', 'f4.o'])
```

Et SCons se rend compte que seuls les fichiers de code source doivent être compilés dans des fichiers d'objet avant de créer la bibliothèque finale:

```
% scons -Q
cc -o -c f1.o f1.c
cc -o -c f3.o f3.c
ar rc libtruc.a f1.o f2.o f3.o f4.o
ranlib libtruc.a
```

Bien sûr, dans cet exemple, les fichiers objets doivent déjà exister pour la construction de réussir. Voir le chapitre 5, *objets Node* , ci - dessous, pour plus d' informations sur la façon dont vous pouvez créer des fichiers objet explicitement et inclure les fichiers construits dans une bibliothèque.

4.1.2. Construction de bibliothèques statiques Explicitement: le `StaticLibrary`constructeur

La `Library`fonction construit une bibliothèque statique traditionnelle. Si vous voulez être explicite sur le type de bibliothèque en cours de construction, vous pouvez utiliser le synonyme `StaticLibrary`fonction au lieu de `Library`:

```
StaticLibrary ( 'foo', [ 'f1.c', 'f2.c', 'f3.c'])
```

Il n'y a pas de différence fonctionnelle entre les `StaticLibrary`et `Library`fonctions.

4.1.3. Mitoyen (DLL) Bibliothèques: le `SharedLibrary`Builder

Si vous voulez construire une bibliothèque partagée (sur les systèmes POSIX) ou un fichier DLL (sur les systèmes Windows), utilisez la `SharedLibrary`fonction:

```
SharedLibrary ( 'foo', [ 'f1.c', 'f2.c', 'f3.c'])
```

La sortie sur POSIX:

```
% scons -Q
cc -o -c f1.os f1.c
cc -o -c f2.os f2.c
cc -o -c f3.os f3.c
cc -o libfoo.so -shared f1.os f2.os f3.os
```

Et la sortie sur Windows:

```
C: \>scons -Q
cl /Fof1.obj / c f1.c / nologo
cl /Fof2.obj / c f2.c / nologo
cl /Fof3.obj / c f3.c / nologo
```

```
lien / nologo / dll /out:foo.dll /implib:foo.lib f1.obj f2.obj f3.obj
RegServerFunc (cible, source, env)
embedManifestDllCheck (cible, source, env)
```

Notez encore que SCons prend soin de construire correctement le fichier de sortie, en ajoutant l' `-shared`option pour une compilation POSIX, et l' `/dll`option de Windows.

4.2. Établir des liens avec les bibliothèques

Habituellement, vous construisez une bibliothèque parce que vous voulez le lien avec un ou plusieurs programmes. Vous liez les bibliothèques avec un programme en spécifiant les bibliothèques dans la `$LIBS`variable de la construction, et en spécifiant le répertoire dans lequel la bibliothèque se trouve dans la `$LIBPATH`variable construction:

```
Bibliothèque ( 'foo', [ 'f1.c', 'f2.c', 'f3.c'])
Programme ( 'prog.c', LIBS = [ 'foo', 'bar'], LIBPATH = '')
```

Avis, bien sûr, que vous n'avez pas besoin de spécifier un préfixe de la bibliothèque (comme `lib`) ou un suffixe (comme `.a`ou `.lib`). SCons utilise le préfixe ou un suffixe correct pour le système actuel.

Sur un système POSIX ou Linux, une version de l'exemple ci-dessus ressemblerait à ceci:

```
% scons -Q
cc -o -c f1.o f1.c
cc -o -c f2.o f2.c
cc -o -c f3.o f3.c
ar rc libtruc.a f1.o f2.o f3.o
ranlib libtruc.a
cc -o -c prog.o prog.c
cc -o prog prog.o -L. -lfoo -lbar
```

Sur un système Windows, une version de l'exemple ci-dessus ressemblerait à ceci:

```
C: \>scons -Q
cl /Fof1.obj / c f1.c / nologo
cl /Fof2.obj / c f2.c / nologo
cl /Fof3.obj / c f3.c / nologo
lib / nologo /OUT:foo.lib f1.obj f2.obj f3.obj
cl /Foprog.obj / c prog.c / nologo
lien / nologo /OUT:prog.exe / LIBPATH :. foo.lib bar.lib prog.obj
embedManifestExeCheck (cible, source, env)
```

Comme d' habitude, notez que SCons a pris en charge la construction des lignes de commande correctes pour créer un lien avec la bibliothèque spécifiée sur chaque système.

Notez également que, si vous avez une seule bibliothèque pour créer un lien avec, vous pouvez spécifier le nom de la bibliothèque dans la chaîne unique, au lieu d'une liste de Python, de sorte que:

```
Programme ( 'prog.c', LIBS = 'foo', LIBPATH = '')
```

est équivalent à:

```
Programme ( 'prog.c', LIBS = [ 'foo'], LIBPATH = '')
```

Ceci est similaire à la façon dont SCons gère une chaîne ou une liste pour spécifier un fichier source unique.

4.3. Les bibliothèques de recherche: la `$LIBPATH`variable de la construction

Par défaut, l'éditeur de liens ne regarder dans certains répertoires définis par le système pour les bibliothèques. SCons sait comment rechercher des bibliothèques dans les répertoires que vous spécifiez avec la `$LIBPATH`variable de construction. `$LIBPATH`se compose d'une liste de noms de répertoires, comme ceci:

```
Programme ( 'prog.c', LIBS = 'm',
                 LIBPATH = [ '/ usr / lib', '/ usr / local / lib'])
```

En utilisant une liste Python est préféré car il est des systèmes à travers portables. Sinon, vous pouvez mettre tous les noms de répertoire dans une seule chaîne, séparés par le caractère de séparation de chemin spécifique au système: deux points sur les systèmes POSIX:

```
LIBPATH = '/ usr / lib: / usr / local / lib'
```

ou un point-virgule sur les systèmes Windows:

```
LIBPATH = 'C: \\ lib; D: \\ lib'
```

(Notez que Python exige que les séparateurs de barre oblique inverse dans un nom de chemin Windows sont échappés dans les chaînes.)

Lorsque l'éditeur de liens est exécuté, SCons va créer des drapeaux appropriés pour que l'éditeur de liens rechercher des bibliothèques dans les mêmes répertoires que SCons . Donc , sur un système Linux ou Posix, une version de l'exemple ci - dessus ressemblerait à ceci:

```
% scons -Q
cc -o -c prog.o prog.c
cc -o prog prog.o -L / usr / lib -L / usr / local / lib -lm
```

Sur un système Windows, une version de l'exemple ci-dessus ressemblerait à ceci:

```
C: \>scons -Q
cl /Foprog.obj / c prog.c / nologo
```

```
lien / nologo /OUT:prog.exe / LIBPATH: \ usr \ lib / LIBPATH: \ usr \ local \ lib m.lib prog.obj
embedManifestExeCheck (cible, source, env)
```

Notez encore que SCons a pris soin des détails spécifiques au système de création des options de ligne de commande à droite.

Chapitre 5. Les objets Node

En interne, SCons représente tous les fichiers et les répertoires qu'il connaît à peu près aussi `Nodes`. Ces objets internes (non objet *fichiers*) peuvent être utilisés dans une variété de façons de rendre vos `SConscript` fichiers portable et facile à lire.

5.1. Builder méthodes Retour Listes de noeuds cibles

Toutes les méthodes de constructeur renvoient une liste des `Node`objets qui identifient le fichier cible ou des fichiers qui sera construit. Ces retour `Nodes`peuvent être passés comme arguments à d' autres méthodes de constructeur.

Par exemple, supposons que nous voulons construire les deux fichiers objets qui composent un programme avec différentes options. Cela signifierait appeler le `Object` constructeur une fois pour chaque fichier objet, en spécifiant les options souhaitées:

```
Object ( 'hello.c', CCFLAGS = '- DHELLO')
Object ( 'goodbye.c', CCFLAGS = '- DGOODBYE')
```

Une façon de combiner ces fichiers objet dans le programme résultant serait d'appeler le `Program` constructeur avec les noms des fichiers objets répertoriés comme sources:

```
Object ( 'hello.c', CCFLAGS = '- DHELLO')
Object ( 'goodbye.c', CCFLAGS = '- DGOODBYE')
Programme ([ 'hello.o', 'goodbye.o'])
```

Le problème avec la spécification des noms comme des chaînes est que notre `SConstruct`fichier n'est plus à travers les systèmes d'exploitation portables. Il ne sera pas, par exemple, travailler sur Windows parce que l'objet de fichiers , il serait nommé `hello.obj`et `goodbye.obj`, non `hello.o`et `goodbye.o`.

Une meilleure solution consiste à attribuer les listes de cibles retournées par les appels au `Object`constructeur aux variables que nous pouvons concaténer dans notre appel au `Program`constructeur:

```
hello_list = Object ( 'hello.c', CCFLAGS = '- DHELLO')
goodbye_list = Object ( 'goodbye.c', CCFLAGS = '- DGOODBYE')
Programme (hello_list + goodbye_list)
```

Cela rend notre `SConstruct`nouveau portable de fichiers, la sortie de génération sur Linux ressembler à :

```
% scons -Q
cc -o -c goodbye.o -DGOODBYE goodbye.c
cc -o -c hello.o -DHELLO hello.c
cc -o bonjour hello.o goodbye.o
```

Et sous Windows:

```
C: \>scons -Q
cl /Fogoodbye.obj / c goodbye.c -DGOODBYE
cl /Fohello.obj / c hello.c -DHELLO
lien / nologo /OUT:hello.exe hello.obj goodbye.obj
embedManifestExeCheck (cible, source, env)
```

Nous verrons des exemples d'utilisation de la liste des noeuds retournés par les méthodes de constructeur dans le reste de ce guide.

5.2. Création de fichiers et Explicitement nœuds Directory

Il convient de mentionner ici que SCons maintient une distinction claire entre les noeuds qui représentent les fichiers et les noeuds qui représentent des répertoires. SCons soutient `File`et `Dir` fonctions, respectivement, renvoient un fichier ou un répertoire Noeud:

```
hello_c = fichier ( 'hello.c')
Programme (hello_c)

Cours = Dir ( 'classes')
Java (classes 'src')
```

Normalement, vous n'avez pas besoin d'appeler `File`ou `Dir`directement, parce que l' appel d' une méthode de constructeur traite automatiquement les chaînes comme les noms des fichiers ou des répertoires, et les convertit dans les objets Node pour vous. Les `File`et les `Dir`fonctions peuvent être utiles dans des situations où vous devez indiquer explicitement SCons sur le type de noeud étant passé à un constructeur ou une autre fonction, ou se référer clairement à un fichier spécifique dans une arborescence de répertoires.

Il y a aussi des moments où vous pourriez avoir besoin de se référer à une entrée dans un système de fichiers sans le savoir à l' avance que ce soit un fichier ou un répertoire. Pour ces situations, SCons prend également en charge une `Entry`fonction qui renvoie un nœud qui peut représenter un fichier ou un répertoire.

```
xyzzy = Entrée ( 'xyzzy')
```

Le retour `xyzzy`Node sera transformé en un fichier ou un répertoire nœud la première fois qu'il est utilisé par une méthode de constructeur ou une autre fonction qui nécessite un par rapport à l'autre.

5.3. Impression des `Node`noms de fichier

Une des choses les plus courantes que vous pouvez faire avec un nœud est l' utiliser pour imprimer le nom du fichier que le nœud représente. Gardez à l' esprit, cependant, parce que l'objet retourné par un appel constructeur est une *liste* de nœuds, vous devez utiliser Python pour chercher différents indices des nœuds de la liste. Par exemple, le suivant `SConstruct`fichier:

```
object_list = Objet ( 'hello.c')
program_list = Programme (object_list)
print "Le fichier objet est:", object_list [0]
print "Le fichier du programme est le suivant:", program_list [0]
```

Imprimerait les noms de fichiers suivants sur un système POSIX:

```
% scons -Q
Le fichier objet est: hello.o
Le fichier du programme est: bonjour
cc -o -c hello.o hello.c
cc -o bonjour hello.o
```

Et les noms de fichiers suivants sur un système Windows:

```
C: \>scons -Q
Le fichier objet est: hello.obj
Le fichier du programme est: hello.exe
cl /Fohello.obj / c hello.c / nologo
lien / nologo /OUT:hello.exe hello.obj
embedManifestExeCheck (cible, source, env)
```

Notez que dans l'exemple ci - dessus, les `object_list[0]` extraits d' un véritable nœud *objet* de la liste, et le Python `print`déclaration convertit l'objet à une chaîne pour l' impression.

5.4. L' utilisation d' un `Node`« s Nom de fichier en tant que chaîne

Impression d' un `Node`« nom de s comme décrit dans la section précédente fonctionne parce que la représentation de chaîne d'un `Node`objet est le nom du fichier. Si vous voulez faire autre chose que d' imprimer le nom du fichier, vous pouvez récupérer en utilisant Python builtin `str`fonction. Par exemple, si vous souhaitez utiliser le Python`os.path.exists` pour savoir si un fichier existe alors que le `SConstruct`fichier est en cours de lecture et exécuté, vous pouvez chercher la chaîne comme suit:

```
importation os.path
program_list = Programme ( 'hello.c')
program_name = str (program_list [0])
sinon os.path.exists (nom_programme):
    impression program_name, «n'existe pas! »
```

Ce qui est exécuté comme suit sur un système POSIX:

```
% scons -Q
bonjour n'existe pas!
cc -o -c hello.o hello.c
cc -o bonjour hello.o
```

5.5. `GetBuildPath`: Obtenir le chemin à partir d' une `Node`ou chaîne

`env.GetBuildPath(file_or_list)` retourne le chemin d'un `Node`ou une chaîne représentant un chemin d' accès. Il peut également prendre une liste de `Node`s et / ou des chaînes, et renvoie la liste des chemins. Si elle est adoptée une seule `Node`, le résultat est le même que l' appel `str(node)`(voir ci - dessus). La chaîne (s) peut avoir intégré les variables de construction, qui sont développées comme d' habitude, en utilisant l'ensemble des variables de l'environnement appelant. Les chemins peuvent être des fichiers ou des répertoires, et ne pas exister.

```
env = environnement (VAR = "valeur")
n = fichier ( "foo.c")
impression env.GetBuildPath ([n, "sous / dir / $ VAR"])
```

Imprimerait les noms de fichiers suivants:

```
% scons -Q
[ 'Foo.c', 'sous / dir / valeur']
scons: ` « . est à jour.
```

Il existe également une version de fonction `GetBuildPath`qui peut être appelée sans `Environment`; qui utilise les SCons par défaut`Environment`pour effectuer la substitution sur les arguments de chaîne.

Chapitre 6. Dépendances

Jusqu'à présent , nous avons vu comment SCons poignées une seule fois construit. Mais l' une des principales fonctions d'un outil de construction comme SCons est de reconstruire uniquement ce qui est nécessaire lorsque les fichiers sources changent - ou, en d' autres termes, SCons devraient *pas* perdre de temps la reconstruction des choses qui ne ont pas besoin d'être reconstruit. Vous pouvez le voir à l' œuvre simplement en re-invoquer SCons après avoir construit notre simple , par `hello`exemple:

```
% scons -Q
cc -o -c hello.o hello.c
cc -o bonjour hello.o
% scons -Q
scons: ` « . est à jour.
```

La deuxième fois qu'il est exécuté, SCons se rend compte que le `hello`programme est mis à jour par rapport au courant `hello.c`fichier source, et évite le reconstruire. Vous pouvez voir plus clairement en nommant le `hello`programme explicitement sur la ligne de commande:

```
% scons -Q hello
cc -o -c hello.o hello.c
cc -o bonjour hello.o
```

```
% scons -Q hello
scons: `bonjour » est à jour.
```

Notez que SCons rapports `"...is up to date"` uniquement pour les fichiers cibles nommés explicitement sur la ligne de commande, afin d' éviter d' encombrer la sortie.

6.1. Décider Quand un fichier d'entrée a changé: la `Decider`fonction

Un autre aspect d'éviter les reconstructions inutiles est le comportement de l' outil de construction fondamental de la *reconstruction* des choses quand un changement de fichier d'entrée, de sorte que le logiciel intégré est à jour. Par défaut, SCons garde la trace de ce à travers un MD5 `signature`, ou la somme de contrôle, du contenu de chaque fichier, mais vous pouvez facilement configurerSCons à utiliser les temps de modification (ou horodatage) au lieu. Vous pouvez même spécifier votre propre fonction Python pour décider si un fichier d'entrée a changé.

6.1.1. En utilisant les signatures MD5 pour décider si un fichier a changé

Par défaut, SCons conserve la trace de savoir si un fichier a changé à partir d'une somme de contrôle MD5 du contenu du fichier, pas le temps de modification du fichier. Cela signifie que vous pourriez être surpris par le défaut SCons comportement si vous êtes habitué à laMarque convention de forcer la mise à jour d' une reconstruction en temps de modification du fichier (en utilisant la touche de commande, par exemple):

```
% scons -Q hello
cc -o -c hello.o hello.c
cc -o bonjour hello.o
% touch hello.c
%scons -Q hello
scons: `bonjour » est à jour.
```

Même si a changé, le temps de modification du fichier SCons se rend compte que le contenu du `hello.c`fichier ont *pas* changé, et donc que le `hello`ne doivent pas être reconstruit programme. Cela permet d' éviter les reconstructions inutiles lorsque, par exemple, quelqu'un réécrit le contenu d'un fichier sans faire un changement. Mais si le contenu du fichier ne change vraiment, alors SCons détecte le changement et reconstruit le programme selon les besoins:

```
% scons -Q hello
cc -o -c hello.o hello.c
cc -o bonjour hello.o
% [CHANGEMENT DU CONTENU DE hello.c]
% scons -Q hello
cc -o -c hello.o hello.c
cc -o bonjour hello.o
```

Notez que vous pouvez, si vous le souhaitez, spécifiez ce comportement par défaut (signatures MD5) en utilisant explicitement la `Decider`fonction comme suit:

```
Programme ( 'hello.c')
Decider ( 'MD5')
```

Vous pouvez également utiliser la chaîne `'content'` comme synonyme de `'MD5'` l'appel de la `Decider`fonction.

6.1.1.1. Ramifications de l'utilisation MD5 Signatures

En utilisant les signatures MD5 pour décider si un fichier d'entrée a changé a un avantage surprenant: si un fichier source a été modifiée de telle sorte que le contenu du fichier cible Reconstruit (s) seront exactement les mêmes que la dernière fois que le fichier a été construit, puis tous les fichiers cibles « en aval » qui dépendent du fichier cible reconstruit mais non-changé en réalité ne doivent pas être reconstruites.

Donc , si, par exemple, un utilisateur devait changer seulement un commentaire dans un `hello.c`fichier, puis le reconstruit `hello.o`fichier serait exactement le même que celui construit précédemment (en supposant que le compilateur ne met pas d'informations spécifiques à la construction dans le fichier d'objet) . SCons serait alors se rendre compte qu'il ne serait pas nécessaire de reconstruire le `hello`programme comme suit:

```
% scons -Q hello
cc -o -c hello.o hello.c
cc -o bonjour hello.o
% [CHANGEMENT DANS UN COMMENTAIRE hello.c]
% scons -Q hello
cc -o -c hello.o hello.c
scons: `bonjour » est à jour.
```

En substance, SCons « courts-circuits » tout dépend builds quand il se rend compte qu'un fichier cible a été reconstruit exactement le même fichier que la dernière construction. Cela prend un certain temps de traitement supplémentaire pour lire le contenu de la cible (`hello.o`fichier), mais enregistre souvent du temps lorsque la recréation a été évité aurait été beaucoup de temps et coûteux.

6.1.2. L'utilisation Timbres temps de décider si un fichier a changé

Si vous préférez, vous pouvez configurer SCons utiliser le temps de modification d'un fichier, pas le contenu du fichier, au moment de décider si une cible doit être reconstruit. SCons vous donne deux façons d'utiliser les timbres de temps pour décider si un fichier d'entrée a changé depuis la dernière fois une cible a été construit.

La façon la plus familière à utiliser l' horodatage est le moyen Make fait: est - ce que, ont SCons décident qu'une cible doit être reconstruit si le temps de modification d'un fichier source est *plus récente* que le fichier cible. Pour ce faire, appelez la `Decider` fonction comme suit:

```
Object ( 'hello.c')
Décideur ( 'horodatage-nouvelle)
```

Cela rend SCons agissent comme Marque lorsque le temps de modification d'un fichier est mis à jour (en utilisant la touche de commande, par exemple):

```
% scons -Q hello.o
cc -o -c hello.o hello.c
% touch hello.c
%scons -Q hello.o
cc -o -c hello.o hello.c
```

Et, en fait, parce que ce comportement est le même que le comportement de Make , vous pouvez également utiliser la chaîne `'make'` comme synonyme de `'timestamp-newer'` l'appel de la `Decider`fonction:

```
Object ( 'hello.c')
Décideur ( « faire »)
```

Un inconvénient à l' utilisation fois timbres exactement comme Make est que si un temps de modification du fichier d'entrée devient soudainement *plus* d'un fichier cible, le fichier cible ne sera pas reconstruit. Cela peut se produire si une ancienne copie d'un fichier source est restauré à partir d' une archive de sauvegarde, par exemple. Le contenu du fichier restauré sera probablement différent de celui qu'ils étaient la dernière fois une cible dépendante a été construit, mais l'objectif ne sera pas reconstruit parce que le temps de modification du fichier source n'est pas plus récente que la cible.

Parce que SCons stocke effectivement des informations sur les horodateurs des fichiers source chaque fois qu'une cible est construit, il peut gérer cette situation en vérifiant une correspondance exacte de l'horodatage du fichier source, au lieu de simplement si le fichier source ou non est plus récente que la cible fichier. Pour ce faire, spécifiez l'argument `'timestamp-match'` lors de l' appel de la `Decider`fonction:

```
Object ( 'hello.c')
Décideur ( 'horodatage match)
```

Lorsqu'il est configuré de cette manière, SCons reconstruira une cible à chaque fois que le temps de modification d'un fichier source a changé. Donc , si nous utilisons l' `touch -t` option pour changer la date de modification `hello.c`à une date ancienne (1 Janvier, 1989), SConsreconstruirons encore le fichier cible:

```
% scons -Q hello.o
cc -o -c hello.o hello.c
% touch -t 198901010000 hello.c
%scons -Q hello.o
cc -o -c hello.o hello.c
```

En général, la seule raison de préférer au `timestamp-newer` lieu de`timestamp-match`, serait si vous avez une raison spécifique d'exiger ce Faites un comportement -comme de ne pas la reconstruction d' une cible lorsqu'un fichier source autre modifiée est plus ancienne.

6.1.3. Décider si un fichier a modifié par les deux signatures MD et Horodateurs

En tant que l' amélioration de la performance, SCons fournit un moyen d'utiliser checksum MD5 du contenu du fichier , mais de lire ces contenus que lorsque l'horodatage du fichier a changé. Pour ce faire, appelez la `Decider` fonction avec l' `'MD5-timestamp'` argument comme suit:

```
Programme ( 'hello.c')
Decider ( 'timestamp MD5')
```

Alors configuré, SCons se comportera toujours comme il le fait lors de l' utilisation `Decider('MD5')`:

```
% scons -Q hello
cc -o -c hello.o hello.c
cc -o bonjour hello.o
% touch hello.c
%scons -Q hello
scons: `bonjour » est à jour.
% edit hello.c
    [CHANGER LE CONTENU DE hello.c]
% scons -Q hello
cc -o -c hello.o hello.c
cc -o bonjour hello.o
```

Cependant, le deuxième appel à SCons dans la sortie ci - dessus, lorsque la construction est mise à jour, aura été réalisée en regardant simplement la date de modification du `hello.c`fichier, pas en l' ouvrant et en effectuant une somme de contrôle MD5 calcuation sur son contenu . Cela peut considérablement accélérer beaucoup la mise à jour builds.

Le seul inconvénient à l' utilisation `Decider('MD5-timestamp')` est que SCons va *pas* reconstruire un fichier cible si un fichier source a été modifiée dans une seconde de la dernière fois SCons construit le fichier. Alors que la plupart des développeurs sont la programmation, ce n'est pas un problème dans la pratique, car il est peu probable que quelqu'un aura construit et pensé assez rapidement pour faire un changement de fond dans un fichier source dans une seconde. Certains scripts de compilation ou d' outils d'intégration continue peuvent toutefois compter sur la possibilité d'appliquer des modifications aux fichiers automatiquement, puis reconstruire le plus rapidement possible, dans lequel l' utilisation de cas `Decider('MD5-timestamp')` peut ne pas être approprié.

6.1.4. Écrire votre propre personnalisée `Decider`Fonction

Les différentes valeurs de chaîne que nous avons passé à la `Decider`fonction sont essentiellement utilisés par SCons pour choisir l' une des nombreuses fonctions internes spécifiques qui mettent en œuvre diverses manières de décider si une dépendance (généralement un fichier source) a changé depuis un fichier cible a été construit. Comme il se trouve, vous pouvez également fournir votre propre fonction de décider si une dépendance a changé.

Par exemple, supposons que nous avons un fichier d'entrée qui contient beaucoup de données, dans un format régulier spécifique, qui est utilisé pour reconstruire beaucoup de différents fichiers cibles, mais chaque fichier cible dépend en réalité que sur une section particulière du fichier d'entrée. Nous aimerions avoir chaque fichier cible dépend uniquement de sa section du fichier d'entrée. Cependant, étant donné que le fichier d'entrée peut contenir un grand nombre de données, nous voulons ouvrir le fichier d'entrée que si son horodatage a changé. Cela pourrait se faire avec une coutume `Decider`fonction qui pourrait ressembler à ceci:

```
Programme ( 'hello.c')
def decide_if_changed (dépendance, cible, prev_ni):
    si self.get_timestamp () = prev_ni.timestamp:
        dep = str (dépendance)
        tgt = str (cible)
        si specific_part_of_file_has_changed (dep, tgt):
            retour vrai
    return false
Decider (decide_if_changed)
```

Notez que dans la définition de la fonction, le `dependency` (fichier d'entrée) est le premier argument, puis la `target`. Ces deux éléments sont transmis aux fonctions comme SCons `Node`objets, que nous convertissons à cordes en utilisant Python `str()`.

Le troisième argument, `prev_ni`est un objet qui contient les informations de signature ou d'un horodatage qui a été enregistrée au sujet de la dépendance à la dernière fois que la cible a été construit. Un `prev_ni`objet peut contenir des informations différentes, en fonction du type de chose que l' `dependency`argument représente. Pour les fichiers normaux, l' `prev_ni`objet possède les attributs suivants:

.csig

La *signature de contenu* , ou la somme de contrôle MD5, du contenu du `dependency` fichier le temps de liste l' `target`a été construite.

.Taille

La taille en octets du `dependency` fichier le temps de liste la cible a été construite.

.timestamp

Le temps de modification du `dependency` fichier le temps de liste l' `target`a été construite.

Notez que ignorant certains des arguments dans votre commande la `Decider`fonction est une chose tout à fait normal de le faire, si elles ne touchent pas la façon dont vous voulez décider si le fichier de dépendance a changé.

Une autre chose à surveiller est le fait que les trois attributs ci - dessus peuvent ne pas être présents au moment de la première manche. Sans accumulation préalable, aucun objectif n'a été créé et aucun `.sconsign`fichier DB existe encore. Donc, vous devriez toujours vérifier si l' `prev_ni`attribut en question est disponible.

Nous présentons enfin un petit exemple pour une `csig`fonction de decider à base. Notez comment les informations de signature du `dependency`fichier doit s'initialisé via `get_csig` lors de chaque appel de fonction (ce qui est obligatoire!).

```
env = environnement ()

def config_file_decider (dépendance, cible, prev_ni):
    importation os.path

    # Nous avons toujours à init la valeur .csig ...
    dep_csig = dependency.get_csig ()
    # .csig peut ne pas exister, car aucune cible a été construit encore ...
    si 'CSIG' pas dir (prev_ni):
        retour vrai
    # Fichier cible ne peut pas exister encore
    sinon os.path.exists (str (target.abspath)):
        retour vrai
    si dep_csig = prev_ni.csig:
        # Certains changements sur le fichier source => mise à jour installée une
        retour vrai
    return false

def update_file ():
    f = open ( "test.txt", "a")
    f.write ( "une ligne \ n")
    f.close ()

update_file ()

# Activer notre propre fonction de decider
env.Decider (config_file_decider)

env.Install ( "installer", "test.txt")
```

6.1.5. Différentes façons de mélange de décider si un fichier a changé

Les exemples précédents ont tous démontré l' appel global `Decider`fonction pour configurer toutes les décisions de dépendance qui SCons fait. , Vous voulez cependant parfois être en mesure de configurer différents processus de prise de décision pour les différentes cibles. Quand cela est nécessaire, vous pouvez utiliser la`env.Decider` méthode pour affecter uniquement les décisions de configuration des cibles construites avec un environnement de construction spécifique.

Par exemple, si nous voulons arbitrairement construire un programme à l'aide checkums MD5 et un autre à l'aide des temps de modification des fichiers de la même source que nous pourrions configurer la façon suivante:

```
env1 = Environnement (CPPPATH = [ ''])
ENV2 = env1.Clone ()
env2.Decider ( 'horodatage match')
env1.Program ( 'prog-MD5', 'program1.c')
env2.Program ( 'prog-timestamp', 'program2.c')
```

Si les deux programmes comprennent le même `inc.h`fichier, puis mettre à jour la date de modification `inc.h` (en utilisant la touche commande) entraînera seulement `prog-timestamp` être reconstruit:

```
% scons -Q
cc -o program1.o -c -I. program1.c
cc -o prog-MD5 program1.o
cc -o program2.o -c -I. program2.c
cc -o program2.o prog-timestamp
% touch inc.h
%scons -Q
cc -o program2.o -c -I. program2.c
cc -o program2.o prog-timestamp
```

6.2. Fonctions plus anciennes pour décider quand un fichier d'entrée a changé

SCons soutient encore deux fonctions qui étaient autrefois les principales méthodes de configuration de la décision de savoir si oui ou non un fichier d'entrée a changé. Ces fonctions ont été officiellement désapprouvée comme SCons version 2.0, et leur utilisation est déconseillée, principalement parce qu'elles reposent sur une distinction quelque peu déroutante entre la façon dont les fichiers source et les fichiers cibles sont traités. Ces fonctions sont documentées ici principalement dans le cas où vous les rencontrez dans les anciens `SConscript`fichiers.

6.2.1. la `SourceSignatures`fonction

La `SourceSignatures`fonction est assez simple, et prend en charge les deux valeurs d'arguments différents pour configurer si les modifications du fichier source doit être décidé en utilisant les signatures MD5:

```
Programme ( 'hello.c')
SourceSignatures ( 'MD5')
```

Ou en utilisant l'horodatage:

```
Programme ( 'hello.c')
SourceSignatures ( 'timestamp')
```

Ce sont à peu près équivalent à la spécification `Decider('MD5')` ou`Decider('timestamp-match')`, respectivement, même si elle ne concerne que la façon dont SCons prend des décisions sur les dépendances sur la *source* des fichiers - qui est, les fichiers qui ne sont pas construits à partir d'autres fichiers.

6.2.2. la `TargetSignatures`fonction

La `TargetSignatures`fonction précise comment SCons décide lorsqu'un fichier cible a changé *quand il est utilisé comme une dépendance de (entrée) une autre cible* --Que est, la `TargetSignatures`fonction configure sont utilisés les signatures de fichiers cibles « intermédiaires » au moment de décider si un " fichier cible en aval » doit être reconstruit. [2]

La `TargetSignatures`fonction prend en charge les mêmes `'MD5'`et les `'timestamp'` valeurs argument qui sont pris en charge par la `SourceSignatures`, avec les mêmes significations, mais appliqué à une cible de fichiers. C'est, dans l'exemple:

```
Programme ( 'hello.c')
TargetSignatures ( 'MD5')
```

La somme de contrôle MD5 du `hello.o`fichier cible sera utilisé pour décider si elle a changé depuis la dernière fois que le « aval » `hello`fichier cible a été construit. Et dans l'exemple:

```
Programme ( 'hello.c')
TargetSignatures ( 'timestamp')
```

Le temps de modification du `hello.o`fichier cible sera utilisé pour décider si elle a changé depuis la dernière fois que le « aval » `hello`fichier cible a été construit.

La `TargetSignatures`fonction prend en charge deux valeurs d'arguments supplémentaires: `'source'`et `'build'`. L' `'source'`argument spécifie que les décisions impliquant si les fichiers cibles ont changé depuis une version précédente devrait utiliser le même comportement pour les décisions configurées pour les fichiers source (en utilisant la `SourceSignatures`fonction). Ainsi , dans l'exemple:

```
Programme ( 'hello.c')
TargetSignatures ( 'source')
SourceSignatures ( 'timestamp')
```

Tous les fichiers, les cibles et les sources, utiliseront les temps de modification au moment de décider si un fichier d'entrée a changé depuis la dernière fois une cible a été construit.

Enfin, l' `'build'`argument spécifie que SCons devrait examiner l'état de la construction d'un fichier cible et reconstruire toujours une cible « en aval » si le fichier cible a été reconstruit lui - même, sans réexaminer le contenu ou l' horodatage du fichier cible nouvellement construit. Si le fichier cible n'a pas été reconstruit au cours de cette `scons`invocation, le fichier cible sera examiné la même manière que configurée par l' `SourceSignature`appel de décider si elle a changé.

Cela imite le comportement des `build signatures` versions antérieures de SCons . Une `build signature`signature combinée ré-de tous les fichiers d'entrée qui sont entrés dans le fichier faisant cible, de sorte que le fichier cible lui - même n'a pas besoin d'avoir lu son contenu pour calculer une signature MD5. Cela peut améliorer les performances pour certaines configurations, mais généralement pas aussi efficace que l' utilisation `Decider('MD5-timestamp')`.

6.3. Dépendances: La Implicite `$CPPPATH`variable de la construction

Supposons maintenant que notre « Bonjour, monde! » programme a en fait une `#include`ligne d'inclure le `hello.h`fichier dans la compilation:

```
#include <hello.h>
int
principale()
{
    printf ( "Bonjour,% s \ n", string);
}
```

Et, pour être complet, le `hello.h`fichier ressemble à ceci:

```
chaîne #define « monde »
```

Dans ce cas, nous voulons SCons de reconnaître que, si le contenu du `hello.h`changement de fichier, le `hello`doit être recompilés programme. Pour ce faire, il faut modifier le `SConstruct`fichier comme ceci:

```
Programme ( 'hello.c', CPPPATH = '')
```

La $CPPPATHvaleur indique SCons de regarder dans le répertoire courant ('.') pour tous les fichiers inclus par les fichiers source C (.cou .hfichiers). Avec cette mission dans le SConstructfichier:

```
% scons -Q hello
cc -o hello.o -c -I. Bonjour c
cc -o bonjour hello.o
% scons -Q hello
scons: `bonjour » est à jour.
% [CHANGEMENT DU CONTENU DE hello.h]
% scons -Q hello
cc -o hello.o -c -I. Bonjour c
cc -o bonjour hello.o
```

Tout d' abord, notez que SCons ajouté l' -I.argument de la $CPPPATHvariable de sorte que la compilation trouverait le hello.hfichier dans le répertoire local.

En second lieu , se rendre compte que SCons sait que le helloprogramme doit être reconstruit car il scanne le contenu du hello.cfichier pour les #includelignes qui indiquent un autre fichier est inclus dans la compilation. SCons enregistre ces comme *dépendances implicites* du fichier cible, par conséquent, lorsque les hello.hmodifications de fichiers, SCons se rend compte que le hello.cfichier comprend, et reconstruit le résultat helloprogramme qui dépend à la fois les hello.cet hello.hfichiers.

Comme la $LIBPATHvariable, la $CPPPATHvariable peut être une liste de répertoires, ou une chaîne séparée par le caractère de séparation de chemin spécifique au système (« : » sur POSIX / Linux, « » sous Windows). De toute façon, SCons crée les options de ligne de commande droite de sorte que l'exemple suivant:

```
Programme ( 'hello.c', CPPPATH = [ 'include', '/ home / projet / inc'])
```

Ressemblera à ceci sur Linux ou Posix:

```
% scons -Q hello
cc -o hello.o -c -Iinclude -I / home / projet / inc hello.c
cc -o bonjour hello.o
```

Et comme celui-ci sous Windows:

```
C: \>scons -Q hello.exe
cl /Fohello.obj / c hello.c / nologo / Iinclude / I \ home \ projet \ inc
lien / nologo /OUT:hello.exe hello.obj
embedManifestExeCheck (cible, source, env)
```

6.4. Mise en cache des dépendances implicites

Numérisation chaque fichier pour les #includelignes prend un certain temps de traitement supplémentaire. Lorsque vous faites une génération complète d'un grand système, le temps de balayage est généralement un très faible pourcentage du temps total consacré à la construction. Vous êtes le plus susceptible de remarquer le temps de balayage, cependant, lorsque vous *reconstruisez* tout ou partie d'un grand système: SCons prendra probablement un peu de temps à « penser à » ce qui doit être construit avant d'émettre la première commande de construction (qui a défini que tout est à jour et rien ne doit être reconstruit).

Dans la pratique, ayant SCons analyser les fichiers de gagner du temps par rapport à la quantité de temps perdu à potentiel traquer les problèmes subtils introduits par des dépendances incorrectes. Néanmoins, le « temps d' attente » , tandis que SCons analyse les fichiers peuvent agacer les développeurs individuels attendant leur builds pour terminer. Par conséquent, SCons vous permet de mettre en cache les dépendances implicites que les scanners trouvent, pour une utilisation par builds plus tard. Vous pouvez le faire en spécifiant l' --implicit-cacheoption sur la ligne de commande:

```
% scons -Q --implicit-cache hello
cc -o -c hello.o hello.c
cc -o bonjour hello.o
% scons -Q hello
scons: `bonjour » est à jour.
```

Si vous ne souhaitez pas spécifier --implicit-cache sur la ligne de commande à chaque fois, vous pouvez le faire le comportement par défaut de votre build en définissant l' implicit_cacheoption dans un SConscriptfichier:

```
SetOption ( 'implicit_cache', 1)
```

SCons ne cache pas les dépendances implicites comme celui - ci par défaut , car les --implicit-cachecauses SCons d'utiliser simplement les dépendances implicites stockés au cours de la dernière exécution, sans aucune vérification pour savoir si oui ou non ces dépendances sont toujours correctes. Plus précisément, cela signifie --implicit-cacheinstruit SCons de *ne pas* reconstruire « correctement » dans les cas suivants:

- Quand --implicit-cacheest utilisé, SCons ignore les changements qui ont été faits pour rechercher des chemins (comme $CPPPATHou $LIBPATH,). Cela peut conduire à SCons pas la reconstruction d' un fichier si une modification $CPPPATHentraînerait normalement un autre, un fichier du même nom à partir d' un autre répertoire à utiliser.
- Quand --implicit-cacheest utilisé, SCons ne détecte pas si un fichier du même nom a été ajouté à un répertoire qui est plus tôt dans le chemin de recherche que le répertoire dans lequel le fichier a été trouvé la dernière fois.

6.4.1. l' --implicit-deps-changedoption

Lors de l' utilisation des dépendances implicites mises en cache, parfois , vous voulez « nouveau départ » et ont SCons re-scanner les fichiers pour lesquels il précédemment mis en mémoire cache les dépendances. Par exemple, si vous avez récemment installé une nouvelle version de code externe que vous utilisez pour la compilation, les fichiers d' en- tête externes ont changé et les dépendances implicites précédemment mises en cache seront mis à jour. Vous pouvez les mettre à jour en exécutant SCons avec l' --implicit-deps-changedoption de :

```
% scons -Q --implicit-deps-changed hello
cc -o -c hello.o hello.c
cc -o bonjour hello.o
% scons -Q hello
scons: `bonjour » est à jour.
```

Dans ce cas, SCons sera analysée à nouveau toutes les dépendances implicites et cache des copies mises à jour des informations.

6.4.2. l' `--implicit-deps-unchanged`option

Par défaut , lorsque la mise en cache des dépendances, SCons notifications lorsqu'un fichier a été modifié et Rescanne le fichier pour toute information de dépendance implicite à jour. Parfois, cependant, vous pouvez forcer SCons à utiliser les dépendances implicites mises en cache, même si les fichiers source modifiés. Cela peut accélérer une construction par exemple, lorsque vous avez changé vos fichiers source mais sachez que vous n'avez pas changé les `#include`lignes. Dans ce cas, vous pouvez utiliser l' `--implicit-deps-unchanged`option de :

```
% scons -Q --implicit-deps-unchanged hello
cc -o -c hello.o hello.c
cc -o bonjour hello.o
% scons -Q hello
scons: `bonjour » est à jour.
```

Dans ce cas, SCons supposera que les dépendances implicites mises en cache sont corrects et ne dérangera pas de re-scan les fichiers modifiés. Pour typique construit après de petites modifications incrémentielles aux fichiers source, les économies peuvent ne pas être très grand, mais parfois tous les bits de compte de l' amélioration des performances.

6.5. Explicite Dépendances: la `Depends`fonction

Parfois , un fichier dépend d'un autre fichier non détecté par un SCons scanner. Pour cette situation, SCons vous permet de spécifique explicitement qu'un fichier dépend d'un autre fichier, et doit être reconstruit à chaque fois que les modifications de fichiers. Ceci est spécifié à l' aide du `Depends`procédé:

```
bonjour = Programme ( 'hello.c')
Cela dépend (bonjour, « autre_fichier »)


% scons -Q hello
cc -c -o hello.c hello.o
cc -o bonjour hello.o
% scons -Q hello
scons: `bonjour » est à jour.
% edit other_file
    [CHANGER LE CONTENU DE autre_fichier]
% scons -Q hello
cc -c -o hello.c hello.o
cc -o bonjour hello.o
```

Notez que la dépendance (le deuxième argument `Depends`) peut aussi être une liste d'objets Node (par exemple, comme renvoyé par un appel à un constructeur):

```
bonjour = Programme ( 'hello.c')
au revoir = Programme ( 'goodbye.c')
En fonction (bonjour, au revoir)
```

dans ce cas, la dépendance ou des dépendances seront construits avant que la cible (s):

```
% scons -Q hello
cc -c -o goodbye.c goodbye.o
cc -o au revoir goodbye.o
cc -c -o hello.c hello.o
cc -o bonjour hello.o
```

6.6. Dépendances A partir de fichiers externes: la `ParseDepends` fonction

SCons a des scanners intégrés pour un certain nombre de langues. Parfois , ces scanners ne parviennent pas à extraire certaines dépendances implicites en raison des limites de la mise en œuvre du scanner.

L'exemple suivant illustre un cas où le scanner intégré C est incapable d'extraire la dépendance implicite sur un fichier d'en-tête.

```
#define FOO_HEADER <foo.h>
FOO_HEADER #include

int main() {
    retourner FOO;
}


% scons -Q
cc -o hello.o -c -I. Bonjour c
cc -o bonjour hello.o
% [CONTENU DE CHANGEMENT foo.h]
% scons -Q
scons: ` « . est à jour.
```

Apparemment, le scanner ne connaît pas la dépendance d'en-tête. Être pas un préprocesseur C à part entière, le scanner ne développe pas la macro.

Dans ce cas, vous pouvez également utiliser le compilateur pour extraire les dépendances implicites. `ParseDepends`peut analyser le contenu de la sortie du compilateur dans le style de Marque et établir explicitement toutes les dépendances énumérées.

L'exemple suivant utilise `ParseDepends`pour traiter un fichier de dépendance générée par compilateur , qui est généré comme un effet secondaire lors de la compilation du fichier d'objet:

```
obj = Object ( 'hello.c', CCFLAGS = '- MD -MF hello.d', CPPPATH = '')
Sideeffect ( 'hello.d', obj)
ParseDepends ( 'hello.d')
Programme ( 'bonjour', obj)
```

```
% scons -Q
cc -o hello.o -c -MD -MF hello.d -I. Bonjour c
cc -o bonjour hello.o
% [CONTENU DE CHANGEMENT foo.h]
% scons -Q
cc -o hello.o -c -MD -MF hello.d -I. Bonjour c
```

Dépendances d'un Parsing généré par le compilateur `.d`fichier a un problème de la poule et de l' œuf, qui provoque des reconstructions inutiles:

```
% scons -Q
cc -o hello.o -c -MD -MF hello.d -I. Bonjour c
cc -o bonjour hello.o
% scons -Q --debug=explain
scons: reconstruction `hello.o « parce que `foo.h » est une nouvelle dépendance
cc -o hello.o -c -MD -MF hello.d -I. Bonjour c
% scons -Q
scons: ` « . est à jour.
```

Dans la première passe, le fichier de dépendance est généré tandis que le fichier d'objet est compilée. À ce moment - là, SCons ne connaît pas la dépendance `foo.h`. Dans la seconde passe, le fichier objet est régénéré en raison `foo.h` est détectée comme une nouvelle dépendance.

`ParseDepends`lit immédiatement le fichier spécifié au moment de l' appel et renvoie simplement si le fichier n'existe pas. Un fichier de dépendance générée pendant le processus de génération ne sont pas automatiquement analysé à nouveau. Par conséquent, les dépendances extraite compilateur ne sont pas stockées dans la base de données de signature pendant la même passe de recharge. Cette limitation de `ParseDepends`conduit à recompilations inutiles. Par conséquent, `ParseDepends`ne devrait être utilisé que si les scanners ne sont pas disponibles pour la langue employée ou pas assez puissant pour la tâche spécifique.

6.7. Ignorer les dépendances: la `Ignore`fonction

Parfois , il est logique de ne pas reconstruire un programme, même si un fichier change de dépendance. Dans ce cas, vous dire SConsspécifiquement d'ignorer une dépendance comme suit:

```
hello_obj = Objet ( 'hello.c')
bonjour = Programme (hello_obj)
Ignorer (hello_obj, 'hello.h')
```

```
% scons -Q hello
cc -c -o hello.o hello.c
cc -o bonjour hello.o
% scons -Q hello
scons: `bonjour » est à jour.
% edit hello.h
  [CHANGER LE CONTENU DE hello.h]
% scons -Q hello
scons: `bonjour » est à jour.
```

Maintenant, l'exemple ci - dessus est un peu artificiel, car il est difficile d'imaginer une situation réelle où vous ne voudriez pas reconstruire `hello` si le `hello.h`fichier a été modifié. Un exemple plus réaliste pourrait être si le `hello` programme est en cours de construction dans un répertoire partagé entre plusieurs systèmes qui ont des copies du`stdio.h`fichier include. Dans ce cas, SCons remarqueraient les différences entre les différents systèmes de copies de de `stdio.h` et se reconstruire `hello` chaque fois que vous changez les systèmes. Vous pouvez éviter ces reconstructions comme suit:

```
bonjour = Programme ( 'hello.c', CPPPATH = [ '/ usr / include'])
Ignorer (bonjour, '/usr/include/stdio.h')
```

`Ignore`peut également être utilisé pour empêcher un fichier généré à partir d'être construit par défaut. Cela est dû au fait que les répertoires dépendent de leur contenu. Donc, pour ignorer un fichier généré à partir de la version par défaut, vous indiquez que le répertoire doit ignorer le fichier généré. Notez que le fichier sera toujours construit si l'utilisateur demande spécifiquement la cible sur scons ligne de commande, ou si le fichier est une dépendance d'un autre fichier qui est demandé et / ou est construit par défaut.

```
hello_obj = Objet ( 'hello.c')
bonjour = Programme (hello_obj)
Ignorer ( '', [bonjour, hello_obj])
```

```
% scons -Q
scons: ` « . est à jour.
% scons -Q hello
cc -o -c hello.o hello.c
cc -o bonjour hello.o
% scons -Q hello
scons: `bonjour » est à jour.
```

6.8. Commander seulement Dépendances: la `Requires`fonction

De temps en temps, il peut être utile de préciser qu'un certain fichier ou le répertoire doit, le cas échéant, être construits ou créés avant une autre cible est construit, mais que les modifications de ce fichier ou répertoire ne *pas* exiger que la cible elle - même être reconstruit. Une telle relation est appelée une *dépendance à l'ordre uniquement* parce qu'elle ne concerne que l'ordre dans lequel les choses doivent être construites - la dépendance avant la cible - mais pas une relation de dépendance stricte parce que la cible ne devrait pas changer en réponse aux changements de le fichier dépendant.

Par exemple, supposons que vous voulez créer un fichier à chaque fois que vous exécutez une génération qui identifie le temps a été réalisé la construction, le numéro de version, etc., et qui est inclus dans tous les programmes que vous construisez. Le contenu du fichier version changera chaque build. Si vous spécifiez une relation de dépendance normale, tout programme qui dépend de ce fichier serait reconstruit à chaque fois que vous avez exécuté SCons . Par exemple, nous pourrions utiliser un code Python dans un `SConstruct`fichier pour créer un nouveau `version.c`fichier avec une chaîne contenant la date actuelle chaque fois que nous courons SCons , puis lier un programme avec le fichier objet résultant en dressant la liste `version.c`des sources:

```
temps d'importation
```

```
version_c_text = "" »
char * date = "% s";
"" » % Time.ctime (time.time ())
open ( 'version.c', 'w'). écrire (version_c_text)

bonjour = Programme ([ 'hello.c', 'version.c'])
```

Si nous listons en `version.c`tant que fichier source réelle, bien que, le `version.o`fichier aura reconstruit à chaque fois que nous courons SCons (parce que le `SConstruct`fichier lui - même modifie le contenu `version.c`) et l' `hello`exécutable se rebranché chaque fois (parce que les `version.o`changements de fichiers):

```
% scons -Q hello
cc -o -c hello.o hello.c
cc -o -c version.o version.c
cc -o bonjour hello.o version.o
% sleep 1
%scons -Q hello
cc -o -c version.o version.c
cc -o bonjour hello.o version.o
% sleep 1
%scons -Q hello
cc -o -c version.o version.c
cc -o bonjour hello.o version.o
```

(Notez que pour l'exemple ci - dessus pour travailler, nous dormons pendant une seconde entre chaque course, de sorte que le `SConstruct`fichier va créer un `version.c`fichier avec une chaîne de temps qui est une seconde plus tard que la course précédente.)

Une solution consiste à utiliser la `Requires`fonction de préciser que le `version.o` doit être reconstruit avant qu'il ne soit utilisé par l'étape de liaison, mais qui change à `version.o` ne devrait pas réellement provoquer l' `hello` exécutable à rebranché:

```
temps d'importation

version_c_text = "" »
char * date = "% s";
"" » % Time.ctime (time.time ())
open ( 'version.c', 'w'). écrire (version_c_text)

version_obj = Objet ( 'version.c')

bonjour = Programme ( 'hello.c',
                LINKFLAGS = str (version_obj [0]))

Nécessite (bonjour, version_obj)
```

Notez que parce que nous ne pouvons plus la liste `version.c` comme l' une des sources pour le `hello`programme, nous devons trouver une autre façon de le faire entrer dans la ligne de commande de lien. Pour cet exemple, nous tricher un peu et bourrer le nom de fichier objet (extrait de la `version_obj` liste renvoyée par l' `Object`appel) dans la `$LINKFLAGS`variable car `$LINKFLAGS`est déjà inclus dans la `$LINKCOM`ligne de commande.

Avec ces changements, nous obtenons le comportement souhaité de ne re-lier l' `hello`exécutable lorsque l' `hello.c`a changé, même si l' `version.o`on reconstruit (parce que le `SConstruct`fichier change encore le`version.c`contenu directement chaque exécution):

```
% scons -Q hello
cc -o -c version.o version.c
cc -o -c hello.o hello.c
cc -o bonjour version.o hello.o
% sleep 1
%scons -Q hello
cc -o -c version.o version.c
scons: `bonjour » est à jour.
% sleep 1
% [CHANGEMENT DU CONTENU DE hello.c]
% scons -Q hello
cc -o -c version.o version.c
cc -o -c hello.o hello.c
cc -o bonjour version.o hello.o
% sleep 1
%scons -Q hello
cc -o -c version.o version.c
scons: `bonjour » est à jour.
```

6.9. la `AlwaysBuild`fonction

Comment SCons gère les dépendances peuvent également être affectées par la `AlwaysBuild`méthode. Lorsqu'un fichier est passé à la `AlwaysBuild`méthode, comme suit:

```
bonjour = Programme ( 'hello.c')
AlwaysBuild (bonjour)
```

Ensuite , le fichier cible spécifié (`hello`dans notre exemple) sera toujours considéré comme hors-date et reconstruit à chaque fois que le fichier cible est évalué en marchant le graphe de dépendance:

```
% scons -Q
cc -o -c hello.o hello.c
cc -o bonjour hello.o
% scons -Q
cc -o bonjour hello.o
```

La `AlwaysBuild`fonction a un nom quelque peu trompeur, parce que cela ne signifie pas réellement le fichier cible sera reconstruit à chaque fois SCons est invoquée. Au lieu de cela, cela signifie que la cible, en effet, être reconstruit chaque fois que le fichier cible est rencontré tout en évaluant les cibles spécifiées sur la ligne de commande (et leurs dépendances). Donc , en spécifiant une autre cible sur la ligne de commande, une cible qui ne *pas* se dépend de la `AlwaysBuild`cible, sera encore reconstruit que si elle est hors de jour par rapport à ses dépendances:

```
% scons -Q
cc -o -c hello.o hello.c
cc -o bonjour hello.o
% scons -Q hello.o
scons: `hello.o » est à jour.
```

---

[2] Cette distinction facilement négligé entre la façon dont SCons décide si la cible elle - même doit être reconstruit et comment la cible est ensuite utilisé pour décider si une autre cible doit être reconstruit est l' une des choses confuses qui a conduit à la `TargetSignatures` et les `SourceSignatures`fonctions étant remplacées par la simple `Decider`fonction.

Chapitre 7. Les environnements

Une `environment` est une collection de valeurs qui peuvent affecter la façon dont un programme est exécuté. SCons distingue trois différents types d'environnements qui peuvent influer sur le comportement de SCons lui - même (sous réserve de la configuration dans les `SConscript`fichiers), ainsi que les compilateurs et autres outils qu'il exécute:

Environnement externe

> Le `external environment` est l'ensemble des variables dans l'environnement de l'utilisateur au moment où l'utilisateur exécute SCons . Ces variables sont disponibles dans les `SConscript`fichiers à travers le Python `os.environ`dictionnaire. Voir la section 7.1, « Utilisation des valeurs de l'environnement extérieur » , ci - dessous.

`Construction Environment`

> A `construction environment` est un objet distinct créer dans un `SConscript`fichier et et qui contient des valeurs qui influent sur la façon dont SCons décide des mesures à utiliser pour construire une cible, et même de définir des objectifs qui doivent être construits à partir de quelles sources. L' une des caractéristiques les plus puissantes de SCons est la possibilité de créer plusieurs `construction environments`, y compris la possibilité de cloner une nouvelle, sur mesure à `construction environment`partir d' un existant `construction environment`. Voir la section 7.2, « environnements de construction » , ci - dessous.

Environnement d'exécution

> Une `execution environment` est que les valeurs SCons fixe lors de l' exécution d' une commande externe (comme un compilateur ou lieur) pour construire une ou plusieurs cibles. Notez que ce n'est pas la même que celle `external environment` (voir ci - dessus). Voir la section 7.3, « Contrôle de l'environnement d'exécution des commandes émises » , ci - dessous.

Contrairement à Make , SCons ne copie pas automatiquement ou les valeurs d'importation entre les différents environnements (à l'exception des clones explicites de `construction environments`, qui héritent des valeurs de leurs parents). Ceci est un choix de conception délibérée de faire en sorte que builds sont, par défaut, répétables quelles que soient les valeurs dans l' environnement externe de l'utilisateur. Cela évite une classe de problèmes avec construit où la construction locale d'un développeur fonctionne car un paramètre variable personnalisée provoque une option compilateur ou construire différents à utiliser, mais le check-in changement casse la construction officielle , car il utilise différents paramètres variables d'environnement.

Notez que l' `SConscript`auteur peut facilement organiser des variables à copier ou importées entre les environnements, ce qui est souvent très utile (voire carrément nécessaire) pour le rendre facile pour les développeurs de personnaliser la construction de manière appropriée. Le point est *pas* que les variables de copie entre les différents environnements est mal et doivent toujours être évités. Au lieu de cela, il devrait être à l'implémenteur du système de construction pour faire des choix conscients quand et comment importer une variable d'un environnement à l' autre, de prendre des décisions éclairées au sujet de trouver le juste équilibre entre faire la construction répétitive d'une part et pratique à utiliser sur l'autre.

7.1. Utilisation des valeurs de l'environnement extérieur

Les `external environment` paramètres variables que l'utilisateur a en vigueur lors de l' exécution SCons sont disponibles dans le Python normale `os.environ` dictionnaire. Cela signifie que vous devez ajouter une `import os`déclaration à tout `SConscript`fichier dans lequel vous souhaitez utiliser les valeurs de l' environnement externe de l'utilisateur.

```
import os
```

Plus utile, vous pouvez utiliser le `os.environ` dictionnaire dans vos `SConscript` fichiers pour initialiser `construction environments` avec les valeurs de l' environnement externe de l'utilisateur. Voir la section suivante, section 7.2, « environnements de construction » , pour plus d' informations sur la façon de le faire.

7.2. Les environnements de construction

Il est rare que tous les logiciels dans un grand système complexe doit être construit de la même façon. Par exemple, les fichiers sources différentes peuvent avoir besoin différentes options activées sur la ligne de commande, ou différents programmes exécutables doivent être liés à différentes bibliothèques. SCons tient compte de ces différentes exigences de construction en vous permettant de créer et de configurer plusieurs `construction environments` qui contrôlent la façon dont le logiciel est construit. A `construction environment`est un objet qui a un certain nombre de associés `construction variables`, chacun ayant un nom et une valeur. (Un environnement de construction a également un ensemble fixé de `Builder`méthodes, dont nous apprendrons plus tard.)

7.2.1. Création d' un `Construction Environment`: la `Environment`fonction

A `construction environment`est créé par le `Environment`procédé:

```
env = environnement ()
```

Par défaut, SCons initialise chaque nouvel environnement de construction avec un ensemble de `construction variables` base sur les outils qu'il trouve sur votre système, ainsi que l'ensemble par défaut des méthodes de constructeur nécessaires à l' utilisation de ces outils. Les variables de construction sont initialisés avec des valeurs décrivant le compilateur C, le compilateur Fortran, l'éditeur de liens, etc., ainsi que les lignes de commande pour les invoquer.

Lorsque vous initialisez un environnement de construction , vous pouvez définir les valeurs de l'environnement de `construction variables` contrôler la façon dont un programme est construit. Par exemple:

```
env = environnement (CC = 'gcc',
                  CCFLAGS = '-O2')

env.Program ( 'foo.c')
```

L'environnement de la construction dans cet exemple est encore initialisé avec les mêmes valeurs de variables de construction par défaut, à l' exception que l'utilisateur a explicitement l' utilisation spécifiée du compilateur GNU C gcc , et précise en outre que le `-O2`drapeau (niveau d'optimisation deux) doit être utilisé lors de la compilation de l'objet fichier. En d' autres termes, les initialisations explicites de `$CC`et `$CCFLAGS` remplacent les valeurs par défaut dans l'environnement de construction nouvellement créé. Ainsi , une course de cet exemple ressemblerait à ceci:

```
% scons -Q
gcc -o foo.o -c -O2 foo.c
gcc -o foo foo.o
```

7.2.2. Valeurs récupérer d'un`Construction Environment`

Vous pouvez chercher des variables de construction individuelles en utilisant la syntaxe normale pour accéder à des éléments individuels nommés dans un dictionnaire Python:

```
env = environnement ()
print "CC est:", env [ 'CC']
```

Cet exemple `SConstruct`fichier ne construit rien, mais parce qu'il est en fait un script Python, il imprime la valeur `$CC`pour nous:

```
% scons -Q
CC est: cc
scons: ` « . est à jour.
```

Un environnement de construction, cependant, est en fait un objet avec des méthodes associées, etc. Si vous voulez avoir un accès direct à la seule dictionnaire des variables de construction, vous pouvez récupérer cette aide de la `Dictionary`méthode:

```
env = environnement (FOO = 'foo', 'bar' = BAR)
dict = env.Dictionary ()
pour la clé dans [ 'OBJSUFFIX', 'LIBSUFFIX', 'PROGSUFFIX']:
    print "key =% s, valeur =% s" % (clé, dict [key])
```

Ce `SConstruct`fichier permet d' imprimer les éléments spécifiés dictionnaire pour nous sur les systèmes POSIX comme suit:

```
% scons -Q
key = OBJSUFFIX, valeur = .o
key = LIBSUFFIX, valeur = .a
key = PROGSUFFIX, valeur =
scons: ` « . est à jour.
```

Et sous Windows:

```
C: \>scons -Q
key = OBJSUFFIX, valeur = obj
key = LIBSUFFIX, valeur = .lib
key = PROGSUFFIX, valeur = Exe
scons: ` « . est à jour.
```

Si vous voulez faire une boucle et d'imprimer les valeurs de toutes les variables de construction dans un environnement de construction, le code Python pour faire dans l'ordre de tri pourrait ressembler à:

```
env = environnement ()
pour objet triées (env.Dictionary () éléments (.)):
    print "variable de construction = '% s', valeur = '% s'" % article
```

7.2.3. L' expansion des valeurs d'un `Construction Environment`: la `subst`méthode

Une autre façon d'obtenir des informations à partir d' un environnement de construction est d'utiliser la `subst`méthode sur une chaîne contenant des `$`extensions de noms de variables de construction. A titre d'exemple simple, l'exemple de la section précédente qui a utilisé `env['CC']` pour récupérer la valeur de `$CC` peut aussi être écrit:

```
env = environnement ()
print "CC est:", env.subst ( '$ CC')
```

Un avantage d'utiliser `subst`pour développer des chaînes est que les variables de construction dans le résultat se redressèrent jusqu'à ce qu'il n'y a pas d' expansion à gauche dans la chaîne. Ainsi , une lecture simple d'une valeur comme `$CCCOM`:

```
env = environnement (CCFLAGS = '-DFOO')
print "CCCOM est:", env [ 'CCCOM']
```

Imprimera la valeur non déployé `$CCCOM`, nous montrant les variables de construction qui doivent encore être élargi:

```
% scons -Q
CCCOM est: $ CC CCFLAGS $ $ $ CPPFLAGS _CPPDEFFLAGS $ _CPPINCFLAGS -c -o $ CIBLES SOURCES $
scons: ` « . est à jour.
```

L' appel de la `subst`méthode sur `$CCOM`cependant:

```
env = environnement (CCFLAGS = '-DFOO')
print "CCCOM est:", env.subst ( 'CCCOM de $)
```

Est-ce que récursive développer toutes les variables de construction préfixées $(signes dollar), nous montrant le résultat final:

```
% scons -Q
CCCOM est: gcc -c -o -DFOO
scons: ` « . est à jour.
```

Notez que parce que nous ne sommes pas en expansion dans le contexte de construire quelque chose , il n'y a pas de fichiers cible ou la source pour $TARGETet $SOURCESà se développer.

7.2.4. Manipulation Problèmes avec l'expansion de la valeur

Si un problème se produit lors de l' expansion d' une variable de construction, par défaut , il est étendu à '' (une chaîne vide), et ne causera pas scons à l' échec.

```
env = environnement ()
print "valeur est:", env.subst ( '-> $ MISSING <-')
```

```
% scons -Q
valeur est: -> <-
scons: ` « . est à jour.
```

Ce comportement par défaut peut être modifié à l' aide de la AllowSubstExceptions fonction. Lorsqu'un problème se produit avec une extension variable , il génère une exception, et les AllowSubstExceptionscontrôles de fonction qui de ces exceptions sont en fait mortels et qui sont autorisés à se produire en toute sécurité. Par défaut, NameErroret IndexError sont les deux exceptions qui sont autorisés à se produire: Ainsi , au lieu de provoquer scons à l' échec, ceux - ci sont pris, la variable élargie pour '' et l' exécution de scons continue. Pour exiger que tous les noms de variables de construction existent, et que les indices hors de portée ne sont pas autorisés, appelez AllowSubstExceptions sans arguments supplémentaires.

```
AllowSubstExceptions ()
env = environnement ()
print "valeur est:", env.subst ( '-> $ MISSING <-')
```

```
% scons -Q
La valeur est la suivante:
scons: *** NameError `MISSING 'essayer d'évaluer` $ MISSING'
Fichier "/ home / mon / projet / SConstruct", ligne 3, dans <module>
```

Cela peut également être utilisé pour permettre à d' autres exceptions qui pourraient se produire, le plus utilement avec la ${...}syntaxe des variables de construction. Par exemple, cela permettrait zéro division de se produire dans une expansion variables en plus des exceptions par défaut autorisés

```
AllowSubstExceptions (IndexError, NameError, ZeroDivisionError)
env = environnement ()
print "valeur est:", env.subst ( '-> $ {1/0} <-')
```

```
% scons -Q
valeur est: -> <-
scons: ` « . est à jour.
```

Si l' AllowSubstExceptionson appelle plusieurs fois, chaque appel complètement la liste remplace précédente des exceptions autorisées.

7.2.5. Contrôle du défaut Construction Environment: la DefaultEnvironmentfonction

Toutes les Builderfonctions que nous avons mis en place jusqu'à présent, comme Programet Library, utilisez effectivement un défaut construction environment qui contient des paramètres pour les différents compilateurs et d' autres outils qui SCons configure par défaut, ou qu'il sait autrement au sujet et a découvert sur votre système. L'objectif de l'environnement de la construction par défaut est de faire de nombreuses configurations « simplement » pour créer des logiciels en utilisant des outils facilement disponibles avec un minimum de changements de configuration.

Vous pouvez toutefois contrôler les paramètres de l'environnement de la construction par défaut en utilisant la DefaultEnvironmentfonction pour initialiser les différents paramètres:

```
DefaultEnvironment (CC = '/ usr / local / bin / gcc')
```

Lorsqu'il est configuré comme ci - dessus, tous les appels vers le Program ou ObjectBuilder construiront des fichiers d'objets avec le/usr/local/bin/gcc compilateur.

Notez que la DefaultEnvironmentfonction retourne l'objet de l' environnement de la construction par défaut initialisé, ce qui peut alors être manipulé comme tout autre environnement de construction. Donc , ce qui suit serait équivalent à l'exemple précédent, le réglage de la $CC variable /usr/local/bin/gcc mais comme une étape distincte après l'environnement de construction par défaut a été initialisé:

```
env = DefaultEnvironment ()
env [ 'CC'] = '/ usr / local / bin / gcc'
```

Une utilisation très courante de la DefaultEnvironmentfonction est d'accélérer SCons initialisation. Dans le cadre d'essayer de faire la plupart des configurations par défaut « fonctionnent », SCons rechercheront en fait le système local pour les compilateurs installés et d' autres services publics. Cette recherche peut prendre du temps, en particulier sur les systèmes avec les systèmes de fichiers lents ou en réseau. Si vous connaissez le compilateur (s) et / ou d' autres utilitaires que vous souhaitez configurer, vous pouvez contrôler la recherche que SCons effectue en spécifiant certains modules d'outils spécifiques qui pour initialiser l'environnement de la construction par défaut:

```
env = DefaultEnvironment (outils = [ 'gcc', 'gnulink'],
                          CC = '/ usr / local / bin / gcc')
```

Ainsi , l'exemple ci - dessus dire SCons configurer explicitement l'environnement par défaut à utiliser ses paramètres normaux du compilateur GNU et GNU Linker (sans avoir à les chercher, ou tout autre utilitaire pour cette question), et plus précisément d'utiliser le compilateur trouvé à /usr/local/bin/gcc.

7.2.6. Plusieurs`Construction Environments`

L'avantage réel des environnements de construction est que vous pouvez créer autant de différents environnements de construction que vous avez besoin, chacun étant adapté à une autre façon de construire un morceau de logiciel ou tout autre fichier. Si, par exemple, nous devons construire un programme avec le `-O2`drapeau et l' autre avec la `-g`(debug) drapeau, nous le ferions comme ceci:

```
opt = environnement (CCFLAGS = '-O2')
dbg = Environnement (CCFLAGS = '-g')

opt.Program ( 'foo', 'foo.c')

dbg.Program ( 'bar', 'bar.c')
```

```
% scons -Q
cc -o bar.o -c -g bar.c
cc -o bar bar.o
cc -o foo.o -c -O2 foo.c
cc -o foo foo.o
```

On peut même utiliser des environnements de construction multiples pour créer plusieurs versions d'un seul programme. Si vous faites cela en essayant simplement d'utiliser le `Program`constructeur avec les deux environnements, bien que, comme ceci:

```
opt = environnement (CCFLAGS = '-O2')
dbg = Environnement (CCFLAGS = '-g')

opt.Program ( 'foo', 'foo.c')

dbg.Program ( 'foo', 'foo.c')
```

Ensuite SCons génère l'erreur suivante:

```
% scons -Q

scons: *** Deux environnements avec des actions différentes ont été spécifiés pour la même cible: foo.o
Fichier "/ home / mon / projet / SConstruct", ligne 6, dans <module>
```

En effet , les deux `Program`appels ont chacun implicitement racontées SCons pour générer un fichier objet nommé `foo.o`, l' une avec une `$CCFLAGS`valeur `-O2` et une avec une `$CCFLAGS`valeur `-g`. SCons ne peut pas simplement décider que l' un d'entre eux devrait l' emporter sur l'autre, il génère l'erreur. Pour éviter ce problème, il faut préciser explicitement que chaque environnement de compilation `foo.c` dans un fichier objet nommé séparément à l' aide du `Object`constructeur, comme suit:

```
opt = environnement (CCFLAGS = '-O2')
dbg = Environnement (CCFLAGS = '-g')

o = opt.Object ( 'opt-foo', 'foo.c')
opt.Program (o)

d = dbg.Object ( 'foo-dbg', 'foo.c')
dbg.Program (d)
```

Notez que chaque appel au `Object`constructeur renvoie une valeur, un interne SCons objet qui représente le fichier objet qui sera construit. Nous utilisons ensuite cet objet comme entrée au `Program`constructeur. Cela évite d' avoir à spécifier explicitement le nom de fichier objet en plusieurs endroits, et rend lisible pour un compact, `SConstruct`fichier. Notre SCons sortie ressemble alors à :

```
% scons -Q
cc -o foo-dbg.o -c -g foo.c
cc -o foo-dbg foo-dbg.o
cc -o foo-opt.o -c -O2 foo.c
cc -o foo-opt-foo opt.o
```

7.2.7. Faire des copies de `Construction Environments`: la `Clone`méthode

Parfois , vous voulez plus d'un environnement de construction pour partager les mêmes valeurs pour une ou plusieurs variables. Plutôt que de toujours avoir à répéter toutes les variables communes lorsque vous créez chaque environnement de construction, vous pouvez utiliser la `Clone`méthode pour créer une copie d'un environnement de construction.

Comme l' `Environment`appel qui crée un environnement de construction, la `Clone`méthode prend les `construction variable`affectations, qui remplacent les valeurs dans l'environnement de construction copié. Par exemple, supposons que nous voulons utiliser gcc pour créer trois versions d'un programme, un optimisé, un débogage, et un avec non plus . Nous pourrions faire cela en créant un environnement de construction « de base » qui met `$CC`à gcc , et la création de deux exemplaires, l' un qui fixe `$CCFLAGS`pour l' optimisation et l'autre qui définit `$CCFLAGS`pour le débogage:

```
env = environnement ( 'de gcc' CC =)
opt = env.Clone (CCFLAGS = '-O2')
dbg = env.Clone (CCFLAGS = '-g')

env.Program ( 'foo', 'foo.c')

o = opt.Object ( 'opt-foo', 'foo.c')
opt.Program (o)

d = dbg.Object ( 'foo-dbg', 'foo.c')
dbg.Program (d)
```

Ensuite, notre sortie ressemblerait à ceci:

```
% scons -Q
gcc -o foo.o -c foo.c
gcc -o foo foo.o
gcc -o foo-dbg.o -c -g foo.c
gcc-foo foo-dbg dbg.o
gcc foo-opt.o -c -O2 foo.c
gcc-foo foo-opt opt.o
```

7.2.8. Remplacement de valeurs: la `Replace`méthode

Vous pouvez remplacer les valeurs de variables de construction existante en utilisant la `Replace`méthode:

```
env = environnement (CCFLAGS = '-DDEFINE1')
env.Replace (CCFLAGS = '-DDEFINE2')
env.Program ( 'foo.c')
```

La valeur en remplaçant (`-DDEFINE2`dans l'exemple ci - dessus) remplace complètement la valeur dans l'environnement de construction:

```
% scons -Q
cc -o -c foo.o -DDEFINE2 foo.c
cc -o foo foo.o
```

Vous pouvez appeler en toute sécurité `Replace` pour les variables de construction qui n'existent pas dans l'environnement de construction:

```
env = environnement ()
env.Replace (nouvelle_variable = 'xyzzy')
print "nouvelle_variable =", env [ 'new_variable']
```

Dans ce cas, la variable de la construction s'ajoute simplement à l'environnement de construction:

```
% scons -Q
Nouvelle_variable = xyzzy
scons: ` « . est à jour.
```

Étant donné que les variables ne sont pas développées jusqu'à ce que l'environnement de la construction est en fait utilisé pour construire les cibles, et parce que SCons fonction et appelle la méthode sont indépendantes de l' ordre, le dernier remplacement « gagne » et est utilisé pour construire toutes les cibles, quel que soit l'ordre que les appels à remplacer () sont entrecoupés par des appels à des méthodes de constructeur:

```
env = environnement (CCFLAGS = '-DDEFINE1')
print "CCFLAGS =", env [ 'CCFLAGS']
env.Program ( 'foo.c')

env.Replace (CCFLAGS = '-DDEFINE2')
print "CCFLAGS =", env [ 'CCFLAGS']
env.Program ( 'bar.c')
```

Le moment où le remplacement se produit en fait par rapport au moment où les cibles sont construites devient évident si nous courons `scons`sans `-Q` possibilité:

```
% scons
scons: Lecture des fichiers SConscript ...
CCFLAGS = -DDEFINE1
CCFLAGS = -DDEFINE2
scons: fait la lecture des fichiers SConscript.
scons: objectifs de construction ...
cc -o -c bar.o -DDEFINE2 bar.c
cc -o bar bar.o
cc -o -c foo.o -DDEFINE2 foo.c
cc -o foo foo.o
scons: fait des objectifs de construction.
```

Parce que le remplacement se produit alors que les `SConscript`fichiers sont en cours de lecture, la `$CCFLAGS` variable a déjà été réglé `-DDEFINE2`au moment où la `foo.o`cible est construit, même si l'appel à la `Replace`méthode ne se produit pas plus tard dans le `SConscript`fichier.

7.2.9. Définition des valeurs que si elles ne sont pas déjà définis: la `SetDefault`méthode

Il est parfois utile de pouvoir spécifier qu'une variable de construction doit être réglé sur une valeur que si l'environnement de la construction ne dispose pas de cette variable définie Vous pouvez le faire avec la `SetDefault`méthode, qui se comporte de façon similaire à la `set_default`méthode des objets dictionnaire Python:

```
env.SetDefault (SPECIAL_FLAG = '-extra-option')
```

Ceci est particulièrement utile lors de l' écriture de vos propres `Tool`modules pour appliquer des variables à des environnements de construction.

7.2.10. Jusqu'à la fin ajoutant des valeurs: la `Append`méthode

Vous pouvez ajouter une valeur à une variable de construction existante en utilisant la `Append`méthode:

```
env = environnement (CCFLAGS = [ '-DMY_VALUE'])
env.Append (CCFLAGS = [ '-DLAST'])
env.Program ( 'foo.c')
```

SCons fournit alors les deux `-DMY_VALUE`et `-DLAST`drapeaux lors de la compilation du fichier objet:

```
% scons -Q
cc -o -c foo.o -DMY_VALUE -DLAST foo.c
cc -o foo foo.o
```

Si la variable de la construction n'existe pas, la `Append`méthode créer:

```
env = environnement ()
env.Append (nouvelle_variable = 'ajouté')
print "nouvelle_variable =", env [ 'new_variable']
```

Ce qui donne:

```
% scons -Q
New_variable = ajouté
```

```
scons: ` « . est à jour.
```

Notez que la `Append`fonction essaie d'être « intelligent » sur la façon dont la nouvelle valeur est ajoutée à la valeur ancienne. Si les deux sont des chaînes, les chaînes précédentes et les nouvelles sont tout simplement concaténés. De même, si les deux sont des listes, les listes sont concaténés. Cependant, si l' on est une chaîne et l'autre est une liste, la chaîne est ajoutée comme un nouvel élément à la liste.

7.2.11. Valeurs uniques ajoutant: la `AppendUnique`méthode

Quelques fois il est utile d'ajouter une nouvelle valeur que si la variable de la construction existante ne contient pas déjà la valeur. Cela peut être fait en utilisant la `AppendUnique`méthode:

```
env.AppendUnique (CCFLAGS = [ '- g'])
```

Dans l'exemple ci - dessus, le `-g`serait ajouté que si la `$CCFLAGS`variable ne contient pas déjà une `-g`valeur.

7.2.12. Au début ajoutant des valeurs: la `Prepend`méthode

Vous pouvez ajouter une valeur au début d'une variable de construction existante en utilisant la `Prepend`méthode:

```
env = environnement (CCFLAGS = [ '-DMY_VALUE'])
env.Prepend (CCFLAGS = [ '-DFIRST'])
env.Program ( 'foo.c')
```

SCons fournit alors les deux `-DFIRST`et `-DMY_VALUE`drapeaux lors de la compilation du fichier objet:

```
% scons -Q
cc -o -c foo.o -DFIRST -DMY_VALUE foo.c
cc -o foo foo.o
```

Si la variable de la construction n'existe pas, la `Prepend`méthode créer:

```
env = environnement ()
env.Prepend (nouvelle_variable = 'ajouté')
print "nouvelle_variable =", env [ 'new_variable']
```

Ce qui donne:

```
% scons -Q
New_variable = ajouté
scons: ` « . est à jour.
```

Comme la `Append`fonction, la `Prepend`fonction essaie d'être « intelligent » sur la façon dont la nouvelle valeur est ajoutée à la valeur ancienne. Si les deux sont des chaînes, les chaînes précédentes et les nouvelles sont tout simplement concaténés. De même, si les deux sont des listes, les listes sont concaténés. Cependant, si l' on est une chaîne et l'autre est une liste, la chaîne est ajoutée comme un nouvel élément à la liste.

7.2.13. Préfixer Valeurs uniques: la `PrependUnique`méthode

Quelques fois il est utile d'ajouter une nouvelle valeur au début d'une variable de construction que si la valeur actuelle ne contient pas déjà la valeur à être ajoutée. Cela peut être fait en utilisant la `PrependUnique`méthode:

```
env.PrependUnique (CCFLAGS = [ '- g'])
```

Dans l'exemple ci - dessus, le `-g`serait ajouté que si la `$CCFLAGS`variable ne contient pas déjà une `-g`valeur.

7.3. Contrôle de l'environnement d'exécution des commandes émises

Lorsque SCons construit un fichier cible, il n'exécute pas les commandes avec le même environnement extérieur que vous avez utilisé pour exécuter SCons . Au lieu de cela, il utilise le dictionnaire stocké dans la `$ENV`variable de construction comme l'environnement extérieur pour l' exécution des commandes.

Le plus important de ramification ce comportement est que la `PATH`variable d'environnement, qui contrôle où le système d'exploitation recherchera les commandes et les services publics, ne sont pas les mêmes que dans l'environnement extérieur à partir de laquelle vous avez appelé SCons . Cela signifie que SCons ne sera pas, par défaut, trouver nécessairement tous les outils que vous pouvez exécuter à partir de la ligne de commande.

La valeur par défaut de la `PATH`variable d'environnement sur un système POSIX est `/usr/local/bin:/bin:/usr/bin`. La valeur par défaut de la `PATH`variable d'environnement sur un système Windows provient de la valeur de Registre Windows pour l'interpréteur de commandes. Si vous voulez exécuter des commandes - compilateurs, linkers, etc .-- qui ne sont pas dans ces emplacements par défaut, vous devez définir la `PATH`valeur dans le `$ENV`dictionnaire dans votre environnement de construction.

La façon la plus simple de le faire est d'initialiser explicitement la valeur lors de la création de l'environnement de la construction; c'est une façon de le faire:

```
path = [ '/ usr / local / bin', '/ bin', '/ usr / bin']
env = environnement (ENV = { 'PATH': path})
```

Attribuez un dictionnaire à la `$ENV` variable de la construction de cette manière complètement l'environnement remet à zéro externe de sorte que la seule variable qui sera définie lors de commandes externes sont exécutées sera la `PATH`valeur. Si vous voulez utiliser le reste des valeurs `$ENV`et seulement définir la valeur `PATH`, est probablement la façon la plus simple:

```
env [ 'ENV'] [ 'PATH'] = [ '/ usr / local / bin', '/ bin', '/ usr / bin']
```

Notez que SCons ne vous permet de définir les répertoires dans la `PATH`dans une chaîne, séparés par le caractère séparateur pour chemin de votre système (« : » sur les systèmes POSIX, « ; » sous Windows):

```
env [ 'ENV'] [ 'PATH'] = '/ usr / local / bin: / bin: / usr / bin'
```

Mais cela rend votre SConscriptfichier moins portable, (bien que dans ce cas , qui ne peut pas être une préoccupation énorme puisque les répertoires que vous énumérez sont spécifiques au système likley, de toute façon).

7.3.1. Propagation PATHde l'environnement extérieur

Vous pouvez propager l'extérieur PATH à l'environnement d'exécution des commandes. Vous faites cela en initialisant la PATH variable avec la PATHvaleur du os.environ dictionnaire, qui est la manière de Python vous permettant de faire à l'environnement extérieur:

```
import os
env = environnement (ENV = { 'PATH': os.environ [ 'PATH']})
```

Vous pouvez également trouver plus facile de se propager tout l'ensemble de l' environnement extérieur à l'environnement d'exécution des commandes. C'est plus simple à coder que la sélection explicity la PATHvaleur:

```
import os
env = environnement (ENV = os.environ)
```

Ces deux méthodes se garantir que SCons sera en mesure d'exécuter une commande que vous pouvez exécuter à partir de la ligne de commande. L'inconvénient est que la construction peut se comporter différemment si elle est dirigée par des gens avec différentes PATHvaleurs dans leur environnement - par exemple, si les deux /binet/usr/local/binrépertoires ont des cc commandes, alors que l' on sera utilisé pour compiler des programmes dépendra de quel répertoire est répertorié en premier dans l'utilisateur PATHla variable.

7.3.2. Ajout de PATHvaleurs dans l'environnement d'exécution

L' une des exigences les plus communes pour manipuler une variable dans l'environnement d'exécution est d'ajouter un ou plusieurs répertoires personnalisés à une recherche comme la $PATHvariable sur les systèmes Linux ou POSIX, ou la %PATH%variable de Windows, de sorte qu'un compilateur ou un autre utilitaire installé localement peut être trouvée quand SCons tente de l' exécuter pour mettre à jour une cible. SCons offre PrependENVPathet AppendENVPathfonctions pour faire ajouter des choses à des variables d'exécution pratique. Vous appelez ces fonctions en spécifiant la variable à laquelle vous voulez que la valeur ajoutée, et elle - même valorisez. Donc , pour ajouter quelques /usr/localrépertoires aux $PATHet les $LIBvariables, vous pouvez:

```
env = environnement (ENV = os.environ)
env.PrependENVPath ( 'PATH', '/ usr / local / bin')
env.AppendENVPath ( 'LIB', '/ usr / local / lib')
```

Notez que les valeurs ajoutées sont des chaînes, et si vous souhaitez ajouter plusieurs répertoires à une variable comme $PATH, vous devez inclure le chemin caractère séparé (:sous Linux ou Posix, ;sous Windows) dans la chaîne.

Chapitre 8. Mise automatiquement les options de ligne de commande dans leurs variables de construction

Ce chapitre décrit les MergeFlags, ParseFlagset les ParseConfigméthodes d'un construction environment.

8.1. La fusion des options dans l'environnement: la MergeFlagsfonction

SCons environnements de construction ont une MergeFlagsméthode qui fusionne un dictionnaire de valeurs dans l'environnement de la construction. MergeFlagstraite chaque valeur dans le dictionnaire comme une liste d'options telles que l' on peut passer à une commande (par exemple un compilateur ou lieur). MergeFlagsne sera pas dupliquer une option si elle existe déjà dans la variable d'environnement de construction.

MergeFlagsessaie d'être intelligent sur les options de fusion. Lors de la fusion des options à une variable dont le nom se termine par PATH,MergeFlagsconserve l'occurrence de l'extrême gauche option parce que dans les listes typiques de chemins d'accès, la première occurrence « gagne ». Lors de la fusion des options à tout autre nom de variable,MergeFlagsconserve l'apparition de l'option la plus à droite, parce que dans une liste d'options de ligne de commande typique, la dernière occurrence « gagne ».

```
env = environnement ()
env.Append (CCFLAGS = '-option -O3 -O1')
drapeaux = { 'CCFLAGS': '-whatever -O3'}
env.MergeFlags (drapeaux)
env imprimer [ 'CCFLAGS']


% scons -Q
[ '-option', '-O1', '-whatever', '-O3']
scons: ` « . est à jour.
```

Notez que la valeur par défaut $CCFLAGS est un interne SCons objet qui convertit automatiquement les options que nous spécifiées sous la forme d' une chaîne dans une liste.

```
env = environnement ()
env.Append (CPPPATH = [ '/ include', '/ usr / local / include', '/ usr / include'])
drapeaux = { 'CPPPATH': [ '/ usr / opt / include', '/ usr / local / include']}
env.MergeFlags (drapeaux)
env imprimer [ 'CPPPATH']


% scons -Q
[ '/ Include', '/ usr / local / include', '/ usr / include', '/ usr / opt / include']
scons: ` « . est à jour.
```

Notez que la valeur par défaut $CPPPATH est une liste Python normale, donc nous devons préciser ses valeurs comme une liste dans le dictionnaire que nous passons à la MergeFlagsfonction.

Si MergeFlagsest passé autre chose qu'un dictionnaire, il appelle la ParseFlagsméthode pour le convertir en un dictionnaire.

```
env = environnement ()
env.Append (CCFLAGS = '-option -O3 -O1')
env.Append (CPPPATH = [ '/ include', '/ usr / local / include', '/ usr / include'])
env.MergeFlags ( '- quel que soit -I / usr / opt / Include -O3 -I / usr / local / include')
env imprimer [ 'CCFLAGS']
env imprimer [ 'CPPPATH']


% scons -Q
[ '-option', '-O1', '-whatever', '-O3']
[ '/ Include', '/ usr / local / include', '/ usr / include', '/ usr / opt / include']
scons: ` « . est à jour.
```

Dans l'exemple ci - dessus combiné, `ParseFlags`a trié les options dans leurs variables correspondantes et a renvoyé un dictionnaire pour `MergeFlags`appliquer aux variables de construction dans l'environnement de construction spécifié.

8.2. Arguments Compile dans la séparation de leurs variables: la `ParseFlags`fonction

SCons a un nombre ahurissant de variables de construction pour différents types d'options lors de la construction des programmes. Parfois , vous ne pouvez pas savoir exactement quelle variable doit être utilisée pour une option particulière.

SCons environnements de construction ont une `ParseFlags`méthode qui prend un ensemble d'options de ligne de commande typiques et les distrbutes dans les variables de construction appropriées. Par le passé, il a été créé pour soutenir la `ParseConfig`méthode, il met l' accent sur les options utilisées par la GNU Compiler Collection (GCC) pour C et C ++ toolchains.

`ParseFlags`renvoie un dictionnaire contenant les options distribuées dans leurs variables de construction respectives. Normalement, ce dictionnaire serait passé à `MergeFlags` fusionner les options en un `construction environment`, mais le dictionnaire peut être modifié si on le souhaite pour fournir des fonctionnalités supplémentaires. (Notez que si les drapeaux ne vont pas à modifier, appelant `MergeFlags`les options directement éviter une étape supplémentaire.)

