Langage Clank : Référence

# Démarrage rapide.

### Présentation

Le langage Clank est un langage permettant de décrire un modèle d’état de serveur dans un langage abstrait, ainsi que des interfaces pour que le serveur et des clients écrits dans des langages différents puissent communiquer par le biais de simples fonctions.

A partir du modèle d’état, et de la description des interfaces, le compilateur Clank génère du code serveur et client (dans les langages demandés) capable de gérer :

* Pour le serveur : la **réception d’une commande** (sous forme de string JSON), et le **renvoi d’une réponse** (sous forme de string JSON). Les mécanismes de réception/envoi de données par TCP ne sont **PAS pris en charge par le langage Clank**, ces mécanismes sont à la charge des développeurs du serveur, pour plus de flexibilité.
* Pour les clients : **l’envoi de commandes** au serveur et **la réception de réponses** par le biais de simples fonctions. L’envoi et la réception de données par TCP est générée (en partie) par le compilateur. **Le client** doit fournir des fonctions permettant :
  + *D’initialiser la connexion TCP avec le serveur*
  + *D’envoyer un string au serveur*
  + *De recevoir un string vers le serveur.*

Pour chaque langage cible, une **classe de génération de code existe**, et cette classe peut être modifiée pour appeler ces fonctions (ou pour changer la manière de sérialiser le code par exemple). Les classes de génération de code sont les seules qu’un utilisateur final doit modifier pour modifier la manière dont le code de sérialisation / envoi par TCP dans un certain langage doit être généré. (Pour plus d’informations consulter la section 5 : Documentation Développeurs).

Le présent document recouvre à la fois une référence exhaustive sur la syntaxe, les fonctionnalités basiques et avancées du langage, certaines parties du code du compilateur, l’utilisation du compilateur, ainsi que des exemples d’utilisation.

### Démarrage rapide

Cette rubrique a pour but de se familiariser avec le langage et le compilateur.

#### Création d’un script basique

Créez un fichier script.clank, et copiez-collez le script suivant :

main

{

# Représente l'état du serveur.

state

{

# Déclaration d'une classe qui doit être connue à la fois par le client

# et le serveur.

public class Car

{

# Une variable accessible par le client et le serveur s'ils disposent

# d'une instance de Car.

public string color;

# Une variable privée.

int m\_speed;

# Un constructeur

public constructor Car New(string c)

{

color = c;

}

# Une fonction publique (exportée chez le client et le serveur)

public void SetSpeed(int speed)

{

m\_speed = speed;

DoSmth();

}

public int GetSpeed()

{

return m\_speed;

}

# Une fonction privée (utilisée seulement par les fonctions publiques

# de cette classe).

void DoSmth() { }

}

# Déclaration d'une classe qui ne sera exportée que sur le serveur.

# Le fonctions des blocs read/write ne doivent pas retourner d'objet de

# ce type.

class PrivateClass

{

public int someVariable;

int somePrivateVariable;

int somePrivateFunction () { }

}

# Déclaration de variables utilisables par le serveur, mais

# inconnues du client.

int myVariable;

Car myCar;

PrivateClass mySecret;

}

# Bloc contenant toutes les fonction permettant au client de récupérer des données depuis le serveur.

access

{

# Les fonctions du block access doivent toutes être publiques.

public int GetMyVariable()

{

return state.myVariable;

}

}

write

{

# Exemples de code produisant des warnings

public void DoBadThings()

{

state.mySecret.somePrivateVariable = 6;

state.mySecret.somePrivateFunction();

}

# Une fonction qui modifie l'état du serveur.

public void DoSecretThing()

{

state.mySecret = PrivateClass.New();

}

# Fonction qui va retourner et prendre en paramètre des objets dont la classe

# est spécifiée dans le block state, et est PUBLIQUE !

# Si la fonction retourne une variable de type PrivateClass, une exception sera

# levée.

public Car GetARedOne(Car c, int acc)

{

Car newCar = Car.New("red");

newCar.SetSpeed(c.GetSpeed() + acc);

return newCar;

}

}

}

#### Compilation du script

Exécutez la ligne de commande suivante :

Clank.Compiler.exe -server=CS:server.cs -clients=CS:client.cs|JAVA:client.java -src=script.clank

Le compilateur doit avoir produit 3 fichiers : server.cs, client.cs, client.java.

Le serveur contient 3 classes : PrivateClass, State et Car. On s’intéresse particulièrement à la classe State :

Server.cs (extrait)

/// <summary>

/// Contient toutes les informations concernant l'état du serveur.

/// </summary>

class State

{

int myVariable;

Car myCar;

PrivateClass mySecret;

/\* .... \*/

public Car GetARedOne(Car c, int acc, int clientId)

{

Car newCar = new Car("red");

newCar.SetSpeed((c.GetSpeed() + acc));

return newCar;

}

/// <summary>

/// Génère le code pour la fonction de traitement des messages.