```
env = environnement ()
d = env.ParseFlags ( "- I / opt / include -L / opt / lib -lfoo")
pour k, v en ordonnée (d.items ()):
    si v:
        print k, v
env.MergeFlags (d)
env.Program ( 'f1.c')


% scons -Q
CPPPATH [ '/ opt / include']
LIBPATH [ '/ opt / lib']
LIBS [ 'foo']
cc -o f1.o -c -I / opt / include f1.c
cc -o f1 f1.o -L / opt / lib -lfoo
```

Notez que si les options sont limitées à des types génériques comme ceux ci-dessus, ils seront traduits correctement pour d'autres types de plate-forme:

```
C: \>scons -Q
CPPPATH [ '/ opt / include']
LIBPATH [ '/ opt / lib']
LIBS [ 'foo']
cl /Fof1.obj / c f1.c / nologo / I \ opt \ include
lien / nologo /OUT:f1.exe / LIBPATH: \ opt \ lib foo.lib f1.obj
embedManifestExeCheck (cible, source, env)
```

Étant donné que l'hypothèse est que les drapeaux sont utilisés pour la toolchain GCC, drapeaux non reconnus sont placés dans de `$CCFLAGS`sorte qu'ils seront utilisés à la fois C et C ++ compile:

```
env = environnement ()
d = env.ParseFlags ( "-") quel que soit
pour k, v en ordonnée (d.items ()):
    si v:
        print k, v
env.MergeFlags (d)
env.Program ( 'f1.c')


% scons -Q
CCFLAGS -whatever
cc -o -c f1.o -whatever f1.c
cc -o f1 f1.o
```

`ParseFlags`également accepter une liste (récursive) de chaînes en entrée; la liste est aplatie avant que les chaînes sont traitées:

```
env = environnement ()
d = env.ParseFlags ([ "- I / opt / include", [ "-L / opt / lib", "-lfoo"]])
pour k, v en ordonnée (d.items ()):
    si v:
        print k, v
env.MergeFlags (d)
env.Program ( 'f1.c')


% scons -Q
CPPPATH [ '/ opt / include']
LIBPATH [ '/ opt / lib']
LIBS [ 'foo']
cc -o f1.o -c -I / opt / include f1.c
cc -o f1 f1.o -L / opt / lib -lfoo
```

Si une chaîne commence par un « ! » (Un point d'exclamation, souvent appelée bang), la chaîne est passée à la coque pour l'exécution. La sortie de la commande est alors analysé:

```
env = environnement ()
d = env.ParseFlags ([ "! echo -I / opt / include", "! echo -L / opt / lib", "-lfoo"])
```

```
pour k, v en ordonnée (d.items ()):
    si v:
        print k, v
env.MergeFlags (d)
env.Program ( 'f1.c')
```

```
% scons -Q
CPPPATH [ '/ opt / include']
LIBPATH [ '/ opt / lib']
LIBS [ 'foo']
cc -o f1.o -c -I / opt / include f1.c
cc -o f1 f1.o -L / opt / lib -lfoo
```

`ParseFlags`est régulièrement mis à jour pour de nouvelles options; consultez la page de manuel pour plus de détails au sujet de ceux qui sont actuellement reconnus.

8.3. Recherche d' informations Bibliothèque installée: la `ParseConfig`fonction

Configuration des bonnes options pour créer des programmes pour travailler avec les bibliothèques - bibliothèques partagées en particulier - qui sont disponibles sur les systèmes POSIX peut être très compliqué. Pour aider à cette situation, plusieurs utilies avec des noms qui se terminent par `config` retour les options de ligne de commande pour la collection de compilateurs GNU (GCC) qui sont nécessaires pour utiliser ces bibliothèques; par exemple, les options de ligne de commande à utiliser une bibliothèque nommée `lib` serait trouvée en appelant un utilitaire nommé `lib-config`.

Une convention plus récente est que ces options sont disponibles à partir du générique `pkg-config`programme, qui a cadre commun, la gestion des erreurs, etc., pour que le créateur de l' emballage doit faire est de fournir l'ensemble des chaînes pour son paquet particulier.

SCons environnements de construction ont une `ParseConfig`méthode qui exécute un `*config`utilitaire (soit `pkg-config`ou un utilitaire plus spécifique) et configure les variables de construction appropriées dans l'environnement sur la base des options de ligne retournées par la commande spécifiée.

```
env = environnement ()
env [ 'CPPPATH'] = [ '/ lib / compat']
env.ParseConfig ( "pkg-config x11 --cflags --libs")
env imprimer [ 'CPPPATH']
```

SCons exécutera la chaîne de commande spécifiée, analyser les drapeaux qui en résultent, et ajoutez les drapeaux aux variables d'environnement appropriées.

```
% scons -Q
[ '/ Lib / compat', '/ usr / X11 / include']
scons: ` « . est à jour.
```

Dans l'exemple ci - dessus, SCons a ajouté le include à `CPPPATH`. (En fonction de ce que les autres drapeaux sont émis par la `pkg-config`commande, d' autres variables peuvent avoir été étendues aussi bien.)

Notez que les options sont fusionnées avec des options existantes à l' aide de la `MergeFlags`méthode, de sorte que chaque option se produit une seule fois dans la variable de construction:

```
env = environnement ()
env.ParseConfig ( "pkg-config x11 --cflags --libs")
env.ParseConfig ( "pkg-config x11 --cflags --libs")
env imprimer [ 'CPPPATH']
```

```
% scons -Q
[ '/ Usr / X11 / include']
scons: ` « . est à jour.
```

Chapitre 9. Contrôle sortie de la construction

Un aspect clé de la création d' une configuration de construction utilisable fournit un bon rendement de la construction de sorte que ses utilisateurs puissent facilement comprendre ce que la construction fait et obtenir des informations sur la façon de contrôler la construction.SCons fournit plusieurs façons de contrôler la sortie de la configuration de construction pour aider à rendre la construction plus utile et compréhensible.

9.1. Fournir Aide Construire: la `Help`fonction

Il est souvent très utile de pouvoir donner aux utilisateurs une aide qui décrit les objectifs spécifiques, les options de construction, etc., qui peut être utilisé pour votre construction. SCons fournit la `Help`fonction pour vous permettre de spécifier ce texte d'aide:

```
Aidez-moi("""
Type: « programme scons » pour construire le programme de production,
      'Scons debug' pour construire la version de débogage.
"" ")
```

En option, on peut spécifier le drapeau append:

```
Aidez-moi("""
Type: « programme scons » pour construire le programme de production,
      'Scons debug' pour construire la version de débogage.
"" », Append = True)
```

(Notez l'utilisation ci-dessus de la syntaxe triple guillemet Python, ce qui est très pratique pour spécifier des chaînes multi-lignes comme texte d'aide.)

Lorsque le `SConstruct`ou les `SConscript`fichiers contiennent un tel appel à la `Help`fonction, le texte d'aide spécifié sera affiché en réponse à la SCons `-h` Option:

```
% scons -h
scons: Lecture des fichiers SConscript ...
scons: fait la lecture des fichiers SConscript.

Type: « programme scons » pour construire le programme de production,
      'Scons debug' pour construire la version de débogage.

Utilisez scons -H pour l'aide sur les options de ligne de commande.
```

Les `SConscript`fichiers peuvent contenir plusieurs appels à la `Help`fonction, dans ce cas , le texte spécifié (s) sera concaténer lors de l' affichage. Cela vous permet de diviser le texte d'aide sur plusieurs `SConscript`fichiers. Dans ce cas, l'ordre dans lequel les `SConscript`fichiers sont appelés déterminera l'ordre dans lequel les `Help`fonctions sont appelées, qui déterminera l'ordre dans lequel les différents morceaux de texte auront concaténer.

Lorsqu'il est utilisé avec l' `AddOption`aide (« text », append = False) sera écraserait toute sortie d'aide associée à addOption (). Pour préserver la sortie de l' aide de addOption (), définissez append = True.

Une autre utilisation serait de rendre le texte d'aide conditionnelle à une variable. Par exemple, supposons que vous souhaitez seulement afficher une ligne sur la construction d' une version Windows uniquement d'un programme lorsqu'il est exécuté en fait sous Windows. Le suivant `SConstruct`fichier:

```
env = environnement ()

Aide ( "\ nType: 'programme scons' pour construire le programme de production \ n".)

si env [ 'PLATEFORME'] == 'win32':
    Aide ( "\ nType: 'scons WinDebug' pour construire la version de débogage Windows \ n".)
```

Affiche le texte d'aide complète sous Windows:

```
C: \>scons -h
scons: Lecture des fichiers SConscript ...
scons: fait la lecture des fichiers SConscript.

Type: « programme scons » pour construire le programme de production.

Type: 'scons WinDebug' pour construire la version de débogage de Windows.

Utilisez scons -H pour l'aide sur les options de ligne de commande.
```

Mais seulement montrer l'option correspondante sur un système Linux ou UNIX:

```
% scons -h
scons: Lecture des fichiers SConscript ...
scons: fait la lecture des fichiers SConscript.

Type: « programme scons » pour construire le programme de production.

Utilisez scons -H pour l'aide sur les options de ligne de commande.
```

S'il n'y a pas de `Help`texte dans les `SConstruct`ou les `SConscript`fichiers,SCons seront rétrocédés à afficher sa liste standard qui décrit les SCons les options de ligne de commande. Cette liste est également toujours affiché chaque fois que l' `-H`on utilise l' option.

9.2. Contrôle Comment SCons impressions des commandes de construction: les `$*COMSTR`variables

Parfois , les commandes exécutées pour compiler des fichiers d'objets ou des programmes de liaison (ou de construire d' autres cibles) peuvent être très long, assez longtemps pour le rendre difficile pour les utilisateurs de distinguer les messages d'erreur ou autre importante sortie de génération des commandes elles - mêmes. Tous les défaut des `$*COM`variables qui spécifient les lignes de commande utilisées pour construire différents types de fichiers cibles ont un correspondant `$*COMSTR`variable qui peut être définie sur une chaîne alternative qui sera affiché lorsque la cible est construit.

Par exemple, supposons que vous voulez avoir SCons afficher un`"Compiling"` message dès qu'il compile un fichier objet, et `"Linking"`quand il relie un exécutable. Vous pouvez écrire un `SConstruct`fichier qui ressemble à :

```
env = environnement (CCCOMSTR = "$ TARGET Compiler",
                     LINKCOMSTR = "Lier target $")
env.Program ( 'foo.c')
```

Ce qui donnerait alors la sortie:

```
% scons -Q
foo.o compiler
lier foo
```

SCons effectue une substitution complète des variables sur les `$*COMSTR`variables, afin qu'ils aient accès à toutes les variables standard comme `$TARGET` `$SOURCES`, etc., ainsi que toutes les variables de construction qui se trouvent être configuré dans l'environnement de construction utilisé pour construire une cible spécifique.

Bien sûr, parfois , il est toujours important d'être en mesure de voir la commande exacte que SCons exécutera pour construire une cible. Par exemple, vous pouvez tout simplement besoin de vérifier que SCons est configuré pour fournir les bonnes options au compilateur ou un développeur peut vouloir couper-coller une commande de compilation pour ajouter quelques options pour un test personnalisé.

Une façon courante de donner le contrôle aux utilisateurs sur si oui ou non SCons doivent imprimer la ligne de commande réelle ou un court, résumé configuré est d'ajouter le support pour une `VERBOSE` variable de commande en ligne à votre `SConstruct`fichier. Une configuration simple pour cela pourrait ressembler:

```
env = environnement ()
si ARGUMENTS.get ( 'verbose') = « 1' :
    env [ 'CCCOMSTR'] = "$ TARGET Compiler"
    env [ 'LINKCOMSTR'] = "Lier $ TARGET"
env.Program ( 'foo.c')
```

En réglant uniquement les appropriés des `$*COMSTR`variables si l'utilisateur spécifie `VERBOSE=1` sur la ligne de commande, l'utilisateur a le contrôle sur la façon dont SCons affiche ces lignes de commande particulières:

```
% scons -Q
foo.o compiler
lier foo
% scons -Q -c
Suppression foo.o
Suppression de foo
% scons -Q VERBOSE=1
cc -o -c foo.o foo.c
cc -o foo foo.o
```

9.3. Fournir sortie de la construction Progrès: la `Progress`fonction

Un autre aspect de fournir une sortie de bonne construction est de donner les commentaires des utilisateurs sur ce que SCons fait même quand rien ne se construit actuellement. Cela peut être particulièrement vrai pour les grandes builds lorsque la plupart des cibles sont déjà à jour. Parce que SCons peut prendre beaucoup de temps à faire absolument sûr que chaque cible est, en fait, la mise à jour par rapport à un grand nombre de fichiers de dépendance, il peut être facile pour les utilisateurs de conclure à tort que SCons est suspendu ou qu'il ya un certain autre problème avec la construction.

Une façon de faire face à cette perception est de configurer SCons pour imprimer quelque chose à laisser l'utilisateur de savoir ce qu'il « y penser. » La `Progress`fonction vous permet de spécifier une chaîne qui sera imprimée pour chaque fichier que SCons « envisage » alors qu'il parcourt le graphe de dépendance de décider quels sont les objectifs sont ou ne sont pas mises à jour.

```
Progrès ( 'évaluation $ TARGET \ n')
Programme ( 'f1.c')
Programme ( 'f2.c')
```

Notez que la `Progress`fonction ne prévoit pas d'une nouvelle ligne à imprimer automatiquement à la fin de la chaîne (comme le fait Python`print`déclaration), et nous devons préciser la `\n` que nous voulons présentées à la fin de la chaîne configurée. Cette configuration, alors, aura SCons impression qu'il est `Evaluating` chaque fichier qu'il rencontre à son tour comme il parcourt le graphe de dépendance:

```
% scons -Q
L'évaluation SConstruct
L'évaluation f1.c
L'évaluation f1.o
cc -o -c f1.o f1.c
L'évaluation f1
cc -o f1 f1.o
L'évaluation f2.c
L'évaluation f2.o
cc -o -c f2.o f2.c
L'évaluation f2
cc -o f2 f2.o
L'évaluation.
```

Bien sûr, normalement vous ne voulez pas ajouter toutes ces lignes supplémentaires à votre sortie de construction, car cela peut rendre difficile pour l'utilisateur de trouver des erreurs ou d' autres messages importants. Une façon d'afficher ces progrès plus utile peut - être d'avoir les noms de fichiers imprimés directement sur l'écran de l' utilisateur, pas au même flux de sortie standard où la production de construction est imprimé, et d'utiliser un caractère de retour chariot (`\r`) de telle sorte que chaque nom de fichier est ré-imprimé sur la même ligne. Une telle configuration ressemblerait à ceci:

```
Progrès ( 'TARGET $ \ r',
         file = open ( '/ dev / tty', 'w'),
         remplacer = True)
Programme ( 'f1.c')
Programme ( 'f2.c')
```

Notez que nous avons spécifié également l' `overwrite=True`argument de la `Progress`fonction, ce qui provoque SCons à « effacer » la chaîne précédente avec des caractères de l' espace avant d' imprimer la prochaine `Progress`chaîne. Sans l' `overwrite=True`argument un nom de fichier plus court ne serait pas remplacer tous les caractères que dans un nom de fichier plus qui le précède, ce qui rend difficile de dire ce que le nom du fichier est sur la sortie. Notez également que nous avons ouvert le `/dev/tty`fichier pour un accès direct (sur POSIX) à l'écran de l'utilisateur. Sous Windows, l'équivalent serait d'ouvrir le `con:`nom du fichier.

En outre, il est important de savoir que même si vous pouvez utiliser`$TARGET`pour remplacer le nom du nœud dans la chaîne, la `Progress`fonction ne *pas* effectuer la substitution de variable générale (parce qu'il n'y a pas nécessairement un environnement de construction impliqué dans l' évaluation d' un noeud comme un fichier source, par exemple).

Vous pouvez également spécifier une liste de chaînes à la `Progress`fonction, auquel cas SCons affichera chaque chaîne à son tour. Cela peut être utilisé pour mettre en œuvre un « spinner » en ayant SCons faire défiler une séquence de chaînes:

```
Progrès ([ '- \ r', 'r \\\', '| \ r', '/ \ r'], intervalle = 5)
Programme ( 'f1.c')
Programme ( 'f2.c')
```

Notez que là , nous avons également utilisé l' `interval=` argument de mot - clé pour avoir SCons seulement imprimer une nouvelle chaîne « spinner » une fois tous les cinq noeuds évalués. L' utilisation d' un `interval=`nombre, même avec des chaînes qui utilisent `$TARGET`comme nos exemples ci - dessus, peut être une bonne façon de réduire le travail que SCons consacre l' impression des `Progress`chaînes, tout en donnant les commentaires des utilisateurs qui indique SCons travaille toujours sur l' évaluation de la construction.

Enfin, vous pouvez avoir un contrôle direct sur la façon d'imprimer chaque noeud évalué en passant une fonction Python (ou autre Python appelable) à la `Progress`fonction. Votre fonction sera appelée pour chaque noeud évalué, ce qui vous permet de mettre en œuvre une logique plus sophistiquée comme l' ajout d' un compteur:

```
écran = open ( '/ dev / tty', 'w')
count = 0
def progress_function (noeud)
```

```
    compter + = 1
    screen.write ( 'Node% 4d:% s \ r' % (chiffre, noeud))

Progress (progress_function)
```

Bien sûr, si vous choisissez, vous pouvez ignorer complètement l'`node`argument de la fonction, et juste imprimer un compte, ou tout ce que vous souhaitez.

(Notez qu'il ya une question de suivi sur évident ici: comment voulez - vous trouver le nombre total de noeuds qui *seront* ? Évalués afin que vous puissiez dire à l'utilisateur la proximité de la construction est de terminer Malheureusement, dans le cas général, il n'y a pas une bonne façon de le faire, à court d'avoir SCons évaluer son graphe de dépendance deux fois, d' abord à compter du total et la deuxième fois pour construire effectivement les cibles. Cela serait nécessaire parce que vous ne pouvez pas savoir à l' avance qui cible (s) du utilisateur effectivement demandé à construire. la construction entière peut être constitué de milliers de nœuds, par exemple, mais peut - être l'utilisateur spécifiquement demandé un seul fichier d'objet construit.)

9.4. Impression détaillée Statut: la `GetBuildFailures`fonction

SCons, comme la plupart des outils à construire, renvoie l'état zéro à la coquille sur le succès et le statut non nul en cas d'échec. Parfois, il est utile de donner plus d'informations sur l'état de construction à la fin de la course, par exemple pour imprimer un message d'information, envoyez un e-mail ou une page les pauvres plouc qui a cassé la construction.

SCons fournit une `GetBuildFailures`méthode que vous pouvez utiliser dans un python `atexit`fonction pour obtenir une liste d'objets décrivant les actions qui ont échoué en essayant de construire des cibles. Il peut y avoir plus d'un si vous utilisez `-j`. Voici un exemple simple:

```
importation atexit

def print_build_failures ():
    de GetBuildFailures à l'importation SCons.Script
    pour bf dans GetBuildFailures ():
        print "% a échoué de:% s" % (bf.node, bf.errstr)
atexit.register (print_build_failures)
```

Les `atexit.register`registres d'appel `print_build_failures` comme un `atexit`rappel, avant d'être appelé SCons sorties. Lorsque cette fonction est appelée, elle appelle `GetBuildFailures`pour aller chercher la liste des objets ayant échoué. Voir la page de manuel pour le contenu détaillé des objets retournés; certains des attributs les plus utiles sont `.node`,`.errstr`, `.filename`et `.command`. Le `filename`est pas nécessairement le même fichier que `node`; la `node`est la cible qui a été construit lorsque l'erreur est survenue, alors que le `filename`est le fichier ou le répertoire qui a effectivement causé l'erreur. Remarque: appeler uniquement `GetBuildFailures`à la fin de la construction; appelant à tout autre moment est indéfini.

Voici un exemple plus complet montrant comment transformer chaque élément `GetBuildFailures`dans une chaîne:

```
# Faire la construction échoue si on passe fail = 1 sur la ligne de commande
si ARGUMENTS.get ( 'échec', 0):
    Commande ( 'cible', 'source', [ '/ bin / false'])

def bf_to_str (bf):
    « » "Convertir un élément de GetBuildFailures () à une chaîne
    de manière utile. « » »
    SCons.Errors d'importation
    si bf est: Aucun # cibles inconnues produit Aucune liste
        retour '(tgt inconnu)'
    elif isinstance (bf, SCons.Errors.StopError):
        retour str (bf)
    Elif bf.node:
        retourner str (bf.node) + ':' + bf.errstr
    Elif bf.filename:
        retour bf.filename + ':' + bf.errstr
    retour « échec inconnu: » + bf.errstr
importation atexit

def build_status ():
    "" "Convertir l'état de construction à un 2-tuple (état, msg)." ""
    de GetBuildFailures à l'importation SCons.Script
    bf = GetBuildFailures ()
    si bf:
        # Bf est normalement une liste des échecs de construction; si un élément est Aucun,
        # C'est à cause d'une cible qui scons ne sait rien.
        status = 'échoué'
        failures_message = "\ n" .join ([ "Echec de l'édifice%" % de bf_to_str (x)
                          pour x dans bf si x est None])
    autre:
        # Si bf est pas, la construction terminée avec succès.
        status = 'ok'
        failures_message = ''
    retour (statut, failures_message)

def display_build_status ():
    « » "Afficher l'état de construction. Appelée par atexit.
    Ici vous pouvez faire toutes sortes de choses compliquées. « » »
    statut, failures_message = build_status ()
    si le statut == « a échoué »:
       print "ECHEC !!!!" # Pourrait afficher alerte, sonnez, etc.
    état Elif == 'ok':
       print "Build a réussi."
    Imprimer failures_message

atexit.register (display_build_status)
```

Lorsque cela fonctionne, vous verrez la sortie appropriée:

```
% scons -Q
scons: ` « . est à jour.
Construire réussi.
% scons -Q fail=1
scons: *** [cible] Source `la source 'introuvable, requis par la cible` cible.
```

```
ÉCHOUÉ!!!!
cible de construction a échoué: Source `la source 'introuvable, requis par la cible` cible.
```

Chapitre 10. Contrôle de construire à partir de la ligne de commande

SCons fournit un certain nombre de façons pour l'auteur des `SConscript`fichiers pour donner aux utilisateurs qui exécuteront SCons beaucoup de contrôle sur l'exécution de la construction. Les arguments que l'utilisateur peut spécifier sur la ligne de commande sont réparties en trois types:

options

> Options de ligne de commande commencent toujours par un ou deux `-`(trait d' union) caractères. SCons fournit des moyens pour vous d'examiner et de définir des valeurs d'options à partir de vos `SConscript`fichiers, ainsi que la possibilité de définir vos propres options personnalisées. Voir la section 10.1, « Options de ligne de commande » , ci - dessous.

Variables

> Tout argument de ligne de commande contenant un `=` (signe égal) est considéré comme un paramètre variable par la forme`variable= value`. SCons offre un accès direct à tous les paramètres variables de ligne de commande, la possibilité d'appliquer des paramètres variables de ligne de commande pour les environnements de construction, et les fonctions de configuration de types de variables spécifiques (valeurs booléennes, les noms de chemin, etc.) avec validation automatique du les valeurs spécifiées de l' utilisateur. Voir la section 10.2, « la ligne de commande `variable= value`Variables de construction » , ci - dessous.

cibles

> Tout argument de ligne de commande qui n'est pas une option ou un paramètre variable (ne commence pas par un trait d' union et ne contient pas un signe égal) est considéré comme une cible que l'utilisateur (probablement) veut SCons de construire. Une liste d'objets Node représentant la cible ou des cibles à construire. SCons donne accès à la liste des cibles spécifiées, ainsi que les moyens pour définir la liste par défaut des cibles à l' intérieur des `SConscript`fichiers. Voir la section 10.3, « Cibles de ligne de commande » , ci - dessous.

10.1. Options de ligne de commande

SCons a beaucoup d' *options de ligne de commande* qui contrôlent son comportement. Une SCons l' *option de ligne de commande* commence toujours par un ou deux `-`(trait d' union) caractères.

10.1.1. Ne pas avoir à spécifier des options de ligne de commande à chaque fois: la `SCONSFLAGS`variable d' environnement

Les utilisateurs peuvent se trouver fournir les mêmes options de ligne de commande à chaque fois qu'ils courent SCons . Par exemple, vous pouvez trouver un gain de temps de spécifier une valeur de `-j 2` avoir SCons peuvent durer jusqu'à deux commandes de construction en parallèle. Pour éviter d' avoir à taper `-j 2`à la main chaque fois, vous pouvez définir la variable d'environnement externe `SCONSFLAGS`à une chaîne contenant les options de ligne de commande que vous souhaitez SCons à utiliser.

Si, par exemple, vous utilisez un shell POSIX qui est compatible avec le shell Bourne, et vous voulez toujours SCons d'utiliser l' `-Q`option, vous pouvez définir l' `SCONSFLAGS` environnement comme suit:

```
% scons
scons: Lecture des fichiers SConscript ...
scons: fait la lecture des fichiers SConscript.
scons: objectifs de construction ...
    ... [sortie de la construction] ...
scons: fait des objectifs de construction.
% export SCONSFLAGS="-Q"
%scons
    ... [sortie de la construction] ...
```

Les utilisateurs de csh coquilles -style sur les systèmes POSIX peuvent définir l' `SCONSFLAGS`environnement comme suit:

```
$ setenv SCONSFLAGS "-Q"
```

Les utilisateurs de Windows peuvent généralement vouloir régler la`SCONSFLAGS`dans l'onglet approprié de la `System Properties`fenêtre.

10.1.2. Obtenir des valeurs définis par des options de ligne de commande: la `GetOption`fonction

SCons fournit la `GetOption`fonction pour obtenir les valeurs définies par les différentes options de ligne de commande. Une utilisation courante de c'est de vérifier si le ou non `-h`ou `--help`option a été spécifiée. Normalement, SCons n'imprime pas son texte d'aide qu'après avoir lu tous les `SConscript`fichiers, car il est possible que le texte d'aide a été ajoutée par une filiale `SConscript`fichier profond dans la hiérarchie de l' arborescence source. Bien sûr, la lecture de tous les `SConscript`fichiers prend du temps.

Si vous savez que votre configuration ne définit aucun texte d'aide supplémentaire dans la filiale `SConscript`fichiers, vous pouvez accélérer l'aide en ligne de commande disponible pour les utilisateurs en utilisant la `GetOption`fonction pour charger la filiale `SConscript`fichiers uniquement si l'utilisateur n'a *pas* spécifié la `-h`ou l' `--help`option, ainsi:

```
sinon GetOption ( 'aide'):
    SConscript ( 'src / SConscript', export = 'env')
```

En général, la chaîne que vous passez à la `GetOption`fonction pour récupérer la valeur d'un paramètre d'option de ligne de commande est le même que le nom long option « la plus commune » (en commençant par deux caractères trait d'union), bien qu'il existe quelques exceptions. La liste des SCons les options de ligne de commande et les `GetOption`chaînes pour aller les chercher, sont disponibles dans la section 10.1.4, « Cordes pour obtenir ou définir des valeurs de SCons de ligne de commande Options » section ci - dessous.

10.1.3. Définition des valeurs des options de ligne de commande: la `SetOption`fonction

Vous pouvez également définir les valeurs de SCons les options de ligne de commande à l' intérieur des `SConscript`fichiers en utilisant la `SetOption`fonction. Les chaînes que vous utilisez pour définir les valeurs de SCons les options de ligne de commande sont disponibles dans lasection 10.1.4, « Cordes pour obtenir ou définir des valeurs de SCons options de ligne de commande » section ci - dessous.

Une utilisation de la `SetOption`fonction est de spécifier une valeur pour la `-j` ou `--jobs`option afin que les utilisateurs obtiennent l'amélioration des performances d'une compilation parallèle sans avoir à spécifier l'option manuellement. Un facteur de complication est que une bonne valeur pour l' `-j`option est peu dépendante du système. Une ligne directrice grosso modo que les processeurs de votre système est élevé, plus vous voulez définir la `-j`valeur, afin de tirer profit du nombre de processeurs.

Par exemple, supposons que les administrateurs de vos systèmes de développement ont normalisé sur la définition d' une `NUM_CPU`variable d'environnement au nombre de processeurs sur chaque système. Un peu de code Python pour accéder à la variable d'environnement et de la `SetOption`fonction fournissent le niveau de flexibilité:

```
import os
num_cpu = int (os.environ.get ( 'NUM_CPU', 2))
SetOption ( 'num_jobs', num_cpu)
print "en cours d'exécution avec -j", GetOption ( 'num_jobs')
```

L'extrait ci - dessus de code définit la valeur de l' `--jobs`option pour la valeur spécifiée dans la `$NUM_CPU`variable d'environnement. (Ceci est l' un des cas d'exception où la chaîne est orthographié différemment de la possibilité de commande en ligne. La chaîne pour aller chercher ou le réglage de la `--jobs` valeur `num_jobs` pour des raisons historiques.) Le code dans cet exemple imprime la `num_jobs` valeur à des fins d' illustration. Il utilise une valeur par défaut de `2` fournir un certain parallélisme minimal même sur des systèmes à processeur unique:

```
% scons -Q
fonctionnant avec -j 2
scons: ` « . est à jour.
```

Mais si la `$NUM_CPU` variable d'environnement est définie, nous utilisons que pour le nombre par défaut d'emplois:

```
% export NUM_CPU="4"
%scons -Q
fonctionnant avec -j 4
scons: ` « . est à jour.
```

Mais toute forme explicite `-j`ou `--jobs` valeur que l'utilisateur spécifie une ligne de commande est utilisée d' abord, peu importe si oui ou non la `$NUM_CPU`variable d'environnement est définie:

```
% scons -Q -j 7
fonctionnant avec -j 7
scons: ` « . est à jour.
% export NUM_CPU="4"
%scons -Q -j 3
fonctionnant avec -j 3
scons: ` « . est à jour.
```

10.1.4. Cordes pour obtenir ou définir des valeurs de SCons options de ligne de commande

Les chaînes que vous pouvez passer aux `GetOption` et `SetOption`fonctions correspondent généralement au premier nom de l' option de longue durée (en commençant par deux caractères trait d'union: `--`), après avoir remplacé tous les caractères restants avec trait d' union underscores.

La liste complète des chaînes et les variables auxquelles elles correspondent est la suivante:

Chaîne pour `GetOption`et`SetOption`	**Option de ligne de commande (s)**
`cache_debug`	`--cache-debug`
`cache_disable`	`--cache-disable`
`cache_force`	`--cache-force`
`cache_show`	`--cache-show`
`clean`	`-c`, `--clean`, `--remove`
`config`	`--config`
`directory`	`-C`, `--directory`
`diskcheck`	`--diskcheck`
`duplicate`	`--duplicate`
`file`	`-f`, `--file`, `--makefile` , `--sconstruct`
`help`	`-h`, `--help`
`ignore_errors`	`--ignore-errors`
`implicit_cache`	`--implicit-cache`
`implicit_deps_changed`	`--implicit-deps-changed`
`implicit_deps_unchanged`	`--implicit-deps-unchanged`
`interactive`	`--interact`, `--interactive`
`keep_going`	`-k`, `--keep-going`
`max_drift`	`--max-drift`
`no_exec`	`-n`, `--no-exec`, `--just-print`, `--dry-run`, `--recon`
`no_site_dir`	`--no-site-dir`
`num_jobs`	`-j`, `--jobs`
`profile_file`	`--profile`
`question`	`-q`, `--question`
`random`	`--random`
`repository`	`-Y`, `--repository`, `--srcdir`
`silent`	`-s`, `--silent`, `--quiet`
`site_dir`	`--site-dir`
`stack_size`	`--stack-size`
`taskmastertrace_file`	`--taskmastertrace`
`warn`	`--warn --warning`

10.1.5. Ajout d' options de ligne de commande personnalisée: la `AddOption`fonction

SCons vous permet également de définir vos propres options en ligne de commande avec la `AddOption`fonction. La `AddOption`fonction prend les mêmes arguments que la `optparse.add_option`fonction de la bibliothèque standard Python. [3] Une fois que vous avez ajouté une option de ligne de commande personnalisée avec la `AddOption`fonction, la valeur de l'option (le cas échéant) est immédiatement disponible en utilisant la norme `GetOption`fonction. (La valeur peut également être définie à l' aide `SetOption`, bien que ce n'est pas très utile dans la pratique , car une valeur par défaut peut être spécifiée directement dans l' `AddOption`appel.)

Un exemple utile d'utiliser cette fonctionnalité est de fournir un `--prefix`pour les utilisateurs:

```
AddOption ( '- préfixe',
          dest = 'préfixe',
          type 'chaîne' =,
          nargs = 1,
          Action = 'magasin',
          metavar = 'DIR',
          help 'préfixe d'installation' =)

env = environnement (PREFIX = GetOption ( 'préfixe'))

installed_foo = env.Install ( '$ PREFIX / usr / bin', 'foo.in')
Par défaut (installed_foo)
```

Le code ci - dessus utilise la `GetOption`fonction pour définir la `$PREFIX`variable de construction à une valeur spécifiée par l'utilisateur avec une option de ligne de commande `--prefix`. Parce que `$PREFIX` se dilatent à une chaîne vide si elle n'est pas initialisé, en cours d' exécution SCons sans l'option de `--prefix` installera le fichier dans le`/usr/bin/`répertoire:

```
% scons -Q -n
Fichier d'installation: « foo.in » comme « /usr/bin/foo.in »
```

Mais spécifiant `--prefix=/tmp/install` sur la ligne de commande , le fichier doit être installé dans le `/tmp/install/usr/bin/`répertoire:

```
% scons -Q -n --prefix=/tmp/install
Fichier d'installation: « foo.in » comme « /tmp/install/usr/bin/foo.in »
```

10.2. Commande en ligne `variable`= `value`Variables de construction

Vous pouvez contrôler divers aspects de votre produit en permettant à l'utilisateur de spécifier `variable= value` valeurs sur la ligne de commande. Par exemple, supposons que vous souhaitez que les utilisateurs soient en mesure de construire une version de débogage d'un programme en exécutant SCons comme suit:

```
% scons -Q debug=1
```

SCons fournit un `ARGUMENTS`dictionnaire qui stocke toutes les `variable= value` missions de la ligne de commande. Cela vous permet de modifier les aspects de votre construction en réponse aux spécifications sur la ligne de commande. (Notez que si vous voulez exiger que les utilisateurs *toujours* spécifier une variable, vous voulez probablement utiliser Python `ARGUMENTS.get()`fonction qui vous permet de spécifier une valeur par défaut à utiliser s'il n'y a aucune spécification sur la ligne de commande.)

Le code suivant définit la `$CCFLAGS`variable de la construction en réponse au `debug` drapeau étant défini dans le `ARGUMENTS`dictionnaire:

```
env = environnement ()
debug = ARGUMENTS.get ( 'debug', 0)
si int (debug):
    env.Append (CCFLAGS = '-g')
env.Program ( 'prog.c')
```

Cela se traduit par l' `-g` option de compilation utilisés lorsque `debug=1`est utilisé sur la ligne de commande:

```
% scons -Q debug=0
cc -o -c prog.o prog.c
cc -o prog prog.o
% scons -Q debug=0
scons: ` « . est à jour.
% scons -Q debug=1
cc -o prog.o -c -g prog.c
cc -o prog prog.o
% scons -Q debug=1
scons: ` « . est à jour.
```

Notez que SCons garde la trace des dernières valeurs utilisées pour construire les fichiers objet, et par conséquent correctement reconstruit les fichiers objet et exécutables uniquement lorsque la valeur de l' `debug` argument a changé.

Le `ARGUMENTS`dictionnaire a deux inconvénients mineurs. Tout d' abord, parce qu'il est un dictionnaire, il ne peut stocker une valeur pour chaque mot - clé spécifié, et donc seulement « se souvient » le dernier réglage pour chaque mot - clé sur la ligne de commande. Cela rend le `ARGUMENTS`dictionnaire inapproprié si les utilisateurs devraient être en mesure de spécifier plusieurs valeurs sur la ligne de commande pour un mot clé donné. En second lieu , il ne conserve pas l'ordre dans lequel ont été spécifiés les paramètres variables, ce qui est un problème si vous souhaitez que la configuration se comporter différemment en réponse à l'ordre dans lequel les paramètres de construction variables ont été spécifiés sur la ligne de commande.

Pour accueillir ces exigences, SCons fournit une `ARGLIST`variable qui vous donne un accès direct à `variable= value` paramètres sur la ligne de commande, dans l'ordre exact où ils ont été spécifiés, et sans supprimer les paramètres en double. Chaque élément de la `ARGLIST`variable elle - même est une liste à deux éléments contenant le mot - clé et la valeur du paramètre, et vous devez boucle à travers, ou autrement choisir, les éléments `ARGLIST`pour traiter les paramètres spécifiques que vous voulez de quelque manière qui est approprié pour votre configuration. Par exemple, le code suivant pour laisser l'utilisateur d' ajouter à la `CPPDEFINES`variable de construction en spécifiant plusieurs `define=` paramètres sur la ligne de commande:

```
cppdefines = []
pour la clé, la valeur en argList:
    si la clé == « définir »:
        cppdefines.append (valeur)
```

```
env = environnement (CPPDEFINES = cppdefines)
env.Object ( 'prog.c')
```

Donne le résultat suivant:

```
% scons -Q define=FOO
cc -o -c prog.o -DFOO prog.c
% scons -Q define=FOO define=BAR
cc -o -c prog.o -DFOO -DBAR prog.c
```

Notez que les ARGLISTet les ARGUMENTS variables ne gênent pas les uns des autres, mais ne fait que fournir des vues légèrement différentes sur la façon dont l'utilisateur a spécifié variable= value paramètres sur la ligne de commande. Vous pouvez utiliser les deux variables dans la même SCons configuration. En général, le ARGUMENTSdictionnaire est plus pratique à utiliser, (puisque vous pouvez simplement récupérer les paramètres variables grâce à un accès au dictionnaire), et la ARGLISTliste est plus souple (puisque vous pouvez examiner l'ordre spécifique dans lequel les paramètres de ligne de commande de l'utilisateur variabe).

10.2.1. Commande en ligne Controlling Variables de construction

Etre capable d'utiliser une variable construction de ligne de commande comme debug=1est à portée de main, mais il peut être une corvée d'écrire du code Python spécifique pour reconnaître chacun de ces variables, vérifier les erreurs et fournir des messages appropriés, et appliquer les valeurs à une variable de construction. Pour aider à cela,SCons soutient une classe pour définir des variables telles construire facilement, et un mécanisme pour appliquer les variables de construction à un environnement de construction. Cela vous permet de contrôler la façon dont les variables de construction affectent les environnements de construction.

Par exemple, supposons que vous souhaitez que les utilisateurs de définir une RELEASEvariable de construction sur la ligne de commande lorsque vient le temps de construire un programme de mise en liberté, et que la valeur de cette variable doit être ajoutée à la ligne de commande avec l'appropriée -Doption (ou autre commande option de ligne) pour transmettre la valeur au compilateur C. Voici comment vous pouvez le faire en définissant la valeur appropriée dans un dictionnaire pour la $CPPDEFINESvariable de construction:

```
vars = Variables (None, arguments)
vars.Add ( 'RELEASE', 'Mis à 1 pour construire la libération', 0)
env = environnement (variables = vars,
                CPPDEFINES = { 'RELEASE_BUILD': '$ {} REJET'})
env.Program ([ 'foo.c', 'bar.c'])
```

Ce SConstructfichier crée d' abord un Variablesobjet qui utilise les valeurs de la ligne de commande options dictionnaire ARGUMENTS (l' vars = Variables(None, ARGUMENTS)appel). Il utilise ensuite l'objet de Add la méthode pour indiquer que la RELEASE variable peut être définie sur la ligne de commande, et que sa valeur par défaut est 0 (le troisième argument de la Addméthode). Le second argument est une ligne de texte d'aide; nous apprendrons comment l'utiliser dans la section suivante.

Nous passons alors créé Variables objet comme un variablesargument de mot - clé à l' Environmentappel utilisé pour créer l'environnement de construction. Ceci permet alors à un utilisateur de définir laRELEASEvariable de compilation sur la ligne de commande et ont le montrent variables dans la ligne de commande utilisée pour construire chaque objet à partir d' un fichier source C:

```
% scons -Q RELEASE=1
cc -o -c bar.o -DRELEASE_BUILD = 1 bar.c
cc -o -c foo.o -DRELEASE_BUILD = 1 foo.c
cc -o foo foo.o bar.o
```

REMARQUE: Avant SCons libérer 0.98.1, ces construire variables ont été connues comme « options de construction de ligne de commande. » La classe a été effectivement nommé la Optionsclasse, et dans les sections ci - dessous, les différentes fonctions ont été nomméesBoolOption, EnumOption, ListOption, PathOption, PackageOptionet AddOptions. Ces anciens noms fonctionnent encore, et vous pouvez les rencontrer dans les anciens SConscriptfichiers, mais ils ont été officiellement OBSOLETE SCons version 2.0.

10.2.2. Aide pour la fourniture Créer ligne de commande Variables

Pour construire la ligne de commande variables les plus utiles, vous voulez idéalement fournir un texte d'aide qui décrira les variables disponibles lorsque l'utilisateur exécute scons -h. Vous pouvez écrire ce texte à la main, mais SCons fournit un moyen plus facile.Variablesobjets prennent en charge une GenerateHelpTextméthode qui, comme son nom l' indique, générer un texte qui décrit les différentes variables qui ont été ajoutées. Vous passez ensuite la sortie de cette méthode à la Helpfonction:

```
vars = Variables (None, arguments)
vars.Add ( 'RELEASE', 'Mis à 1 pour construire la libération', 0)
env = environnement (variable = vars)
Aide (vars.GenerateHelpText (env))
```

SCons affiche maintenant un texte utile lorsque l' -hon utilise l' option:

```
% scons -Q -h

DE PRESSE: Mis à 1 pour construire la libération
    par défaut: 0
    réelle: 0

Utilisez scons -H pour l'aide sur les options de ligne de commande.
```

Notez que la sortie d'aide indique la valeur par défaut, et la valeur réelle actuelle de la variable de construction.

10.2.3. Lecture Variables de construction partir d'un fichier

Donner à l'utilisateur un moyen de spécifier la valeur d'une variable de construction sur la ligne de commande est utile, mais peut encore être fastidieux si les utilisateurs doivent spécifier à chaque fois que la variable qu'ils courent SCons . Nous pouvons permettre aux utilisateurs de fournir des paramètres variables de construction personnalisés dans un fichier local en fournissant un nom de fichier lorsque nous créons l' Variablesobjet:

```
vars = Variables ( 'custom.py')
vars.Add ( 'RELEASE', 'Mis à 1 pour construire la libération', 0)
env = environnement (variables = vars,
                CPPDEFINES = { 'RELEASE_BUILD': '$ {} REJET'})
env.Program ([ 'foo.c', 'bar.c'])
```

```
Aide (vars.GenerateHelpText (env))
```

Cela permet ensuite à l'utilisateur de contrôler la `RELEASE` variable en lui dans le `custom.py`fichier:

```
DE PRESSE = 1
```

Notez que ce fichier est exécuté comme un script Python. Maintenant , quand nous courons SCons :

```
% scons -Q
cc -o -c bar.o -DRELEASE_BUILD = 1 bar.c
cc -o -c foo.o -DRELEASE_BUILD = 1 foo.c
cc -o foo foo.o bar.o
```

Et si l' on change le contenu `custom.py`à:

```
DE PRESSE = 0
```

Les fichiers objet sont reconstruits correctement avec la nouvelle variable:

```
% scons -Q
cc -o -c bar.o -DRELEASE_BUILD = 0 bar.c
cc -o -c foo.o -DRELEASE_BUILD = 0 foo.c
cc -o foo foo.o bar.o
```

Enfin, vous pouvez combiner les deux méthodes avec:

```
vars = Variables ( 'custom.py', arguments)
```

où les valeurs dans le fichier d'options `custom.py`seront écrasés par ceux spécifiés sur la ligne de commande.

10.2.4. Prédéfinis Fonctions variables Construisons

SCons fournit un certain nombre de fonctions qui fournissent des comportements prêts à l'emploi pour les différents types de commande en ligne construire des variables.

10.2.4.1. Vrai / Faux: Les valeurs de la `BoolVariable`construction Fonction variable

Il est souvent utile de pouvoir spécifier une variable qui contrôle d' une simple variable booléenne avec une `true`ou `false`valeur. Il serait encore plus pratique pour accueillir les utilisateurs qui ont des préférences différentes sur la façon de représenter `true`ou de `false`valeurs. La `BoolVariable`fonction rend facile à ces représentations communes accueillir de `true`ou `false`.

La `BoolVariable`fonction prend trois arguments: le nom de la variable de construction, la valeur par défaut de la variable de construction, et la chaîne d'aide pour la variable. Il retourne ensuite les informations appropriées pour passer à la `Add`méthode d'un `Variables`objet, comme suit:

```
vars = Variables ( 'custom.py')
vars.Add (BoolVariable ( 'RELEASE', 'Ensemble pour construire la libération', 0))
env = environnement (variables = vars,
                  CPPDEFINES = { 'RELEASE_BUILD': '$ {} REJET'})
env.Program ( 'foo.c')
```

Avec cette variable de construction, la `RELEASE`variable peut maintenant être activé par la mise à la valeur `yes` ou `t`:

```
% scons -Q RELEASE=yes foo.o
cc -o foo.o -c -DRELEASE_BUILD = True foo.c

% scons -Q RELEASE=t foo.o
cc -o foo.o -c -DRELEASE_BUILD = True foo.c
```

D' autres valeurs qui assimilent à `true`comprennent `y`, `1`, `on` et `all`.

A l' inverse, `RELEASE`peut maintenant être donné une `false` valeur en mettant à `no` ou `f`:

```
% scons -Q RELEASE=no foo.o
cc -o foo.o -c -DRELEASE_BUILD = False foo.c

% scons -Q RELEASE=f foo.o
cc -o foo.o -c -DRELEASE_BUILD = False foo.c
```

D' autres valeurs qui assimilent à `false`comprennent `n`, `0`, `off` et `none`.

Enfin, si un utilisateur tente de préciser une autre valeur, SConsfournit un message d'erreur approprié:

```
% scons -Q RELEASE=bad_value foo.o

scons: *** Erreur de conversion Option: RELEASE
Valeur non valide pour une option booléenne: bad_value
Fichier "/ home / mon / projet / SConstruct", ligne 4, dans <module>
```

10.2.4.2. Seule valeur dans une liste: la `EnumVariable`construction Fonction variable

Supposons que nous voulons qu'un utilisateur soit en mesure de définir une `COLOR`variable qui sélectionne une couleur d'arrière - plan à afficher par une application, mais que nous voulons limiter les choix à un ensemble spécifique de couleurs autorisées. Cela peut être mis en place assez facilement en utilisant la `EnumVariable`, qui prend une liste de `allowed_values` en plus du nom de la variable, la valeur par défaut, et les arguments de l' aide de texte:

```
vars = Variables ( 'custom.py')
vars.Add (EnumVariable ( 'COLOR', 'Définir la couleur d'arrière-plan', 'rouge',
                   allowed_values = ( 'rouge', 'vert', 'bleu')))
env = environnement (variables = vars,
```

```
                  CPPDEFINES = { 'color': ' "$ {color}"'})
env.Program ( 'foo.c')
```

L'utilisateur peut maintenant définir la explicity `COLOR`variable de compilation à l' une des valeurs autorisées spécifiées:

```
% scons -Q COLOR=red foo.o
cc -o foo.o -c -DCOLOR = foo.c "rouge"
% scons -Q COLOR=blue foo.o
cc -o foo.o -c -DCOLOR = foo.c "bleu"
% scons -Q COLOR=green foo.o
cc -o foo.o -c -DCOLOR = foo.c "vert"
```

Mais, presque plus important encore , une tentative de mettre `COLOR` à une valeur qui ne figure pas dans la liste génère un message d'erreur:

```
% scons -Q COLOR=magenta foo.o

scons: *** Valeur non valide pour l'option COULEUR: magenta. Les valeurs valides sont les suivantes: ( « rouge », « vert », « bleu »)
Fichier "/ home / mon / projet / SConstruct", ligne 5, dans <module>
```

La `EnumVariable`fonction prend également en charge un moyen de cartographier d' autres noms à des valeurs autorisées. Supposons, par exemple, que nous voulons permettre à l'utilisateur d'utiliser le mot `navy`comme synonyme de `blue`. Nous faisons cela en ajoutant un `map`dictionnaire qui tracera ses valeurs clés à la valeur juridique souhaitée:

```
vars = Variables ( 'custom.py')
vars.Add (EnumVariable ( 'COLOR', 'Définir la couleur d'arrière-plan', 'rouge',
                    allowed_values = ( 'rouge', 'vert', 'bleu'),
                    map = { 'marine': 'bleu'}))
env = environnement (variables = vars,
                  CPPDEFINES = { 'color': ' "$ {color}"'})
env.Program ( 'foo.c')
```

Si on le souhaite, l'utilisateur peut alors utiliser `navy`la ligne de commande, et SCons se traduira dans `blue` quand vient le temps d'utiliser la `COLOR` variable pour construire une cible:

```
% scons -Q COLOR=navy foo.o
cc -o foo.o -c -DCOLOR = foo.c "bleu"
```

Par défaut, lorsque vous utilisez la `EnumVariable`fonction, les arguments qui diffèrent des valeurs légales qu'en cas sont considérés comme des valeurs illégales:

```
% scons -Q COLOR=Red foo.o

scons: *** Valeur non valide pour l'option Couleur: Rouge. Les valeurs valides sont les suivantes: ( « rouge », « vert », « bleu »)
Fichier "/ home / mon / projet / SConstruct", ligne 5, dans <module>
% scons -Q COLOR=BLUE foo.o

scons: *** Valeur non valide pour l'option Couleur: BLEU. Les valeurs valides sont les suivantes: ( « rouge », « vert », « bleu »)
Fichier "/ home / mon / projet / SConstruct", ligne 5, dans <module>
% scons -Q COLOR=nAvY foo.o

scons: *** Valeur non valide pour l'option Couleur: navy. Les valeurs valides sont les suivantes: ( « rouge », « vert », « bleu »)
Fichier "/ home / mon / projet / SConstruct", ligne 5, dans <module>
```

La `EnumVariable`fonction peut prendre un supplément`ignorecase`argument mot - clé qui, lorsqu'il est réglé `1`, dit SCons pour permettre des différences de cas lorsque les valeurs sont spécifiées:

```
vars = Variables ( 'custom.py')
vars.Add (EnumVariable ( 'COLOR', 'Définir la couleur d'arrière-plan', 'rouge',
                    allowed_values = ( 'rouge', 'vert', 'bleu'),
                    map = { 'marine': 'bleu'},
                    ignorecase = 1))
env = environnement (variables = vars,
                  CPPDEFINES = { 'color': ' "$ {color}"'})
env.Program ( 'foo.c')
```

Ce qui donne la sortie:

```
% scons -Q COLOR=Red foo.o
cc -o foo.o -c -DCOLOR = foo.c "Red"
% scons -Q COLOR=BLUE foo.o
cc -o foo.o -c -DCOLOR = foo.c "BLUE"
% scons -Q COLOR=nAvY foo.o
cc -o foo.o -c -DCOLOR = foo.c "bleu"
% scons -Q COLOR=green foo.o
cc -o foo.o -c -DCOLOR = foo.c "vert"
```

Notez qu'une `ignorecase`valeur `1` préserve le cas épellation que l'utilisateur fourni. Si vous voulez SCons traduire les noms en minuscules, quel que soit le cas utilisé par l'utilisateur, spécifiez une `ignorecase`valeur de `2`:

```
vars = Variables ( 'custom.py')
vars.Add (EnumVariable ( 'COLOR', 'Définir la couleur d'arrière-plan', 'rouge',
                    allowed_values = ( 'rouge', 'vert', 'bleu'),
                    map = { 'marine': 'bleu'},
                    ignorecase = 2))
env = environnement (variables = vars,
                  CPPDEFINES = { 'color': ' "$ {color}"'})
env.Program ( 'foo.c')
```

Maintenant SCons utilisera les valeurs de `red`, `green`ou `blue` quelle que soit la façon dont l'utilisateur précise ces valeurs sur la ligne de commande:

```
% scons -Q COLOR=Red foo.o
cc -o foo.o -c -DCOLOR = foo.c "rouge"
% scons -Q COLOR=nAvY foo.o
cc -o foo.o -c -DCOLOR = foo.c "bleu"
```

```
% scons -Q COLOR=GREEN foo.o
cc -o foo.o -c -DCOLOR = foo.c "vert"
```

10.2.4.3. Les valeurs multiples à partir d' une liste: la `ListVariable`construction Fonction variable

Une autre façon dont vous pouvez permettre aux utilisateurs de contrôler une variable de construction est de spécifier une liste d'une ou plusieurs valeurs juridiques. SCons soutient cela par la `ListVariable`fonction. Si, par exemple, nous voulons qu'un utilisateur soit en mesure de définir une `COLORS`variable d' un ou plusieurs de la liste légale des valeurs:

```
vars = Variables ( 'custom.py')
vars.Add (ListVariable ( 'couleurs', 'Liste des couleurs', 0,
                    [ 'Rouge', 'vert', 'bleu']))
env = environnement (variables = vars,
                 CPPDEFINES = { 'Couleurs': ' "$ {} COULEURS"'})
env.Program ( 'foo.c')
```

Un utilisateur peut spécifier une liste de valeurs juridiques séparées par des virgules, qui se traduit dans une liste séparée par des espaces pour passer aux commandes de construction de tout:

```
% scons -Q COLORS=red,blue foo.o
cc -o foo.o -c -DCOLORS = foo.c "bleu rouge"
% scons -Q COLORS=blue,green,red foo.o
cc -o foo.o -c -DCOLORS = foo.c "rouge vert bleu"
```

De plus, la `ListVariable`fonction permet à l'utilisateur de spécifier des mots - clés explicites de `all`ou `none` pour sélectionner toutes les valeurs, ou aucune d'entre elles, respectivement:

```
% scons -Q COLORS=all foo.o
cc -o foo.o -c -DCOLORS = foo.c "bleu rouge vert"
% scons -Q COLORS=none foo.o
cc -o foo.o -c -DCOLORS = "" foo.c
```

Et, bien sûr, une valeur illégale génère toujours un message d'erreur:

```
% scons -Q COLORS=magenta foo.o

scons: *** Erreur option de la conversion: COULEURS
La valeur non valide (s) pour l'option: magenta
Fichier "/ home / mon / projet / SConstruct", ligne 5, dans <module>
```

10.2.4.4. Chemin Noms: la `PathVariable`construction fonction variable

SCons prend en charge une `PathVariable`fonction pour le rendre facile de créer une variable de construction pour contrôler un nom de chemin prévu. Si, par exemple, vous devez définir une variable dans le préprocesseur qui contrôle l'emplacement d'un fichier de configuration:

```
vars = Variables ( 'custom.py')
vars.Add (PathVariable ( 'CONFIG',
                    « Chemin vers le fichier de configuration »,
                    '/ Etc / my_config'))
env = environnement (variables = vars,
                 CPPDEFINES = { 'CONFIG_FILE': ' "$ CONFIG"'})
env.Program ( 'foo.c')
```

Ceci permet alors à l'utilisateur de remplacer la `CONFIG`variable de compilation sur la ligne de commande si nécessaire:

```
% scons -Q foo.o
cc -o foo.o -c -DCONFIG_FILE = "/ etc / my_config" foo.c
% scons -Q CONFIG=/usr/local/etc/other_config foo.o
scons: `foo.o » est à jour.
```

Par défaut, `PathVariable`vérifie pour vous assurer que le chemin existe et génère une erreur si elle ne fonctionne pas:

```
% scons -Q CONFIG=/does/not/exist foo.o

scons: *** chemin pour l'option n'existe pas CONFIG: / ne / ne / existe
Fichier "/ home / mon / projet / SConstruct", ligne 6, dans <module>
```

`PathVariable`fournit un certain nombre de méthodes que vous pouvez utiliser pour modifier ce comportement. Si vous voulez vous assurer que tous les chemins spécifiés sont, en fait, les fichiers et répertoires non, utilisez la `PathVariable.PathIsFile`méthode:

```
vars = Variables ( 'custom.py')
vars.Add (PathVariable ( 'CONFIG',
                    « Chemin vers le fichier de configuration »,
                    '/ Etc / my_config',
                    PathVariable.PathIsFile))
env = environnement (variables = vars,
                 CPPDEFINES = { 'CONFIG_FILE': ' "$ CONFIG"'})
env.Program ( 'foo.c')
```

A l' inverse, pour faire en sorte que tous les chemins spécifiés sont des répertoires et des fichiers non, utilisez la `PathVariable.PathIsDir`méthode:

```
vars = Variables ( 'custom.py')
vars.Add (PathVariable ( 'DBDIR',
                    « Chemin vers le répertoire de base de données »,
                    '/ Var / my_dbdir',
                    PathVariable.PathIsDir))
env = environnement (variables = vars,
                 CPPDEFINES = { 'DBDIR': ' "$ DBDIR"'})
env.Program ( 'foo.c')
```

Si vous voulez vous assurer que tous les chemins spécifiés sont des répertoires, et que vous souhaitez le répertoire créé si elle n'existe pas, utilisez la `PathVariable.PathIsDirCreate`méthode:

```
vars = Variables ( 'custom.py')
vars.Add (PathVariable ( 'DBDIR',
                    « Chemin vers le répertoire de base de données »,
                    '/ Var / my_dbdir',
                    PathVariable.PathIsDirCreate))
env = environnement (variables = vars,
                  CPPDEFINES = { 'DBDIR': ' "$ DBDIR"'})
env.Program ( 'foo.c')
```

Enfin, si vous ne vous inquiétez pas si le chemin existe, est un fichier ou un répertoire, utilisez la `PathVariable.PathAccept`méthode pour accepter tout chemin que l'utilisateur fournit:

```
vars = Variables ( 'custom.py')
vars.Add (PathVariable ( 'production',
                    « Chemin vers le fichier de sortie ou le répertoire »,
                    Aucun,
                    PathVariable.PathAccept))
env = environnement (variables = vars,
                  CPPDEFINES = { 'SORTIE': ' "SORTIE $"'})
env.Program ( 'foo.c')
```

10.2.4.5. Activé / Désactivé Noms de chemin: la `PackageVariable`construction Fonction variable

Parfois , vous voulez donner aux utilisateurs un contrôle encore plus sur une variable de nom de chemin, ce qui leur permet d'activer ou de désactiver explicitement le nom de chemin en utilisant `yes`ou `no`mots - clés, en plus de leur permettre de fournir un nom de chemin d' accès explicite. SCons soutient la `PackageVariable` fonction pour soutenir ceci:

```
vars = Variables ( 'custom.py')
vars.Add (PackageVariable ( 'paquet',
                      « Paquet Localisation »,
                      '/ Opt / emplacement'))
env = environnement (variables = vars,
                  CPPDEFINES = { 'FORFAIT': ' "$ PACKAGE"'})
env.Program ( 'foo.c')
```

Lorsque le `SConscript`fichier utilise le `PackageVariable`funciton, l' utilisateur peut désormais utiliser encore la valeur par défaut ou fournir un nom de chemin primordial, mais peut désormais définir explicitement la variable spécifiée à une valeur qui indique le paquet doit être activé (dans ce cas , la valeur par défaut doit être utilisé) ou désactivée:

```
% scons -Q foo.o
cc -o foo.o -c -DPACKAGE = "/ opt / emplacement" foo.c
% scons -Q PACKAGE=/usr/local/location foo.o
cc -o foo.o -c -DPACKAGE = "/ usr / local / emplacement" foo.c
% scons -Q PACKAGE=yes foo.o
cc -o foo.o -c -DPACKAGE = foo.c "True"
% scons -Q PACKAGE=no foo.o
cc -o foo.o -c -DPACKAGE = foo.c "Faux"
```

10.2.5. Ajout de la ligne de commande multiples Créer variables à la fois

Enfin, SCons fournit un moyen d'ajouter plusieurs variables de construction à un `Variables`objet à la fois. Au lieu d'avoir à appeler la `Add`méthode plusieurs fois, vous pouvez appeler la `AddVariables`méthode avec une liste de variables de construction à ajouter à l'objet. Chaque construction variable est spécifié comme un tuple d'arguments, comme vous le feriez passer à la `Add`méthode elle - même, ou comme un appel à l' une des fonctions prédéfinies pour la commande en ligne pré-emballés construire des variables. dans un ordre quelconque:

```
vars = Variables ()
vars.AddVariables (
    ( 'RELEASE', 'Mis à 1 pour construire la libération', 0),
    ( 'CONFIG', 'fichier de configuration', '/ etc / my_config'),
    BoolVariable ( 'avertissements', 'compilation avec -Wall et similaire', 1),
    EnumVariable ( 'debug', 'sortie de débogage et des symboles', 'non',
                allowed_values = ( 'oui', 'non', 'plein'),
                map = {}, ignorecase = 0), # sensible à la casse
    ListVariable ( 'partagée',
                « Bibliothèques pour construire des bibliothèques partagées »,
                'tout',
                nom = list_of_libs),
    PackageVariable ( 'x11',
                   « Utiliser X11 installé ici (oui = chercher des places) »,
                   'Oui'),
    PathVariable ( 'qtdir', 'où la racine de Qt est installé', qtdir),
)
```

10.2.6. Manipulation Unknown ligne de commande Variables de construction: la `UnknownVariables`fonction

Les utilisateurs peuvent, bien sûr, parfois orthographié les noms de variables dans leurs paramètres de ligne de commande. SCons ne génère pas d'erreur ou d' avertissement pour toutes les variables inconnues des utilisateurs spécifierait sur la ligne de commande. (Ceci est en grande partie parce que vous pouvez traiter les arguments directement en utilisant le `ARGUMENTS`dictionnaire, et donc SCons ne peut pas savoir dans le cas général si une donnée variable « mal orthographié » est vraiment inconnu et un problème potentiel, ou quelque chose que votre `SConscript`fichier traitera directement avec un code Python.)

Toutefois, si vous utilisez un `Variables`objet pour définir un ensemble spécifique de construction de ligne de commande variables que vous attendez que les utilisateurs soient en mesure de définir, vous pouvez fournir un message d'erreur ou d' avertissement de votre propre si l'utilisateur fournit une variable paramètre qui est *pas* dans la liste définie des noms de variables connues à l' `Variables`objet. Vous pouvez le faire en appelant la `UnknownVariables` méthode de l' `Variables`objet:

```
vars = Variables (Aucun)
vars.Add ( 'RELEASE', 'Mis à 1 pour construire la libération', 0)
env = environnement (variables = vars,
                  CPPDEFINES = { 'RELEASE_BUILD': '$ {} REJET'})
inconnu = vars.UnknownVariables ()
```

```
si elle est inconnue:
    print "Les variables inconnues:", unknown.keys ()
    Sortie (1)
env.Program ( 'foo.c')
```

La `UnknownVariables`méthode renvoie un dictionnaire contenant les mots - clés et les valeurs de toutes les variables de l'utilisateur spécifié sur la ligne de commande qui ne sont *pas* entre les variables connues de l' `Variables`objet (d'avoir été spécifié en utilisant le `Variables`de l' objet `Add`de la méthode). Dans le examble ci - dessus, nous vérifions si le dictionnaire retourné par la `UnknownVariables`non-vide, et si oui imprimer la liste Python contenant les noms des variables unknwown puis appeler la `Exit`fonction de mettre fin à SCons :

```
% scons -Q NOT_KNOWN=foo
variables inconnues: [ 'NOT_KNOWN']
```

Bien sûr, vous pouvez traiter les éléments dans le dictionnaire retourné par la `UnknownVariables`fonction de toute façon appropriée à votre configuration de construction, y compris l' impression juste un message d'avertissement , mais pas la sortie, la connexion d' une erreur quelque part, etc.

Notez que vous devez retarder l'appel `UnknownVariables` avant d' avoir appliqué l' `Variables`objet à un environnement de construction avec l' `variables=` argument de mot - clé d'un `Environment`appel.

10.3. Cibles de ligne de commande

10.3.1. Fetching Cibles de ligne de commande: la `COMMAND_LINE_TARGETS`variable

SCons prend en charge une `COMMAND_LINE_TARGETS`variable qui vous permet de récupérer la liste des cibles que l'utilisateur spécifié sur la ligne de commande. Vous pouvez utiliser les cibles pour manipuler la construction de quelque façon que vous le souhaitez. A titre d'exemple simple, supposons que vous souhaitez imprimer un rappel à l'utilisateur chaque fois qu'un programme spécifique est construit. Vous pouvez le faire en vérifiant la cible dans la `COMMAND_LINE_TARGETS`liste:

```
si 'bar' en COMMAND_LINE_TARGETS:
    print « Ne pas oublier de copier` bar » dans l'archive! »
Par défaut (Programme ( 'foo.c'))
Programme ( 'bar.c')
```

Puis, en cours d' exécution SCons avec la cible par défaut fonctionne comme il le fait toujours, mais en spécifiant explicitement la `bar`cible sur la ligne de commande génère le message d'avertissement:

```
% scons -Q
cc -o -c foo.o foo.c
cc -o foo foo.o
% scons -Q bar
Ne pas oublier de copier `bar » dans l'archive!
cc -o -c bar.o bar.c
cc -o bar bar.o
```

Une autre utilisation pratique pour la `COMMAND_LINE_TARGETS`variable peut être d'accélérer une construction en ne lisant que certaines filiales de `SConscript` fichiers si est demandé une cible spécifique.

10.3.2. Contrôle des cibles par défaut: la `Default`fonction

L' une des choses les plus élémentaires que vous pouvez contrôler est qui cible SCons construiront par défaut - qui est, quand il n'y a pas de cibles spécifiées sur la ligne de commande. Comme mentionné précédemment, SCons normalement construire chaque cible dans ou sous le répertoire courant par défaut - qui est, lorsque vous ne spécifiez pas explicitement une ou plusieurs cibles sur la ligne de commande. Parfois, cependant, vous pouvez spécifier explicitement que seuls certains programmes, ou des programmes dans certains répertoires, doivent être édifiés par défaut. Vous faites cela avec la `Default`fonction:

```
env = environnement ()
bonjour = env.Program ( 'hello.c')
env.Program ( 'goodbye.c')
Par défaut (bonjour)
```

Ce `SConstruct`fichier sait comment construire deux programmes,`hello`et `goodbye`, mais construit seulement le `hello`programme par défaut:

```
% scons -Q
cc -o -c hello.o hello.c
cc -o bonjour hello.o
% scons -Q
scons: `bonjour » est à jour.
% scons -Q goodbye
cc -o -c goodbye.o goodbye.c
cc -o au revoir goodbye.o
```

Notez que, même lorsque vous utilisez la `Default` fonction dans votre `SConstruct`fichier, vous pouvez toujours spécifier explicitement le répertoire courant (`.`) sur la ligne de commande pour dire SCons tout construire (ou ci - dessous) le répertoire courant:

```
% scons -Q .
cc -o -c goodbye.o goodbye.c
cc -o au revoir goodbye.o
cc -o -c hello.o hello.c
cc -o bonjour hello.o
```

Vous pouvez également appeler la `Default` fonction plus d'une fois, dans ce cas , chaque appel ajoute à la liste des cibles à construire par défaut:

```
env = environnement ()
prog1 = env.Program ( 'prog1.c')
Par défaut (PROG1)
prog2 = env.Program ( 'prog2.c')
PROG3 = env.Program ( 'prog3.c')
Par défaut (PROG3)
```

Vous pouvez également spécifier plusieurs cibles dans un seul appel à la `Default`fonction:

```
env = environnement ()
prog1 = env.Program ( 'prog1.c')
prog2 = env.Program ( 'prog2.c')
PROG3 = env.Program ( 'prog3.c')
Par défaut (PROG1, PROG3)
```

Chacune de ces deux derniers exemples bâtirai seulement les PROG1et PROG3 programmes par défaut:

```
% scons -Q
cc -o -c prog1.o prog1.c
cc -o prog1 prog1.o
cc -o -c prog3.o prog3.c
cc -o PROG3 prog3.o
% scons -Q .
cc -o -c prog2.o prog2.c
cc -o prog2 prog2.o
```

Vous pouvez lister un répertoire comme argument pour `Default`:

```
env = environnement ()
env.Program ([ 'prog1 / main.c', 'prog1 / foo.c'])
env.Program ([ 'prog2 / main.c', 'prog2 / bar.c'])
Par défaut ( 'prog1')
```

Dans ce cas, seule la cible (s) dans ce répertoire sera construit par défaut:

```
% scons -Q
cc -o prog1 / foo.o -c prog1 / foo.c
cc -o prog1 / main.o -c prog1 / main.c
cc -o prog1 / main prog1 / main.o prog1 / foo.o
% scons -Q
scons: `prog1' est à jour.
% scons -Q .
cc -o prog2 / bar.o -c prog2 / bar.c
cc -o prog2 / main.o -c prog2 / main.c
cc -o prog2 / main prog2 / main.o prog2 / bar.o
```

Enfin, si pour une raison quelconque , vous ne voulez pas de cibles construites par défaut, vous pouvez utiliser le Python `None` variable:

```
env = environnement ()
prog1 = env.Program ( 'prog1.c')
prog2 = env.Program ( 'prog2.c')
Par défaut (Aucun)
```

Ce qui produirait une sortie comme la construction:

```
% scons -Q
scons: *** Aucune cible définie et si aucun cibles par défaut () trouvé. Arrêtez.
Rien trouvé à construire
% scons -Q .
cc -o -c prog1.o prog1.c
cc -o prog1 prog1.o
cc -o -c prog2.o prog2.c
cc -o prog2 prog2.o
```

10.3.2.1. Récupérer la liste des cibles par défaut: la `DEFAULT_TARGETS`variable

SCons prend en charge une `DEFAULT_TARGETS`variable qui vous permet d' obtenir la liste actuelle des cibles par défaut. La `DEFAULT_TARGETS`variable a deux différences importantes de la `COMMAND_LINE_TARGETS`variable. Tout d' abord, la `DEFAULT_TARGETS`variable est une liste des internes SCons noeuds, donc vous devez convertir les éléments de la liste en chaînes si vous voulez les imprimer ou rechercher un nom cible spécifique. Heureusement, vous pouvez le faire facilement en utilisant Python `map`fonction pour exécuter la liste par `str`:

```
prog1 = Programme ( 'prog1.c')
Par défaut (PROG1)
impression "DEFAULT_TARGETS est", la carte (str, DEFAULT_TARGETS)
```

(Gardez à l' esprit que toute la manipulation de la `DEFAULT_TARGETS`liste a lieu pendant la première phase où SCons est en train de lire les `SConscript`fichiers, ce qui est évident si nous laissons hors du `-Q`drapeau quand nous courons SCons :)

```
% scons
scons: Lecture des fichiers SConscript ...
DEFAULT_TARGETS est [ 'prog1']
scons: fait la lecture des fichiers SConscript.
scons: objectifs de construction ...
cc -o -c prog1.o prog1.c
cc -o prog1 prog1.o
scons: fait des objectifs de construction.
```

En second lieu , le contenu du `DEFAULT_TARGETS`changement de liste en réponse aux appels à la `Default`fonction, comme vous pouvez le voir dans ce qui suit le `SConstruct`fichier:

```
prog1 = Programme ( 'prog1.c')
Par défaut (PROG1)
print "DEFAULT_TARGETS est maintenant", la carte (str, DEFAULT_TARGETS)
prog2 = Programme ( 'prog2.c')
Par défaut (PROG2)
print "DEFAULT_TARGETS est maintenant", la carte (str, DEFAULT_TARGETS)
```

Ce qui donne la sortie:

```
% scons
scons: Lecture des fichiers SConscript ...
DEFAULT_TARGETS est maintenant [ 'prog1']
DEFAULT_TARGETS est maintenant [ 'prog1', 'prog2']
scons: fait la lecture des fichiers SConscript.
scons: objectifs de construction ...
cc -o -c prog1.o prog1.c
cc -o prog1 prog1.o
cc -o -c prog2.o prog2.c
cc -o prog2 prog2.o
scons: fait des objectifs de construction.
```

En pratique, cela signifie simplement que vous devez faire attention à l'ordre dans lequel vous appelez la `Default`fonction et reportez - vous à la `DEFAULT_TARGETS`liste, pour vous assurer que vous ne porte pas sur la liste avant que vous avez ajouté les cibles par défaut nous vous en elle.

10.3.3. Récupérer la liste des cibles de construire, indépendamment de leur origine: la `BUILD_TARGETS`variable

Nous avons déjà été mis en place à la `COMMAND_LINE_TARGETS`variable, qui contient une liste de cibles spécifiées sur la ligne de commande, et la `DEFAULT_TARGETS`variable qui contient une liste de cibles spécifiées par les appels à la `Default`méthode ou la fonction. Parfois, cependant, vous voulez une liste de ce que les objectifs SCons essaieront de construire, que les cibles provenaient de la ligne de commande ou d' un `Default`appel. Vous pouvez coder cela en main, comme suit:

```
si COMMAND_LINE_TARGETS:
    cibles = COMMAND_LINE_TARGETS
autre:
    cibles = DEFAULT_TARGETS
```

SCons , cependant, fournit une pratique `BUILD_TARGETS`variable qui élimine le besoin de cette manipulation par la main. Essentiellement, la `BUILD_TARGETS`variable contient une liste des cibles de ligne de commande, le cas échéant ont été spécifiés, et si aucune cible de ligne de commande ont été spécifiés, il contient une liste des cibles spécifiées par la `Default`méthode ou fonction.

Parce que `BUILD_TARGETS`peut contenir une liste de SCons noeuds, vous devez convertir les éléments de la liste à des chaînes si vous voulez les imprimer ou rechercher un nom cible spécifique, tout comme la `DEFAULT_TARGETS`liste:

```
prog1 = Programme ( 'prog1.c')
Programme ( 'prog2.c')
Par défaut (PROG1)
print "BUILD_TARGETS est", la carte (str, BUILD_TARGETS)
```

Remarquez que la valeur des `BUILD_TARGETS` changements en fonction si une cible est spécifiée sur la ligne de commande:

```
% scons -Q
BUILD_TARGETS est [ 'prog1']
cc -o -c prog1.o prog1.c
cc -o prog1 prog1.o
% scons -Q prog2
BUILD_TARGETS est [ 'prog2']
cc -o -c prog2.o prog2.c
cc -o prog2 prog2.o
% scons -Q -c .
BUILD_TARGETS est [ '']
Suppression prog1.o
Suppression prog1
Suppression prog2.o
Suppression prog2
```

---

[3] La `AddOption`fonction est, en effet, mis en oeuvre en utilisant une sous - classe de la `optparse.OptionParser`.

Chapitre 11. Installation des fichiers dans d' autres répertoires: le `Install`constructeur

Une fois qu'un programme est construit, il est souvent approprié de l'installer dans un autre répertoire pour un usage public. Vous utilisez la `Install`méthode pour organiser un programme, ou tout autre fichier, à copier dans un répertoire de destination:

```
env = environnement ()
bonjour = env.Program ( 'hello.c')
env.Install ( '/ usr / bin', bonjour)
```

Notez toutefois que l' installation d' un fichier est toujours considéré comme un type de fichier « build ». Ceci est important quand vous vous rappelez que le comportement par défaut de SCons est de créer des fichiers dans ou sous le répertoire courant. Si, comme dans l'exemple ci - dessus, vous installez des fichiers dans un répertoire en dehors du haut niveau `SConstruct`l'arborescence des répertoires de fichiers, vous devez spécifier ce répertoire (ou un répertoire supérieur, par exemple `/`) pour qu'il installer quoi que ce soit là - bas:

```
% scons -Q
cc -o -c hello.o hello.c
cc -o bonjour hello.o
% scons -Q /usr/bin
Fichier d'installation: "bonjour" comme "/ usr / bin / bonjour"
```

Il peut toutefois être lourd à retenir (et tapez) le répertoire de destination dans lequel le programme (ou tout autre fichier) doivent être installés. Ceci est une zone où la `Alias` fonction est très pratique, vous permettant, par exemple, de créer un pseudo-cible nommée `install` qui peut élargir le répertoire de destination spécifié:

```
env = environnement ()
bonjour = env.Program ( 'hello.c')
env.Install ( '/ usr / bin', bonjour)
env.Alias ( 'install', '/ usr / bin')
```

On obtient ainsi la capacité plus naturelle à installer le programme dans sa destination comme suit:

```
% scons -Q
cc -o -c hello.o hello.c
cc -o bonjour hello.o
% scons -Q install
Fichier d'installation: "bonjour" comme "/ usr / bin / bonjour"
```

11.1. Installation de plusieurs fichiers dans un répertoire

Vous pouvez installer plusieurs fichiers dans un répertoire simplement en appelant la `Install`fonction à plusieurs reprises:

```
env = environnement ()
bonjour = env.Program ( 'hello.c')
au revoir = env.Program ( 'goodbye.c')
env.Install ( '/ usr / bin', bonjour)
env.Install ( '/ usr / bin', au revoir)
env.Alias ( 'install', '/ usr / bin')
```

Ou, plus succinctement, la liste des multiples fichiers d'entrée dans une liste (comme vous pouvez le faire avec un autre constructeur):

```
env = environnement ()
bonjour = env.Program ( 'hello.c')
au revoir = env.Program ( 'goodbye.c')
env.Install ( '/ usr / bin', [bonjour, au revoir])
env.Alias ( 'install', '/ usr / bin')
```

Chacune de ces deux exemples, on obtient:

```
% scons -Q install
cc -o -c goodbye.o goodbye.c
cc -o au revoir goodbye.o
Fichier d'installation: "au revoir" comme "/ usr / bin / au revoir"
cc -o -c hello.o hello.c
cc -o bonjour hello.o
Fichier d'installation: "bonjour" comme "/ usr / bin / bonjour"
```

11.2. Installation d'un fichier sous un autre nom

La `Install`méthode permet de conserver le nom du fichier lorsqu'il est copié dans le répertoire de destination. Si vous avez besoin de changer le nom du fichier lorsque vous copiez, utilisez la `InstallAs`fonction:

```
env = environnement ()
bonjour = env.Program ( 'hello.c')
env.InstallAs ( '/ usr / bin / bonjour nouveau', bonjour)
env.Alias ( 'install', '/ usr / bin')
```

Ceci installe le `hello` programme avec le nom `hello-new` comme suit:

```
% scons -Q install
cc -o -c hello.o hello.c
cc -o bonjour hello.o
Fichier d'installation: « bonjour » comme « / usr / bin / bonjour nouveau »
```

11.3. Installation de plusieurs fichiers sous différents noms

Enfin, si vous avez plusieurs fichiers qui doivent tous être installés avec des noms de fichiers, vous pouvez appeler le `InstallAs`plusieurs fois la fonction, ou comme un raccourci, vous pouvez fournir des listes de même longueur pour les deux arguments source et cible:

```
env = environnement ()
bonjour = env.Program ( 'hello.c')
au revoir = env.Program ( 'goodbye.c')
env.InstallAs ([ '/ usr / bin / bonjour nouveau',
                 '/ Usr / bin / adieu nouveau'],
                 [Bonjour au revoir])
env.Alias ( 'install', '/ usr / bin')
```

Dans ce cas, la `InstallAs`fonction boucle dans les deux listes simultanément, et des copies de chaque fichier source dans son nom de fichier cible correspondant:

```
% scons -Q install
cc -o -c goodbye.o goodbye.c
cc -o au revoir goodbye.o
Fichier d'installation: « au revoir » comme « / usr / bin / adieu nouveau »
cc -o -c hello.o hello.c
cc -o bonjour hello.o
Fichier d'installation: « bonjour » comme « / usr / bin / bonjour nouveau »
```

Chapitre 12. Système de fichiers multi-plateforme Manipulation

SCons fournit un certain nombre de fonctions de la plate-forme indépendante, appelés `factories`, qui effectuent des manipulations du système de fichiers courants comme copier, déplacer ou supprimer des fichiers et des répertoires, ou faire des répertoires. Ces fonctions sont `factories` parce qu'ils ne remplissent pas l'action au moment où ils sont appelés, ils retournent un `Action`objet qui peut être exécuté au moment opportun.

12.1. Copie de fichiers ou de répertoires: L' `Copy`usine

Supposons que vous voulez prendre des dispositions pour faire une copie d'un fichier, et ne dispose pas d' un constructeur préexistant approprié. [4] Un moyen serait d'utiliser l' `Copy`usine d'action en collaboration avec le `Command`constructeur:

```
Commande ( "file.out", "file.in", Copie ( "TARGET $", "$ SOURCE"))
```

Notez que l'action retournée par l' `Copy`usine étendra la `$TARGET`et `$SOURCE`cordes à l'époque `file.out`est construit, et que l'ordre des arguments est la même que celle d'un constructeur lui - même - à savoir, cibler en premier, suivi par la source:

```
% scons -Q
Copie ( "file.out", "file.in")
```

Vous pouvez, bien sûr, le nom d' un fichier explicitement au lieu d'utiliser `$TARGET`ou `$SOURCE`:

```
Commande ( "file.out", [], Copie ( "TARGET $", "file.in"))
```

Qui exécute comme:

```
% scons -Q
Copie ( "file.out", "file.in")
```

L'utilité de l' `Copy`usine devient plus évidente lorsque vous l' utilisez dans une liste d'actions transmises au `Command`constructeur. Par exemple, supposons que vous avez besoin d'exécuter un fichier via un utilitaire qui modifie uniquement les fichiers en place, et ne peut pas entrée « pipe » à la sortie. Une solution consiste à copier le fichier source à un nom de fichier temporaire, exécutez l'utilitaire, puis copiez le fichier temporaire modifié à la cible, que l' `Copy`usine rend extrêmement facile:

```
Commande ( "file.out", "file.in",
        [
          Copie ( "tempfile", "$ SOURCE"),
          "Modifier tempfile",
          Copie ( "TARGET $", "tempfile"),
        ])
```

La sortie ressemble alors à:

```
% scons -Q
Copie ( "tempfile", "file.in")
modifier tempfile
Copie ( "file.out", "tempfile")
```

L' `Copy`usine a un troisième argument optionnel qui contrôle la façon dont les liens symboliques sont copiés.

```
faible profondeur # lien symbolique copié comme un nouveau lien symbolique:
Commande ( "linkin", "LinkOut", Copie ( "TARGET $", "$ SOURCE" [, true]))

cible # lien symbolique copié en tant que fichier ou répertoire:
Commande ( "linkin", "FileOrDirectoryOut", Copie ( "TARGET $", "$ SOURCE", False))
```

12.2. Suppression de fichiers ou de répertoires: L' `Delete`usine

Si vous devez supprimer un fichier, l' `Delete`usine peut être utilisé de la même façon que l' `Copy`usine. Par exemple, si nous voulons nous assurer que le fichier temporaire dans notre dernier exemple n'existe pas avant que nous copions à elle, nous pourrions ajouter `Delete`au début de la liste de commande:

```
Commande ( "file.out", "file.in",
        [
          Supprimer ( "tempfile"),
          Copie ( "tempfile", "$ SOURCE"),
          "Modifier tempfile",
          Copie ( "TARGET $", "tempfile"),
        ])
```

Ce qui exécute alors comme suit:

```
% scons -Q
Supprimer ( "tempfile")
Copie ( "tempfile", "file.in")
modifier tempfile
Copie ( "file.out", "tempfile")
```

Bien sûr, comme toutes ces `Action`usines, l' `Delete`usine se développe également `$TARGET`et les `$SOURCE`variables de façon appropriée. Par exemple:

```
Commande ( "file.out", "file.in",
        [
          Supprimer ( "$ TARGET"),
          Copie ( "$ TARGET", "$ SOURCE")
        ])
```

Comme Exécute:

```
% scons -Q
Supprimer ( "file.out")
Copie ( "file.out", "file.in")
```

Notez toutefois que vous ne avez généralement pas besoin d'appeler l' `Delete`usine explicitement de cette façon; par défaut, SCons supprime sa cible (s) pour vous avant d' exécuter une action.

Un mot d'avertissement sur l' utilisation de l' `Delete`usine: il a les mêmes extensions variables disponibles que toute autre usine, y compris la `$SOURCE`variable. Spécification `Delete("$SOURCE")` n'est pas quelque chose que vous voulez faire habituellement!

12.3. Déplacement (renommer) fichiers ou des répertoires: L' `Move`usine

L' `Move`usine vous permet de renommer un fichier ou un répertoire. Par exemple, si nous ne voulons pas copier le fichier temporaire, nous pourrions utiliser:

```
Commande ( "file.out", "file.in",
        [
          Copie ( "tempfile", "$ SOURCE"),
          "Modifier tempfile",
          Déplacer ( "TARGET $", "tempfile"),
        ])
```

Ce qui exécuterait comme:

```
% scons -Q
Copie ( "tempfile", "file.in")
modifier tempfile
Déplacer ( "file.out", "tempfile")
```

12.4. Mise à jour du temps Modification d'un fichier: L' `Touch`usine

Si vous avez juste besoin de mettre à jour la date de modification enregistrée pour un fichier, utilisez l' `Touch`usine:

```
Commande ( "file.out", "file.in",
        [
          Copie ( "$ TARGET", "$ SOURCE"),
          Touchez ( "$ TARGET"),
        ])
```

Qui exécute comme:

```
% scons -Q
Copie ( "file.out", "file.in")
Appuyez sur ( "file.out")
```

12.5. Création d' un répertoire: L' `Mkdir`usine

Si vous avez besoin de créer un répertoire, utilisez l' `Mkdir`usine. Par exemple, si nous avons besoin de traiter un fichier dans un répertoire temporaire dans lequel l'outil de traitement va créer d' autres fichiers que nous ne se soucient pas, vous pouvez utiliser:

```
Commande ( "file.out", "file.in",
        [
          Supprimer ( "tempdir"),
          Mkdir ( "tempdir"),
          Copie ( "tempdir / $ {} SOURCE.file", "$ SOURCE"),
          "Processus tempdir",
          Déplacer ( "TARGET $", "tempdir / output_file"),
          Supprimer ( "tempdir"),
        ])
```

Qui exécute comme:

```
% scons -Q
Supprimer ( "tempdir")
Mkdir ( "tempdir")
Copie ( "tempdir / file.in", "file.in")
procédé tempdir
Déplacer ( "file.out", "tempdir / output_file")
scons: *** [file.out] tempdir / output_file: Aucun fichier ou répertoire
```

12.6. La modification des autorisations de fichiers ou de répertoires: L' `Chmod`usine

Pour modifier les autorisations sur un fichier ou un répertoire, utilisez l' `Chmod`usine. L'argument d'autorisation utilise les bits d'autorisation de style POSIX et devrait généralement être exprimé en octal, non décimal, nombre:

```
Commande ( "file.out", "file.in",
        [
          Copie ( "$ TARGET", "$ SOURCE"),
          Chmod ( "target $", 0755),
        ])
```

Ce qui exécute:

```
% scons -Q
Copie ( "file.out", "file.in")
Chmod ( "file.out", 0755)
```

12.7. Exécution d' une action immédiatement: la `Execute`fonction

Nous avons pour vous montrer comment utiliser les `Action`usines dans la `Command`fonction. Vous pouvez également exécuter un `Action`retourné par une usine (ou en fait, tout `Action`) au moment où le `SConscript`fichier est lu à l'aide de la `Execute`fonction. Par exemple, si nous devons nous assurer qu'un répertoire existe avant de construire des cibles,

```
Execute (Mkdir ( '/ tmp / my_temp_directory'))
```

Notez que cela va créer le répertoire alors que le `SConscript`fichier est en cours de lecture:

```
% scons
scons: Lecture des fichiers SConscript ...
Mkdir ( "/ tmp / my_temp_directory")
```

```
scons: fait la lecture des fichiers SConscript.
scons: objectifs de construction ...
scons: ` « . est à jour.
scons: fait des objectifs de construction.
```

Si vous êtes familier avec Python, vous pouvez vous demander pourquoi vous voulez utiliser au lieu d'appeler le Python natif`os.mkdir()`fonction. L'avantage ici est que l' `Mkdir` action se comporter de façon appropriée si l'utilisateur spécifie les SCons `-n` ou les `-q`options - qui est, il imprimera l'action mais ne faites pas réellement le répertoire où `-n`est spécifié, ou rendre le répertoire mais pas imprimer l'action quand `-q`est spécifié .

La `Execute`fonction retourne l'état de sortie ou de la valeur de retour de l'action sous - jacente en cours d' exécution. Il imprime également un message d'erreur si l'action échoue et renvoie une valeur non nulle.SCons sera *pas* , cependant, fait arrêter la construction si l'action échoue. Si vous voulez arrêter le build en réponse à un échec dans une action appelée par `Execute`, vous devez le faire en vérifiant explicitement la valeur de retour et d' appeler la `Exit`fonction (ou un équivalent Python):

```
si Execute (Mkdir ( '/ tmp / my_temp_directory')):
    # Un problème est survenu tout en rendant le répertoire temporaire.
    Sortie (1)
```

---

[4] Malheureusement, dans les premiers jours de la conception SCons, nous avons utilisé le nom `Copy`de la fonction qui retourne une copie de l'environnement, sinon ce serait le choix logique pour un constructeur qui copie un fichier ou d' arborescence à un emplacement cible.

Chapitre 13. Contrôle de suppression des objectifs

Il y a deux occasions où SCons , par défaut, supprimer les fichiers cibles. Le premier est quand SCons détermine qu'un fichier cible doit être reconstruit et supprime la version existante de la cible avant d' exécuter la seconde est lorsque SCons est invoqué avec l' `-c`option « propre » un arbre de ses cibles construites. Ces comportements peuvent être supprimés avec les `Precious`et `NoClean`fonctions, respectivement.

13.1. Empêcher le retrait de la cible lors de la construction: la `Precious`fonction

Par défaut, SCons supprime des cibles avant de les construire. Parfois, cependant, c'est pas ce que vous voulez. Par exemple, vous pouvez mettre à jour une bibliothèque progressivement, et non par l' avoir supprimé, puis reconstruit à partir de tous les fichiers objets constitutifs. Dans ce cas, vous pouvez utiliser la `Precious`méthode pour empêcher SCons d'enlever la cible avant qu'il ne soit construit:

```
  env = environnement (RANLIBCOM = '')
  lib = env.Library ( 'foo', [ 'f1.c', 'f2.c', 'f3.c'])
  env.Precious (lib)
```

Bien que la sortie ne semble pas différent, SCons n'a pas, en fait, supprimez la bibliothèque cible avant de la reconstruire:

```
% scons -Q
cc -o -c f1.o f1.c
cc -o -c f2.o f2.c
cc -o -c f3.o f3.c
ar rc libtruc.a f1.o f2.o f3.o
```

SCons sera, cependant, toujours supprimer les fichiers marqués comme `Precious` lorsque l' `-c`on utilise l' option.

13.2. Empêcher le retrait de la cible pendant le nettoyage: la `NoClean`fonction

Par défaut, SCons supprime toutes les cibles construites lorsqu'il est appelé avec l' `-c`option pour nettoyer un arbre source de cibles construites. Parfois, cependant, c'est pas ce que vous voulez. Par exemple, vous pouvez supprimer uniquement les fichiers générés intermédiaires (tels que les fichiers d'objets), mais laissez les objectifs finaux (les bibliothèques) intactes. Dans ce cas, vous pouvez utiliser la `NoClean`méthode pour empêcher SCons d'enlever une cible lors d' un nettoyage:

```
env = environnement (RANLIBCOM = '')
lib = env.Library ( 'foo', [ 'f1.c', 'f2.c', 'f3.c'])
env.NoClean (lib)
```

Notez que `libfoo.a` ne figure pas en tant que fichier supprimé:

```
% scons -Q
cc -o -c f1.o f1.c
cc -o -c f2.o f2.c
cc -o -c f3.o f3.c
ar rc libtruc.a f1.o f2.o f3.o
% scons -c
scons: Lecture des fichiers SConscript ...
scons: fait la lecture des fichiers SConscript.
scons: objectifs de nettoyage ...
Suppression f1.o
Suppression f2.o
Suppression f3.o
scons: fait des cibles de nettoyage.
```

13.3. Suppression des fichiers supplémentaires pendant le nettoyage: la `Clean`fonction

Il peut y avoir des fichiers supplémentaires que vous souhaitez supprimer lorsque l' `-c`on utilise l' option, mais qui SCons ne connaît pas parce qu'ils ne sont pas des fichiers cibles normales. Par exemple, peut - être une commande que vous invoquez crée un fichier journal dans le cadre de la construction du fichier cible que vous souhaitez. Vous souhaitez que le fichier journal nettoyé, mais vous ne voulez pas avoir à enseigner SCons que la commande « construit » deux fichiers.

Vous pouvez utiliser la `Clean`fonction d'organiser des fichiers supplémentaires à supprimer lorsque l' `-c`on utilise l' option. Notez, cependant, que la `Clean`fonction prend deux arguments, et le *second* argument est le nom du fichier supplémentaire que vous voulez nettoyé (`foo.log`dans cet exemple):

```
t = Commande ( 'foo.out', 'foo.in', 'construire -o $ TARGET $ SOURCE)
Nettoyer (t, 'foo.log')
```

Le premier argument est la cible avec laquelle vous voulez que le nettoyage de ce fichier supplémentaire associé. Dans l'exemple ci - dessus, nous avons utilisé la valeur de retour de la `Command`fonction, ce qui représente la `foo.out` cible. Maintenant , chaque fois que la`foo.out`cible est nettoyée par l' `-c`option, le `foo.log`fichier sera supprimé ainsi:

```
% scons -Q
construire -o foo.out foo.in
% scons -Q -c
Suppression foo.out
Suppression foo.log
```

Chapitre 14. hiérarchique Builds

Le code source pour les grands projets de logiciels reste rarement dans un seul répertoire, mais il est presque toujours divisé en une hiérarchie des répertoires. L' organisation d' une grande version du logiciel en utilisant SCons implique la création d' une hiérarchie de créer des scripts à l' aide de la `SConscript`fonction.

14.1. `SConscript`Des dossiers

Comme nous l' avons déjà vu, le script de compilation en haut de l'arbre est appelé `SConstruct`. Le niveau supérieur `SConstruct`fichier peut utiliser la `SConscript`fonction d'inclure d' autres scripts subsidiaires dans la construction. Ces scripts peuvent filiale, à son tour, utilisez la `SConscript`fonction d'inclure encore d' autres scripts dans la construction. Par convention, ces scripts filiales sont généralement nommés `SConscript`. Par exemple, un niveau supérieur `SConstruct`fichier peut prendre des dispositions pour quatre scripts filiale à inclure dans la construction comme suit:

```
SConscript ([ 'pilotes / affichage / SConscript',
            'Pilotes / souris / SConscript',
            'Analyseur / SConscript',
            'Utilitaires / SConscript'])
```

Dans ce cas, le `SConstruct`fichier répertorie tous les `SConscript`fichiers dans la version explicitement. (Notez, cependant, que tous les répertoires dans l'arborescence a nécessairement un `SConscript`fichier.) Sinon, le `drivers` sous - répertoire peut contenir un intermédiaire`SConscript`fichier, auquel cas l' `SConscript`appel dans le haut niveau `SConstruct`fichier ressemblerait à ceci:

```
SConscript ([ 'pilotes / SConscript',
            'Analyseur / SConscript',
            'Utilitaires / SConscript'])
```

Et la filiale `SConscript`fichier dans le `drivers`ressemblerait sous - répertoire:

```
SConscript ([ 'affichage / SConscript',
            'Souris / SConscript'])
```

Que vous énumérer tous les `SConscript`fichiers dans le haut niveau `SConstruct`fichier ou placer une filiale `SConscript`fichier dans les répertoires intermédiaires, ou d' utiliser un certain mélange des deux régimes, est à vous et les besoins de votre logiciel.

14.2. Noms de chemin sont relatives au `SConscript`répertoire

Subsidiaires `SConscript`fichiers , il est facile de créer une hiérarchie de construction parce que tous les noms de fichiers et répertoires dans une filiale `SConscript`fichiers sont interprétés par rapport au répertoire dans lequel les `SConscript`vies de fichiers. En règle générale, cela permet au `SConscript`fichier contenant les instructions pour construire un fichier cible à vivre dans le même répertoire que les fichiers source dont la cible sera construit, le rendant facile à mettre à jour la façon dont le logiciel est construit chaque fois que les fichiers sont ajoutés ou supprimés (ou d' autres modifications sont apportées).

Par exemple, supposons que nous voulons construire deux programmes `prog1`et `prog2`dans deux répertoires distincts avec les mêmes noms que les programmes. Une façon typique de le faire serait avec un haut niveau `SConstruct`fichier comme ceci:

```
SConscript ([ 'prog1 / SConscript',
            'Prog2 / SConscript'])
```

Et filiale `SConscript`fichiers qui ressemblent à ceci:

```
env = environnement ()
env.Program ( 'prog1', [ 'main.c', 'foo1.c', 'foo2.c'])
```

Et ça:

```
env = environnement ()
env.Program ( 'prog2', [ 'main.c', 'bar1.c', 'bar2.c'])
```

Puis, quand nous courons SCons dans le répertoire de haut niveau, notre construction ressemble à :

```
% scons -Q
cc -o prog1 / foo1.o -c prog1 / foo1.c
cc -o prog1 / foo2.o -c prog1 / foo2.c
cc -o prog1 / main.o -c prog1 / main.c
cc -o prog1 / prog1 prog1 / main.o prog1 / foo1.o prog1 / foo2.o
cc -o prog2 / bar1.o -c prog2 / bar1.c
cc -o prog2 / bar2.o -c prog2 / bar2.c
```

```
cc -o prog2 / main.o -c prog2 / main.c
cc -o prog2 / prog2 prog2 / main.o prog2 / bar1.o prog2 / bar2.o
```

Notez ce qui suit: D' abord, vous pouvez avoir des fichiers avec le même nom dans plusieurs répertoires, comme main.c dans l'exemple ci - dessus. En second lieu , contrairement à l' utilisation récursive standard Marque , SCons reste dans le répertoire de niveau supérieur (où la `SConstruct`vie de fichiers) et émet des commandes qui utilisent les noms de chemin à partir du répertoire de niveau supérieur pour les fichiers source et cible dans la hiérarchie.

14.3. Chemin de haut niveau dans les noms filiale `SConscript`Fichiers

Si vous avez besoin d'utiliser un fichier à partir d' un autre répertoire, il est parfois plus commode de spécifier le chemin vers un fichier dans un autre répertoire à partir du haut niveau `SConstruct`répertoire, même lorsque vous utilisez ce fichier dans une filiale `SConscript`fichier dans un sous - répertoire. Vous pouvez dire SCons d'interpréter un nom de chemin que par rapport au niveau supérieur `SConstruct`répertoire, pas le répertoire local du `SConscript`fichier, en ajoutant un `#`(signe dièse) au début du nom du chemin:

```
env = environnement ()
env.Program ( 'prog', [ 'main.c', '# lib / foo1.c', 'foo2.c'])
```

Dans cet exemple, le `lib`répertoire est directement sous le haut niveau `SConstruct`répertoire. Si ce qui précède `SConscript`fichier est dans un sous - répertoire nommé `src/prog`, la sortie ressemblerait à ceci:

```
% scons -Q
cc -o lib / foo1.o -c lib / foo1.c
cc -o src / prog / foo2.o -c src / prog / foo2.c
cc -o src / prog / main.o -c src / prog / main.c
cc -o src / prog / prog src / prog / main.o lib / foo1.o src / prog / foo2.o
```

(Notez que le `lib/foo1.o`fichier objet est construit dans le même répertoire que le fichier source. Voir le chapitre 15, *séparatrice Source et répertoires de construction* , ci - dessous, pour plus d' informations sur la façon de construire le fichier objet dans un sous - répertoire différent.)

14.4. Noms absolus Path

Bien sûr, vous pouvez toujours spécifier un nom de chemin absolu pour un fichier - par exemple:

```
env = environnement ()
env.Program ( 'prog', [ 'main.c', '/usr/joe/lib/foo1.c', 'foo2.c'])
```

Ce qui, lorsqu'il est exécuté, produirait:

```
% scons -Q
cc -o src / prog / foo2.o -c src / prog / foo2.c
cc -o src / prog / main.o -c src / prog / main.c
cc -o -c /usr/joe/lib/foo1.o /usr/joe/lib/foo1.c
cc -o src / prog / prog src / prog / main.o /usr/joe/lib/foo1.o src / prog / foo2.o
```

(Comme ce fut le cas avec les noms de chemin haut par rapport, notez que le `/usr/joe/lib/foo1.o`fichier objet est construit dans le même répertoire que le fichier source. Voir le chapitre 15, *séparatrice Source et construire des répertoires* ci - dessous pour plus d' informations sur la façon de construire le fichier objet un autre sous - répertoire).

14,5. Partage des environnements (et d' autres variables) entre `SConscript`fichiers

Dans l'exemple précédent, chacun des filiales `SConscript`fichiers créé son propre environnement de construction en appelant `Environment`séparément. Cela fonctionne évidemment très bien, mais si chaque programme doit être construit avec les mêmes variables de construction, il est lourd et sujette aux erreurs d'initialiser des environnements de construction séparés de la même manière à plusieurs reprises dans chaque filiale `SConscript`fichier.

SCons soutient la possibilité d' *exporter* les variables d'un parent `SConscript`fichier à sa filiale `SConscript`fichiers, ce qui vous permet de partager des valeurs communes initialisés dans toute la hiérarchie de construction.

14.5.1. Exportation de variables

Il existe deux façons d'exporter une variable, comme un environnement de construction, à partir d' un `SConscript`fichier, de sorte qu'il peut être utilisé par d' autres `SConscript`fichiers. Tout d' abord, vous pouvez appeler la `Export` fonction avec une liste de variables, ou une chaîne d'espaces blancs séparés des noms de variables. Chaque appel à `Export`ajouter une ou plusieurs variables à une liste globale des variables qui sont disponibles pour l' importation par d' autres `SConscript`fichiers.

```
env = environnement ()
Export ( 'env')
```

Vous pouvez exporter plus d'un nom variable à la fois:

```
env = environnement ()
debug = ARGUMENTS [ 'debug']
Export ( 'env', 'debug')
```

Parce que l' espace blanc est pas légal dans les noms de variables Python, la `Export`fonction même divisée automatiquement une chaîne en noms séparés pour vous:

```
Export ( 'debug env')
```

Deuxièmement, vous pouvez spécifier une liste de variables à exporter en tant que second argument à l' `SConscript`appel de la fonction:

```
SConscript ( 'src / SConscript', 'env')
```

Ou comme `exports`argument de mot - clé:

```
SConscript ( 'src / SConscript', 'env' exportation =)
```

Ces appels exporter les variables spécifiées aux seuls énumérés `SConscript`fichiers. Vous pouvez toutefois spécifier plus d'un`SConscript`fichier dans une liste:

```
SConscript ([ 'src1 / SConscript',
            'Src2 / SConscript'], 'env' export =)
```

Ceci est fonctionnellement équivalent à appeler la `SConscript`fonction à plusieurs reprises avec le même `exports`raisonnement, un par `SConscript`fichier.

14.5.2. Variables importation

Une fois qu'une variable a été exportée à partir d' un appel`SConscript`fichier, il peut être utilisé dans d' autres `SConscript`fichiers en appelant la `Import`fonction:

```
Import ( 'env')
env.Program ( 'prog', [ 'prog.c'])
```

L' `Import`appel rend l' `env`environnement de la construction à la disposition du `SConscript`fichier, après quoi la variable peut être utilisée pour créer des programmes, les bibliothèques, etc.

Comme la `Export`fonction, la `Import`fonction peut être utilisée avec plusieurs noms de variables:

```
Import ( 'env', 'debug')
env = env.Clone (debug = debug)
env.Program ( 'prog', [ 'prog.c'])
```

Et la `Import`fonction de la même diviser une chaîne le long de l' espace blanc dans les noms de variables séparés:

```
Importation ( 'debug env')
env = env.Clone (debug = debug)
env.Program ( 'prog', [ 'prog.c'])
```

Enfin, comme un cas particulier, vous pouvez importer toutes les variables qui ont été exportées en fournissant un astérisque à la `Import`fonction:

```
Importer('*')
env = env.Clone (debug = debug)
env.Program ( 'prog', [ 'prog.c'])
```

Si vous traitez avec beaucoup de `SConscript`fichiers, cela peut être beaucoup plus simple que de garder des listes arbitraires de variables importées dans chaque fichier.

14.5.3. De retour valeurs à partir d' un `SConscript`fichier

Parfois, vous voulez être en mesure d'utiliser les informations d'une filiale `SConscript`fichier en quelque sorte. Par exemple, supposons que vous voulez créer une bibliothèque de fichiers source dispersés dans un certain nombre de filiales `SConscript`fichiers. Vous pouvez le faire en utilisant la `Return` fonction pour renvoyer des valeurs de la filiale `SConscript`fichiers au fichier appelant.

Si, par exemple, nous avons deux sous - répertoires `foo`et `bar` qui devrait y contribuer à un fichier source à une bibliothèque, ce que nous aimerions être en mesure de faire est de collecter les fichiers objet de la filiale `SConscript`appelle comme ceci:

```
env = environnement ()
Export ( 'env')
objs = []
pour subdir dans [ 'foo', 'bar']:
    o = SConscript ( '% s / SConscript' % subdir)
    objs.append (o)
env.Library ( 'prog', OBJS)
```

Nous pouvons le faire en utilisant la `Return` fonction dans le`foo/SConscript`fichier comme ceci:

```
Import ( 'env')
obj = env.Object ( 'foo.c')
Retour ( 'obj')
```

(Le correspondant `bar/SConscript` fichier devrait être assez évident.) Puis , quand nous courons SCons , les fichiers objets des sous - répertoires sont tous archivés filiale correctement dans la bibliothèque souhaitée:

```
% scons -Q
cc s bar / bar.o -c bar / bar.c
cc -o foo / foo.o -c foo / foo.c
ar rc libprog.a foo / foo.o bar / bar.o
ranlib libprog.a
```

Chapitre 15. Séparation Source et répertoires de construction

Il est souvent utile de conserver des fichiers construits complètement séparés des fichiers source. En SCons , cela se fait habituellement en créant un ou plusieurs séparés des *arborescences variante* qui sont utilisés pour contenir les fichiers d'objets construits, des bibliothèques et des programmes exécutables, etc. pour une saveur spécifique, ou une variante, de construction. SCons propose deux façons de le faire, l' une par la `SConscript`fonction que nous avons déjà vu, et le second par un système plus souple `VariantDir`fonction.

Une note historique: la `VariantDir`fonction utilisée pour appeler `BuildDir`. Ce nom est toujours pris en charge , mais a été dépréciée parce que la SCons fonctionnalité diffère du modèle d'un « répertoire de construction » mis en œuvre par d' autres systèmes de construction comme GNU

autotools.

15.1. Spécification d' un arbre Variant Directory en tant que partie d'un `SConscript`appel

La façon la plus simple d'établir un arbre de répertoire variante utilise le fait que la façon habituelle de mettre en place une hiérarchie de construction est d'avoir un `SConscript`fichier dans le sous - répertoire source. Si vous passez ensuite un `variant_dir`argument à l'`SConscript`appel de la fonction:

```
SConscript ( 'src / SConscript', variant_dir = 'construire')
```

SCons construira ensuite tous les fichiers dans le `build`sous - répertoire:

```
% ls src
SConscript hello.c
% scons -Q
cc -o construire / hello.o -c build / hello.c
cc -o build / build bonjour / hello.o
% ls build
SConscript bonjour hello.c hello.o
```

Mais attendez une minute - ce qui se passe ici? SCons a créé le fichier objet `build/hello.o` dans le `build`sous - répertoire, comme prévu. Mais même si notre `hello.c`demeure fichier dans le `src`sous - répertoire,SCons a effectivement compilé un `build/hello.c`fichier pour créer le fichier objet.

Ce qui est arrivé est que SCons a *dupliqué* le `hello.c`fichier à partir du `src`sous - répertoire du `build`sous - répertoire, et construit le programme à partir de là. La section suivante explique pourquoi SConsfait cela.

15.2. Pourquoi SCons Doublons fichiers source dans un arbre Variant Répertoire

SCons doublons fichiers source dans des arborescences variante , car il est le moyen le plus simple de garantir une construction correcte *quel que soit inclure-fichiers chemins de répertoire, des références relatives entre les fichiers ou support d'outil pour placer les fichiers dans des endroits différents* , et la SCons philosophie est de, par par défaut, garantir une construction correcte dans tous les cas.

La raison la plus directe de dupliquer les fichiers source dans les répertoires de variantes est tout simplement que certains outils (surtout les anciennes versions) sont écrits pour ne construire que leurs fichiers de sortie dans le même répertoire que les fichiers source. Dans ce cas, les choix sont soit pour construire le fichier de sortie dans le répertoire source et le déplacer vers le répertoire variant ou dupliquer les fichiers source dans le répertoire variant.

En outre, des références relatives entre les fichiers peuvent causer des problèmes si nous ne faisons pas dupliquer la hiérarchie des fichiers source dans le répertoire variant. Vous pouvez le voir à l' œuvre dans l' utilisation du C préprocesseur `#include` mécanisme avec des guillemets doubles, pas équerres:

```
#include « fichier.h »
```

Le *de facto* un comportement standard pour la plupart des compilateurs C dans ce cas est d'abord regarder dans le même répertoire que le fichier source qui contient la `#include`ligne, puis de regarder dans les répertoires dans le chemin de recherche de préprocesseur. Ajoutez à cela que la SCons mise en œuvre du soutien aux référentiels de code (décrits ci - dessous) ne signifie pas tous les fichiers se trouvent dans la même hiérarchie des répertoires, et la façon la plus simple pour vous assurer que le droit inclure le fichier se trouve est de dupliquer la source fichiers dans le répertoire variant, qui fournit une version correcte quel que soit l'emplacement d' origine (s) des fichiers source.

Bien que la duplication de fichier source garantit une construction correcte même dans ces cas-fin, il *peut* généralement être en toute sécurité désactivée. La section suivante décrit comment vous pouvez désactiver la duplication des fichiers source dans le répertoire variant.

15.3. Dire SCons à ne dupliquer les fichiers source dans l'arborescence Annuaire Variant

Dans la plupart des cas et la plupart des jeux d'outils, SCons peut placer ses fichiers cibles dans un sous - répertoire de construction *sans* dupliquer les fichiers source et tout fonctionne très bien. Vous pouvez désactiver le défaut SCons comportement en spécifiant `duplicate=0`lorsque vous appelez la `SConscript`fonction:

```
SConscript ( 'src / SConscript', variant_dir = 'build', en double = 0)
```

Lorsque cet indicateur est spécifié, SCons utilise le répertoire variant comme la plupart des gens attendent - à savoir les fichiers de sortie sont placés dans le répertoire variant alors que les fichiers source restent dans le répertoire source:

```
% ls src
SConscript
Bonjour c
% scons -Q
cc -c src / hello.c -o build / hello.o
cc -o build / build bonjour / hello.o
% ls build
Bonjour
hello.o
```

15.4. la `VariantDir`fonction

Utilisez la `VariantDir`fonction d'établir que les fichiers cibles doivent être construits dans un répertoire distinct des fichiers source:

```
VariantDir ( 'build', 'src')
env = environnement ()
env.Program ( 'build / hello.c')
```

Notez que lorsque vous n'êtes pas en utilisant un `SConscript`fichier dans le `src`sous - répertoire, vous devez effectivement préciser que le programme doit être construit à partir du `build/hello.c` fichier qui SCons dupliquera dans le `build`sous - répertoire.

Lorsque vous utilisez la `VariantDir`fonction directement, SCons doublons encore les fichiers source dans le répertoire variant par défaut:

```
% ls src
Bonjour c
% scons -Q
cc -o construire / hello.o -c build / hello.c
cc -o build / build bonjour / hello.o
% ls build
bonjour hello.c hello.o
```

Vous pouvez spécifier le même `duplicate=0`argument que vous pouvez spécifier pour un `SConscript`appel:

```
VariantDir ( 'build', 'src', en double = 0)
env = environnement ()
env.Program ( 'build / hello.c')
```

Dans ce cas , SCons désactivera la duplication des fichiers source:

```
% ls src
Bonjour c
% scons -Q
cc -o construire / hello.o -c src / hello.c
cc -o build / build bonjour / hello.o
% ls build
bonjour hello.o
```

15,5. L' utilisation `VariantDir`avec un `SConscript`fichier

Même lorsque vous utilisez la `VariantDir`fonction, il est beaucoup plus naturel de l' utiliser avec une filiale `SConscript`fichier. Par exemple, si les `src/SConscript` regards comme celui - ci:

```
env = environnement ()
env.Program ( 'hello.c')
```

Ensuite , notre `SConstruct`fichier pourrait ressembler à :

```
VariantDir ( 'build', 'src')
SConscript ( 'build / SConscript')
```

Cédant la sortie suivante:

```
% ls src
SConscript hello.c
% scons -Q
cc -o construire / hello.o -c build / hello.c
cc -o build / build bonjour / hello.o
% ls build
SConscript bonjour hello.c hello.o
```

Notez que ceci est tout à fait équivalente à l'utilisation de `SConscript`ce que nous avons appris dans la section précédente.

15.6. L' utilisation `Glob`avec`VariantDir`

La `Glob`fonction de correspondance de modèle de nom de fichier fonctionne comme d' habitude lors de l' utilisation `VariantDir`. Par exemple, si les `src/SConscript` regards comme celui - ci:

```
env = environnement ()
env.Program ( 'bonjour', Glob ( '*. c'))
```

Ensuite , avec le même `SConstruct`fichier que dans la section précédente, et les fichiers source `f1.c` et `f2.c`dans src, nous verrions la sortie suivante:

```
% ls src
SConscript f1.c f2.c f2.h
% scons -Q
cc -o build / f1.o -c build / f1.c
cc -o build / f2.o -c build / f2.c
cc -o construire / bonjour build / f1.o build / f2.o
% ls build
SConscript f1.c f1.o f2.c f2.h f2.o bonjour
```

La `Glob`fonction retourne nœuds dans l' `build/`arbre, comme on pouvait s'y attendre.

Chapitre 16. Variante Builds

L' `variant_dir`argument de mot - clé de la `SConscript`fonction fournit tout ce dont nous avons besoin pour montrer combien il est facile de créer variante builds grâce SCons . Supposons, par exemple, que nous voulons construire un programme pour les plates - formes Windows et Linux, mais que nous voulons construire dans un répertoire partagé avec des répertoires de construction côte à côte séparés pour les versions Windows et Linux du programme.

```
plate-forme = ARGUMENTS.get ( 'OS', la plate-forme ())

include = "# export / $ PLATEFORME / include"
lib = "# export / $ PLATEFORME / lib"
bin = "# export / $ PLATEFORME / bin"

env = environnement (platform = plate-forme,
                  BINDIR = bin,
                  INCDIR = comprennent,
                  LIBDIR = lib,
                  CPPPATH = [inclure],
```

```
                LIBPATH = [lib],
                'monde' LIBS =)

Export ( 'env')

env.SConscript ( 'src / SConscript', variant_dir = 'build / $ PLATFORM)
```

Ce fichier SConstruct, lorsqu'il est exécuté sur un système Linux, les rendements:

```
% scons -Q OS=linux
Fichier d'installation: "build / linux / monde / world.h" comme "export / linux / include / world.h"
cc -o build / linux / bonjour / hello.o -c -Iexport / linux / include build / linux / bonjour / hello.c
cc -o build / linux / monde / world.o -c -Iexport / linux / include build / linux / monde / world.c
ar rc build / linux / world / libworld.a build / linux / monde / world.o
ranlib build / linux / monde / libworld.a
Fichier d'installation: "build / linux / monde / libworld.a" comme "export / linux / lib / libworld.a"
cc -o build / linux / bonjour / bonjour build / linux / bonjour / hello.o -Lexport / linux / lib -lworld
Fichier d'installation: "build / linux / bonjour / bonjour" comme "export / linux / bin / bonjour"
```

Le même fichier SConstruct sur Windows construirait:

```
C: \>scons -Q OS=windows
Fichier d'installation: "construire / windows / monde / world.h" comme "export / fenêtres / include / world.h"
cl /Fobuild\windows\hello\hello.obj / c construire \ windows \ bonjour \ hello.c / nologo / IExport windows \ \ include
cl /Fobuild\windows\world\world.obj / c construire \ windows \ world \ world.c / nologo / IExport windows \ \ include
lib / nologo /OUT:build\windows\world\world.lib construire \ windows \ world \ world.obj
Fichier d'installation: "construire / windows / monde / world.lib" comme "exportation / windows / lib / world.lib"
lien / nologo /OUT:build\windows\hello\hello.exe / LIBPATH: fenêtres d'exportation \ \ lib build world.lib \ windows \ bonjour \ hello.obj
embedManifestExeCheck (cible, source, env)
Fichier d'installation: "construire / windows / bonjour / hello.exe" comme "export / windows / bin / hello.exe"
```

Chapitre 17. Internationalisation et localisation avec gettext

Le `gettext`jeu d' outils supporte l' internationalisation et la localisation des projets sur la base SCons. Builders fournies par `gettext`génération automatize et mises à jour des fichiers de traduction. Vous pouvez gérer les traductions et les modèles de traduction de la même façon dont il est fait avec autotools.

17.1. Conditions préalables

Pour suivre les exemples fournis dans ce chapitre configurer votre système d'exploitation pour soutenir deux ou plusieurs langues. Dans les exemples qui suivent , nous utilisons des endroits `en_US`, `de_DE`et`pl_PL`.

Assurez -vous que vous avez des utilitaires GNU gettext installés sur votre système.

Pour modifier les fichiers de traduction que vous pouvez installer poedit éditeur.

17.2. simple projet

Commençons par un projet très simple, le programme « Bonjour tout le monde », par exemple

```
/* Bonjour c */
#include <stdio.h>
int main (int argc, char * argv [])
{
  printf ( "Bonjour tout le monde \ n");
  return 0;
}
```

Préparer un `SConstruct`pour compiler le programme comme d' habitude.

```
# SConstruct
env = environnement ()
bonjour = Programme ([ "hello.c"])
```

Maintenant , nous allons convertir le projet à un multi-langues. Si vous ne possédez pas déjà utilitaires GNU gettext installés, les installer à partir de votre dépôt de paquets preffered, ou télécharger à partir http://ftp.gnu.org/gnu/gettext/ . Aux fins de cet exemple, vous devriez avoir après trois endroits installés sur votre système: `en_US`, `de_DE`et`pl_PL`. Sur debian, par exemple, vous pouvez activer certains paramètres régionaux par **dpkg-reconfigure locales** .

Tout d' abord préparer le `hello.c`programme d'internationalisation. Modifiez le code précédent il se lit comme suit:

```
/* Bonjour c */
#include <stdio.h>
#include <libintl.h>
#include <locale.h>
int main (int argc, char * argv [])
{
  bindtextdomain ( "bonjour", "locale");
  setlocale (LC_ALL, "");
  textdomain ( "bonjour");
  printf (gettext ( "Bonjour tout le monde \ n"));
  return 0;
}
```

Recettes détaillées pour une telle conversion se trouvent à http://www.gnu.org/software/gettext/manual/gettext.html#Sources . Le `gettext("...")`a deux objectifs. Tout d' abord, il marque des messages pour le **xgettext (1)** programme, que nous utiliserons pour extraire des sources les messages pour la localisation. En second lieu , il appelle la `gettext`bibliothèque pour traduire le fonctionnement interne un message lors de l' exécution.

Maintenant , nous allons donner des instructions SCons comment générer et maintenir des fichiers de traduction. Pour cela, utilisez le `Translate`constructeur et le `MOFiles`constructeur. Le premier prend les fichiers source, extrait des messages internationalisés d'eux, crée ce qu'on appelle le `POT`fichier (modèle de traduction), et crée ensuite les `PO`fichiers de traduction, un pour chaque langue demandée. Plus tard, au cours du cycle de vie de développement, le constructeur conserve tous ces fichiers mis à jour. Le `MOFiles`constructeur compile les `PO`fichiers à forme binaire. Ensuite , installez les `MO`fichiers sous le répertoire appelé `locale`.

Le complété `SConstruct`est la suivante:

```
# SConstruct
env = environnement (outils = [ 'default', 'gettext'])
bonjour = env.Program ([ "hello.c"])
env [ 'XGETTEXTFLAGS'] = [
  '--package-name =% s' % 'bonjour',
  '--package-version =% s' % '1,0',
]
po = env.Translate ([ "pl", "fr", "de"], [ "hello.c"], POAUTOINIT = 1)
mo = env.MOFiles (po)
InstallAs ([ "locale / fr / LC_MESSAGES / hello.mo"], [ "en.mo"])
InstallAs ([ "pl.mo"] [ "locale / pl / LC_MESSAGES / hello.mo"])
InstallAs ([ "locale / de / LC_MESSAGES / hello.mo"], [ "de.mo"])
```

Générer les fichiers de traduction avec **scons po-mise à jour** . Vous devriez voir la sortie de SCons simillar à ceci:

```
user @ host: scons $ po-mise à jour
scons: Lecture des fichiers SConscript ...
scons: fait la lecture des fichiers SConscript.
scons: objectifs de construction ...
Saisie '/ home / ptomulik / projets / tmp'
xgettext --package-name = bonjour --package version = 1.0 -o - hello.c
Laissant '/ home / ptomulik / projets / tmp'
Writting 'messages.pot' (nouveau fichier)
msginit --no-traducteur pl -l -i -o messages.pot pl.po
Créé pl.po.
msginit --no-traducteur en -l -i -o messages.pot en.po
Créé en.po.
msginit --no-traducteur -l de -i messages.pot -o de.po
Créé de.po.
scons: fait des objectifs de construction.
```

Si tout est correct, vous devriez voir suiviez les nouveaux fichiers.

```
user @ host: $ ls * .po *
de.po en.po messages.pot pl.po
```

Ouvrez `en.po`dans **poedit** et de fournir la traduction en anglais au message `"Hello world\n"`. Faites la même chose pour `de.po`(deutsch) et`pl.po`(polonais). Que les traductions soient, par exemple:

- `en: "Welcome to beautiful world!\n"`
- `de: "Hallo Welt!\n"`
- `pl: "Witaj swiecie!\n"`

Maintenant compiler le projet en exécutant **scons** . La sortie devrait ressembler à ceci:

```
user @ host: $ scons
scons: Lecture des fichiers SConscript ...
scons: fait la lecture des fichiers SConscript.
scons: objectifs de construction ...
msgfmt -c -o de.mo de.po
msgfmt -c -o en.mo en.po
gcc -o hello.o -c hello.c
gcc -o bonjour hello.o
Fichier d'installation: "de.mo" comme "locale / de / LC_MESSAGES / hello.mo"
Fichier d'installation: "en.mo" comme "locale / fr / LC_MESSAGES / hello.mo"
msgfmt -c -o pl.mo pl.po
Fichier d'installation: "pl.mo" comme "locale / pl / LC_MESSAGES / hello.mo"
scons: fait des objectifs de construction.
```

SCons compilé automatiquement les `PO`fichiers au format binaire `MO`, et les `InstallAs`lignes installées ces fichiers sous `locale`dossier.

Votre programme devrait être prêt. Vous pouvez l'essayer comme suit (linux):

```
user @ host: $ LANG = en_US.UTF-8 ./hello
Bienvenue dans beau monde


user @ host: $ LANG = de_DE.UTF-8 ./hello
Hallo Welt


user @ host: $ LANG = pl_PL.UTF-8 ./hello
witaj swiecie
```

Pour démontrer la vie ultérieure des fichiers de traduction, nous allons changer la traduction polonaise (**poedit pl.po**) à `"Witaj drogi swiecie\n"`. Exécuter **scons** pour voir comment scons réagit à cette

```
user @ host: $ scons
scons: Lecture des fichiers SConscript ...
scons: fait la lecture des fichiers SConscript.
scons: objectifs de construction ...
msgfmt -c -o pl.mo pl.po
Fichier d'installation: "pl.mo" comme "locale / pl / LC_MESSAGES / hello.mo"
```

```
scons: fait des objectifs de construction.
```

Maintenant, ouvrez `hello.c`et ajouter une autre `printf`ligne avec un nouveau message.

```
/* Bonjour c */
#include <stdio.h>
#include <libintl.h>
#include <locale.h>
int main (int argc, char * argv [])
{
  bindtextdomain ( "bonjour", "locale");
  setlocale (LC_ALL, "");
  textdomain ( "bonjour");
  printf (gettext ( "Bonjour tout le monde \ n"));
  printf (gettext ( "et au revoir \ n"));
  return 0;
}
```

Compiler projet avec **scons** . Cette fois, le **msgmerge (1)** programme est utilisé par SCons mettre à jour le `po`fichier. La sortie de la compilation est comme:

```
user @ host: $ scons
scons: Lecture des fichiers SConscript ...
scons: fait la lecture des fichiers SConscript.
scons: objectifs de construction ...
Saisie '/ home / ptomulik / projets / tmp'
xgettext --package-name = bonjour --package version = 1.0 -o - hello.c
Laissant '/ home / ptomulik / projets / tmp'
Writting 'messages.pot' (messages dans le dossier étaient pas à jour)
msgmerge --update de.po messages.pot
... terminé.
msgfmt -c -o de.mo de.po
msgmerge --update en.po messages.pot
... terminé.
msgfmt -c -o en.mo en.po
gcc -o hello.o -c hello.c
gcc -o bonjour hello.o
Fichier d'installation: "de.mo" comme "locale / de / LC_MESSAGES / hello.mo"
Fichier d'installation: "en.mo" comme "locale / fr / LC_MESSAGES / hello.mo"
msgmerge --update pl.po messages.pot
... terminé.
msgfmt -c -o pl.mo pl.po
Fichier d'installation: "pl.mo" comme "locale / pl / LC_MESSAGES / hello.mo"
scons: fait des objectifs de construction.
```

L'exemple suivant montre ce qui se passe si nous changeons le code source de telle sorte que les messages internationalisés ne changent pas. La réponse est qu'aucun des fichiers de traduction (`pot`, `po`) sont touchés (aucun changement de contenu, pas de temps de création / modification changé et ainsi de suite). Nous allons append une autre ligne au programme (après la dernière printf), de sorte que son code devient:

```
/* Bonjour c */
#include <stdio.h>
#include <libintl.h>
#include <locale.h>
int main (int argc, char * argv [])
{
  bindtextdomain ( "bonjour", "locale");
  setlocale (LC_ALL, "");
  textdomain ( "bonjour");
  printf (gettext ( "Bonjour tout le monde \ n"));
  printf (gettext ( "et au revoir \ n"));
  printf ( "---------------- \ n");
  retourner un;
}
```

Compiler le projet. Vous verrez sur votre écran

```
user @ host: $ scons
scons: Lecture des fichiers SConscript ...
scons: fait la lecture des fichiers SConscript.
scons: objectifs de construction ...
Saisie '/ home / ptomulik / projets / tmp'
xgettext --package-name = bonjour --package version = 1.0 -o - hello.c
Laissant '/ home / ptomulik / projets / tmp'
Non writting « messages.pot » (messages dans le fichier trouvé pour être à jour)
gcc -o hello.o -c hello.c
gcc -o bonjour hello.o
scons: fait des objectifs de construction.
```

Comme vous le voyez, les messages internationalisés ditn't changement, de sorte que le `pot`et le reste des fichiers de traduction ont même pas été touché.

Chapitre 18. Rédaction de vos propres constructeurs

Bien que SCons fournit de nombreuses méthodes utiles pour la construction de produits logiciels courants (programmes, bibliothèques, documents, etc.), vous voulez souvent être en mesure de construire un autre type de fichier non pris en charge directement par SCons . Heureusement, SCons il est très facile de définir vos propres `Builder`objets pour tous les types de fichiers personnalisés que vous voulez construire. (En fait, les SCons interfaces pour créer des `Builder`objets sont suffisamment souples et assez facile à utiliser que tous les les SCons intégré des `Builder`objets créés à l' aide des mécanismes décrits dans cette section.)

18.1. Builders écriture qui exécutent des commandes externes

Le plus simple `Builder`de créer est celui qui exécute une commande externe. Par exemple, si nous voulons construire un fichier de sortie en exécutant le contenu du fichier d'entrée par une commande nommée`foobuild`, la création qui `Builder`pourrait ressembler à :

```
BLD = Builder (action = 'foobuild <$ SOURCE> $ TARGET')
```

Toute la ligne ci - dessus n'est de créer un autoportant `Builder`objet. La section suivante nous montrera comment utiliser réellement.

18.2. Fixation d'un constructeur à un`Construction Environment`

Un `Builder`objet n'est pas utile jusqu'à ce qu'il soit attaché à un `construction environment` afin que nous puissions l' appeler pour organiser à construire des fichiers. Cela se fait par le dans un environnement. La variable est un dictionnaire Python qui associe les noms par lesquels vous souhaitez appeler divers objets aux objets eux - mêmes. Par exemple, si l' on veut appeler le nous venons de définir le nom , notre fichier pourrait ressembler à : `$BUILDERS` `construction variable$BUILDERSBuilderBuilderFooSConstruct`

```
BLD = Builder (action = 'foobuild <$ SOURCE> $ TARGET')
env = environnement (CONSTRUCTEURS = { 'Foo': BLD})
```

Avec le `Builder`joint à notre `construction environment` avec le nom `Foo`, nous pouvons maintenant appeler réellement comme ceci:

```
env.Foo ( 'file.foo', 'file.input')
```

Puis , quand nous courons SCons il ressemble:

```
% scons -Q
foobuild <file.input> file.foo
```

Notez toutefois que la valeur par défaut `$BUILDERS` variable dans un `construction environment` est livré avec un jeu par défaut d' `Builder`objets déjà définis: `Program`, `Library`, etc. Et quand nous fixons explicitement la `$BUILDERS`variable lorsque nous créons `construction environment`, la valeur par défaut `Builder`de ne font plus partie de l'environnement:

```
BLD = Builder (action = 'foobuild <$ SOURCE> $ TARGET')
env = environnement (CONSTRUCTEURS = { 'Foo': BLD})
env.Foo ( 'file.foo', 'file.input')
env.Program ( 'hello.c')
```

```
% scons -Q
AttributeError: objet « SConsEnvironment » n'a pas d'attribut « programme »:
  Fichier "/ home / mon / projet / SConstruct", ligne 4:
    env.Program ( 'hello.c')
```

Pour être en mesure d'utiliser nos propres définis des `Builder`objets et la valeur par défaut des `Builder`objets dans le même `construction environment`, vous pouvez ajouter à la `$BUILDERS`variable en utilisant la `Append`fonction:

```
env = environnement ()
BLD = Builder (action = 'foobuild <$ SOURCE> $ TARGET')
env.Append (CONSTRUCTEURS = { 'Foo': BLD})
env.Foo ( 'file.foo', 'file.input')
env.Program ( 'hello.c')
```

Ou vous pouvez explicitement définir la clé appropriée nommée dans le `$BUILDERS`dictionnaire:

```
env = environnement ()
BLD = Builder (action = 'foobuild <$ SOURCE> $ TARGET')
env [ 'BUILDERS'] [ 'Foo'] = bld
env.Foo ( 'file.foo', 'file.input')
env.Program ( 'hello.c')
```

De toute façon, même `construction environment` peut alors utiliser à la fois la nouvelle définition `Foo Builder` et la valeur par défaut : `Program`Builder

```
% scons -Q
foobuild <file.input> file.foo
cc -o -c hello.o hello.c
cc -o bonjour hello.o
```

18.3. Letting SCons gérer les extensions de fichier

En fournissant des informations supplémentaires lorsque vous créez un `Builder`, vous pouvez laisser SCons ajouter les suffixes des fichiers appropriés à la cible et / ou le fichier source. Par exemple, plutôt que d' avoir à spécifier explicitement que vous voulez le `Foo Builder`pour construire le `file.foo` fichier cible à partir du `file.input`fichier source, vous pouvez donner les `.foo` et `.input`suffixes au `Builder`, faisant des appels plus compacts et lisibles au `Foo Builder`:

```
BLD = Builder (action = 'foobuild <$ SOURCE> $ TARGET',
               suffixe = '.foo',
               src_suffix = '.input')
env = environnement (CONSTRUCTEURS = { 'Foo': BLD})
env.Foo ( 'file1')
env.Foo ( 'file2')
```

```
% scons -Q
foobuild <file1.input> file1.foo
foobuild <file2.input> file2.foo
```

Vous pouvez également fournir un `prefix`argument de mot - clé s'il est approprié d'avoir SCons append un préfixe au début des noms de fichiers cible.

18.4. Les constructeurs qui exécutent des fonctions Python

En SCons , vous ne devez pas appeler une commande externe pour créer un fichier. Vous pouvez, au lieu, définir une fonction Python qu'un `Builder`objet peut invoquer pour construire votre fichier cible (ou fichiers). Une telle `builder function`définition ressemble à :

```
def build_function (cible, source, env):
    Code # pour construire « cible » de « source »
    Aucun retour
```

Les arguments d'un `builder function`sont les suivants :

cible

Une liste d'objets Node représentant la cible ou des cibles à construire par cette fonction constructeur. Les noms de fichiers de ces cibles (s) peuvent être extraits en utilisant Python `str`fonction.

la source

Une liste d'objets Node représentant les sources à utiliser par cette fonction de constructeur pour construire les cibles. Les noms de fichiers de ces sources (s) peuvent être extraits en utilisant Python `str`fonction.

env

L' `construction environment`utilisés pour la construction de la cible (s). La fonction constructeur peut utiliser l' une des variables de construction de l'environnement de quelque façon d'influer sur la façon dont il construit les cibles.

La fonction constructeur doit retourner une `0`ou `None`valeur si la cible (s) sont construits avec succès. La fonction constructeur peut soulever une exception ou retourner une valeur non nulle pour indiquer que la construction échoue.

Une fois que vous avez défini la fonction Python qui construirez votre fichier cible, la définition d' un `Builder`objet car il est aussi simple que spécifiant le nom de la fonction, au lieu d'une commande externe, comme `Builder`l » `action` argument:

```
def build_function (cible, source, env):
    Code # pour construire « cible » de « source »
    Aucun retour
BLD = Builder (action = build_function,
               suffixe = '.foo',
               src_suffix = '.input')
env = environnement (CONSTRUCTEURS = { 'Foo': BLD})
env.Foo ( 'fichier')
```

Et remarquez que les changements de sortie légèrement, ce qui reflète le fait qu'une fonction Python, pas une commande externe, est maintenant appelée à construire le fichier cible:

```
% scons -Q
build_function ([ "file.foo"], [ "file.input"])
```

18.5. Les constructeurs qui créent des actions en utilisant un`Generator`

SCons objets Builder peuvent créer une action « à la volée » en utilisant une fonction appelée `generator`. Cela offre une grande flexibilité pour construire juste la liste de droite de commandes pour construire votre cible. A `generator`ressemble:

```
def generate_actions (source, cible, env, for_signature):
    retourner 'foobuild <% s>% s' % (cible [0], la source [0])
```

Les arguments d'un `generator`sont les suivants :

la source

Une liste d'objets représentant les sources de noeud devant être construit par la commande ou toute autre action générée par cette fonction. Les noms de fichiers de ces sources (s) peuvent être extraits en utilisant Python `str`fonction.

cible

Une liste d'objets noeud représentant la cible ou les cibles devant être construit par la commande ou toute autre action générée par cette fonction. Les noms de fichiers de ces cibles (s) peuvent être extraits en utilisant Python `str`fonction.

env

L' `construction environment`utilisés pour la construction de la cible (s). Le générateur peut utiliser l' une des variables de construction de l'environnement de quelque façon de déterminer quelle commande ou toute autre action pour revenir.

à signer

Un indicateur qui spécifie si le générateur est appelé à contribuer à une signature de construction, par opposition à l'exécution de la commande réellement.

La `generator`doit renvoyer une chaîne de commande ou toute autre action qui sera utilisé pour construire la cible spécifiée (s) à partir de la source spécifiée (s).

Une fois que vous avez défini un `generator`, vous créez un `Builder`pour l' utiliser en spécifiant l'argument mot - clé du générateur au lieu de `action`.

```
def generate_actions (source, cible, env, for_signature):
    retourner 'foobuild <% s>% s' % (source [0], target [0])
bld = Builder (générateur = generate_actions,
               suffixe = '.foo',
               src_suffix = '.input')
env = environnement (CONSTRUCTEURS = { 'Foo': BLD})
env.Foo ( 'fichier')
```

```
% scons -Q
foobuild <file.input> file.foo
```

Notez qu'il est illégal de spécifier à la fois un `action` et un `generator`pour `Builder`.

18.6. Les constructeurs qui Modifier la cible ou source listes à l'aide d'un`Emitter`

SCons prend en charge la capacité d'un générateur pour modifier la liste des cible (s) à partir de la source spécifiée (s). Vous faites cela en définissant une `emitter`fonction qui prend comme arguments la liste des cibles transmises au constructeur, la liste des sources Transmis à le constructeur et l'environnement de la construction. La fonction d'émetteur doit renvoyer les listes modifiées de cibles qui devraient être construites et les sources dont les objectifs seront construits.

Par exemple, supposons que vous souhaitez définir un constructeur qui appelle toujours un `foobuild`programme et que vous souhaitez ajouter automatiquement un nouveau fichier cible nommé `new_target`et un nouveau fichier source nommé à `new_source` chaque fois qu'il est appelé. Le `SConstruct`fichier pourrait ressembler à ceci:

```
def (modify_targets cible, la source, env):
    target.append ( 'new_target')
    source.append ( 'new_source')
    retour cible, de la source
BLD = Builder (action = 'foobuild CIBLES de $ - $ SOURCES',
               suffixe = '.foo',
               src_suffix = '.input',
               émetteur = modify_targets)
env = environnement (CONSTRUCTEURS = { 'Foo': BLD})
env.Foo ( 'fichier')
```

Et produirait la sortie suivante:

```
% scons -Q
foobuild file.foo new_target - file.input new_source
```

Une chose très flexible que vous pouvez faire est d' utiliser une variable de construction pour spécifier différentes fonctions d'émetteur pour différentes variables de construction. Pour ce faire, spécifiez une chaîne contenant une extension variable de construction comme l'émetteur lorsque vous appelez la `Builder`fonction, et définir cette variable de construction à la fonction d'émetteur souhaité dans différents environnements de construction:

```
BLD = Builder (action = 'ma_commande $ SOURCES> $ TARGET',
               suffixe = '.foo',
               src_suffix = '.input',
               = 'émetteur MY_EMITTER de $)
def modify1 (cible, source, env):
    cible de retour, la source + [ 'modify1.in']
def modify2 (cible, source, env):
    cible de retour, la source + [ 'modify2.in']
env1 = Environnement (CONSTRUCTEURS = { 'Foo': BLD},
                      MY_EMITTER = modify1)
ENV2 = Environnement (CONSTRUCTEURS = { 'Foo': BLD},
                      MY_EMITTER = modify2)
env1.Foo ( 'file1')
env2.Foo ( 'file2')
import os
ENV1 [ 'ENV'] [ 'PATH'] = ENV2 [ 'ENV'] [ 'PATH'] + os.pathsep + os.getcwd ()
ENV2 [ 'ENV'] [ 'PATH'] = ENV2 [ 'ENV'] [ 'PATH'] + os.pathsep + os.getcwd ()


BLD = Builder (action = 'ma_commande $ SOURCES> $ TARGET',
               suffixe = '.foo',
               src_suffix = '.input',
               = 'émetteur MY_EMITTER de $)
def modify1 (cible, source, env):
    cible de retour, la source + [ 'modify1.in']
def modify2 (cible, source, env):
    cible de retour, la source + [ 'modify2.in']
env1 = Environnement (CONSTRUCTEURS = { 'Foo': BLD},
                      MY_EMITTER = modify1)
ENV2 = Environnement (CONSTRUCTEURS = { 'Foo': BLD},
                      MY_EMITTER = modify2)
env1.Foo ( 'file1')
env2.Foo ( 'file2')
```

Dans cet exemple, les `modify1.in` et les `modify2.in`fichiers ajoutés à la liste des sources des différentes commandes:

```
% scons -Q
ma_commande file1.input modify1.in> file1.foo
ma_commande file2.input modify2.in> file2.foo
```

18.7. Où mettre votre Custom Builders et outils

Les `site_scons`répertoires donnent un endroit pour mettre des modules Python et des paquets que vous pouvez importer dans vos `SConscript`fichiers (`site_scons`), outils complémentaires qui peuvent intégrer dans SCons (`site_scons/site_tools`) et un `site_scons/site_init.py`fichier qui est lu avant tout `SConstruct`ou `SConscript`fichier, vous permettant de changer SCons « le comportement par défaut de s.

Chaque type de système (Windows, Mac, Linux, etc.) recherche un ensemble canonique de répertoires pour site_scons; voir la page de manuel pour plus de détails. Le haut niveau site_scons dir de SConstruct est toujours recherché dernier, et son répertoire est placé d'abord dans la trajectoire de l'outil il l'emporte sur tous les autres.

Si vous obtenez un outil de quelque part (le SCons wiki ou un tiers, par exemple) et que vous souhaitez l' utiliser dans votre projet, un`site_scons`répertoire est l'endroit le plus simple de le mettre. Outils sont disponibles en deux saveurs; soit une fonction Python qui fonctionne sur un `Environment`ou d' un module de python ou emballage contenant deux fonctions, `exists()`et `generate()`.

Un outil unique fonction peut simplement être inclus dans votre`site_scons/site_init.py`fichier où il sera analysé et mis à disposition pour une utilisation. Par exemple, vous pourriez avoir un`site_scons/site_init.py`fichier comme celui - ci:

```
def TOOL_ADD_HEADER (env):
   « » « Un outil pour ajouter un en-tête de $ au fichier HEADER source » « »
   add_header = Builder (action = [ 'echo "$ HEADER"> $ TARGET',
                                    'Cat $ SOURCE >> TARGET $'])
   env.Append (CONSTRUCTEURS = { 'AddHeader': add_header})
   env [ 'HEADER'] = '' # set valeur par défaut
```

et `SConstruct`comme celui - ci:

```
# Utilisez TOOL_ADD_HEADER de site_scons / site_init.py
env = environnement (outils = [ 'default', TOOL_ADD_HEADER], HEADER = "=====")
env.AddHeader ( 'tgt', 'src')
```

La `TOOL_ADD_HEADER`méthode d'outil sera appelé à ajouter l' `AddHeader`outil à l'environnement.

Un outil supplémentaire à part entière avec `exists()`et `generate()`méthodes peut être installé comme un module dans le fichier`site_scons/site_tools/toolname.py`ou un package dans le répertoire `site_scons/site_tools/toolname`. Dans le cas d'utilisation d' un paquet, le `exists()` et `generate()`sont dans le fichier `site_scons/site_tools/toolname/__init__.py`. (Dans tous les cas ci - dessus `toolname`est remplacé par le nom de l'outil.) Depuis `site_scons/site_tools`est automatiquement ajouté à la tête du chemin de recherche d'outil, un outil trouvé il sera disponible à tous les environnements. De plus, un outil trouvé il y passer outre un outil intégré du même nom, donc si vous avez besoin de changer le comportement d'un outil intégré, `site_scons`vous donne le crochet dont vous avez besoin.

Beaucoup de gens ont une bibliothèque de fonctions Python utilitaire qu'ils aimeraient inclure dans `SConscript`s; il suffit de mettre ce module`site_scons/my_utils.py`ou tout autre nom de module Python valide de votre choix. Par exemple , vous pouvez faire quelque chose comme ça dans `site_scons/my_utils.py`ajouter `build_id`et `MakeWorkDir` fonctions:

```
d'importation SCons.Script * # pour exécuter et Mkdir
def build_id ():
   « » « Retourne un ID de construction (version bouchonnée) » « »
   retour « 100 »
def MakeWorkDir (workdir):
   « » « Créer immédiatement le répertoire spécifié » « »
   Execute (Mkdir (workdir))
```

Et puis dans votre `SConscript`ou tout sous - `SConscript`partout dans votre construction, vous pouvez importer `my_utils`et utiliser:

```
my_utils d'importation
print "build_id =" + my_utils.build_id ()
my_utils.MakeWorkDir ( '/ tmp / travail')
```

Notez que bien que vous pouvez mettre cette bibliothèque`site_scons/site_init.py`, il n'y a pas mieux là que `site_scons/my_utils.py` puisque vous avez encore à importer ce module dans votre `SConscript`. A noter également que pour faire référence à des objets dans l'espace de noms SCons tels que `Environment`ou `Mkdir`ou `Execute`dans un fichier autre qu'un `SConstruct`ou `SConscript`vous devez toujours faire

```
* de l'importation SCons.Script
```

Cela est vrai dans les modules dans `site_scons`tels que`site_scons/site_init.py`aussi bien.

Vous pouvez utiliser l' un des sites ou par l' utilisateur à l' échelle tels que la machine dirs au `~/.scons/site_scons`lieu de `./site_scons`, ou utiliser l' `--site-dir`option pour pointer vers votre propre dir.`site_init.py`et `site_tools`sera situé dans ce répertoire. Pour éviter d' utiliser un `site_scons`répertoire du tout, même si elle existe, utilisez l' `--no-site-dir` option.

Chapitre 19. Non Rédaction d' un bâtisseur: `Command`Builder

Création d' un `Builder`et l' attacher à un `construction environment`permet beaucoup de flexibilité lorsque vous souhaitez réutiliser des actions pour créer plusieurs fichiers du même type. Cela peut cependant être lourd si vous avez besoin d'exécuter une commande spécifique pour construire un seul fichier (ou un groupe de fichiers). Pour ces situations, SCons soutient un `Command Builder`qui prend des dispositions pour une action spécifique à exécuter pour construire un ou plusieurs fichiers spécifiques. Cela ressemble beaucoup comme les autres constructeurs (comme `Program`, `Object`, etc.), mais prend comme argument supplémentaire de la commande à exécuter pour construire le fichier:

```
env = environnement ()
env.Command ( 'foo.out', 'foo.in', "sed 's / x / y /' <$ SOURCE> $ TARGET")
```

Lorsqu'il est exécuté, SCons exécute la commande spécifiée, en remplaçant `$SOURCE`et `$TARGET` comme prévu:

```
% scons -Q
sed 's / x / y /' <foo.in> foo.out
```

Cela est souvent plus pratique que la création d' un `Builder`objet et l' ajouter à la `$BUILDERS`variable d'un`construction environment`

Notez que l'action que vous spécifiez à `Command Builder`peut être tout droit SCons `Action` , comme une fonction Python:

```
env = environnement ()
def build (cible, la source, env):
    # Quoi qu'il en faut pour construire
    Aucun retour
env.Command ( 'foo.out', 'foo.in', build)
```

Ce qui est exécuté comme suit:

```
% scons -Q
construire ([ "foo.out"], [ "foo.in"])
```

Notez que $SOURCEet $TARGETsont étendus dans la source et la cible, ainsi que de SCons 1.1, vous pouvez écrire:

```
env.Command ( '$ {} SOURCE.basename .out', 'foo.in', construire)
```

qui fait la même chose que l'exemple précédent, mais vous permet d'éviter de vous répéter.

Chapitre 20. Les pseudo-constructeurs: la fonction AddMethod

La AddMethodfonction permet d'ajouter une méthode à un environnement. Il est généralement utilisé pour ajouter un « pseudo-constructeur » , une fonction qui ressemble à un , Buildermais se termine des appels à plusieurs autres Buildersou processus autrement ses arguments avant d' appeler un ou plusieurs Builders. Dans l'exemple suivant, nous voulons installer le programme dans la norme/usr/binhiérarchie des répertoires, mais aussi le copier dans local install/bin répertoire à partir duquel un paquet peut être construit:

```
def install_in_bin_dirs (env source):
    « » « Installer la source dans les deux bin dirs » « »
    i1 = env.Install ( "$ BIN", source)
    i2 = env.Install ( "$ LOCALBIN", source)
    retour [i1 [0], i2 [0]] # Retourne une liste, comme un constructeur normale
env = environnement (NIE = '/ usr / bin', LOCALBIN = '# install / bin')
env.AddMethod (install_in_bin_dirs, "InstallInBinDirs")
env.InstallInBinDirs (Programme ( 'hello.c')) # installe bonjour dans les deux bin dirs
```

Cela produit les éléments suivants:

```
% scons -Q /
cc -o -c hello.o hello.c
cc -o bonjour hello.o
Fichier d'installation: "bonjour" comme "/ usr / bin / bonjour"
Fichier d'installation: "bonjour" comme "install / bin / bonjour"
```

Comme mentionné précédemment, un pseudo-constructeur offre également une plus grande souplesse dans les arguments d'analyse que vous pouvez obtenir avec Builder. L'exemple suivant montre un pseudo-constructeur avec un argument nommé qui modifie le nom du fichier, et un argument distinct pour le fichier de ressources (plutôt que d' avoir la figure de constructeur dehors par extension de fichier). Cet exemple démontre également l' aide de la globale AddMethodfonction d'ajouter une méthode à la classe mondiale de l' environnement, il sera utilisé dans tous les environnements créés par la suite.

```
def BuildTestProg (env, toto, RESOURCEFILE, TestDir = "tests"):
    « » "Construire le programme de test;
    précèder « TEST_ » à src et cible,
    et met cible en testdir. « » »
    srcfile = "test_% sc" % testfile
    target = "% s / test_% s" % (testdir, testfile)
    si env [ 'PLATEFORME'] == 'win32':
        resfile = env.RES (RESOURCEFILE)
        p = env.Program (cible, [srcfile, resfile])
    autre:
        p = env.Program (cible, srcfile)
    retour p
AddMethod (Environnement, BuildTestProg)

env = environnement ()
env.BuildTestProg ( 'stuff', '= RESOURCEFILE res.rc')
```

Cela produit les éléments suivants sur Linux:

```
% scons -Q
cc -o -c test_stuff.o test_stuff.c
cc tests -o / test_stuff test_stuff.o
```

Et ce qui suit sous Windows:

```
C: \>scons -Q
rc /fores.res res.rc
cl /Fotest_stuff.obj / c test_stuff.c / nologo
lien / nologo /OUT:tests\test_stuff.exe test_stuff.obj res.res
embedManifestExeCheck (cible, source, env)
```

L' utilisation AddMethodest mieux que le simple ajout d' une méthode d'instance à une construction environmentcar elle est appelée comme méthode appropriée, et parce que AddMethodprévoit la copie de la méthode à tous les clones de l' construction environmentinstance.

Chapitre 21. Scanners d'écriture

SCons a des scanners intégrés qui savent regarder dans C, les fichiers sources Fortran et IDL pour obtenir des informations sur d' autres fichiers cibles construites à partir de ces fichiers dépendent - par exemple, dans le cas des fichiers qui utilisent le préprocesseur C, les .hfichiers qui sont spécifiés en utilisant des #includelignes dans la source. Vous pouvez utiliser les mêmes mécanismes que SCons utilise pour créer ses scanners intégrés pour écrire des scanners de votre propre pour les types de fichiers qui SCons ne sait pas comment numériser « de la boîte. »

21.1. Un scanner Exemple simple

Supposons, par exemple, que nous voulons créer un scanner simple pour les .foofichiers. Un .foofichier contient un texte qui sera traité, et peut inclure d' autres fichiers sur les lignes qui commencent par include suivi d'un nom de fichier:

```
comprennent filename.foo
```

Numérisation d' un fichier sera géré par une fonction Python que vous devez fournir. Voici une fonction qui utilisera le Python remodule pour analyser les includelignes dans notre exemple:

```
import re

include_re = re.compile (r '^ include \ s + (\ S +) $', RE.M)

def kfile_scan (noeud, env, chemin, arg):
    contenu = node.get_text_contents ()
    retourner env.File (include_re.findall (contenu))
```

Il est important de noter que vous devez retourner une liste de nœuds de fichiers de la fonction du scanner, des chaînes simples pour les noms de fichiers ne fera pas. Comme dans les exemples que nous montrons ici, vous pouvez utiliser la `File` fonction de votre environnement actuel afin de créer des nœuds à la volée à partir d' une séquence de noms de fichiers avec des chemins relatifs.

La fonction du scanner doit accepter les quatre arguments spécifiés et renvoie une liste de dépendances implicites. On peut supposer que ceux-ci seraient dépendances trouvées d'examiner le contenu du fichier, bien que la fonction peut effectuer toute manipulation du tout pour générer la liste des dépendances.

nœud

> Un SCons objet de noeud représentant le fichier en cours de balayage. Le nom de chemin d'accès au fichier peut être utilisé en convertissant le noeud à une chaîne en utilisant la `str()`fonction, ou une interne SCons `get_text_contents()` procédé objet peut être utilisé pour récupérer le contenu.

env

> L'environnement de construction en vigueur pour cette analyse. La fonction scanner peut choisir d'utiliser des variables de construction de cet environnement d'influer sur son comportement.

chemin

> Une liste des répertoires qui forment le chemin de recherche de fichiers inclus pour ce scanner. Voici comment SCons gère les `$CPPPATH`et les `$LIBPATH` variables.

arg

> Un argument optionnel que vous pouvez choisir d'avoir passé à cette fonction du scanner par différentes instances du scanner.

Un objet de scanner est créé en utilisant la `Scanner`fonction, qui prend typiquement un `skeys`argument pour associer le type de suffixe avec le scanneur. L'objet du scanner doit alors être associée à la`$SCANNERS`variable de construction d'un environnement de construction, typiquement à l'aide de la `Append`méthode:

```
kscan = Scanner (fonction = kfile_scan,
                 skeys = [ '.k'])
env.Append (SCANNEURS = kscan)
```

Lorsque nous mettons tous ensemble, il ressemble à:

```
import re

include_re = re.compile (r '^ include \ s + (\ S +) $', RE.M)

def kfile_scan (noeud, env, path):
    contenu = node.get_text_contents ()
    comprend include_re.findall = (contenu)
    retourner env.File (comprend)

kscan = Scanner (fonction = kfile_scan,
                 skeys = [ '.k'])

env = environnement (ENV = { 'PATH': '/ usr / local / bin'})
env.Append (SCANNEURS = kscan)

env.Command ( 'foo', 'foo.k', 'kprocess <$ SOURCES> $ TARGET')
```

21.2. Ajout d'un chemin de recherche à un scanner:`FindPathDirs`

De nombreux scanners doivent rechercher des fichiers inclus ou dépendances en utilisant une variable de chemin; c'est ainsi `$CPPPATH`et`$LIBPATH`travail. Le chemin de recherche est passé à votre scanner comme `path`argument. Les variables de chemin peuvent être des listes de noeuds, chaînes séparées par des points-virgules, ou même contenir des variables de SCons qui doivent être élargis. Heureusement, SCons fournit la `FindPathDirs`fonction qui renvoie elle - même une fonction pour développer un chemin donné (donné comme nom de variable de construction SCons) à une liste de chemins au moment où le scanner est appelé. Évaluation jusqu'à ce point report permet, par exemple, le chemin d'accès contient $ CIBLES références qui diffèrent pour chaque fichier numérisé.

L' utilisation `FindPathDirs`est assez facile. En reprenant l'exemple ci - dessus, en utilisant KPATH comme variable de construction du chemin de recherche (analogue à `$CPPPATH`), nous modifions simplement l' `Scanner` appel du constructeur d'inclure un arg mot - clé chemin:

```
kscan = Scanner (fonction = kfile_scan,
                 skeys = [ '.k'],
                 path_function = FindPathDirs ( 'KPATH'))
```

FindPathDirs retourne un objet appelable que, lorsqu'il est appelé, va essentiellement développer les éléments dans env [« KPATH »] et de dire au scanner pour rechercher dans ces dirs. Il ajoutera aussi bien référentiel connexes et variantes dirs à la liste de recherche. Comme une note de côté, la méthode de retour stocke le chemin d'une manière efficace si rapide sont même lookups lorsque des substitutions variables peuvent être nécessaires. Ceci est important car de nombreux fichiers sont numérisés dans une version typique.

Chapitre 22. À partir du code Référentiels

Souvent, un projet de logiciel aura un ou plusieurs référentiels centraux, arborescences de répertoires qui contiennent du code source ou des fichiers dérivés, ou les deux. Vous pouvez éliminer les reconstructions inutiles supplémentaires de fichiers en ayant SCons utilisent des fichiers à partir d' un ou plusieurs référentiels de code pour créer des fichiers dans l' arborescence de construction locale.

22.1. la `Repository`Méthode

Il est souvent utile pour permettre aux programmeurs multiples travaillant sur un projet de construction d'un logiciel à partir des fichiers source et / ou les fichiers dérivés qui sont stockés dans un référentiel centralisée accessible, une copie de l'arborescence du code source. (Notez que ce n'est pas le genre de dépôt maintenu par un système de gestion de code source comme BitKeeper, CVS ou Subversion.) Vous utilisez la `Repository`méthode pour dire SCons pour rechercher un ou plusieurs dépôts centraux de code (dans l' ordre) pour les fichiers source et les fichiers dérivés qui ne sont pas présents dans l'arborescence de construction locale:

```
env = environnement ()
env.Program ( 'hello.c')
Repository ( '/ usr / repository1', '/ usr / repository2')
```

Plusieurs appels à la `Repository`méthode simplement ajouter des dépôts à la liste globale qui SCons maintient, à l'exception que SCons éliminera automatiquement le répertoire courant et tous les répertoires inexistants dans la liste.

22.2. Trouver les fichiers source dans des dépôts

L'exemple ci - dessus indique que SCons d' abord rechercher des fichiers sous l' `/usr/repository1`arbre et ensuite sous l' `/usr/repository2`arbre. SCons attend à ce que tous les fichiers il recherche se trouvent dans la même position par rapport au répertoire de niveau supérieur. Dans l'exemple ci - dessus, si le `hello.c`fichier ne se trouve pas dans l'arborescence de construction locale, SCons cherchera d' abord un `/usr/repository1/hello.c`fichier, puis pour un `/usr/repository2/hello.c`fichier à utiliser à sa place.

Donc , étant donné le `SConstruct`fichier ci - dessus, si le `hello.c`fichier existe dans le répertoire de construction locale, SCons reconstruira le `hello`programme normal:

```
% scons -Q
cc -o -c hello.o hello.c
cc -o bonjour hello.o
```

Si, toutefois, il n'y a pas locale `hello.c`fichier, mais existe dans `/usr/repository1`, SCons recompile le `hello`programme à partir du fichier source , il trouve dans le référentiel:

```
% scons -Q
cc -o -c hello.o /usr/repository1/hello.c
cc -o bonjour hello.o
```

Et de même, s'il n'y a pas locale `hello.c`fichier et pas `/usr/repository1/hello.c`, mais il en existe dans `/usr/repository2`:

```
% scons -Q
cc -o -c hello.o /usr/repository2/hello.c
cc -o bonjour hello.o
```

22.3. Trouver des `#include`fichiers dans des dépôts

Nous avons déjà vu que SCons va analyser le contenu d'un fichier source pour les `#include`noms de fichiers et de réaliser que les objectifs construits à partir de ce fichier source dépendent également du `#include`fichier (s). Pour chaque répertoire dans la `$CPPPATH`liste, SCons rechercheront effectivement les répertoires correspondants dans tous les arbres du référentiel et d' établir les dépendances correctes sur tous les `#include`fichiers qu'il trouve dans le répertoire référentiel.

À moins que le compilateur C sait aussi sur ces répertoires dans les arbres du référentiel, cependant, il sera incapable de trouver les `#include`fichiers. Si, par exemple, le `hello.c`fichier dans notre exemple précédent inclut le `hello.h` dans son répertoire courant, et `hello.h`existe uniquement dans le référentiel:

```
% scons -Q
cc -o -c hello.o hello.c
hello.c: 1: hello.h: Aucun fichier ou répertoire
```

Afin d'informer le compilateur C sur les dépôts, SCons ajouteront appropriés `-I`drapeaux aux commandes de compilation pour chaque répertoire dans la `$CPPPATH`liste. Donc , si l' on ajoute le répertoire courant à l'environnement de la construction `$CPPPATH`comme ceci:

```
env = environnement (CPPPATH = [ ''])
env.Program ( 'hello.c')
Repository ( '/ usr / repository1')
```

Puis réexécutant SCons rendements:

```
% scons -Q
cc -o hello.o -c -I. -I / usr / repository1 hello.c
cc -o bonjour hello.o
```

L'ordre des `-I`options de réplication, pour le préprocesseur C, le même chemin de recherche référentiel-répertoire SCons utilise pour sa propre analyse de dépendance. S'il y a plusieurs référentiels et plusieurs `$CPPPATH` répertoires, SCons ajoutera les répertoires du référentiel au début de chaque `$CPPPATH`répertoire, multipliant rapidement le nombre de `-I`drapeaux. Si, par exemple, `$CPPPATH`contient trois répertoires (et les noms de référentiel chemin plus court!):

```
env = environnement (CPPPATH = [ 'rep1', 'dir2', 'dos3'])
env.Program ( 'hello.c')
Repository ( '/ r1', '/ R2')
```

Ensuite , nous allons finir avec neuf `-I`options sur la ligne de commande, trois (pour chacun des `$CPPPATH`répertoires) trois fois (pour le répertoire local ainsi que les deux référentiels):

```
% scons -Q
cc -o -c hello.o -Idir1 -I / r1 / rep1 -I / r2 / rep1 -Idir2 -I / r1 / dir2 -I / r2 / dir2 -Idir3 -I / r1 / dos3 -I / r2 / dos3 Bonjour c
cc -o bonjour hello.o
```

22.3.1. Limitations des `#include`fichiers dans les référentiels

SCons repose sur le compilateur de C `-I`options pour contrôler l'ordre dans lequel le préprocesseur recherche des répertoires du référentiel pour les `#include`fichiers. Cela pose un problème, cependant, avec la façon dont les poignées C préprocesseur `#include`lignes avec le nom de fichier inclus dans des guillemets doubles.

Comme nous l' avons vu, SCons va compiler le `hello.c`fichier à partir du référentiel si elle n'existe pas dans le répertoire local. Toutefois, si le `hello.c`fichier dans le référentiel contient une `#include`ligne avec le nom de fichier entre guillemets:

```
#include "hello.h"
int
main (int argc, char * argv [])
{
    printf (HELLO_MESSAGE);
    return (0);
}
```

Ensuite , le préprocesseur C sera *toujours* utiliser un `hello.h`fichier à partir du répertoire de dépôt d' abord, même s'il y a un `hello.h`fichier dans le répertoire local, en dépit du fait que la ligne de commande indique `-I`que la première option:

```
% scons -Q
cc -o hello.o -c -I. -I / usr / repository1 /usr/repository1/hello.c
cc -o bonjour hello.o
```

Ce comportement du préprocesseur C - recherche toujours un `#include`fichier entre guillemets d' abord dans le même répertoire que le fichier source, et ne recherche alors le `-I`--can pas, en général, être changé. En d' autres termes, il est une limitation qui doit vivre avec si vous voulez utiliser des référentiels de code de cette façon. Il y a trois façons dont vous pouvez éventuellement contourner ce problème préprocesseur C:

1. Certaines versions modernes des compilateurs C ont une option pour désactiver ou contrôler ce comportement. Si oui, ajoutez cette option `$CFLAGS` (ou `$CXXFLAGS`ou les deux) dans votre environnement de construction (s). Assurez - vous que l'option est utilisée pour tous les environnements de construction qui utilisent C pré - traitement!

2. Changer toutes les occurrences de `#include "file.h"` la `#include <file.h>`. Utilisation d' `#include`avec crochets n'a pas le même comportement - les `-I` répertoires sont d' abord pour les `#include`fichiers - qui donne SCons un contrôle direct sur la liste des répertoires préprocesseur C recherchera.

3. Exiger que tous ceux qui travaillent avec la compilation de dépôts vérifier et de travailler sur des répertoires entiers de fichiers, et non pas des fichiers individuels. (Si vous utilisez des scripts wrapper locaux autour de la commande de votre système de contrôle de code source, vous pouvez ajouter une logique d'appliquer cette restriction là.

22.4. Trouver le `SConstruct`fichier dans des dépôts

SCons cherchera également dans des dépôts pour le `SConstruct`fichier et tous spécifiés `SConscript`fichiers. Cela pose un problème, cependant: comment SCons chercher un arbre référentiel pour un `SConstruct`fichier si le `SConstruct`fichier lui - même contient les informations sur le chemin du dépôt? Pour résoudre ce problème, SCons vous permet de spécifier des répertoires du référentiel sur la ligne de commande en utilisant l' `-Y`option de :

```
% scons -Q -Y /usr/repository1 -Y /usr/repository2
```

Lorsque vous cherchez la source ou les fichiers dérivés, SCons recherche d' abord les référentiels spécifiés sur la ligne de commande, puis recherchez les référentiels spécifiés dans les `SConstruct`ou les `SConscript`fichiers.

22.5. Recherche de fichiers dérivés dans les dépôts

Si un dépôt contient non seulement des fichiers source, mais aussi les fichiers dérivés (tels que les fichiers d'objets, des bibliothèques ou des exécutables), SCons remplirai son calcul normal de la signature MD5 pour décider si un fichier dérivé dans un dépôt est à jour, ou le fichier dérivé doit être reconstruit dans le répertoire de construction locale. Pour le SCons calcul de la signature à fonctionner correctement, un arbre référentiel doit contenir les `.sconsign`fichiers qui SCons utilise pour garder une trace des informations de signature.

Habituellement, cela se fait par un intégrateur de construction qui courrait SCons dans le référentiel pour créer tous ses fichiers dérivés et des `.sconsign`fichiers, ou qui exécuter SCons dans un répertoire de construction séparé et copiez l'arbre résultant du dépôt souhaité:

```
% cd /usr/repository1
%scons -Q
cc -o -c file1.o file1.c
cc -o -c file2.o file2.c
cc -o -c hello.o hello.c
cc -o bonjour hello.o file1.o file2.o
```

(Notez que ceci est en sécurité , même si les `SConstruct`listes de fichiers en `/usr/repository1`tant que référentiel, car SCons va supprimer le répertoire en cours de construction liste de référentiels pour cette invocation.)

Maintenant, avec le dépôt peuplé, il suffit de créer un seul fichier source locale , nous sommes intéressés à travailler avec pour le moment, et utiliser l' `-Y`option de dire SCons chercher tous les autres fichiers dont il a besoin à partir du référentiel:

```
% cd $HOME/build
% edit hello.c
%scons -Q -Y /usr/repository1
cc -c -o hello.o hello.c
cc -o bonjour hello.o /usr/repository1/file1.o /usr/repository1/file2.o
```

Notez que SCons se rend compte qu'il n'a pas besoin de reconstruire des copies locales `file1.o`et des `file2.o`fichiers, mais utilise les fichiers déjà compilés à partir du référentiel.

22.6. Copies locales de garantir les fichiers

Si l'arborescence du référentiel contient les résultats complets d'une construction, et nous essayons de construire à partir du référentiel sans aucun fichier dans notre arbre local, quelque chose modérément surprenant se produit:

```
% mkdir $HOME/build2
% cd $HOME/build2
%scons -Q -Y /usr/all/repository hello
scons: `bonjour » est à jour.
```

Pourquoi SCons dire que le `hello`programme est à jour quand il n'y a pas le `hello`programme dans le répertoire de construction locale? Du fait que le référentiel (pas le répertoire local) contient la mise à jour `hello`programme et SCons correctement détermine qu'il n'y a rien à faire pour reconstruire cette copie à jour du fichier.

Il y a, cependant, beaucoup de fois où vous voulez vous assurer qu'une copie locale d'un fichier existe toujours. Un script d'emballage ou d' essai, par exemple, peut supposer que certains fichiers générés existent localement. Pour dire SCons de faire une copie d'un fichier de référentiel mis à jour dans le répertoire de construction locale, utilisez la `Local`fonction:

```
env = environnement ()
bonjour = env.Program ( 'hello.c')
Local (bonjour)
```

Si nous courons alors la même commande, SCons fera une copie locale du programme de la copie de dépôt, et vous dire qu'il est en train de faire ceci:

```
% scons -Y /usr/all/repository hello
Copie locale de bonjour de / usr / all / repository / bonjour
scons: `bonjour » est à jour.
```

(Notez que, parce que le fait de rendre la copie locale est pas considérée comme une « construction » du `hello`fichier, SCons signale encore qu'il est à jour.)

Chapitre 23. Configuration multi-plateforme (Autoconf fonctionnalité)

SCons a intégré le support pour la configuration de construction multi-plate - forme similaire à celle offerte par GNU Autoconf , comme déterminer quels bibliothèques ou les fichiers d' en- tête sont disponibles sur le système local. Cette section décrit comment utiliser cette SCons fonction.

> **Remarque**
>
> Ce chapitre est encore en développement, afin de ne pas tout est expliqué aussi bien qu'il devrait être. Voir la SCons page de manuel pour plus d' informations.

23.1. `Configure Contexts`

Le cadre de base pour la configuration de construction multi-plateforme SCons est d'attacher un `configure context`à un environnement de construction en appelant la `Configure`fonction, effectuer un certain nombre de contrôles pour les bibliothèques, les fonctions, les fichiers d' en- tête, etc., et d'appeler ensuite le contexte configure `Finish`la méthode pour terminer de la configuration:

```
env = environnement ()
conf = Configure (env)
# Les chèques pour les bibliothèques, les fichiers d'en-tête, etc. rendez-vous ici!
env = conf.Finish ()
```

SCons fournit un certain nombre de contrôles de base, ainsi qu'un mécanisme permettant d' ajouter vos propres contrôles personnalisés.

Notez que SCons utilise son propre mécanisme de dépendance pour déterminer quand un chèque doit être exécuté - à savoir, SCons ne fonctionne pas les contrôles à chaque fois qu'il est appelé, mais met en cache les valeurs renvoyées par les contrôles précédents et utilise les valeurs mises en cache à moins que quelque chose a modifié. Cela permet d' économiser une quantité énorme de temps de développement tout en travaillant sur les questions de construction multi-plateforme.

Les sections suivantes décrivent les contrôles de base qui SCons soutient, ainsi que la façon d'ajouter vos propres contrôles personnalisés.

23.2. Vérification de l'existence d'en-tête des fichiers

Test de l'existence d'un fichier d' en- tête , il faut savoir quelle langue le fichier d' en- tête est. Un contexte de configuration a une `CheckCHeader`méthode qui vérifie l'existence d'un fichier d' en- tête C:

```
env = environnement ()
conf = Configure (env)
sinon conf.CheckCHeader ( 'math.h'):
    print 'math.h doit être installé!
    Sortie (1)
si conf.CheckCHeader ( 'foo.h'):
    conf.env.Append ( '- DHAS_FOO_H')
env = conf.Finish ()
```

Notez que vous pouvez choisir de mettre fin à la construction si un fichier d'en-tête donné n'existe pas, ou vous pouvez modifier l'environnement de la construction fondée sur l'existence d'un fichier d'en-tête.

Si vous devez vérifier l'existence d' un fichier d' en- tête C ++, utilisez la `CheckCXXHeader`méthode:

```
env = environnement ()
conf = Configure (env)
sinon conf.CheckCXXHeader ( 'vector.h'):
    print 'vector.h doit être installé!
```

```
    Sortie (1)
env = conf.Finish ()
```

23.3. Vérification de la disponibilité d'une fonction

Vérifiez la disponibilité d'une fonction spécifique en utilisant la `CheckFunc`méthode:

```
env = environnement ()
conf = Configure (env)
sinon conf.CheckFunc ( 'strcpy'):
    print 'Vous ne trouvez pas strcpy (), en utilisant la version locale'
    conf.env.Append (CPPDEFINES = '-Dstrcpy = my_local_strcpy')
env = conf.Finish ()
```

23.4. Vérification de la disponibilité d'une bibliothèque

Vérifiez la disponibilité d'une bibliothèque à l' aide de la `CheckLib`méthode. Vous spécifiez uniquement le nom de base de la bibliothèque, vous n'avez pas besoin d'ajouter un `lib` préfixe ou un `.a`ou `.lib`suffixe:

```
env = environnement ()
conf = Configure (env)
sinon conf.CheckLib ( 'm'):
    print 'Vous ne trouvez pas libm.a ou m.lib, sortir!
    Sortie (1)
env = conf.Finish ()
```

Parce que la capacité d'utiliser une bibliothèque avec succès dépend souvent d'avoir accès à un fichier d' en- tête qui décrit l'interface de la bibliothèque, vous pouvez vérifier une bibliothèque *et* un fichier d' en- tête en même temps en utilisant la `CheckLibWithHeader`méthode:

```
env = environnement ()
conf = Configure (env)
sinon conf.CheckLibWithHeader ( 'm', 'math.h', 'c'):
    print 'Vous ne trouvez pas libm.a ou m.lib, sortir!
    Sortie (1)
env = conf.Finish ()
```

Ceci est essentiellement un raccourci pour les appels séparés aux `CheckHeader`et `CheckLib` fonctions.

23.5. Vérification de la disponibilité d'un`typedef`

Vérifiez la disponibilité d'un `typedef` en utilisant la `CheckType`méthode:

```
env = environnement ()
conf = Configure (env)
sinon conf.CheckType ( 'off_t'):
    print 'Vous ne trouvez pas off_t typedef, en supposant int'
    conf.env.Append (CCFLAGS = '-Doff_t = int')
env = conf.Finish ()
```

Vous pouvez également ajouter une chaîne qui sera placé au début du fichier de test qui sera utilisé pour vérifier la `typedef`. Cette offre un moyen de spécifier les fichiers qui doivent être inclus pour trouver le `typedef`:

```
env = environnement ()
conf = Configure (env)
sinon conf.CheckType ( 'off_t', '#include <sys / types.h> \ n'):
    print 'Vous ne trouvez pas off_t typedef, en supposant int'
    conf.env.Append (CCFLAGS = '-Doff_t = int')
env = conf.Finish ()
```

23.6. Vérification de la taille d'un type de données

Vérifiez la taille d'un type de données en utilisant la `CheckTypeSize`méthode:

```
env = environnement ()
conf = Configure (env)
int_size = conf.CheckTypeSize ( 'unsigned int')
imprimer 'sizeof unsigned int est', int_size
env = conf.Finish ()
```

```
% scons -Q
sizeof unsigned int est 4
scons: ` « . est à jour.
```

23.7. Vérification de la présence d'un programme

Vérifier la présence d'un programme en utilisant la `CheckProg`méthode:

```
env = environnement ()
conf = Configure (env)
sinon conf.CheckProg ( 'toto'):
  imprimer « Impossible de trouver le foobar du programme sur le système »
  Sortie (1)
env = conf.Finish ()
```

23,8. Ajout de vos propres contrôles personnalisés

Une vérification personnalisée est une fonction Python qui vérifie une certaine condition d'exister sur le système en cours d' exécution, généralement en utilisant des méthodes qui SCons fournitures pour prendre soin des détails de vérifier si une compilation réussit, un lien réussit, un programme est runnable, etc. une simple vérification personnalisée de l'existence d'une bibliothèque spécifique peut se présenter comme suit:

```
mylib_test_source_file = "" »
#include <malib.h>
int main (int argc, char ** argv)
{
    MyLibrary mylib (argc, argv);
    return 0;
}
« » »

def CheckMyLibrary (contexte):
    context.Message ( «Vérification de MyLibrary ...)
    result = context.TryLink (mylib_test_source_file, '.c')
    context.Result (suite)
    return result
```

Les `Message`et `Result`méthodes doivent généralement commencer et mettre fin à une vérification personnalisée pour laisser l'utilisateur de savoir ce qui se passe: les `Message`impressions d'appel le message spécifié (sans retour à la ligne de fuite) et les `Result`impressions d'appel`yes`si la vérification réussit et `no`si elle ne fonctionne pas. La `TryLink`méthode teste en fait pour savoir si le texte du programme spécifié reliera avec succès.

(Notez qu'une vérification personnalisée peut modifier son contrôle basé sur des arguments que vous choisissez de le transmettre, ou en utilisant ou en modifiant l'environnement de contexte de configuration dans l' `context.env`attribut.)

Cette fonction de contrôle personnalisé est ensuite attaché au `configure context` en passant un dictionnaire à l' `Configure`appel qui associe un nom du chèque à la fonction sous - jacente:

```
env = environnement ()
= Configurer conf (env, custom_tests = { 'CheckMyLibrary': CheckMyLibrary})
```

Vous voulez généralement faire le chèque et le nom de la fonction même, comme nous l'avons fait ici, pour éviter toute confusion possible.

On peut alors mettre ces pièces ensemble et en fait appeler le `CheckMyLibrary`chèque comme suit:

```
mylib_test_source_file = "" »
#include <malib.h>
int main (int argc, char ** argv)
{
    MyLibrary mylib (argc, argv);
    return 0;
}
« » »

def CheckMyLibrary (contexte):
    context.Message ( «Vérification de MyLibrary ...)
    result = context.TryLink (mylib_test_source_file, '.c')
    context.Result (suite)
    return result

env = environnement ()
= Configurer conf (env, custom_tests = { 'CheckMyLibrary': CheckMyLibrary})
sinon conf.CheckMyLibrary ():
    print 'MyLibrary est pas installé!
    Sortie (1)
env = conf.Finish ()

# Nous pourrions alors ajouter des appels réels comme programme () pour construire
# Quelque chose en utilisant l'environnement de construction « env ».
```

Si MyLibrary est pas installé sur le système, la sortie ressemblera:

```
% scons
scons: Lecture du fichier SConscript ...
Vérification des MyLibrary ... échoué
MyLibrary est pas installé!
```

Si MyLibrary est installé, la sortie ressemblera:

```
% scons
scons: Lecture du fichier SConscript ...
Vérification des MyLibrary ... échoué
scons: fait la lecture SConscript
scons: objectifs de construction ...
    .
    .
    .
```

23.9. Non Configuration lors du nettoyage des cibles

En utilisant une configuration multi-plateforme comme décrit dans les sections précédentes géreront les commandes de configuration , même lors de l' appel **`scons -c`** pour nettoyer les cibles:

```
% scons -Q -c
Vérification de MyLibrary ... oui
Suppression foo.o
Suppression de foo
```

Bien que l' exécution des contrôles de la plate - forme lors de la suppression des cibles ne fait pas mal quoi que ce soit, il est généralement inutile. Vous pouvez éviter cela en utilisant la`GetOption`méthode pour vérifier si l' `-c`option de (propre) a été appelée sur la ligne de commande:

```
env = environnement ()
sinon env.GetOption ( 'propre'):
    = Configurer conf (env, custom_tests = { 'CheckMyLibrary': CheckMyLibrary})
    sinon conf.CheckMyLibrary ():
        print 'MyLibrary est pas installé!
        Sortie (1)
    env = conf.Finish ()
```

```
% scons -Q -c
Suppression foo.o
Suppression de foo
```

Chapitre 24. Mise en cache des fichiers intégré

Sur des projets logiciels multi-développeurs, vous pouvez parfois accélérer builds beaucoup de tous les développeurs en leur permettant de partager les fichiers dérivés qu'ils construisent. SCons rend cela facile, ainsi que fiable.

24.1. Spécification du répertoire du cache partagé

Pour activer le partage de fichiers dérivés, utilisez la `CacheDir`fonction dans un `SConscript`fichier:

```
CacheDir ( '/ usr / local / build_cache')
```

Notez que le répertoire que vous spécifiez doit déjà exister et être lisible et modifiable par tous les développeurs qui partageront des fichiers dérivés. Il devrait également être dans un endroit centralisé que tous les builds seront en mesure d'accéder. Dans les environnements où les développeurs utilisent des systèmes séparés (comme les postes de travail individuels) pour constructions, ce répertoire serait généralement sur un système de fichiers partagé ou NFS monté.

Voici ce qui se passe: Quand une construction a `CacheDir`spécifié, chaque fois qu'un fichier est construit, il est stocké dans le répertoire de cache partagé avec sa signature MD5 construire. [5] Sur la suite builds, avant qu'une action est invoqué pour créer un fichier, SCons vérifiera le répertoire de cache partagé pour voir si un fichier avec la signature exacte même de construction existe déjà. Dans ce cas, le fichier dérivé ne sera pas construit localement, mais sera copié dans le répertoire de construction local à partir du répertoire de cache partagé, comme suit:

```
% scons -Q
cc -o -c hello.o hello.c
cc -o bonjour hello.o
% scons -Q -c
Suppression hello.o
Suppression bonjour
% scons -Q
Récupérée `hello.o » de cache
Récupérée `bonjour » de cache
```

Notez que la `CacheDir`fonction calcule toujours MD5 pour construire sigantures les noms de fichiers de cache partagé , même si vous configurez SCons d'utiliser des estampilles pour décider si les fichiers sont à jour. (Voir le chapitre 6, *Dépendances* chapitre pour des informations sur la `Decider`fonction.) Par conséquent, l' utilisation `CacheDir`peut réduire ou éliminer les améliorations de performances potentielles de l' utilisation des horodateurs pour les décisions mises à jour.

24.2. Garder sortie de la construction cohérente

Un inconvénient potentiel à l' aide d' un cache partagé est que la sortie imprimée par SCons peut être incompatible d'invocation à l' invocation, car un fichier donné peut être reconstruit une fois et récupéré à partir du cache partagé la prochaine fois. Cela peut rendre la sortie analyse de construction plus difficile, en particulier pour les scripts automatisés qui attendent une sortie cohérente à chaque fois.

Toutefois, si vous utilisez l' `--cache-show`option SCons imprimeront la ligne de commande qu'il *aurait* pu exécuter pour construire le fichier, même quand il récupère le fichier à partir du cache partagé. Cela rend la sortie de génération constante chaque fois que la construction est exécuté:

```
% scons -Q
cc -o -c hello.o hello.c
cc -o bonjour hello.o
% scons -Q -c
Suppression hello.o
Suppression bonjour
% scons -Q --cache-show
cc -o -c hello.o hello.c
cc -o bonjour hello.o
```

Le compromis, bien sûr, est que vous ne savez plus si oui ou non SConsa récupéré un fichier dérivé de cache ou a reconstruit localement.

24.3. Non Utilisation du cache partagé pour les fichiers spécifiques

Vous pouvez désactiver la mise en cache pour certains fichiers spécifiques dans votre configuration. Par exemple, si vous ne souhaitez que mettre des fichiers exécutables dans un cache central, mais pas les fichiers d'objets intermédiaires, vous pouvez utiliser la `NoCache` fonction pour spécifier que les fichiers objets ne doivent pas être mises en cache:

```
env = environnement ()
obj = env.Object ( 'hello.c')
env.Program ( 'hello.c')
CacheDir ( 'cache')
NoCache ( 'hello.o')
```

Ensuite , lorsque vous exécutez `scons`après le nettoyage des cibles construites, il recompiler le fichier objet localement (car il n'existe pas dans le répertoire de cache partagé), mais toujours se rendre compte que le répertoire de cache partagé contient un programme exécutable mise à jour qui

peut être récupérée au lieu de re-liaison:

```
% scons -Q
cc -o -c hello.o hello.c
cc -o bonjour hello.o
% scons -Q -c
Suppression hello.o
Suppression bonjour
% scons -Q
cc -o -c hello.o hello.c
Récupérée `bonjour » de cache
```

24.4. La désactivation du cache partagé

Récupération d'un fichier déjà construit à partir du cache partagé est généralement un gain de temps significatif sur la reconstruction du fichier, mais combien d'économies (ou même si elle fait gagner du temps tout) peut dépendre beaucoup sur votre système ou la configuration du réseau. Par exemple, la récupération de fichiers mis en cache à partir d'un serveur occupé sur un réseau occupé pourrait finir par être plus lent que la reconstruction des fichiers localement.

Dans ce cas, vous pouvez spécifier l' `--cache-disable` option de ligne de commande pour dire SCons de ne pas récupérer les fichiers déjà construits à partir du répertoire de cache partagé:

```
% scons -Q
cc -o -c hello.o hello.c
cc -o bonjour hello.o
% scons -Q -c
Suppression hello.o
Suppression bonjour
% scons -Q
Récupérée `hello.o » de cache
Récupérée `bonjour » de cache
% scons -Q -c
Suppression hello.o
Suppression bonjour
% scons -Q --cache-disable
cc -o -c hello.o hello.c
cc -o bonjour hello.o
```

24.5. Peuplant un cache partagé des fichiers déjà intégré

Parfois, vous pouvez avoir un ou plusieurs fichiers dérivés déjà construits dans l'arbre de construction locale que vous souhaitez mettre à la disposition d'autres personnes faisant builds. Par exemple, vous trouverez peut-être plus efficace d'effectuer l'intégration construit avec le cache désactivé (par la section précédente) et seulement remplir le répertoire de cache partagé avec les fichiers construits après la construction de l'intégration a terminé avec succès. De cette façon, le cache ne se rempli de fichiers dérivés qui font partie d'une construction complète, réussie pas avec les fichiers qui pourraient être plus tard oblitérées lors du débogage des problèmes d'intégration.

Dans ce cas, vous pouvez utiliser la la `--cache-force`possibilité de dire SCons de mettre tous les fichiers dérivés dans le cache, même si les fichiers existent déjà dans votre arbre local d'avoir été construit par une invocation précédente:

```
% scons -Q --cache-disable
cc -o -c hello.o hello.c
cc -o bonjour hello.o
% scons -Q -c
Suppression hello.o
Suppression bonjour
% scons -Q --cache-disable
cc -o -c hello.o hello.c
cc -o bonjour hello.o
% scons -Q --cache-force
scons: ` « . est à jour.
% scons -Q
scons: ` « . est à jour.
```

Remarquez comment la course ci - dessus exemple démontre que l' `--cache-disable` option évite de mettre le haut `hello.o` et les `hello`fichiers dans le cache, mais après avoir utilisé l' `--cache-force`option, les fichiers ont été mis dans le cache pour l'invocation suivante pour récupérer.

24.6. Cache Contention réduisant au minimum: l' `--random`option

Si vous autorisez plusieurs compositions simultanément mettre à jour le répertoire de cache partagé, deux builds qui se produisent en même temps peut parfois commencer « course » entre eux pour construire les mêmes fichiers dans le même ordre. Si, par exemple, vous liez plusieurs fichiers dans un programme exécutable:

```
Programme ( 'prog',
        [ 'F1.c', 'f2.c', 'f3.c', 'F4.c', 'f5.c'])
```

SCons normalement construire les fichiers objet d'entrée sur lequel le programme dépend dans leur ordre normal, triés:

```
% scons -Q
cc -o -c f1.o f1.c
cc -o -c f5.o f5.c
cc -o -c f3.o f3.c
cc -o -c f2.o f2.c
cc -o -c f4.o F4.c
cc -o prog f1.o f2.o f3.o f4.o f5.o
```

Mais si deux de ces builds lieu simultanément, ils peuvent se regarder dans le cache à peu près en même temps et les deux décident que `f1.o`doit être reconstruit et poussé dans le répertoire de cache partagé, puis les deux décident que `f2.o` doivent être remises en état (et poussé dans le cache partagé répertoire), puis les deux décident que `f3.o` doivent être recréés ... Cela ne posera pas de problèmes de construction réels - les deux builds réussira, générer des fichiers de sortie correcte et remplir le cache - mais il ne représente un gaspillage d' efforts.

Pour remédier à cette affirmation pour le cache, vous pouvez utiliser l' `--random`option de ligne de commande pour dire SCons de construire des dépendances dans un ordre aléatoire:

```
% scons -Q --random
cc -o -c f3.o f3.c
cc -o -c f1.o f1.c
cc -o -c f5.o f5.c
cc -o -c f2.o f2.c
cc -o -c f4.o F4.c
cc -o prog f1.o f2.o f3.o f4.o f5.o
```

Multiple construit en utilisant l' `--random`option de construire généralement leurs dépendances dans différents ordres aléatoires, ce qui minimise les chances de beaucoup de discorde pour les fichiers du même nom dans le répertoire de cache partagé. Simultanée multiples construit peut encore courir pour essayer de construire le même fichier cible à l' occasion, mais de longues séquences de contention inefficaces devraient être rares.

A noter, bien sûr, l' `--random`option permet la sortie que SCons impressions étaient incompatibles d'invocation à l' invocation, qui peut être un problème en essayant de comparer la production de séries différentes de construction.

Si vous voulez faire des dépendances sûres seront construites dans un ordre aléatoire sans avoir à spécifier la `--random`ligne de commande très, vous pouvez utiliser la `SetOption`fonction pour définir l' `random`option dans tout `SConscript`fichier:

```
SetOption ( 'aléatoire', 1)
Programme ( 'prog',
        [ 'F1.c', 'f2.c', 'f3.c', 'F4.c', 'f5.c'])
```

---

[5] En fait, la signature MD5 est utilisé comme nom du fichier dans le répertoire de cache partagé dans lequel les contenus sont stockés.

Chapitre 25. Les cibles d'Alias

Nous avons déjà vu comment vous pouvez utiliser la `Alias` fonction pour créer une cible nommée `install`:

```
env = environnement ()
bonjour = env.Program ( 'hello.c')
env.Install ( '/ usr / bin', bonjour)
env.Alias ( 'install', '/ usr / bin')
```

Vous pouvez ensuite utiliser cet alias sur la ligne de commande pour dire SCons plus naturellement que vous souhaitez installer les fichiers:

```
% scons -Q install
cc -o -c hello.o hello.c
cc -o bonjour hello.o
Fichier d'installation: "bonjour" comme "/ usr / bin / bonjour"
```

Comme d' autres `Builder`méthodes, cependant, la `Alias`méthode retourne un objet représentant l'alias en cours de construction. Vous pouvez ensuite utiliser cet objet comme entrée anothother `Builder`. Ceci est particulièrement utile si vous utilisez un tel objet en entrée à un autre appel à la `Alias` `Builder`, vous permettant de créer une hiérarchie d'alias imbriqués:

```
env = environnement ()
p = env.Program ( 'foo.c')
l = env.Library ( 'bar.c')
env.Install ( '/ usr / bin', p)
env.Install ( '/ usr / lib', l)
ib = env.Alias ( 'install-bin', '/ usr / bin')
il = env.Alias ( 'install-lib', '/ usr / lib')
env.Alias ( 'install', [ib, il])
```

Cet exemple définit séparément `install`, `install-bin`et les `install-lib`alias, vous permettant un contrôle plus précis sur ce qui est installé:

```
% scons -Q install-bin
cc -o -c foo.o foo.c
cc -o foo foo.o
Fichier d'installation: "foo" comme "/ usr / bin / foo"
% scons -Q install-lib
cc -o -c bar.o bar.c
ar rc libbar.a bar.o
ranlib libbar.a
Fichier d'installation: « libbar.a » comme « /usr/lib/libbar.a »
% scons -Q -c /
Suppression foo.o
Suppression de foo
Suppression / usr / bin / foo
Suppression bar.o
Suppression libbar.a
Suppression /usr/lib/libbar.a
% scons -Q install
cc -o -c foo.o foo.c
cc -o foo foo.o
Fichier d'installation: "foo" comme "/ usr / bin / foo"
cc -o -c bar.o bar.c
ar rc libbar.a bar.o
ranlib libbar.a
Fichier d'installation: « libbar.a » comme « /usr/lib/libbar.a »
```

Chapitre 26. builds Java

Jusqu'à présent, nous avons utilisé des exemples de construction C et de C ++ pour démontrer les caractéristiques de SCons . SCons soutient également la création de programmes Java, mais builds Java sont traitées de façon légèrement différente, ce qui reflète la façon dont le compilateur Java et des outils construisent des programmes différemment que les autres chaînes d'outils de langues.

26,1. Bâtiment de classe Java Fichiers: le `Java`constructeur

L'activité de base lors de la programmation en Java, bien sûr, est de prendre un ou plusieurs `.java`fichiers contenant du code source Java et d'appeler le compilateur Java pour les transformer en un ou plusieurs`.class`fichiers. En SCons , vous faites cela en donnant au `Java`constructeur un répertoire cible dans lequel placer les `.class`fichiers, et un répertoire source qui contient les `.java`fichiers:

```
Java ( 'classes', 'src')
```

Si le `src`répertoire contient trois `.java`fichiers source, puis en cours d' exécution SCons pourrait ressembler à ceci:

```
% scons -Q
javac cours -d -sourcepath src src / Example1.java src / Example2.java src / Example3.java
```

SCons cherchera en fait l' `src` arborescence de répertoires pour tous les `.java`fichiers. Le compilateur Java crée alors les fichiers de classe nécessaires dans le `classes`sous - répertoire, sur la base des noms de classe se trouvent dans les `.java`fichiers.

26.2. Comment SCons Poignées Java dépendances

En plus de rechercher le répertoire source des `.java`fichiers, SCons exécute réellement les `.java`fichiers via un analyseur Java stripped-down que les chiffres sur ce que les classes sont définies. En d' autres termes, SCons sait, sans avoir à vous dire, quels `.class`fichiers seront produits par le javac appel. Ainsi , notre exemple un revêtement de la section précédente:

```
Java ( 'classes', 'src')
```

Non seulement vous dire de manière fiable que les `.class`fichiers dans le `classes`sous - répertoire sont à jour:

```
% scons -Q
javac cours -d -sourcepath src src / Example1.java src / Example2.java src / Example3.java
% scons -Q classes
scons: classes `est à jour.
```

Mais il va également supprimer tous les générés `.class`fichiers, même pour les classes internes, sans avoir à les spécifier manuellement. Par exemple, si nos `Example1.java` et `Example3.java` fichiers à la fois définir des classes supplémentaires, et la classe définie dans `Example2.java` a une classe interne, la course **`scons -c`** va nettoyer tous les `.class`fichiers ainsi:

```
% scons -Q
javac cours -d -sourcepath src src / Example1.java src / Example2.java src / Example3.java
% scons -Q -c classes
cours Removed / Example1.class
cours Removed / AdditionalClass1.class
cours Removed / Example2 $ Inner2.class
cours Removed / Example2.class
cours Removed / Example3.class
cours Removed / AdditionalClass3.class
```

Pour assurer une manipulation correcte des `.class` dépendances dans tous les cas, vous devez indiquer SCons quelle version Java est utilisé. Cela est nécessaire parce que Java 1.5 a changé les `.class`noms de fichiers pour les classes internes anonymes imbriquées. Utilisez la `JAVAVERSION`variable de construction pour spécifier la version en cours d' utilisation. Avec Java 1.6, l'exemple d' un revêtement peut alors être défini comme ceci:

```
Java ( 'classes', 'src', JAVAVERSION = '1.6')
```

Voir `JAVAVERSION`la page de manuel pour plus d' informations.

26.3. Building Java Archive (`.jar`) Fichiers: le `Jar`constructeur

Après avoir construit les fichiers de classe, il est fréquent de les rassembler dans une archive Java (`.jar`fichier), que vous faites en appelant la `Jar`méthode Builder. Si vous voulez simplement récupérer tous les fichiers de classe dans un sous - répertoire, il vous suffit de spécifier que sous - répertoire comme la `Jar`source:

```
Java (target = 'classes', source = 'src')
Pot (target = 'test.jar', source = 'classes')
```

SCons passera ensuite ce répertoire à la jarre commande, qui rassemblera tous les sous - jacents des `.class`fichiers:

```
% scons -Q
javac cours -d -sourcepath src src / Example1.java src / Example2.java src / Example3.java
jar cf cours de test.jar
```

Si vous voulez garder tous les `.class`fichiers de plusieurs programmes dans un seul endroit, et seulement archiver certains d'entre eux dans chaque `.jar`fichier, vous pouvez passer le `Jar`constructeur une liste de fichiers comme sa source. Il est extrêmement simple de créer plusieurs`.jar`fichiers de cette façon, en utilisant les listes de fichiers de classe cibles créés par des appels au `Java`constructeur comme sources aux différents `Jar`appels:

```
prog1_class_files = Java (target = 'classes', source = 'prog1')
prog2_class_files = Java (target = 'classes', source = 'prog2')
Pot (target = 'prog1.jar', source = prog1_class_files)
Pot (target = 'prog2.jar', source = prog2_class_files)
```

Cela crée ensuite `prog1.jar` et à `prog2.jar` côté des sous - répertoires contenant leurs `.java`fichiers:

```
% scons -Q
javac cours -d -sourcepath prog1 prog1 / Example1.java prog1 / Example2.java
javac cours -d -sourcepath prog2 prog2 / Example3.java prog2 / Example4.java
jar cf prog1.jar les classes -C Example1.class des classes -C Example2.class
jar cf prog2.jar les classes -C Example3.class des classes -C Example4.class
```

26.4. Bâtiment C - tête et Stub Fichiers: le JavaHconstructeur

Vous pouvez générer des fichiers d' en- tête de C et de source pour la mise en œuvre des méthodes natives, à l'aide du JavaHconstructeur. Il y a plusieurs façons d'utiliser le JavaHconstructeur. Une invocation typique pourrait ressembler à :

```
les classes Java (= target = 'classes', source = 'src / pkg / Sub')
JavaH (target = 'native', source = classes)
```

La source est une liste de fichiers de classe générés par l'appel au Javaconstructeur, et la cible est le répertoire de sortie dans lequel nous voulons que les fichiers d' en- tête C placés. L'objectif est converti en le -d quand SCons court javah :

```
% scons -Q
Les cours de javac -sourcepath src / pkg / sous src / pkg / sous / Example1.java src / pkg / sous / Example2.java src / pkg / sous / Example3.java
classes -classpath natif de javah pkg.sub.Example1 pkg.sub.Example2 pkg.sub.Example3
```

Dans ce cas, l'appel à javah va générer les fichiers d' en- têtenative/pkg_sub_Example1.h, native/pkg_sub_Example2.h etnative/pkg_sub_Example3.h. Notez que SCons rappeler que les fichiers de classe ont été générés avec un répertoire cible classes, et il a précisé que le répertoire puis cible comme -classpathoption de l'appel à javah .

Bien qu'il soit plus pratique d'utiliser la liste des fichiers de classe renvoyés par le Javaconstructeur comme source d'un appel au JavaHconstructeur, vous *pouvez* spécifier la liste des fichiers de classe à la main, si vous préférez. Si vous le faites, vous devez définir la$JAVACLASSDIRvariable de construction lors de l' appel JavaH:

```
Java (target = 'classes', source = 'src / pkg / sous')
class_file_list = [ 'cours / pkg / sous / Example1.class',
                    'Cours / pkg / sous / Example2.class',
                    'Cours / pkg / sous / Example3.class']
JavaH (target = 'native', source = class_file_list, JAVACLASSDIR = 'classes')
```

La $JAVACLASSDIRvaleur se transforme ensuite en le -classpath quand SCons court javah :

```
% scons -Q
Les cours de javac -sourcepath src / pkg / sous src / pkg / sous / Example1.java src / pkg / sous / Example2.java src / pkg / sous / Example3.java
classes -classpath natif de javah pkg.sub.Example1 pkg.sub.Example2 pkg.sub.Example3
```

Enfin, si vous ne voulez pas un fichier d' en- tête séparé généré pour chaque fichier source, vous pouvez spécifier un nœud explicite de fichier comme cible du JavaHconstructeur:

```
les classes Java (= target = 'classes', source = 'src / pkg / Sub')
JavaH (target = fichier ( 'Native.h'), source = classes)
```

Parce que SCons suppose par défaut que l'objectif du JavaHconstructeur est un répertoire, vous devez utiliser la Filefonction pour vous assurer que SCons ne crée pas un répertoire nommé native.h. Lorsqu'un fichier est utilisé, bien que, SCons convertit correctement le nom du fichier dans la javah -o Option:

```
% scons -Q
Les cours de javac -sourcepath src / pkg / sous src / pkg / sous / Example1.java src / pkg / sous / Example2.java src / pkg / sous / Example3.java
javah cours -o Native.h -classpath pkg.sub.Example1 pkg.sub.Example2 pkg.sub.Example3
```

26.5. Bâtiment RMI Stub et Skeleton classe Fichiers: le RMICconstructeur

Vous pouvez générer Remote Method Invocation en utilisant les talons du RMICconstructeur. La source est une liste de répertoires, généralement renvoyés par un appel au Javaconstructeur, et la cible est un répertoire de sortie où les _Stub.class et _Skel.classseront placés les fichiers:

```
les classes Java (= target = 'classes', source = 'src / pkg / Sub')
CRIM (target = 'outdir', la source des classes =)
```

Comme il l'a fait avec le JavaHconstructeur, SCons se souvient du répertoire de classe et il passe comme l' -classpathoption de CRIM :

```
% scons -Q
Les cours de javac -sourcepath src / pkg / sous src / pkg / sous / Example1.java src / pkg / sous / Example2.java
rmic -d outdir cours -classpath pkg.sub.Example1 pkg.sub.Example2
```

Cet exemple générer les
fichiers outdir/pkg/sub/Example1_Skel.class,outdir/pkg/sub/Example1_Stub.class,outdir/pkg/sub/Example2_Skel.classetoutdir/pkg/sub/Example2_Stub.class.

Chapitre 27. Fonctionnalité Divers

SCons prend en charge un grand nombre de fonctionnalités supplémentaires qui ne correspond pas facilement dans les autres chapitres.

27,1. Vérification de la version Python: la EnsurePythonVersionfonction

Bien que le SCons code lui - même fonctionne sur toutes les versions de Python 2.x 2.7 ou version ultérieure, vous êtes parfaitement libre d'utiliser la syntaxe Python et des modules de versions ultérieures lors de l' écriture de vos SConscriptfichiers ou vos propres modules locaux. Si vous faites cela, il est généralement utile de configurer SCons pour sortir gracieusement avec un message d'erreur si elle est en cours d' exécution avec une version de Python qui ne fonctionnera pas avec votre code. Cela est particulièrement vrai si vous allez utiliser SCons pour compiler le code source que vous envisagez de distribuer publiquement, où vous ne pouvez pas être sûr de la version Python qu'un utilisateur distant anonyme peut utiliser pour essayer de construire votre logiciel.

SCons fournit une EnsurePythonVersionfonction pour cela. Vous passez simplement les versions majeures et mineures numéros de la version de Python dont vous avez besoin:

```
EnsurePythonVersion (2, 5)
```

Et puis SCons se termine avec le message d'erreur suivant lorsqu'un utilisateur il fonctionne avec une version antérieure non prise en charge de Python:

```
% scons -Q
Python 2.5 ou plus nécessaire, mais vous avez Python 2.3.6
```

27.2. Vérification de la version SCons: la `EnsureSConsVersion`fonction

Vous pouvez, bien sûr, écrivez vos `SConscript`fichiers à utiliser des fonctionnalités qui ont été ajoutées que dans les versions récentes de SCons . Lorsque vous diffusez publiquement un logiciel qui est construit en utilisant SCons , il est utile d'avoir SCons vérifier la version utilisée et la sortie grâce à un message d'erreur si la version de l'utilisateur de SCons ne fonctionnera pas avec vos `SConscript`fichiers.SCons fournit une `EnsureSConsVersion`fonction qui vérifie la version de SCons dans la même la `EnsurePythonVersion`fonction vérifie la version de Python, en passant dans les versions majeures et mineures numéros de la version de SCons vous avez besoin:

```
EnsureSConsVersion (1, 0)
```

Et puis SCons se termine avec le message d'erreur suivant lorsqu'un utilisateur il fonctionne avec une version antérieure de non prise en charge SCons :

```
% scons -Q
SCons 1.0 ou plus nécessaire, mais vous avez SCons 0.98.5
```

27,3. Explicitement Mettre fin SCons lors de la lecture de `SConscript`fichiers: la `Exit`fonction

SCons prend en charge une `Exit`fonction qui peut être utilisé pour mettre fin à SCons en lisant les `SConscript`fichiers, généralement parce que vous avez détecté une condition dans laquelle il n'a pas de sens de procéder:

```
si ARGUMENTS.get ( 'FUTURE'):
    print « L'option avenir n'est pas encore pris en charge! »
    Sortie (2)
env = environnement ()
env.Program ( 'hello.c')
```

```
% scons -Q FUTURE=1
L'option avenir n'est pas encore pris en charge!
% scons -Q
cc -o -c hello.o hello.c
cc -o bonjour hello.o
```

La `Exit`fonction prend comme argument l'état de sortie (numérique) que vous voulez SCons pour sortir avec. Si vous ne spécifiez pas de valeur, la valeur par défaut est de sortir avec `0`, ce qui indique une exécution réussie.

Notez que la `Exit`fonction équivaut à appeler la Python`sys.exit`fonction (qui l'appelle en fait), mais parce que `Exit`est une SCons fonction, vous ne devez pas importer le Python `sys`module pour l' utiliser.

27.4. Recherche de fichiers: la `FindFile`fonction

La `FindFile`fonction recherche d'un fichier dans une liste de répertoires. S'il n'y a qu'un seul répertoire, il peut être donné comme une chaîne simple. La fonction retourne un nœud de fichier si un fichier correspondant existe, ou None si aucun fichier n'a été trouvé. (Consultez la documentation de la `Glob`fonction pour une autre façon de rechercher des entrées dans un répertoire.)

```
# Un répertoire
imprimer FindFile ( 'disparus', '')
t = FindFile ( 'existe', '')
imprimer t .__ class__, t
```

```
% scons -Q
Aucun
<Class 'SCons.Node.FS.File'> existe
scons: ` « . est à jour.
```

```
# plusieurs répertoires
inclut = [ '', 'comprennent', 'src / include']
headers = [ 'nonesuch.h', 'config.h', 'private.h', 'dist.h']
pour hdr les en-têtes:
    impression de l'% '% -12s de ( '% s:' % RDH), FindFile (RDH comprend)
```

```
% scons -Q
nonesuch.h: Aucun
config.h: config.h
private.h: src / include / private.h
dist.h: include / dist.h
scons: ` « . est à jour.
```

Si le fichier existe dans plus d'un répertoire, seule la première occurrence est renvoyée.

```
imprimer FindFile ( 'multiple', [ 'sub1', 'SUB2', 'sub3'])
imprimer FindFile ( 'multiple', [ 'SUB2', 'sub3', 'sub1'])
imprimer FindFile ( 'multiple', [ 'sub3', 'sub1', 'SUB2'])
```

```
% scons -Q
SUB1 / multiple
SUB2 / multiple
SUB3 / multiple
scons: ` « . est à jour.
```

Outre les fichiers existants, `FindFile`trouveront également des fichiers dérivés (qui est, les fichiers non-feuilles) qui ne sont pas encore construits. (Fichiers feuilles devraient déjà exister, ou la construction échouera!)

```
# Aucun fichier existe, afin de bâtir échouera
Commande ( 'dérivé', 'feuille', 'cat> $ TARGET $ SOURCE)
imprimer FindFile ( 'feuille', '')
imprimer FindFile ( 'dérivé', '')
```

```
% scons -Q
feuille
dérivé
chat> feuille dérivé
```

```
# Seulement feuillettent existe
Commande ( 'dérivé', 'feuille', 'cat> $ TARGET $ SOURCE)
imprimer FindFile ( 'feuille', '')
imprimer FindFile ( 'dérivé', '')
```

```
% scons -Q
feuille
dérivé
chat> feuille dérivé
```

Si un fichier source existe, `FindFile`sera correctement renvoyer le nom dans le répertoire de construction.

```
# Seulement « src / feuille » existe
VariantDir ( 'build', 'src')
imprimer FindFile ( 'feuille', 'build')
```

```
% scons -Q
construire / feuille
scons: ` « . est à jour.
```

27.5. Manipulation listes: la `Flatten`fonction

SCons prend en charge une `Flatten`fonction qui prend une séquence de python d'entrée (liste ou tuple) et retourne une liste contenant aplatie seulement les éléments individuels de la séquence. Cela peut être pratique lorsque vous essayez d'examiner une liste composée des listes retournées par les appels à différents constructeurs. Par exemple, vous pouvez collecter des fichiers d'objets construits de différentes manières en un seul appel au `Program`constructeur simplement en les enfermant dans une liste, comme suit:

```
objets = [
    Object ( 'prog1.c'),
    Object ( 'prog2.c', CCFLAGS = '- DFOO'),
]
Programme (objets)
```

Parce que le constructeur appelle à SCons aplatir leurs listes d'entrée, cela fonctionne très bien pour construire le programme:

```
% scons -Q
cc -o -c prog1.o prog1.c
cc -o -c prog2.o -DFOO prog2.c
cc -o prog1 prog1.o prog2.o
```

Mais si vous déboguer votre construction et je voulais imprimer le chemin absolu de chaque fichier objet dans la `objects`liste, vous pouvez essayer l'approche simple suivante, en essayant d'imprimer chaque nœud `abspath` attribut:

```
objets = [
    Object ( 'prog1.c'),
    Object ( 'prog2.c', CCFLAGS = '- DFOO'),
]
Programme (objets)

pour object_file dans les objets:
    Imprimer object_file.abspath
```

Cela ne fonctionne pas comme prévu parce que chaque appel `str`fonctionne une liste intégrée renvoyée par chaque `Object`appel, non pas sur les nœuds sous - jacents dans ces listes:

```
% scons -Q
AttributeError: objet 'NodeList' n'a pas d'attribut 'abspath':
  Fichier "/ home / mon / projet / SConstruct", ligne 8:
    Imprimer object_file.abspath
```

La solution est d'utiliser la `Flatten`fonction de sorte que vous pouvez passer chaque nœud à la `str`séparément:

```
objets = [
    Object ( 'prog1.c'),
    Object ( 'prog2.c', CCFLAGS = '- DFOO'),
]
Programme (objets)

pour object_file en Flatten (objets):
    Imprimer object_file.abspath
```

```
% scons -Q
/home/me/project/prog1.o
/home/me/project/prog2.o
cc -o -c prog1.o prog1.c
cc -o -c prog2.o -DFOO prog2.c
cc -o prog1 prog1.o prog2.o
```

27.6. Trouver le répertoire Invocation: la `GetLaunchDir`fonction

Si vous avez besoin de trouver le répertoire à partir duquel l'utilisateur a invoqué la `scons`commande, vous pouvez utiliser la `GetLaunchDir`fonction:

```
env = environnement (
    LAUNCHDIR = GetLaunchDir (),
)
env.Command ( 'directory_build_info',
            '$ LAUNCHDIR / build_info'
            Copie ( 'TARGET $', '$ SOURCE'))
```

Parce que SCons est généralement appelé à partir du répertoire de niveau supérieur dans lequel le `SConstruct`vit fichier, le Python `os.getcwd()` est souvent équivalent. Cependant, les SCons `-u` , `-U` et les `-D` options de ligne de commande, lorsqu'elle est appelée à partir d' un sous - répertoire, causeront SCons pour changer le répertoire dans lequel le `SConstruct`fichier est trouvé. Lorsque ces options sont utilisées, `GetLaunchDir`va encore retourner le chemin vers le sous - répertoire invocateur de l'utilisateur, ce qui permet la `SConscript`configuration d'obtenir encore en configuration (ou autre) des fichiers à partir du répertoire d' origine.

Chapitre 28. Dépannage

L'expérience de la configuration d' un outil de construction de logiciels pour construire une grande base de code en général, à un moment donné, consiste à essayer de comprendre pourquoi l'outil se comporte d' une certaine façon, et comment obtenir à se comporter comme vous le souhaitez. SCons est pas différent. Cette annexe contient un certain nombre de façons dont vous pouvez obtenir des informations supplémentaires sur SCons le comportement.

Notez que nous sommes toujours intéressés à essayer d'améliorer la façon dont vous pouvez résoudre les problèmes de configuration. Si vous rencontrez un problème qui vous a gratter la tête, et qui il ne semble pas être une bonne façon de débogage, les chances sont très bonnes que quelqu'un d' autre se déroulera dans le même problème, aussi. Si oui, s'il vous plaît laissez l'équipe de développement de SCons savoir (de préférence en déposant un rapport de bogue ou demande de fonctionnalité à nos pages de projet à tigris.org) afin que nous puissions utiliser vos commentaires pour essayer de trouver une meilleure façon de vous aider, et d' autres, obtenir les connaissances nécessaires sur SCons comportement pour aider à identifier et résoudre les problèmes de configuration.

28,1. Pourquoi est - ce être la cible Reconstruit? l' `--debug=explain`option

Regardons un exemple simple d'une construction misconfigured qui provoque une cible à chaque fois reconstruit SCons est exécuté:

```
# Orthographier Intentionnellement le nom du fichier de sortie dans la
# Commande utilisée pour créer le fichier:
Commande ( 'file.out', 'file.in', 'cp $ SOURCE file.oout')
```

(Note aux utilisateurs de Windows: La POSIX cp commande copie le premier fichier nommé sur la ligne de commande au second fichier Dans notre exemple, il copie le. `file.in`Fichier dans le `file.out`fichier.)

Maintenant , si nous courons SCons plusieurs fois sur cet exemple, on voit que des rediffusions de la cp commande à chaque fois:

```
% scons -Q
cp file.in file.oout
% scons -Q
cp file.in file.oout
% scons -Q
cp file.in file.oout
```

Dans cet exemple, la cause sous - jacente est évidente: nous avons volontairement mal orthographié le nom du fichier de sortie dans la cp commande, de sorte que la commande ne construit pas réellement le `file.out`fichier que nous avons dit SCons à attendre. Mais si le problème était pas évident, il serait utile de préciser l' `--debug=explain`option sur la ligne de commande pour avoir SCons nous dire très précisément pourquoi il a décidé de reconstruire la cible:

```
% scons -Q --debug=explain
scons: construction `file.out » parce qu'il n'existe pas
cp file.in file.oout
```

Si cela avait été un exemple plus complexe impliquant un grand nombre de sorties de la construction, ayant SCons nous disent qu'il essaie de reconstruire le fichier cible car il n'existe pas serait un indice important que quelque chose ne va pas avec la commande que nous invoquions pour construire il.

Notez que vous pouvez également utiliser --warn = cible non construit qui vérifie si oui ou non des objectifs prévus existent après une règle de construction est exécutée.

```
% scons -Q --warn=target-not-built
cp file.in file.oout

scons: avertissement: Vous ne trouvez pas cibler file.out après la construction
Fichier "/home/bdbaddog/devel/scons/bootstrap/src/script/scons.py", ligne 199, dans <module>
```

La `--debug=explain`possibilité est également utile pour aider à comprendre ce fichier d'entrée a changé. Étant donné une configuration simple qui construit un programme de trois fichiers source, en changeant l' un des fichiers source et la reconstruction avec l' `--debug=explain` option montre très précisément pourquoi SConsreconstruit les fichiers qu'il fait:

```
% scons -Q
cc -o -c file1.o file1.c
cc -o -c file2.o file2.c
cc -o -c file3.o file3.c
cc -o prog file1.o file2.o file3.o
% [CHANGEMENT DU CONTENU DE fichier2.c]
% scons -Q --debug=explain
scons: reconstruction `file2.o 'parce que` fichier2.c' a changé
cc -o -c file2.o file2.c
scons: reconstruction `prog 'parce que` file2.o' a changé
cc -o prog file1.o file2.o file3.o
```

Cela devient encore plus utile pour identifier quand un fichier est reconstruit en raison d'un changement dans une dépendance implicite, comme un incuded `.h`fichier. Si les `file1.c` et `file3.c`fichiers dans notre exemple tous deux inclus un `hello.h`fichier, puis en changeant que les fichiers et la réexécution SCons avec l' `--debug=explain`option préciser que c'est la modification du fichier inclus qui commence la chaîne de reconstructions:

```
% scons -Q
cc -o file1.o -c -I. fichier1.c
cc -o file2.o -c -I. fichier2.c
cc -o file3.o -c -I. file3.c
cc -o prog file1.o file2.o file3.o
% [CHANGEMENT DU CONTENU DE hello.h]
% scons -Q --debug=explain
scons: reconstruction `file1.o 'parce que` hello.h' a changé
cc -o file1.o -c -I. fichier1.c
scons: reconstruction `file3.o 'parce que` hello.h' a changé
cc -o file3.o -c -I. file3.c
scons: reconstruction `prog » parce que:
           `File1.o » changé
           `File3.o » changé
cc -o prog file1.o file2.o file3.o
```

(Notez que l' `--debug=explain`option ne vous dire pourquoi SCons a décidé de reconstruire les objectifs nécessaires. Il ne vous dit quels fichiers il a examiné non au moment de décider *pas* de reconstruire un fichier cible, ce qui est souvent une question plus précieux pour répondre.)

28.2. Ce qui est dans cet environnement de la construction? la `Dump`Méthode

Lorsque vous créez un environnement de construction, SCons il remplit avec des variables de construction qui sont mis en place pour différents compilateurs, linkers et les utilitaires qu'il trouve sur votre système. Bien que ce soit généralement utile et ce que vous voulez, il peut être frustrant si SCons ne définit pas certaines variables que vous attendez à régler. Dans des situations comme cela, il est parfois utile d'utiliser l'environnement de construction `Dump`méthode pour imprimer tout ou partie des variables de construction. Notez que la `Dump`méthode*retourne* la représentation des variables dans l'environnement pour vous d'imprimer (ou manipuler):

```
env = environnement ()
imprimer env.Dump ()
```

Sur un système POSIX avec gcc, cela pourrait générer:

```
% scons
scons: Lecture des fichiers SConscript ...
{ '' CONSTRUCTEURS: { '_InternalInstall': <fonction InstallBuilderWrapper à 0x700000 & gt ;, '_InternalInstallVersionedLib': <fonction InstallVersion
  'CONFIGUREDIR': '# / sconf_temp.',
  'CONFIGURELOG': '# / config.log,
  'CPPSUFFIXES': [ '.c',
                   '.C',
                   '.cxx',
                   « Cpp »,
                   '.c ++',
                   '.cc',
                   '.h',
                   '.H',
                   '.hxx',
                   '.hpp',
                   '.hh',
                   '.F',
                   '.fpp',
                   '.FPP',
                   '.M',
                   '.mm',
                   .S ',
                   '.SPP',
                   '.SPP',
                   '.sx'],
  'DSUFFIXES': [ '.d'],
  'Dir': <object SCons.Defaults.Variable_Method_Caller à & gt ;, 0x700000
  'REPS': <object SCons.Defaults.Variable_Method_Caller à & gt ;, 0x700000
  'ENV': { 'PATH': '/ usr / local / bin: / opt / bin: / bin: / usr / bin'},
  'ESCAPE': <échapper à la fonction & gt ;, 0x700000
  'Fichier': <object SCons.Defaults.Variable_Method_Caller à & gt ;, 0x700000
  'HOST_ARCH': Aucun,
  'HOST_OS': Aucun,
  'IDLSUFFIXES': [ '.idl', '.IDL'],
  'INSTALLER': <fonction copyFunc à & gt ;, 0x700000
  'INSTALLVERSIONEDLIB': <fonction copyFuncVersionedLib à & gt ;, 0x700000
  'LIBPREFIX': 'lib',
  'LIBPREFIXES': [ '$ LIBPREFIX'],
  'LIBSUFFIX': '.a',
  'LIBSUFFIXES': [ '$ LIBSUFFIX', '$ SHLIBSUFFIX'],
  'MAXLINELENGTH': 128072,
  'OBJPREFIX': '',
  'OBJSUFFIX': '.o',
  'PLATEFORME': 'posix',
  'PROGPREFIX': '',
  'PROGSUFFIX': '',
  'PSPAWN': <fonction piped_env_spawn à & gt ;, 0x700000
  '' RDirs: <object SCons.Defaults.Variable_Method_Caller à & gt ;, 0x700000
  'SCANNERS': [],
  'SHELL': 'sh',
  'SHLIBPREFIX': 'LIBPREFIX $',
  'SHLIBSUFFIX': '.donc',
  'SHOBJPREFIX': 'OBJPREFIX $',
  'SHOBJSUFFIX': 'OBJSUFFIX $',
  'SPAWN': <fonction subprocess_spawn à & gt ;, 0x700000
  'TARGET_ARCH': Aucun,
  'TARGET_OS': Aucun,
  'TEMPFILE': <class 'SCons.Platform.TempFileMunge'>,
  'TEMPFILEPREFIX': '@',
  'Outils': [ 'install', 'install'],
  '_CPPDEFFLAGS': '$ {_ définit (CPPDEFPREFIX, CPPDEFINES, CPPDEFSUFFIX, __env__)}',
  '_CPPINCFLAGS': '$ ($ {_ concat (INCPREFIX, CPPPATH, INCSUFFIX, __env__, RDirs, TARGET, SOURCE)} $)',
```

```
   '_LIBDIRFLAGS': '$ ($ {_ concat (LIBDIRPREFIX, LIBPATH, LIBDIRSUFFIX, __env__, RDirs, TARGET, SOURCE)} $)',
   '_LIBFLAGS': '$ {_ concat (LIBLINKPREFIX, LIBS, LIBLINKSUFFIX, __env__)}',
   '__DRPATH': '_DRPATH $',
   '__RPATH': '_RPATH $',
   '_concat': <fonction _concat à & gt ;, 0x700000
   '_defines': <_defines fonction à & gt ;, 0x700000
   '_stripixes': <fonction _stripixes à 0x700000 & gt;}
scons: fait la lecture des fichiers SConscript.
scons: objectifs de construction ...
scons: ` « . est à jour.
scons: fait des objectifs de construction.
```

Sur un système Windows avec Visual C ++ la sortie pourrait ressembler à:

```
C: \>scons
scons: Lecture des fichiers SConscript ...
{ 'CONSTRUCTEURS': { '_InternalInstallVersionedLib': <fonction InstallVersionedBuilderWrapper à 0x700000 & gt ;, '_InternalInstall': <fonction Instal
  'CC': 'cl',
  'CCCOM': <object SCons.Action.FunctionAction à & gt ;, 0x700000
  'CCFLAGS': [ '/ nologo'],
  'CCPCHFLAGS': [ '$ {(PCH et "/ Yu% s \\" / Fp% s \\ "" % (PCHSTOP ou "", Fichier (PCH))) ou ""}'],
  'CCPDBFLAGS': [ '$ {(APB et "/ Z7") ou ""}'],
  'CFILESUFFIX': '.c',
  'CFLAGS': [],
  'CONFIGUREDIR': '# / sconf_temp.',
  'CONFIGURELOG': '# / config.log,
  'CPPDEFPREFIX': '/ D',
  'CPPDEFSUFFIX': '',
  'CPPSUFFIXES': [ '.c',
                   '.C',
                   '.cxx',
                   « Cpp »,
                   '.c ++',
                   '.cc',
                   '.h',
                   '.H',
                   '.hxx',
                   '.hpp',
                   '.hh',
                   '.F',
                   '.fpp',
                   '.FPP',
                   '.M',
                   '.mm',
                   .S ',
                   '.SPP',
                   '.SPP',
                   '.sx'],
  'CXX': '$ CC',
  'CXXCOM': '$ {TEMPFILE ( "$ CXX $ _MSVC_OUTPUT_FLAG / c $ CHANGED_SOURCES $ CXXFLAGS $ CCFLAGS $ _CCCOMCOM", "CXXCOMSTR $")}',
  'CXXFILESUFFIX': '.cc',
  'CXXFLAGS': [ '$ (', '/ TP', '$)'],
  'DSUFFIXES': [ '.d'],
  'Dir': <object SCons.Defaults.Variable_Method_Caller à & gt ;, 0x700000
  'REPS': <object SCons.Defaults.Variable_Method_Caller à & gt ;, 0x700000
  'ENV': { 'PATH': 'C: \\ WINDOWS \\ System32',
           'PATHEXT': '.COM, .EXE, .BAT, .CMD',
           'SystemRoot': 'C: \\ WINDOWS},
  'ESCAPE': <échapper à la fonction & gt ;, 0x700000
  'Fichier': <object SCons.Defaults.Variable_Method_Caller à & gt ;, 0x700000
  'HOST_ARCH': '',
  'HOST_OS': 'win32',
  'IDLSUFFIXES': [ '.idl', '.IDL'],
  'INCPREFIX': '/ I',
  'INCSUFFIX': '',
  'INSTALLER': <fonction copyFunc à & gt ;, 0x700000
  'INSTALLVERSIONEDLIB': <fonction copyFuncVersionedLib à & gt ;, 0x700000
  'LIBPREFIX': '',
  'LIBPREFIXES': [ '$ LIBPREFIX'],
  'LIBSUFFIX': '.lib',
  'LIBSUFFIXES': [ '$ LIBSUFFIX'],
  'MAXLINELENGTH': 2048,
  'MSVC_SETUP_RUN': Certes,
  'OBJPREFIX': '',
  'OBJSUFFIX': '.obj',
  'PCHCOM': '$ CXX / Fo $ {CIBLES [1]} $ CXXFLAGS $ CCFLAGS $ CPPFLAGS $ _CPPDEFFLAGS $ _CPPINCFLAGS / c $ SOURCES / Yc $ PCHSTOP / Fp $ {CIBLES [0]}
  'PCHPDBFLAGS' [ '$ {(PDB et "/ yd") ou ""}'],
  'PLATEFORME': 'win32',
  'PROGPREFIX': '',
  'PROGSUFFIX': '.exe',
  'PSPAWN': <fonction piped_spawn à & gt ;, 0x700000
  'RC': 'rc',
  'RCCOM': <object SCons.Action.FunctionAction à & gt ;, 0x700000
  'RCFLAGS': [],
  'RCSUFFIXES': [ '.rc', » .rc2' ],
  '' RDirs: <object SCons.Defaults.Variable_Method_Caller à & gt ;, 0x700000
  'SCANNERS': [],
  'SHCC': '$ CC',
  'SHCCCOM': <object SCons.Action.FunctionAction à & gt ;, 0x700000
  'SHCCFLAGS': [ '$ CCFLAGS'],
  'SHCFLAGS': [ '$ CFLAGS'],
  'SHCXX': 'CXX $',
  'SHCXXCOM': '$ {TEMPFILE ( "$ SHCXX $ _MSVC_OUTPUT_FLAG / c $ CHANGED_SOURCES $ SHCXXFLAGS $ SHCCFLAGS $ _CCCOMCOM", "SHCXXCOMSTR $")}',
  'SHCXXFLAGS': [ '$ CXXFLAGS'],
  'SHELL': Aucun,
  'SHLIBPREFIX': '',
  'SHLIBSUFFIX': '.dll',
  'SHOBJPREFIX': 'OBJPREFIX $',
  'SHOBJSUFFIX': 'OBJSUFFIX $',
  'SPAWN': <fraient fonction à & gt ;, 0x700000
  'STATIC_AND_SHARED_OBJECTS_ARE_THE_SAME': 1,
  'TARGET_ARCH': Aucun,
  'TARGET_OS': Aucun,
  'TEMPFILE': <class 'SCons.Platform.TempFileMunge'>,
```

```
   'TEMPFILEPREFIX': '@',
   'Outils': [ 'msvc', 'install', 'install'],
   '_CCCOMCOM': '$ CPPFLAGS $ _CPPDEFFLAGS $ _CPPINCFLAGS $ CCPCHFLAGS $ CCPDBFLAGS',
   '_CPPDEFFLAGS': '$ {_ définit (CPPDEFPREFIX, CPPDEFINES, CPPDEFSUFFIX, __env__)}',
   '_CPPINCFLAGS': '$ ($ {_ concat (INCPREFIX, CPPPATH, INCSUFFIX, __env__, RDirs, TARGET, SOURCE)} $)',
   '_LIBDIRFLAGS': '$ ($ {_ concat (LIBDIRPREFIX, LIBPATH, LIBDIRSUFFIX, __env__, RDirs, TARGET, SOURCE)} $)',
   '_LIBFLAGS': '$ {_ concat (LIBLINKPREFIX, LIBS, LIBLINKSUFFIX, __env__)}',
   '_MSVC_OUTPUT_FLAG': <fonction msvc_output_flag à & gt ;, 0x700000
   '_concat': <fonction _concat à & gt ;, 0x700000
   '_defines': <_defines fonction à & gt ;, 0x700000
   '_stripixes': <fonction _stripixes à 0x700000 & gt;}
scons: fait la lecture des fichiers SConscript.
scons: objectifs de construction ...
scons: ` « . est à jour.
scons: fait des objectifs de construction.
```

Les environnements de construction dans ces exemples ont été effectivement limités à un peu gcc et Visual C ++, respectivement. Dans une situation réelle, les environnements de construction contiendront probablement un grand nombre de plusieurs variables. Notez également que nous avons massé la sortie exemple ci-dessus pour rendre l'adresse mémoire de tous les objets d'une 0x700000 constante. En réalité, vous verriez un nombre hexadécimal différent pour chaque objet.

Pour le rendre plus facile de voir tout ce que vous êtes intéressé, la `Dump`méthode vous permet de spécifier une variable constrcution spécifique que vous voulez disply. Par exemple, il est pas rare de vouloir vérifier l'environnement externe utilisé pour exécuter des commandes de construction, pour vous assurer que le chemin et d' autres variables d'environnement sont mis en place la façon dont ils devraient être. Vous pouvez le faire comme suit:

```
env = environnement ()
imprimer env.Dump (ENV)
```

Ce qui peut afficher les éléments suivants lorsqu'il est exécuté sur un système POSIX:

```
% scons
scons: Lecture des fichiers SConscript ...
{ 'PATH': '/ usr / local / bin: / opt / bin: / bin: / usr / bin'}
scons: fait la lecture des fichiers SConscript.
scons: objectifs de construction ...
scons: ` « . est à jour.
scons: fait des objectifs de construction.
```

Et ce qui suit lorsqu'il est exécuté sur un système Windows:

```
C: \>scons
scons: Lecture des fichiers SConscript ...
{ 'PATH': 'C: \\ WINDOWS \\ System32',
  'PATHEXT': '.COM, .EXE, .BAT, .CMD',
  'SystemRoot': 'C: \\ WINDOWS}
scons: fait la lecture des fichiers SConscript.
scons: objectifs de construction ...
scons: ` « . est à jour.
scons: fait des objectifs de construction.
```

28.3. Qu'est - ce que les dépendances Est-ce que SCons savoir sur? l' `--tree`option

Parfois , la meilleure façon d'essayer de comprendre ce que SCons fait est tout simplement de jeter un oeil sur le graphe de dépendance qu'il construit en fonction de vos `SConscript`fichiers. L' `--tree`option affichera tout ou partie du SCons graphe de dépendance dans un « art ASCII » format graphique qui montre la hiérarchie des dépendances.

Par exemple, compte tenu de l'entrée suivante `SConstruct`fichier:

```
env = environnement (CPPPATH = [ ''])
env.Program ( 'prog', [ 'f1.c', 'f2.c', 'f3.c'])
```

Exécution SCons avec les `--tree=all` rendements d'options:

```
% scons -Q --tree=all
cc -o f1.o -c -I. f1.c
cc -o f2.o -c -I. f2.c
cc -o f3.o -c -I. f3.c
cc -o prog f1.o f2.o f3.o
+ -.
  + -SConstruct
  + -f1.c
  + -f1.o
  | + -f1.c
  | + -inc.h
  + -f2.c
  + -f2.o
  | + -f2.c
  | + -inc.h
  + -f3.c
  + -f3.o
  | + -f3.c
  | + -inc.h
  + -inc.h
  + -prog
    + -f1.o
    | + -f1.c
    | + -inc.h
    + -f2.o
    | + -f2.c
    | + -inc.h
    + -f3.o
      + -f3.c
      + -inc.h
```

L'arbre sera également imprimé lorsque l' `-n`option de (no execute) est utilisé, ce qui vous permet d'examiner le graphe de dépendance pour une configuration sans reconstruire quoi que ce soit dans l'arbre.

L' `--tree`option imprime uniquement le graphe de dépendance pour les cibles spécifiées (ou la cible par défaut (s) si aucune sont indiqués sur la ligne de commande). Donc , si vous spécifiez une cible comme `f2.o`sur la ligne de commande, l' `--tree`option n'imprimer le graphe de dépendances de ce fichier:

```
% scons -Q --tree=all f2.o
cc -o f2.o -c -I. f2.c
+ -f2.o
  + -f2.c
  + -inc.h
```

Ceci est, bien sûr, utile pour restreindre la sortie d'une configuration de construction très grande pour juste une partie dans laquelle vous êtes intéressé. Plusieurs cibles sont très bien, dans ce cas, un arbre sera imprimé pour chaque cible spécifiée:

```
% scons -Q --tree=all f1.o f3.o
cc -o f1.o -c -I. f1.c
+ -f1.o
  + -f1.c
  + -inc.h
cc -o f3.o -c -I. f3.c
+ -f3.o
  + -f3.c
  + -inc.h
```

L' `status`argument peut être utilisé pour indiquer SCons pour imprimer des informations d'état sur chaque fichier dans le graphe de dépendance:

```
% scons -Q --tree=status
cc -o f1.o -c -I. f1.c
cc -o f2.o -c -I. f2.c
cc -o f3.o -c -I. f3.c
cc -o prog f1.o f2.o f3.o
 E = existe
  R = existe dans le dépôt uniquement
   b = constructeur implicite
   B = constructeur explicite
    S = effet secondaire
     P = précieux
      A = toujours construire
       C = courant
        N = pas propre
         H = pas de cache

[E b] + -.
[CE] + -SConstruct
[CE] + -f1.c
[EBC] + -f1.o
[CE] | + -f1.c
[CE] | + -inc.h
[CE] + -f2.c
[EBC] + -f2.o
[CE] | + -f2.c
[CE] | + -inc.h
[CE] + -f3.c
[EBC] + -f3.o
[CE] | + -f3.c
[CE] | + -inc.h
[CE] + -inc.h
[EBC] + -prog
[EBC] + -f1.o
[CE] | + -f1.c
[CE] | + -inc.h
[EBC] + -f2.o
[CE] | + -f2.c
[CE] | + -inc.h
[EBC] + -f3.o
[CE] + -f3.c
[CE] + -inc.h
```

Notez que `--tree=all,status`est équivalent; l' `all` on suppose que si `status`est présent. Comme alternative à `all`, vous pouvez spécifier `--tree=derived` d'avoir SCons uniquement imprimer des cibles dérivées dans la sortie de l' arbre, en sautant les fichiers sources (comme `.c`et `.h`fichiers):

```
% scons -Q --tree=derived
cc -o f1.o -c -I. f1.c
cc -o f2.o -c -I. f2.c
cc -o f3.o -c -I. f3.c
cc -o prog f1.o f2.o f3.o
+ -.
  + -f1.o
  + -f2.o
  + -f3.o
  + -prog
    + -f1.o
    + -f2.o
    + -f3.o
```

Vous pouvez utiliser le `status` modificateur avec `derived`aussi bien:

```
% scons -Q --tree=derived,status
cc -o f1.o -c -I. f1.c
cc -o f2.o -c -I. f2.c
cc -o f3.o -c -I. f3.c
cc -o prog f1.o f2.o f3.o
 E = existe
  R = existe dans le dépôt uniquement
   b = constructeur implicite
   B = constructeur explicite
    S = effet secondaire
     P = précieux
```

```
       A = toujours construire
        C = courant
         N = pas propre
          H = pas de cache

[E b] + -.
[EBC] + -f1.o
[EBC] + -f2.o
[EBC] + -f3.o
[EBC] + -prog
[EBC] + -f1.o
[EBC] + -f2.o
[EBC] + -f3.o
```

Notez que l'ordre des `--tree=` arguments n'a pas d' importance; `--tree=status,derived`est tout à fait équivalent.

Le comportement par défaut de l' `--tree`option est de répéter toutes les dépendances à chaque fois que la dépendance bibliothèque (ou tout autre fichier de dépendance) est rencontré dans l'arbre. Si certains fichiers cibles partagent d' autres fichiers cibles, tels que les deux programmes qui utilisent la même bibliothèque:

```
env = environnement (CPPPATH = [ ''],
                  LIBS = [ 'foo'],
                  LIBPATH = [ ''])
env.Library ( 'foo', [ 'f1.c', 'f2.c', 'f3.c'])
env.Program ( 'prog1.c')
env.Program ( 'prog2.c')
```

Ensuite , il peut y avoir un *grand nombre* de répétitions dans la `--tree=`sortie:

```
% scons -Q --tree=all
cc -o f1.o -c -I. f1.c
cc -o f2.o -c -I. f2.c
cc -o f3.o -c -I. f3.c
ar rc libtruc.a f1.o f2.o f3.o
ranlib libtruc.a
cc -o prog1.o -c -I. prog1.c
cc -o prog1 prog1.o -L. -lfoo
cc -o prog2.o -c -I. prog2.c
cc -o prog2 prog2.o -L. -lfoo
+ -.
  + -SConstruct
  + -f1.c
  + -f1.o
  | + -f1.c
  | + -inc.h
  + -f2.c
  + -f2.o
  | + -f2.c
  | + -inc.h
  + -f3.c
  + -f3.o
  | + -f3.c
  | + -inc.h
  + -inc.h
  + -libfoo.a
  | + -f1.o
  | | + -f1.c
  | | + -inc.h
  | + -f2.o
  | | + -f2.c
  | | + -inc.h
  | + -f3.o
  | + -f3.c
  | + -inc.h
  + -prog1
  | + -prog1.o
  | | + -prog1.c
  | | + -inc.h
  | + -libfoo.a
  | + -f1.o
  | | + -f1.c
  | | + -inc.h
  | + -f2.o
  | | + -f2.c
  | | + -inc.h
  | + -f3.o
  | + -f3.c
  | + -inc.h
  + -prog1.c
  + -prog1.o
  | + -prog1.c
  | + -inc.h
  + -prog2
  | + -prog2.o
  | | + -prog2.c
  | | + -inc.h
  | + -libfoo.a
  | + -f1.o
  | | + -f1.c
  | | + -inc.h
  | + -f2.o
  | | + -f2.c
  | | + -inc.h
  | + -f3.o
  | + -f3.c
  | + -inc.h
  + -prog2.c
  + -prog2.o
    + -prog2.c
    + -inc.h
```

Dans une grande configuration avec de nombreuses bibliothèques internes et inclure des fichiers, cela peut très rapidement conduire à d' énormes arbres de sortie. Pour aider à rendre cela plus facile à gérer, un `prune`modificateur peut être ajouté à la liste des options, auquel cas SCons imprimeront le nom d'une cible qui a déjà été visité pendant la impression arbre `[square brackets]` comme une indication que les dépendances du fichier cible peut se trouve en regardant plus loin l'arbre:

```
% scons -Q --tree=prune
cc -o f1.o -c -I. f1.c
cc -o f2.o -c -I. f2.c
cc -o f3.o -c -I. f3.c
ar rc libtruc.a f1.o f2.o f3.o
ranlib libtruc.a
cc -o prog1.o -c -I. prog1.c
cc -o prog1 prog1.o -L. -lfoo
cc -o prog2.o -c -I. prog2.c
cc -o prog2 prog2.o -L. -lfoo
+ -.
  + -SConstruct
  + -f1.c
  + -f1.o
  | + -f1.c
  | + -inc.h
  + -f2.c
  + -f2.o
  | + -f2.c
  | + -inc.h
  + -f3.c
  + -f3.o
  | + -f3.c
  | + -inc.h
  + -inc.h
  + -libfoo.a
  | + - [f1.o]
  | + - [f2.o]
  | + - [f3.o]
  + -prog1
  | + -prog1.o
  | | + -prog1.c
  | | + -inc.h
  | + - [libtruc.a]
  + -prog1.c
  + - [prog1.o]
  + -prog2
  | + -prog2.o
  | | + -prog2.c
  | | + -inc.h
  | + - [libtruc.a]
  + -prog2.c
  + - [prog2.o]
```

Comme le `status`mot - clé, l' `prune`argument lui - même est équivalent à `--tree=all,prune`.

28.4. Comment est SCons la construction des lignes de commande exécutée? l' `--debug=presub`option

Parfois , il est utile de regarder la chaîne pré-substitution qui SCons utilise pour générer les lignes de commande exécutée. Cela peut être fait avec l' `--debug=presub`option de :

```
% scons -Q --debug=presub
Construire prog.o l'action:
  $ CC -o $ TARGET -c $ $ CFLAGS CCFLAGS $ _CCOMCOM $ SOURCES
cc -o prog.o -c -I. prog.c
Construire prog avec l'action:
  $ SMART_LINKCOM
cc -o prog prog.o
```

28.5. Où est SCons la recherche des bibliothèques? l' `--debug=findlibs`option

Pour obtenir un aperçu de ce que les noms de bibliothèques SCons est à la recherche et répertoires dans lesquels il est à la recherche, utilisez l' `--debug=findlibs`option. Compte tenu de l'entrée suivante `SConstruct`fichier:

```
env = environnement (LIBPATH = [ 'libs1', 'libs2'])
env.Program ( 'prog.c', LIBS = [ 'foo', 'bar'])
```

Et les bibliothèques `libfoo.a` et `libbar.a` dans `libs1`et `libs2`, respectivement, l' utilisation des `--debug=findlibs`rendements d'options:

```
% scons -Q --debug=findlibs
  findlibs: recherche de « libtruc.a » dans « libs1 » ...
  findlibs: ... TROUVE 'libtruc.a' dans 'libs1'
  findlibs: recherche de 'libfoo.so' dans 'libs1' ...
  findlibs: recherche de 'libfoo.so' dans 'libs2' ...
  findlibs: recherche de 'libbar.a' dans 'libs1' ...
  findlibs: recherche de 'libbar.a' dans 'libs2' ...
  findlibs: ... TROUVE 'libbar.a' dans 'libs2'
  findlibs: recherche de 'libbar.so' dans 'libs1' ...
  findlibs: recherche de 'libbar.so' dans 'libs2' ...
cc -o -c prog.o prog.c
cc -o prog prog.o -Llibs1 -Llibs2 -lfoo -lbar
```

28.6. Où est SCons Blowing Up? l' `--debug=stacktrace`option

En général, SCons essaie de garder ses messages d'erreur court et informatif. Cela signifie que nous essayons généralement d'éviter de montrer les traces de pile qui sont familiers aux programmeurs Python expérimentés, car ils contiennent généralement beaucoup plus d' informations que utile pour la plupart des gens.

Par exemple, le suivant `SConstruct`fichier:

```
Programme ( 'prog.c')
```

Génère l'erreur suivante si le `prog.c`fichier n'existe pas:

```
% scons -Q
scons: *** [prog.o] Source `prog.c 'introuvable, nécessaire à la cible` prog.o'.
```

Dans ce cas, l'erreur est assez évidente. Mais si ce ne, et que vous vouliez essayer d'obtenir plus d' informations sur l'erreur, l' `--debug=stacktrace`option vous montrer exactement où dans le SConscode source du problème se produit:

```
% scons -Q --debug=stacktrace
scons: *** [prog.o] Source `prog.c 'introuvable, nécessaire à la cible` prog.o'.
scons: trace de pile interne:
  Fichier "bootstrap / src / moteur / SCons / Job.py", ligne 199, en début
    task.prepare ()
  Fichier "bootstrap / src / moteur / SCons / Script / Main.py", ligne 173, en préparation
    retourner SCons.Taskmaster.OutOfDateTask.prepare (auto-)
  Fichier "bootstrap / src / moteur / SCons / Taskmaster.py", à la ligne 191, à préparer
    executor.prepare ()
  Fichier "bootstrap / src / moteur / SCons / Executor.py", à la ligne 396, à préparer
    soulever SCons.Errors.StopError (msg% (s, self.batches [0] .targets [0]))
```

Bien sûr, si vous avez besoin de plonger dans le SCons code source, nous aimerions savoir si, ou comment, les messages d'erreur ou d' options de dépannage aurait pu être amélioré pour éviter cela. Pas tout le monde a le temps nécessaire ou de compétences Python pour plonger dans le code source, et nous aimerions améliorer SCons pour les personnes et ...

28.7. Comment SCons prendre ses décisions? l' `--taskmastertrace`option

Le interne SCons sous - système qui gère la marche le graphe de dépendance et contrôle la prise de décision sur ce qu'il faut reconstruire est `Taskmaster`. SCons prend en charge une `--taskmastertrace` option qui indique au Taskmaster d'imprimer des informations sur les enfants (dépendances) des différents nœuds sur la promenade sur le graphique, les noeuds dépendants spécifiques sont en cours d' évaluation, et dans quel ordre.

L' `--taskmastertrace`option prend comme argument le nom d'un fichier dans lequel placer la sortie de trace, avec `-`(un seul trait d' union) indiquant que les messages de trace doivent être imprimés sur la sortie standard:

```
env = environnement (CPPPATH = [ ''])
env.Program ( 'prog.c')
```

```
% scons -Q --taskmastertrace=- prog

Taskmaster: Vous cherchez un noeud pour évaluer
Taskmaster: noeud Considérant <0 no_state 'prog'> et ses enfants:
Taskmaster: <no_state 0 'prog.o'>
Taskmaster: ref nombre ajusté: <en attente 1 'prog'>, enfant prog.o '
Taskmaster: Considérant noeud <no_state 0 'prog.o'> et ses enfants:
Taskmaster: <no_state 0 'prog.c'>
Taskmaster: <no_state 0 'inc.h'>
Taskmaster: ref nombre ajusté: <en attente 1 'prog.o'>, enfant prog.c '
Taskmaster: ref nombre ajusté: <en attente 2 'prog.o'>, enfant inc.h '
Taskmaster: Considérant noeud <no_state 0 'prog.c'> et ses enfants:
Taskmaster: Évaluation <attente 0 'prog.c'>

Task.make_ready_current (): nœud <en attente 0 'prog.c'>
Task.prepare (): nœud <up_to_date 0 'prog.c'>
Task.executed_with_callbacks (): nœud <up_to_date 0 'prog.c'>
Task.postprocess (): nœud <up_to_date 0 'prog.c'>
Task.postprocess (): suppression <up_to_date 0 'prog.c'>
Task.postprocess (): rajusté ref parent <attendant 1 'prog.o'>

Taskmaster: Vous cherchez un noeud pour évaluer
Taskmaster: Considérant noeud <no_state 0 'inc.h'> et ses enfants:
Taskmaster: Évaluation <attente 0 'inc.h'>

Task.make_ready_current (): nœud <en attente 0 'inc.h'>
Task.prepare (): nœud <up_to_date 0 'inc.h'>
Task.executed_with_callbacks (): nœud <up_to_date 0 'inc.h'>
Task.postprocess (): nœud <up_to_date 0 'inc.h'>
Task.postprocess (): suppression <up_to_date 0 'inc.h'>
Task.postprocess (): rajusté ref parent <attente 0 'prog.o'>

Taskmaster: Vous cherchez un noeud pour évaluer
Taskmaster: noeud Considérant <0 en attente 'prog.o'> et ses enfants:
Taskmaster: <up_to_date 0 'prog.c'>
Taskmaster: <up_to_date 0 'inc.h'>
Taskmaster: Évaluation <attente 0 'prog.o'>

Task.make_ready_current (): nœud <en attente 0 'prog.o'>
Task.prepare (): nœud <exécutant 0 'prog.o'>
Task.execute (): nœud <exécutant 0 'prog.o'>
cc -o prog.o -c -I. prog.c
Task.executed_with_callbacks (): nœud <exécutant 0 'prog.o'>
Task.postprocess (): nœud <exécuté 0 'prog.o'>
Task.postprocess (): suppression <0 exécuté 'prog.o'>
Task.postprocess (): rajusté ref parent <attente 0 'prog'>

Taskmaster: Vous cherchez un noeud pour évaluer
Taskmaster: noeud Considérant <en attente 0 'prog'> et ses enfants:
Taskmaster: <0 exécuté 'prog.o'>
Taskmaster: Évaluation <attente 0 'prog'>

Task.make_ready_current (): nœud <0 en attente 'prog'>
Task.prepare (): nœud <exécution 0 'prog'>
Task.execute (): nœud <exécution 0 'prog'>
cc -o prog prog.o
```

```
Task.executed_with_callbacks (): nœud <exécution 0 'prog'>
Task.postprocess (): nœud <exécuté 0 'prog'>

Taskmaster: Vous cherchez un noeud pour évaluer
Taskmaster: Aucun candidat plus.
```

L' `--taskmastertrace`option ne fournit pas d' informations sur les calculs réels impliqués pour décider si un fichier est, mais il montre la mise à jour toutes les dépendances qu'il connaît pour chaque nœud, et l'ordre dans lequel ces dépendances sont évaluées. Cela peut être utile comme un autre moyen pour déterminer si oui ou non votre SCons configuration, ou l'analyse de dépendance implicite, a effectivement identifié toutes les dépendances correctes que vous le souhaitez.

28.8. Regardez SCons préparer des cibles pour la construction: l' `--debug=prepare`option

Parfois , SCons ne construit pas la cible que vous voulez et il est difficile de comprendre pourquoi. Vous pouvez utiliser l' `--debug=prepare`option pour voir toutes les cibles SCons envisage, si elles sont déjà ou pas à jour. Le message est imprimé avant SCons décide de construire la cible.

28,9. Pourquoi un fichier en train de disparaître? l'--debug = double option

Lorsque vous utilisez l' `Duplicate`option pour créer dirs variantes, parfois , vous pouvez trouver les fichiers ne se copiés à l' endroit où vous vous attendez (ou pas du tout), ou les fichiers disparaissent mystérieusement. Ceux - ci sont généralement en raison d'une mauvaise configuration d'une sorte dans le SConstruct / SConscript, mais ils peuvent être difficiles à déboguer. Le --debug = option de double indique chaque fois qu'un fichier variante est dissociées et Relinked de sa source (ou copié, en fonction des paramètres), et montre également un message pour la suppression des fichiers variants-dir « périmé » qui n'ont plus une source correspondante fichier. Elle imprime également une ligne pour chaque cible qui est retirée juste avant la construction, car cela peut aussi être confondu avec la même chose.

Annexe A. Variables de construction

Cette annexe contient la description de toutes les variables de construction qui sont *potentiellement* disponibles « hors de la boîte » dans cette version de SCons. Que ce soit la mise ou non une variable de construction dans un environnement de construction aura effectivement un effet dépend du fait que l' un des outils et / ou les constructeurs qui utilisent la variable ont été inclus dans l'environnement de la construction.

Dans cette annexe, nous avons joint le premier `$` (signe dollar) au début de chaque nom de variable quand il apparaît dans le texte, mais a laissé hors du signe dollar dans la colonne de gauche où le nom apparaît pour chaque entrée.

__LDMODULEVERSIONFLAGS

Cette variable de construction introduit automatiquement `$_LDMODULEVERSIONFLAGS` si `$LDMODULEVERSION`est réglé. Othervise il évalue à une chaîne vide.

__SHLIBVERSIONFLAGS

Cette variable de construction introduit automatiquement `$_SHLIBVERSIONFLAGS` si `$SHLIBVERSION`est réglé. Othervise il évalue à une chaîne vide.

AR

Le archiveur de bibliothèque statique.

ARCHITECTURE

Indique l'architecture du système pour lequel le package est en cours de construction. La valeur par défaut est l'architecture du système de la machine sur laquelle SCons est en cours d' exécution. Il est utilisé pour remplir le `Architecture:` champ dans un Ipkg `control`fichier, et dans le cadre du nom d'un fichier RPM généré.

ARCOM

La ligne de commande utilisée pour générer une bibliothèque statique des fichiers objets.

ARCOMSTR

La chaîne affichée lorsqu'un fichier objet est généré à partir d' un fichier source en langage assembleur. Si ce n'est pas défini, `$ARCOM`est affiché (la ligne de commande).

```
env = environnement (ARCOMSTR = "Archiving $ TARGET")
```

ARFLAGS

Options générales transmises à l'archiveur de bibliothèque statique.

COMME

L'assembleur.

ASCOM

La ligne de commande utilisée pour générer un fichier objet à partir d'un fichier source en langage assembleur.

ASCOMSTR

La chaîne affichée lorsqu'un fichier objet est généré à partir d' un fichier source en langage assembleur. Si ce n'est pas défini, `$ASCOM`est affiché (la ligne de commande).

```
env = environnement (ASCOMSTR = "Assemblage $ TARGET")
```

ASFLAGS

Options générales transmises à l'assembleur.

ASPPCOM

La ligne de commande utilisé pour assembler un fichier source en langage assembleur en un fichier objet après la première exécution du fichier par l'intermédiaire du préprocesseur C. Toutes les options spécifiées dans les $ASFLAGSet les $CPPFLAGSvariables de construction sont inclus dans cette ligne de commande.

ASPPCOMSTR

La chaîne affichée lorsqu'un fichier d'objet est généré à partir d' un fichier source en langage assembleur après la première exécution du fichier par l'intermédiaire du préprocesseur C. Si ce n'est pas défini, $ASPPCOMest affiché (la ligne de commande).

```
env = environnement (ASPPCOMSTR = "Assemblage de $ TARGET")
```

ASPPFLAGS

Options générales lors d' un assemblage d' un fichier source en langage assembleur dans un fichier objet après la première exécution du fichier par le préprocesseur C. La valeur par défaut est d'utiliser la valeur $ASFLAGS.

BIBTEX

Le générateur de bibliographie pour la formatter TeX et typographe et la structure de formatter LaTeX et typographe.

BIBTEXCOM

La ligne de commande utilisée pour appeler le générateur de bibliographie pour la formatter TeX et typographe et la structure de formatter LaTeX et typographe.

BIBTEXCOMSTR

La chaîne affichée lors de la génération d' une bibliographie pour TeX ou LaTeX. Si ce n'est pas défini, $BIBTEXCOMest affiché (la ligne de commande).

```
env = environnement (BIBTEXCOMSTR = "bibliographie Génération target $")
```

BIBTEXFLAGS

Options générales transmises au générateur de bibliographie pour la formatter TeX et typographe et la structure de formatter LaTeX et typographe.

BitKeeper

L'exécutable BitKeeper.

BITKEEPERCOM

La ligne de commande pour extraire les fichiers source en utilisant BitKeeper.

BITKEEPERCOMSTR

La chaîne affichée lors de la récupération d' un fichier source à l' aide BitKeeper. Si ce n'est pas défini, $BITKEEPERCOM est affiché (la ligne de commande).

BITKEEPERGET

La commande ($BITKEEPER) et pour récupérer les fichiers sous - commande de source en utilisant BitKeeper.

BITKEEPERGETFLAGS

Les options qui sont passés à la BitKeeper **obtenir** sous - commande.

CONSTRUCTEURS

Un dictionnaire cartographiant les noms des constructeurs disponibles dans cet environnement à des objets sous-jacents Builder. Les constructeurs nommés Alias, CFile, CXXFile, DVI, bibliothèque, objets, PDF, PostScript, et le programme sont disponibles par défaut. Si vous initialisez cette variable lorsqu'un environnement est créé:

```
env = environnement (CONSTRUCTEURS = { 'NewBuilder': foo})
```

les constructeurs par défaut ne seront plus disponibles. Pour utiliser un nouvel objet Builder en plus des constructeurs par défaut, ajoutez votre nouvel objet Builder comme ceci:

```
env = environnement ()
env.Append (CONSTRUCTEURS = { 'NewBuilder': foo})
```

ou ca:

```
env = environnement ()
env [ 'BUILDERS] [' NewBuilder '] = foo
```

CC

Le compilateur C.

CCCOM

La ligne de commande utilisée pour compiler un fichier source de C dans un fichier objet (statique). Toutes les options spécifiées dans le $CFLAGS, $CCFLAGSet les $CPPFLAGSvariables de construction sont inclus dans cette ligne de commande.

CCCOMSTR

La chaîne affichée lorsqu'un fichier source de C est compilé dans un fichier objet (statique). Si ce n'est pas défini, $CCCOMest affiché (la ligne de commande).

```
env = environnement (CCCOMSTR = "objet statique Compiler $ TARGET")
```

CCFLAGS

Options générales qui sont passés à la C et compilateurs C. de

CCPCHFLAGS

Options ajoutées à la ligne de commande du compilateur pour soutenir la construction avec les en- têtes précompilés. La valeur par défaut se développe à l'agrandit options de ligne de commande de Microsoft Visual C appropriée lorsque la $PCHvariable de construction est définie.

CCPDBFLAGS

Options ajoutées à la ligne de commande du compilateur pour soutenir le stockage des informations de débogage dans un fichier PDB Microsoft Visual C ++. La valeur par défaut se développe se développe pour approprier options de ligne de commande de Microsoft Visual C lorsque la $PDBvariable de construction est définie.

L'option du compilateur Visual C ++ que SCons utilise par défaut pour générer des informations PDB est `/Z7`. Cela fonctionne correctement avec parallèle (`-j`) construit parce qu'il intègre les informations de débogage dans les fichiers objets intermédiaires, par opposition à partager un seul fichier PDB entre plusieurs fichiers objet. Ceci est aussi la seule façon d'obtenir des informations de débogage intégré dans une bibliothèque statique. En utilisant le `/Zi`peut au contraire obtenir de meilleures performances en temps de liaison, bien que des constructions parallèles ne fonctionneront plus.

Vous pouvez générer des fichiers PDB avec le `/Zi` commutateur en remplaçant la valeur par défaut $CCPDBFLAGSvariable comme suit:

```
env [ 'CCPDBFLAGS'] = [ '$ {(APB et "/ Zi / Fd% de" % Fichier (APB)) ou ""}]
```

Une alternative serait d'utiliser la `/Zi` mettre les informations de débogage dans un document distinct `.pdb` fichier pour chaque fichier objet en remplaçant la $CCPDBFLAGSvariable comme suit:

```
env [ 'CCPDBFLAGS'] = '/ Zi /Fd${TARGET}.pdb'
```

CCVERSION

Le numéro de version du compilateur C. Cela peut ou ne peut pas être réglé, selon le compilateur C spécifique utilisé.

CFILESUFFIX

Le suffixe pour les fichiers source C. Ceci est utilisé par le générateur de CFile interne lors de la génération des fichiers de C de Lex (.l) ou fichiers d'entrée YACC (.y). Le suffixe par défaut, bien sûr, est `.c` (en minuscules). Sur les systèmes sensibles à la casse (comme Windows), SCons traite également des `.C` fichiers (MAJUSCULES) sous forme de fichiers C.

CFLAGS

Options générales qui sont passés au compilateur C (C seulement, et non C ++).

CHANGE_SPECFILE

Un crochet pour modifier le fichier qui contrôle l'accumulation d'emballage (le `.spec`RPM, le `control`pour Ipkg, le `.wxs`MSI). Si elle est définie, la fonction sera appelée après le modèle SCons du fichier a été écrit. XXX

CHANGED_SOURCES

Un nom de variable réservé qui ne peut pas être défini ou utilisé dans un environnement de construction. (Voir « Remplacement variable » ci-dessous.)

CHANGED_TARGETS

Un nom de variable réservé qui ne peut pas être défini ou utilisé dans un environnement de construction. (Voir « Remplacement variable » ci-dessous.)

CHANGELOG

Le nom d'un fichier contenant le texte du journal des modifications à inclure dans le package. Ceci est inclus dans la`%changelog` section du RPM `.spec`fichier.

_concat

Une fonction utilisée pour produire des variables telles que `$_CPPINCFLAGS`. Il faut quatre ou cinq arguments: un préfixe à concaténer sur chaque élément, une liste d'éléments, un suffixe à concaténer sur chaque élément, un environnement pour l' interpolation variable et une fonction facultative qui sera appelée à transformer la liste avant concaténation.

```
env [ '_ CPPINCFLAGS'] = '$ ($ {_ concat (INCPREFIX, CPPPATH, INCSUFFIX, __env__, RDirs)} $)',
```

CONFIGUREDIR

Le nom du répertoire dans lequel les fichiers de test de contexte Configurer sont écrits. La valeur par défaut est `.sconf_temp` dans le répertoire de niveau supérieur contenant le `SConstruct` fichier.

CONFIGURELOG

Le nom du fichier journal contextuel Configurer. La valeur par défaut est `config.log` dans le répertoire de niveau supérieur contenant le `SConstruct` fichier.

_CPPDEFFLAGS

Une variable de construction généré automatiquement contenant les options de ligne de commande C préprocesseur pour définir des valeurs. La valeur de la $_CPPDEFFLAGScréation en annexant $CPPDEFPREFIXet $CPPDEFSUFFIX au début et à la fin de chaque définition $CPPDEFINES.

CPPDEFINES

Une plate - forme spécification indépendante des définitions de préprocesseur C. Les définitions seront ajoutées pour commander les lignes à travers la généré automatiquement$_CPPDEFFLAGSvariable de la construction (voir ci - dessus), qui est construit selon le type de valeur de $CPPDEFINES:

Si $CPPDEFINESest une chaîne, les valeurs des $CPPDEFPREFIXet $CPPDEFSUFFIX seront ajoutées les variables de construction au début et à la fin.

```
# Ajoute -Dxyz Posix lignes de commande du compilateur,
# Et / dxyz aux lignes de commande de Microsoft Visual C.
env = environnement (CPPDEFINES = 'xyz')
```

Si $CPPDEFINESune liste, les valeurs des $CPPDEFPREFIXet les $CPPDEFSUFFIX variables de construction seront ajoutées au début et à la fin de chaque élément dans la liste. Si un élément est une liste ou tuple, le premier élément est le nom étant défini et le deuxième élément est sa valeur:

```
# Ajoute -DB = 2 à -DA POSIX lignes de commande du compilateur,
# Et / DB = 2 / DA aux lignes de commande de Microsoft Visual C.
env = environnement (CPPDEFINES = [( 'B', 2), 'A'])
```

Si $CPPDEFINESest un dictionnaire, les valeurs des $CPPDEFPREFIXet les $CPPDEFSUFFIX variables de construction seront ajoutées au début et à la fin de chaque élément du dictionnaire. La clé de chaque élément de dictionnaire est un nom étant définie à la valeur correspondante de l'élément de dictionnaire; si la valeur est None, le nom est défini sans valeur explicite. Notez que les drapeaux résultants sont classés par mot - clé pour assurer que l'ordre des options sur la ligne de commande est cohérente chaque fois scons est exécuté.

```
# Ajoute -DA -DB = 2 Posix lignes de commande du compilateur,
# Et / DA / DB = 2 à Microsoft lignes Visual C commande de.
env = environnement (CPPDEFINES = { 'B': 2, 'A': Aucun})
```

CPPDEFPREFIX

Le préfixe permet de spécifier des définitions de préprocesseur sur la ligne de commande du compilateur C. Ce sera ajouté au début de chaque définition dans la $CPPDEFINESvariable de construction lorsque la $_CPPDEFFLAGSvariable est générée automatiquement.

CPPDEFSUFFIX

Le suffixe utilisé pour spécifier des définitions de préprocesseur sur la ligne de commande du compilateur C. Ce sera ajouté à la fin de chaque définition dans la $CPPDEFINESvariable de construction lorsque la $_CPPDEFFLAGSvariable est générée automatiquement.

CPPFLAGS

Spécifiés par l' utilisateur des options préprocesseur C. Ceux - ci seront inclus dans toute commande qui utilise le préprocesseur C, incluant non seulement la compilation des fichiers source de C et C via $CCCOM, $SHCCCOM, $CXXCOMet les $SHCXXCOMlignes de commande, mais aussi $FORTRANPPCOM, $SHFORTRANPPCOM, $F77PPCOMet les $SHF77PPCOMlignes de commande utilisées pour compiler un fichier source Fortran, et la $ASPPCOMcommande ligne utilisé pour assembler un fichier source en langage assembleur, après la première exécution de chaque fichier par l'intermédiaire du préprocesseur C. Notez que cette variable ne *pas* contenir -I (ou similaire) comprennent la recherche d' options de chemin qui SCons génère automatiquement à partir $CPPPATH. Voir $_CPPINCFLAGSci - dessous pour la variable qui se développe pour ces options.

_CPPINCFLAGS

Une variable de construction généré automatiquement contenant les options de ligne de commande C préprocesseur pour spécifier les répertoires à rechercher inclure des fichiers. La valeur de la $_CPPINCFLAGScréation en annexant $INCPREFIXet $INCSUFFIX au début et à la fin de chaque répertoire $CPPPATH.

CPPPATH

La liste des répertoires que le préprocesseur C va rechercher include. Le C / C ++ scanner de dépendance implicite va rechercher ces répertoires pour inclure des fichiers. Ne pas mettre explicitement inclure des arguments de répertoire dans CCFLAGS ou CXXFLAGS parce que le résultat sera non-portables et les répertoires ne sera pas recherché par le scanner de dépendance. Remarque: les noms de répertoire dans CPPPATH seront examinés en place par rapport au répertoire SConscript quand ils sont utilisés dans une commande. Pour forcer scons à Recherch un répertoire relatif à la racine de l'utilisation de l' arbre source #:

```
env = environnement (CPPPATH = '# / include')
```

Le look-up de répertoire peut également être configurée en utilisant la Dir() fonction:

```
include = Dir (include)
env = environnement (CPPPATH = comprennent)
```

La liste d'annuaire sera ajouté pour commander les lignes à travers la généré automatiquement $_CPPINCFLAGS variable de construction, qui est construit en ajoutant les valeurs de la$INCPREFIXet les $INCSUFFIX variables de construction pour le début et la fin de chaque répertoire $CPPPATH. Toutes les lignes de commande vous définissez qui ont besoin de la liste des répertoires CPPPATH devrait inclure $_CPPINCFLAGS:

```
env = environnement (CCCOM = "my_compiler $ _CPPINCFLAGS -c -o $ CIBLE SOURCE DE $")
```

CPPSUFFIXES

La liste des suffixes des fichiers qui seront scannés pour les dépendances implicites C préprocesseur (lignes #include). La liste par défaut est le suivant:

```
[ ".c", ".C", ".cxx", ".cpp", ".c ++", ".cc",
 ".h", ".H", ".hxx", ".hpp", ".hh",
 ".F", ".fpp", ".FPP",
 ".m", ".mm",
 ".S", ".SPP", ".SPP"]
```

CVS

L'exécutable CVS.

CVSCOFLAGS

Les options qui sont transmises à la sous-commande checkout CVS.

CVSCOM

La ligne de commande utilisée pour récupérer des fichiers sources à partir d'un référentiel CVS.

CVSCOMSTR

La chaîne affichée lors de la récupération d' un fichier source à partir d' un référentiel CVS. Si ce n'est pas défini, `$CVSCOM` est affiché (la ligne de commande).

CVSFLAGS

Options générales qui sont passés à CVS. Par défaut, ce paramètre est réglé `-d $CVSREPOSITORY` pour indiquer où les fichiers doivent être récupérés.

CVSREPOSITORY

Le chemin d' accès au dépôt CVS. Ceci est référencé dans la valeur par défaut la `$CVSFLAGS`valeur.

CXX

Le compilateur C ++.

CXXCOM

La ligne de commande utilisée pour compiler un fichier source en C ++ d'un fichier objet. Toutes les options spécifiées dans les `$CXXFLAGS`et les `$CPPFLAGS`variables de construction sont inclus dans cette ligne de commande.

CXXCOMSTR

La chaîne affichée lorsqu'un fichier source C de la compilation en un fichier objet (statique). Si ce n'est pas défini, `$CXXCOM`est affiché (la ligne de commande).

```
env = environnement (CXXCOMSTR = "objet statique Compiler $ TARGET")
```

CXXFILESUFFIX

Le suffixe pour les fichiers source C. Ceci est utilisé par le générateur de CXXFile interne lors de la génération des fichiers de C ++ de Lex (.ll) ou fichiers d'entrée YACC (.yy). Le suffixe par défaut est `.cc`. SCons traite également des fichiers avec les suffixes `.cpp`, `.cxx`, `.c++`et `.C++` sous forme de fichiers C ++ et les fichiers avec `.mm` suffixes sous forme de fichiers C ++ Objectif. Sur les systèmes sensibles à la casse (Linux, UNIX, et d' autres POSIX-alikes), SCons traite également des `.C` fichiers (majuscules) sous forme de fichiers C ++.

CXXFLAGS

Options générales qui sont passés au compilateur C. Par défaut, ce qui inclut la valeur `$CCFLAGS`, de sorte que ce paramètre `$CCFLAGS`affecte à la fois C et la compilation C. Si vous voulez ajouter C ++ - drapeaux spécifiques, vous devez définir ou modifier la valeur de `$CXXFLAGS`.

CXXVERSION

Le numéro de version du compilateur C. Cela peut ou ne peut pas être réglé, selon le compilateur spécifique C ++ utilisé.

DC

DC.

DCOM

DCOM.

DDEBUG

DDEBUG.

_DDEBUGFLAGS

_DDEBUGFLAGS.

DDEBUGPREFIX

DDEBUGPREFIX.

DDEBUGSUFFIX

DDEBUGSUFFIX.

LA DESCRIPTION

Une longue description du projet étant emballé. Ceci est inclus dans la section correspondante du fichier qui contrôle l'accumulation d'emballage.

DESCRIPTION_lang

Une longue description spécifique à la langue dans le but spécifique `lang`. Il est utilisé pour remplir une `%description -l`section d'un RPM `.spec`fichier.

DFILESUFFIX

DFILESUFFIX.

DFLAGPREFIX

DFLAGPREFIX.

DFLAGS

DFLAGS.

_DFLAGS

_DFLAGS.

DFLAGSUFFIX

DFLAGSUFFIX.

_DINCFLAGS

_DINCFLAGS.

DINCPREFIX

DINCPREFIX.

DINCSUFFIX

DINCSUFFIX.

dir

Une fonction qui convertit une chaîne de caractères dans une instance direction par rapport à la cible en cours de construction.

Une fonction qui convertit une chaîne de caractères dans une instance direction par rapport à la cible en cours de construction.

dirs

Une fonction qui convertit une liste de chaînes dans une liste des instances Dir par rapport à la cible en cours de construction.

Dlib

Dlib.

DLIBCOM

DLIBCOM.

_DLIBDIRFLAGS

_DLIBDIRFLAGS.

DLIBDIRPREFIX

DLIBDIRPREFIX.

DLIBDIRSUFFIX

DLIBDIRSUFFIX.

DLIBFLAGPREFIX

DLIBFLAGPREFIX.

_DLIBFLAGS

_DLIBFLAGS.

DLIBFLAGSUFFIX

DLIBFLAGSUFFIX.

DLIBLINKPREFIX

DLIBLINKPREFIX.

DLIBLINKSUFFIX

DLIBLINKSUFFIX.

DLINK

DLINK.

DLINKCOM

DLINKCOM.

DLINKFLAGPREFIX

DLINKFLAGPREFIX.

DLINKFLAGS

DLINKFLAGS.

DLINKFLAGSUFFIX

DLINKFLAGSUFFIX.

DOCBOOK_DEFAULT_XSL_EPUB

Le fichier XSLT par défaut du `DocbookEpub`constructeur dans l'environnement actuel, si aucune autre XSLT se spécifié par mot - clé.

DOCBOOK_DEFAULT_XSL_HTML

Le fichier XSLT par défaut du `DocbookHtml`constructeur dans l'environnement actuel, si aucune autre XSLT se spécifié par mot - clé.

DOCBOOK_DEFAULT_XSL_HTMLCHUNKED

Le fichier XSLT par défaut du `DocbookHtmlChunked`constructeur dans l'environnement actuel, si aucune autre XSLT se spécifié par mot - clé.

DOCBOOK_DEFAULT_XSL_HTMLHELP

Le fichier XSLT par défaut du `DocbookHtmlhelp`constructeur dans l'environnement actuel, si aucune autre XSLT se spécifié par mot - clé.

DOCBOOK_DEFAULT_XSL_MAN

Le fichier XSLT par défaut du `DocbookMan`constructeur dans l'environnement actuel, si aucune autre XSLT se spécifié par mot - clé.

DOCBOOK_DEFAULT_XSL_PDF

Le fichier XSLT par défaut du `DocbookPdf`constructeur dans l'environnement actuel, si aucune autre XSLT se spécifié par mot - clé.

DOCBOOK_DEFAULT_XSL_SLIDESHTML

Le fichier XSLT par défaut du `DocbookSlidesHtml`constructeur dans l'environnement actuel, si aucune autre XSLT se spécifié par mot - clé.

DOCBOOK_DEFAULT_XSL_SLIDESPDF

Le fichier XSLT par défaut du `DocbookSlidesPdf`constructeur dans l'environnement actuel, si aucune autre XSLT se spécifié par mot - clé.

DOCBOOK_FOP

Le chemin vers le moteur de rendu PDF `fop`ou `xep`, si l' un d'entre eux est installé (`fop`est vérifié en premier).

DOCBOOK_FOPCOM

La commande en ligne complète pour le moteur de rendu PDF `fop`ou `xep`.

DOCBOOK_FOPCOMSTR

La chaîne affiche lorsqu'un renderer comme `fop`ou `xep`est utilisé pour créer une sortie PDF à partir d' un fichier XML.

DOCBOOK_FOPFLAGS

Drapeaux de ligne de commande pour le moteur de rendu additonal PDF `fop`ou `xep`.

DOCBOOK_XMLLINT

Le chemin vers l'exécutable externe `xmllint`, si elle est installée. Notez que ce n'est utilisé comme dernière fallback pour résoudre XIncludes, si aucun libxml2 ou lxml binding Python peuvent être importés dans le système actuel.

DOCBOOK_XMLLINTCOM

La ligne de commande complet pour l'exécutable externe `xmllint`.

DOCBOOK_XMLLINTCOMSTR

La chaîne affichée lorsque `xmllint`est utilisé pour résoudre les XIncludes pour un fichier XML donné.

DOCBOOK_XMLLINTFLAGS

Drapeaux de ligne de commande additionnels pour l'exécutable externe `xmllint`.

DOCBOOK_XSLTPROC

Le chemin de l'exécutable externe `xsltproc` (ou `saxon`, `xalan`), si l' un d'entre eux est installée. Notez que ceci est utilisé uniquement en dernier repli pour les transformations XSL, si aucune libxml2 ou lxml binding Python peuvent être importés dans le système actuel.

DOCBOOK_XSLTPROCCOM

La ligne de commande complet pour l'exécutable externe`xsltproc`(ou `saxon`, `xalan`).

DOCBOOK_XSLTPROCCOMSTR

La chaîne affichée lorsque `xsltproc`est utilisé pour transformer un fichier XML via une feuille de style XSLT donné.

DOCBOOK_XSLTPROCFLAGS

Drapeaux de ligne de commande additionnels pour l'exécutable externe `xsltproc`(ou `saxon`, `xalan`).

DOCBOOK_XSLTPROCPARAMS

Paramètres additionnels qui ne sont pas destinés à l'exécutable du processeur XSLT, mais la transformation XSL lui - même. Par défaut, ils sont ajoutés à la fin de la ligne de commande pour `saxon`et `saxon-xslt`, respectivement.

DPATH

DPATH.

DSUFFIXES

La liste des suffixes des fichiers qui seront scannés pour les fichiers de paquets importés D. La liste par défaut est le suivant:

```
['.ré']
```

_DVERFLAGS

_DVERFLAGS.

DVERPREFIX

DVERPREFIX.

DVERSIONS

DVERSIONS.

DVERSUFFIX

DVERSUFFIX.

dvipdf

Le fichier DVI TeX convertisseur de fichier PDF.

DVIPDFCOM

La ligne de commande utilisée pour convertir les fichiers DVI TeX dans un fichier PDF.

DVIPDFCOMSTR

La chaîne affichée lorsqu'un fichier DVI TeX est converti en un fichier PDF. Si ce n'est pas défini, `$DVIPDFCOM`est affiché (la ligne de commande).

DVIPDFFLAGS

Options générales transmises au fichier DVI TeX convertisseur de fichier PDF.

DVIPS

Le fichier DVI TeX convertisseur PostScript.

DVIPSFLAGS

Options générales transmises au fichier DVI TeX convertisseur PostScript.

ENV

Un dictionnaire des variables d'environnement à utiliser lors de l' appel des commandes. Quand `$ENV`est utilisé dans une commande toutes les valeurs de la liste seront joints à l' aide du séparateur de chemin et toutes les autres valeurs non-chaîne simplement convertie en une chaîne. Notez que, par défaut, `scons`ne *pas* propager l'environnement en vigueur lors de l' exécution`scons` des commandes utilisées pour créer des fichiers cibles. Il en est ainsi que builds sera garantie reproductible quelles que soient les variables d'environnement définies au moment `scons` est invoquée.

Si vous voulez propager vos variables d'environnement aux commandes exécutées pour créer des fichiers cibles, vous devez le faire explicitement:

```
import os
env = environnement (ENV = os.environ)
```

Notez que vous pouvez choisir de ne propager certaines variables d'environnement. Un exemple courant est le système `PATH`variable d'environnement, de sorte que `scons` utilise les mêmes utilités que la coque appelant (ou autre procédé):

```
import os
env = environnement (ENV = { 'PATH': os.environ [ 'PATH']})
```

ÉCHAPPER

Une fonction qui sera appelée à échapper à la coquille des caractères spéciaux dans les lignes de commande. La fonction doit prendre un argument: la chaîne de ligne de commande pour échapper; et doit retourner la ligne de commande échappé.

F03

Le compilateur Fortran 03. Vous devez normalement régler la `$FORTRAN`variable qui spécifie le compilateur par défaut Fortran pour toutes les versions Fortran. Vous ne devez définir `$F03`si vous avez besoin d'utiliser un compilateur ou une version du compilateur spécifique pour les fichiers Fortran 03.

F03COM

La ligne de commande utilisée pour compiler un fichier source Fortran 03 à un fichier objet. Vous ne devez définir $F03COMsi vous avez besoin d'utiliser une ligne de commande spécifique pour les fichiers Fortran 03. Vous devez normalement régler la $FORTRANCOMvariable qui spécifie la ligne de commande par défaut pour toutes les versions Fortran.

F03COMSTR

La chaîne affichée lorsqu'un fichier source Fortran 03 est compilé dans un fichier objet. Si ce n'est pas défini, $F03COMou $FORTRANCOM (la ligne de commande) est affichée.

F03FILESUFFIXES

La liste des extensions de fichiers pour lesquels le dialecte F03 sera utilisé. Par défaut, c'est [» .f03']

F03FLAGS

Options générales définies par l' utilisateur qui sont passés au compilateur Fortran 03. Notez que cette variable ne *pas* contenir `-I` (ou similaire) comprennent la recherche d' options de chemin qui SCons génère automatiquement à partir $F03PATH. Voir$_F03INCFLAGS ci - dessous, pour la variable qui se développe pour ces options. Vous ne devez définir $F03FLAGSsi vous avez besoin de définir des options spécifiques de l' utilisateur pour Fortran 03 fichiers. Vous devez normalement régler la $FORTRANFLAGSvariable qui spécifie les options définies par l' utilisateur transmis au compilateur par défaut Fortran pour toutes les versions Fortran.

_F03INCFLAGS

Une variable de construction généré automatiquement contenant les options de ligne de commande Fortran 03 compilateur pour spécifier les répertoires à rechercher inclure des fichiers. La valeur de la $_F03INCFLAGScréation en annexant $INCPREFIXet $INCSUFFIX au début et à la fin de chaque répertoire $F03PATH.

F03PATH

La liste des répertoires que le compilateur Fortran 03 va rechercher include. Le scanner de dépendance implicite va rechercher ces répertoires pour inclure des fichiers. Ne pas mettre explicitement inclure des arguments de répertoire en $F03FLAGScar le résultat sera non-portables et les répertoires ne seront pas recherchés par le scanner de dépendance. Remarque: les noms de répertoire en $F03PATHseront examinés en place par rapport au répertoire SConscript quand ils sont utilisés dans une commande. Pour forcer `scons` à Recherch un répertoire relatif à la racine de l'utilisation de l' arbre source `#`: Vous ne devez définir $F03PATHsi vous devez définir un chemin spécifique pour inclure des fichiers Fortran 03. Vous devez normalement régler la $FORTRANPATHvariable qui spécifie le chemin d' inclusion pour le compilateur Fortran par défaut pour toutes les versions Fortran.

```
env = environnement (F03PATH = '# / include')
```

Le look-up de répertoire peut également être configurée en utilisant la `Dir`() fonction:

```
include = Dir (include)
env = environnement (F03PATH = comprennent)
```

La liste d'annuaire sera ajouté pour commander les lignes à travers la généré automatiquement $_F03INCFLAGS variable de construction, qui est construit en ajoutant les valeurs de la$INCPREFIXet les $INCSUFFIX variables de construction pour le début et la fin de chaque répertoire $F03PATH. Toutes les lignes de commande vous définissez qui ont besoin de la liste des répertoires F03PATH devrait inclure $_F03INCFLAGS:

```
env = environnement (F03COM = "my_compiler $ _F03INCFLAGS -c -o $ CIBLE SOURCE DE $")
```

F03PPCOM

La ligne de commande utilisée pour compiler un fichier source Fortran 03 à un fichier objet après la première exécution du fichier par l'intermédiaire du préprocesseur C. Toutes les options spécifiées dans les $F03FLAGSet les $CPPFLAGSvariables de construction sont inclus dans cette ligne de commande. Vous ne devez définir $F03PPCOMsi vous avez besoin d'utiliser une ligne de commande C-préprocesseur spécifique pour les fichiers Fortran 03. Vous devez normalement régler la $FORTRANPPCOMvariable qui spécifie la ligne de commande par défaut C-préprocesseur pour toutes les versions Fortran.

F03PPCOMSTR

La chaîne affichée lorsqu'un fichier source Fortran 03 est compilé dans un fichier objet après la première exécution du fichier par le préprocesseur C. Si ce n'est pas défini, $F03PPCOMou $FORTRANPPCOM (la ligne de commande) est affichée.

F03PPFILESUFFIXES

La liste des extensions de fichiers pour lesquels la compilation + préprocesseur passent pour dialecte F03 sera utilisé. Par défaut, c'est vide

F08

Le compilateur Fortran 08. Vous devez normalement régler la $FORTRANvariable qui spécifie le compilateur par défaut Fortran pour toutes les versions Fortran. Vous ne devez définir $F08si vous avez besoin d'utiliser un compilateur ou une version du compilateur spécifique pour les fichiers Fortran 08.

F08COM

La ligne de commande utilisée pour compiler un fichier source Fortran 08 à un fichier objet. Vous ne devez définir $F08COMsi vous avez besoin d'utiliser une ligne de commande spécifique pour les fichiers Fortran 08. Vous devez normalement régler la $FORTRANCOMvariable qui spécifie la ligne de commande par défaut pour toutes les versions Fortran.

F08COMSTR

La chaîne affichée lorsqu'un fichier source Fortran 08 est compilé dans un fichier objet. Si ce n'est pas défini, $F08COMou $FORTRANCOM (la ligne de commande) est affichée.

F08FILESUFFIXES

La liste des extensions de fichiers pour lesquels le dialecte F08 sera utilisé. Par défaut, c'est [» .f08']

F08FLAGS

Options générales définies par l' utilisateur qui sont passés au compilateur Fortran 08. Notez que cette variable ne *pas* contenir `-I` (ou similaire) comprennent la recherche d' options de chemin qui SCons génère automatiquement à partir $F08PATH. Voir$_F08INCFLAGS ci - dessous, pour la variable qui se développe pour ces options. Vous ne devez définir $F08FLAGSsi vous avez besoin de définir des options spécifiques de l' utilisateur pour Fortran 08 fichiers. Vous devez normalement régler la $FORTRANFLAGSvariable qui spécifie les options définies par l' utilisateur transmis au compilateur par défaut Fortran pour toutes les versions Fortran.

_F08INCFLAGS

Une variable de construction généré automatiquement contenant les options de ligne de commande Fortran 08 compilateur pour spécifier les répertoires à rechercher inclure des fichiers. La valeur de la $_F08INCFLAGScréation en annexant $INCPREFIXet $INCSUFFIX au début et à la fin de chaque répertoire $F08PATH.

F08PATH

La liste des répertoires que le compilateur Fortran 08 va rechercher include. Le scanner de dépendance implicite va rechercher ces répertoires pour inclure des fichiers. Ne pas mettre explicitement inclure des arguments de répertoire en $F08FLAGScar le résultat sera non-portables et les répertoires ne seront pas recherchés par le scanner de dépendance. Remarque: les noms de répertoire en $F08PATHseront examinés en place par rapport au répertoire SConscript quand ils sont utilisés dans une commande. Pour forcer `scons` à Recherch un répertoire relatif à la racine de l'utilisation de l' arbre source `#`: Vous ne devez définir $F08PATHsi vous devez définir un chemin spécifique pour inclure des fichiers Fortran 08. Vous devez normalement régler la $FORTRANPATHvariable qui spécifie le chemin d' inclusion pour le compilateur Fortran par défaut pour toutes les versions Fortran.

```
env = environnement (F08PATH = '# / include')
```

Le look-up de répertoire peut également être configurée en utilisant la `Dir`() fonction:

```
include = Dir (include)
env = environnement (F08PATH = comprennent)
```

La liste d'annuaire sera ajouté pour commander les lignes à travers la généré automatiquement $_F08INCFLAGS variable de construction, qui est construit en ajoutant les valeurs de la$INCPREFIXet les $INCSUFFIX variables de construction pour le début et la fin de chaque répertoire $F08PATH. Toutes les lignes de commande vous définissez qui ont besoin de la liste des répertoires F08PATH devrait inclure $_F08INCFLAGS:

```
env = environnement (F08COM = "my_compiler $ _F08INCFLAGS -c -o $ CIBLE SOURCE DE $")
```

F08PPCOM

La ligne de commande utilisée pour compiler un fichier source Fortran 08 à un fichier objet après la première exécution du fichier par l'intermédiaire du préprocesseur C. Toutes les options spécifiées dans les $F08FLAGSet les $CPPFLAGSvariables de construction sont inclus dans cette ligne de commande. Vous ne devez définir $F08PPCOMsi vous avez besoin d'utiliser une ligne de commande C-préprocesseur spécifique pour les fichiers Fortran 08. Vous devez normalement régler la $FORTRANPPCOMvariable qui spécifie la ligne de commande par défaut C-préprocesseur pour toutes les versions Fortran.

F08PPCOMSTR

La chaîne affichée lorsqu'un fichier source Fortran 08 est compilé dans un fichier objet après la première exécution du fichier par le préprocesseur C. Si ce n'est pas défini, $F08PPCOMou $FORTRANPPCOM (la ligne de commande) est affichée.

F08PPFILESUFFIXES

La liste des extensions de fichiers pour lesquels la compilation + préprocesseur passent pour dialecte F08 sera utilisé. Par défaut, c'est vide

F77

Le compilateur Fortran 77. Vous devez normalement régler la $FORTRANvariable qui spécifie le compilateur par défaut Fortran pour toutes les versions Fortran. Vous ne devez définir $F77si vous avez besoin d'utiliser un compilateur ou une version du compilateur spécifique pour les fichiers Fortran 77.

F77COM

La ligne de commande utilisée pour compiler un fichier source Fortran 77 à un fichier objet. Vous ne devez définir $F77COMsi vous avez besoin d'utiliser une ligne de commande spécifique pour les fichiers Fortran 77. Vous devez normalement régler la $FORTRANCOMvariable qui spécifie la ligne de commande par défaut pour toutes les versions Fortran.

F77COMSTR

La chaîne affichée lorsqu'un fichier source Fortran 77 est compilé dans un fichier objet. Si ce n'est pas défini, $F77COMou $FORTRANCOM(la ligne de commande) est affichée.

F77FILESUFFIXES

La liste des extensions de fichiers pour lesquels le dialecte F77 sera utilisé. Par défaut, c'est [» .f77']

F77FLAGS

Options générales définies par l' utilisateur qui sont passés au compilateur Fortran 77. Notez que cette variable ne *pas* contenir `-I` (ou similaire) comprennent la recherche d' options de chemin qui SCons génère automatiquement à partir $F77PATH. Voir$_F77INCFLAGS ci - dessous, pour la variable qui se développe pour ces options. Vous ne devez définir $F77FLAGSsi vous avez besoin de définir des options spécifiques de l' utilisateur pour les fichiers Fortran 77. Vous devez normalement régler la $FORTRANFLAGSvariable qui spécifie les options définies par l' utilisateur transmis au compilateur par défaut Fortran pour toutes les versions Fortran.

_F77INCFLAGS

Une variable de construction généré automatiquement contenant les options de ligne de commande Fortran 77 compilateur pour spécifier les répertoires à rechercher inclure des fichiers. La valeur de la $_F77INCFLAGScréation en annexant $INCPREFIXet $INCSUFFIX au début et à la fin de chaque répertoire $F77PATH.

F77PATH

La liste des répertoires que le compilateur Fortran 77 va rechercher include. Le scanner de dépendance implicite va rechercher ces répertoires pour inclure des fichiers. Ne pas mettre explicitement inclure des arguments de répertoire en $F77FLAGScar le résultat sera non-portables et les répertoires ne seront pas recherchés par le scanner de dépendance. Remarque: les noms de répertoire en $F77PATHseront examinés en place par rapport au répertoire SConscript quand ils sont utilisés dans une commande. Pour forcer `scons` à Recherch un répertoire relatif à la racine de l'utilisation de l' arbre source #: Vous ne devez définir $F77PATHsi vous devez définir un chemin spécifique pour inclure des fichiers Fortran 77. Vous devez normalement régler la $FORTRANPATHvariable qui spécifie le chemin d' inclusion pour le compilateur Fortran par défaut pour toutes les versions Fortran.

```
env = environnement (F77PATH = '# / include')
```

Le look-up de répertoire peut également être configurée en utilisant la `Dir`() fonction:

```
include = Dir (include)
env = environnement (F77PATH = comprennent)
```

La liste d'annuaire sera ajouté pour commander les lignes à travers la généré automatiquement $_F77INCFLAGS variable de construction, qui est construit en ajoutant les valeurs de la$INCPREFIXet les $INCSUFFIX variables de construction pour le début et la fin de chaque répertoire $F77PATH. Toutes les lignes de commande vous définissez qui ont besoin de la liste des répertoires F77PATH devrait inclure $_F77INCFLAGS:

```
env = environnement (F77COM = "my_compiler $ _F77INCFLAGS -c -o $ CIBLE SOURCE DE $")
```

F77PPCOM

La ligne de commande utilisée pour compiler un fichier source Fortran 77 à un fichier objet après la première exécution du fichier par l'intermédiaire du préprocesseur C. Toutes les options spécifiées dans les $F77FLAGSet les $CPPFLAGSvariables de construction sont inclus dans cette ligne de commande. Vous ne devez définir $F77PPCOMsi vous avez besoin d'utiliser une ligne de commande C-préprocesseur spécifique pour les fichiers Fortran 77. Vous devez normalement régler la $FORTRANPPCOMvariable qui spécifie la ligne de commande par défaut C-préprocesseur pour toutes les versions Fortran.

F77PPCOMSTR

La chaîne affichée lorsqu'un fichier source Fortran 77 est compilé en un fichier objet après la première exécution du fichier par l'intermédiaire du préprocesseur C. Si ce n'est pas défini, $F77PPCOMou $FORTRANPPCOM (la ligne de commande) est affichée.

F77PPFILESUFFIXES

La liste des extensions de fichiers pour lesquels la compilation + préprocesseur passent pour dialecte F77 sera utilisé. Par défaut, c'est vide

F90

Le compilateur Fortran 90. Vous devez normalement régler la $FORTRANvariable qui spécifie le compilateur par défaut Fortran pour toutes les versions Fortran. Vous ne devez définir $F90si vous avez besoin d'utiliser un compilateur ou une version du compilateur spécifique pour les fichiers Fortran 90.

F90COM

La ligne de commande utilisée pour compiler un fichier source Fortran 90 à un fichier objet. Vous ne devez définir $F90COMsi vous avez besoin d'utiliser une ligne de commande spécifique pour les fichiers Fortran 90. Vous devez normalement régler la $FORTRANCOMvariable qui spécifie la ligne de commande par défaut pour toutes les versions Fortran.

F90COMSTR

La chaîne affichée lorsqu'un fichier source Fortran 90 est compilé dans un fichier objet. Si ce n'est pas défini, $F90COMou $FORTRANCOM (la ligne de commande) est affichée.

F90FILESUFFIXES

La liste des extensions de fichiers pour lesquels le dialecte F90 sera utilisé. Par défaut, c'est [» .f90']

F90FLAGS

Options générales définies par l' utilisateur qui sont passés au compilateur Fortran 90. Notez que cette variable ne *pas* contenir`-I` (ou similaire) comprennent la recherche d' options de chemin qui SCons génère automatiquement à partir $F90PATH. Voir$_F90INCFLAGS ci - dessous, pour la variable qui se développe pour ces options. Vous ne devez définir $F90FLAGSsi vous avez besoin de définir des options spécifiques de l' utilisateur pour les fichiers Fortran 90. Vous devez normalement régler la $FORTRANFLAGSvariable qui spécifie les options définies par l' utilisateur transmis au compilateur par défaut Fortran pour toutes les versions Fortran.

_F90INCFLAGS

Une variable de construction généré automatiquement contenant les options de ligne de commande Fortran 90 compilateur pour spécifier les répertoires à rechercher inclure des fichiers. La valeur de la $_F90INCFLAGScréation en annexant $INCPREFIXet $INCSUFFIX au début et à la fin de chaque répertoire $F90PATH.

F90PATH

La liste des répertoires que le compilateur Fortran 90 va rechercher include. Le scanner de dépendance implicite va rechercher ces répertoires pour inclure des fichiers. Ne pas mettre explicitement inclure des arguments de répertoire en $F90FLAGScar le résultat sera non-portables et les répertoires ne seront pas recherchés par le scanner de dépendance. Remarque: les noms de répertoire en $F90PATHseront examinés en place par rapport au répertoire SConscript quand ils sont utilisés dans une commande. Pour forcer `scons` à Recherch un répertoire relatif à la racine de l'utilisation de l' arbre source #: Vous ne devez définir $F90PATHsi vous devez définir un chemin spécifique pour inclure des fichiers Fortran

90. Vous devez normalement régler la $FORTRANPATHvariable qui spécifie le chemin d' inclusion pour le compilateur Fortran par défaut pour toutes les versions Fortran.

```
env = environnement (F90PATH = '# / include')
```

Le look-up de répertoire peut également être configurée en utilisant la Dir() fonction:

```
include = Dir (include)
env = environnement (F90PATH = comprennent)
```

La liste d'annuaire sera ajouté pour commander les lignes à travers la généré automatiquement $_F90INCFLAGS variable de construction, qui est construit en ajoutant les valeurs de la$INCPREFIXet les $INCSUFFIX variables de construction pour le début et la fin de chaque répertoire $F90PATH. Toutes les lignes de commande vous définissez qui ont besoin de la liste des répertoires F90PATH devrait inclure $_F90INCFLAGS:

```
env = environnement (F90COM = "my_compiler $ _F90INCFLAGS -c -o $ CIBLE SOURCE DE $")
```

F90PPCOM

La ligne de commande utilisée pour compiler un fichier source Fortran 90 à un fichier objet après la première exécution du fichier par l'intermédiaire du préprocesseur C. Toutes les options spécifiées dans les $F90FLAGSet les $CPPFLAGSvariables de construction sont inclus dans cette ligne de commande. Vous ne devez définir $F90PPCOMsi vous avez besoin d'utiliser une ligne de commande C-préprocesseur spécifique pour les fichiers Fortran 90. Vous devez normalement régler la $FORTRANPPCOMvariable qui spécifie la ligne de commande par défaut C-préprocesseur pour toutes les versions Fortran.

F90PPCOMSTR

La chaîne affichée lorsqu'un fichier source Fortran 90 est compilé après la première exécution du fichier par l'intermédiaire du préprocesseur C. Si ce n'est pas défini, $F90PPCOMou $FORTRANPPCOM (la ligne de commande) est affichée.

F90PPFILESUFFIXES

La liste des extensions de fichiers pour lesquels la compilation + préprocesseur passent pour dialecte F90 sera utilisé. Par défaut, c'est vide

F95

Le compilateur Fortran 95. Vous devez normalement régler la $FORTRANvariable qui spécifie le compilateur par défaut Fortran pour toutes les versions Fortran. Vous ne devez définir $F95si vous avez besoin d'utiliser un compilateur ou une version du compilateur spécifique pour les fichiers Fortran 95.

F95COM

La ligne de commande utilisée pour compiler un fichier source Fortran 95 à un fichier objet. Vous ne devez définir $F95COMsi vous avez besoin d'utiliser une ligne de commande spécifique pour les fichiers Fortran 95. Vous devez normalement régler la $FORTRANCOMvariable qui spécifie la ligne de commande par défaut pour toutes les versions Fortran.

F95COMSTR

La chaîne affichée lorsqu'un fichier source Fortran 95 est compilé dans un fichier objet. Si ce n'est pas défini, $F95COMou $FORTRANCOM(la ligne de commande) est affichée.

F95FILESUFFIXES

La liste des extensions de fichiers pour lesquels le dialecte F95 sera utilisé. Par défaut, c'est [» .f95']

F95FLAGS

Options générales définies par l' utilisateur qui sont passés au compilateur Fortran 95. Notez que cette variable ne *pas* contenir-I (ou similaire) comprennent la recherche d' options de chemin qui SCons génère automatiquement à partir $F95PATH. Voir$_F95INCFLAGS ci - dessous, pour la variable qui se développe pour ces options. Vous ne devez définir $F95FLAGSsi vous avez besoin de définir des options spécifiques de l' utilisateur pour les fichiers Fortran 95. Vous devez normalement régler la $FORTRANFLAGSvariable qui spécifie les options définies par l' utilisateur transmis au compilateur par défaut Fortran pour toutes les versions Fortran.

_F95INCFLAGS

Une variable de construction généré automatiquement contenant les options de ligne de commande Fortran 95 compilateur pour spécifier les répertoires à rechercher inclure des fichiers. La valeur de la $_F95INCFLAGScréation en annexant $INCPREFIXet $INCSUFFIX au début et à la fin de chaque répertoire $F95PATH.

F95PATH

La liste des répertoires que le compilateur Fortran 95 va rechercher include. Le scanner de dépendance implicite va rechercher ces répertoires pour inclure des fichiers. Ne pas mettre explicitement inclure des arguments de répertoire en $F95FLAGScar le résultat sera non-portables et les répertoires ne seront pas recherchés par le scanner de dépendance. Remarque: les noms de répertoire en $F95PATHseront examinés en place par rapport au répertoire SConscript quand ils sont utilisés dans une commande. Pour forcer scons à Recherch un répertoire relatif à la racine de l'utilisation de l' arbre source #: Vous ne devez définir $F95PATHsi vous devez définir un chemin pour les fichiers spécifiques comprennent Fortran 95. Vous devez normalement régler la $FORTRANPATHvariable qui spécifie le chemin d' inclusion pour le compilateur Fortran par défaut pour toutes les versions Fortran.

```
env = environnement (F95PATH = '# / include')
```

Le look-up de répertoire peut également être configurée en utilisant la Dir() fonction:

```
include = Dir (include)
env = environnement (F95PATH = comprennent)
```

La liste d'annuaire sera ajouté pour commander les lignes à travers la généré automatiquement $_F95INCFLAGS variable de construction, qui est construit en ajoutant les valeurs de la$INCPREFIXet les $INCSUFFIX variables de construction pour le début et la fin de chaque

répertoire $F95PATH. Toutes les lignes de commande vous définissez qui ont besoin de la liste des répertoires F95PATH devrait inclure $_F95INCFLAGS:

```
env = environnement (F95COM = "my_compiler $ _F95INCFLAGS -c -o $ CIBLE SOURCE DE $")
```

F95PPCOM

La ligne de commande utilisée pour compiler un fichier source Fortran 95 à un fichier objet après la première exécution du fichier par l'intermédiaire du préprocesseur C. Toutes les options spécifiées dans les $F95FLAGSet les $CPPFLAGSvariables de construction sont inclus dans cette ligne de commande. Vous ne devez définir $F95PPCOMsi vous avez besoin d'utiliser une ligne de commande C-préprocesseur spécifique pour les fichiers Fortran 95. Vous devez normalement régler la $FORTRANPPCOMvariable qui spécifie la ligne de commande par défaut C-préprocesseur pour toutes les versions Fortran.

F95PPCOMSTR

La chaîne affichée lorsqu'un fichier source Fortran 95 est compilé en un fichier objet après la première exécution du fichier par l'intermédiaire du préprocesseur C. Si ce n'est pas défini, $F95PPCOMou $FORTRANPPCOM (la ligne de commande) est affichée.

F95PPFILESUFFIXES

La liste des extensions de fichiers pour lesquels la compilation + préprocesseur passent pour dialecte F95 sera utilisé. Par défaut, c'est vide

Fichier

Une fonction qui convertit une chaîne de caractères dans une instance de fichier par rapport à la cible en cours de construction.

Une fonction qui convertit une chaîne de caractères dans une instance de fichier par rapport à la cible en cours de construction.

FORTRAN

Le compilateur Fortran par défaut pour toutes les versions de Fortran.

FORTRANCOM

La ligne de commande utilisée pour compiler un fichier source Fortran à un fichier objet. Par défaut, toutes les options spécifiées dans le $FORTRANFLAGS, $CPPFLAGS, $_CPPDEFFLAGS,$_FORTRANMODFLAGet les $_FORTRANINCFLAGSvariables de construction sont inclus dans cette ligne de commande.

FORTRANCOMSTR

La chaîne affichée lorsqu'un fichier source Fortran est compilé dans un fichier objet. Si ce n'est pas défini, $FORTRANCOM est affiché (la ligne de commande).

FORTRANFILESUFFIXES

La liste des extensions de fichiers pour lesquels le dialecte Fortran sera utilisé. Par défaut, c'est ['.f', '.pour', '.ftn']

FORTRANFLAGS

Options générales définies par l' utilisateur qui sont passés au compilateur Fortran. Notez que cette variable ne *pas* contenir `-I`(ou similaire) comprennent ou options de chemin de recherche de module qui génère automatiquement à partir SCons $FORTRANPATH. Voir $_FORTRANINCFLAGSet $_FORTRANMODFLAGci - dessous pour les variables qui élargissent ces options.

_FORTRANINCFLAGS

Une variable de construction généré automatiquement contenant les options de ligne de commande compilateur Fortran pour spécifier les répertoires à rechercher pour inclure des fichiers et des fichiers de module. La valeur de $_FORTRANINCFLAGSest créée par préfixer / annexant $INCPREFIXet $INCSUFFIX au début et à la fin de chaque répertoire $FORTRANPATH.

FORTRANMODDIR

emplacement du répertoire où le compilateur Fortran doit placer tous les fichiers du module génère. Cette variable est vide, par défaut. Certains Fortran vont ajouter à l'intérieur de ce répertoire dans le chemin de recherche des fichiers du module, ainsi.

FORTRANMODDIRPREFIX

Le préfixe utilisé pour spécifier un répertoire de module sur la ligne de commande du compilateur Fortran. Ce sera ajouté au début du répertoire dans les $FORTRANMODDIRvariables de construction lorsque les $_FORTRANMODFLAGest généré automatiquement les variables.

FORTRANMODDIRSUFFIX

Le suffixe utilisé pour spécifier un répertoire de module sur la ligne de commande du compilateur Fortran. Ce sera ajouté au début du répertoire dans les $FORTRANMODDIRvariables de construction lorsque les $_FORTRANMODFLAGest généré automatiquement les variables.

_FORTRANMODFLAG

Une variable de construction généré automatiquement contenant l'option de ligne de commande compilateur Fortran pour spécifier l'emplacement du répertoire où le compilateur Fortran doit placer tous les fichiers de module qui arrivent à se produit lors de la compilation. La valeur de $_FORTRANMODFLAGest créée par préfixer / annexant $FORTRANMODDIRPREFIXet $FORTRANMODDIRSUFFIXau début et à la fin du répertoire $FORTRANMODDIR.

FORTRANMODPREFIX

Le préfixe de fichier de module utilisé par le compilateur Fortran. SCons suppose que le compilateur Fortran suit la convention de nommage quasi-standard pour les fichiers de module `module_name.mod`. En conséquence, cette variable est vide, par défaut. Pour les situations dans lesquelles le compilateur ne suit pas nécessairement la convention normale, l'utilisateur peut utiliser cette variable. Sa valeur sera ajoutée à chaque nom de fichier du module comme scons tente de résoudre les dépendances.

FORTRANMODSUFFIX

Le suffixe du fichier module utilisé par le compilateur Fortran. SCons suppose que le compilateur Fortran suit la convention de nommage quasi-standard pour les fichiers de module`module_name.mod`. En conséquence, cette variable est réglée sur « .mod », par défaut. Pour les situations dans lesquelles le compilateur ne suit pas nécessairement la convention normale, l'utilisateur peut utiliser cette variable. Sa valeur sera ajoutée à chaque nom de fichier du module comme scons tente de résoudre les dépendances.

FORTRANPATH

La liste des répertoires que le compilateur Fortran recherchera les fichiers à inclure et (pour certains compilateurs) fichiers du module. Le scanner de dépendance implicite Fortran va rechercher ces répertoires pour inclure des fichiers (mais pas le module fichiers car ils sont générées automatiquement et, en tant que telle, ne peut pas réellement exister au moment de l'analyse a lieu). Ne pas mettre explicitement inclure des arguments de répertoire dans FORTRANFLAGS car le résultat sera non-portables et les répertoires ne sera pas recherché par le scanner de dépendance. Remarque: les noms de répertoire dans FORTRANPATH seront examinés en place par rapport au répertoire SConscript quand ils sont utilisés dans une commande. Pour forcer `scons` à Recherch un répertoire relatif à la racine de l'utilisation de l' arbre source `#`:

```
env = environnement (FORTRANPATH = '# / include')
```

Le look-up de répertoire peut également être configurée en utilisant la `Dir`() fonction:

```
include = Dir (include)
env = environnement (FORTRANPATH = include)
```

La liste d'annuaire sera ajouté pour commander les lignes à travers la généré automatiquement $_FORTRANINCFLAGS variable de construction, qui est construit en ajoutant les valeurs de la$INCPREFIXet les $INCSUFFIX variables de construction pour le début et la fin de chaque répertoire $FORTRANPATH. Toutes les lignes de commande vous définissez qui ont besoin de la liste des répertoires FORTRANPATH devrait inclure $_FORTRANINCFLAGS:

```
env = environnement (FORTRANCOM = "my_compiler $ _FORTRANINCFLAGS -c -o $ CIBLE SOURCE DE $")
```

FORTRANPPCOM

La ligne de commande utilisée pour compiler un fichier source Fortran à un fichier objet après la première exécution du fichier par l'intermédiaire du préprocesseur C. Par défaut, toutes les options spécifiées dans le $FORTRANFLAGS, $CPPFLAGS, $_CPPDEFFLAGS,$_FORTRANMODFLAGet les $_FORTRANINCFLAGS variables de construction sont inclus dans cette ligne de commande.

FORTRANPPCOMSTR

La chaîne affichée lorsqu'un fichier source Fortran est compilé en un fichier objet après la première exécution du fichier par l'intermédiaire du préprocesseur C. Si ce n'est pas défini, $FORTRANPPCOM est affiché (la ligne de commande).

FORTRANPPFILESUFFIXES

La liste des extensions de fichiers pour lesquels le passage de compilation + préprocesseur pour le dialecte Fortran sera utilisé. Par défaut, c'est ['.fpp', '.FPP']

FORTRANSUFFIXES

La liste des suffixes des fichiers qui seront scannés pour les dépendances implicites Fortran (lignes et déclarations COMPREND USE). La liste par défaut est le suivant:

```
[ ".F", ".F", ".pour", ".FOR", ".ftn", ".FTN", ".fpp", ".FPP",
".f77" ," .F77" , ".f90" ," .F90" , ".f95" ," .F95" ]
```

FRAMEWORKPATH

Sous Mac OS X avec gcc, une liste contenant les chemins à rechercher des cadres. Utilisé par le compilateur pour trouver le style de cadre comprend comme #include <fmwk / header.h>. Utilisé par l'éditeur de liens pour trouver des cadres définis par l' utilisateur lors de la liaison (voir $FRAMEWORKS). Par exemple:

```
 env.AppendUnique (FRAMEWORKPATH = '# myframeworkdir')
```

ajouterai

```
  ... -Fmyframeworkdir
```

les lignes de commande compilateur et l'éditeur de liens.

_FRAMEWORKPATH

Sous Mac OS X avec gcc, une variable de construction généré automatiquement contenant les options de liaison de ligne de commande correspondant à $FRAMEWORKPATH.

FRAMEWORKPATHPREFIX

Sous Mac OS X avec gcc, le préfixe à utiliser pour les entrées de FRAMEWORKPATH. (voir $FRAMEWORKPATH). La valeur par défaut est `-F`.

FRAMEWORKPREFIX

Sous Mac OS X avec gcc, le préfixe à utiliser pour relier dans les cadres (voir $FRAMEWORKS). La valeur par défaut est `-framework`.

_FRAMEWORKS

Sous Mac OS X avec gcc, une variable de construction généré automatiquement contenant les options de ligne de commande éditeur de liens pour lier avec CADRES.

CADRES

Sous Mac OS X avec gcc, une liste des noms de cadres à lier dans un programme ou d'une bibliothèque ou d'un ensemble partagé. La valeur par défaut est la liste vide. Par exemple:

```
env.AppendUnique (CADRES = split ( 'Système Cocoa SystemConfiguration'))
```

FRAMEWORKSFLAGS

Sous Mac OS X avec gcc, cadres généraux fournis par l' utilisateur des options à ajouter à la fin d'une ligne de commande la construction d' un module chargeable. (Cela a été largement supplanté par le $FRAMEWORKPATH, $FRAMEWORKPATHPREFIX,$FRAMEWORKPREFIXet $FRAMEWORKSvariables décrites ci - dessus.)

GS

Le programme Ghostscript utilisé, par exemple pour convertir PostScript en fichiers PDF.

GSCOM

La ligne de commande complète Ghostscript utilisée pour le processus de conversion. Sa valeur par défaut est « `$GS $GSFLAGS -sOutputFile=$TARGET $SOURCES` » .

GSCOMSTR

La chaîne affichée lorsque Ghostscript est appelé pour le processus de conversion. Si ce n'est pas définie (la valeur par défaut), puis $GSCOM(la ligne de commande) est affichée.

GSFLAGS

Options générales passées au programme Ghostscript, lors de la conversion PostScript en fichiers PDF par exemple. Sa valeur par défaut est « `-dNOPAUSE -dBATCH -sDEVICE=pdfwrite` »

HOST_ARCH

Définit l'architecture hôte pour le compilateur Visual Studio. Si non défini, par défaut à l'architecture hôte détectée: notez que cela peut dépendre du python que vous utilisez. Cette variable doit être passé comme argument au constructeur de l'environnement (); il a plus tard la mise en aucun effet.

Les valeurs valides sont les mêmes que pour `$TARGET_ARCH`.

Ceci est actuellement utilisé uniquement sous Windows, mais dans l'avenir, il sera utilisé sur d'autres systèmes d'exploitation aussi bien.

Le nom de l'architecture du matériel hôte utilisé pour créer l'environnement. Si une plate-forme est spécifiée lors de la création de l'environnement, alors que la logique de la plate-forme traitera définir cette valeur. Cette valeur est immuable, et ne doit pas être modifié par l'utilisateur après l'environnement est initialisé. Actuellement, seulement défini pour Win32.

HOST_OS

Le nom du système d'exploitation hôte utilisé pour créer l'environnement. Si une plate-forme est spécifiée lors de la création de l'environnement, alors que la logique de la plate-forme traitera définir cette valeur. Cette valeur est immuable, et ne doit pas être modifié par l'utilisateur après l'environnement est initialisé. Actuellement, seulement défini pour Win32.

IDLSUFFIXES

La liste des suffixes des fichiers qui seront scannés pour les dépendances implicites IDL (#include ou lignes importation). La liste par défaut est le suivant:

```
[ ".idl", ".IDL"]
```

IMPLIBNOVERSIONSYMLINKS

Utilisé pour remplacer $SHLIBNOVERSIONSYMLINKS/ $LDMODULENOVERSIONSYMLINKSlors de la création de bibliothèque d'importation versionné pour une bibliothèque partagée / module chargeable. Si non défini, $SHLIBNOVERSIONSYMLINKS/ $LDMODULENOVERSIONSYMLINKSest utilisé pour déterminer si pour désactiver la génération symlink ou non.

IMPLIBPREFIX

Le préfixe utilisé pour les noms de bibliothèque d'importation. Par exemple, Cygwin utilise des bibliothèques d'importation (`libfoo.dll.a`) en paire avec des bibliothèques dynamiques (`cygfoo.dll`). L' cyglinkéditeur de liens met $IMPLIBPREFIXà `'lib'`et $SHLIBPREFIX à `'cyg'`.

IMPLIBSUFFIX

Le suffixe utilisé pour les noms de bibliothèque d'importation. Par exemple, Cygwin utilise des bibliothèques d'importation (`libfoo.dll.a`) en paire avec des bibliothèques dynamiques (`cygfoo.dll`). L' cyglinkéditeur de liens met $IMPLIBSUFFIXà `'.dll.a'`et $SHLIBSUFFIX à `'.dll'`.

IMPLIBVERSION

Utilisé pour remplacer $SHLIBVERSION/ $LDMODULEVERSIONlors de la génération bibliothèque d'importation versionnée pour une bibliothèque partagée / module chargeable. Si non défini, le $SHLIBVERSION/ $LDMODULEVERSIONest utilisé pour déterminer la version de la bibliothèque d'importation versionné.

IMPLICIT_COMMAND_DEPENDENCIES

Détermine si SCons ajoutera les dépendances implicites pour les commandes exécutées pour construire des cibles.

Par défaut, SCons ajoutera à chaque cible une dépendance implicite sur la commande représentée par le premier argument sur une ligne de commande exécutée. Le fichier spécifique pour la dépendance se trouve en recherchant la PATH variable dans l'ENV environnement utilisé pour exécuter la commande.

Si la variable de construction `$IMPLICIT_COMMAND_DEPENDENCIES` est définie sur une valeur fausse (`None`, `False`, `0`, etc.), la dépendance implicite ne sera pas ajoutée aux cibles construites avec cet environnement de construction.

```
env = environnement (IMPLICIT_COMMAND_DEPENDENCIES = 0)
```

INCPREFIX

Le préfixe utilisé pour spécifier un include sur la ligne de commande du compilateur C. Ce sera ajouté au début de chaque répertoire dans les `$CPPPATH`et `$FORTRANPATH`variables de construction lorsque les `$_CPPINCFLAGS`et `$_FORTRANINCFLAGS`variables sont générées automatiquement.

INCSUFFIX

Le suffixe utilisé pour spécifier un include sur la ligne de commande du compilateur C. Ce sera ajouté à la fin de chaque répertoire dans les `$CPPPATH`et `$FORTRANPATH`variables de construction lorsque les `$_CPPINCFLAGS`et `$_FORTRANINCFLAGS`variables sont générées automatiquement.

INSTALLER

Une fonction à appeler à installer un fichier dans un nom de fichier de destination. La fonction par défaut copie le fichier dans la destination (et définit le mode de fichier de destination et les bits de permission pour qu'il corresponde à celui du fichier source). La fonction prend les arguments suivants:

```
def installer (dest, source, env):
```

`dest` est le nom du chemin du fichier de destination. `source` est le nom du chemin du fichier source. `env` est l'environnement de construction (un dictionnaire de valeurs de construction) en vigueur pour cette installation de fichiers.

INSTALLSTR

La chaîne affichée lorsqu'un fichier est installé dans un nom de fichier de destination. La valeur par défaut est le suivant:

```
Fichier d'installation: "$ SOURCE" que "$ TARGET"
```

INTEL_C_COMPILER_VERSION

Défini par l'outil « intelc » au numéro de version du compilateur Intel C sélectionné pour être utilisé.

POT

L'outil archive Java.

L'outil archive Java.

JARCHDIR

Le répertoire dans lequel l'outil archive Java doit changer (en utilisant l' `-C` option).

Le répertoire dans lequel l'outil archive Java doit changer (en utilisant l' `-C` option).

JARCOM

La ligne de commande utilisée pour appeler l'outil archive Java.

La ligne de commande utilisée pour appeler l'outil archive Java.

JARCOMSTR

La chaîne affichée lorsque l'outil archive Java est appelé Si ce n'est pas défini, `$JARCOM`(la ligne de commande) est affichée.

```
env = environnement (JARCOMSTR = "JARchiving SOURCES $ en $ TARGET")
```

La chaîne affichée lorsque l'outil archive Java est appelé Si ce n'est pas défini, `$JARCOM`(la ligne de commande) est affichée.

```
env = environnement (JARCOMSTR = "JARchiving SOURCES $ en $ TARGET")
```

JARFLAGS

Options générales transmises à l'outil archive Java. Par défaut , ce paramètre est réglé `cf` pour créer le nécessaire **pot** fichier.

Options générales transmises à l'outil archive Java. Par défaut , ce paramètre est réglé `cf` pour créer le nécessaire **pot** fichier.

JARSUFFIX

Le suffixe pour les archives Java: `.jar` par défaut.

Le suffixe pour les archives Java: `.jar` par défaut.

JAVABOOTCLASSPATH

Indique la liste des répertoires qui seront ajoutés à la javac ligne de commande via l' `-bootclasspath`option. Les noms de répertoires individuels seront séparés par le chemin caractère distinct du système d'exploitation (`:`sous UNIX / Linux / POSIX, `;`sous Windows).

JAVAC

Le compilateur Java.

JAVACCOM

La ligne de commande utilisée pour compiler une arborescence de répertoires contenant des fichiers source Java correspondant à des fichiers de classe Java. Toutes les options spécifiées dans la `$JAVACFLAGS`variable de construction sont inclus dans cette ligne de commande.

JAVACCOMSTR

La chaîne affichée lors de la compilation d' une arborescence de fichiers source Java correspondant à des fichiers de classe Java. Si ce n'est pas défini, $JAVACCOMest affiché (la ligne de commande).

```
env = environnement (JAVACCOMSTR = "fichiers classe $ Compiler à partir de $ SOURCES CIBLES")
```

JAVACFLAGS

Options générales qui sont passés au compilateur Java.

JAVACLASSDIR

Le répertoire dans lequel les fichiers de classe Java peuvent être trouvés. Ceci est dépouillé depuis le début de tous les noms de fichiers .class Java fournis au JavaH constructeur.

JAVACLASSPATH

Indique la liste des répertoires qui seront à la recherche de Java.classfichier. Les répertoires dans cette liste seront ajoutés auxjavac et javah lignes de commande via l' -classpathoption. Les noms de répertoires individuels seront séparés par le chemin caractère distinct du système d'exploitation (:sous UNIX / Linux / POSIX, ;sous Windows).

Notez que cela ajoute actuellement tout le répertoire spécifié par l' -classpathoption. SCons ne recherche pas les$JAVACLASSPATHrépertoires pour dépendance .classfichiers.

JAVACLASSSUFFIX

Le suffixe pour les fichiers de classes Java; .class par défaut.

javaH

Le générateur Java pour les fichiers d'en-tête C et le talon.

JAVAHCOM

La ligne de commande utilisée pour générer tête C et les fichiers stub des classes Java. Toutes les options spécifiées dans la $JAVAHFLAGSvariable de construction sont inclus dans cette ligne de commande.

JAVAHCOMSTR

La chaîne affichée lorsque les fichiers d' en- tête C et stub sont générés à partir des classes Java. Si ce n'est pas défini, $JAVAHCOMest affiché (la ligne de commande).

```
env = environnement (JAVAHCOMSTR = "fichier en-tête / stub Génération (s) $ $ SOURCES CIBLES de")
```

JAVAHFLAGS

Options générales passées à l'en-tête C et générateur de fichiers stub pour les classes Java.

JAVASOURCEPATH

Indique la liste des répertoires qui seront à la recherche de l' entrée .javafichier. Les répertoires dans cette liste seront ajoutés à la javac ligne de commande via l' -sourcepathoption. Les noms de répertoires individuels seront séparés par le chemin caractère distinct du système d'exploitation (:sous UNIX / Linux / POSIX,;sous Windows).

Notez que cela ajoute actuellement tout le répertoire spécifié par l' -sourcepathoption. SCons ne recherche pas les$JAVASOURCEPATHrépertoires pour dépendance .javafichiers.

JAVASUFFIX

Le suffixe pour les fichiers Java; .java par défaut.

JAVAVERSION

Indique la version Java utilisée par le Javaconstructeur. C'est *pas* actuellement utilisé pour sélectionner une version du compilateur Java par rapport à une autre. , Vous devriez plutôt définir cette option pour spécifier la version de Java pris en charge par votre javac compilateur. La valeur par défaut est 1.4.

Ceci est parfois nécessaire parce que Java 1.5 a changé les noms de fichiers qui sont créés pour les classes internes anonymes imbriquées, ce qui peut provoquer un décalage avec les fichiers qui SCons attend sera généré par le javac compilateur. Réglage $JAVAVERSIONà 1.5 (ou 1.6, selon le cas) peut faire SCons réaliser qu'une version Java 1.5 ou 1.6 est en fait à jour.

LATEX

La structure formatter et typographe LATEX.

LATEXCOM

La ligne de commande utilisée pour appeler le formatter structuré de LaTeX et typographe.

LATEXCOMSTR

La chaîne affichée lorsque vous appelez le formatter structuré de LaTeX et typographe. Si ce n'est pas défini, $LATEXCOMest affiché (la ligne de commande).

```
env = environnement (LATEXCOMSTR = "target $ Bâtiment d'entrée LATEX SOURCES $")
```

LATEXFLAGS

Options générales transmises à la structure de formatter LaTeX et typographe.

LATEXRETRIES

Le nombre maximum de fois que LATEX sera ré-exécuter si le `.log` produit par la $LATEXCOMcommande indique qu'il ya des références non définies. La valeur par défaut est d'essayer de résoudre les références non définies par LATEX réexécution jusqu'à trois fois.

LATEXSUFFIXES

La liste des suffixes des fichiers qui seront scannés pour les dépendances implicites LaTeX (`\include`ou `\import`fichiers). La liste par défaut est le suivant :

```
[ ".tex", ".ltx", ".latex"]
```

LDMODULE

Le segment de liaison pour la construction de modules chargeables. Par défaut, c'est le même que $SHLINK.

LDMODULECOM

La ligne de commande pour la construction de modules chargeables. Sous Mac OS X, celui - ci utilise les $LDMODULE,$LDMODULEFLAGSet les $FRAMEWORKSFLAGSvariables. Sur d' autres systèmes, c'est le même que $SHLINK.

LDMODULECOMSTR

La chaîne affichée lors de la construction des modules chargeables. Si ce n'est pas défini, $LDMODULECOMest affiché (la ligne de commande).

LDMODULEFLAGS

options utilisateur général transmises à l'éditeur de liens pour la construction de modules chargeables.

LDMODULENOVERSIONSYMLINKS

Charge le LoadableModuleconstructeur de ne pas créer automatiquement des liens symboliques pour les modules versionnés. Par défaut`$SHLIBNOVERSIONSYMLINKS`

LDMODULEPREFIX

Le préfixe utilisé pour les noms de fichiers du module chargeables. Sous Mac OS X, c'est nulle; sur d' autres systèmes, c'est le même que $SHLIBPREFIX.

_LDMODULESONAME

Une macro qui génère automatiquement SONAME du module chargeable établi sur base de $, LDMODULEVERSION de $ et LDMODULESUFFIX $. Utilisé par le LoadableModuleconstructeur lorsque l'outil de liaison prend en charge SONAME (par exemple gnulink).

LDMODULESUFFIX

Le suffixe utilisé pour les noms de fichiers du module chargeables. Sous Mac OS X, c'est nulle; sur d'autres systèmes, c'est le même que $ SHLIBSUFFIX.

LDMODULEVERSION

Lorsque cette variable de construction est définie, un module chargeable versionné est créé par le LoadableModuleconstructeur. Ceci active le $_LDMODULEVERSIONFLAGSet modifie ainsi le $LDMODULECOMcas échéant, ajoute le numéro de version au nom de la bibliothèque, et crée les liens symboliques nécessaires. $LDMODULEVERSIONversions devraient exister dans le même format que $SHLIBVERSION.

LDMODULEVERSIONFLAGS

Drapeaux supplémentaires ajoutés au $LDMODULECOMmoment de la construction versionné LoadableModule. Ces drapeaux ne sont utilisés que lorsque $LDMODULEVERSIONest réglé.

_LDMODULEVERSIONFLAGS

Cette macro introduit automatiquement des drapeaux supplémentaires à $LDMODULECOMlors de la construction versionné LoadableModule(qui est quand $LDMODULEVERSIONest réglé). `_LDMODULEVERSIONFLAGS` ajoute habituellement $SHLIBVERSIONFLAGSet quelques options supplémentaires générées dynamiquement (par exemple `-Wl,-soname=$_LDMODULESONAME`). Il est utilisé par des modules chargeables simples (sans version).

LEX

Le générateur d'analyseur lexical.

LEXCOM

La ligne de commande utilisé pour appeler le générateur d'analyseur lexical pour générer un fichier source.

LEXCOMSTR

La chaîne affichée lors de la génération d' un fichier source à l' aide du générateur d'analyseur lexical. Si ce n'est pas défini, $LEXCOMest affiché (la ligne de commande).

```
env = environnement (LEXCOMSTR = "Lex'ing $ TARGET de $ SOURCES")
```

LEXFLAGS

Options générales transmises au générateur d'analyseur lexical.

_LIBDIRFLAGS

Une variable de construction généré automatiquement contenant les options de ligne de commande éditeur de liens pour spécifier les répertoires à rechercher bibliothèque. La valeur de la `$_LIBDIRFLAGS`création en annexant `$LIBDIRPREFIX`et `$LIBDIRSUFFIX`au début et à la fin de chaque répertoire `$LIBPATH`.

LIBDIRPREFIX

Le préfixe utilisé pour spécifier un répertoire de bibliothèque sur la ligne de commande de liaison. Ce sera ajouté au début de chaque répertoire dans la `$LIBPATH`variable de construction lorsque la `$_LIBDIRFLAGS`variable est générée automatiquement.

LIBDIRSUFFIX

Le suffixe utilisé pour spécifier un répertoire de bibliothèque sur la ligne de commande de liaison. Ce sera ajouté à la fin de chaque répertoire dans la `$LIBPATH`variable de construction lorsque la `$_LIBDIRFLAGS`variable est générée automatiquement.

LIBEMITTER

FAIRE

_LIBFLAGS

Une variable de construction généré automatiquement contenant les options de ligne de commande spécifiant l' éditeur de liens pour les bibliothèques à être liée à la cible résultante. La valeur de `$_LIBFLAGS`est créée en ajoutant `$LIBLINKPREFIX`et `$LIBLINKSUFFIX`au début et à la fin de chaque nom de fichier dans `$LIBS`.

LIBLINKPREFIX

Le préfixe utilisé pour spécifier une bibliothèque de lien sur la ligne de commande de liaison. Ce sera ajouté au début de chaque bibliothèque dans la `$LIBS`variable de construction lorsque la `$_LIBFLAGS`variable est générée automatiquement.

LIBLINKSUFFIX

Le suffixe utilisé pour spécifier une bibliothèque de lien sur la ligne de commande de liaison. Ce sera ajouté à la fin de chaque bibliothèque dans la `$LIBS`variable de construction lorsque la `$_LIBFLAGS`variable est générée automatiquement.

LIBPATH

La liste des répertoires qui seront à la recherche de bibliothèques. Le scanner de dépendance implicite va rechercher ces répertoires pour inclure des fichiers. Ne pas mettre explicitement inclure des arguments d'annuaire dans `$LINKFLAGS`ou `$SHLINKFLAGS` parce que le résultat sera non-portables et les répertoires ne seront pas recherchés par le scanner de dépendance. Remarque: les noms de répertoire dans LIBPATH seront examinés en place par rapport au répertoire SConscript quand ils sont utilisés dans une commande. Pour forcer `scons` à Recherch un répertoire relatif à la racine de l'utilisation de l' arbre source `#`:

```
env = environnement (LIBPATH = '# / libs)
```

Le look-up de répertoire peut également être configurée en utilisant la `Dir`() fonction:

```
libs = DIR ( '') libs
env = environnement (LIBPATH = libs)
```

La liste d'annuaire sera ajouté pour commander les lignes à travers la généré automatiquement `$_LIBDIRFLAGS` variable de construction, qui est construit en ajoutant les valeurs de la`$LIBDIRPREFIX`et les `$LIBDIRSUFFIX` variables de construction pour le début et la fin de chaque répertoire `$LIBPATH`. Toutes les lignes de commande vous définissez qui ont besoin la liste des répertoires LIBPATH doit inclure `$_LIBDIRFLAGS`:

```
env = environnement (LINKCOM = "my_linker $ _LIBDIRFLAGS $ _LIBFLAGS -o $ CIBLE SOURCE DE $")
```

LIBPREFIX

Le préfixe utilisé pour (statique) les noms de fichiers de bibliothèque. Une valeur par défaut est définie pour chaque plate-forme (posix, win32, OS2, etc.), mais la valeur est remplacée par des outils individuels (ar, mslib, sgiar, Sunar, TLIB, etc.) afin de refléter les noms des bibliothèques qu'ils créent .

LIBPREFIXES

Une liste de tous les préfixes juridiques pour les noms de fichiers bibliothèque. Lors de la recherche des dépendances de bibliothèque, SCons va rechercher les fichiers avec ces préfixes, le nom de la bibliothèque de base et suffixes dans la `$LIBSUFFIXES`liste.

LIBS

Une liste d'une ou plusieurs bibliothèques qui seront liés à des programmes exécutables créés par cet environnement.

La liste de la bibliothèque sera ajoutée à commander les lignes à travers la généré automatiquement `$_LIBFLAGS` variable de construction, qui est construit en ajoutant les valeurs de la`$LIBLINKPREFIX`et les `$LIBLINKSUFFIX` variables de construction pour le début et la fin de chaque nom de fichier dans `$LIBS`. Toutes les lignes de commande vous définissez qui ont besoin la liste des bibliothèques LIBS devrait inclure `$_LIBFLAGS`:

```
env = environnement (LINKCOM = "my_linker $ _LIBDIRFLAGS $ _LIBFLAGS -o $ CIBLE SOURCE DE $")
```

Si vous ajoutez un objet de fichier à la `$LIBS` liste, le nom de ce fichier sera ajouté à `$_LIBFLAGS`, et donc la ligne de liaison, comme cela est, sans `$LIBLINKPREFIX` ou `$LIBLINKSUFFIX`. Par exemple:

```
env.Append (LIBS = fichier ( '/ tmp / mylib.so'))
```

Dans tous les cas, scons ajoutera les dépendances du programme exécutable à toutes les bibliothèques dans cette liste.

LIBSUFFIX

Le suffixe utilisé pour (statique) les noms de fichiers de bibliothèque. Une valeur par défaut est définie pour chaque plate-forme (posix, win32, OS2, etc.), mais la valeur est remplacée par des outils individuels (ar, mslib, sgiar, Sunar, TLIB, etc.) afin de refléter les noms des bibliothèques qu'ils créent .

LIBSUFFIXES

Une liste de tous les suffixes juridiques pour les noms de fichiers bibliothèque. Lors de la recherche des dépendances de bibliothèque, SCons recherchera des fichiers avec des préfixes, dans la `$LIBPREFIXES`liste, le nom de la bibliothèque de base, et ces suffixes.

LICENCE

Le nom abrégé de la licence en vertu de laquelle ce projet est libéré (gpl, LGPL, bsd, etc.). Voir http://www.opensource.org/licenses/alphabetical pour une liste de noms de licence.

LINESEPARATOR

Le séparateur utilisé par les `Substfile`et `Textfile`constructeurs. Cette valeur est utilisée entre les sources lors de la construction de la cible. Valeur par défaut est le séparateur courant de ligne du système.

LINGUAS_FILE

Le `$LINGUAS_FILE`définit fichier (s) contenant la liste des linguas supplémentaires à traiter par `POInit`, `POUpdate`ou `MOFiles`adjuvants. Il affecte également le `Translate`constructeur. Si la variable contient une chaîne, il définit le nom du fichier de la liste. Le `$LINGUAS_FILE`peut - être une liste de noms de fichiers ainsi. Si `$LINGUAS_FILE`est réglé sur `True`(ou valeur non nulle numérique), la liste sera lu à partir du fichier par défaut nommé`LINGUAS`.

LIEN

L'éditeur de liens.

LINKCOM

La ligne de commande utilisée pour lier des fichiers d'objets dans un fichier exécutable.

LINKCOMSTR

La chaîne affichée lorsque les fichiers d'objets sont liés dans un fichier exécutable. Si ce n'est pas défini, `$LINKCOM`est affiché (la ligne de commande).

```
env = environnement (LINKCOMSTR = "Lier $ TARGET")
```

LINKFLAGS

Options utilisateur général transmises à l'éditeur de liens. Notez que cette variable ne doit *pas* contenir `-l` (ou similaire) des options de liaison avec les bibliothèques énumérées dans `$LIBS`, ni `-L` (ou similaire) Options de chemin de recherche de bibliothèque qui SCons génère automatiquement à partir `$LIBPATH`. Voir `$_LIBFLAGS` ci - dessus, pour la variable qui élargit les options bibliothèque de liens, et au- `$_LIBDIRFLAGS` dessus, pour la variable qui élargit les options de chemin de recherche de bibliothèque.

M4

La M4 préprocesseur macro.

M4COM

La ligne de commande utilisée pour transmettre des fichiers via la M4 préprocesseur macro.

M4COMSTR

La chaîne affichée lorsqu'un fichier est passé à travers la M4 préprocesseur macro. Si ce n'est pas défini, `$M4COM`est affiché (la ligne de commande).

M4FLAGS

Options générales passées à la M4 préprocesseur macro.

makeindex

Le générateur de makeindex pour le formatter TeX et typographe et la structure de formatter LaTeX et typographe.

MAKEINDEXCOM

La ligne de commande utilisée pour appeler le générateur de makeindex pour le formatter TeX et typographe et la structure de formatter LaTeX et typographe.

MAKEINDEXCOMSTR

La chaîne affichée lorsque vous appelez le générateur de makeindex pour le formatter TeX et typographe et la structure de formatter LaTeX et typographe. Si ce n'est pas défini, `$MAKEINDEXCOM`est affiché (la ligne de commande).

MAKEINDEXFLAGS

Options générales transmises au générateur de makeindex pour le formatter TeX et typographe et la structure de formatter LaTeX et typographe.

MAXLINELENGTH

Le nombre maximal de caractères autorisé sur une ligne de commande externe. Sur les systèmes Win32, lignes de liaison plus longue que ce nombre de caractères sont liés par un nom de fichier temporaire.

MIDL

Le compilateur Microsoft IDL.

MIDLCOM

La ligne de commande utilisée pour transmettre des fichiers au compilateur Microsoft IDL.

MIDLCOMSTR

La chaîne affichée lorsque le copmiler Microsoft IDL est appelé. Si ce n'est pas défini, `$MIDLCOM`est affiché (la ligne de commande).

MIDLFLAGS

Options générales transmises au compilateur Microsoft IDL.

MOSUFFIX

Suffixe utilisé pour les `MO`fichiers (par défaut: `'.mo'`). Voir `msgfmt`outil et `MOFiles`constructeur.

MSGFMT

Chemin absolu **msgfmt (1)** binaire, trouvé par `Detect()`. Voir `msgfmt`outil et `MOFiles`constructeur.

MSGFMTCOM

Ligne de commande pour exécuter **msgfmt (1)** programme. Voir `msgfmt`outil et `MOFiles`constructeur.

MSGFMTCOMSTR

Chaîne à afficher lorsque **msgfmt (1)** est invoquée (par défaut: `''`, ce qui signifie que `` print `$MSGFMTCOM`"). Voir `msgfmt`outil et `MOFiles`constructeur.

MSGFMTFLAGS

Drapeaux supplémentaires **msgfmt (1)** . Voir `msgfmt`outil et `MOFiles`constructeur.

msginit

Path to **msginit (1)** programme (trouvé via `Detect()`). Voir `msginit`outil et `POInit`constructeur.

MSGINITCOM

Ligne de commande pour exécuter **msginit (1)** programme. Voir `msginit`outil et `POInit`constructeur.

MSGINITCOMSTR

Chaîne à afficher lorsque **msginit (1)** est invoquée (par défaut: `''`, ce qui signifie que `` print `$MSGINITCOM`"). Voir `msginit`outil et `POInit`constructeur.

MSGINITFLAGS

Liste des drapeaux supplémentaires **msginit (1)** (par défaut:`[]`). Voir `msginit`outil et `POInit`constructeur.

_MSGINITLOCALE

`Macro " interne`. Calcule locale nom (langue) basé sur le nom de fichier cible (par défaut: `'${TARGET.filebase}'`).

Voir `msginit`outil et `POInit`constructeur.

msgmerge

Chemin absolu **msgmerge (1)** binaire comme l'a constaté`Detect()`. Voir `msgmerge`outil et `POUpdate`constructeur.

MSGMERGECOM

Ligne de commande pour exécuter **msgmerge (1)** commande. Voir `msgmerge`outil et `POUpdate`constructeur.

MSGMERGECOMSTR

Chaîne à afficher lorsque **msgmerge (1)** est invoquée (par défaut: `''`, ce qui signifie que `` print `$MSGMERGECOM`"). Voir `msgmerge`outil et `POUpdate`constructeur.

MSGMERGEFLAGS

Drapeaux supplémentaires **msgmerge (1)** commande. Voir `msgmerge`outil et `POUpdate`constructeur.

MSSDK_DIR

Le répertoire contenant le kit de développement Microsoft (soit la plate-forme SDK ou Windows SDK) à utiliser pour la compilation.

MSSDK_VERSION

La chaîne de version du SDK Microsoft (soit la plate - forme SDK ou Windows SDK) à utiliser pour la compilation. Versions prises en charge `6.1`, `6.0A`, `6.0`, `2003R2` et `2003R1`.

MSVC_BATCH

Lorsqu'il est réglé sur une valeur réelle, précise que SCons doit la compilation par lots de fichiers objet lorsque vous appelez le Microsoft Visual C / C ++. Toutes les compilations de fichiers source du même répertoire source qui génèrent des fichiers cibles dans un même répertoire de sortie et ont été configurés dans SCons en utilisant le même environnement de construction seront construits dans un seul appel au compilateur. Seuls les fichiers sources qui ont changé depuis leurs fichiers objets ont été construits seront transmis à chaque invocation du

compilateur (via la $CHANGED_SOURCESvariable de construction). Toute compilation où l'objet nom de base de fichier (cible) (moins le .obj) ne correspond pas au fichier source nom de base seront compilés séparément.

MSVC_USE_SCRIPT

Utilisez un script batch pour configurer le compilateur Microsoft Visual Studio

$MSVC_USE_SCRIPToverrides $MSVC_VERSIONet $TARGET_ARCH. Si elle est définie sur le nom d'un Visual Studio fichier .bat (par exemple vcvars.bat), SCons courront ce fichier de chauve - souris et extraire les variables pertinentes du résultat (généralement% INCLUDE%,% LIB% et% PATH%). Réglage MSVC_USE_SCRIPT sur Aucun court - circuite entièrement l'autodétection Visual Studio; utiliser si vous exécutez SCons dans une fenêtre Visual Studio cmd et l' importation des variables d'environnement du shell.

MSVC_VERSION

Définit la version préférée de Microsoft Visual C / C ++ à utiliser.

Si $MSVC_VERSIONnon défini, SCons sera (par défaut) , sélectionnez la dernière version de Visual C / C ++ installé sur votre système. Si la version spécifiée est pas installé, l' initialisation de l' outil échouera. Cette variable doit être passé comme argument au constructeur de l' environnement (); il a plus tard la mise en aucun effet.

Les valeurs valides pour Windows sont 12.0, 12.0Exp, 11.0,11.0Exp, 10.0, 10.0Exp, 9.0, 9.0Exp, 8.0, 8.0Exp, 7.1, 7.0et 6.0. Versions se terminant en Expréférer aux éditions « Express » ou « Express » pour le bureau.

VSM

Lorsque les outils Microsoft Visual Studio sont initialisés, ils ont créé ce dictionnaire avec les touches suivantes:

VERSION

la version du VSM utilisé (peut être réglé via $MSVS_VERSION)

VERSIONS

les versions disponibles de MSVS installés

VCINSTALLDIR

répertoire d'installation de Visual C ++

VSINSTALLDIR

répertoire d'installation de Visual Studio

FRAMEWORKDIR

répertoire d'installation du framework .NET

FRAMEWORKVERSIONS

liste des versions installées du framework .NET, triée récent au plus ancien.

FRAMEWORKVERSION

dernière version installée du framework .NET

FRAMEWORKSDKDIR

emplacement d'installation du SDK .NET.

PLATFORMSDKDIR

emplacement d'installation du SDK de la plate-forme.

PLATFORMSDK_MODULES

dictionnaire des modules SDK de la plate-forme installée, où les clés de dictionnaire sont des mots clés pour les différents modules, et les valeurs sont 2-tuples où le premier est la date de sortie, et le second est le numéro de version.

Si une valeur est pas définie, il n'a pas été disponible dans le Registre.

MSVS_ARCH

Définit l'architecture pour laquelle le projet généré (s) doit construire.

La valeur par défaut est x86. amd64est également pris en charge par SCons pour certaines versions de Visual Studio. Essayer de mettre $MSVS_ARCHà une architecture qui n'est pas pris en charge pour une version donnée Visual Studio génère une erreur.

MSVS_PROJECT_GUID

La chaîne placée dans un fichier de projet Microsoft Visual Studio généré en tant que la valeur de l' ProjectGUIDattribut. Il n'y a aucune valeur par défaut. Si non défini, un nouveau GUID est généré.

MSVS_SCC_AUX_PATH

Le nom de chemin placé dans un fichier de projet Microsoft Visual Studio généré la valeur de l' SccAuxPathattribut si laMSVS_SCC_PROVIDERvariable de la construction est également définie. Il n'y a aucune valeur par défaut.

MSVS_SCC_CONNECTION_ROOT

Le chemin racine des projets dans votre espace de travail du CCN, à savoir le chemin dans lequel tous les fichiers de projet et solutions seront générés. Il est utilisé comme chemin de référence à partir de laquelle les chemins relatifs du projet Visual Studio Microsoft et les fichiers générés sont calculés solution. Le chemin du fichier de projet relatif est placé en tant que valeur de l' `SccLocalPath`attribut du fichier de projet et les valeurs de la`SccProjectFilePathRelativizedFromConnection[i]`(où [i] compris entre 0 et le nombre de projets dans la solution) des attributs de la `GlobalSection(SourceCodeControl)`section de la solution Microsoft Visual Studio fichier. De même , le chemin de fichier relative en solution est placée en tant que les valeurs de la `SccLocalPath[i]`(où [i] compris entre 0 et le nombre de projets dans la solution) des attributs de la`GlobalSection(SourceCodeControl)`section du fichier de solution Microsoft Visual Studio. Il est utilisé que si la`MSVS_SCC_PROVIDER`variable de la construction est également définie. La valeur par défaut est le répertoire de travail courant.

MSVS_SCC_PROJECT_NAME

Le nom du projet placé dans un fichier de projet Microsoft Visual Studio généré la valeur de l' `SccProjectName`attribut si la`MSVS_SCC_PROVIDER`variable de la construction est également définie. Dans ce cas , la chaîne est également placé dans l' `SccProjectName0` attribut de la `GlobalSection(SourceCodeControl)`section du fichier de solution Microsoft Visual Studio. Il n'y a aucune valeur par défaut.

MSVS_SCC_PROVIDER

La chaîne placée dans un fichier de projet Microsoft Visual Studio généré en tant que la valeur de l' `SccProvider`attribut. La chaîne est également placé dans l' `SccProvider0`attribut de la`GlobalSection(SourceCodeControl)`section du fichier de solution Microsoft Visual Studio. Il n'y a aucune valeur par défaut.

MSVS_VERSION

Définit la version préférée de Microsoft Visual Studio à utiliser.

Si `$MSVS_VERSION`non défini, SCons sera (par défaut) , sélectionnez la dernière version de Visual Studio installé sur votre système. Donc, si vous avez la version 6 et la version 7 (VSM .NET) installé, il préférera la version 7. Vous pouvez remplacer en spécifiant la `MSVS_VERSION`variable dans l'initialisation de l' environnement, la mise à la version appropriée (« 6.0 » ou « 7.0 » , par exemple). Si la version spécifiée est pas installé, l' initialisation de l' outil échouera.

Ceci est obsolète: utiliser à la `$MSVC_VERSION`place. Si `$MSVS_VERSION`est réglé et `$MSVC_VERSION`n'est pas, `$MSVC_VERSION`sera automatiquement réglé `$MSVS_VERSION`. Si les deux sont mis à des valeurs différentes, scons soulèvera une erreur.

MSVSBUILDCOM

La ligne de commande de construction placée dans un fichier de projet Microsoft Visual Studio généré. La valeur par défaut est d'avoir Visual Studio avec des SCons invoquer des objectifs de construction spécifiés.

MSVSCLEANCOM

La ligne de commande propre placée dans un fichier de projet Microsoft Visual Studio généré. La valeur par défaut est d'avoir Visual Studio avec l'SCons appeler option -c pour éliminer toutes les cibles spécifiées.

MSVSENCODING

La chaîne de codage placé dans un fichier de projet Microsoft Visual Studio généré. La valeur par défaut est encode `Windows-1252`.

MSVSPROJECTCOM

L'action utilisée pour générer des fichiers de projet Microsoft Visual Studio.

MSVSPROJECTSUFFIX

Le suffixe utilisé pour les fichiers Microsoft projet Visual Studio (DSP). La valeur par défaut est `.vcproj` lorsque vous utilisez la version Visual Studio 7.x (.NET) ou version ultérieure, et `.dsp`lors de l' utilisation des versions antérieures de Visual Studio.

MSVSREBUILDCOM

La ligne de commande de recréation placée dans un fichier de projet Microsoft Visual Studio généré. La valeur par défaut est d'avoir Visual Studio avec SCons appeler les cibles spécifiées reconstruction.

MSVSSCONS

Les SCons utilisés dans les fichiers générés de projet Microsoft Visual Studio. La valeur par défaut est la version de SCons utilisé pour générer le fichier de projet.

MSVSSCONSCOM

La commande de la valeur par défaut utilisée dans les fichiers générés de projet Microsoft Visual Studio.

MSVSSCONSCRIPT

Le fichier sconscript (qui est, `SConstruct`ou `SConscript`fichier) qui sera appelé par les fichiers de projet Visual Studio (à travers la `$MSVSSCONSCOM`variable). La valeur par défaut est le même fichier sconscript qui contient l'appel à `MSVSProject` construire le fichier de projet.

MSVSSCONSFLAGS

Les drapeaux SCons utilisés dans les fichiers générés de projet Microsoft Visual Studio.

MSVSSOLUTIONCOM

L'action utilisée pour générer des fichiers de solution Microsoft Visual Studio.

MSVSSOLUTIONSUFFIX

Le suffixe utilisé pour les fichiers Microsoft solution Visual Studio (DSW). La valeur par défaut est `.sln`lorsque vous utilisez la version Visual Studio 7.x (.NET), et `.dsw`lors de l' utilisation des versions antérieures de Visual Studio.

MT

Le programme utilisé sur les systèmes Windows pour intégrer dans manifeste DLL et EXE. Voir aussi $WINDOWS_EMBED_MANIFEST.

MTEXECOM

La ligne de commande Windows utilisé pour intégrer manifeste dans executables. Voir aussi $MTSHLIBCOM.

MTFLAGS

Les drapeaux transmis au $MTprogramme d'intégration manifeste (Windows uniquement).

MTSHLIBCOM

La ligne de commande Windows utilisé pour intégrer dans des bibliothèques partagées manifeste (DLL). Voir aussi $MTEXECOM.

MWCW_VERSION

Le numéro de version du compilateur C MetroWerks CodeWarrior à utiliser.

MWCW_VERSIONS

Une liste des versions installées du compilateur C MetroWerks CodeWarrior sur ce système.

PRÉNOM

Specfies le nom du projet pour emballer.

no_import_lib

Lorsqu'il est réglé à la non-zéro, supprime la création d'un ordinateur Windows statique import lib correspondant par leSharedLibrary constructeur lorsqu'il est utilisé avec MinGW, Microsoft Visual Studio ou Metrowerks. Cela supprime également la création d'un fichier d'exportation (.exp) lorsque vous utilisez Microsoft Visual Studio.

OBJPREFIX

Le préfixe utilisé pour (statique) les noms de fichiers d'objets.

OBJSUFFIX

Le suffixe utilisé pour (statique) les noms de fichiers d'objets.

P4

L'exécutable Perforce.

P4COM

La ligne de commande utilisée pour récupérer des fichiers source de Perforce.

P4COMSTR

La chaîne affichée lors de la récupération d' un fichier source à partir de Perforce. Si ce n'est pas défini, $P4COMest affiché (la ligne de commande).

P4FLAGS

Options générales qui sont passés à Perforce.

PACKAGEROOT

Indique le répertoire dans lequel tous les fichiers dans l'archive résultant seront placés le cas échéant. La valeur par défaut est "$ NAME- $ VERSION".

TYPE D'EMBALLAGE

Sélectionne le type de package à construire. À l'heure actuelle ils sont disponibles:

* Msi - Microsoft Installer * rpm - Redhat Package Manager * ipkg - Itsy système de gestion des paquets * tarbz2 - goudron comprimé * targz - tar compressé * zip - fichier zip * src_tarbz2 - source de goudron comprimé * src_targz - source de goudron comprimé * src_zip - fichier zip la source

Cela peut être modifié avec l'option de ligne de commande « package_type ».

PackageVersion

La version du paquet (pas le projet sous-jacent). Ceci est actuellement utilisé par le régime emballeur et doit tenir compte des changements dans l'emballage, pas le code du projet sous-jacent lui-même.

PCH

L'en-tête de Visual C ++ précompilé Microsoft qui sera utilisé lors de la compilation des fichiers d'objets. Cette variable est ignorée par d'autres outils que Microsoft Visual C ++. Lorsque cette variable est définie SCons ajoutera des options à la ligne de commande du compilateur pour l'amener à utiliser l'en-tête précompilé, et sera également mis en place les dépendances du fichier PCH. Exemple:

```
env [PCH '] = 'StdAfx.pch'
```

PCHCOM

La ligne de commande utilisé par le PCH générateur pour produite un en- tête précompilés.

PCHCOMSTR

La chaîne affichée lors de la génération d' un en- tête précompilés. Si ce n'est pas défini, $PCHCOMest affiché (la ligne de commande).

PCHPDBFLAGS

Une variable de construction qui, lorsqu'il est dilaté, ajoute le /yDdrapeau à la ligne de commande uniquement si la $PDBvariable de construction est définie.

PCHSTOP

Cette variable indique la quantité d'un fichier source est précompilés. Cette variable est ignorée par d'autres outils que Microsoft Visual C ++, ou lorsque la variable PCH n'est pas utilisé. Lorsque cette variable est le définir doit être une chaîne qui est le nom de l'en-tête qui est inclus à la fin de la partie précompilé des fichiers source, ou la chaîne vide si est utilisé la construction « #pragma hrdstop »:

```
env [ 'PCHSTOP'] = 'StdAfx.h'
```

APB

Microsoft Visual C ++ fichier PDB qui stockera les informations de débogage pour les fichiers d'objets, des bibliothèques partagées et des programmes. Cette variable est ignorée par d'autres outils que Microsoft Visual C ++. Lorsque cette variable est définie SCons ajoutera des options au compilateur et ligne de commande éditeur de liens pour les amener à générer des informations de débogage externe, et sera également mis en place les dépendances pour le fichier PDB. Exemple:

```
env [ 'PDB'] = 'hello.pdb'
```

Visual C ++ commutateur de compilateur que SCons utilise par défaut pour générer des informations PDB est /Z7. Cela fonctionne correctement avec parallèle (-j) construit parce qu'il intègre les informations de débogage dans les fichiers objets intermédiaires, par opposition à partager un seul fichier PDB entre plusieurs fichiers objet. Ceci est aussi la seule façon d'obtenir des informations de débogage intégré dans une bibliothèque statique. En utilisant le /Zipeut au contraire obtenir de meilleures performances en temps de liaison, bien que des constructions parallèles ne fonctionneront plus. Vous pouvez générer des fichiers PDB avec le /Zi commutateur en remplaçant la valeur par défaut $CCPDBFLAGSvariable; voir l'entrée de cette variable pour des exemples spécifiques.

PDFCOM

Un synonyme dépréciée pour $DVIPDFCOM.

PDFLATEX

Le pdflatex utilitaire.

PDFLATEXCOM

La ligne de commande utilisé pour appeler le pdflatex utilitaire.

PDFLATEXCOMSTR

La chaîne affichée lorsque vous appelez l' pdflatex utilitaire. Si ce n'est pas défini, $PDFLATEXCOMest affiché (la ligne de commande).

```
env = environnement (PDFLATEX; COMSTR = "Building $ TARGET d'entrée de $ SOURCES LATEX")
```

PDFLATEXFLAGS

Options générales transmises à l' pdflatex utilitaire.

PDFPREFIX

Le préfixe utilisé pour les noms de fichiers PDF.

PDFSUFFIX

Le suffixe utilisé pour les noms de fichiers PDF.

pdftex

Le pdftex utilitaire.

PDFTEXCOM

La ligne de commande utilisé pour appeler le pdftex utilitaire.

PDFTEXCOMSTR

La chaîne affichée lorsque vous appelez l' pdftex utilitaire. Si ce n'est pas défini, $PDFTEXCOMest affiché (la ligne de commande).

```
env = environnement (PDFTEXCOMSTR = "du bâtiment de $ TARGET d'entrée TeX SOURCES $")
```

PDFTEXFLAGS

Options générales transmises à l' pdftex utilitaire.

pkgchk

Sur les systèmes Solaris, le programme de vérification du package qui sera utilisé (avec $PKGINFO) pour rechercher des versions installées du PRO Sun compilateur C ++. La valeur par défaut est /usr/sbin/pgkchk.

pkgInfo

Sur les systèmes Solaris, le programme d'information de package qui sera utilisé (avec `$PKGCHK`) pour rechercher des versions installées du PRO Sun compilateur C ++. La valeur par défaut est`pkginfo`.

PLATE-FORME

Le nom de la plate - forme utilisée pour créer l'environnement. Si aucune plate - forme est spécifiée lorsque l'environnement est créé, `scons` détecte automatiquement la plate - forme.

```
env = environnement (outils = [])
si env [ ''] == PLATEFORME 'Cygwin':
    Outil ( 'mingw') (env)
autre:
    Outil ( 'msvc') (env)
```

POAUTOINIT

La `$POAUTOINIT`variable si elle est définie à `True`(sur la valeur numérique non nulle), laissez l' `msginit`outil pour initialiser automatiquement *manquants* `PO` des fichiers avec **msginit (1)** . Cela vaut à la fois, `POInit`et les `POUpdate`constructeurs (et d' autres qui utilisent l' un d'eux).

POCREATE_ALIAS

Alias commun pour tous les `PO`fichiers créés avec le `POInit`constructeur (par défaut: `'po-create'`). Voir `msginit`outil et `POInit`constructeur.

POSUFFIX

Suffixe utilisé pour les `PO`fichiers (par défaut: `'.po'`) Voir `msginit`outil et `POInit`constructeur.

POTDOMAIN

Le `$POTDOMAIN`définit domaine par défaut, utilisé pour générer le `POT`nom de fichier comme `$POTDOMAIN.pot`lorsqu'aucun `POT`nom de fichier est fourni par l'utilisateur. Cela vaut pour `POTUpdate`, `POInit`et les `POUpdate`constructeurs (et les constructeurs, qui les utilisent, par exemple `Translate`). Normalement (si `$POTDOMAIN`non défini), les constructeurs utilisent `messages.pot`par défaut le `POT`nom du fichier.

POTSUFFIX

Suffixe utilisé pour les fichiers de modèle PO (par défaut: `'.pot'`). Voir `xgettext`outil et `POTUpdate`constructeur.

POTUPDATE_ALIAS

Nom de la cible commune pour tous les faux modèles PO créés avec `POUpdate`(par défaut: `'pot-update'`). Voir `xgettext`outil et `POTUpdate`constructeur.

POUPDATE_ALIAS

Alias commun pour tous les `PO`fichiers étant définis par le `POUpdate`constructeur (par défaut: `'po-update'`). Voir `msgmerge`outil et `POUpdate`constructeur.

PRINT_CMD_LINE_FUNC

Une fonction Python utilisée pour imprimer les lignes de commande quand elles sont exécutées (en supposant l' impression de commande ne soit pas désactivée par les `-q` ou `-s`options ou leurs équivalents). La fonction doit prendre quatre arguments: `s`, la commande en cours d' exécution (une chaîne),`target`la cible en cours de construction (nœud de fichier, une liste ou nom de chaîne (s)) `source`, la source (s) utilisé (nœud de fichier, une liste ou nom de chaîne (s)) et `env`, l'environnement utilisé.

La fonction doit faire l'impression elle-même. L'implémentation par défaut, utilisée si cette variable est pas définie ou est None, est:

```
def print_cmd_line (s, cible, de la source, env):
  sys.stdout.write (s + "\ n")
```

Voici un exemple d'une fonction plus intéressante:

```
def print_cmd_line (s, cible, de la source, env):
   sys.stdout.write ( "Bâtiment% s ->% s ... \ n" %
    ( 'Et' .join ([str (x) pour x dans la source]),
     'Et' .join ([str (x) pour x dans la cible])))
env = environnement (PRINT_CMD_LINE_FUNC = print_cmd_line)
env.Program ( 'foo', 'foo.c')
```

Cette impression « Construire seulement `targetname`de `sourcename`... » au lieu des commandes réelles. Une telle fonction pourrait également enregistrer les commandes réelles dans un fichier journal, par exemple.

PROGEMITTER

FAIRE

PROGPREFIX

Le préfixe utilisé pour les noms de fichiers exécutables.

PROGSUFFIX

Le suffixe utilisé pour les noms de fichiers exécutables.

PSCOM

La ligne de commande utilisée pour convertir les fichiers DVI TeX dans un fichier PostScript.

PSCOMSTR

La chaîne affichée lorsqu'un fichier DVI TeX est converti en un fichier PostScript. Si ce n'est pas défini, $PSCOMest affiché (la ligne de commande).

PSPREFIX

Le préfixe utilisé pour les noms de fichiers PostScript.

PSSUFFIX

Le préfixe utilisé pour les noms de fichiers PostScript.

QT_AUTOSCAN

Désactiver l'analyse des fichiers mocable. Utilisez le Builder Moc pour spécifier explicitement les fichiers à exécuter sur moc.

QT_BINPATH

Le chemin où les binaires qt sont installés. La valeur par défaut est « $QTDIR/ bin ».

QT_CPPPATH

Le chemin dans lequel les fichiers d' en- tête qt sont installés. La valeur par défaut est « $QTDIR/ include ». Remarque: Si vous définissez cette variable sur None, l'outil ne changera pas la $CPPPATH variable de construction.

QT_DEBUG

beaucoup Imprime de débogage des informations lors de la numérisation des fichiers moc.

QT_LIB

La valeur par défaut est « qt ». Vous pouvez définir cela 'qt-mt. Remarque: Si vous définissez cette variable sur None, l'outil ne changera pas la $LIBSvariable.

QT_LIBPATH

Le chemin où les bibliothèques qt sont installées. La valeur par défaut est « $QTDIR/ lib ». Remarque: Si vous définissez cette variable sur None, l'outil ne changera pas la $LIBPATH variable de construction.

QT_MOC

Valeur par défaut est « $QT_BINPATH/ moc ».

QT_MOCCXXPREFIX

La valeur par défaut est « ». Prefix pour les fichiers de sortie moc, lorsque la source est un fichier cxx.

QT_MOCCXXSUFFIX

La valeur par défaut est « .moc ». Suffixe pour les fichiers de sortie moc, lorsque la source est un fichier cxx.

QT_MOCFROMCXXCOM

Commande pour générer un fichier moc à partir d'un fichier cpp.

QT_MOCFROMCXXCOMSTR

La chaîne affichée lors de la génération d' un fichier moc à partir d' un fichier cpp. Si ce n'est pas défini, $QT_MOCFROMCXXCOMest affiché (la ligne de commande).

QT_MOCFROMCXXFLAGS

La valeur par défaut est « -i ». Ces drapeaux sont passés à moc, lorsque moccing un fichier C ++.

QT_MOCFROMHCOM

Commande pour générer un fichier moc à partir d'un en-tête.

QT_MOCFROMHCOMSTR

La chaîne affichée lors de la génération d' un fichier moc à partir d' un fichier cpp. Si ce n'est pas défini, $QT_MOCFROMHCOMest affiché (la ligne de commande).

QT_MOCFROMHFLAGS

La valeur par défaut est « ». Ces drapeaux sont passés à moc, lorsque moccing un fichier d'en-tête.

QT_MOCHPREFIX

La valeur par défaut est « moc_ ». Prefix pour les fichiers de sortie moc, lorsque la source est un en-tête.

QT_MOCHSUFFIX

La valeur par défaut est « $CXXFILESUFFIX». Suffixe pour les fichiers de sortie moc, lorsque la source est un en- tête.

QT_UIC

Valeur par défaut est « $QT_BINPATH/ uic ».

QT_UICCOM

Commande pour générer des fichiers d'en-tête des fichiers .ui.

QT_UICCOMSTR

La chaîne affichée lors de la génération des fichiers d' en- tête des fichiers .ui. Si ce n'est pas défini, $QT_UICCOMest affiché (la ligne de commande).

QT_UICDECLFLAGS

La valeur par défaut est « ». Ces drapeaux sont passés à uIc, lors de la création du fichier aah à partir d'un fichier .ui.

QT_UICDECLPREFIX

La valeur par défaut est « ». Préfixe pour les fichiers d'en-tête générés uic.

QT_UICDECLSUFFIX

La valeur par défaut est « .h ». Suffixe pour les fichiers d'en-tête générés uic.

QT_UICIMPLFLAGS

La valeur par défaut est « ». Ces drapeaux sont passés à uIc, lors de la création d'un fichier cxx à partir d'un fichier .ui.

QT_UICIMPLPREFIX

La valeur par défaut est « uic_ ». Préfixe pour uic fichiers générés par la mise en œuvre.

QT_UICIMPLSUFFIX

La valeur par défaut est « $CXXFILESUFFIX». Suffixe pour uic fichiers générés par la mise en œuvre.

QT_UISUFFIX

La valeur par défaut est « .ui ». Suffixe des fichiers d'entrée concepteur.

QTDIR

L'outil qt essaie de prendre ce billet depuis os.environ. Il initialise toutes les variables de construction QT_ * ci - dessous. (Notez que tous les chemins sont construits avec la méthode de os.path.join de python (), mais sont listés ici avec le séparateur « / » pour faciliter la lecture.) En outre, les variables d'environnement de la construction $CPPPATH, $LIBPATHet$LIBSpeuvent être modifiés et les variables $PROGEMITTER, $SHLIBEMITTERet $LIBEMITTER sont modifiés. Parce que est affecté lors de l' utilisation de cet outil, vous devez spécifier explicitement à la création de l' environnement l'accumulation des performances:

```
Environnement (outils = [ 'default', 'qt'])
```

L'outil prend en charge qt les opérations suivantes:

Génération automatique de fichiers moc à partir des fichiers d' en- tête. Vous ne devez pas spécifierfichiers moc explicitement, l'outil fait pour vous. Cependant, il y a quelques conditions préalables pourfaire: Votre fichier d'tête doit avoir le même Filebase que votre fichier de miseœuvre et doit rester dans le même répertoire. Il doit avoirun des suffixes .H, .hpp, .H, .hxx, .hh. Vous pouvez désactivergénération de fichiers moc automatique en réglant QT_AUTOSCAN à 0. Voir aussi la correspondante Mocméthode constructeur ().

Génération automatique de fichiers moc à partir de fichiers Cxx. Commeindiqué dans la documentation qt, inclure le fichier moc à la fin du fichier cxx. Notez que vous devez inclure le fichier, qui est généré par le $ de transformation {QT_MOCCXXPREFIX} <basename> $ {} QT_MOCCXXSUFFIX, par défaut <basename> .moc. Un avertissement est généré aprèsconstruction du fichier moc, si vous ne pas inclure le fichier correct. Si vous utilisez VariantDir, vous devrez peutêtre spécifier double = 1. Vous pouvez désactivergénération de fichiers moc automatique en réglant QT_AUTOSCAN à 0. Voir aussi la correspondante Moc méthode constructeur.

Gestion automatique des fichiers .ui. Les fichiers d'implémentation généréspartirfichiers .ui sont traitésla même que yacc ou fichiers lex. Chaque fichier .ui donné comme source de programme,bibliothèque ou SharedLibrary génère trois fichiers, le fichier de déclaration, le fichier de miseœuvre et un fichier moc. Parce qu'il ya aussitêtes générés, vous devrez peutêtre spécifier double = 1 dansappels à VariantDir. Voir aussi la correspondante Uic méthode constructeur.

RANLIB

L'indexeur d'archives.

RANLIBCOM

La ligne de commande utilisée pour indexer une archive de bibliothèque statique.

RANLIBCOMSTR

La chaîne affiche quand une archive bibliothèque statique est indexée. Si ce n'est pas défini, $RANLIBCOMest affiché (la ligne de commande).

```
env = environnement (RANLIBCOMSTR = "Indexation target $")
```

RANLIBFLAGS

Options générales passées à l'indexeur d'archives.

RC

Le compilateur de ressources utilisées pour construire un fichier de ressources Visual C ++ Microsoft.

RCCOM

La ligne de commande utilisée pour créer un fichier de ressources Visual C ++ Microsoft.

RCCOMSTR

La chaîne affichée lors de l' appel du compilateur de ressources pour construire un fichier de ressources Visual C ++ Microsoft. Si ce n'est pas défini, $RCCOMest affiché (la ligne de commande).

RCFLAGS

Les drapeaux passés au compilateur de ressources par le constructeur RES.

RCINCFLAGS

Une variable de construction généré automatiquement contenant les options de ligne de commande pour spécifier les répertoires à rechercher par le compilateur de ressources. La valeur de la $RCINCFLAGScréation en annexant $RCINCPREFIXet $RCINCSUFFIX au début et à la fin de chaque répertoire $CPPPATH.

RCINCPREFIX

Le préfixe (drapeau) utilisé pour spécifier un répertoire include sur la ligne de commande du compilateur de ressources. Ce sera ajouté au début de chaque répertoire dans la $CPPPATHvariable de construction lorsque la $RCINCFLAGSvariable est développée.

RCINCSUFFIX

Le suffixe utilisé pour spécifier un include sur la ligne de commande du compilateur de ressources. Ce sera ajouté à la fin de chaque répertoire dans la $CPPPATHvariable de construction lorsque la $RCINCFLAGSvariable est développée.

RCS

L'exécutable RCS. Notez que cette variable est pas effectivement utilisé pour la commande pour récupérer des fichiers source de RCS; voir la $RCS_CO variable de construction, ci - dessous.

RCS_CO

Le RCS « checkout » exécutable, utilisé pour récupérer des fichiers source du RCS.

RCS_COCOM

La ligne de commande utilisée pour récupérer les fichiers source (caisse) du RCS.

RCS_COCOMSTR

La chaîne affichée lors de la récupération d' un fichier source à partir de RCS. Si ce n'est pas défini, $RCS_COCOM est affiché (la ligne de commande).

RCS_COFLAGS

Les options qui sont passés à la $RCS_COcommande.

RDirs

Une fonction qui convertit une chaîne en une liste d'instances Dir en recherchant les dépôts.

regSvr

Le programme utilisé sur les systèmes Windows pour enregistrer une bibliothèque DLL nouvellement construite chaque fois que le SharedLibraryconstructeur est passé un argument mot - clé de register=1.

REGSVRCOM

La ligne de commande utilisée sur les systèmes Windows pour enregistrer une bibliothèque DLL nouvellement construite chaque fois que le SharedLibraryconstructeur est passé un argument mot - clé de register=1.

REGSVRCOMSTR

La chaîne affichée lors de l' enregistrement d' un fichier DLL nouvellement construit. Si ce n'est pas défini, $REGSVRCOMest affiché (la ligne de commande).

REGSVRFLAGS

Les drapeaux transmis au programme d'enregistrement des DLL sur les systèmes Windows quand une bibliothèque de DLL nouvellement construite est enregistrée. Par défaut, cela inclut la /s qui empêche les boîtes de dialogue de sauter et nécessitant l' attention des utilisateurs.

CRIM

Le compilateur Java stub RMI.

RMICCOM

La ligne de commande utilisée pour compiler des fichiers classe stub et le squelette des classes Java qui contiennent des implémentations RMI. Toutes les options spécifiées dans la $RMICFLAGSvariable de construction sont inclus dans cette ligne de commande.

RMICCOMSTR

La chaîne affichée lors de la compilation des fichiers classe stub et le squelette des classes Java qui contiennent des implémentations RMI. Si ce n'est pas défini, $RMICCOMest affiché (la ligne de commande).

```
env = environnement (RMICCOMSTR = "fichiers stub Génération / classe squelette $ CIBLES de $ SOURCES")
```

RMICFLAGS

Options générales passées au compilateur de Java stub RMI.

_RPATH

Une variable de construction généré automatiquement contenant les drapeaux rpath à utiliser lors de la liaison d' un programme avec des bibliothèques partagées. La valeur de la `$_RPATH`création en annexant `$RPATHPREFIX`et `$RPATHSUFFIX` au début et à la fin de chaque répertoire `$RPATH`.

RPATH

Une liste de chemins pour rechercher des bibliothèques partagées lorsque les programmes en cours d' exécution. Actuellement utilisé uniquement dans la linkers GNU (gnulink), IRIX (sgilink) et Sun (SunLink). Ignoré sur les plates - formes et toolchains qui ne supportent pas. Notez que les chemins ajoutés à RPATH ne sont pas transformées par `scons` de quelque façon: si vous voulez un chemin absolu, vous devez faire vous - même absolu.

RPATHPREFIX

Le préfixe utilisé pour spécifier un répertoire à rechercher des bibliothèques partagées lorsque les programmes en cours d' exécution. Ce sera ajouté au début de chaque répertoire dans la `$RPATH`variable de construction lorsque la `$_RPATH`variable est générée automatiquement.

RPATHSUFFIX

Le suffixe utilisé pour spécifier un répertoire à rechercher des bibliothèques partagées lorsque les programmes en cours d' exécution. Ce sera ajouté à la fin de chaque répertoire dans la `$RPATH`variable de construction lorsque la `$_RPATH`variable est générée automatiquement.

rpcgen

Le compilateur de protocole RPC.

RPCGENCLIENTFLAGS

Les options passées au compilateur de protocole RPC lors de la génération des talons de côté client. Ceux - ci sont en plus des indicateurs spécifiés dans la `$RPCGENFLAGS` variable de construction.

RPCGENFLAGS

Options générales passés au compilateur de protocole RPC.

RPCGENHEADERFLAGS

Les options passées au compilateur de protocole RPC lors de la génération d' un fichier d' en- tête. Ceux - ci sont en plus des indicateurs spécifiés dans la `$RPCGENFLAGS` variable de construction.

RPCGENSERVICEFLAGS

Les options passées au compilateur de protocole RPC lors de la génération des talons de côté serveur. Ceux - ci sont en plus des indicateurs spécifiés dans la `$RPCGENFLAGS` variable de construction.

RPCGENXDRFLAGS

Les options passées au compilateur de protocole RPC lors de la génération des routines XDR. Ceux - ci sont en plus des indicateurs spécifiés dans la `$RPCGENFLAGS` variable de construction.

LECTEURS

Une liste des scanners de dépendance implicites disponibles. Les nouveaux scanners de fichiers peuvent être ajoutés en ajoutant à cette liste, bien que l'approche plus souple est d'associer les scanners avec un constructeur spécifique. Voir les sections « Objets Builder » et « Objets du scanner » ci-dessous pour plus d'informations.

CSSC

L'exécutable CSSC.

SCCSCOM

La ligne de commande utilisée pour récupérer des fichiers sources à partir CSSC.

SCCSCOMSTR

La chaîne affichée lors de la récupération d' un fichier source à partir d' un référentiel CVS. Si ce n'est pas défini, `$SCCSCOM` est affiché (la ligne de commande).

SCCSFLAGS

Options générales qui sont passés à CSSC.

SCCSGETFLAGS

Les options qui sont passés spécifiquement à la CSSC « get » sous - commande. Cela peut être réglé, par exemple, `-e` de vérifier les fichiers modifiables à partir CSSC.

SCONS_HOME

Le chemin (en option) dans le répertoire de la bibliothèque SCons, initialisé de l'environnement extérieur. Si elle est définie, il est utilisé pour construire un chemin de recherche plus court et plus efficace dans la `$MSVSSCONS`ligne de commande exécutée à partir des fichiers de projet Microsoft Visual Studio.

SHCC

Le compilateur C utilisé pour générer des objets partagés bibliothèque.

SHCCCOM

La ligne de commande utilisée pour compiler un fichier source de C dans un fichier objet partagé bibliothèque. Toutes les options spécifiées dans le $SHCFLAGS, $SHCCFLAGSet les $CPPFLAGSvariables de construction sont inclus dans cette ligne de commande.

SHCCCOMSTR

La chaîne affiche lorsqu'un fichier source C est compilé dans un fichier objet partagé. Si ce n'est pas défini, $SHCCCOMest affiché (la ligne de commande).

```
env = environnement (SHCCCOMSTR = "objet partagé de compilation $ TARGET")
```

SHCCFLAGS

Les options qui sont transmises aux compilateurs C et C ++ pour générer des objets partagés bibliothèque.

SHCFLAGS

Les options qui sont passés au compilateur C (seulement; pas C ++) pour générer des objets partagés bibliothèques.

SHCXX

Le compilateur C utilisé pour générer des objets partagés bibliothèque.

SHCXXCOM

La ligne de commande utilisée pour compiler un fichier source en C ++ dans un fichier objet partagé bibliothèque. Toutes les options spécifiées dans les $SHCXXFLAGSet les $CPPFLAGSvariables de construction sont inclus dans cette ligne de commande.

SHCXXCOMSTR

La chaîne affichée lorsqu'un fichier source C de compilation dans un fichier objet partagé. Si ce n'est pas défini, $SHCXXCOMest affiché (la ligne de commande).

```
env = environnement (SHCXXCOMSTR = "objet partagé de compilation $ TARGET")
```

SHCXXFLAGS

Les options qui sont passés au compilateur C pour générer des objets partagés bibliothèque.

SHDC

SHDC.

SHDCOM

SHDCOM.

SHDLINK

SHDLINK.

SHDLINKCOM

SHDLINKCOM.

SHDLINKFLAGS

SHDLINKFLAGS.

COQUILLE

Une chaîne nommant le programme shell qui sera transmis à la$SPAWN fonction. Voir la $SPAWN variable de construction pour plus d' informations.

SHF03

Le compilateur Fortran 03 utilisé pour générer des objets partagés bibliothèque. Vous devez normalement régler la $SHFORTRANvariable qui spécifie le compilateur par défaut Fortran pour toutes les versions Fortran. Vous ne devez définir $SHF03si vous avez besoin d'utiliser un compilateur ou une version du compilateur spécifique pour les fichiers Fortran 03.

SHF03COM

La ligne de commande utilisée pour compiler un fichier source Fortran 03 à un fichier objet partagé bibliothèque. Vous ne devez définir $SHF03COMsi vous avez besoin d'utiliser une ligne de commande spécifique pour les fichiers Fortran 03. Vous devez normalement régler la $SHFORTRANCOMvariable qui spécifie la ligne de commande par défaut pour toutes les versions Fortran.

SHF03COMSTR

La chaîne affichée lorsqu'un fichier source Fortran 03 est compilé dans un fichier objet partagé bibliothèque. Si ce n'est pas défini, $SHF03COMou $SHFORTRANCOM (la ligne de commande) est affichée.

SHF03FLAGS

Les options qui sont passés au compilateur Fortran 03 à générés objets bibliothèque partagée. Vous ne devez définir $SHF03FLAGSsi vous avez besoin de définir des options spécifiques de l' utilisateur pour Fortran 03 fichiers. Vous devez normalement régler la $SHFORTRANFLAGSvariable qui spécifie les options définies par l' utilisateur transmis au compilateur par défaut Fortran pour toutes les versions Fortran.

SHF03PPCOM

La ligne de commande utilisée pour compiler un fichier source Fortran 03 à un fichier de bibliothèque partagée objet après la première exécution du fichier par l'intermédiaire du préprocesseur C. Toutes les options spécifiées dans les $SHF03FLAGSet les $CPPFLAGSvariables de construction sont inclus dans cette ligne de commande. Vous ne devez définir $SHF03PPCOMsi vous avez besoin d'utiliser une ligne de commande C-préprocesseur spécifique pour les fichiers Fortran 03. Vous devez normalement régler la $SHFORTRANPPCOMvariable qui spécifie la ligne de commande par défaut C-préprocesseur pour toutes les versions Fortran.

SHF03PPCOMSTR

La chaîne affichée lorsqu'un fichier source Fortran 03 est compilé dans un fichier objet bibliothèque partagée après la première exécution du fichier par le préprocesseur C. Si ce n'est pas défini, $SHF03PPCOMou $SHFORTRANPPCOM (la ligne de commande) est affichée.

SHF08

Le compilateur Fortran 08 utilisé pour générer des objets partagés bibliothèque. Vous devez normalement régler la $SHFORTRANvariable qui spécifie le compilateur par défaut Fortran pour toutes les versions Fortran. Vous ne devez définir $SHF08si vous avez besoin d'utiliser un compilateur ou une version du compilateur spécifique pour les fichiers Fortran 08.

SHF08COM

La ligne de commande utilisée pour compiler un fichier source Fortran 08 à un fichier objet partagé bibliothèque. Vous ne devez définir $SHF08COMsi vous avez besoin d'utiliser une ligne de commande spécifique pour les fichiers Fortran 08. Vous devez normalement régler la $SHFORTRANCOMvariable qui spécifie la ligne de commande par défaut pour toutes les versions Fortran.

SHF08COMSTR

La chaîne affichée lorsqu'un fichier source Fortran 08 est compilé dans un fichier objet partagé bibliothèque. Si ce n'est pas défini, $SHF08COMou $SHFORTRANCOM (la ligne de commande) est affichée.

SHF08FLAGS

Les options qui sont passés au compilateur Fortran 08 à générés objets bibliothèque partagée. Vous ne devez définir $SHF08FLAGSsi vous avez besoin de définir des options spécifiques de l' utilisateur pour Fortran 08 fichiers. Vous devez normalement régler la $SHFORTRANFLAGSvariable qui spécifie les options définies par l' utilisateur transmis au compilateur par défaut Fortran pour toutes les versions Fortran.

SHF08PPCOM

La ligne de commande utilisée pour compiler un fichier source Fortran 08 à un fichier de bibliothèque partagée objet après la première exécution du fichier par l'intermédiaire du préprocesseur C. Toutes les options spécifiées dans les $SHF08FLAGSet les $CPPFLAGSvariables de construction sont inclus dans cette ligne de commande. Vous ne devez définir $SHF08PPCOMsi vous avez besoin d'utiliser une ligne de commande C-préprocesseur spécifique pour les fichiers Fortran 08. Vous devez normalement régler la $SHFORTRANPPCOMvariable qui spécifie la ligne de commande par défaut C-préprocesseur pour toutes les versions Fortran.

SHF08PPCOMSTR

La chaîne affichée lorsqu'un fichier source Fortran 08 est compilé dans un fichier objet bibliothèque partagée après la première exécution du fichier par le préprocesseur C. Si ce n'est pas défini, $SHF08PPCOMou $SHFORTRANPPCOM (la ligne de commande) est affichée.

SHF77

Le compilateur Fortran 77 utilisé pour produire des objets partagés bibliothèque. Vous devez normalement régler la $SHFORTRANvariable qui spécifie le compilateur par défaut Fortran pour toutes les versions Fortran. Vous ne devez définir $SHF77si vous avez besoin d'utiliser un compilateur ou une version du compilateur spécifique pour les fichiers Fortran 77.

SHF77COM

La ligne de commande utilisée pour compiler un fichier source Fortran 77 dans un fichier objet partagé bibliothèque. Vous ne devez définir $SHF77COMsi vous avez besoin d'utiliser une ligne de commande spécifique pour les fichiers Fortran 77. Vous devez normalement régler la $SHFORTRANCOMvariable qui spécifie la ligne de commande par défaut pour toutes les versions Fortran.

SHF77COMSTR

La chaîne affichée lorsqu'un fichier source Fortran 77 est compilé dans un fichier objet partagé bibliothèque. Si ce n'est pas défini, $SHF77COMou $SHFORTRANCOM (la ligne de commande) est affichée.

SHF77FLAGS

Les options qui sont passés au compilateur Fortran 77 à générés objets bibliothèque partagée. Vous ne devez définir $SHF77FLAGSsi vous avez besoin de définir des options spécifiques de l' utilisateur pour les fichiers Fortran 77. Vous devez normalement régler la $SHFORTRANFLAGSvariable qui spécifie les options définies par l' utilisateur transmis au compilateur par défaut Fortran pour toutes les versions Fortran.

SHF77PPCOM

La ligne de commande utilisée pour compiler un fichier source Fortran 77 à un fichier de bibliothèque partagée objet après la première exécution du fichier par l'intermédiaire du préprocesseur C. Toutes les options spécifiées dans les $SHF77FLAGSet les $CPPFLAGSvariables de construction sont inclus dans cette ligne de commande. Vous ne devez définir $SHF77PPCOMsi vous avez besoin d'utiliser une ligne de commande C-préprocesseur spécifique pour les fichiers Fortran 77. Vous devez normalement régler la $SHFORTRANPPCOMvariable qui spécifie la ligne de commande par défaut C-préprocesseur pour toutes les versions Fortran.

SHF77PPCOMSTR

La chaîne affichée lorsqu'un fichier source Fortran 77 est compilé dans un fichier d'objets-bibliothèque partagée après la première exécution du fichier par l'intermédiaire du préprocesseur C. Si ce n'est pas défini, $SHF77PPCOMou $SHFORTRANPPCOM (la ligne de commande) est affichée.

SHF90

Le compilateur Fortran 90 utilisé pour produire des objets partagés bibliothèque. Vous devez normalement régler la $SHFORTRANvariable qui spécifie le compilateur par défaut Fortran pour toutes les versions Fortran. Vous ne devez définir $SHF90si vous avez besoin d'utiliser un compilateur ou une version du compilateur spécifique pour les fichiers Fortran 90.

SHF90COM

La ligne de commande utilisée pour compiler un fichier source Fortran 90 dans un fichier objet partagé bibliothèque. Vous ne devez définir $SHF90COMsi vous avez besoin d'utiliser une ligne de commande spécifique pour les fichiers Fortran 90. Vous devez normalement régler la $SHFORTRANCOMvariable qui spécifie la ligne de commande par défaut pour toutes les versions Fortran.

SHF90COMSTR

La chaîne affichée lorsqu'un fichier source Fortran 90 est compilé dans un fichier objet partagé bibliothèque. Si ce n'est pas défini, $SHF90COMou $SHFORTRANCOM (la ligne de commande) est affichée.

SHF90FLAGS

Les options qui sont passés au compilateur Fortran 90 générés objets bibliothèque partagée. Vous ne devez définir $SHF90FLAGSsi vous avez besoin de définir des options spécifiques de l' utilisateur pour les fichiers Fortran 90. Vous devez normalement régler la $SHFORTRANFLAGSvariable qui spécifie les options définies par l' utilisateur transmis au compilateur par défaut Fortran pour toutes les versions Fortran.

SHF90PPCOM

La ligne de commande utilisée pour compiler un fichier source Fortran 90 à un fichier de bibliothèque partagée objet après la première exécution du fichier par l'intermédiaire du préprocesseur C. Toutes les options spécifiées dans les $SHF90FLAGSet les $CPPFLAGSvariables de construction sont inclus dans cette ligne de commande. Vous ne devez définir $SHF90PPCOMsi vous avez besoin d'utiliser une ligne de commande C-préprocesseur spécifique pour les fichiers Fortran 90. Vous devez normalement régler la $SHFORTRANPPCOMvariable qui spécifie la ligne de commande par défaut C-préprocesseur pour toutes les versions Fortran.

SHF90PPCOMSTR

La chaîne affichée lorsqu'un fichier source Fortran 90 est compilé dans un fichier d'objets-bibliothèque partagée après la première exécution du fichier par l'intermédiaire du préprocesseur C. Si ce n'est pas défini, $SHF90PPCOMou $SHFORTRANPPCOM (la ligne de commande) est affichée.

SHF95

Le compilateur Fortran 95 utilisé pour produire des objets partagés bibliothèque. Vous devez normalement régler la $SHFORTRANvariable qui spécifie le compilateur par défaut Fortran pour toutes les versions Fortran. Vous ne devez définir $SHF95si vous avez besoin d'utiliser un compilateur ou une version du compilateur spécifique pour les fichiers Fortran 95.

SHF95COM

La ligne de commande utilisée pour compiler un fichier source Fortran 95 dans un fichier objet partagé bibliothèque. Vous ne devez définir $SHF95COMsi vous avez besoin d'utiliser une ligne de commande spécifique pour les fichiers Fortran 95. Vous devez normalement régler la $SHFORTRANCOMvariable qui spécifie la ligne de commande par défaut pour toutes les versions Fortran.

SHF95COMSTR

La chaîne affichée lorsqu'un fichier source Fortran 95 est compilé dans un fichier objet partagé bibliothèque. Si ce n'est pas défini, $SHF95COMou $SHFORTRANCOM (la ligne de commande) est affichée.

SHF95FLAGS

Les options qui sont passés au compilateur Fortran 95 générés objets bibliothèque partagée. Vous ne devez définir $SHF95FLAGSsi vous avez besoin de définir des options spécifiques de l' utilisateur pour les fichiers Fortran 95. Vous devez normalement régler la $SHFORTRANFLAGSvariable qui spécifie les options définies par l' utilisateur transmis au compilateur par défaut Fortran pour toutes les versions Fortran.

SHF95PPCOM

La ligne de commande utilisée pour compiler un fichier source Fortran 95 à un fichier de bibliothèque partagée objet après la première exécution du fichier par l'intermédiaire du préprocesseur C. Toutes les options spécifiées dans les $SHF95FLAGSet les $CPPFLAGSvariables de construction sont inclus dans cette ligne de commande. Vous ne devez définir $SHF95PPCOMsi vous avez besoin d'utiliser une ligne de commande C-préprocesseur spécifique pour les fichiers Fortran 95. Vous devez normalement régler la $SHFORTRANPPCOMvariable qui spécifie la ligne de commande par défaut C-préprocesseur pour toutes les versions Fortran.

SHF95PPCOMSTR

La chaîne affichée lorsqu'un fichier source Fortran 95 est compilé dans un fichier d'objets-bibliothèque partagée après la première exécution du fichier par l'intermédiaire du préprocesseur C. Si ce n'est pas défini, $SHF95PPCOMou $SHFORTRANPPCOM (la ligne de commande) est affichée.

SHFORTRAN

Le compilateur Fortran par défaut utilisé pour générer des objets partagés bibliothèque.

SHFORTRANCOM

La ligne de commande utilisée pour compiler un fichier source Fortran à un fichier d'objet partagé bibliothèque.

SHFORTRANCOMSTR

La chaîne affichée lorsqu'un fichier source Fortran est compilé dans un fichier objet partagé bibliothèque. Si ce n'est pas défini, $SHFORTRANCOM est affiché (la ligne de commande).

SHFORTRANFLAGS

Les options qui sont passés au compilateur Fortran pour générer des objets shared-bibliothèque.

SHFORTRANPPCOM

La ligne de commande utilisée pour compiler un fichier source Fortran à un fichier de bibliothèque partagée objet après la première exécution du fichier par l'intermédiaire du préprocesseur C. Toutes les options spécifiées dans les $SHFORTRANFLAGSet les $CPPFLAGSvariables de construction sont inclus dans cette ligne de commande.

SHFORTRANPPCOMSTR

La chaîne affichée lorsqu'un fichier source Fortran est compilé dans un fichier d'objets-bibliothèque partagée après la première exécution du fichier par l'intermédiaire du préprocesseur C. Si ce n'est pas défini, $SHFORTRANPPCOM est affiché (la ligne de commande).

SHLIBEMITTER

FAIRE

SHLIBNOVERSIONSYMLINKS

Charge le SharedLibraryconstructeur de ne pas créer des liens symboliques pour les bibliothèques partagées versionnées.

SHLIBPREFIX

Le préfixe utilisé pour les noms de fichiers bibliothèque partagée.

_SHLIBSONAME

Une macro qui génère automatiquement SONAME de bibliothèque partagée basée sur TARGET $, $ SHLIBVERSION et SHLIBSUFFIX $. Utilisé par le SharedLibraryconstructeur lorsque l'outil de liaison prend en charge SONAME (par exemple gnulink).

SHLIBSUFFIX

Le suffixe utilisé pour les noms de fichiers bibliothèque partagée.

SHLIBVERSION

Lorsque cette variable de construction est définie, une bibliothèque partagée versionné est créée par le SharedLibraryconstructeur. Ceci active le $_SHLIBVERSIONFLAGSet modifie ainsi le $SHLINKCOMcas échéant, ajoute le numéro de version au nom de la bibliothèque, et crée les liens symboliques nécessaires. $SHLIBVERSIONversions devraient exister en tant que valeurs décimales délimité par des alpha-numériques, tel que défini par l'expression régulière "\ w + [\. \ w +] *". Exemple $SHLIBVERSIONvaleurs incluent '1', '1.2.3' et «1.2.gitaa412c8b.

_SHLIBVERSIONFLAGS

Cette macro introduit automatiquement des drapeaux supplémentaires à $SHLINKCOMlors de la construction versionné SharedLibrary(qui est quand $SHLIBVERSION est réglé). _SHLIBVERSIONFLAGSajoute habituellement $SHLIBVERSIONFLAGS et quelques options supplémentaires générées dynamiquement (par exemple `-Wl,-soname=$_SHLIBSONAME`. Il est utilisé par « ordinaire » (sans version) des bibliothèques partagées.

SHLIBVERSIONFLAGS

Drapeaux supplémentaires ajoutés au $SHLINKCOMmoment de la construction versionné SharedLibrary. Ces drapeaux ne sont utilisés que lorsque $SHLIBVERSIONest réglé.

Shlink

L'éditeur de liens pour les programmes qui utilisent des bibliothèques partagées.

SHLINKCOM

La ligne de commande utilisée pour lier les programmes utilisant des bibliothèques partagées.

SHLINKCOMSTR

La chaîne affiche lorsque les programmes utilisant les bibliothèques partagées sont liées. Si ce n'est pas défini, $SHLINKCOMest affiché (la ligne de commande).

```
env = environnement (SHLINKCOMSTR = "Lien partagé target $")
```

SHLINKFLAGS

Options utilisateur général transmises à l'éditeur de liens pour les programmes utilisant des bibliothèques partagées. Notez que cette variable ne doit *pas* contenir `-l` (ou similaire) des options de liaison avec les bibliothèques énumérées dans $LIBS, ni `-L` (ou similaire) comprennent la recherche d' options de chemin qui SCons génère automatiquement à partir $LIBPATH. Voir $_LIBFLAGSci - dessus, pour la variable qui élargit les options bibliothèque de liens, et au- $_LIBDIRFLAGS dessus, pour la variable qui élargit les options de chemin de recherche de bibliothèque.

SHOBJPREFIX

Le préfixe utilisé pour les noms de fichiers d'objets partagés.

SHOBJSUFFIX

Le suffixe utilisé pour les noms de fichiers d'objets partagés.

SONAME

Variable utilisée pour SONAME dur code pour la bibliothèque partagée versionnée / module chargeable.

```
env.SharedLibrary ( 'test', 'test.c', SHLIBVERSION = '0.1.2', SONAME = 'libtest.so.2')
```

La variable est utilisée, par exemple, par l' gnulinkoutil de liaison.

LA SOURCE

Un nom de variable réservé qui ne peut pas être défini ou utilisé dans un environnement de construction. (Voir « Remplacement variable » ci-dessous.)

SOURCE_URL

L'URL (adresse web) de l'emplacement à partir de laquelle le projet a été récupéré. Il est utilisé pour remplir le `Source:` champ dans les informations de contrôle des paquets RPM et Ipkg.

SOURCES

Un nom de variable réservé qui ne peut pas être défini ou utilisé dans un environnement de construction. (Voir « Remplacement variable » ci-dessous.)

FRAYER

Une fonction d'interpréteur de commandes qui sera appelée à exécuter des chaînes de ligne de commande. La fonction doit attendre les arguments suivants:

```
frayer def (shell, évasion, cmd, args, env):
```

`sh` est une chaîne nommant le programme shell à utiliser. `escape`est une fonction qui peut être appelée pour échapper à des caractères spéciaux de coquille dans la ligne de commande. `cmd`est le chemin d' accès à la commande à exécuter. `args` est les arguments de la commande. `env` est un dictionnaire des variables d'environnement dans lequel la commande doit être exécutée.

STATIC_AND_SHARED_OBJECTS_ARE_THE_SAME

Lorsque cette variable est true, les objets statiques et des objets partagés sont supposés être les mêmes; à savoir, SCons ne vérifie pas pour lier des objets statiques dans une bibliothèque partagée.

SUBST_DICT

Le dictionnaire utilisé par le `Substfile`ou les `Textfile`constructeurs pour les valeurs de substitution. Il peut être quelque chose jugée acceptable par le constructeur dict (), donc en plus un dictionnaire, des listes de tuples sont également acceptables.

SUBSTFILEPREFIX

Le préfixe utilisé pour les `Substfile`noms de fichiers, la chaîne vide par défaut.

SUBSTFILESUFFIX

Le suffixe utilisé pour les `Substfile`noms de fichiers, la chaîne vide par défaut.

RÉSUMÉ

Un bref résumé de ce que le projet est sur le point . Il est utilisé pour remplir le `Summary:` champ dans les informations de contrôle des paquets RPM et Ipkg, et comme le `Description:` champ dans les packages MSI.

LAMPÉE

L'enveloppe de langage de script et le générateur d'interface.

SWIGCFILESUFFIX

Le suffixe qui sera utilisé pour les fichiers source intermédiaires C générés par l'enveloppe de langage de script et le générateur d'interface. La valeur par défaut est `_wrap`$CFILESUFFIX. Par défaut, cette valeur est utilisée chaque fois que l' `-c++` option est *non* spécifiée dans le cadre de la $SWIGFLAGS variable de la construction.

SWIGCOM

La ligne de commande utilisée pour appeler l'emballage de langage de script et le générateur d'interface.

SWIGCOMSTR

La chaîne affichée lorsque vous appelez l'emballage de langage de script et le générateur d'interface. Si ce n'est pas défini, $SWIGCOMest affiché (la ligne de commande).

SWIGCXXFILESUFFIX

Le suffixe qui sera utilisé pour les fichiers source intermédiaires ++ C générés par l'enveloppe de langage de script et le générateur d'interface. La valeur par défaut est `_wrap`$CFILESUFFIX. Par défaut, cette valeur est utilisée chaque fois que l' `-c++` option est spécifiée en tant que partie de la $SWIGFLAGS variable de construction.

SWIGDIRECTORSUFFIX

Le suffixe qui sera utilisé pour les fichiers intermédiaires d' en- tête ++ C générés par l'enveloppe de langage de script et le générateur d'interface. Ceux - ci ne sont générés pour le code C lorsque la fonction 'administration de la SWIG est activée. La valeur par défaut est `_wrap.h`.

SWIGFLAGS

Options générales transmises à l'emballage de langage de script et le générateur d'interface. C'est là que vous devez définir `-python`, `-perl5`, `-tcl`ou toutes les autres options que vous souhaitez spécifier à SWIG. Si vous réglez l' `-c++` option dans cette variable,`scons` sera, par défaut, générer un fichier source C ++ intermédiaire avec l'extension qui est spécifié comme$CXXFILESUFFIX variable.

_SWIGINCFLAGS

Une variable de construction généré automatiquement contenant les options de ligne de commande SWIG pour spécifier les répertoires à rechercher les fichiers inclus. La valeur de la `$_SWIGINCFLAGS`création en annexant `$SWIGINCPREFIX`et `$SWIGINCSUFFIX` au début et à la fin de chaque répertoire `$SWIGPATH`.

SWIGINCPREFIX

Le préfixe utilisé pour spécifier un répertoire include sur la ligne de commande SWIG. Ce sera ajouté au début de chaque répertoire dans la `$SWIGPATH`variable de construction lorsque la `$_SWIGINCFLAGS`variable est générée automatiquement.

SWIGINCSUFFIX

Le suffixe utilisé pour spécifier un répertoire include sur la ligne de commande SWIG. Ce sera ajouté à la fin de chaque répertoire dans la `$SWIGPATH`variable de construction lorsque la `$_SWIGINCFLAGS`variable est générée automatiquement.

SWIGOUTDIR

Indique le répertoire de sortie dans lequel l'enveloppe de langage de script et le générateur d'interface doit placer les fichiers générés spécifiques à chaque langue. Il sera utilisé par SCons pour identifier les fichiers qui seront générés par le rasade appel, et traduit en `swig -outdir`option sur la ligne de commande.

SWIGPATH

La liste des répertoires que l'emballage de langage de script et l'interface génèrent recherchera les fichiers inclus. Le scanner de dépendance implicite SWIG va rechercher ces répertoires pour inclure des fichiers. La valeur par défaut est une liste vide.

Ne pas mettre explicitement inclure des arguments de répertoire dans SWIGFLAGS; le résultat sera non-portables et les répertoires ne sera pas recherché par le scanner de dépendance. Remarque: les noms de répertoire dans SWIGPATH seront examinés en place par rapport au répertoire SConscript quand ils sont utilisés dans une commande. Pour forcer `scons` à Recherch un répertoire relatif à la racine de l'utilisation de l' arbre source `#`:

```
env = environnement (SWIGPATH = '# / include')
```

Le look-up de répertoire peut également être configurée en utilisant la `Dir`() fonction:

```
include = Dir (include)
env = environnement (SWIGPATH = comprennent)
```

La liste d'annuaire sera ajouté pour commander les lignes à travers la généré automatiquement `$_SWIGINCFLAGS` variable de construction, qui est construit en ajoutant les valeurs de la`$SWIGINCPREFIX`et les `$SWIGINCSUFFIX` variables de construction pour le début et la fin de chaque répertoire `$SWIGPATH`. Toutes les lignes de commande vous définissez qui ont besoin de la liste des répertoires SWIGPATH devrait inclure `$_SWIGINCFLAGS`:

```
env = environnement (SWIGCOM = "my_swig -o $ CIBLES $ SOURCES _SWIGINCFLAGS $")
```

SWIGVERSION

Le numéro de version de l'outil de SWIG.

LE GOUDRON

Le archiveur de goudron.

Tarcom

La ligne de commande utilisé pour appeler le dispositif d'archivage de goudron.

TARCOMSTR

La chaîne affichée lors de l' archivage des fichiers en utilisant l'archiveur tar. Si ce n'est pas défini, `$TARCOM`est affiché (la ligne de commande).

```
env = environnement (TARCOMSTR = "Archiving $ TARGET")
```

TARFLAGS

Options générales transmises à l'archiveur de goudron.

CIBLE

Un nom de variable réservé qui ne peut pas être défini ou utilisé dans un environnement de construction. (Voir « Remplacement variable » ci-dessous.)

TARGET_ARCH

Définit l'architecture cible pour le compilateur Visual Studio (c. -à l'arc des binaires générés par le compilateur). Si non défini, par défaut à `$HOST_ARCH`, ou, si ce n'est pas définie, à l'architecture du système d' exploitation de la machine en marche (notez que la construction de python ou de l' architecture n'a pas d' effet). Cette variable doit être passé comme argument au constructeur de l' environnement (); il a plus tard la mise en aucun effet. Ceci est actuellement utilisé uniquement sous Windows, mais dans l'avenir , il sera utilisé sur d' autres systèmes d' exploitation aussi bien.

Les valeurs valides pour Windows sont `x86`, `i386` (pour 32 bits);`amd64`, `emt64`, `x86_64` (Pour 64 bits); et `ia64`(Itanium). Par exemple, si vous voulez compiler les binaires 64 bits, vous devez définir`TARGET_ARCH='x86_64'`dans votre environnement SCons.

Le nom de l'architecture matérielle cible pour les objets compilés créés par cet environnement. La valeur par défaut de la valeur de HOST_ARCH, et l'utilisateur peut la remplacer. Actuellement, seulement défini pour Win32.

TARGET_OS

Le nom du système d'exploitation cible pour les objets compilés créés par cet environnement. La valeur par défaut de la valeur de HOST_OS, et l'utilisateur peut la remplacer. Actuellement, seulement défini pour Win32.

CIBLES

Un nom de variable réservé qui ne peut pas être défini ou utilisé dans un environnement de construction. (Voir « Remplacement variable » ci-dessous.)

TARSUFFIX

Le suffixe utilisé pour les noms de fichiers tar.

TEMPFILEPREFIX

Le préfixe pour un fichier temporaire utilisé pour exécuter des lignes de plus de $ MAXLINELENGTH. La valeur par défaut est « @ ». Cela peut être défini pour toolchains qui utilisent d'autres valeurs, telles que « - @ » pour le compilateur ou diab « -via » pour ARM toolchain.

TEXAS

Le formatter TeX et typographe.

Texcom

La ligne de commande utilisée pour appeler le formatter TeX et typographe.

TEXCOMSTR

La chaîne affichée lorsque vous appelez la formatter TeX et typographe. Si ce n'est pas défini, `$TEXCOM`est affiché (la ligne de commande).

```
env = environnement (TEXCOMSTR = "du bâtiment de $ TARGET d'entrée TeX SOURCES $")
```

TEXFLAGS

Options générales transmises au formatter TeX et typographe.

TEXINPUTS

Liste des répertoires que le programme LaTeX va rechercher include. Le scanner de dépendance implicite LaTeX va rechercher ces répertoires pour \ include et \ fichiers d'importation.

TEXTFILEPREFIX

Le préfixe utilisé pour les `Textfile`noms de fichiers, la chaîne vide par défaut.

TEXTFILESUFFIX

Le suffixe utilisé pour les `Textfile`noms de fichiers; `.txt`par défaut.

OUTILS

Une liste des noms des spécifications d'outils qui font partie de cet environnement de construction.

UNCHANGED_SOURCES

Un nom de variable réservé qui ne peut pas être défini ou utilisé dans un environnement de construction. (Voir « Remplacement variable » ci-dessous.)

UNCHANGED_TARGETS

Un nom de variable réservé qui ne peut pas être défini ou utilisé dans un environnement de construction. (Voir « Remplacement variable » ci-dessous.)

VENDEUR

La personne ou l' organisation qui fournissent le logiciel emballé. Il est utilisé pour remplir le `Vendor:` champ dans les informations de contrôle des paquets RPM et le `Manufacturer:` champ dans les informations de contrôle des paquets MSI.

VERSION

La version du projet, spécifié comme une chaîne.

WIN32_INSERT_DEF

Un synonyme dépréciée pour `$WINDOWS_INSERT_DEF`.

WIN32DEFPREFIX

Un synonyme dépréciée pour `$WINDOWSDEFPREFIX`.

WIN32DEFSUFFIX

Un synonyme dépréciée pour `$WINDOWSDEFSUFFIX`.

WIN32EXPPREFIX

Un synonyme dépréciée pour `$WINDOWSEXPSUFFIX`.

WIN32EXPSUFFIX

Un synonyme dépréciée pour `$WINDOWSEXPSUFFIX`.

WINDOWS_EMBED_MANIFEST

Définissez cette variable sur True ou 1 pour intégrer le compilateur généré manifeste (normalement `${TARGET}.manifest`) dans tous les exes Windows et DLL construit avec cet environnement, comme une ressource au cours de leur étape de liaison. Cela se fait à l' aide `$MT`et `$MTEXECOM`et `$MTSHLIBCOM`.

WINDOWS_INSERT_DEF

Lorsque ce paramètre est réglé sur true, une version de la bibliothèque d'une bibliothèque partagée de Windows (`.dll`fichier) construira également un correspondant `.def`fichier en même temps, si un `.def`fichier est déjà répertorié comme une cible de construction. La valeur par défaut est 0 (ne pas construire un `.def`fichier).

WINDOWS_INSERT_MANIFEST

Lorsque ce paramètre est réglé sur true, `scons` sera au courant des`.manifest` fichiers générés par Microsoft Visua C / C ++ 8.

WINDOWSDEFPREFIX

Le préfixe utilisé pour Windows `.def`noms de fichiers.

WINDOWSDEFSUFFIX

Le suffixe utilisé pour Windows `.def`noms de fichiers.

WINDOWSEXPPREFIX

Le préfixe utilisé pour Windows `.exp`noms de fichiers.

WINDOWSEXPSUFFIX

Le suffixe utilisé pour Windows `.exp`noms de fichiers.

WINDOWSPROGMANIFESTPREFIX

Le préfixe utilisé pour programme exécutables `.manifest`fichiers générés par Microsoft Visual C / C ++.

WINDOWSPROGMANIFESTSUFFIX

Le suffixe utilisé pour programme exécutables `.manifest`fichiers générés par Microsoft Visual C / C ++.

WINDOWSSHLIBMANIFESTPREFIX

Le préfixe utilisé pour la bibliothèque partagée `.manifest`fichiers générés par Microsoft Visual C / C ++.

WINDOWSSHLIBMANIFESTSUFFIX

Le suffixe utilisé pour la bibliothèque partagée `.manifest`fichiers générés par Microsoft Visual C / C ++.

X_IPK_DEPENDS

Il est utilisé pour remplir le `Depends:` champ dans les informations de contrôle des paquets Ipkg.

X_IPK_DESCRIPTION

Il est utilisé pour remplir le `Description:` champ dans les informations de contrôle des paquets Ipkg. La valeur par défaut est `$SUMMARY\n$DESCRIPTION`

X_IPK_MAINTAINER

Il est utilisé pour remplir le `Maintainer:` champ dans les informations de contrôle des paquets Ipkg.

X_IPK_PRIORITY

Il est utilisé pour remplir le `Priority:` champ dans les informations de contrôle des paquets Ipkg.

X_IPK_SECTION

Il est utilisé pour remplir le `Section:` champ dans les informations de contrôle des paquets Ipkg.

X_MSI_LANGUAGE

Il est utilisé pour remplir l' `Language:` attribut dans les informations de contrôle des paquets MSI.

X_MSI_LICENSE_TEXT

Le texte de la licence du logiciel au format RTF. caractères de retour chariot seront remplacés par le pair \\ équivalent RTF.

X_MSI_UPGRADE_CODE

FAIRE

X_RPM_AUTOREQPROV

Il est utilisé pour remplir le `AutoReqProv:` champ dans le RPM`.spec`fichier.

X_RPM_BUILD

interne, mais Overridable

X_RPM_BUILDREQUIRES

Il est utilisé pour remplir le `BuildRequires:` champ dans le RPM `.spec`fichier.

X_RPM_BUILDROOT

interne, mais Overridable

X_RPM_CLEAN

interne, mais Overridable

X_RPM_CONFLICTS

Il est utilisé pour remplir le `Conflicts:` champ dans le RPM `.spec`fichier.

X_RPM_DEFATTR

Cette valeur est utilisée comme les attributs par défaut pour les fichiers du paquetage RPM. La valeur par défaut est`(-,root,root)`.

X_RPM_DISTRIBUTION

Il est utilisé pour remplir le `Distribution:` champ dans le RPM `.spec`fichier.

X_RPM_EPOCH

Il est utilisé pour remplir le `Epoch:` champ dans les informations de contrôle des paquets RPM.

X_RPM_EXCLUDEARCH

Il est utilisé pour remplir le `ExcludeArch:` champ dans le RPM `.spec`fichier.

X_RPM_EXLUSIVEARCH

Il est utilisé pour remplir le `ExclusiveArch:` champ dans le RPM `.spec`fichier.

X_RPM_GROUP

Il est utilisé pour remplir le `Group:` champ dans le RPM `.spec`fichier.

X_RPM_GROUP_lang

Il est utilisé pour remplir le `Group(lang):` champ dans le RPM `.spec`fichier. Notez que `lang` n'est pas littérale et doit être remplacé par le code de langue approprié.

X_RPM_ICON

Il est utilisé pour remplir le `Icon:` champ dans le RPM `.spec`fichier.

X_RPM_INSTALL

interne, mais Overridable

X_RPM_PACKAGER

Il est utilisé pour remplir le `Packager:` champ dans le RPM `.spec`fichier.

X_RPM_POSTINSTALL

Il est utilisé pour remplir la `%post:` section du RPM `.spec`fichier.

X_RPM_POSTUNINSTALL

Il est utilisé pour remplir la `%postun:` section du RPM `.spec`fichier.

X_RPM_PREFIX

Il est utilisé pour remplir le `Prefix:` champ dans le RPM `.spec`fichier.

X_RPM_PREINSTALL

Il est utilisé pour remplir la `%pre:` section du RPM `.spec`fichier.

X_RPM_PREP

interne, mais Overridable

X_RPM_PREUNINSTALL

Il est utilisé pour remplir la `%preun:` section du RPM `.spec`fichier.

X_RPM_PROVIDES

Il est utilisé pour remplir le `Provides:` champ dans le RPM `.spec`fichier.

X_RPM_REQUIRES

Il est utilisé pour remplir le `Requires:` champ dans le RPM `.spec`fichier.

X_RPM_SERIAL

Il est utilisé pour remplir le `Serial:` champ dans le RPM `.spec`fichier.

X_RPM_URL

Il est utilisé pour remplir le `Url:` champ dans le RPM `.spec`fichier.

xgettext

Path to **xgettext (1)** programme (trouvé via `Detect()`). Voir `xgettext`outil et `POTUpdate`constructeur.

XGETTEXTCOM

Ligne de commande xgettext. Voir `xgettext`outil et `POTUpdate`constructeur.

XGETTEXTCOMSTR

Une chaîne de caractères qui est affichée lorsque **xgettext (1)** commande est invoquée (par défaut: `''`qui signifie « impression `$XGETTEXTCOM`»). Voir `xgettext`outil et `POTUpdate`constructeur.

_XGETTEXTDOMAIN

"Macro" interne. Génère **xgettext** forme nom de domaine source et la cible (par défaut: `'${TARGET.filebase}'`).

XGETTEXTFLAGS

Drapeaux supplémentaires **xgettext (1)** . Voir `xgettext`outil et `POTUpdate`constructeur.

XGETTEXTFROM

Nom du fichier contenant la liste des **xgettext (1)** des fichiers source d ». Les utilisateurs de autotools savent comme `POTFILES.in`ils fixeront dans la plupart des cas `XGETTEXTFROM="POTFILES.in"`ici. Les `$XGETTEXTFROM`fichiers ont la même syntaxe et sémantique que la GNU bien connue `POTFILES.in`. Voir `xgettext`outil et `POTUpdate`constructeur.

_XGETTEXTFROMFLAGS

"Macro" interne. Genrates liste des `-D<dir>`drapeaux de la `$XGETTEXTPATH`liste.

XGETTEXTFROMPREFIX

Ce drapeau est utilisé pour ajouter unique `$XGETTEXTFROM`fichier à**xgettext (1)** de commandline (par défaut: `'-f'`).

XGETTEXTFROMSUFFIX

(par défaut: `''`)

XGETTEXTPATH

Liste des répertoires, il **xgettext (1)** recherchera les fichiers sources (par défaut: `[]`).

Remarque

Cette variable ne fonctionne qu'avec `$XGETTEXTFROM`

Voir aussi l' `xgettext`outil et le `POTUpdate`constructeur.

_XGETTEXTPATHFLAGS

"Macro" interne. Génère liste des `-f<file>`drapeaux de `$XGETTEXTFROM`.

XGETTEXTPATHPREFIX

Ce drapeau est utilisé pour ajouter le chemin de recherche simple à **xgettext (1)** de commandline (par défaut: `'-D'`).

XGETTEXTPATHSUFFIX

(par défaut: `''`)

YACC

Le générateur d'analyseur.

YACCCOM

La ligne de commande utilisé pour appeler le générateur d'analyseur pour générer un fichier source.

YACCCOMSTR

La chaîne affichée lors de la génération d' un fichier source en utilisant le générateur d'analyseurs. Si ce n'est pas défini, `$YACCCOM`est affiché (la ligne de commande).

```
env = environnement (YACCCOMSTR = "Yacc'ing $ CIBLES de $ SOURCES")
```

YACCFLAGS

Options générales transmises au générateur d'analyseur syntaxique. Si `$YACCFLAGS`contient une `-d`option SCons suppose que l'appel sera également créer un fichier .h (si le fichier source Yacc se termine par un suffixe .y) ou un fichier .hpp (si le fichier source Yacc se termine par un suffixe .yy)

YACCHFILESUFFIX

Le suffixe du fichier d' en- tête C généré par le générateur de l' analyseur lorsque l' `-d` on utilise l' option. Notez que la définition de cette variable ne provoque pas le générateur d'analyseur pour générer un fichier d' en- tête avec le suffixe spécifié, il existe pour vous permettre de spécifier quel suffixe le générateur d'analyseur utilisera de son propre gré. La valeur par défaut est `.h`.

YACCHXXFILESUFFIX

Le suffixe du fichier d' en- tête de C généré par le générateur de l' analyseur lorsque l' `-d` on utilise l' option. Notez que la définition de cette variable ne provoque pas le générateur d'analyseur pour générer un fichier d' en- tête avec le suffixe spécifié, il existe pour vous permettre de spécifier quel suffixe le générateur d'analyseur utilisera de son propre gré. La valeur par défaut est `.hpp`, sauf sous Mac OS X, la valeur par défaut est `${TARGET.suffix}.h`. parce que le défaut bison générateur d'analyseur syntaxique juste ajoute `.h` au nom du fichier C ++ généré.

YACCVCGFILESUFFIX

Le suffixe du fichier contenant la définition d'automate de grammaire VCG lorsque l' `--graph=` on utilise l' option. Notez que la définition de cette variable ne provoque pas le générateur d'analyseur pour générer un fichier VCG avec le suffixe spécifié, il existe pour vous permettre de spécifier quel suffixe le générateur d'analyseur utilisera de son propre gré. La valeur par défaut est `.vcg`.

ZIP *: FRANÇAIS

La compression zip et utilitaire de personnalisation du fichier.

ZIPCOM

La ligne de commande utilisé pour appeler l'utilitaire postal, ou la fonction de python interne utilisé pour créer une archive zip.

ZIPCOMPRESSION

Le `compression` drapeau du python `zipfile` module utilisé par la fonction de python interne pour contrôler si l'archive zip est comprimée ou non. La valeur par défaut est `zipfile.ZIP_DEFLATED`, ce qui crée une archive compressée zip. Cette valeur n'a pas d' effet si le `zipfile` module est indisponible.

ZIPCOMSTR

La chaîne affichée lors de l' archivage des fichiers en utilisant l'utilitaire zip. Si ce paramètre est réglé pas, `$ZIPCOM` (la ligne de commande ou de la fonction Python interne) est affiché.

```
env = environnement (ZIPCOMSTR = "Zipping $ TARGET")
```

ZIPFLAGS

Options générales transmises à l'utilitaire zip.

ZIPROOT

Un répertoire racine zip en option (par défaut vide). Les noms de fichiers stockés dans le fichier zip sera par rapport à ce répertoire, si on leur donne. Sinon, les fichiers sont relatifs au répertoire courant de la commande. Par exemple:

```
env = environnement ()
env.Zip ( 'foo.zip', 'subdir1 / subdir2 / file1', ZIPROOT = 'subdir1')
```

produira un fichier zip `foo.zip` contenant un fichier avec le nom`subdir2/file1`plutôt que `subdir1/subdir2/file1`.

ZIPSUFFIX

Le suffixe utilisé pour les noms de fichiers zip.

Annexe B. Builders

Cette annexe contient la description de tous les constructeurs qui sont *potentiellement* disponibles « hors de la boîte » dans cette version de SCons.

`CFile()` , `env.CFile()`

Construit un fichier source C donné une lex (`.l`) ou yacc (`.y`fichier d'entrée). Le suffixe spécifié par la `$CFILESUFFIX`variable de construction (`.c`par défaut) est automatiquement ajouté à la cible si elle est pas déjà présent. Exemple:

```
# Construit foo.c
env.CFile (target = 'foo.c', source = 'foo.l')
# Construit bar.c
env.CFile ( 'barre' target = source = 'bar.y')
```

`Command()` , `env.Command()`

Le `Command`« Builder » est effectivement mis en œuvre en fonction qui ressemble à un constructeur, mais prend en fait un argument supplémentaire de l'action à partir de laquelle le constructeur devrait être fait. Voir la `Command`description de la fonction de la syntaxe d'appel et les détails.

`CXXFile()` , `env.CXXFile()`

Construit un fichier source C ++ donné une lex (`.ll`) ou yacc (`.yy`fichier d'entrée). Le suffixe spécifié par la `$CXXFILESUFFIX`variable de construction (`.cc`par défaut) est automatiquement ajouté à la cible si elle est pas déjà présent. Exemple:

```
# Construit foo.cc
env.CXXFile (target = 'foo.cc', source = 'foo.ll')
# Construit bar.cc
env.CXXFile ( 'barre' target = source = 'bar.yy')
```

`DocbookEpub()` , `env.DocbookEpub()`

Un pseudo-générateur, en fournissant un ensemble d'outils pour la sortie Docbook EPUB.

```
env = environnement (outils = [ 'DocBook'])
env.DocbookEpub ( 'manual.epub', 'manual.xml')
```

ou simplement

```
env = environnement (outils = [ 'DocBook'])
env.DocbookEpub ( 'manuel')
```

`DocbookHtml()` , `env.DocbookHtml()`

Un pseudo-Builder, fournissant un ensemble d'outils DocBook pour la sortie HTML.

```
env = environnement (outils = [ 'DocBook'])
env.DocbookHtml ( 'manual.html', 'manual.xml')
```

ou simplement

```
env = environnement (outils = [ 'DocBook'])
env.DocbookHtml ( 'manuel')
```

`DocbookHtmlChunked()` , `env.DocbookHtmlChunked()`

Un pseudo-Builder, fournissant un ensemble d' outils DocBook pour la sortie HTML chunked. Il prend en charge le `base.dir`paramètre. Le `chunkfast.xsl`fichier (nécessite « EXSLT ») est utilisé comme feuille de style par défaut. La syntaxe de base:

```
env = environnement (outils = [ 'DocBook'])
env.DocbookHtmlChunked ( 'manuel')
```

où `manual.xml`est le fichier d'entrée.

Si vous utilisez le `root.filename` paramètre dans vos propres feuilles de style , vous devez indiquer le nouveau nom de la cible. Cela garantit que les dépendances se corriger, en particulier pour le nettoyage via « `scons -c` » :

```
env = environnement (outils = [ 'DocBook'])
env.DocbookHtmlChunked ( 'mymanual.html', 'manuel', xsl = 'htmlchunk.xsl')
```

Un soutien de base pour l' `base.dir`est fourni. Vous pouvez ajouter le `base_dir`mot - clé à votre appel Builder, et le préfixe donné gets à tous les préfixé créés noms:

```
env = environnement (outils = [ 'DocBook'])
env.DocbookHtmlChunked ( 'manuel', XSL = 'htmlchunk.xsl', base_dir = 'output /')
```

Assurez-vous que vous ne pas oublier la barre oblique de fin pour le dossier de base, sinon vos fichiers sont renommés seulement!

`DocbookHtmlhelp()` , `env.DocbookHtmlhelp()`

Un pseudo-générateur, en fournissant un ensemble d'outils pour la sortie Docbook HtmlHelp. Sa syntaxe de base est:

```
env = environnement (outils = [ 'DocBook'])
env.DocbookHtmlhelp ( 'manuel')
```

où `manual.xml`est le fichier d'entrée.

Si vous utilisez le `root.filename` paramètre dans vos propres feuilles de style , vous devez indiquer le nouveau nom de la cible. Cela garantit que les dépendances se corriger, en particulier pour le nettoyage via « `scons -c` » :

```
env = environnement (outils = [ 'DocBook'])
env.DocbookHtmlhelp ( 'mymanual.html', 'manuel', xsl = 'htmlhelp.xsl')
```

Un soutien de base pour le `base.dir`paramètre est fourni. Vous pouvez ajouter le `base_dir`mot - clé à votre appel Builder, et le préfixe donné gets à tous les préfixé créés noms:

```
env = environnement (outils = [ 'DocBook'])
env.DocbookHtmlhelp ( 'manuel', XSL = 'htmlhelp.xsl', base_dir = 'output /')
```

Assurez-vous que vous ne pas oublier la barre oblique de fin pour le dossier de base, sinon vos fichiers sont renommés seulement!

`DocbookMan()` , `env.DocbookMan()`

Un pseudo-Builder, fournissant un ensemble d'outils DocBook pour la sortie de la page Man. Sa syntaxe de base est:

```
env = environnement (outils = [ 'DocBook'])
env.DocbookMan ( 'manuel')
```

où `manual.xml`est le fichier d'entrée. Notez que vous pouvez spécifier un nom cible, mais les noms de sortie réels sont réglés automatiquement à partir des `refname`entrées dans votre source XML.

`DocbookPdf()` , `env.DocbookPdf()`

Un pseudo-Builder, fournissant un ensemble d'outils DocBook pour la sortie PDF.

```
env = environnement (outils = [ 'DocBook'])
env.DocbookPdf ( 'manual.pdf', 'manual.xml')
```

ou simplement

```
env = environnement (outils = [ 'DocBook'])
env.DocbookPdf ( 'manuel')
```

`DocbookSlidesHtml()` , `env.DocbookSlidesHtml()`

Un pseudo-Builder, fournissant un ensemble d'outils DocBook pour diapositives HTML sortie.

```
env = environnement (outils = [ 'DocBook'])
env.DocbookSlidesHtml ( 'manuel')
```

Si vous utilisez le `titlefoil.html`paramètre dans vos propres feuilles de style que vous devez donner le nouveau nom de la cible. Cela garantit que les dépendances se corriger, en particulier pour le nettoyage via « `scons -c` » :

```
env = environnement (outils = [ 'DocBook'])
env.DocbookSlidesHtml ( 'mymanual.html', 'manuel', xsl = 'slideshtml.xsl')
```

Un soutien de base pour le `base.dir`paramètre est fourni. Vous pouvez ajouter le `base_dir`mot - clé à votre appel Builder, et le préfixe donné gets à tous les préfixé créés noms:

```
env = environnement (outils = [ 'DocBook'])
env.DocbookSlidesHtml ( 'manuel', XSL = 'slideshtml.xsl', base_dir = 'output /')
```

Assurez-vous que vous ne pas oublier la barre oblique de fin pour le dossier de base, sinon vos fichiers sont renommés seulement!

`DocbookSlidesPdf()` , `env.DocbookSlidesPdf()`

Un pseudo-Builder, fournissant un ensemble d'outils DocBook pour la sortie de diapositives PDF.

```
env = environnement (outils = [ 'DocBook'])
env.DocbookSlidesPdf ( 'manual.pdf', 'manual.xml')
```

ou simplement

```
env = environnement (outils = [ 'DocBook'])
env.DocbookSlidesPdf ( 'manuel')
```

`DocbookXInclude()` , `env.DocbookXInclude()`

Un pseudo-Builder, pour résoudre XIncludes dans une étape de traitement séparée.

```
env = environnement (outils = [ 'DocBook'])
env.DocbookXInclude ( 'manual_xincluded.xml', 'manual.xml')
```

`DocbookXslt()` , `env.DocbookXslt()`

Un pseudo-Builder, l'application d'une transformation XSL donné dans le fichier d'entrée.

```
env = environnement (outils = [ 'DocBook'])
env.DocbookXslt ( 'manual_transformed.xml', 'manual.xml', XSL = 'transform.xslt')
```

Notez que ce constructeur exige que le `xsl`paramètre à régler.

`DVI()` , `env.DVI()`

Construit un `.dvi`fichier à partir d' un `.tex`, `.ltx`ou `.latex`fichier d'entrée. Si le suffixe du fichier source est `.tex`, `scons` examinera le contenu du fichier; si la chaîne `\documentclass` ou`\documentstyle` est trouvée, le fichier est supposé être un fichier de latex et la cible est construite en appelant la `$LATEXCOM`ligne de commande; sinon, la `$TEXCOM`ligne de commande est utilisé. Si le fichier est un fichier LaTeX, la `DVI` méthode constructeur va également examiner le contenu du `.aux` fichier et appeler la `$BIBTEX`ligne de commande si la chaîne `bibdata` est trouvée, commencez `$MAKEINDEX`à générer un index si un `.ind` fichier est trouvé et examinera le contenu `.log` fichier et réexécution la `$LATEXCOM`commande si le fichier journal indique qu'il est nécessaire.

Le suffixe `.dvi` (codé en dur à l'intérieur de TeX lui - même) est automatiquement ajouté à la cible si elle est pas déjà présent. Exemples:

```
# Construit à partir aaa.tex
env.DVI (target = 'aaa.dvi', source = 'aaa.tex')
# Construit bbb.dvi
env.DVI (target = 'bbb', source = 'bbb.ltx')
# Construit à partir ccc.latex
env.DVI (target = 'ccc.dvi', source = 'ccc.latex')
```

`Gs()` , `env.Gs()`

Un Builder pour appeler explicitement l' `gs`exécutable. Selon le système d' exploitation sous - jacents, les différents noms `gs`,`gsos2`et `gswin32c` sont jugés.

```
env = environnement (outils = [ 'gs'])
env.Gs ( 'cover.jpg', 'scons-scons.pdf',
       GSFLAGS = '- dNOPAUSE -dBATCH -sDEVICE = jpeg -dFirstPage = 1 -dLastPage = 1 -q')
       )
```

`Install()` , `env.Install()`

Installe un ou plusieurs fichiers source ou répertoires dans la cible spécifiée, qui doit être un répertoire. Les noms des fichiers source spécifiés ou les répertoires restent les mêmes dans le répertoire de destination. Les sources peuvent être donnés sous forme de chaîne ou un nœud retourné par un constructeur.

```
env.Install ( '/ usr / local / bin', source = [ 'foo', 'bar'])
```

`InstallAs()` , `env.InstallAs()`

Installe un ou plusieurs fichiers sources ou des répertoires à des noms spécifiques, permettant la modification d'un nom de fichier ou répertoire dans le cadre de l'installation. Il est une erreur si les arguments sources et cibles liste des numéros de fichiers ou répertoires.

```
env.InstallAs (target = '/ usr / local / bin / foo',
               source = 'foo_debug')
env.InstallAs (target = [ '../lib/libfoo.a', '../lib/libbar.a'],
               source = [ 'libtruc.a', 'libBAR.a'])
```

`InstallVersionedLib()` , `env.InstallVersionedLib()`

Installe une bibliothèque partagée versionné. Les liens symboliques appropriés à l'architecture seront générés sur la base des liens symboliques de la bibliothèque source.

```
env.InstallVersionedLib (target = '/ usr / local / bin / foo',
              source = 'libxyz.1.5.2.so')
```

`Jar()` , `env.Jar()`

Construit une archive Java (`.jar`fichier) dans la liste spécifiée des sources. Tous les répertoires dans la liste des sources seront recherchées pour les `.class`fichiers). Tous les `.java`fichiers dans la liste des sources seront compilés pour les `.class`fichiers en appelant le `Java`constructeur.

Si la `$JARCHDIR`valeur est définie, le pot commande va changer dans le répertoire spécifié en utilisant l' `-C` option. Si `$JARCHDIR`est pas explicitement définie, SCons utilisera le haut de tout arbre de sous - répertoire dans lequel Java `.class` ont été construits par le `Java`constructeur.

Si le contenu des fichiers des sources commencent par la chaîne`Manifest-Version`, le fichier est supposé être un manifeste et est passé à la jarre commande avec le `m` jeu d'options.

```
env.Jar (target = 'foo.jar', source = 'classes')

env.Jar (target = 'bar.jar',
        source = [ 'bar1.java', 'bar2.java'])
```

`Java()` , `env.Java()`

Construit un ou plusieurs fichiers de classe Java. Les sources peuvent être une combinaison explicite des `.java`fichiers ou des arborescences qui seront scannés pour les `.java`fichiers.

SCons analysera chaque source `.java`fichier pour trouver les classes (y compris les classes internes) définies dans ce fichier, et à partir de ce chiffre sur la cible des `.class`fichiers qui seront créés. Les fichiers de classe seront placés sous le répertoire cible spécifié.

SCons cherchera également chaque fichier Java pour le nom du package Java, qu'il suppose se trouve sur une ligne commençant par la chaîne `package` dans la première colonne; les résultants `.class`fichiers seront placés dans un répertoire reflétant le nom du package spécifié. Par exemple, le fichier `Foo.java` définissant un seul publique `Foo` classe et contenant un nom de paquet de`sub.dir` va générer un correspondant `sub/dir/Foo.class` fichier de classe.

Exemples:

```
env.Java (target = 'classes', source = 'src')
env.Java (target = 'classes', source = [ 'src1', 'src2'])
env.Java (target = 'classes', source = [ 'File1.java', 'File2.java'])
```

Les fichiers source Java peuvent utiliser le codage natif pour le système d' exploitation sous - jacent. Depuis SCons compile en mode ASCII simple par défaut, le compilateur génère des avertissements sur les caractères qui peuvent impossible à cartographier conduire à des erreurs que le fichier est traité plus loin. Dans ce cas, l'utilisateur doit spécifier la `LANG` variable d'environnement pour indiquer au compilateur ce que le codage est utilisé. Pour portibility, il est préférable si l'encodage est codé en dur de sorte que la compilation ne fonctionnera que si elle se fait sur un système avec un codage différent.

```
env = environnement ()
env [ 'ENV'] [ 'LANG'] = 'en_GB.UTF-8'
```

`JavaH()` , `env.JavaH()`

Builds fichiers d' en- tête de C et de source pour la mise en œuvre de méthodes Java natives. La cible peut être un répertoire dans lequel les fichiers d' en- tête seront écrits, ou un nom de fichier d' en- tête qui contiendra toutes les définitions. La source peut être les noms des `.class`fichiers, les noms des `.java`fichiers compilés dans des `.class`fichiers en appelant la `Java`méthode constructeur, ou les objets renvoyés par la `Java` méthode constructeur.

Si la variable de la construction `$JAVACLASSDIR` est définie, que ce soit dans l'environnement ou dans l'appel à la `JavaH` méthode constructeur lui - même, la valeur de la variable sera retirée du début des `.class`noms de fichiers.

Exemples:

```
# Construit java_native.h
les classes = env.Java (target = 'classdir', source = 'src')
env.JavaH (target = 'java_native.h', la source des classes =)

# Construit include / package_foo.h et include / package_bar.h
env.JavaH (target = 'comprennent',
          source = [ 'paquet / foo.class', 'package / bar.class'])

# Construit l'exportation / foo.h et à l'exportation / bar.h
env.JavaH (target = 'export',
          source = [ 'des classes / foo.class', 'cours / bar.class'],
          JAVACLASSDIR = 'classes')
```

`Library()` , `env.Library()`

Synonyme de la `StaticLibrary` méthode constructeur.

`LoadableModule()` , `env.LoadableModule()`

Sur la plupart des systèmes, c'est le même que `SharedLibrary`. Sur les plates - formes Mac OS X (Darwin), ce qui crée un faisceau de module chargeable.

`M4()` , `env.M4()`

Construit un fichier de sortie à partir d' un fichier d'entrée M4. Celui - ci utilise une valeur par défaut `$M4FLAGS`valeur `-E`qui tient compte de tous les avertissements être fatale et qui arrête le premier avertissement lorsque vous utilisez la version GNU de m4. Exemple:

```
env.M4 (target = 'foo.c', source = 'foo.c.m4')
```

`Moc()` , `env.Moc()`

Construit un fichier de sortie à partir d' un fichier d'entrée moc. Les fichiers d'entrée Moc sont des fichiers d' en- tête ou des fichiers Cxx. Ce constructeur est disponible après avoir utilisé l'outil « qt ». Voir la $QTDIRvariable pour plus d' informations. Exemple:

```
env.Moc ( 'foo.h') # génère moc_foo.cc
env.Moc ( 'foo.cpp') # génère foo.moc
```

`MOFiles()` , `env.MOFiles()`

Ce constructeur appartient à l' msgfmtoutil. Le constructeur compile des `po`fichiers vers des `mo`fichiers.

Exemple 1 . Créer`pl.mo`et`en.mo`en compilant `pl.po`et`en.po`:

```
# ...
env.MOFiles ([ 'j', 'en'])
```

Exemple 2 . Compilerfichiers pourlangues définies dans`LINGUAS`fichier:

```
# ...
env.MOFiles (LINGUAS_FILE = 1)
```

Exemple 3 . Créer`pl.mo`et`en.mo`en compilant`pl.po`et`en.po`plusfichiers pourlangues définies dans`LINGUAS`fichier:

```
# ...
env.MOFiles ([ 'j', 'en'], LINGUAS_FILE = 1)
```

Exemple 4 . Compilerfichiers pourlangues définies dans`LINGUAS`fichier (autre version):

```
# ...
env [ 'LINGUAS_FILE'] = 1
env.MOFiles ()
```

`MSVSProject()` , `env.MSVSProject()`

Construit un fichier de projet Microsoft Visual Studio, et par défaut construit un fichier de solution ainsi.

On construit ainsi un fichier de projet Visual Studio, basé sur la version de Visual Studio qui est configuré (soit la dernière version installée, ou la version spécifiée par $MSVS_VERSIONdans le constructeur de l' environnement). Pour Visual Studio 6, il va générer un `.dsp`fichier. Pour Visual Studio 7 (.NET) et les versions ultérieures, il va générer un `.vcproj`fichier.

Par défaut, cela génère également un fichier de solution pour le projet spécifié, un `.dsw`fichier pour Visual Studio 6 ou un`.sln`fichier pour Visual Studio 7 (.NET). Ce comportement peut être désactivé en spécifiant `auto_build_solution=0`lorsque vous appelez `MSVSProject`, dans ce cas , vous voulez sans doute pour construire le fichier de solution (s) en appelant le `MSVSSolution`constructeur (voir ci - dessous).

Le `MSVSProject`constructeur prend plusieurs listes de noms de fichiers à placer dans le dossier du projet. Ceux - ci sont actuellement limités à `srcs`, `incs`, `localincs`, `resources`et `misc`. Ceux - ci sont assez explicites, mais il convient de noter que ces listes sont ajoutées à la $SOURCESvariable de construction sous forme de chaînes, non pas comme SCons nœuds de fichiers. En effet , ils représentent les noms de fichiers à ajouter au fichier de projet, et non pas les fichiers source utilisés pour construire le fichier de projet.

Les listes de noms de fichiers ci-dessus sont facultatifs, mais au moins un doit être spécifié pour le fichier de projet résultant d'être vide.

En plus des listes ci-dessus des valeurs, peuvent être spécifiées les valeurs suivantes:

cible

Le nom de la cible `.dsp`ou `.vcproj`fichier. Le suffixe correct doit être utilisé la version de Visual Studio, mais la $MSVSPROJECTSUFFIX variable de construction sera définie à la valeur correcte (voir exemple ci - dessous).

une variante

Le nom de cette variante particulière. Pour Visual Studio 7 projets, cela peut aussi être une liste de noms de variantes. Ce sont généralement des choses comme « Debug » ou « Release », mais vraiment peut être tout ce que vous voulez. Pour Visual Studio 7 projets, ils peuvent également spécifier une plate - forme cible séparée du nom de variante par un `|` caractère (tube vertical): `Debug|Xbox`. La plate - forme cible par défaut est Win32. Plusieurs appels à `MSVSProject`différentes variantes sont autorisées; toutes les variantes seront ajoutés au fichier de projet avec leurs objectifs et sources de construction appropriées.

cmdargs

Arguments de ligne de commande supplémentaires pour les différentes variantes. Le nombre d' `cmdargs`entrées doit correspondre au nombre d' `variant`entrées, ou être vide (non spécifié). Si vous donnez une seule, il sera automatiquement propagée à toutes les variantes.

buildtarget

Une chaîne en option, noeud, ou la liste des chaînes ou des noeuds (une par variante de construction), de dire le débogueur Visual Studio quelle cible sortie à utiliser dans ce que la variante de construction. Le nombre d'`buildtarget`entrées doit correspondre au nombre d'`variant`entrées.

runfile

Le nom du fichier que Visual Studio 7 et plus tard se déroulera et le débogage. Cela apparaît comme la valeur du`Output`champ dans le fichier de projet Visual Studio résultant. Si ce n'est pas spécifié, la valeur par défaut est la même que celle spécifiée `buildtarget`valeur.

Notez que parce SCons exécute toujours son commandes de construction à partir du répertoire dans lequel le `SConstruct`fichier est situé, si vous créez un fichier de projet dans un autre répertoire que le `SConstruct` répertoire, les utilisateurs ne seront pas en mesure de double-cliquer sur le nom du fichier dans les messages d'erreur de compilation affiché dans la fenêtre de sortie de la console Visual studio. Cela peut être résolu en ajoutant Visual C / C ++ `/FC` option de compilateur à la $CCFLAGSvariable de sorte que le compilateur imprime le nom de chemin complet de tous les fichiers qui provoquent des erreurs de compilation.

Exemple d'utilisation:

```
barsrcs = [ 'bar.cpp'],
barincs = [ 'bar.h'],
barlocalincs = [ 'StdAfx.h']
barresources = [ 'bar.rc', 'resource.h']
barmisc = [ 'bar_readme.txt']

dll = env.SharedLibrary (target = 'bar.dll',
                         source = barsrcs)

env.MSVSProject ( 'Barre' target = + env [ 'MSVSPROJECTSUFFIX'],
                SRC = barsrcs,
                INCS = barincs,
                localincs = barlocalincs,
                = ressources barresources,
                misc = barmisc,
                buildtarget = dll,
                variante = 'Release')
```

A partir de la version 2.4 de SCons il est également possible de spécifier l'argument optionnel *DebugSettings*, ce qui crée des fichiers pour le débogage dans Visual Studio:

DebugSettings

> Un dictionnaire des paramètres de débogage qui sont écrits dans le `.vcproj.user`ou le `.vcxproj.user`fichier, en fonction de la version installée. Comme il est fait pour cmdargs (voir ci - dessus), vous pouvez spécifier un*DebugSettings*dictionnaire par variante. Si vous donnez une seule, il sera propagé à toutes les variantes.

À l' heure actuelle, seul Visual Studio v9.0 et Visual Studio version v11 sont mises en œuvre, pour les autres versions ne fichier est généré. Pour générer le fichier utilisateur, il vous suffit d'ajouter un *DebugSettings*dictionnaire à l'environnement avec les bons paramètres pour votre version VSM. Si le dictionnaire est vide ou ne contient pas de bonne valeur, aucun fichier ne sera généré.

Voici un exemple plus artificiel, impliquant la mise en place d'un projet pour les variantes et DebugSettings:

```
# En supposant que vous stockez vos paramètres par défaut dans un fichier
vars = Variables ( 'variables.py')
msvcver = vars.args.get ( 'vc', '9')

# Vérifiez commande args pour forcer une version Microsoft Visual Studio
si msvcver == '9' ou msvcver == '11':
  env = environnement (MSVC_VERSION = msvcver + ». 0' , MSVC_BATCH = False)
autre:
  env = environnement ()

AddOption ( '- userfile', action = 'store_true', dest = s userfile ', par défaut = False,
          help = "Créer un fichier utilisateur de projet Visual Studio")

#
# 1. Configurez votre dictionnaire de réglage de mise au point avec les options que vous voulez dans la liste
Nombre d'options autorisées, par exemple, si vous voulez créer un fichier utilisateur pour lancer
# Une application spécifique pour tester votre dll avec Microsoft Visual Studio 2008 (v9):
#
V9DebugSettings = {
    'Commande': 'c: \\ monapp \\ \\ en utilisant thisdll.exe',
    'WorkingDirectory': 'c: \\ \\ monapp utilisant \\',
    'CommandArguments': 'Mot de passe -p',
# 'Joindre': 'false',
# 'DebuggerType': '3',
# 'À distance': '1',
# 'RemoteMachine': Aucun,
# 'RemoteCommand': Aucun,
# 'HTTPUrl': Aucun,
# 'PDBPath': Aucun,
# 'SQLDebugging': Aucun,
# 'Environnement': '',
# 'EnvironmentMerge': 'true',
# 'DebuggerFlavor': Aucun,
# 'MPIRunCommand': Aucun,
# 'MPIRunArguments': Aucun,
# 'MPIRunWorkingDirectory': Aucun,
# 'ApplicationCommand': Aucun,
# 'ApplicationArguments': Aucun,
# 'ShimCommand': Aucun,
# 'MPIAcceptMode': Aucun,
# 'MPIAcceptFilter': Aucun,
}

#
# 2. Parce qu'il ya beaucoup d'options différentes en fonction de la Microsoft
# Visual Studio version, si vous utilisez plus d'une version que vous devez
# Définir un dictionnaire par version, par exemple si vous voulez créer un utilisateur
fichier # pour lancer une application spécifique pour tester votre dll avec Microsoft
# Visual Studio 2012 (v11):
#
V10DebugSettings = {
    'LocalDebuggerCommand': 'c: \\ \\ monapp utilisant \\ thisdll.exe',
    'LocalDebuggerWorkingDirectory': 'c: \\ \\ monapp utilisant \\',
    'LocalDebuggerCommandArguments': 'Mot de passe -p',
# 'LocalDebuggerEnvironment': Aucun,
# 'DebuggerFlavor': 'WindowsLocalDebugger',
# 'LocalDebuggerAttach': Aucun,
# 'LocalDebuggerDebuggerType': Aucun,
# 'LocalDebuggerMergeEnvironment': Aucun,
# 'LocalDebuggerSQLDebugging': Aucun,
# 'RemoteDebuggerCommand': Aucun,
# 'RemoteDebuggerCommandArguments': Aucun,
# 'RemoteDebuggerWorkingDirectory': Aucun,
# 'RemoteDebuggerServerName': Aucun,
# 'RemoteDebuggerConnection': Aucun,
# 'RemoteDebuggerDebuggerType': Aucun,
# 'RemoteDebuggerAttach': Aucun,
# 'RemoteDebuggerSQLDebugging': Aucun,
```

```
    # 'DeploymentDirectory': Aucun,
    # 'AdditionalFiles': Aucun,
    # 'RemoteDebuggerDeployDebugCppRuntime': Aucun,
    # 'WebBrowserDebuggerHttpUrl': Aucun,
    # 'WebBrowserDebuggerDebuggerType': Aucun,
    # 'WebServiceDebuggerHttpUrl': Aucun,
    # 'WebServiceDebuggerDebuggerType': Aucun,
    # 'WebServiceDebuggerSQLDebugging': Aucun,
    }

    #
    # 3. Sélectionnez le dictionnaire que vous souhaitez en fonction de la version de Visual Studio
    # Les fichiers que vous voulez générer.
    #
    sinon env.GetOption ( 'userfile'):
        dbgsettings = Aucun
    elif env.get ( 'MSVC_VERSION', None) == '9,0':
        dbgsettings = V9DebugSettings
    Elif env.get ( 'MSVC_VERSION', None) == '11 .0' :
        dbgsettings = V10DebugSettings
    autre:
        dbgsettings = Aucun

    #
    # 4. Ajouter le dictionnaire au mot-clé DebugSettings.
    #
    barsrcs = [ 'bar.cpp', 'dllmain.cpp', 'stdafx.cpp']
    barincs = [ 'targetver.h']
    barlocalincs = [ 'StdAfx.h']
    barresources = [ 'bar.rc', 'resource.h']
    barmisc = [ 'LisezMoi.txt']

    dll = env.SharedLibrary (target = 'bar.dll',
                             source = barsrcs)

    env.MSVSProject ( 'Barre' target = + env [ 'MSVSPROJECTSUFFIX'],
                    SRC = barsrcs,
                    INCS = barincs,
                    localincs = barlocalincs,
                    = ressources barresources,
                    misc = barmisc,
                    buildtarget = [dll [0]] * 2,
                    variante = ( 'Debug | Win32', 'Release | Win32'),
                    cmdargs = 'vc =% s' % msvcver,
                    DebugSettings = (dbgsettings, {}))
```

`MSVSSolution()` , `env.MSVSSolution()`

Construit un fichier de solution Microsoft Visual Studio.

On construit ainsi un fichier de solution Visual Studio, basé sur la version de Visual Studio qui est configuré (soit la dernière version installée, ou la version spécifiée par `$MSVS_VERSION`dans l'environnement de la construction). Pour Visual Studio 6, il va générer un `.dsw`fichier. Pour Visual Studio 7 (.NET), il va générer un `.sln`fichier.

Les valeurs suivantes doivent être spécifiées:

cible

> Le nom de la .dsw cible ou d'un fichier .sln. Le suffixe correct pour la version de Visual Studio doit être utilisé, mais la valeur `$MSVSSOLUTIONSUFFIX`sera définie à la valeur correcte (voir exemple ci - dessous).

une variante

> Le nom de cette variante particulière, ou une liste de noms de variantes (celui-ci est pris en charge pour SVSM 7 solutions). Ce sont généralement des choses comme « Debug » ou « Release », mais vraiment peut être tout ce que vous voulez. Pour MSVS 7 ils peuvent également spécifier la plate-forme cible, comme celui-ci « Debug | Xbox ». la plate-forme par défaut est Win32.

projets

> Une liste de noms de fichiers de projet, ou les noeuds du projet renvoyés par les appels au `MSVSProject`constructeur, à placer dans le fichier de solution. Il convient de noter que ces noms de fichiers ne sont pas ajoutés aux sources de variable d'environnement $ sous forme de fichiers, mais plutôt sous forme de chaînes. En effet , ils représentent les noms de fichiers à ajouter au fichier de solution, et non les fichiers source utilisés pour construire le fichier de solution.

Exemple d'utilisation:

```
env.MSVSSolution ( 'Barre' target = + env [ 'MSVSSOLUTIONSUFFIX'], les projets = [ 'bar'
+ Env [ 'MSVSPROJECTSUFFIX']], variante = 'Release')
```

`Object()` , `env.Object()`

Synonyme de la `StaticObject` méthode constructeur.

`Package()` , `env.Package()`

Construit un paquet binaire des fichiers source données.

```
env.Package (source = FindInstalledFiles ())
```

Paquets de distribution builds de logiciels. Paquets se composent de fichiers à installer et des informations d'emballage. Le premier peut être spécifié avec le `source`paramètre et peut être omis, dans ce cas , la `FindInstalledFiles`fonction recueillera tous les fichiers qui ont un `Install`ou `InstallAs`constructeur attaché. Si l' `target`est pas spécifié , il sera déduit des informations complémentaires fournies à ce constructeur.

Les informations d'emballage est spécifié à l'aide de variables de construction documentées ci - dessous. Cette information est appelée une étiquette de souligner que certains d'entre eux peuvent également être attachés à des fichiers avec la `Tag`fonction. Ceux obligatoires se plaignent si elles ne sont pas précisés. Ils varient en fonction emballeur cible choisie.

Le conditionneur de cible peut être sélectionné avec l'option de ligne de commande « de forfait » ou avec la `$PACKAGETYPE`variable de construction. Actuellement , les emballeurs suivantes disponibles:

* Msi - Microsoft Installer * rpm - Redhat Package Manager * ipkg - Itsy système de gestion des paquets * tarbz2 - goudron comprimé * targz - tar compressé * zip - fichier zip * src_tarbz2 - source de goudron comprimé * src_targz - source de goudron comprimé * src_zip - fichier zip la source

Une liste mise à jour est toujours disponible sous l'option « package_type » lors de l'exécution « scons --help » sur un projet qui a activé emballages.

```
env = environnement (outils = [ 'default', 'emballage'])
env.Install ( '/ bin /', 'mon_programme')
env.Package (NAME = 'foo',
             VERSION = '1.2.3',
             PackageVersion = 0,
             = 'Rpm de forfait',
             LICENCE = 'gpl',
             RESUME = 'balalalalal',
             DESCRIPTION = « cela devrait être vraiment très long »,
             X_RPM_GROUP = 'Application / fu',
             SOURCE_URL = 'http://foo.org/foo-1.2.3.tar.gz'
           )
```

`PCH()` , `env.PCH()`

Construit un en-tête de Visual C ++ précompilé Microsoft. L'appel de cette méthode constructeur renvoie une liste de deux objectifs: la PCH comme premier élément, et le fichier objet comme le deuxième élément. Normalement, le fichier objet est ignoré. Cette méthode de constructeur est fourni uniquement lorsque ++ Microsoft Visual C est utilisé comme le compilateur. La méthode de constructeur PCH est généralement utilisé en conjonction avec la variable de construction PCH pour forcer les fichiers d'objets à utiliser l'en-tête précompilé:

```
env [PCH '] = env.PCH ( 'StdAfx.cpp') [0]
```

`PDF()` , `env.PDF()`

Construit un `.pdf`fichier à partir d' un `.dvi`fichier d'entrée (ou, par extension, un `.tex`, `.ltx`ou `.latex`fichier d'entrée). Le suffixe spécifié par la `$PDFSUFFIX`variable de construction (`.pdf`par défaut) est automatiquement ajouté à la cible si elle est pas déjà présent. Exemple:

```
# Construit à partir aaa.tex
env.PDF (target = 'aaa.pdf', source = 'aaa.tex')
# Construit bbb.pdf de bbb.dvi
env.PDF (target = 'bbb', source = 'bbb.dvi')
```

`POInit()` , `env.POInit()`

Ce constructeur appartient à l' `msginit`outil. Le constructeur initialise manquant `PO`fichier (s) si `$POAUTOINIT`est réglé. Si`$POAUTOINIT`est pas encore défini (par défaut), l' `POInit`instruction imprime pour l' utilisateur (qui est censé être un traducteur), indiquant comment le `PO`doit être initialisé fichier. Dans les projets normaux *vous ne devriez pas utiliser `POInit`et utiliser à la `POUpdate` place* . `POUpdate`choisit intelligemment entre **msgmerge (1)** et **msginit (1)** . `POInit` utilise toujours **msginit (1)** et doit être considéré comme constructeur à des fins particulières ou pour une utilisation temporaire (par exemple pour l' initialisation rapide, une fois d'un tas de `PO`fichiers) ou pour les tests.

Nœuds cibles définies par `POInit`ne sont pas construits par défaut (ils sont `Ignore`d du `'.'`nœud) , mais sont ajoutés spéciale `Alias`(`'po-create'`par défaut). Le nom d'alias peut être modifié par la `$POCREATE_ALIAS` variable de construction. Tous les `PO`fichiers définis par `POInit`peuvent facilement être initialisés par **scons po-create** .

Exemple 1 . Initialiser`en.po`et`pl.po`de `messages.pot`:

```
# ...
env.POInit ([ 'fr', 'pl']) # messages.pot -> [en.po, pl.po]
```

Exemple 2 . Initialiser`en.po`et`pl.po`de `foo.pot`:

```
# ...
env.POInit ([ 'fr', 'pl'], [ 'foo']) # foo.pot -> [en.po, pl.po]
```

Exemple 3 . Initialiser`en.po`et`pl.po`de`foo.pot`mais utilisant`$POTDOMAIN`variable de construction:

```
# ...
env.POInit ([ 'fr', 'pl'], POTDOMAIN = 'foo') # foo.pot -> [en.po, pl.po]
```

Exemple 4 . Initialiser`PO`fichiers pour langues définies dans`LINGUAS`fichier. Les fichiers seront initialisées partir modèle`messages.pot`:

```
# ...
env.POInit (LINGUAS_FILE = 1) fichier de LINGUAS 'a besoin #
```

Exemple 5 . Initialiser`en.po`et`pl.pl` `PO`fichiers ainsi fichiers pour langues définies dans `LINGUAS`fichier. Les fichiers seront initialisées partir modèle `messages.pot`:

```
# ...
env.POInit ([ 'fr', 'j'], LINGUAS_FILE = 1)
```

Exemple 6 . Vous pouvez préconfigurer votre environnement abord, puis initialiser `PO`fichiers:

```
# ...
env [ 'POAUTOINIT'] = 1
env [ 'LINGUAS_FILE'] = 1
env [ 'POTDOMAIN'] = 'foo'
env.POInit ()
```

qui a la même efect:

```
# ...
env.POInit (POAUTOINIT = 1, LINGUAS_FILE = 1, POTDOMAIN = 'foo')
```

`PostScript()`, `env.PostScript()`

Construit un `.ps`fichier à partir d' un `.dvi`fichier d'entrée (ou, par extension, un `.tex`, `.ltx`ou `.latex`fichier d'entrée). Le suffixe spécifié par la `$PSSUFFIX`variable de construction (`.ps`par défaut) est automatiquement ajouté à la cible si elle est pas déjà présent. Exemple:

```
# Construit à partir aaa.tex
env.PostScript (cible = 'aaa.ps', source = 'aaa.tex')
# Construit bbb.ps de bbb.dvi
env.PostScript (target = 'bbb', source = 'bbb.dvi')
```

`POTUpdate()`, `env.POTUpdate()`

Le constructeur appartient à l' `xgettext`outil. Les mises à jour de constructeur cible `POT`fichier si elle existe ou crée une si elle ne fonctionne pas. Le noeud ne se construit pas par défaut (il est `Ignore`d à partir `'.'`), mais seulement sur demande (lorsque donné `POT`fichier est requis ou lorsque alias spécial est invoqué). Ce générateur ajoute son nœud targe (`messages.pot`, par exemple) à un alias spécial (`pot-update`par défaut, voir `$POTUPDATE_ALIAS`) de sorte que vous pouvez mettre à jour / créer facilement avec les**scons pot de mise à jour** . Le fichier est pas écrit jusqu'à ce qu'il n'y a pas de changement réel dans les messages internationalisés (ou dans les commentaires qui entrent dans le `POT`fichier).

> ### Remarque
>
> Vous pouvez voir **xgettext (1)** invoqué par l'`xgettext`outil , même s'il n'y a pas de changement réel dans les messages internationalisés (de sorte que le `POT`fichier est pas mis à jour). Cela se produit chaque fois qu'un fichier source a changé. Dans ce cas , nous invoquons **xgettext (1)** et comparer sa sortie avec le contenu du `POT`fichier pour déterminer si le fichier doit être mis à jour ou non.

Exemple 1. Créons`po/`répertoire et lieu suivant `SConstruct`il script:

```
# SConstruct dans 'po /' subdir
env = environnement (outils = [ 'default', 'xgettext'])
env.POTUpdate ([ 'foo'] [ '../a.cpp', '../b.cpp'])
env.POTUpdate ([ 'bar'] [ '../c.cpp', '../d.cpp'])
```

Puis invoquer scons quelques fois:

```
user @ host: $ de la # ne crée pas foo.pot ni bar.pot
user @ host: $ scons foo.pot # Mises à jour ou crée foo.pot
user @ host: pot mise à jour de $ # Les mises à jour ou crée foo.pot et bar.pot
user @ host: # -c de $ ne pas foo.pot ni bar.pot propre.
```

les résultats sont comme les commentaires disent ci-dessus.

Exemple 2. Le`POTUpdate`constructeur peut être utilisé sans cible spécifiée, auquel cas cible par défaut`messages.pot`sera utilisé. La cible par défaut peut être annulé en définissant`$POTDOMAIN`variable de construction oufournir comme une dérogation au`POTUpdate`constructeur:

```
# script SConstruct
env = environnement (outils = [ 'default', 'xgettext'])
env [ 'POTDOMAIN'] = "toto"
env.POTUpdate (source = [ "a.cpp", "b.cpp"]) # Crée foo.pot ...
env.POTUpdate (POTDOMAIN = "bar", source = [ "c.cpp", "d.cpp"]) # et bar.pot
```

Exemple 3. Les sources peut être spécifié dansfichier séparé, par exemple `POTFILES.in`:

```
# POTFILES.in dans 'po /' sous-répertoire
../a.cpp
../b.cpp
# Fin du fichier
```

Le nom du fichier (`POTFILES.in`) contenant la liste des sources est fournie par `$XGETTEXTFROM`:

```
# Fichier SConstruct dans 'po /' sous-répertoire
env = environnement (outils = [ 'default', 'xgettext'])
env.POTUpdate (XGETTEXTFROM = 'POTFILES.in')
```

Exemple 4. Vous pouvez utiliser`$XGETTEXTPATH`pour définirchemin de recherche desource. Supposons, par exemple, que vous avezfichiers`a.cpp`, `b.cpp`,`po/SConstruct`, `po/POTFILES.in`. Ensuitevos`POT`fichiers pourraient secomme concernantPI cidessous:

```
# POTFILES.in dans 'po /' sous-répertoire
a.cpp
b.cpp
# Fin du fichier

# Fichier SConstruct dans 'po /' sous-répertoire
env = environnement (outils = [ 'default', 'xgettext'])
env.POTUpdate (XGETTEXTFROM = 'POTFILES.in', XGETTEXTPATH = '.. /')
```

Exemple 5. répertoires de recherche multiples peuvent être définis dans une liste,exemple `XGETTEXTPATH = ['dir1', 'dir2', ...]`. L'ordre dans la liste détermine l'ordre de recherche des fichiers source. Le chemin vers le premier fichier trouvé est utilisé.

Créons `0/1/po/SConstruct`script:

```
# Fichier SConstruct dans '0/1 / po /' sous-répertoire
env = environnement (outils = [ 'default', 'xgettext'])
env.POTUpdate (XGETTEXTFROM = 'POTFILES.in', XGETTEXTPATH = [ '../', '../../'])
```

et `0/1/po/POTFILES.in`:

```
# POTFILES.in dans '0/1 / po /' sous-répertoire
a.cpp
# Fin du fichier
```

Écrire deux `*.cpp`fichiers, le premier est `0/a.cpp`:

```
/ * 0 / a.cpp * /
gettext ( "Bonjour de ../../a.cpp")
```

et le second est `0/1/a.cpp`:

```
/ * 0/1 / a.cpp * /
gettext ( "Bonjour de ../a.cpp")
```

puis exécutez scons. Vous obtiendrez `0/1/po/messages.pot`le message `"Hello from ../a.cpp"`. Lorsque vous inversez l' ordre dans `$XGETTEXTFOM`, par exemple lorsque vous écrivez SConscript comme

```
# Fichier SConstruct dans '0/1 / po /' sous-répertoire
env = environnement (outils = [ 'default', 'xgettext'])
env.POTUpdate (XGETTEXTFROM = 'POTFILES.in', XGETTEXTPATH = [ '../../', '../'])
```

alors la `messages.pot`contiendra `msgid "Hello from ../../a.cpp"`ligne et non `msgid "Hello from ../a.cpp"`.

`POUpdate()` , `env.POUpdate()`

Le constructeur appartient à l' `msgmerge`outil. Les mises à jour de constructeur `PO`fichiers avec **msgmerge (1)** , ou manquants initialise `PO`fichiers comme décrit dans la documentation de l' `msginit`outil et le `POInit`constructeur (voir aussi `$POAUTOINIT`). Notez que `POUpdate` *ne pas ajouter à ses objectifs* `po-create`*alias* comme le `POInit` fait.

Noeuds cibles définies par le biais `POUpdate`ne sont pas construits par défaut (ils sont `Ignore`d du `'.'`noeud). , Ils sont automatiquement ajoutés à la place spéciale `Alias` (`'po-update'`par défaut). Le nom d'alias peut être modifié par la `$POUPDATE_ALIAS`variable de construction. Vous pouvez facilement mettre à jour les `PO`fichiers de votre projet par **scons po-mise à jour** .

Exemple 1. Misejour`en.po`et`pl.po`de `messages.pot`modèle (voir aussi`$POTDOMAIN`),supposant que celui plus tard existe ou il y arègle pourconstruire (voir `POTUpdate`):

```
# ...
env.POUpdate ([ 'fr', 'pl']) # messages.pot -> [en.po, pl.po]
```

Exemple 2. Misejour`en.po`et`pl.po`du `foo.pot`modèle:

```
# ...
env.POUpdate ([ 'fr', 'pl'], [ 'foo']) # foo.pot -> [en.po, pl.pl]
```

Exemple 3. Misejour`en.po`et`pl.po`de `foo.pot`(autre version):

```
# ...
env.POUpdate ([ 'fr', 'pl'], POTDOMAIN = 'foo') # foo.pot -> [en.po, pl.pl]
```

Exemple 4. fichiers de misejour pourlangues définies dans`LINGUAS`fichier. Les fichiers sont misjourpartir`messages.pot`modèle:

```
# ...
env.POUpdate (LINGUAS_FILE = 1) fichier de LINGUAS 'a besoin #
```

Exemple 5. Comme cidessus, maismisejour du`foo.pot`modèle:

```
# ...
env.POUpdate (LINGUAS_FILE = 1, source = [ 'foo'])
```

Exemple 6. Misejour`en.po`et`pl.po`plusfichiers pourlangues définies dans`LINGUAS`fichier. Les fichiers sont misjourpartir`messages.pot`modèle:

```
# Produits 'en.po', 'pl.po' + fichiers définis dans 'LINGUAS':
env.POUpdate ([ 'fr', 'j'], LINGUAS_FILE = 1)
```

Exemple 7. Utilisez`$POAUTOINIT`pour initialiser automatiquement`PO`fichier si elle n'existe pas:

```
# ...
env.POUpdate (LINGUAS_FILE = 1, POAUTOINIT = 1)
```

Exemple 8. Mettrejour`PO`fichiers pourlangues définies dans`LINGUAS`fichier. Les fichiers sont misjourpartir `foo.pot`modèle. Tousréglages nécessaires sont pré-configurés viaenvironnement.

```
# ...
env [ 'POAUTOINIT'] = 1
env [ 'LINGUAS_FILE'] = 1
env [ 'POTDOMAIN'] = 'foo'
env.POUpdate ()
```

`Program()` , `env.Program()`

Construit un exécutable donné un ou plusieurs fichiers d'objets ou C, C ++, D, ou des fichiers source FORTRAN. En cas de C, C ++, D ou les fichiers source Fortran sont spécifiés, ils seront automatiquement compilés aux fichiers objet à l' aide de la `Object`méthode constructeur; voir que la description de la méthode constructeur pour une liste des suffixes de fichiers source juridique et la façon dont elles sont interprétées. Le préfixe de fichier exécutable cible (spécifié par la `$PROGPREFIX`variable de construction, rien par défaut) et le suffixe (spécifié par la `$PROGSUFFIX`variable de la construction, par défaut, `.exe`sur les systèmes Windows, rien sur les systèmes POSIX) sont automatiquement ajoutés à la cible si pas déjà présent. Exemple:

```
env.Program (target = 'foo', source = [ 'foo.o', 'bar.c', 'baz.f'])
```

`RES()` , `env.RES()`

Construit un fichier de ressources Visual C ++ Microsoft. Cette méthode constructeur est fournie uniquement lorsque vous utilisez comme le compilateur Microsoft Visual C ++ ou MinGW. Le `.res` (ou `.o` pour MinGW) suffixe est ajouté au nom de la cible si aucun autre suffixe est donné. Le fichier source est analysé pour les dépendances implicites comme si elle était un fichier C. Exemple:

```
env.RES ( 'resource.rc')
```

`RMIC()` , `env.RMIC()`

Builds fichiers stub et squelette de classe pour les objets distants de Java `.class`fichiers. La cible est un répertoire par rapport auquel les fichiers de classe stub et squelette seront écrits. La source peut être les noms des `.class`fichiers, ou les objets retour de la `Java` méthode constructeur.

Si la variable de la construction `$JAVACLASSDIR` est définie, que ce soit dans l'environnement ou dans l'appel à la `RMIC` méthode constructeur lui - même, la valeur de la variable sera retirée du début des `.class` noms de fichiers.

```
les classes = env.Java (target = 'classdir', source = 'src')
env.RMIC (target = 'outdir1', la source des classes =)

env.RMIC (target = 'outdir2',
         source = [ 'paquet / foo.class', 'package / bar.class'])

env.RMIC (target = 'outdir3',
         source = [ 'des classes / foo.class', 'cours / bar.class'],
         JAVACLASSDIR = 'classes')
```

`RPCGenClient()` , `env.RPCGenClient()`

Génère un relais de client RPC (`_clnt.c`fichiers) à partir d' un RPC spécifié (`.x`fichier source). Parce que rpcgen construit uniquement les fichiers de sortie dans le répertoire local, la commande sera exécutée dans le répertoire du fichier source par défaut.

```
# Builds src / rpcif_clnt.c
env.RPCGenClient ( 'src / rpcif.x')
```

`RPCGenHeader()` , `env.RPCGenHeader()`

Génère un en- tête de RPC (`.h`fichiers) à partir d' un RPC spécifié (`.x`fichier source). Parce que rpcgen construit uniquement les fichiers de sortie dans le répertoire local, la commande sera exécutée dans le répertoire du fichier source par défaut.

```
# Builds src / rpcif.h
env.RPCGenHeader ( 'src / rpcif.x')
```

`RPCGenService()` , `env.RPCGenService()`

Génère un squelette serveur-RPC (`_svc.c`fichiers) à partir d' un RPC spécifié (`.x`fichier source). Parce que rpcgen construit uniquement les fichiers de sortie dans le répertoire local, la commande sera exécutée dans le répertoire du fichier source par défaut.

```
# Builds src / rpcif_svc.c
env.RPCGenClient ( 'src / rpcif.x')
```

`RPCGenXDR()` , `env.RPCGenXDR()`

Génère un sous - programme de RPC XDR (`_xdr.c`fichiers) à partir d' un RPC spécifié (`.x`fichier source). Parce que rpcgen construit uniquement les fichiers de sortie dans le répertoire local, la commande sera exécutée dans le répertoire du fichier source par défaut.

```
# Builds src / rpcif_xdr.c
env.RPCGenClient ( 'src / rpcif.x')
```

`SharedLibrary()` , `env.SharedLibrary()`

Construit une bibliothèque partagée (`.so`sur un système POSIX,`.dll`sous Windows) donné un ou plusieurs fichiers d'objet ou C, C ++, les fichiers source D ou Fortran. Si les fichiers source sont donnés, ils seront automatiquement compilés aux fichiers objet. Le préfixe de la bibliothèque statique et le suffixe (le cas échéant) sont automatiquement ajoutés à la cible. Le préfixe de fichier de bibliothèque cible (spécifiée par la `$SHLIBPREFIX`variable de la construction, par défaut, `lib`sur les systèmes POSIX, rien sur les systèmes Windows) et le suffixe (spécifié par la `$SHLIBSUFFIX`variable de la construction, par défaut, `.dll`sur les systèmes Windows, `.so`sur les systèmes POSIX) sont automatiquement ajoutés à la cible si pas déjà présent. Exemple:

```
env.SharedLibrary ( 'bar' target =, source = [ 'bar.c', 'foo.o'])
```

Sur les systèmes Windows, la `SharedLibrary` méthode constructeur sera toujours construire une importation (`.lib`bibliothèque) , en plus de la commune (`.dll`bibliothèque), l' ajout d' une `.lib`bibliothèque avec le même nom de base s'il n'y a pas déjà un `.lib`fichier explicitement dans la liste des cibles.

Sur les systèmes Cygwin, la `SharedLibrary` méthode constructeur sera toujours construire une importation (`.dll.a`bibliothèque) , en plus de la commune (`.dll`bibliothèque), l' ajout d' une `.dll.a`bibliothèque avec le même nom de base s'il n'y a pas déjà un `.dll.a`fichier explicitement dans la liste des cibles.

Tous les fichiers d'objets répertoriés dans le `source` moût ont été construits pour une bibliothèque partagée (qui est, en utilisant la`SharedObject` méthode de constructeur). `scons` soulèvera une erreur en cas de non - concordance.

Sur certaines plateformes, il y a une distinction entre une bibliothèque partagée (chargée automatiquement par le système pour résoudre les références externes) et un module chargeable (explicitement chargé par action de l' utilisateur). Pour une portabilité maximale, utilisez le `LoadableModule`constructeur pour ce dernier.

Lorsque la `$SHLIBVERSION`variable de construction est définie une bibliothèque partagée versionné est créée. Cela modifie le `$SHLINKFLAGS`cas échéant, ajoute le numéro de version au nom de la bibliothèque, et crée les liens symboliques nécessaires.

```
env.SharedLibrary ( 'barre' target =, source = [ 'bar.c', 'foo.o'], SHLIBVERSION = '1.5.2')
```

Sur un système POSIX, les versions avec un seul jeton créer exactement un lien symbolique: libbar.so.6 aurait des liens symboliques libbar.so seulement. Sur un système POSIX, les versions avec deux ou plusieurs jetons créer exactement deux liens symboliques: libbar.so.2.3.1 aurait des liens symboliques libbar.so et libbar.so.2; sur un système Darwin (OSX) la bibliothèque serait libbar.2.3.1.dylib et le lien serait libbar.dylib.

Sur les systèmes Windows, en précisant `register=1` provoquera le `.dll`à enregistrer après sa construction en utilisant REGSVR32. La commande qui est exécutée (« regsvr32 » par défaut) est déterminée par la `$REGSVR`variable de la construction, et les drapeaux transmis sont déterminés par `$REGSVRFLAGS`. Par défaut, `$REGSVRFLAGS`inclut l' `/s`option pour empêcher les boîtes de dialogue de sauter et nécessitant l' attention des utilisateurs lorsqu'il est exécuté. Si vous modifiez `$REGSVRFLAGS`, assurez - vous d'inclure l' `/s`option. Par exemple,

```
env.SharedLibrary (target = 'bar',
                   source = [ 'bar.cxx', 'foo.obj'],
```

```
                    enregistrer = 1)
```

enregistrera `bar.dll`comme un objet COM lorsqu'il est fait lier.

`SharedObject()` , `env.SharedObject()`

Construit un fichier objet à inclure dans une bibliothèque partagée. Les fichiers sources doivent avoir une du même ensemble d'extensions spécifiées ci - dessus pour la `StaticObject`méthode constructeur. Sur certaines plates - formes de construction d' un objet partagé nécessite l' option de compilateur supplémentaire (par exemple `-fPIC`pour gcc) en plus de ceux nécessaires à la construction d' un objet normal (statique), mais sur certaines plates - formes , il n'y a aucune différence entre un objet partagé et une normale (statique). Quand il y a une différence, SCons n'autoriserons que des objets partagés à lier dans une bibliothèque partagée, et utilisera un suffixe différent pour les objets partagés. Sur les plates - formes où il n'y a pas de différence, SCons permettra à la fois des objets normaux (statique) et partagé à lier dans une bibliothèque partagée, et utilisera le même suffixe partagées et des objets normaux (statiques). Le préfixe de fichier de l' objet cible (spécifié par la `$SHOBJPREFIX`variable de construction, par défaut, le même que `$OBJPREFIX`) et le suffixe (spécifié par la `$SHOBJSUFFIX`variable de construction) sont automatiquement ajoutés à la cible si pas déjà présent. Exemples:

```
env.SharedObject (target = 'ddd', source = 'ddd.c')
env.SharedObject (target = 'eee.o', source = 'eee.cpp')
env.SharedObject (target = 'fff.obj', source = 'fff.for')
```

Notez que les fichiers source seront analysés en fonction des correspondances de suffixe dans l' `SourceFileScanner` objet. Voir la section « Objets du scanner » ci - dessous pour plus d' informations.

`StaticLibrary()` , `env.StaticLibrary()`

Construit une bibliothèque statique donné un ou plusieurs fichiers d'objet ou C, C ++, les fichiers source D ou Fortran. Si les fichiers source sont donnés, ils seront automatiquement compilés aux fichiers objet. Le préfixe de la bibliothèque statique et le suffixe (le cas échéant) sont automatiquement ajoutés à la cible. Le préfixe de fichier de bibliothèque cible (spécifiée par la `$LIBPREFIX`variable de la construction, par défaut, `lib`sur les systèmes POSIX, rien sur les systèmes Windows) et le suffixe (spécifié par la `$LIBSUFFIX`variable de la construction, par défaut, `.lib`sur les systèmes Windows, `.a`sur les systèmes POSIX) sont automatiquement ajoutés à la cible si pas déjà présent. Exemple:

```
env.StaticLibrary ( 'bar' target =, source = [ 'bar.c', 'foo.o'])
```

Tous les fichiers d'objets répertoriés dans le `source` moût ont été construits pour une bibliothèque statique (qui est, en utilisant la`StaticObject` méthode de constructeur). `scons` soulèvera une erreur en cas de non - concordance.

`StaticObject()` , `env.StaticObject()`

Construit un fichier objet statique d'un ou plusieurs C, C ++, D, ou les fichiers source Fortran. Les fichiers sources doivent avoir l'une des extensions suivantes:

```
  fichier langage assembleur asm
  fichier langage assembleur .ASM
  .c C
  .C de Windows: fichier C
          Posix: C ++ fichier
  dossier de .cc C
  dossier de cpp C
  dossier de .cxx C
  dossier de .cxx C
  .c ++ C ++ fichier
  .C ++ C ++ fichier
  fichier .d D
  .F fichier Fortran
  .F de Windows: fichier Fortran
          POSIX: fichier Fortran + C pré-processeur
  .pour fichier Fortran
  fichier .FOR Fortran
  .fpp fichier Fortran + C pré-processeur
  .FPP fichier Fortran + C pré-processeur
  fichier .m objet C
  fichier .mm objet C ++
  fichier langage assembleur .s
  .S de Windows: fichier de langage assembleur
          ARM: CodeSourcery Sourcery Lite
  .sx fichier en langage assembleur + C pré-processeur
          Posix: fichier de langage assembleur + C pré-processeur
  .SPP fichier en langage assembleur + C pré-processeur
  .SPP fichier en langage assembleur + C pré-processeur
```

Le préfixe de fichier objet cible (spécifié par la `$OBJPREFIX`variable de construction, rien par défaut) et le suffixe (spécifié par la `$OBJSUFFIX`variable de la construction, `.obj`sur les systèmes Windows, `.o`sur les systèmes POSIX) sont automatiquement ajoutés à la cible si pas déjà présent. Exemples:

```
env.StaticObject (target = 'aaa', source = 'aaa.c')
env.StaticObject (target = 'bbb.o', source = 'bbb.c ++')
env.StaticObject (target = 'ccc.obj', source = 'ccc.f')
```

Notez que les fichiers source seront analysés en fonction des correspondances de suffixe dans l' `SourceFileScanner` objet. Voir la section « Objets du scanner » ci - dessous pour plus d' informations.

`Substfile()` , `env.Substfile()`

Le `Substfile`constructeur crée un fichier texte à partir d' un autre fichier ou un ensemble de fichiers en les concaténer avec `$LINESEPARATOR`et remplacer le texte en utilisant la `$SUBST_DICT`variable de construction. Autres listes de fichiers source sont aplaties. Voir aussi `Textfile`.

Si un seul fichier source est présent avec un `.in`suffixe, le suffixe est dépouillé et le reste est utilisé comme nom de cible par défaut.

Le préfixe et le suffixe spécifié par les `$SUBSTFILEPREFIX` et les `$SUBSTFILESUFFIX`variables de construction (la chaîne vide par défaut dans les deux cas) sont automatiquement ajoutés à la cible si elles ne sont pas déjà présents.

Si une variable de construction nommée `$SUBST_DICT`est présent, il peut être soit un dictionnaire Python ou une séquence de (clé, valeur) tuples. S'il est un dictionnaire , il est converti en une liste de tuples dans un ordre arbitraire, donc si une clé est un préfixe d' une autre clé ou si

une substitution pourrait être encore renforcé par un autre subsitition, il est imprévisible si l'expansion se produira.

Toutes les occurrences d'une clé dans la source sont remplacées par la valeur correspondante, qui peut être une fonction Python appelable ou une chaîne. Si la valeur est appelable, il est appelé sans argument pour obtenir une chaîne. Les chaînes sont *Subst* -expanded et le résultat remplace la clé.

```
env = environnement (outils = [ 'default', 'textfile'])

env [ 'préfixe'] = '/ usr / bin'
script_dict = { 'préfixe @ @': '/ bin', @ exec_prefix @: '$ prefix'}
env.Substfile ( 'script.in', SUBST_DICT = script_dict)

conf_dict = { '% VERSION%': '1.2.3', '% BASE%': 'MONPROG'}
env.Substfile ( 'config.h.in', conf_dict, SUBST_DICT = conf_dict)

# IMPRÉVISIBLE - une clé est un préfixe d'un autre
bad_foo = { '$ foo': '$ foo', 'foobar $': 'foobar $'}
env.Substfile ( 'foo.in', SUBST_DICT = bad_foo)

# PRÉVISIBLES - clés sont appliquées plus longue première
good_foo = [( 'foobar $', 'foobar $'), ( '$ foo', '$ foo')]
env.Substfile ( 'foo.in', SUBST_DICT = good_foo)

# IMPRÉVISIBLE - une substitution pourrait être élargi futher
bad_bar = { '@ bar @': '@ savon @', '@ savon @': Lye '}
env.Substfile ( 'bar.in', SUBST_DICT = bad_bar)

# PRÉVISIBLES - substitutions sont développées pour
good_bar = (( '@ bar @', '@ @ savon'), ( '@ @ savon', 'lye'))
env.Substfile ( 'bar.in', SUBST_DICT = good_bar)

# Le SUBST_DICT peut être en commun (et non un remplacement)
} = {substitutions par
= Environnement subst (outils = [ 'textfile'], SUBST_DICT = substitutions)
substitutions 'foo' = [ '@ foo @']
subst [ 'SUBST_DICT'] [ '@ bar @'] 'bar' =
subst.Substfile ( 'pgm1.c', [Valeur ( '# include "@ foo @ .h"'),
                             Valeur ( '# include "@ bar @ .h"'),
                             "Common.in",
                             "Pgm1.in"
                            ])
subst.Substfile ( 'pgm2.c', [Valeur ( '# include "@ foo @ .h"'),
                             Valeur ( '# include "@ bar @ .h"'),
                             "Common.in",
                             "Pgm2.in"
                            ])
```

`Tar()` , `env.Tar()`

Construit une archive tar des fichiers spécifiés et / ou des répertoires. Contrairement à la plupart des méthodes de constructeur, la `Tar` méthode constructeur peut être appelé plusieurs fois pour une cible donnée; chaque appel supplémentaire ajoute à la liste des entrées qui sera construit dans l'archive. Tous les répertoires source seront scannés pour des modifications à tous les fichiers sur le disque, peu importe si oui ou non au `scons` courant les autres d'appels Builder ou fonction.

```
env.Tar ( 'src.tar', 'src')

# Créez le fichier stuff.tar.
env.Tar ( 'stuff', [ 'subdir1', 'subdir2'])
# Ajouter également « une autre » au fichier stuff.tar.
env.Tar ( 'substance', 'autre')

# Set TARFLAGS pour créer une archive filtrée gzip.
env = environnement (TARFLAGS = '-c -Z')
env.Tar ( 'foo.tar.gz', 'foo')

# Affecte également le suffixe .tgz.
env = environnement (TARFLAGS = '-c -Z',
                     TARSUFFIX = 'tgz')
env.Tar ( 'foo')
```

`Textfile()` , `env.Textfile()`

Le `Textfile`constructeur génère un fichier texte. Les chaînes de sources constituent les lignes; sont aplaties listes de sources imbriquées. `$LINESEPARATOR`est utilisé pour séparer les chaînes.

Le cas échéant, la `$SUBST_DICT`variable de construction permet de modifier les chaînes avant qu'elles ne soient écrites; voir la `Substfile`description pour les détails.

Le préfixe et suffixe spécifié par les `$TEXTFILEPREFIX` et les `$TEXTFILESUFFIX`variables de construction (la chaîne vide et `.txt`par défaut, respectivement) sont automatiquement ajoutés à la cible si elles ne sont pas déjà présents. Exemples:

```
# Construit / écrit foo.txt
env.Textfile (target = 'foo.txt', source = [Goethe ', 42 ' Schiller'])

# Construit / écrit bar.txt
env.Textfile (target = 'bar',
              source = [ 'lalala', 'tanteratei'],
              LINESEPARATOR = '| *')

# Listes imbriquées sont aplaties automatiquement
env.Textfile (target = 'blob',
              source = [ 'lalala', [ 'Goethe', 42 'Schiller'], 'tanteratei'])

# Les fichiers peuvent être utilisés comme entrée par les wraping dans le fichier ()
env.Textfile (target = 'concat', les fichiers # concatène avec un marqueur entre
              source = [File ( 'concat1'), Dossier ( 'concat2')],
              LINESEPARATOR = '==================== \ n')

Les résultats sont les suivants:
```

```
foo.txt
  .... 8 <----
  Goethe
  42
  Schiller
  .... 8 <---- (pas linefeed à la fin)

bar.txt:
  .... 8 <----
  lalala | * tanteratei
  .... 8 <---- (pas linefeed à la fin)

blob.txt
  .... 8 <----
  lalala
  Goethe
  42
  Schiller
  tanteratei
  .... 8 <---- (pas linefeed à la fin)
```

`Translate()` , `env.Translate()`

Ce pseudo-builder appartient à gettextToolset. Le constructeur extrait des messages internationalisés de fichiers source, mises à jour POT modèle (si nécessaire), puis les mises à jour des POtraductions (le cas échéant). Si $POAUTOINITest défini, manquants POfichiers seront créés automatiquement (sans intervention de personne traducteur). Les variables $LINGUAS_FILEet $POTDOMAINsont prises en acount aussi. Toutes les autres variables de construction utilisés par POTUpdate, etPOUpdatetravaillent ici aussi.

Exemple 1 . La façonsimple est de spécifierfichiers d'entrée et de sortie languesligne dans un script SCons lorsappelTranslate

```
# SConscript dans le répertoire 'po /'
env = environnement (outils = [ "default", "gettext"])
env [ 'POAUTOINIT'] = 1
env.Translate ([ 'fr', 'j'], [ '../a.cpp','../b.cpp'])
```

Exemple 2 . Si voussouhaitez, vous pouvez également tenir àstyle classique connu de autotools ,savoirutilisationPOTFILES.inetLINGUASfichiers

```
# LINGUAS
en pl
#fin

# POTFILES.in
a.cpp
b.cpp
# fin

# SConscript
env = environnement (outils = [ "default", "gettext"])
env [ 'POAUTOINIT'] = 1
env [ 'XGETTEXTPATH'] = [ '../']
env.Translate (LINGUAS_FILE = 1, XGETTEXTFROM = 'POTFILES.in')
```

La dernière approche est peut - être celle qui est recommandée. Il permet l' internationalisation / localisation facilement divisé sur des scripts SCons séparés, où un script dans l' arborescence source est responsable des traductions (de sources de POfichiers) et l' écriture (s) sous répertoires variantes sont responsables de la compilation des POà des MO fichiers vers et pour l' installation des MOfichiers. Le « facteur de collage » synchroniser ces deux scripts est alors le contenu du LINGUASfichier. Notez que la mise à jourPOTet les POfichiers vont généralement être commis dans le dépôt, ils doivent donc être mis à jour dans le répertoire source (et non dans des répertoires variantes). Additionnellement, la liste des fichiers du po/répertoire contient le LINGUASfichier, donc l'arbre source semble familier aux traducteurs, et ils peuvent travailler avec le projet de leur manière habituelle.

Exemple 3 . Préparons un arbre de développement comme cidessous

```
 projet/
  + SConstruct
  + Build /
  + Src /
      + Po /
          + SConscript
          + SConscript.i18n
          + POTFILES.in
          + LINGUAS
```

avec buildêtre répertoire variant. Écrivez le haut niveauSConstructscript comme suit

```
  # SConstruct
  env = environnement (outils = [ "default", "gettext"])
  VariantDir ( 'build', 'src', en double = 0)
  env [ 'POAUTOINIT'] = 1
  SConscript ( 'src / po / SConscript.i18n', 'exportations env' =)
  SConscript ( 'build / po / SConscript', 'env' export =)
```

le src/po/SConscript.i18ncomme

```
  # Src / po / SConscript.i18n
  Import ( 'env')
  env.Translate (LINGUAS_FILE = 1, XGETTEXTFROM = 'POTFILES.in', XGETTEXTPATH = [ '../'])
```

et le src/po/SConscript

```
  # Src / po / SConscript
  Import ( 'env')
  env.MOFiles (LINGUAS_FILE = 1)
```

Une telle configuration produit POTet POfichiers sous l' arbre source dans src/po/et binaire des MOfichiers sous l' arbre de variante build/po/. De cette façon , les POTet les POfichiers sont séparés des autres fichiers de sortie, qui ne doit pas être de retour engagés à des dépôts de source (par exemple , MO fichiers).

Remarque

Dans l' exemple ci - dessus, les POfichiers ne sont pas mis à jour, ni créé automatiquement lors de l' émission **scons « » commander. Les fichiers doivent être mis à jour (créé) à la main via **scons po-mise à jour** , puis les MOfichiers peuvent être compilés en exécutant **scons « » .

`TypeLibrary()` , `env.TypeLibrary()`

Construit une bibliothèque (type de Windows `.tlb`fichier) à partir d' un fichier IDL d'entrée (`.idl`). De plus, il construira le talon d'interface associés et les fichiers source de proxy, en les nommant d' après le nom de base du `.idl`fichier. Par exemple,

```
env.TypeLibrary (source = "foo.idl")
```

Créera `foo.tlb`, `foo.h`, `foo_i.c`, `foo_p.c` et les `foo_data.c` fichiers.

`Uic()` , `env.Uic()`

Construit un fichier d' en- tête, un fichier de mise en œuvre et un fichier moc à partir d' un fichier ui. et renvoie les noeuds correspondants dans l'ordre ci - dessus. Ce constructeur est disponible après avoir utilisé l'outil « qt ». Remarque: vous pouvez spécifier des `.ui`fichiers directement sous forme de fichiers source au `Program`, `Library`et les `SharedLibrary`constructeurs sans utiliser ce constructeur. L' utilisation de ce constructeur vous permet de passer outre les conventions de nommage standard (attention: préfixes sont toujours préfixé aux noms de fichiers construits, si vous ne voulez pas préfixes, vous pouvez les mettre à ``). Voir la $OTDIRvariable pour plus d' informations. Exemple:

```
env.Uic ( 'foo.ui') # -> [ 'foo.h', 'uic_foo.cc', 'moc_foo.cc']
env.Uic (target = split ( 'include / foo.h gen / uicfoo.cc gen / mocfoo.cc'),
        source = 'foo.ui') # -> [ 'include / foo.h', 'gen / uicfoo.cc', 'gen / mocfoo.cc']
```

`Zip()` , `env.Zip()`

Construit une archive zip des fichiers spécifiés et / ou des répertoires. Contrairement à la plupart des méthodes de constructeur, la `Zip` méthode constructeur peut être appelé plusieurs fois pour une cible donnée; chaque appel supplémentaire ajoute à la liste des entrées qui sera construit dans l'archive. Tous les répertoires source seront scannés pour des modifications à tous les fichiers sur le disque, peu importe si oui ou non au `scons` courant les autres d'appels Builder ou fonction.

```
env.Zip ( 'src.zip', 'src')

# Créez le fichier stuff.zip.
env.Zip ( 'stuff', [ 'subdir1', 'subdir2'])
# Ajouter également « une autre » au fichier stuff.tar.
env.Zip ( 'substance', 'autre')
```

Annexe C. Outils

Cette annexe contient la description de tous les modules outils disponibles « de la boîte » dans cette version de SCons.

386asm

Définit les variables de construction pour l'assembleur 386ASM pour le système d'exploitation embarqué ETS Phar Lap.

Ensembles: $AS, $ASCOM, $ASFLAGS, $ASPPCOM, $ASPPFLAGS.

Utilisations: $CC, $CPPFLAGS, $_CPPDEFFLAGS, $_CPPINCFLAGS.

AIXC ++

Définit les variables de construction pour l'IMB xlc / visuel âge compilateur C ++.

Ensembles: $CXX, $CXXVERSION, $SHCXX, $SHOBJSUFFIX.

aixcc

Définit les variables de construction pour IBM xlc / visuel âge compilateur C.

Définit: $CC, $CCVERSION, $SHCC.

aixf77

Définit les variables de construction pour l'VisualAge IBM compilateur Fortran F77.

Définit: $F77, $SHF77.

aixlink

Définit les variables de construction pour l'éditeur de liens Visual Age IBM.

Définit: $LINKFLAGS, $SHLIBSUFFIX, $SHLINKFLAGS.

AppleLink

Définit les variables de construction pour l'éditeur de liens d'Apple (similaire à l'éditeur de liens GNU).

Ensembles: $FRAMEWORKPATHPREFIX, $LDMODULECOM, $LDMODULEFLAGS, $LDMODULEPREFIX, $LDMODULESUFFIX, $LINKCOM, $SHLINKCOM, $SHLINKFLAGS, $_FRAMEWORKPATH, $

Utilisations: $FRAMEWORKSFLAGS.

ar

Définit les variables de construction pour le ar archiveur de la bibliothèque.

Ensembles: $AR, $ARCOM, $ARFLAGS, $LIBPREFIX, $LIBSUFFIX, $RANLIB, $RANLIBCOM, $RANLIBFLAGS.

comme

Définit les variables de construction pour l' en assembleur.

Ensembles: $AS, $ASCOM, $ASFLAGS, $ASPPCOM, $ASPPFLAGS.

Utilisations: $CC, $CPPFLAGS, $_CPPDEFFLAGS, $_CPPINCFLAGS.

bcc32

Définit les variables de construction pour le compilateur bcc32.

Ensembles: $CC, $CCCOM, $CCFLAGS, $CFILESUFFIX, $CFLAGS, $CPPDEFPREFIX, $CPPDEFSUFFIX, $INCPREFIX, $INCSUFFIX, $SHCC, $SHCCCOM, $SHCCFLAGS, $SHCFLAGS, $SH

Utilisations: $_CPPDEFFLAGS, $_CPPINCFLAGS.

BitKeeper

Définit les variables de construction pour le système de contrôle de code source BitKeeper.

Ensembles: $BITKEEPER, $BITKEEPERCOM, $BITKEEPERGET, $BITKEEPERGETFLAGS.

Utilisations: $BITKEEPERCOMSTR.

cc

Définit les variables de construction pour copmilers C Posix génériques.

Ensembles: $CC, $CCCOM, $CCFLAGS, $CFILESUFFIX, $CFLAGS, $CPPDEFPREFIX, $CPPDEFSUFFIX, $FRAMEWORKPATH, $FRAMEWORKS, $INCPREFIX, $INCSUFFIX, $SHCC, $SHCCC

Utilisations: $PLATFORM.

CVF

Définit les variables de construction pour le compilateur Visual Fortran Compaq.

Ensembles: $FORTRAN, $FORTRANCOM, $FORTRANMODDIR, $FORTRANMODDIRPREFIX, $FORTRANMODDIRSUFFIX, $FORTRANPPCOM, $OBJSUFFIX, $SHFORTRANCOM, $SHFORTRANPPCC

Utilisations: $CPPFLAGS, $FORTRANFLAGS, $SHFORTRANFLAGS, $_CPPDEFFLAGS, $_FORTRANINCFLAGS, $_FORTRANMODFLAG.

CVS

Définit les variables de construction pour le système de gestion de code source CVS.

Ensembles: $CVS, $CVSCOFLAGS, $CVSCOM, $CVSFLAGS.

Utilisations: $CVSCOMSTR.

cxx

Définit les variables de construction pour les compilateurs C ++ génériques POSIX.

Ensembles: $CPPDEFPREFIX, $CPPDEFSUFFIX, $CXX, $CXXCOM, $CXXFILESUFFIX, $CXXFLAGS, $INCPREFIX, $INCSUFFIX, $OBJSUFFIX, $SHCXX, $SHCXXCOM, $SHCXXFLAGS, $S

Utilisations: $CXXCOMSTR.

cyglink

Définir des variables de construction pour linker Cygwin / chargeur.

Ensembles: $IMPLIBPREFIX, $IMPLIBSUFFIX, $LDMODULEVERSIONFLAGS, $LINKFLAGS, $RPATHPREFIX, $RPATHSUFFIX, $SHLIBPREFIX, $SHLIBSUFFIX, $SHLIBVERSIONFLAGS

défaut

Définit les variables en appelant une liste par défaut des modules d'outils pour la plate-forme sur laquelle SCons est en cours d'exécution.

DMD

Définit les variables de construction pour le compilateur de langage D DMD.

Ensembles: $DC, $DCOM, $DDEBUG, $DDEBUGPREFIX, $DDEBUGSUFFIX, $DFILESUFFIX, $DFLAGPREFIX, $DFLAGS, $DFLAGSUFFIX, $DINCPREFIX, $DINCSUFFIX, $DLIB, $DLIBCO

DocBook

Cet outil tente de faire travailler avec DocBook en SCons un peu plus facile. Il fournit plusieurs toolchains pour créer différents formats de sortie, comme HTML ou PDF. Contenu dans le paquet est une distribution des feuilles de style XSL DocBook comme la version 1.76.1. Tant que vous ne spécifiez pas vos propres feuilles de style pour la personnalisation, ces versions officielles sont cueillies par défaut ... ce qui devrait réduire les tracas de configuration inévitables pour vous.

Dépendances implicites aux images et XIncludes sont détectés automatiquement si vous répondez aux exigences HTML. La feuille de style supplémentaire `utils/xmldepend.xsl`par Paul DuBois est utilisé à cette fin.

Notez qu'il n'y a pas de support pour le catalogue XML offert la résolution! Cet outil appelle les processeurs XSLT et PDF avec équarrisseurs les feuilles de style que vous avez spécifié, c'est tout. Le reste est entre vos mains et vous avez encore besoin de savoir ce que vous faites lors de la résolution des noms par un catalogue.

Pour activer l'outil « DocBook », vous devez ajouter son nom au constructeur environnement, comme celui-ci

```
env = environnement (outils = [ 'DocBook'])
```

Sur son démarrage, l'outil DocBook essaie de trouver un nécessaire `xsltproc`processeur, et un moteur de rendu PDF, par exemple `fop`. Donc , assurez - vous que ceux - ci sont ajoutés à l'environnement de votre système `PATH`et peuvent être appelés directement, sans spécifier leur chemin complet.

Pour la plupart des traitements de base de DocBook en HTML, vous devez avoir installé

- le python se `lxml`lier à `libxml2`, ou
- les liaisons directes pour python `libxml2/libxslt`, ou
- un processeur XSLT autonome, sont actuellement détectée `xsltproc`, `saxon`, `saxon-xslt` et `xalan`.

Rendu au format PDF vous oblige à avoir l' une des applications`fop`ou `xep`installées.

Création d'un document HTML ou PDF est très simple et directe. Dire

```
env = environnement (outils = [ 'DocBook'])
env.DocbookHtml ( 'manual.html', 'manual.xml')
env.DocbookPdf ( 'manual.pdf', 'manual.xml')
```

pour obtenir les deux sorties de votre source XML `manual.xml`. Comme un raccourci, vous pouvez donner la tige des seuls noms de fichiers, comme ceci:

```
env = environnement (outils = [ 'DocBook'])
env.DocbookHtml ( 'manuel')
env.DocbookPdf ( 'manuel')
```

et obtenir le même résultat. listes de cibles et la source sont également pris en charge:

```
env = environnement (outils = [ 'DocBook'])
env.DocbookHtml ([ 'manual.html', 'reference.html'], [ 'manual.xml', 'reference.xml'])
```

ou même

```
env = environnement (outils = [ 'DocBook'])
env.DocbookHtml ([ 'manuel', 'référence'])
```

Important

Chaque fois que vous quittez la liste des sources, vous ne pouvez pas spécifier une extension de fichier! L'outil utilise les noms donnés comme des tiges fichier, et ajoute les suffixes des fichiers source et cible en conséquence.

Les règles données ci - dessus sont valables pour les constructeurs `DocbookHtml`, `DocbookPdf`, `DocbookEpub`, `DocbookSlidesPdf`et `DocbookXInclude`. Pour la`DocbookMan`transformation , vous pouvez spécifier un nom cible, mais les noms de sortie réels sont réglés automatiquement à partir des `refname`entrées dans votre source XML.

Les constructeurs `DocbookHtmlChunked`, `DocbookHtmlhelp`et`DocbookSlidesHtml`sont spéciaux, en ce que:

1. ils créent un grand nombre de fichiers, où les noms exacts et leur nombre dépendent du contenu du fichier source, et
2. l'objectif principal est toujours nommé `index.html`, à savoir le nom de sortie pour la transformation XSL n'est pas repris par les feuilles de style.

En conséquence, il n'y a tout simplement pas d'utilisation en spécifiant un nom HTML cible. Donc, la syntaxe de base pour ces constructeurs est toujours:

```
env = environnement (outils = [ 'DocBook'])
env.DocbookHtmlhelp ( 'manuel')
```

Si vous souhaitez utiliser un fichier XSL spécifique, vous pouvez définir le montant supplémentaire de `xsl`paramètre à votre appel Builder comme suit:

```
env.DocbookHtml ( 'autre.html', 'manual.xml', XSL = 'html.xsl')
```

Étant donné que cela peut être fastidieux si vous utilisez toujours la même dénomination locale pour vos fichiers personnalisés, par exemple XSL `html.xsl`pour HTML et `pdf.xsl`pour la sortie PDF, un ensemble de variables pour définir le nom par défaut XSL est fourni. Ceux-ci sont:

```
DOCBOOK_DEFAULT_XSL_HTML
DOCBOOK_DEFAULT_XSL_HTMLCHUNKED
DOCBOOK_DEFAULT_XSL_HTMLHELP
DOCBOOK_DEFAULT_XSL_PDF
DOCBOOK_DEFAULT_XSL_EPUB
DOCBOOK_DEFAULT_XSL_MAN
DOCBOOK_DEFAULT_XSL_SLIDESPDF
DOCBOOK_DEFAULT_XSL_SLIDESHTML
```

et vous pouvez les régler lors de la construction de votre environnement:

```
env = environnement (outils = [ 'DocBook'],
                  DOCBOOK_DEFAULT_XSL_HTML = 'html.xsl',
                  DOCBOOK_DEFAULT_XSL_PDF = 'pdf.xsl')
env.DocbookHtml ( 'manuel') # utilise maintenant html.xsl
```

Ensembles: `$DOCBOOK_DEFAULT_XSL_EPUB`, `$DOCBOOK_DEFAULT_XSL_HTML`, `$DOCBOOK_DEFAULT_XSL_HTMLCHUNKED`, `$DOCBOOK_DEFAULT_XSL_HTMLHELP`, `$DOCBOOK_DEFAULT_`

Utilisations: `$DOCBOOK_FOPCOMSTR`, `$DOCBOOK_XMLLINTCOMSTR`, `$DOCBOOK_XSLTPROCCOMSTR`.

dvi

Rattache le `DVI`constructeur à l'environnement de la construction.

dvipdf

Définit les variables de construction pour l'utilitaire dvipdf.

Définit: $DVIPDF, $DVIPDFCOM, $DVIPDFFLAGS.

Utilisations: $DVIPDFCOMSTR.

dvips

Définit les variables de construction pour l'utilitaire dvips.

Ensembles: $DVIPS, $DVIPSFLAGS, $PSCOM, $PSPREFIX, $PSSUFFIX.

Utilisations: $PSCOMSTR.

F03

Définir des variables de construction pour Fortran 03 compilateurs génériques. Posix

Ensembles: $F03, $F03COM, $F03FLAGS, $F03PPCOM, $SHF03, $SHF03COM, $SHF03FLAGS, $SHF03PPCOM, $_F03INCFLAGS.

Utilisations: $F03COMSTR, $F03PPCOMSTR, $SHF03COMSTR, $SHF03PPCOMSTR.

F08

Définir des variables de construction pour Fortran 08 compilateurs génériques. Posix

Ensembles: $F08, $F08COM, $F08FLAGS, $F08PPCOM, $SHF08, $SHF08COM, $SHF08FLAGS, $SHF08PPCOM, $_F08INCFLAGS.

Utilisations: $F08COMSTR, $F08PPCOMSTR, $SHF08COMSTR, $SHF08PPCOMSTR.

F77

Définir des variables de construction pour 77 compilateurs Fortran génériques. Posix

Ensembles: $F77, $F77COM, $F77FILESUFFIXES, $F77FLAGS, $F77PPCOM, $F77PPFILESUFFIXES, $FORTRAN, $FORTRANCOM, $FORTRANFLAGS, $SHF77, $SHF77COM, $SHF77FLA

Utilisations: $F77COMSTR, $F77PPCOMSTR, $FORTRANCOMSTR, $FORTRANPPCOMSTR, $SHF77COMSTR, $SHF77PPCOMSTR, $SHFORTRANCOMSTR, $SHFORTRANPPCOMSTR.

F90

Définir des variables de construction pour Fortran 90 génériques Posix compilateurs.

Ensembles: $F90, $F90COM, $F90FLAGS, $F90PPCOM, $SHF90, $SHF90COM, $SHF90FLAGS, $SHF90PPCOM, $_F90INCFLAGS.

Utilisations: $F90COMSTR, $F90PPCOMSTR, $SHF90COMSTR, $SHF90PPCOMSTR.

F95

Définir des variables de construction pour Fortran 95 compilateurs génériques. Posix

Ensembles: $F95, $F95COM, $F95FLAGS, $F95PPCOM, $SHF95, $SHF95COM, $SHF95FLAGS, $SHF95PPCOM, $_F95INCFLAGS.

Utilisations: $F95COMSTR, $F95PPCOMSTR, $SHF95COMSTR, $SHF95PPCOMSTR.

Fortran

Définir des variables de construction pour les compilateurs Fortran génériques. Posix

Ensembles: $FORTRAN, $FORTRANCOM, $FORTRANFLAGS, $SHFORTRAN, $SHFORTRANCOM, $SHFORTRANFLAGS, $SHFORTRANPPCOM.

Utilisations: $FORTRANCOMSTR, $FORTRANPPCOMSTR, $SHFORTRANCOMSTR, $SHFORTRANPPCOMSTR.

g ++

Définir des variables de construction pour le gxx compilateur C ++.

Ensembles: $CXX, $CXXVERSION, $SHCXXFLAGS, $SHOBJSUFFIX.

G77

Définir les variables de construction pour le G77 compilateur Fortran. Appelle le `f77`module d'outil pour définir des variables.

gaz

Définit les variables de construction pour le gaz assembleur. Appelle le `as`module.

Définit: $AS.

gcc

Définir des variables de construction pour le gcc compilateur C.

Définit: $CC, $CCVERSION, $SHCCFLAGS.

gdc

Définit les variables de construction pour le compilateur du langage D GDC.

Ensembles: $DC, $DCOM, $DDEBUG, $DDEBUGPREFIX, $DDEBUGSUFFIX, $DFILESUFFIX, $DFLAGPREFIX, $DFLAGS, $DFLAGSUFFIX, $DINCPREFIX, $DINCSUFFIX, $DLIB, $DLIBCO

gettext

Ceci est en fait un ensemble d'outils, ce qui soutient l'internationalisation et la localisation de logiciels en cours de construction avec SCons. Les charges de jeu d'outils suivants: outils

- xgettext- pour extraire les messages internationalisés à partir du code source de POTfichier (s),
- msginit- peut éventuellement être utilisé pour initialiser les PO fichiers,
- msgmerge- mettre à jour des POfichiers, qui contiennent déjà des messages traduits,
- msgfmt- pour compiler textuelle POfichier binaire installable MOfichier.

Lorsque vous activez gettext, il charge en interne tous les outils mentionnés ci - dessus, vous êtes encouragés à consulter leur documentation individuelle.

Chacun des outils ci - dessus fournit son propre constructeur (s) qui peuvent être utilisés pour effectuer des activités particulières liées à l' internationalisation du logiciel. Vous pouvez cependant être intéressé par *haut niveau* constructeur Translatedécrit quelques paragraphes plus loin.

Pour utiliser les gettextoutils ajouter 'gettext'outil à votre environnement:

```
env = environnement (outils = [ 'default', 'gettext'])
```

gfortran

Définit les variables de construction pour la F95 GNU / F2003 compilateur GNU.

Ensembles: $F77, $F90, $F95, $FORTRAN, $SHF77, $SHF77FLAGS, $SHF90, $SHF90FLAGS, $SHF95, $SHF95FLAGS, $SHFORTRAN, $SHFORTRANFLAGS.

gnulink

Définir des variables de construction pour linker GNU / chargeur.

Ensembles: $LDMODULEVERSIONFLAGS, $RPATHPREFIX, $RPATHSUFFIX, $SHLIBVERSIONFLAGS, $SHLINKFLAGS, $_LDMODULESONAME, $_SHLIBSONAME.

gs

Cet outil définit les variables de construction nécessaires pour travailler avec la commande Ghostscript. Il enregistre également une action appropriée avec le générateur de PDF (PDF), de sorte que la conversion du PS / EPS au format PDF se fait automatiquement pour la TeX / LaTeX toolchain. Enfin, il ajoute une explicite Ghostscript Builder (Gs) à l'environnement.

Définit: $GS, $GSCOM, $GSFLAGS.

Utilisations: $GSCOMSTR.

++ hpc

Définissez des variables de construction pour les compilateurs sur les systèmes HP aCC / UX.

hpcc

Définir des variables de construction pour le aCC sur les systèmes HP / UX. Appelle l' cxxoutil pour les variables supplémentaires.

Définit: $CXX, $CXXVERSION, $SHCXXFLAGS.

hplink

Définit les variables de construction pour l'éditeur de liens sur les systèmes HP / UX.

Définit: $LINKFLAGS, $SHLIBSUFFIX, $SHLINKFLAGS.

icc

Définit les variables de construction pour le cpi compilateur sur les systèmes OS / 2.

Ensembles: $CC, $CCCOM, $CFILESUFFIX, $CPPDEFPREFIX, $CPPDEFSUFFIX, $CXXCOM, $CXXFILESUFFIX, $INCPREFIX, $INCSUFFIX.

Utilisations: $CCFLAGS, $CFLAGS, $CPPFLAGS, $_CPPDEFFLAGS, $_CPPINCFLAGS.

ICL

Définit les variables de construction pour Intel C / C ++. Appelle le intelcmodule d'outil pour définir ses variables.

IFL

Définit les variables de construction pour le compilateur Intel Fortran.

Ensembles: $FORTRAN, $FORTRANCOM, $FORTRANPPCOM, $SHFORTRANCOM, $SHFORTRANPPCOM.

Utilisations: $CPPFLAGS, $FORTRANFLAGS, $_CPPDEFFLAGS, $_FORTRANINCFLAGS.

ifort

Définit les variables de construction pour les nouvelles versions du compilateur Intel Fortran pour Linux.

Ensembles: $F77, $F90, $F95, $FORTRAN, $SHF77, $SHF77FLAGS, $SHF90, $SHF90FLAGS, $SHF95, $SHF95FLAGS, $SHFORTRAN, $SHFORTRANFLAGS.

iLink

Définit les variables de construction pour le ilink éditeur de liens sur les systèmes OS / 2.

Ensembles: $LIBDIRPREFIX, $LIBDIRSUFFIX, $LIBLINKPREFIX, $LIBLINKSUFFIX, $LINK, $LINKCOM, $LINKFLAGS.

ilink32

Définit les variables de construction pour le Borland de l'éditeur de liens.

Ensembles: $LIBDIRPREFIX, $LIBDIRSUFFIX, $LIBLINKPREFIX, $LIBLINKSUFFIX, $LINK, $LINKCOM, $LINKFLAGS.

installer

Définit les variables de construction pour l'installation de fichiers et de répertoires.

Définit: $INSTALL, $INSTALLSTR.

intelc

Définit les variables de construction pour Intel C / C ++ (Linux et Windows, version 7 et versions ultérieures). Appelle la `gcc`ou `msvc`(sous Linux et Windows, respectivement) pour définir des variables sous - jacentes.

Ensembles: $AR, $CC, $CXX, $INTEL_C_COMPILER_VERSION, $LINK.

pot

Définit les variables de construction pour le pot utilitaire.

Ensembles: $JAR, $JARCOM, $JARFLAGS, $JARSUFFIX.

Utilisations: $JARCOMSTR.

javac

Définit les variables de construction pour le javac compilateur.

Ensembles: $JAVABOOTCLASSPATH, $JAVAC, $JAVACCOM, $JAVACFLAGS, $JAVACLASSPATH, $JAVACLASSSUFFIX, $JAVASOURCEPATH, $JAVASUFFIX.

Utilisations: $JAVACCOMSTR.

javah

Définit les variables de construction pour le javah outil.

Ensembles: $JAVACLASSSUFFIX, $JAVAH, $JAVAHCOM, $JAVAHFLAGS.

Utilisations: $JAVACLASSPATH, $JAVAHCOMSTR.

latex

Définit les variables de construction pour le latex utilitaire.

Définit: $LATEX, $LATEXCOM, $LATEXFLAGS.

Utilisations: $LATEXCOMSTR.

ldc

Définit les variables de construction pour le compilateur du langage D LDC2.

Ensembles: $DC, $DCOM, $DDEBUG, $DDEBUGPREFIX, $DDEBUGSUFFIX, $DFILESUFFIX, $DFLAGPREFIX, $DFLAGS, $DFLAGSUFFIX, $DINCPREFIX, $DINCSUFFIX, $DLIB, $DLIBCO

Lex

Définit les variables de construction pour la lex analyseur lexical.

Définit: $LEX, $LEXCOM, $LEXFLAGS.

Utilisations: $LEXCOMSTR.

lien

Définit les variables de construction pour linkers génériques POSIX.

Ensembles: $LDMODULE, $LDMODULECOM, $LDMODULEFLAGS, $LDMODULENOVERSIONSYMLINKS, $LDMODULEPREFIX, $LDMODULESUFFIX, $LDMODULEVERSION, $LDMODULEVERSIONF

Utilisations: $LDMODULECOMSTR, $LINKCOMSTR, $SHLINKCOMSTR.

linkloc

Définit les variables de construction pour le LinkLoc éditeur de liens pour le système d'exploitation embarqué ETS Phar Lap.

Ensembles: $LIBDIRPREFIX, $LIBDIRSUFFIX, $LIBLINKPREFIX, $LIBLINKSUFFIX, $LINK, $LINKCOM, $LINKFLAGS, $SHLINK, $SHLINKCOM, $SHLINKFLAGS.

Utilisations: $LINKCOMSTR, $SHLINKCOMSTR.

m4

Définit les variables de construction pour le m4 processeur macro.

Définit: $M4, $M4COM, $M4FLAGS.

Utilisations: $M4COMSTR.

masm

Définit les variables de construction pour l'assembleur Microsoft.

Ensembles: $AS, $ASCOM, $ASFLAGS, $ASPPCOM, $ASPPFLAGS.

Utilisations: $ASCOMSTR, $ASPPCOMSTR, $CPPFLAGS, $_CPPDEFFLAGS, $_CPPINCFLAGS.

midl

Définit les variables de construction pour le compilateur Microsoft IDL.

Définit: $MIDL, $MIDLCOM, $MIDLFLAGS.

Utilisations: $MIDLCOMSTR.

MinGW

Définit les variables de construction pour MinGW (Gnu minimale sous Windows).

Ensembles: $AS, $CC, $CXX, $LDMODULECOM, $LIBPREFIX, $LIBSUFFIX, $OBJSUFFIX, $RC, $RCCOM, $RCFLAGS, $RCINCFLAGS, $RCINCPREFIX, $RCINCSUFFIX, $SHCCFLAGS, $S

Utilisations: $RCCOMSTR, $SHLINKCOMSTR.

msgfmt

Cet outil scons fait partie de scons gettextensemble d' outils. Il fournit une interface de scons à **msgfmt (1)** commande, qui génère catalogue de messages binaires (MO) d'une description textuelle de la traduction (PO).

Ensembles: $MOSUFFIX, $MSGFMT, $MSGFMTCOM, $MSGFMTCOMSTR, $MSGFMTFLAGS, $POSUFFIX.

Utilisations: $LINGUAS_FILE.

msginit

Cet outil scons fait partie de scons gettextensemble d' outils. Il fournit une interface de scons à **msginit (1)** programme, ce qui crée un nouveau Pofichier, l' initialisation des méta - informations avec les valeurs de l'environnement de l' utilisateur (ou options).

Ensembles: $MSGINIT, $MSGINITCOM, $MSGINITCOMSTR, $MSGINITFLAGS, $POAUTOINIT, $POCREATE_ALIAS, $POSUFFIX, $POTSUFFIX, $_MSGINITLOCALE.

Utilisations: $LINGUAS_FILE, $POAUTOINIT, $POTDOMAIN.

msgmerge

Cet outil scons fait partie de scons gettextensemble d' outils. Il fournit une interface de scons à **msgmerge (1)** commande, qui fusionne deux styles uniformes .pofichiers ensemble.

Ensembles: $MSGMERGE, $MSGMERGECOM, $MSGMERGECOMSTR, $MSGMERGEFLAGS, $POSUFFIX, $POTSUFFIX, $POUPDATE_ALIAS.

Utilisations: $LINGUAS_FILE, $POAUTOINIT, $POTDOMAIN.

mslib

Définit les variables de construction pour Microsoft de archiveur bibliothèque.

Ensembles: $AR, $ARCOM, $ARFLAGS, $LIBPREFIX, $LIBSUFFIX.

Utilisations: $ARCOMSTR.

MSLINK

Définit les variables de construction pour l'éditeur de liens Microsoft.

Ensembles: $LDMODULE, $LDMODULECOM, $LDMODULEFLAGS, $LDMODULEPREFIX, $LDMODULESUFFIX, $LIBDIRPREFIX, $LIBDIRSUFFIX, $LIBLINKPREFIX, $LIBLINKSUFFIX, $LI

Utilisations: $LDMODULECOMSTR, $LINKCOMSTR, $REGSVRCOMSTR, $SHLINKCOMSTR.

mssdk

Définit les variables pour Microsoft Platform SDK et / ou Windows SDK. Notez que , contrairement à la plupart des autres modules de l' outil, mssdk ne définit pas les variables de construction, mais définit les *variables d'environnement* dans l'environnement SCons utilise pour exécuter l'ensemble des outils Microsoft: %INCLUDE%, %LIB%, %LIBPATH%et %PATH%.

Utilisations: $MSSDK_DIR, $MSSDK_VERSION, $MSVS_VERSION.

msvc

Définit les variables de construction pour Microsoft Visual C / C ++.

Ensembles: $BUILDERS, $CC, $CCCOM, $CCFLAGS, $CCPCHFLAGS, $CCPDBFLAGS, $CFILESUFFIX, $CFLAGS, $CPPDEFPREFIX, $CPPDEFSUFFIX, $CXX, $CXXCOM, $CXXFILESUFFIX

Utilisations: $CCCOMSTR, $CXXCOMSTR, $PCH, $PCHSTOP, $PDB, $SHCCCOMSTR, $SHCXXCOMSTR.

msvs

Définit les variables de construction pour Microsoft Visual Studio.

Ensembles: $MSVSBUILDCOM, $MSVSCLEANCOM, $MSVSENCODING, $MSVSPROJECTCOM, $MSVSREBUILDCOM, $MSVSSCONS, $MSVSSCONSCOM, $MSVSSCONSCRIPT, $MSVSSCONSFLAGS,

MWCC

Définit les variables de construction pour le compilateur Metrowerks CodeWarrior.

Ensembles: $CC, $CCCOM, $CFILESUFFIX, $CPPDEFPREFIX, $CPPDEFSUFFIX, $CXX, $CXXCOM, $CXXFILESUFFIX, $INCPREFIX, $INCSUFFIX, $MWCW_VERSION, $MWCW_VERSIONS

Utilisations: $CCCOMSTR, $CXXCOMSTR, $SHCCCOMSTR, $SHCXXCOMSTR.

MWLD

Définit les variables de construction pour l'éditeur de liens Metrowerks CodeWarrior.

Ensembles: $AR, $ARCOM, $LIBDIRPREFIX, $LIBDIRSUFFIX, $LIBLINKPREFIX, $LIBLINKSUFFIX, $LINK, $LINKCOM, $SHLINK, $SHLINKCOM, $SHLINKFLAGS.

nasm

Définit les variables de construction pour le nasm Netwide Assembleur.

Ensembles: $AS, $ASCOM, $ASFLAGS, $ASPPCOM, $ASPPFLAGS.

Utilisations: $ASCOMSTR, $ASPPCOMSTR.

emballage

Un cadre pour la construction de paquets binaires et sources.

Emballage

Définit les variables de construction pour le `Package`constructeur.

pdf

Définit les variables de construction pour le constructeur de Portable Document Format.

Définit: $PDFPREFIX, $PDFSUFFIX.

pdflatex

Définit les variables de construction pour l' pdflatex utilitaire.

Ensembles: $LATEXRETRIES, $PDFLATEX, $PDFLATEXCOM, $PDFLATEXFLAGS.

Utilisations: $PDFLATEXCOMSTR.

pdftex

Définit les variables de construction pour l' pdftex utilitaire.

Ensembles: $LATEXRETRIES, $PDFLATEX, $PDFLATEXCOM, $PDFLATEXFLAGS, $PDFTEX, $PDFTEXCOM, $PDFTEXFLAGS.

Utilisations: $PDFLATEXCOMSTR, $PDFTEXCOMSTR.

Forcément

Définit les variables de construction pour interagir avec le système de gestion de code source Perforce.

Définit: $P4, $P4COM, $P4FLAGS.

Utilisations: $P4COMSTR.

qt

Définit les variables de construction pour la construction d'applications Qt.

Ensembles: $QTDIR, $QT_AUTOSCAN, $QT_BINPATH, $QT_CPPPATH, $QT_LIB, $QT_LIBPATH, $QT_MOC, $QT_MOCCXXPREFIX, $QT_MOCCXXSUFFIX, $QT_MOCFROMCXXCOM, $QT_MO

RCS

Définit les variables de construction pour l'interaction avec le système de contrôle de révision.

Ensembles: $RCS, $RCS_CO, $RCS_COCOM, $RCS_COFLAGS.

Utilisations: $RCS_COCOMSTR.

rmic

Définit les variables de construction pour le CRIM utilitaire.

Ensembles: $JAVACLASSSUFFIX, $RMIC, $RMICCOM, $RMICFLAGS.

Utilisations: $RMICCOMSTR.

rpcgen

Définit les variables de construction pour la construction avec rpcgen.

Ensembles: $RPCGEN, $RPCGENCLIENTFLAGS, $RPCGENFLAGS, $RPCGENHEADERFLAGS, $RPCGENSERVICEFLAGS, $RPCGENXDRFLAGS.

CSSC

Définit les variables de construction pour interagir avec le code source du système de contrôle.

Ensembles: $SCCS, $SCCSCOM, $SCCSFLAGS, $SCCSGETFLAGS.

Utilisations: $SCCSCOMSTR.

sgiar

Définit les variables de construction pour l'archivage de la bibliothèque SGI.

Ensembles: $AR, $ARCOMSTR, $ARFLAGS, $LIBPREFIX, $LIBSUFFIX, $SHLINK, $SHLINKFLAGS.

Utilisations: $ARCOMSTR, $SHLINKCOMSTR.

SGIC ++

Définit les variables de construction pour le SGI compilateur C ++.

Ensembles: $CXX, $CXXFLAGS, $SHCXX, $SHOBJSUFFIX.

SGICC

Définit les variables de construction pour le compilateur C SGI.

Définit: $CXX, $SHOBJSUFFIX.

sgilink

Définit les variables de construction pour l'éditeur de liens SGI.

Ensembles: $LINK, $RPATHPREFIX, $RPATHSUFFIX, $SHLINKFLAGS.

Sunar

Définit les variables de construction pour l'archivage de la bibliothèque du Soleil.

Ensembles: $AR, $ARCOM, $ARFLAGS, $LIBPREFIX, $LIBSUFFIX.

Utilisations: $ARCOMSTR.

sunc ++

Définit les variables de construction pour le compilateur Sun C ++.

Ensembles: $CXX, $CXXVERSION, $SHCXX, $SHCXXFLAGS, $SHOBJPREFIX, $SHOBJSUFFIX.

suncc

Définit les variables de construction pour le compilateur Sun C.

Ensembles: $CXX, $SHCCFLAGS, $SHOBJPREFIX, $SHOBJSUFFIX.

sunf77

Définir des variables de construction pour le Sun F77compilateur Fortran.

Ensembles: $F77, $FORTRAN, $SHF77, $SHF77FLAGS, $SHFORTRAN, $SHFORTRANFLAGS.

sunf90

Définir des variables de construction pour le Sun F90compilateur Fortran.

Ensembles: $F90, $FORTRAN, $SHF90, $SHF90FLAGS, $SHFORTRAN, $SHFORTRANFLAGS.

sunf95

Définir des variables de construction pour le Sun f95 compilateur Fortran.

Ensembles: $F95, $FORTRAN, $SHF95, $SHF95FLAGS, $SHFORTRAN, $SHFORTRANFLAGS.

sunlink

Définit les variables de construction pour l'éditeur de liens du Soleil.

Définit: $RPATHPREFIX, $RPATHSUFFIX, $SHLINKFLAGS.

lampée

Définit les variables de construction pour le générateur d'interface SWIG.

Ensembles: $SWIG, $SWIGCFILESUFFIX, $SWIGCOM, $SWIGCXXFILESUFFIX, $SWIGDIRECTORSUFFIX, $SWIGFLAGS, $SWIGINCPREFIX, $SWIGINCSUFFIX, $SWIGPATH, $SWIGVERS

Utilisations: $SWIGCOMSTR.

le goudron

Définit les variables de construction pour le goudron archiveur.

Ensembles: $TAR, $TARCOM, $TARFLAGS, $TARSUFFIX.

Utilisations: $TARCOMSTR.

Texas

Définit les variables de construction pour le formatter TeX et typographe.

Ensembles: $BIBTEX, $BIBTEXCOM, $BIBTEXFLAGS, $LATEX, $LATEXCOM, $LATEXFLAGS, $MAKEINDEX, $MAKEINDEXCOM, $MAKEINDEXFLAGS, $TEX, $TEXCOM, $TEXFLAGS.

Utilisations: $BIBTEXCOMSTR, $LATEXCOMSTR, $MAKEINDEXCOMSTR, $TEXCOMSTR.

fichier texte

Définir des variables de construction pour les `Textfile`et `Substfile`constructeurs.

Ensembles: $LINESEPARATOR, $SUBSTFILEPREFIX, $SUBSTFILESUFFIX, $TEXTFILEPREFIX, $TEXTFILESUFFIX.

Utilisations: $SUBST_DICT.

tlib

Définit les variables de construction pour le Borlan tib archiveur bibliothèque.

Ensembles: $AR, $ARCOM, $ARFLAGS, $LIBPREFIX, $LIBSUFFIX.

Utilisations: $ARCOMSTR.

xgettext

Cet outil scons fait partie de scons `gettext`ensemble d' outils. Il fournit une interface de scons à **xgettext (1)** programme, qui extrait des messages internationalisé à partir du code source. L'outil fournit `POTUpdate`constructeur pour faire de `PO` *modèles* de fichiers.

Ensembles: $POTSUFFIX, $POTUPDATE_ALIAS, $XGETTEXTCOM, $XGETTEXTCOMSTR, $XGETTEXTFLAGS, $XGETTEXTFROM, $XGETTEXTFROMPREFIX, $XGETTEXTFROMSUFFIX, $XGET

Utilisations: $POTDOMAIN.

yacc

Définit les variables de construction pour le yacc générateur d'analyse syntaxique.

Ensembles: $YACC, $YACCCOM, $YACCFLAGS, $YACCHFILESUFFIX, $YACCHXXFILESUFFIX, $YACCVCGFILESUFFIX.

Utilisations: $YACCCOMSTR.

Zip *: français

Définit les variables de construction pour le zip archiveur.

Ensembles: $ZIP, $ZIPCOM, $ZIPCOMPRESSION, $ZIPFLAGS, $ZIPSUFFIX.

Utilisations: $ZIPCOMSTR.

Annexe D. Fonctions et méthodes Environnement

Cette annexe contient la description de toutes les méthodes de la fonction et de l' environnement de la construction dans cette version de SCons

`Action(action, [cmd/str/fun, [var, ...]] [option=value, ...])` , `env.Action(action, [cmd/str/fun, [var, ...]] [option=value, ...])`

Crée un objet d'action pour la spécifiée `action`. Voir la section « Objets d'action » ci - dessous pour une explication complète des arguments et des comportements.

Notez que la `env.Action`forme de l'invocation () élargira les variables de construction dans toutes les chaînes d'arguments, y compris l' `action` argument au moment où il est appelé en utilisant les variables de construction dans l' `env` environnement de la construction à travers lequel `env.Action`() a été appelée. La`Action`forme () retarde toute l' extension de variable jusqu'à ce que l'objet d'action est effectivement utilisé.

`AddMethod(object, function, [name])` , `env.AddMethod(function, [name])`

Lorsqu'il est appelé avec la `AddMethod`forme (), ajoute le spécifié`function` à la spécifiée `object` comme la méthode spécifiée `name`. Lorsqu'il est appelé avec la `env.AddMethod`forme (), ajoute le spécifié `function` à l'environnement de construction `env` comme la méthode spécifiée `name`. Dans les deux cas, si `name` est omis ou`None`, le nom spécifié `function` lui - même est utilisé pour le nom de la méthode.

Exemples:

```
# Notez que le premier argument de la fonction
# Être fixé comme méthode doit être l'objet à travers
# Lequel la méthode sera appelée; Python
convention # est de l'appeler « soi ».
def ma_methode (self, arg):
    print "ma_methode () obtenu", arg

# Utilisez le AddMethod global () pour ajouter une méthode
# À la classe environnement. Ce
AddMethod (Environnement, ma_methode)
env = environnement ()
```

```
env.my_method ( 'arg')

# Ajoutez la fonction comme méthode, en utilisant la fonction
# nom pour l'appel de méthode.
env = environnement ()
env.AddMethod (ma_methode, 'other_method_name')
env.other_method_name ( 'autre arg')
```

`AddOption(arguments)`

Cette fonction ajoute une nouvelle option de ligne de commande pour être reconnu. La spécifiée `arguments` sont les mêmes que supporté par la norme Python `optparse.add_option`méthode () (avec quelques fonctions supplémentaires indiquées ci - dessous); consultez la documentation `optparse` pour une discussion approfondie de ses capabities-traitement option.

En plus des arguments et valeurs prises en charge par la`optparse.add_option`méthode (), la SCons `AddOption` fonction vous permet de définir la `nargs` valeur de mot - clé à `'?'` (une chaîne avec juste le point d'interrogation) pour indiquer que l'option longue spécifiée (s) prend (s) d' une *option* argument. Quand`nargs = '?'` est passé à la `AddOption` fonction, l' `const` argument de mot - clé peut être utilisé pour fournir la valeur « par défaut » qui doit être utilisé lorsque l'option est spécifiée sur la ligne de commande sans argument explicite.

Si aucun `default=` argument de mot - clé est fourni lors de l' appel`AddOption`, l'option aura une valeur par défaut `None`.

Une fois qu'une nouvelle option de ligne de commande a été ajoutée avec `AddOption`, peut accéder à la valeur d'option en utilisant `GetOption` ou `env.GetOption`(). La valeur peut également être définie, en utilisant `SetOption` ou `env.SetOption`(), si les conditions d'un `SConscript` besoin valeur supplante la valeur par défaut. Notez cependant qu'une valeur spécifiée sur la ligne de commande *toujours* passer outre une valeur définie par un fichier SConscript.

Toutes les spécifiées `help=` chaînes pour la nouvelle option (s) seront affichés par la `-H` ou les `-h` options (celle - ci que si aucun autre texte d'aide est spécifié dans les fichiers SConscript). Le texte d'aide pour les options locales spécifiées par `AddOption`apparaîtra sous les options SCons eux - mêmes, sous séparé `Local Options` rubrique. Les options apparaissent dans le texte d'aide dans l'ordre dans lequel les `AddOption` appels se produisent.

Exemple:

```
AddOption ( '- préfixe',
          dest = 'préfixe',
          nargs = 1, type = 'string',
          Action = 'magasin',
          metavar = 'DIR',
          help 'préfixe d'installation' =)
env = environnement (PREFIX = GetOption ( 'préfixe'))
```

`AddPostAction(target, action)` , `env.AddPostAction(target, action)`

Organise pour le spécifié `action` à effectuer après spécifié `target` a été construit. L'action spécifiée (s) peut être un objet d'action, ou tout ce qui peut être transformé en un objet d'action (voir ci - dessous).

Lorsque plusieurs cibles sont fournis, l'action peut être appelée plusieurs fois, une fois après chaque action qui génère une ou plusieurs cibles dans la liste.

`AddPreAction(target, action)` , `env.AddPreAction(target, action)`

Organise pour le spécifié `action` à effectuer avant la spécifiée`target` est construit. L'action spécifiée (s) peut être un objet d'action, ou tout ce qui peut être transformé en un objet d'action (voir ci - dessous).

Lorsque plusieurs cibles sont spécifiés, l'action (s) peut être appelé plusieurs fois, une fois avant chaque action qui génère une ou plusieurs cibles dans la liste.

Notez que si l' une des cibles sont construits en plusieurs étapes, l'action sera invoquée juste avant l'action « finale » qui génère spécifiquement la cible spécifiée (s). Par exemple, lors de la construction d' un programme exécutable à partir d' une source spécifiée `.c` fichier par l' intermédiaire d' un fichier objet intermédiaire:

```
foo = Programme ( 'foo.c')
AddPreAction (foo 'pre_action')
```

Le spécifié `pre_action` sera exécuté avant `scons` appelle la commande lien qui génère en fait le binaire de programme exécutable `foo`, pas avant de compiler le `foo.c` fichier dans un fichier objet.

`Alias(alias, [targets, [action]])` , `env.Alias(alias, [targets, [action]])`

Crée une ou plusieurs cibles factices qui se dilatent à un ou plusieurs autres cibles. Une option `action` (commande) ou une liste d'actions peuvent être spécifiées qui sera exécutée chaque fois que l'une des cibles d'alias sont hors jour. Renvoie l'objet nœud représentant l'alias, qui existe en dehors de tout système de fichiers. Cet objet de noeud, ou le nom d'alias, peuvent être utilisés en tant que dépendance d'une autre cible, y compris une autre alias. `Alias` peut être appelé plusieurs fois pour le même alias pour ajouter des cibles supplémentaires à l'alias, ou des actions supplémentaires à la liste pour cet alias.

Exemples:

```
Alias ( 'install')
Alias ( 'install', '/ usr / bin')
Alias ([ 'install', 'install-lib'], '/ usr / local / lib')

env.Alias ( 'install', [ '/ usr / local / bin', '/ usr / local / lib'])
env.Alias ( 'install', [ '/ usr / local / man'])

env.Alias ( 'mise à jour', [ 'file1', 'file2'], "update_database $ SOURCES")
```

`AllowSubstExceptions([exception, ...])`

Indique les exceptions qui seront autorisées lors de l' expansion des variables de construction. Par défaut, les extensions de variables de construction qui génèrent `NameError` ou `IndexError`exception gonflera à une `''` (une chaîne vide) et ne pas provoquer scons à l' échec. Toutes les exceptions non dans la liste spécifiée génère un message d'erreur et terminer le traitement.

Si l' `AllowSubstExceptions` on appelle plusieurs fois, chaque appel complètement la liste remplace précédente des exceptions autorisées.

Exemple:

```
# Exige que tous les noms de variables de construction existent.
# (Vous pouvez le faire si vous voulez appliquer strictement
# Que toutes les variables de construction doivent être définies avant l'utilisation.)
AllowSubstExceptions ()

# Permet également une chaîne contenant une expansion zéro division
# Comme '$ {1/0}' à evalute à ''.
AllowSubstExceptions (IndexError, NameError, ZeroDivisionError)
```

`AlwaysBuild(target, ...)` , `env.AlwaysBuild(target, ...)`

Marques chaque donnée `target` afin qu'il soit toujours supposé être à jour, et sera toujours reconstruit en cas de besoin. Notez toutefois que `AlwaysBuild` n'ajoute pas son objectif (s) à la liste cible par défaut, de sorte que les objectifs ne seront construits si elles sont spécifiées sur la ligne de commande, ou sont une personne à charge d'une cible spécifiée sur la ligne de commande - mais ils seront *toujours* construits si cela est indiqué. Cibles multiples peuvent être passés pour un seul appel à `AlwaysBuild`.

`env.Append(key=val, [...])`

Les arguments clés concatène spécifiés à la fin des variables de construction dans l'environnement. Si l'environnement n'a pas la variable de construction spécifiée, il est simplement ajouté à l'environnement. Si les valeurs de la variable de la construction et l'argument mot-clé sont du même type, les deux valeurs sont simplement additionnés. Dans le cas contraire, la variable de la construction et la valeur de l'argument mot-clé sont à la fois sous la contrainte à des listes et les listes sont additionnés. (Voir aussi la méthode Prepend, ci-dessous.)

Exemple:

```
env.Append (CCFLAGS = '-g', FOO = [ 'foo.yyy'])
```

`env.AppendENVPath(name, newpath, [envname, sep, delete_existing])`

Cela ajoute de nouveaux éléments de chemin d'accès au chemin donné de l'environnement extérieur spécifiée (`ENV` par défaut). Cela n'ajouter une voie particulière une fois (en laissant la dernière rencontre et en ignorant le reste, pour préserver l' ordre de chemin), et pour aider à assurer cela, normalisera tous les chemins (en utilisant `os.path.normpath` et `os.path.normcase`). Cela peut aussi gérer le cas où la variable de chemin ancienne donnée est une liste au lieu d'une chaîne, auquel cas une liste sera retournée au lieu d'une chaîne.

Si `delete_existing` est 0, puis en ajoutant un chemin qui existe déjà ne se déplace pas à la fin; il reste où il est dans la liste.

Exemple:

```
print 'avant:', env [ 'ENV'] [ 'INCLUDE']
include_path = '/ foo / bar: / foo'
env.AppendENVPath ( 'include', include_path)
print 'après:', env [ 'ENV'] [ 'INCLUDE']

rendement:
avant: / foo: / biz
après: / biz: / toto / titi: / foo
```

`env.AppendUnique(key=val, [...], delete_existing=0)`

Les arguments clés concatène spécifiés à la fin des variables de construction dans l'environnement. Si l'environnement n'a pas la variable de construction spécifiée, il est simplement ajouté à l'environnement. Si la variable de construction étant ajoutée à une liste, alors toute valeur (s) qui existent déjà dans la variable de la construction *ne* sera ajouté à nouveau à la liste. Toutefois, si delete_existing est 1, les valeurs correspondantes existantes sont supprimées d' abord, si les valeurs existantes dans la liste arg aller à la fin de la liste.

Exemple:

```
env.AppendUnique (CCFLAGS = '-g', FOO = [ 'foo.yyy'])
```

`env.BitKeeper()`

Une fonction d'usine qui retourne un objet constructeur à utiliser pour récupérer des fichiers source en utilisant BitKeeper. Le constructeur retourné est destiné à être transmis à la `SourceCode`fonction.

Cette fonction est dépréciée. Pour plus de détails, voir l'entrée pour la `SourceCode` fonction.

Exemple:

```
env.SourceCode ( '', env.BitKeeper ())
```

`BuildDir(build_dir, src_dir, [duplicate])` , `env.BuildDir(build_dir, src_dir, [duplicate])`

Synonymes déconseillés pour `VariantDir` et `env.VariantDir`(). L'`build_dir` argument de devient l' `variant_dir` argument `VariantDir`ou `env.VariantDir`().

`Builder(action, [arguments])` , `env.Builder(action, [arguments])`

Crée un objet Builder pour le spécifié `action`. Voir la section « Objets Builder » ci - dessous pour une explication complète des arguments et des comportements.

Notez que la `env.Builder`forme de l'invocation étendra les variables de construction dans toutes les chaînes d'arguments, y compris le () l' `action` argument, au moment où il est appelé en utilisant les variables de construction dans l' `env` environnement de la construction à travers lequel `env.Builder`() a été appelée. La`Builder` forme retarde toute expansion des variables jusqu'après l'objet Builder est en fait appelé.

`CacheDir(cache_dir)` , `env.CacheDir(cache_dir)`

Indique que `scons` maintiendrons un cache de fichiers dérivés dans `cache_dir`. Les fichiers dérivés du cache seront partagés entre tous les construit en utilisant le même `CacheDir` appel. Spécification d' un `cache_dir` de `None` désactive la mise en cache de fichiers dérivés.

L' appel `env.CacheDir()` n'affectera des cibles construites dans l'environnement de construction spécifié. Appel `CacheDir` fixe un défaut global qui sera utilisé par toutes les cibles construites dans des environnements de construction qui ne *pas* avoir un`env.CacheDir()` spécifié.

Lorsqu'un `CacheDir()` est utilisée et `scons` trouve un fichier dérivé qui doit être reconstruit, il va d' abord regarder dans le cache pour voir si un fichier dérivé a déjà été construit à partir de fichiers d'entrée identiques et une action de construction identique (tel que repris dans le MD5 construire la signature). Si oui, `scons` va récupérer le fichier à partir du cache. Si le fichier dérivé n'est pas présent dans le cache, `scons` va reconstruire et puis placez une copie du fichier intégré dans le cache (identifié par son MD5 construire la signature), afin qu'il puisse être récupéré par d' autres constructions qui ont besoin de construire le même fichier dérivé d'entrées identiques.

L' utilisation d'un spécifié `CacheDir` peut être désactivé pour toute invocation en utilisant l' `--cache-disable` option.

Si l' `--cache-force` option est utilisée, `scons` placera une copie de*tous* les fichiers dérivés dans le cache, même si elles existaient déjà et ne sont pas construits par cette invocation. Ceci est utile pour remplir un cache la première fois `CacheDir` est ajoutée à une construction, ou après avoir utilisé l' `--cache-disable` option.

Lors de l' utilisation `CacheDir`, `scons` fera rapport, « fichier Récupérée` » du cache, » à moins que l' `--cache-show` on utilise l' option. Lorsque l' `--cache-show` option est utilisée, `scons`imprimera l'action qui *aurait* ont été utilisées pour construire le fichier, sans aucune indication que le fichier a été effectivement récupéré à partir du cache. Ceci est utile pour générer des journaux de construction équivalentes quel que soit si un fichier dérivé donné a été construit en place ou récupéré à partir du cache.

La `NoCache` méthode peut être utilisée pour désactiver la mise en cache des fichiers spécifiques. Cela peut être utile si les entrées et / ou sorties de certains outils sont impossibles à prévoir ou trop grand.

`Clean(targets, files_or_dirs)` , `env.Clean(targets, files_or_dirs)`

Ceci indique une liste de fichiers ou de répertoires qui doivent être supprimés lorsque les cibles sont spécifiées avec l' `-c` option de ligne de commande. Les cibles spécifiées peuvent être une liste ou une cible individuelle. Plusieurs appels à `Clean` sont légaux, et de créer de nouvelles cibles ou ajouter des fichiers et des répertoires à la liste propre pour les cibles spécifiées.

Plusieurs fichiers ou répertoires doivent être spécifiés comme arguments séparés à la `Clean` méthode, ou comme une liste. `Clean`également accepter la valeur de retour de l' un des environnement de construction méthodes Builder. Exemples:

Les associés `NoClean` fonction overrides appel `Clean` à la même cible et les cibles sont passées aux fonctions seront *pas* être supprimées par l' `-c` option.

Exemples:

```
Clean ( 'foo', [ 'bar', 'baz'])
Clean ( 'dist', env.Program ( 'bonjour', 'hello.c'))
Clean ([ 'foo', 'bar'], 'something_else_to_clean')
```

Dans cet exemple, l'installation du projet crée un sous-répertoire de la documentation. Cette déclaration provoque le sous-répertoire à supprimer si le projet est désinstallée.

```
Clean (docdir, os.path.join (docdir, projectname))
```

`env.Clone([key=val, ...])`

Renvoie une copie séparée d'un environnement de construction. S'il y a des arguments de mots-clés spécifiés, ils sont ajoutés à la copie retournée, en remplaçant toutes les valeurs existantes pour les mots-clés.

Exemple:

```
ENV2 = env.Clone ()
ENV3 = env.Clone (CCFLAGS = '-g')
```

En outre, une liste d'outils et un parcours d'outil peut être spécifié, comme dans le constructeur Environnement:

```
def MyTool (env): env [ 'foo'] 'bar' =
ENV4 = env.Clone (outils = [ 'msvc', MyTool])
```

L' `parse_flags` argument mot - clé est également reconnu:

```
# Créer un environnement pour la compilation de programmes qui utilisent wxWidgets
wx_env = env.Clone (parse_flags = '! WX-config --cflags --cxxflags')
```

`Command(target, source, action, [key=val, ...])` , `env.Command(target, source, action, [key=val, ...])`

Executer une action spécifique (ou une liste d'actions) pour créer un fichier cible ou des fichiers. Ceci est plus pratique que la définition d'un objet séparé Builder pour une seule version spéciale cas.

En cas particulier, l' `source_scanner` argument de mot - clé peut être utilisé pour spécifier un objet Scanner qui sera utilisé pour analyser les sources. (Le global `DirScanner` objet peut être utilisé si l' une des sources seront les répertoires qui doivent être scannés sur disque pour des modifications aux fichiers qui ne sont pas déjà spécifiés dans d' autres Builder d'appels de fonction.)

Tous les autres arguments spécifiés par mot-clé remplacent toutes les variables de construction existantes du même nom.

Une action peut être une commande externe, spécifiée en tant que chaîne de caractères, ou un objet Python appelable; voir « Objets d'action » ci - dessous pour des informations plus complètes. Notez également qu'une chaîne spécifiant une commande externe peut être précédée d'un `@` (arobase) pour supprimer l' impression de la commande en question ou par un `-`(trait d' union) d'ignorer l'état de sortie de la commande externe.

Exemples:

```
env.Command ( 'foo.out', 'foo.in',
            "FOO_BUILD $ <$ SOURCES> $ TARGET")

env.Command ( 'bar.out', 'bar.in',
            [ "Rm -f $ TARGET",
             "BAR_BUILD $ <$ SOURCES> $ TARGET"],
            ENV = { 'PATH': '/ usr / local / bin /'})
```

```
def renommer (env, cible, de la source):
    import os
    os.rename ( 'tmp', str (target [0]))

env.Command ( 'baz.out', 'baz.in',
            [ "BAZ_BUILD $ <$ SOURCES> .tmp",
             renommer])
```

Notez que la `Command` fonction suppose généralement, par défaut, que les cibles spécifiées et / ou sources sont des fichiers, si aucune autre partie de la configuration identifie ce type d'entrée il est. Si nécessaire, vous pouvez spécifier explicitement que les cibles ou les noeuds source doivent être traités comme directoriese en utilisant les `Dir` ou `env.Dir`fonctions ().

Exemples:

```
env.Command ( 'ddd.list', Dir ( 'ddd'), 'ls -l $ SOURCE> $ TARGET')

env [ 'DISTDIR'] = 'destination / répertoire'
env.Command (env.Dir ( '$ DISTDIR')), Aucun, make_distdir)
```

(Notez également que SCons va créer automatiquement un répertoire habituellement nécessaire pour contenir un fichier cible, de sorte que vous ne normalement pas besoin de créer des répertoires à la main.)

`Configure(env, [custom_tests, conf_dir, log_file, config_h])` , `env.Configure([custom_tests, conf_dir, log_file, config_h])`

Crée un objet Configurer pour la fonctionnalité intégrée similaire à GNU autoconf. Voir la section « Configurer Contextes » ci-dessous pour une explication complète des arguments et des comportements.

`env.Copy([key=val, ...])`

Un synonyme maintenant dépréciée pour `env.Clone`().

`env.CVS(repository, module)`

Une fonction d'usine qui retourne un objet constructeur à utiliser pour récupérer des fichiers sources des CVS spécifiées `repository`. Le constructeur retourné est destiné à être transmis à la`SourceCode` fonction.

Cette fonction est dépréciée. Pour plus de détails, voir l'entrée pour la `SourceCode` fonction.

L'option spécifiée `module` sera ajouté au début de tous les noms de chemin du référentiel; cela peut être utilisé, en substance, à dépouiller les noms de répertoire initial des noms de chemin du référentiel, de sorte que vous suffit de reproduire une partie de la hiérarchie des répertoires du référentiel dans votre répertoire de construction locale.

Exemples:

```
# Va chercher foo / bar / src.c
# De /usr/local/CVSROOT/foo/bar/src.c.
env.SourceCode ( ' ' env.CVS ('/ usr / local / CVSROOT'))

# Va chercher bar / src.c
# De /usr/local/CVSROOT/foo/bar/src.c.
env.SourceCode ( ' ' env.CVS ('/ usr / local / CVSROOT', 'foo'))

# Va chercher src.c
# De /usr/local/CVSROOT/foo/bar/src.c.
env.SourceCode ( ' ' env.CVS ('/ usr / local / CVSROOT', 'foo / bar'))
```

`Decider(function)` , `env.Decider(function)`

Indique que seront traitées par toutes les décisions spécifié pour cibles mises à jour dans cet environnement construit de la construction `function`. Le `function` peut être l' une des chaînes suivantes qui spécifient le type de fonction de décision à effectuer:

`timestamp-newer`

Indique qu'une cible doit être considérée comme obsolète et reconstruit si l'horodatage de la dépendance est plus récente que l'horodatage du fichier cible. Ceci est le comportement de l'utilitaire Make classique, et `make` peut être utilisé synonyme de `timestamp-newer`.

`timestamp-match`

Indique qu'une cible doit être considérée comme obsolète et reconstruit si l'horodatage de la dépendance est différente de l'horodatage enregistré la dernière fois que la cible a été construit. Cela donne un comportement très similaire à l'utilitaire Marque classique (en particulier, les fichiers ne sont pas ouverts afin que leur contenu peut être checksummed) , sauf que la cible sera également reconstruit si un fichier de dépendance a été restauré dans une version avec un *plus tôt* horodatage, tels que peut se produire lors de la restauration des fichiers à partir des archives de sauvegarde.

`MD5`

Indique qu'une cible doit être considérée comme obsolète et reconstruit si le contenu de la dépendance a changé sine la dernière fois que la cible a été construit, tel que déterminé soit effectuer une somme de contrôle MD5 sur le contenu de la dépendance et de la comparer à la somme de contrôle enregistrée la dernière fois que la cible a été construit. `content` peut être utilisé comme synonyme de `MD5`.

`MD5-timestamp`

Indique qu'une cible doit être considérée comme obsolète et reconstruit si la dernière fois que la cible a été construit a changé le contenu de la dépendance sinusoïdale, sauf que les dépendances avec un horodatage qui correspond à la dernière fois que la cible a été reconstruit sera supposé être à -date et *non* reconstruit. Cela donne un comportement très similaire au `MD5` comportement du contenu du fichier checksum d' toujours, avec une optimisation de ne pas vérifier le contenu des fichiers dont les horodateurs ont pas changé. L'inconvénient est que SCons va *pas* détecter si un contenu de fichier a changé , mais son horodatage est le même, comme cela peut arriver dans un script automatisé qui exécute une génération, met à jour un fichier, et exécute le rebâtis, le tout dans une seule seconde.

Exemples:

```
# Utilisez les matchs de l'horodatage exact par défaut.
Décideur ( 'horodatage match)

# Utiliser les signatures MD5 contenu pour toutes les cibles construites
# Avec l'environnement de construction ci-joint.
env.Decider ( 'contenu')
```

Outre les fonctions ci - dessus déjà disponibles, l' `function`argument peut être une fonction Python réelle qui prend les trois arguments suivants:

dependency

Le noeud (fichier) qui doit faire l' `target` être reconstruit si elle a « changé » depuis la dernière TME `target` a été construit.

target

Le nœud (fichier) en cours de construction. Dans le cas normal, voici ce qui devrait se reconstruire si l' `dependency` a « changé ».

prev_ni

Les informations stockées sur l'état du `dependency` la dernière fois `target` a été construit. Cela peut être consulté pour correspondre à différentes caractéristiques de fichiers tels que l'horodatage, la taille ou la signature du contenu.

Le `function` doit renvoyer une `True` valeur (non nul) si l' `dependency`a « changé » depuis la dernière fois que le `target` construit a été (ce qui indique que la cible *doit* être reconstruit) et `False` (zéro) sinon (ce qui indique que la cible doit *pas* être reconstruit). Notez que la décision peut être prise en utilisant tous les critères sont appopriate. Ignorant une partie ou l' ensemble des arguments de la fonction est tout à fait normal.

Exemple:

```
def my_decider (dépendance, cible, prev_ni):
    retourne pas os.path.exists (str (cible))

env.Decider (my_decider)
```

`Default(targets)` , `env.Default(targets)`

Ceci indique une liste de cibles par défaut, qui sera construit par`scons` si aucune cible explicite sont données sur la ligne de commande. Plusieurs appels à `Default` sont légaux, et ajouter à la liste des cibles par défaut.

Plusieurs cibles doivent être spécifiés comme arguments séparés à la `Default` méthode, ou une liste. `Default` également accepter le nœud retourné par l' une des méthodes de constructeur d'un environnement de construction.

Exemples:

```
Par défaut ( 'foo', 'bar', 'baz')
env.Default ([ 'a', 'b', 'c'])
bonjour = env.Program ( 'bonjour', 'hello.c')
env.Default (bonjour)
```

Un argument `Default` de `None` effacera toutes les cibles par défaut. Appelle plus tard `Default` ajoutera à la (maintenant vide) liste-cible par défaut comme normal.

La liste actuelle des cibles ajoutées à l' aide de la `Default` fonction ou méthode est disponible dans la `DEFAULT_TARGETS` liste; voir ci-dessous.

`DefaultEnvironment([args])`

Crée et retourne un objet de l'environnement de la construction par défaut. Cet environnement de construction est utilisé en interne par SCons afin d'exécuter la plupart des fonctions globales dans cette liste, et pour récupérer des fichiers source transparente à partir des systèmes de gestion de code source.

`Depends(target, dependency)` , `env.Depends(target, dependency)`

Indique une dépendance explicite; le `target` sera reconstruit à chaque fois que l' `dependency` a changé. Les deux spécifié `target` et`dependency` peut être une chaîne (généralement le nom de chemin d'un fichier ou un répertoire) ou objets Node, ou une liste de chaînes ou objets Node (comme renvoyé par un appel Builder). Cela ne devrait être nécessaire pour les cas où la dépendance ne se coince pas un scanner pour le fichier.

Exemple:

```
env.Depends ( 'foo', 'autres entrées-fichier-for-foo')

mylib = env.Library ( 'mylib.c')
installed_lib = env.Install ( 'lib', mylib)
bar = env.Program ( 'bar.c')

# Prendre des dispositions pour la bibliothèque à copier dans l'installation
# Répertoire avant d'essayer de construire le programme « bar ».
# (Notez que ceci est par exemple. Une bibliothèque « réelle »
# Dépendance serait normalement configuré par le LIBS $
# et les variables LIBPATH $, ne pas utiliser un appel env.Depends ().)

env.Depends (bar, installed_lib)
```

`env.Dictionary([vars])`

Renvoie un objet dictionnaire contenant des copies de toutes les variables de construction dans l'environnement. S'il y a des noms de variables spécifiques, seules les variables de construction spécifiées sont retournées dans le dictionnaire.

Exemple:

```
dict = env.Dictionary ()
cc_dict = env.Dictionary ( 'CC', 'CCFLAGS', 'CCCOM')
```

`Dir(name, [directory])` , `env.Dir(name, [directory])`

Cela renvoie un nœud Directory, un objet qui représente le répertoire spécifié `name`. `name` peut être un chemin relatif ou absolu. `directory` est un répertoire optionnel qui sera utilisé comme répertoire parent. Si aucune `directory` est spécifié, est utilisé dans le répertoire du script en cours en tant que parent.

Si `name` une liste, SCons retourne une liste de nœuds Dir. Les variables de construction sont développées dans `name`.

Les nœuds de répertoire peuvent être utilisés partout où vous fournir une chaîne comme un nom de répertoire à une méthode Builder ou la fonction. Noeuds Directory ont des attributs et des méthodes qui sont utiles dans de nombreuses situations; voir « répertoires et fichiers nœuds » ci-dessous.

`env.Dump([key])`

Renvoie une représentation assez imprimable de l'environnement. `key`, Sinon `None`, devrait être une chaîne contenant le nom de la variable d'intérêt.

Ce SConstruct:

```
env = environnement ()
imprimer env.Dump ( 'CCCOM')
```

imprimera:

```
'$ CC -c -o $ TARGET $ CCFLAGS $ CPPFLAGS $ _CPPDEFFLAGS $ _CPPINCFLAGS SOURCES $'
```

Bien que ce SConstruct:

```
env = environnement ()
imprimer env.Dump ()
```

imprimera:

```
{ 'AR': 'ar',
  'ARCOM': '$ AR $ ARFLAGS $ CIBLES SOURCES $ \ n $ RANLIB $ RANLIBFLAGS $ TARGET',
  'ARFLAGS': [ 'r'],
  'AS': 'comme',
  'ASCOM': '$ AS $ ASFLAGS -o $ CIBLES SOURCES $',
  'ASFLAGS': [],
  ...
```

`EnsurePythonVersion(major, minor)` , `env.EnsurePythonVersion(major, minor)`

Assurez -vous que la version Python est au moins `major`. `minor`. Cette fonction vous permet d' imprimer un message d'erreur et SCons de sortie avec un code de sortie non nul si la version Python réelle ne suffit pas en retard.

Exemple:

```
EnsurePythonVersion (2,2)
```

`EnsureSConsVersion(major, minor, [revision])` , `env.EnsureSConsVersion(major, minor, [revision])`

Assurez -vous que la version SCons est au moins `major.minor`, ou`major.minor.revision`. si `revision` est spécifié. Cette fonction vous permet d' imprimer un message d'erreur et SCons de sortie avec un code de sortie non nul si la version SCons réelle ne suffit pas en retard.

Exemples:

```
EnsureSConsVersion (0,14)

EnsureSConsVersion (0,96,90)
```

`Environment([key=value, ...])` , `env.Environment([key=value, ...])`

Retour un environnement de nouvelle construction initialisé avec les spécifiées `key=value` paires.

`Execute(action, [strfunction, varlist])` , `env.Execute(action, [strfunction, varlist])`

Exécute un objet d'action. La spécifiée `action` peut être un objet d'action (voir la section « Objets d'action », ci - dessous, pour une explication complète des arguments et le comportement), ou il peut être une chaîne de ligne de commande, une liste de commandes ou exécutable fonction Python, chacun qui sera transformé en un objet action, puis exécuté. La valeur de sortie de la valeur de commande ou de retour de la fonction Python sera retourné.

Notez que `scons` affichera un message d'erreur si l'exécution`action` échoue - à savoir, sort avec ou retourne une valeur non nulle. `scons` sera *pas* , cependant, de mettre fin automatiquement la construction si spécifié `action` échoue. Si vous voulez arrêter le build en réponse à un échec `Execute` appel, vous devez explicitement vérifier une valeur de retour non nulle:

```
Exécuter (Copier ( 'file.out', 'file.in'))

si Exécuter ( "mkdir sous / dir / ectory"):
    # Le mkdir a échoué, ne pas essayer de construire.
    Sortie (1)
```

`Exit([value])` , `env.Exit([value])`

Cela dit `scons` de sortir immédiatement spécifié `value`. Une valeur de sortie par défaut `0` est utilisé (zéro) si aucune valeur est spécifiée.

`Export(vars)` , `env.Export(vars)`

Cela dit `scons` d'exporter une liste de variables à partir du fichier SConscript en cours à tous les autres fichiers SConscript. Les variables exportées sont conservées dans une collection mondiale, les appels à la suite si la `Export` volonté sur-écrire les exportations antérieures qui ont le même nom. Plusieurs noms de variables peuvent être transmises à des `Export` arguments comme distincts ou une liste. Arguments de mots - clés peuvent être utilisés pour fournir les noms et leurs valeurs. Un dictionnaire peut être utilisé pour mapper des variables à un autre nom lors de l' exportation. Les deux variables locales et variables globales peuvent être exportées.

Exemples:

```
env = environnement ()
# Faire env disponible pour tous les fichiers à importer SConscript ().
Export ( "env")

package = 'my_name'
# Faire env et package disponible pour tous les fichiers SConscript :.
Export ( "env", "paquet")

# Faire env et package disponible pour tous les fichiers SConscript:
Export ([ "paquet" "env",])

# Faire env disponible en utilisant le débogage de nom:
Export (debug = env)

# Faire env disponible en utilisant le débogage de nom:
Export ({ "debug": env})
```

Notez que la `SConscript` fonction prend en charge un `exports`argument selon lequel il est plus facile à exporter une variable ou un ensemble de variables à un seul fichier SConscript. Voir la description de la `SConscript` fonction, ci - dessous.

`File(name, [directory])` , `env.File(name, [directory])`

Cela renvoie un nœud de fichier, un objet qui représente le fichier spécifié `name`. `name` peut être un chemin relatif ou absolu.`directory` est un répertoire optionnel qui sera utilisé comme répertoire parent.

Si `name` une liste, SCons retourne une liste de nœuds de fichiers. Les variables de construction sont développées dans `name`.

Les nœuds de fichiers peuvent être utilisés partout où vous fournir une chaîne comme un nom de fichier à une méthode Builder ou la fonction. Les nœuds de fichiers ont des attributs et des méthodes qui sont utiles dans de nombreuses situations; voir « répertoires et fichiers nœuds » ci-dessous.

`FindFile(file, dirs)` , `env.FindFile(file, dirs)`

Rechercher `file` dans le chemin d' accès spécifié par `dirs`. `dirs`peut être une liste de noms de répertoire ou un nom de répertoire unique. En plus de rechercher des fichiers qui existent dans le système de fichiers, cette fonction recherche également des fichiers dérivés qui n'ont pas encore été construits.

Exemple:

```
foo = env.FindFile ( 'foo', [ 'rep1', 'dir2'])
```

`FindInstalledFiles()` , `env.FindInstalledFiles()`

Renvoie la liste des objectifs fixés par le `Install` ou les `InstallAs`constructeurs.

Cette fonction sert de méthode pratique pour sélectionner le contenu d'un paquet binaire.

Exemple:

```
Installer ( '/ bin', [ 'executable_a', 'executable_b'])

# Retourne la liste des nœuds de fichiers
# [ '/ Bin / executable_a', '/ bin / executable_b']
FindInstalledFiles ()

Installer ( '/ lib', [ 'some_library'])

# Retourne la liste des nœuds de fichiers
# [ '/ Bin / executable_a', '/ bin / executable_b', '/ lib / some_library']
FindInstalledFiles ()
```

`FindPathDirs(variable)`

Renvoie une fonction (en fait un objet Python appelable) destiné à être utilisé comme `path_function` un objet du scanner. L'objet retourné recherchera spécifié `variable` dans un environnement de construction et de traiter la valeur de la variable de la construction comme une liste de chemins d'accès qui doivent être recherchées (comme `$CPPPATH`, `$LIBPATH`, etc.).

Notez que l' utilisation `FindPathDirs` est généralement préférable d'écrire votre propre `path_function` pour les raisons suivantes: 1) La liste retournée contiendra tous les répertoires appropriés trouvés dans les arbres source (quand `VariantDir` est utilisé) ou dans des dépôts de code (quand `Repository` ou l' `-Y` option sont utilisés). 2) scons identifieront expansions de `variable` qui évaluent à la même liste des répertoires que, en fait, la même liste, et éviter une nouvelle vérification des répertoires des fichiers, lorsque cela est possible.

Exemple:

```
def my_scan (noeud, env, chemin, arg):
    Code # pour analyser le contenu des fichiers ici ...
    retour include_files

scanner = Scanner (name = 'myscanner',
                   = fonction my_scan,
                   path_function = FindPathDirs ( 'MyPath'))
```

`FindSourceFiles(node='"."')` , `env.FindSourceFiles(node='"."')`

Renvoie la liste des noeuds qui servent de la source des fichiers construits. Il le fait en inspectant le départ de l' arbre de dépendance à l'argument optionnel `node` qui est par défaut le « « » » - nœud. Il retournera ensuite toutes les feuilles de `node`. Ce sont tous les enfants qui ont pas d' autres enfants.

Cette fonction est une méthode pratique pour sélectionner le contenu d'un paquet source.

Exemple:

```
Programme ( 'src / main_a.c')
Programme ( 'src / main_b.c')
Programme ( 'main_c.c')

# retours [ 'main_c.c', 'src / main_a.c', 'SConstruct', 'src / main_b.c']
FindSourceFiles ()

# retours [ 'src / main_b.c', 'src / main_a.c']
FindSourceFiles ( 'src')
```

Comme vous pouvez le voir construire des fichiers de support (SConstruct dans l'exemple ci-dessus) sera également renvoyée par cette fonction.

`Flatten(sequence)` , `env.Flatten(sequence)`

Prend une séquence (à savoir, une liste Python ou tuple) qui peuvent contenir des séquences imbriquées et retourne une liste aplatie contenant tous les éléments individuels dans un ordre quelconque. Cela peut être utile pour recueillir les listes retournées par les appels aux constructeurs; d'autres constructeurs aplatissent automatiquement des listes spécifiées en entrée, mais la manipulation de Python direct de ces listes ne fonctionne pas.

Exemples:

```
foo = Objet ( 'foo.c')
bar = Objet ( 'bar.c')

# Parce que `foo « et `bar » sont des listes retournées par l'objet () Constructeur,
# Objets `sera une liste contenant des listes imbriquées:
objets = [ 'f1.o', foo, 'f2.o', bar, 'f3.o']

# Le passage d'une telle liste à un autre constructeur est bien parce que
# Le constructeur va aplatir automatiquement la liste:
Programme (source = objets)

# Si vous avez besoin de manipuler la liste en utilisant directement Python, vous devez
# Appel Aplatir () vous-même, ou autrement gérer des listes imbriquées:
pour objet Flatten (objets):
    print str (objet)
```

`GetBuildFailures()`

Retourne une liste d'exceptions pour les actions qui ont échoué en essayant de construire des cibles. Chaque élément de la liste retournée est un `BuildError` objet avec les attributs suivants que les aspects enregistrer différents de l'échec de construction:

`.node` Le nœud qui a été construit lorsque l'échec de la construction a eu lieu.

`.status` L'état de sortie numérique renvoyée par la fonction de commande ou Python qui a échoué lors de la tentative de construire le nœud spécifié.

`.errstr` La chaîne d'erreur SCons décrivant l'échec de la compilation. (Ceci est souvent un message générique comme « Erreur 2 » pour indiquer qu'une commande exécutée est sortie avec un statut de 2.)

`.filename` Le nom du fichier ou du répertoire effectivement causé l'échec. Cela peut être différent de l' `.node` attribut. Par exemple, si une tentative de construire une cible nommée `sub/dir/target`échoue parce que le `sub/dir` répertoire n'a pas pu être créée, l'`.node` attribut sera , `sub/dir/target` mais l' `.filename` attribut sera`sub/dir`.

`.executor` Le SCons objet Executor pour le nœud cible en cours de construction. Ceci peut être utilisé pour récupérer l'environnement de construction utilisé pour l'action a échoué.

`.action` L'objet réel SCons action qui a échoué. Ce sera une action spécifique sur la liste possible des actions qui auraient été exécutées pour construire la cible.

`.command` La commande réelle étendue qui a été exécuté et a échoué, après l' expansion de `$TARGET`, `$SOURCE`et d' autres variables de construction.

Notez que la `GetBuildFailures` fonction renvoie toujours une liste vide jusqu'à ce que toute défaillance de construction a eu lieu, ce qui signifie que `GetBuildFailures` renvoie toujours une liste vide tandis que les `SConscript` fichiers sont en cours de lecture. Son utilisation principale prévue est pour les fonctions qui seront exécutées avant SCons sorties en les passant à la norme Python`atexit.register`fonction (). Exemple:

```
importation atexit

def print_build_failures ():
    de GetBuildFailures à l'importation SCons.Script
    pour bf dans GetBuildFailures ():
        print "% a échoué de:% s" % (bf.node, bf.errstr)

atexit.register (print_build_failures)
```

`GetBuildPath(file, [...])` , `env.GetBuildPath(file, [...])`

Retourne le `scons` nom du chemin (ou les noms) pour les spécifiés`file` (ou fichiers). Les spécifiés `file` fichiers ou peuvent être des `scons` nœuds ou des chaînes représentant les noms de chemin.

`GetLaunchDir()` , `env.GetLaunchDir()`

Retourne le nom de chemin absolu du répertoire à partir duquel`scons` a été initialement invoqué. Cela peut être utile lorsque vous utilisez les `-u`, `-U` ou les `-D` options qui changent en interne au répertoire dans lequel le `SConstruct` fichier est trouvé.

`GetOption(name)` , `env.GetOption(name)`

Cette fonction fournit un moyen d'interroger la valeur des options SCons définies sur scons ligne de commande (ou un ensemble à l' aide de la `SetOption` fonction). Les options prises en charge sont les suivants :

`cache_debug`

ce qui correspond à --cache-debug;

`cache_disable`

ce qui correspond à --cache-disable;

`cache_force`

ce qui correspond à --cache force;

`cache_show`

ce qui correspond à --cache-présentation;

`clean`

ce qui correspond à -c, --clean et --remove;

`config`

ce qui correspond à --config;

`directory`

qui correspond à -C et --directory;

`diskcheck`

ce qui correspond à --diskcheck

`duplicate`

ce qui correspond à --duplicate;

`file`

ce qui correspond à -F, --file, --makefile et --sconstruct;

`help`

ce qui correspond à -H et --help;

`ignore_errors`

ce qui correspond à des erreurs --ignore-;

`implicit_cache`

ce qui correspond à --implicit-cache;

`implicit_deps_changed`

ce qui correspond à --implicit-deps-modifiés;

`implicit_deps_unchanged`

ce qui correspond à --implicit-deps-inchangé;

`interactive`

ce qui correspond à --interact et --interactive;

`keep_going`

ce qui correspond à -k et --keep en cours;

`max_drift`

ce qui correspond à --max dérive;

`no_exec`

ce qui correspond à -N, --no-exec, --just-print, --dry-run et --recon;

`no_site_dir`

ce qui correspond à --no-site-dir;

`num_jobs`

ce qui correspond à -j et --jobs;

`profile_file`

ce qui correspond à --profile;

`question`

ce qui correspond à -q et --question;

`random`

ce qui correspond à --random;

`repository`

ce qui correspond à -y, --repository et --srcdir;

`silent`

ce qui correspond à -s, --silent et --quiet;

`site_dir`

ce qui correspond à --site-dir;

`stack_size`

ce qui correspond à --stack-size;

`taskmastertrace_file`

ce qui correspond à --taskmastertrace; et

`warn`

ce qui correspond à --warn et --warning.

Consultez la documentation de l'objet de ligne de commande correspondante pour obtenir des informations sur chaque option spécifique.

`Glob(pattern, [ondisk, source, strings, exclude])` , `env.Glob(pattern, [ondisk, source, strings, exclude])`

Renvoie les nœuds (ou chaînes) qui correspondent à l' endroit spécifié `pattern`, par rapport au répertoire du courant `SConscript`fichier. La `env.Glob`forme () effectue Substition chaîne sur `pattern`et retourne tout ce qui correspond à la résultante modèle élargi.

Spécifié `pattern` utilise métacaractères de style shell Unix pour la correspondance:

```
* Correspond à tout
? un caractère unique
[Suivants] correspond à tout caractère suivants
[! Seq] correspond à tout omble chevalier non seq
```

Si le premier caractère d'un nom de fichier est un point, il doit être mis en correspondance explicitement. Matchs de caractères ne *pas* couvrent des séparateurs de répertoires.

Le `Glob` connaît des dépôts (voir la `Repository` fonction) et les répertoires source (voir la `VariantDir` fonction) et retourne un nœud (ou la chaîne, le cas échéant configuré) dans le répertoire local (SConscript) si correspondant à nœud se trouve nulle part dans un répertoire de dépôt ou d'une source correspondant.

L' `ondisk` argument peut être réglé sur `False` (ou toute autre valeur non réelle) pour désactiver la recherche de correspondances sur le disque, ce qui ne retour correspondances entre fichiers ou noeuds Dir déjà configuré. Le comportement par défaut est de retour noeuds correspondants pour toutes les correspondances sur disque trouvés.

L' `source` argument peut être réglé sur `True` (ou toute valeur équivalente) pour indiquer que, lorsque le répertoire local est un`VariantDir`, les nœuds doivent être retournés à partir du répertoire source correspondant, pas le répertoire local.

L' `strings` argument peut être réglé sur `True` (ou toute valeur équivalente) d'avoir les `Glob` chaînes de retour de la fonction, pas des nœuds, qui représentent les fichiers correspondants ou des répertoires. Les chaînes renvoyées seront par rapport au répertoire local (SConscript). (Notez que cela peut le rendre plus facile à effectuer des manipulations arbitraires des noms de fichiers, mais si les chaînes renvoyées sont transmises à un autre`SConscript` fichier, traduction de nœud sera par rapport à l'autre`SConscript` répertoire, pas l'original `SConscript` répertoire.)

L' `exclude` argument peut être réglé sur un motif ou d' une liste de motifs (suivant la même sémantique shell Unix) qui doit être filtré à partir d'éléments retournés. Les éléments correspondant à un moins un motif de cette liste seront exclus.

Exemples:

```
Programme ( 'foo', Glob ( '*. C'))
Zip ( '/ tmp / tout', Glob ( '?? * ') + Glob (' *'))
sources = glob ( '*. cpp', excluent = [ 'os _ * _ * _ spécifique. cpp']) + Glob ( 'os_% s_specific _ *. cpp' % currentOS)
```

`Help(text, append=False)` , `env.Help(text, append=False)`

Ceci indique le texte d'aide à imprimer si l' `-h` argument est donné `scons`. Si l' `Help` on appelle plusieurs fois, le texte est joint en annexe ainsi que dans l'ordre `Help` appelé. Avec append à Faux, tout `Help` texte généré avec `AddOption` est mis à mal. Si append est vrai, l'aide de addOption est préfixé à la chaîne d'aide, préservant ainsi le `-h` message.

`Ignore(target, dependency)` , `env.Ignore(target, dependency)`

Le fichier de dépendance spécifié (s) sera ignorée au moment de décider si le fichier cible (s) doivent être reconstruits.

Vous pouvez également utiliser `Ignore` pour supprimer une cible de la construction par défaut. Pour ce faire , vous devez spécifier le répertoire cible sera construit en tant que cible, et le fichier que vous voulez sauter bâtiment que la dépendance.

Notez que cela ne supprime les dépendances répertoriées à partir des fichiers compilés par défaut. Il sera toujours construit si cette dépendance est nécessaire par un autre objet en cours de construction. Voir les troisième et quatrième exemples ci-dessous.

Exemples:

```
env.Ignore ( 'foo', 'foo.c')
env.Ignore ( 'bar', [ 'bar1.h', 'bar2.h'])
env.Ignore ( ' '' foobar.obj')
env.Ignore ( 'bar', 'bar / foobar.obj')
```

`Import(vars)` , `env.Import(vars)`

Cela dit `scons` importer une liste de variables dans le fichier SConscript en cours. Cela importera des variables qui ont été exportées avec `Export` ou dans l' `exports` argument `SConscript`. Les variables exportées par `SConscript` ont la priorité. Plusieurs noms de variables peuvent être transmises à des `Import` arguments comme distincts ou une liste. La variable « * » peut être utilisé pour importer toutes les variables.

Exemples:

```
Import ( "env")
Import ( "env", "variable")
Importation ([ "env", "variable"])
Importer("*")
```

`Literal(string)` , `env.Literal(string)`

Spécifié `string` sera conservé en l' état et ne pas avoir des variables de construction élargies.

`Local(targets)` , `env.Local(targets)`

Spécifié `targets` aura des copies faites dans l'arbre local, même si une déjà à jour une copie existe dans un référentiel. Retourne une liste du nœud cible ou nœuds.

`env.MergeFlags(arg, [unique])`

Fusionne les spécifiées `arg` valeurs aux variables de construction de l'environnement de la construction. Si l' `arg` argument n'est pas un dictionnaire, il est converti en un en appelant`env.ParseFlags` l'argument avant que les valeurs sont fusionnées. Notez que `arg` doit être une valeur unique, donc plusieurs chaînes doivent être transmis en tant que liste, non pas comme arguments séparés pour `env.MergeFlags`.

Par défaut, les valeurs en double sont éliminées; vous pouvez toutefois spécifier `unique=0` pour que les valeurs en double à ajouter. Lorsque l' élimination des doublons, toutes les variables de construction qui se terminent par la chaîne `PATH` garder la valeur la plus à gauche unique. Toutes les autres variables de construction gardent la valeur la plus à droite unique.

Exemples:

```
# Ajouter un drapeau d'optimisation à $ CCFLAGS.
env.MergeFlags ( '- O3')

# Combinons les drapeaux sont revenus de l'exécution pkg-config avec une optimisation
# Drapeau et fusionner le résultat dans les variables de construction.
env.MergeFlags ([ '! pkg-config gtk + -2,0 --cflags', '-O3'])

# Combiner un drapeau d'optimisation avec les drapeaux sont retournés en cours d'exécution de pkg-config
# Deux fois et fusionner le résultat dans les variables de construction.
env.MergeFlags ([ '- O3',
                  '! Pkg-config gtk + -2,0 --cflags --libs',
                  '! Pkg-config libpng12 --cflags --libs'])
```

`NoCache(target, ...)` , `env.NoCache(target, ...)`

Indique une liste de fichiers qui doivent *pas* être mises en cache chaque fois que la `CacheDir` méthode a été activée. Les cibles spécifiées peuvent être une liste ou une cible individuelle.

Plusieurs fichiers doivent être spécifiés comme arguments séparés à la `NoCache` méthode, ou comme une liste. `NoCache`également accepter la valeur de retour de l' un des environnement de construction méthodes Builder.

L' appel `NoCache` sur les répertoires et d' autres types de nœuds non-fichier n'a pas d' effet parce que les nœuds de fichiers sont mis en cache.

Exemples:

```
NoCache ( 'foo.elf')
NoCache (env.Program ( 'bonjour', 'hello.c'))
```

`NoClean(target, ...)` , `env.NoClean(target, ...)`

Indique une liste de fichiers ou de répertoires qui doivent *pas* être supprimés chaque fois que les cibles (ou leurs dépendances) sont spécifiées avec l' `-c` option de ligne de commande. Les cibles spécifiées peuvent être une liste ou une cible individuelle. Plusieurs appels à `NoClean` sont légaux, et empêcher chaque cible spécifiée d'être retirée par des appels à l' `-c` option.

Plusieurs fichiers ou répertoires doivent être spécifiés comme arguments séparés à la `NoClean` méthode, ou comme une liste.`NoClean` également accepter la valeur de retour de l' un des environnement de construction méthodes Builder.

Appel `NoClean` pour une cible prioritaire sur l' appel `Clean` pour la même cible, et les cibles sont passées aux fonctions sera *pas* être supprimé par l' `-c` option.

Exemples:

```
Noclean ( 'foo.elf')
Noclean (env.Program ( 'bonjour', 'hello.c'))
```

`env.ParseConfig(command, [function, unique])`

Appels spécifié `function` pour modifier l'environnement tel que spécifié par la sortie `command`. La valeur par défaut `function` est`env.MergeFlags`, qui attend la sortie d'un type -config * commande (par exemple, gtk-config) et ajoute les options aux variables de construction appropriées. Par défaut, les valeurs en double ne sont pas ajoutées aux variables de la construction; vous pouvez spécifier `unique=0` pour que les valeurs en double à ajouter.

Les options et les variables Interprété de construction qu'ils affectent sont tels que spécifiés pour la `env.ParseFlags` méthode (qui appelle cette méthode). Voir la description de cette méthode, ci - dessous, pour une table d'options et les variables de construction.

`ParseDepends(filename, [must_exist, only_one])` , `env.ParseDepends(filename, [must_exist, only_one])`

Parse le contenu spécifié `filename` comme une liste de dépendances dans le style de Marque ou mkdep , et établit explicitement toutes les dépendances énumérées.

Par défaut, ce n'est pas une erreur si spécifié `filename` n'existe pas. L'option `must_exist` argument peut être réglé à une valeur non nulle d'avoir scons renvoient une exception et de générer une erreur si le fichier n'existe pas, ou est autrement inaccessible.

L'option `only_one` argument peut être réglé à une valeur non nulle d'avoir jeté scons une exception et générer une erreur si le fichier contient des informations de dépendance pour plus d'une cible. Cela peut fournir un petit chèque de santé mentale pour les fichiers destinés à générer, par exemple, le `gcc -M` drapeau, qui devrait normalement écrire que les informations de dépendance pour un fichier de sortie en un correspondant `.d` fichier.

Le `filename` et tous les fichiers qui y sont énumérés seront interprétés par rapport au répertoire du `SConscript` fichier qui appelle la `ParseDepends` fonction.

`env.ParseFlags(flags, ...)`

Parse une ou plusieurs chaînes contenant des drapeaux de ligne de commande typiques pour les chaînes d'outils GCC et renvoie un dictionnaire avec les valeurs d'indicateur séparés dans les variables de construction de SCons appropriées. Ceci est conçu comme un compagnon à la `env.MergeFlags` méthode, mais permet les valeurs dans le cas échéant, avant de les fusionner dictionnaire retourné à modifier, dans l'environnement de la construction. (Notez que `env.MergeFlags` appellera cette méthode si son argument n'est pas un dictionnaire, il est donc généralement pas nécessaire d'appeler `env.ParseFlags`directement à moins que vous souhaitez manipuler les valeurs.)

Si le premier caractère dans une chaîne est un point d'exclamation (!), Le reste de la chaîne est exécutée comme une commande, et la sortie de la commande est analysée comme des drapeaux de ligne de commande de la chaîne d'outil GCC et ajouté au dictionnaire résultant.

Les valeurs de drapeau sont convertis accordig au préfixe trouvé et ajouté aux variables de construction suivantes:

```
-arch CCFLAGS, LINKFLAGS
-D CPPDEFINES
-Cadre CADRES
-frameworkdir = FRAMEWORKPATH
-include CCFLAGS
-isysroot CCFLAGS, LINKFLAGS
-I CPPPATH
-l LIBS
-L LIBPATH
-mno-Cygwin CCFLAGS, LINKFLAGS
-mwindows LINKFLAGS
-pthread CCFLAGS, LINKFLAGS
std = CFLAGS
-wa, ASFLAGS, CCFLAGS
-wl, -rpath = RPATH
-Wl, -R, RPATH
-Wl, -R RPATH
-wl, LINKFLAGS
-wp, CPPFLAGS
- CCFLAGS
+ CCFLAGS, LINKFLAGS
```

Toutes les autres chaînes ne sont pas associés avec des options sont supposés être les noms des bibliothèques et ajoutées à la`$LIBS` variable de la construction.

Des exemples (tous qui produisent le même résultat):

```
dict = env.ParseFlags ( '- O2 -Dfoo -Dbar = 1)
dict = env.ParseFlags ( '- O2', '-Dfoo', '-Dbar = 1)
dict = env.ParseFlags ([ '- O2', '-Dfoo -Dbar = 1'])
dict = env.ParseFlags (- '! echo -Dfoo -Dbar = 1' 'O2',)
```

`env.Perforce()`

Une fonction d'usine qui retourne un objet constructeur à utiliser pour récupérer des fichiers sources à partir du système de gestion de code source Perforce. Le constructeur retourné est destiné à être transmis à la `SourceCode` fonction.

Cette fonction est dépréciée. Pour plus de détails, voir l'entrée pour la `SourceCode` fonction.

Exemple:

```
env.SourceCode ( '', env.Perforce ())
```

Perforce utilise un certain nombre de variables d'environnement externes pour son fonctionnement. Par conséquent, cette fonction ajoute les variables suivantes de l'environnement externe de l'utilisateur au dictionnaire ENV de l'environnement de construction: P4CHARSET, P4CLIENT, P4LANGUAGE, P4PASSWD, P4PORT, P4USER, SystemRoot, USER et USERNAME.

`Platform(string)`

La `Platform` forme retourne un objet appelable qui peut être utilisé pour initialiser un environnement de construction en utilisant le mot - clé de la plate - forme de la `Environment` fonction.

Exemple:

```
env = environnement (plate-forme = Plate-forme ( 'win32'))
```

La `env.Platform` forme applique l'objet appelable pour la plate - forme spécifiée `string` à l'environnement à travers lequel la méthode a été appelée.

```
env.Platform ( 'posix')
```

Notez que la `win32` plate - forme ajoute les `SystemDrive` et les `SystemRoot` variables de l' environnement externe de l'utilisateur de l'environnement de construction `$ENV` dictionnaire. Il en est ainsi que toutes les commandes exécutées qui utilisent des prises pour se connecter avec d' autres systèmes (tels que la récupération des fichiers sources à partir des spécifications du référentiel CVS externes comme`:pserver:anonymous@cvs.sourceforge.net:/cvsroot/scons`) fonctionnent sur les systèmes Windows.

`Precious(target, ...)` , `env.Precious(target, ...)`

Marques chaque donnée `target` aussi précieux de sorte qu'il ne soit pas supprimé avant qu'il ne soit reconstruit. Normalement , `scons` supprime une cible avant de la construire. Cibles multiples peuvent être passés pour un seul appel à `Precious`.

`env.Prepend(key=val, [...])`

Les arguments ajoute de mots-clés spécifiés au début des variables de construction dans l'environnement. Si l'environnement n'a pas la variable de construction spécifiée, il est simplement ajouté à l'environnement. Si les valeurs de la variable de la construction et l'argument mot-clé sont du même type, les deux valeurs sont simplement additionnés. Dans le cas contraire, la variable de la construction et la valeur de l'argument mot-clé sont à la fois sous la contrainte à des listes et les listes sont additionnés. (Voir aussi la méthode Append, ci-dessus.)

Exemple:

```
env.Prepend (CCFLAGS = '-g', FOO = [ 'foo.yyy'])
```

`env.PrependENVPath(name, newpath, [envname, sep, delete_existing])`

Cela ajoute de nouveaux éléments de chemin d'accès au chemin donné de l'environnement extérieur spécifiée (`$ENV` par défaut). Cela n'ajouter une voie particulière une fois (en laissant la première rencontre et en ignorant le reste, pour préserver l' ordre de chemin), et pour aider à assurer cela, normalisera tous les chemins (en utilisant `os.path.normpath` et `os.path.normcase`). Cela peut aussi gérer le cas où la variable de chemin ancienne donnée est une liste au lieu d'une chaîne, auquel cas une liste sera retournée au lieu d'une chaîne.

Si `delete_existing` est 0, puis en ajoutant un chemin qui existe déjà ne se déplace pas au début; il reste où il est dans la liste.

Exemple:

```
print 'avant:', env [ 'ENV'] [ 'INCLUDE']
include_path = '/ foo / bar: / foo'
env.PrependENVPath ( 'include', include_path)
print 'après:', env [ 'ENV'] [ 'INCLUDE']
```

L'exemple ci-dessus affichera:

```
avant: / biz: / foo
après: / toto / titi: / foo: / biz
```

`env.PrependUnique(key=val, delete_existing=0, [...])`

Les arguments ajoute de mots - clés spécifiés au début des variables de construction dans l'environnement. Si l'environnement n'a pas la variable de construction spécifiée, il est simplement ajouté à l'environnement. Si la variable de construction étant ajoutée à une liste, alors toute valeur (s) qui existent déjà dans la variable de la construction *ne* sera ajouté à nouveau à la liste. Toutefois, si delete_existing est 1, les valeurs correspondantes existantes sont supprimées d' abord, si les valeurs existantes dans la liste arg déplacer à l'avant de la liste.

Exemple:

```
env.PrependUnique (CCFLAGS = '-g', FOO = [ 'foo.yyy'])
```

`Progress(callable, [interval]) , , Progress(string, [interval, file, overwrite]) Progress(list_of_strings, [interval, file, overwrite])`

Permet SCons de montrer les progrès réalisés lors de la construction en affichant une chaîne ou d'appeler une fonction tout en évaluant les nœuds (par exemple, fichiers).

Si le premier argument spécifié est un Python appelable (une fonction ou un objet qui a une `__call__`méthode ()), la fonction sera appelée une fois tous les `interval` temps un nœud est évaluée. Le appelable sera transmis le noeud évalué comme seul argument. (Pour la compatibilité future, il est une bonne idée d'ajouter aussi `*args` et `**kw` comme arguments à votre fonction ou méthode. Cela empêchera le code de rupture si SCons ne change jamais l'interface pour appeler la fonction avec des arguments supplémentaires à l'avenir.)

Un exemple d'une fonction de progression personnalisé simple qui imprime une chaîne contenant le nom du noeud tous les 10 nœuds:

```
def my_progress_function (noeud, * args, ** kw):
    print 'évaluation noeud% s! % noeud
Progress (my_progress_function, intervalle = 10)
```

Un exemple plus complexe d'un objet d'affichage de progression personnalisée qui imprime une chaîne contenant un compte 100 nœuds évalués. Notez l'utilisation de `\r` (un retour chariot) à la fin afin que la chaîne va s'écraser sur un écran:

```
import sys
ProgressCounter classe (objet):
    count = 0
    def __call __ (self, noeud, * args, ** kw):
        self.count + = 100
        sys.stderr.write ( '% de Evalué noeuds \ r' % self.count)
Progress (ProgressCounter (), intervalle = 100)
```

Si le premier argument `Progress` est une chaîne, la chaîne sera affiché tous les `interval` nœuds évalués. La valeur par défaut est d'imprimer la chaîne sur la sortie standard; un courant de sortie alternatif peut être spécifié avec l' `file=` argument. Ce qui suit imprimer une série de points sur la sortie d'erreur, un point pour chaque tranche de 100 nœuds évalués:

```
import sys
Progress ( '', intervalle = 100, file = sys.stderr)
```

Si la chaîne contient la sous - mot pour mot `$TARGET`, il sera remplacé par le nœud. Notez que, pour des raisons de performance, c'est *pas* une SCons régulière Substition variables, de sorte que vous ne pouvez pas utiliser d' autres variables ou utiliser des accolades. L'exemple suivant imprime le nom de chaque nœud, à l' aide d' une évaluation `\r` (retour chariot) pour que chaque ligne écrasée par la ligne suivante, et l' `overwrite=`argument mot - clé pour vous assurer que le nom du fichier précédemment imprimé est remplacé par des espaces vides:

```
import sys
Progrès ( '$ TARGET \ r', remplacer = True)
```

Si le premier argument `Progress` est une liste de chaînes, chaque chaîne dans la liste sera affichée en mode rotation tous les `interval` nœuds évalués. Cela peut être utilisé pour mettre en œuvre un « spinner » sur l'écran de l'utilisateur comme suit:

```
Progrès ([ '- \ r', 'r \\\', '| \ r', '/ \ r'], intervalle = 5)
```

`Pseudo(target, ...) , env.Pseudo(target, ...)`

Cela indique que chaque donnée `target` ne doit pas être créé par la règle de construction, et si la cible est créée, une erreur sera générée. Ceci est similaire à la cible gnu font .PHONY. Cependant, dans la grande majorité des cas, un `Alias` est plus approprié. Cibles multiples peuvent être

passés pour un seul appel à `Pseudo`.

`env.RCS()`

Une fonction d'usine qui retourne un objet constructeur à utiliser pour récupérer des fichiers source du RCS. Le constructeur retourné est destiné à être transmis à la `SourceCode` fonction:

Cette fonction est dépréciée. Pour plus de détails, voir l'entrée pour la `SourceCode` fonction.

Exemples:

```
env.SourceCode ( '', env.RCS ())
```

Notez que `scons` va chercher des fichiers sources à partir des sous - répertoires RCS automatiquement, configuration RCS comme le montre l'exemple ci - dessus ne devrait être nécessaire si vous récupérez de RCS, les fichiers v dans le même répertoire que les fichiers source, ou si vous devez spécifier explicitement RCS pour un sous - répertoire spécifique.

`env.Replace(key=val, [...])`

Remplace les variables de construction dans l'environnement avec les arguments spécifiés par mot-clé.

Exemple:

```
env.Replace (CCFLAGS = '-g', FOO = 'foo.xxx')
```

`Repository(directory)` , `env.Repository(directory)`

Indique que `directory` est un référentiel à rechercher des fichiers. Plusieurs appels à `Repository` sont légaux, et chacun ajoute à la liste des dépôts qui sera recherché.

Pour `scons`, un dépôt est une copie de l'arbre source, à partir du répertoire de haut niveau sur le bas, qui peut contenir des fichiers source et les fichiers dérivés qui peuvent être utilisés pour construire des cibles dans l'arborescence de source locale. L'exemple canonique serait un arbre source officielle tenue par un intégrateur. Si le référentiel contient des fichiers dérivés, les fichiers dérivés auraient dû être construits à l' aide `scons`, de sorte que le dépôt contient les informations de signature nécessaire pour permettre `scons` de savoir quand il convient d'utiliser la copie de dépôt d'un fichier dérivé, au lieu de construire un local .

Notez que si un fichier dérivé mise à jour existe déjà dans un dépôt, `scons` ne *pas* faire une copie dans l'arborescence du répertoire local. Afin de garantir que sera une copie locale, utilisez la `Local` méthode.

`Requires(target, prerequisite)` , `env.Requires(target, prerequisite)`

Spécifie une relation d'ordre uniquement entre le fichier cible spécifié (s) et le fichier de condition spécifiée (s). Le fichier condition (s) sera (re) construit, le cas échéant, *avant* le fichier cible (s), mais le fichier cible (s) ne dépendent pas réellement sur les conditions préalables et ne sera pas reconstruit simplement parce que le fichier condition (s) changement.

Exemple:

```
env.Requires ( 'foo', 'fichier qui doit-être-avant-intégré-foo')
```

`Return([vars..., stop=])`

Par défaut, ce qui arrête le traitement du fichier SConscript en cours et retourne au fichier SConscript appelant les valeurs des variables nommées dans les `vars` arguments de chaîne. Plusieurs chaînes contaning noms de variables peuvent être transmises à`Return`. Toutes les chaînes qui contiennent un espace blanc

L'option `stop=` argument mot - clé peut être réglé sur une valeur fausse pour continuer à traiter le reste du fichier SConscript après l' `Return` appel. Cela a été le comportement par défaut avant SCons 0,98. Cependant, les valeurs retournées sont toujours les valeurs des variables du nom `vars` au point `Return` est appelé.

Exemples:

```
# Les retours sans retourner une valeur.
Revenir()

# Renvoie la valeur de la « foo » variables Python.
Retour ( "foo")

# Renvoie les valeurs de foo 'les variables Python et « bar ».
Retour ( "foo", "bar")

# Renvoie les valeurs des variables Python de la val1 'et 'val2'.
Retour ( 'val1 val2')
```

`Scanner(function, [argument, keys, path_function, node_class, node_factory, scan_check, recursive])` , `env.Scanner(function, [argument, keys, path_function, node_class, node_factory, scan_check, recursive])`

Crée pour l'objet spécifié un scanner `function`. Voir la section « Objets du scanner » ci - dessous pour une explication complète des arguments et des comportements.

`env.SCCS()`

Une fonction d'usine qui retourne un objet constructeur à utiliser pour récupérer des fichiers sources à partir CSSC. Le constructeur retourné est destiné à être transmis à la `SourceCode`fonction.

Exemple:

```
env.SourceCode ( '', env.SCCS ())
```

Notez que `scons` va chercher des fichiers sources à partir des sous - répertoires SCCS automatiquement, configuration CSSC comme le montre l'exemple ci - dessus ne devrait être nécessaire si vous récupérez de `s.SCCS` fichiers dans le même répertoire que les fichiers source, ou si vous devez spécifier explicitement CSSC pour un particulier sous - répertoire.

SConscript(scripts, [exports, variant_dir, duplicate]) , , ,env.SConscript(scripts, [exports, variant_dir, duplicate])SConscript(dirs=subdirs, [name=script, exports, variant_dir, duplicate]) env.SConscript(dirs=subdirs, [name=script, exports, variant_dir, duplicate])

Cela dit scons d'exécuter une ou plusieurs filiales fichiers SConscript (configuration). Toutes les variables renvoyées par un script appelé à l' aide Return seront renvoyés par l'appel àSConscript. Il y a deux façons d'appeler la SConscript fonction.

La première façon , vous pouvez appeler SConscript est de spécifier explicitement un ou plusieurs scripts comme premier argument. Un script unique peut être spécifié comme une chaîne; plusieurs scripts doivent être spécifiés comme une liste (soit explicitement , soit créé par une fonction comme Split). Exemples:

```
SConscript ( 'SConscript') # faites tourner SConscript dans le répertoire courant
SConscript ( 'src / SConscript') # faites tourner SConscript dans le répertoire src
SConscript ([ 'src / SConscript', 'doc / SConscript'])
config = SConscript ( 'MyConfig.py')
```

La deuxième façon , vous pouvez appeler SConscript est de spécifier une liste de (sous) les noms de répertoires comme undirs=subdirs argument de mot - clé. Dans ce cas, scons sera, par défaut, exécuter un fichier de configuration filiale nomméeSConscript dans chacun des répertoires spécifiés. Vous pouvez spécifier un autre nom que SConscript en fournissant un optionname=script argument mot - clé. Les trois premiers exemples ci - dessous ont le même effet que les trois premiers exemples ci - dessus:

```
SConscript (dirs = '') # run SConscript dans le répertoire courant
SConscript ( 'src' dirs =) # exécuter SConscript dans le répertoire src
SConscript (dirs = [ 'src', 'doc'])
SConscript (dirs = [ 'SUB1', 'SUB2'], name = 'MySConscript')
```

L'option l' exports argument fournit une liste de noms de variables ou un dictionnaire de valeurs nommées à exporter vers la script(s). Ces variables sont localement exportées uniquement spécifié script(s), et ne touchent pas le pool global des variables utilisées par la Export fonction. La filiale script(s) doit utiliser laImport fonction pour importer les variables. Exemples:

```
foo = SConscript ( 'sub / SConscript', 'env' exportation =)
SConscript ( 'dir / SConscript', exporte = [ 'env', 'variable'])
SConscript (dirs = 'subdir', 'exportations variables env' =)
SConscript (dirs = [ 'un', 'deux', 'trois'], exporte = 'shared_info')
```

Si l'option variant_dir argument est présent, il provoque un effet équivalent à la VariantDir méthode décrite ci - dessous. (Sivariant_dir n'est pas présent, l' duplicate argument est ignoré.) L'variant_dir argument est interprété par rapport au répertoire de l'appel SConscript fichier. Voir la description de la VariantDirfonction ci - dessous pour plus de détails et les restrictions.

Si variant_dir est présent, le répertoire source est le répertoire dans lequel le SConscript fichier réside et le SConscript fichier est évalué comme si elle était dans le variant_dir répertoire:

```
SConscript ( 'src / SConscript', variant_dir = 'construire')
```

est équivalent à

```
VariantDir ( 'build', 'src')
SConscript ( 'build / SConscript')
```

Ce paradigme plus tard est souvent utilisé lorsque les sources sont dans le même répertoire que le SConstruct:

```
SConscript ( 'SConscript', variant_dir = 'construire')
```

est équivalent à

```
VariantDir ( 'construire', '')
SConscript ( 'build / SConscript')
```

Voici quelques exemples composites:

```
# Recueillir les informations de configuration et de l'utiliser pour construire et src doc
shared_info = SConscript ( 'MyConfig.py')
SConscript ( 'src / SConscript', exporte = 'shared_info')
SConscript ( 'doc / SConscript', exportations = 'shared_info')

# Débogage de construction et les versions de production. SConscript
# Peut utiliser Dir ( '') chemin. Pour déterminer la variante.
SConscript ( 'SConscript', variant_dir = 'debug', dupliquer = 0)
SConscript ( 'SConscript', variant_dir = 'prod', dupliquer = 0)

# Débogage de construction et les versions de production. SConscript
# Est passé des drapeaux à utiliser.
opts = { 'CPPDEFINES': [ 'DEBUG'], 'CCFLAGS': '-pgdb'}
SConscript ( 'SConscript', variant_dir = 'debug', dupliquer = 0, les exportations = opts)
opts = { 'CPPDEFINES': [ 'NODEBUG'], 'CCFLAGS': 'O'}
SConscript ( 'SConscript', variant_dir = 'prod', dupliquer = 0, = exportations opte)

# Construire une documentation commune et compiler pour différentes architectures
SConscript ( 'doc / SConscript', variant_dir = 'build / doc', en double = 0)
SConscript ( 'src / SConscript', variant_dir = 'build / x86', en double = 0)
SConscript ( 'src / SConscript', variant_dir = 'build / ppc', en double = 0)
```

SConscriptChdir(value) , env.SConscriptChdir(value)

Par défaut, scons modifie son répertoire de travail dans le répertoire dans lequel chaque filiale de vie de fichier SConscript. Ce comportement peut être désactivé en spécifiant:

```
SConscriptChdir (0)
env.SConscriptChdir (0)
```

auquel cas scons restera dans le répertoire de haut niveau lors de la lecture de tous les fichiers SConscript. (Cela peut être nécessaire lors de la construction de référentiels, lorsque tous les répertoires dans lesquels les fichiers SConscript peuvent être trouvés n'existent pas nécessairement sur place.) Vous pouvez activer et désactiver cette capacité en appelant SConscriptChdir () à plusieurs reprises.

Exemple:

```
env = environnement ()
SConscriptChdir (0)
SConscript ( 'foo / SConscript') # ne chdir à foo
env.SConscriptChdir (1)
SConscript ( 'bar / SConscript') # sera chdir à la barre
```

`SConsignFile([file, dbm_module]) , env.SConsignFile([file, dbm_module])`

Cela indique `scons` de stocker toutes les signatures de fichiers dans la base de données spécifiée `file`. Si le `file` nom est omis, `.sconsign` est utilisé par défaut. (Le nom du fichier (s) stocké sur le disque peut avoir un suffixe approprié en annexe par le `dbm_module`.) Si `file` n'est pas un nom de chemin absolu, le fichier est placé dans le même répertoire que le haut niveau `SConstruct`fichier.

Si `file` est `None`, puis `scons` stockera les signatures de fichiers dans un document distinct `.sconsign` fichier dans chaque répertoire, pas dans un seul fichier de base de données mondiale. (Ce fut le comportement par défaut avant SCons 0.96.91 et 0.97.)

L'option `dbm_module` argument peut être utilisé pour spécifier quel module de base de données Python La valeur par défaut est d'utiliser un custom `SCons.dblite` module qui utilise des structures de données Python décapé et qui fonctionne sur toutes les versions de Python.

Exemples:

```
# Stocke les signatures dans Explicitement « .sconsign.dblite »
# Dans le répertoire SConstruct haut niveau (la
# Comportement par défaut).
SConsignFile ()

# Signatures Magasins dans le fichier "/ etc SCons signatures"
# Par rapport au répertoire SConstruct haut niveau.
SConsignFile ( "etc / SCons signatures")

# Signatures Magasins dans le nom de fichier absolu spécifié.
SConsignFile ( "/ home / moi / SCons / signatures")

# Signatures stocke dans un fichier séparé .sconsign
# Dans chaque répertoire.
SConsignFile (Aucun)
```

`env.SetDefault(key=val, [...])`

Définit les variables de construction à des valeurs par défaut spécifiées avec les arguments de mot-clé si (et seulement si) les variables ne sont pas déjà définies. Les affirmations suivantes sont équivalentes:

```
env.SetDefault (FOO = 'foo')

si 'FOO' non env: env [ 'FOO'] = 'foo'
```

`SetOption(name, value) , env.SetOption(name, value)`

Cette fonction fournit un moyen de définir un sous-ensemble des options de ligne de commande à partir d'un fichier scons SConscript. Les options prises en charge sont les suivants:

`clean`

ce qui correspond à -c, --clean et --remove;

`duplicate`

ce qui correspond à --duplicate;

`help`

ce qui correspond à -H et --help;

`implicit_cache`

ce qui correspond à --implicit-cache;

`max_drift`

ce qui correspond à --max dérive;

`no_exec`

ce qui correspond à -N, --no-exec, --just-print, --dry-run et --recon;

`num_jobs`

ce qui correspond à -j et --jobs;

`random`

ce qui correspond à --random; et

`stack_size`

ce qui correspond à --stack dimension.

Consultez la documentation de l'objet de ligne de commande correspondante pour obtenir des informations sur chaque option spécifique.

Exemple:

```
SetOption ( 'max_drift', 1)
```

`SideEffect(side_effect, target) , env.SideEffect(side_effect, target)`

Déclare `side_effect` comme un effet secondaire de la construction`target`. Les deux `side_effect` et `target` peut être une liste, un nom de fichier ou un nœud. Un effet secondaire est un fichier cible qui est créé ou mis à jour comme un effet secondaire de la construction d' autres cibles. Par exemple, un fichier PDB Windows est créé comme un effet secondaire de la construction des fichiers OBJ pour une bibliothèque statique, et divers fichiers journaux sont mis à jour des effets créé secondaires des différentes commandes TeX. Si une cible est un effet secondaire de plusieurs commandes de construction, `scons` fera en sorte que seul un ensemble de commandes est exécutée à la fois. Par conséquent, il vous suffit d'utiliser cette méthode pour des cibles d'effets secondaires qui sont construits à la suite de plusieurs commandes de construction.

Étant donné que plusieurs commandes de compilation peuvent mettre à jour le même fichier d'effets secondaires, par défaut , la`side_effect` cible est *pas* automatiquement supprimé lorsque l'`target` on retire par l' `-c` option. (Notez, cependant, que la`side_effect` peut être retirée dans le cadre du nettoyage du répertoire dans lequel il vit.) Si vous voulez vous assurer que l'`side_effect` est nettoyée chaque fois qu'un particulier `target` est nettoyé, vous devez spécifier explicitement la `Clean` ou `env.Clean`fonction.

`SourceCode(entries, builder)` , `env.SourceCode(entries, builder)`

Cette fonction et ses fonctions d'usine associé sont déconseillés. Il n'y a pas de remplacement. L'utilisation prévue était de garder un arbre local en phase avec une archive, mais en réalité , la fonction fait que l'archive soit tiré par les cheveux sur la première manche. Synchronisation avec l'archive est mieux fait externe à SCons .

Prendre des dispositions pour les fichiers source non existants à extraire d'un système de gestion de code source à l' aide spécifié`builder`. Spécifié `entries` peut être un nœud, une chaîne ou d'une liste des deux, et peut représenter soit des fichiers source individuels ou des répertoires dans lesquels les fichiers source peuvent être trouvés.

Pour tous les fichiers source non existants, `scons` recherchera l'arborescence de répertoires et utiliser le premier `SourceCode`constructeur trouve. Spécifié `builder` peut être `None`, dans ce cas , `scons` ne pas utiliser un constructeur pour récupérer des fichiers source pour spécifié `entries`, même si un `SourceCode` constructeur a été spécifié pour un répertoire plus haut de l'arbre.

`scons` , par défaut, récupérer des fichiers de sous-répertoires SCCS ou RCS sans configuration explicite. Cela prend du temps de traitement supplémentaire pour rechercher les fichiers de gestion de code source nécessaire sur le disque. Vous pouvez éviter ces recherches supplémentaires et d'accélérer votre construction un peu en désactivant ces recherches comme suit:

```
env.SourceCode ( '', None)
```

Notez que si spécifié `builder` est celui que vous créez à la main, il doit avoir un environnement de construction associé à utiliser lors de la récupération d' un fichier source.

`scons` fournit un ensemble de fonctions d'usine en conserve qui renvoient les constructeurs appropriés pour différents systèmes de gestion de code source populaire. Des exemples canoniques d'invocation comprennent:

```
env.SourceCode ( ». 'env.BitKeeper (' / usr / local / BKsources'))
env.SourceCode ( 'src', env.CVS ( '/ usr / local / CVSROOT'))
env.SourceCode ( '/', env.RCS ())
env.SourceCode ([ 'f1.c', 'f2.c'], env.SCCS ())
env.SourceCode ( 'no_source.c', None)
```

`SourceSignatures(type)` , `env.SourceSignatures(type)`

Remarque: Bien qu'il ne soit pas encore officiellement désapprouvée, l' utilisation de cette fonction est déconseillée. Voir la `Decider` fonction d'une manière plus souple et facile à configurer la prise de décision de SCons.

La `SourceSignatures` fonction indique `scons` comment décider si un fichier source (un fichier qui ne construit à partir d'autres fichiers) a changé depuis la dernière fois qu'il a été utilisé pour construire un fichier cible particulier. Les valeurs légales sont `MD5`ou `timestamp`.

Si la méthode de l'environnement est utilisé, le type spécifié de la signature de source est utilisé pour décider si les cibles construites avec cet environnement sont à jour ou doivent être recréés. Si la fonction globale est utilisée, le type spécifié de la signature de la source devient la valeur par défaut utilisée pour toutes les décisions quant à savoir si les objectifs sont à jour.

`MD5` des moyens `scons` décide qu'un fichier source a changé si la somme de contrôle MD5 de son contenu a changé depuis la dernière fois qu'il a été utilisé pour reconstruire un fichier cible particulier.

`timestamp` des moyens `scons` décide qu'un fichier source a changé si son horodatage (heure de modification) a changé depuis la dernière fois qu'il a été utilisé pour reconstruire un fichier cible particulier. (Notez que bien que ce soit similaire au comportement de Make, par défaut , il sera également reconstruire si la dépendance est *plus* que la dernière fois qu'il a été utilisé pour reconstruire le fichier cible.)

Il n'y a pas de différences entre les deux comportements pour Python `Value` objets de nœud.

`MD5` signatures prennent plus de temps à calculer, mais sont plus précises que les `timestamp` signatures. La valeur par défaut est `MD5`.

Notez que la valeur par défaut `TargetSignatures` réglage (voir ci - dessous) est d'utiliser ce `SourceSignatures` paramètre pour tous les fichiers cibles qui sont utilisés pour construire d' autres fichiers cibles. Par conséquent, en changeant la valeur de`SourceSignatures` , par défaut, influer sur la décision de tous les fichiers de la construction (ou tous les fichiers construits avec un environnement de construction spécifique lors de la mise à jour`env.SourceSignatures` est utilisée).

`Split(arg)` , `env.Split(arg)`

Renvoie la liste des noms de fichiers ou d'autres objets. Si arg est une chaîne, elle est divisée sur les chaînes de caractères d'espace blanc dans la chaîne, ce qui rend plus facile d'écrire de longues listes de noms de fichiers. Si arg est déjà une liste, la liste sera retournée intacte. Si arg est tout autre type d'objet, il sera retourné comme une liste contenant uniquement l'objet.

Exemple:

```
fichiers = de Split ( "f1.c f2.c f3.c")
fichiers = env.Split ( "F4.c f5.c f6.c")
fichiers = de Split ( "" »
        f7.c
        f8.c
        f9.c
"" ")
```

`env.subst(input, [raw, target, source, conv])`

Effectue une interpolation de variable de construction sur la chaîne spécifiée ou un argument séquence `input`.

Par défaut, avant ou arrière espace blanc sera retiré du résultat. et toutes les séquences d'espaces blancs seront compressés à un seul caractère d'espace. De plus, tout `$(` et des `$)` séquences de caractères seront supprimés de la chaîne retournée, l'option `raw`argument peut être réglé sur `1` si vous voulez préserver l' espace blanc et `$(- $)` séquences. L' `raw` argument peut être réglé sur `2` si vous souhaitez supprimer tous les caractères entre une `$(` et `$)`paires (comme cela se fait pour le calcul de la signature).

Si l'entrée est une séquence (liste ou tuple), les éléments individuels de la séquence sera élargi, et les résultats seront retournés sous forme de liste.

Les options `target` et `source` arguments de mots - clés doivent être réglés sur la liste des noeuds source et cible, respectivement, si vous voulez `$TARGET`, `$TARGETS`, `$SOURCE` et `$SOURCES` être disponible pour l' expansion. Cela est généralement nécessaire si vous appelez à `env.subst` partir d'une fonction Python utilisée comme une action SCons.

Des valeurs de chaîne retournée ou éléments de séquence sont converties en leur représentation en chaîne par défaut. L'option`conv` argument peut spécifier une fonction de conversion qui sera utilisé à la place de la valeur par défaut. Par exemple, si vous voulez des objets Python (y compris SCons nœuds) à retourner sous forme d' objets Python, vous pouvez utiliser le langage Python Λ pour passer dans une fonction sans nom qui renvoie simplement son argument non converti.

Exemple:

```
print env.subst ( "Le compilateur C est: $ CC")

def compiler (cible, la source, env):
    sourceDir = env.subst ( "$ {} SOURCE.srcdir",
                            target = cible,
                            source = source)

source_nodes = env.subst ( 'EXPAND_TO_NODELIST $',
                           conv = lambda x: x)
```

`Tag(node, tags)`

Annote nœuds de fichiers ou de répertoires avec des informations sur la manière dont le `Package` constructeur devrait regrouper ces fichiers ou répertoires. Tous les tags sont facultatifs.

Exemples:

```
# Fait que la bibliothèque sera construite installée avec 0644 fichier
# Mode d'accès
Tag (Library ( 'lib.c'), UNIX_ATTR = "0644")

marques # FILE2.TXT être un fichier de documentation
Tag ( 'file2.txt', DOC)
```

`TargetSignatures(type)` , `env.TargetSignatures(type)`

Remarque: Bien qu'il ne soit pas encore officiellement désapprouvée, l' utilisation de cette fonction est déconseillée. Voir la `Decider` fonction d'une manière plus souple et facile à configurer la prise de décision de SCons.

La `TargetSignatures` fonction indique `scons` comment déterminer si un fichier cible (un fichier qui *est* construit à partir de tout autre fichier) a changé depuis la dernière fois qu'il a été utilisé pour construire un autre fichier cible. Les valeurs légales sont`"build"`; `"content"` (ou son synonyme `"MD5"`); `"timestamp"`; ou`"source"`.

Si la méthode de l'environnement est utilisé, le type spécifié de la signature cible est utilisée uniquement pour cibles construites avec cet environnement. Si la fonction globale est utilisée, le type spécifié de la signature devient la valeur par défaut utilisée pour tous les fichiers cibles qui ne disposent pas d'un type explicite de signature cible spécifiée pour leur environnement.

`"content"` (ou son synonyme `"MD5"`) des moyens `scons` décide qu'un fichier cible a changé si la somme de contrôle MD5 de son contenu a changé depuis la dernière fois qu'il a été utilisé pour reconstruire un autre fichier cible. Ce moyen `scons` ouvriront somme MD5 le contenu des fichiers cibles après leur construction, et peut décider qu'il n'a pas besoin de reconstruire des fichiers cibles « en aval » si un fichier a été reconstruit avec exactement le même contenu que la dernière fois.

`"timestamp"` des moyens `scons` décide qu'un fichier cible a changé si son horodatage (heure de modification) a changé depuis la dernière fois qu'il a été utilisé pour reconstruire un autre fichier cible. (Notez que bien que ce soit similaire au comportement de Make, par défaut , il sera également reconstruire si la dépendance est *plus* que la dernière fois qu'il a été utilisé pour reconstruire le fichier cible.)

`"source"` un moyen `scons` décide qu'un fichier cible a été modifié comme indiqué par le correspondant `SourceSignatures` réglage (`"MD5"` ou `"timestamp"`). Cela signifie que `scons` traitera tous les fichiers d'entrée à une cible de la même manière, que ce soit des fichiers source ou ont été construits à partir d' autres fichiers.

`"build"` des moyens `scons` décide qu'un fichier cible a changé si elle a été reconstruite dans cette invocation ou si son contenu ou l' horodatage ont changé comme spécifié par le correspondant`SourceSignatures` réglage. Cette « propage » l'état d'un fichier reconstruit afin que d' autres fichiers cibles « en aval » seront reconstruits toujours, même si le contenu ou l'horodatage n'ont pas changé.

`"build"` signatures sont les plus rapides parce que `"content"` (ou`"MD5"`) les signatures prennent plus de temps à calculer, mais sont plus précises que les `"timestamp"` signatures, et peuvent empêcher inutiles « aval » reconstruit lorsqu'un fichier cible est reconstruit au même contenu exact que la version précédente. Le `"source"`paramètre fournit le comportement le plus cohérent lorsque d' autres fichiers cibles peuvent être reconstruites à partir des deux sources et les fichiers d'entrée cible. La valeur par défaut est`"source"`.

Parce que le réglage par défaut est `"source"`, à l' aide`SourceSignatures` est généralement préférable `TargetSignatures`, de sorte que la décision mise à jour sera compatible pour tous les fichiers (ou tous les fichiers construits avec un environnement de construction spécifique). L' utilisation de `TargetSignatures` fournit un contrôle spécifique de la façon dont les fichiers cibles construites affectent leurs dépendances « en aval ».

`Tool(string, [toolpath, **kw])` , `env.Tool(string, [toolpath, **kw])`

La `Tool` forme de la fonction retourne un objet appelable qui peut être utilisé pour initialiser un environnement de construction en utilisant le mot - clé des outils de la méthode de l' environnement (). L'objet peut être appelé avec un environnement de construction comme argument,

auquel cas l'objet ajoutera les variables nécessaires à l'environnement de la construction et le nom de l'outil sera ajouté à la $TOOLS variable de la construction.

Arguments de mots clés supplémentaires sont transmis à l'outil`generate`de la méthode ().

Exemples:

```
env = environnement (outils = [Outil ( 'msvc')])

env = environnement ()
Outil t = ( 'msvc')
t (env) # ajoute 'msvc' à la variable TOOLS
u = Tool ( 'opengl', parcours = [ 'outils'])
u (env) # ajoute 'opengl' à la variable TOOLS
```

La `env.Tool` forme de la fonction applique l'objet appelable pour l'outil spécifié `string` à l'environnement à travers lequel la méthode a été appelée.

Arguments de mots clés supplémentaires sont transmis à l'outil`generate`de la méthode ().

```
env.Tool ( 'gcc')
env.Tool ( 'opengl', parcours = [ 'build / outils'])
```

`Value(value, [built_value])` , `env.Value(value, [built_value])`

Renvoie un objet nœud représentant la valeur Python spécifiée. Les nœuds de valeur peuvent être utilisés comme cibles de dépendances. Si le résultat de l' appel `str`(`value`) change entre SCons court, toutes les cibles en fonction `Value`(`value`) seront reconstruits. (Cela est vrai même lorsque vous utilisez des estampilles pour décider si les fichiers sont mis à jour.) Lors de l' utilisation des signatures source d'horodatage, les horodateurs de valeur nœuds sont égaux au temps du système lorsque le nœud est créé.

La valeur retournée objet nœud a une `write`méthode () qui peut être utilisé pour « construire » un noeud de valeur en définissant une nouvelle valeur. L'option `built_value` argument peut être spécifié lorsque le nœud valeur est créée pour indiquer le nœud doit déjà être considéré comme « construit ». Il y a une correspondante `read`méthode () qui retourne la valeur construite du nœud.

Exemples:

```
env = environnement ()

def créer (cible, la source, env):
    # Une fonction qui va écrire un « préfixe = $ SOURCE »
    # Chaîne dans le nom de fichier spécifié comme
    # $ TARGET.
    f = open (str (cible [0]), 'wb')
    f.write ( '= préfixe' + source de [0] .get_contents ())

# Fetch, le préfixe = l'argument le cas échéant, de la commande
# Ligne, et l'utilisation / usr / local par défaut.
préfixe = ARGUMENTS.get ( 'préfixe', '/ usr / local)

# Fixer un constructeur .config () pour l'action de la fonction ci-dessus
# À l'environnement de la construction.
env [ 'CONSTRUCTEURS'] [ 'Config'] = Builder (action = créer)
env.Config (target = 'paquet-config', source = Valeur (préfixe))

def build_value (cible, source, env):
    # Une fonction « construit » une valeur Python en mettant à jour
    # La valeur Python avec le contenu du fichier
    # Spécifié comme la source de l'appel Builder (SOURCE $).
    cible [0] .write (source [0] .get_contents ())

output = env.Value ( 'avant')
entrée = env.Value ( 'après')

# Fixer un constructeur .UpdateValue () pour la fonction ci-dessus
# Action pour l'environnement de la construction.
env [ 'BUILDERS'] [ 'UpdateValue'] = Builder (action = build_value)
env.UpdateValue (target = Valeur (sortie), source = Valeur (entrée))
```

`VariantDir(variant_dir, src_dir, [duplicate])` , `env.VariantDir(variant_dir, src_dir, [duplicate])`

Utilisez la `VariantDir` fonction pour créer une copie de vos sources dans un autre lieu: si un nom sous `variant_dir` ne se trouve pas , mais existe sous `src_dir`, le fichier ou le répertoire est copié `variant_dir`. Les fichiers cibles peuvent être construits dans un autre répertoire que les sources originales en se référant simplement aux sources (et cibles) dans l'arbre variante.

`VariantDir` peut être appelé plusieurs fois avec le même `src_dir` de mettre en place plusieurs construit avec différentes options (`variants`). L' `src_dir` emplacement doit être dans ou sous le répertoire du fichier SConstruct, et `variant_dir` peut ne pas être en dessous `src_dir`.

Le comportement par défaut est `scons` de reproduire physiquement les fichiers source dans l'arborescence variante. Ainsi, une construction réalisée dans l'arbre variante est garanti d'être identique à une construction réalisée dans l'arbre source , même si les fichiers sources intermédiaires sont générés lors de la construction ou préprocesseurs ou d' autres scanners recherche de fichiers inclus par rapport au fichier source ou d'une personne compilateurs ou d' autres outils sont difficiles invoquées codés pour mettre les fichiers dérivés dans le même répertoire que les fichiers source.

Si possible sur la plate - forme, la duplication est réalisée en liant au lieu de copier; voir aussi l' `--duplicate` option de ligne de commande. De plus, seuls les fichiers nécessaires à la construction sont dupliqués; les fichiers et répertoires qui ne sont pas utilisés ne sont pas présents dans `variant_dir`.

Duplication l'arbre source peut être désactivée en définissant l'`duplicate` argument `0` (zéro). Cela entraînera `scons` pour appeler les constructeurs en utilisant les noms de chemin des fichiers source dans `src_dir` et les noms de chemin de fichiers dérivés à l' intérieur `variant_dir`. Ceci est toujours plus efficace que`duplicate=1`, et est généralement sans danger pour la plupart builds (mais voir ci - dessus pour les cas qui peuvent causer des problèmes).

Notez que `VariantDir` fonctionne le plus naturellement avec une filiale fichier SConscript. Cependant, vous appelleriez alors la filiale fichier SConscript pas dans le répertoire source, mais dans la `variant_dir`, quelle que soit la valeur de `duplicate`. Voici comment vous dire `scons` quelle

variante d'un arbre source pour construire:

```
# Run src / SConscript dans deux répertoires variantes
VariantDir ( 'build / variant1', 'src')
SConscript ( 'build / variant1 / SConscript')
VariantDir ( 'build / variant2', 'src')
SConscript ( 'build / variant2 / SConscript')
```

Voir aussi la SConscript fonction décrite ci - dessus, pour une autre façon de spécifier un répertoire variant conjointement avec une filiale appelant fichier SConscript.

Exemples:

```
Les noms d'utilisation # dans le répertoire de construction, pas le répertoire source
VariantDir ( 'build', 'src', en double = 0)
Programme ( 'build / prog', 'build / source.c')

# Ceci construit à la fois la source et docs dans un sous-arbre séparé
VariantDir ( 'build', '', dupliquer = 0)
SConscript (dirs = [ 'build / src', 'build / doc'])

# Comme dans l'exemple précédent, mais seulement utilise SConscript
SConscript (dirs = 'src', variant_dir = 'build / src', dupliquer = 0)
SConscript (dirs = 'doc', variant_dir = 'build / doc', dupliquer = 0)
```

`WhereIs(program, [path, pathext, reject])` , `env.WhereIs(program, [path, pathext, reject])`

Recherches pour l'exécutable spécifié `program`, retournant le nom de chemin complet au programme si elle se trouve, et de retour Aucun sinon. Recherches spécifié `path`, la valeur du PATH de l' environnement d'appel (`env['ENV']['PATH']`), ou PATH externe actuel de l'utilisateur (`os.environ['PATH']`) par défaut. Sur les systèmes Windows, les recherches des programmes exécutables avec l' une des extensions de fichiers répertoriés dans spécifié`pathext`, l'environnement de la PATHEXT (appelant `env['ENV']['PATHEXT']`) ou en cours de l'utilisateur PATHEXT (`os.environ['PATHEXT']`) par défaut. Ne sélectionnez aucun nom de chemin ou le nom dans la spécifié `reject` liste, le cas échéant.

Annexe E. Gestion Tâches courantes

Il y a un ensemble de tâches simples que de nombreuses configurations de construction dépendent qu'ils deviennent plus complexes. La plupart des outils de construction ont des constructions à des fins spéciales pour l' exécution de ces tâches, mais puisque les `SConscript` fichiers sont Python scripts, vous pouvez utiliser plus flexible intégré Python services pour effectuer ces tâches. Cette annexe énumère un certain nombre de ces tâches et comment les mettre en œuvre en Python et SCons .

Exemple E.1. Wildcard englobement pour créer une liste de noms

```
fichiers = Glob (wildcard)
```

Exemple E.2. substitution d'extension Nom du fichier

```
importation os.path
filename = os.path.splitext (nom de fichier) [0] + prolongation
```

Exemple E.3. Un préfixe de l'ajout chemin d'accès à une liste de noms de fichiers

```
importation os.path
les noms de fichiers = [os.path.join (préfixe, x) pour x dans les noms de fichiers]
```

Exemple E.4. Un préfixe de substituant chemin avec un autre

```
si filename.find (old_prefix) == 0:
    filename = filename.replace (old_prefix, new_prefix)
```

Exemple E.5. Filtrage d'une liste de noms de fichiers à exclure / conserver uniquement un ensemble spécifique d'extensions

```
importation os.path
les noms de fichiers = [x pour x dans les noms de fichiers si os.path.splitext (x) [1] dans les extensions]
```

Exemple E.6. La « fonction de backtick »: exécuter une commande shell et capturer la sortie

```
importation subprocess
output = subprocess.check_output (commande)
```

Exemple E.7. Génération du code source: comment le code peut être généré et utilisé par SCons

Les constructeurs de copie ici pourrait être une fonction shell ou python arbitraire qui produit un ou plusieurs fichiers. Cet exemple montre comment créer ces fichiers et les utiliser dans SCons .

```
SConstruct ####
env = environnement ()
env.Append (CPPPATH = "#")

exemple ## en-tête
env.Append (BUILDERS =
   { 'Copie1': Builder (action = 'cat <$ SOURCE> $ TARGET',
                        suffixe = 'h.', src_suffix = 'bar')})
env.Copy1 ( 'test.bar') # produit test.h de test.bar.
```

```
env.Program ( 'app', 'main.cpp') # dépend indirectement de test.bar

## Exemple de fichier source
env.Append (BUILDERS =
  { 'Copy2': Builder (action = 'cat <$ SOURCE> $ TARGET',
                      suffixe = 'cpp', src_suffix = ». bar2' )})
foo = env.Copy2 ( 'foo.bar2') # produit foo.cpp de foo.bar2.
env.Program ( 'App2', [ 'main2.cpp'] + foo) # compile main2.cpp et foo.cpp dans App2.
```

Où main.cpp ressemble à ceci:

```
#include "test.h"
```

produit ceci:

```
% scons -Q
cc -o application main.cpp
cat <foo.bar2> foo.cpp
cc -o App2 main2.cpp foo.cpp
cat <test.bar> test.h
```