/// </summary>

public string ProcessRequest(string request, int clientId)

{

Newtonsoft.Json.Linq.JArray o = (Newtonsoft.Json.Linq.JArray)Newtonsoft.Json.JsonConvert.DeserializeObject(request);

int functionId = o.Value<int>(0);

switch (functionId)

{

case 0:

return Newtonsoft.Json.JsonConvert.SerializeObject(new List<object>() { GetMyVariable(clientId) });

case 1:

return Newtonsoft.Json.JsonConvert.SerializeObject(new List<object>() { DoBadThings(clientId) });

case 2:

return Newtonsoft.Json.JsonConvert.SerializeObject(new List<object>() { DoSecretThing(clientId) });

case 3:

Car arg3\_0 = (Car)o[1][0].ToObject(typeof(Car));

int arg3\_1 = o[1].Value<int>(1);

return Newtonsoft.Json.JsonConvert.SerializeObject(new List<object>() { GetARedOne(arg3\_0, arg3\_1, clientId) });

}

return "";

}

}

On note la présence de ProcessRequest, LA fonction importante qui à partir d’un message JSON, renvoie une réponse et modifie l’état du serveur.

Regardons désormais côté client : Il n’y a que 2 classes State et Car (la dernière étant privée, seul le serveur la connaît).

Client.cs (extrait)

/// <summary>

/// Contient toutes les informations concernant l'état du serveur.

/// </summary>

class State

{

/\* .... \*/

public Car GetARedOne(Car c, int acc)

{

// Send

List<object> args = new List<object>() { c, acc };

int funcId = 3;

List<object> obj = new List<object>() { funcId, args };

TCPHelper.Send(Newtonsoft.Json.JsonConvert.SerializeObject(obj));

// Receive

string str = TCPHelper.Receive();

Newtonsoft.Json.Linq.JArray o = (Newtonsoft.Json.Linq.JArray)Newtonsoft.Json.JsonConvert.DeserializeObject(str);

return (Car)o[0].ToObject(typeof(Car));

}

}

Chaque fonction de la classe state est utilisable par le client pour récupérer des informations depuis le serveur. La classe TCPHelper doit être codée pour le client, et contenir de quoi : initialiser la connexion vers le serveur, lire et envoyer des données sous forme de string vers/depuis cette connexion.

#### Intégration du script

TODO

### Erreurs / Warning du compilateur

Lorsque le compilateur rencontre quelque chose d’anormal, il peut réagir de 2 manières :

**Envoyer une Error :** Cela se produit lorsqu’une erreur de syntaxe est dans le script (dans ce cas l’erreur donnée par le compilateur n’est pas toujours très précise, désolé :/), ou qu’une erreur empêchant la compilation de se poursuivre se produit. Par exemple, lorsque le compilateur ne peut pas typer une expression, il arrête la compilation et explique pourquoi.

**Envoyer un Warning :** Cela se produit lorsqu’une erreur dans la « logique » du code est détectée. Cela inclut les erreurs de typage, d’accessibilité de variables / fonctions, etc… Ces warnings n’empêchent pas la génération des fichiers finaux, mais ceux-ci auront alors des risques de ne pas compiler (une erreur de typage va faire que le client C++ ne compile pas, alors que le client python va s’interpréter correctement, mais avec du code bien pourri :D).

# Référence Syntaxe

## Instructions

### Déclaration de block nommé

Un block nommé sert à identifier l’utilité d’une potion de code. Les blocs nommés reconnus par le langage Clank sont : main, state, access, write, macro.

BlockName

{

# Code

}

La totalité du code doit être contenue dans le bloc main. Voir la section « blocs nommés » pour plus d’informations sur les blocs access, write, macro, state.

### Déclaration de classe

[modifiers] class ClassName<GenericParam1, …> { }

Les modificateurs de classe peuvent être : public. Les paramètres génériques ainsi définis peuvent être utilisés dans la classe pour représenter un type passé en paramètre à ce type.

Ex :

class Container<T> {

T value ;

}

Container<int> c = Container<int>.New() ;

int val = c.value ;

### Déclaration de constructeur de classe

public constructor ClassName<GenericParam1, …> New(ArgType arg1, ArgType arg2, …) { }

### Déclaration de fonction

[modifers] ReturnType FunctionName(ArgType arg1, ArgType arg2…) { }

Les modificateurs peuvent être : public, constructor, static.

### Déclaration de variable

VariableType variableName ;

### Affectation de variable

variableName = <expression> ;

Les expressions valides sont détaillées plus tard dans ce document.

## Expressions.

Les expressions sont des morceaux de code evaluables, retournant une valeur typée.

### Litéraux

Les litéraux suivants sont supportés : int (digits 0-9), float (digits 0-9 . digits 0-9), string (“str”), bool (true/false).

Exemples :

int number = 9;

bool Boolean = true;

string str = “hahaha”;

### float floatingPointNumber = 0.65;

### Groupes d’expression

Les groupes d’expression sont constituées d’une ou deux opérandes et d’un opérateur.

[todo : tableau des opérateurs]

### Référence à une variable

Toute référence à une variable est une expression. Le type de l’expression correspond au type de la variable.

Exemple :

int var = 6 ;

### int var2 = var ;

### Appel de fonction

Tout appel de fonction est une expression. Le type de l’expression correspond au type de retour de la fonction appelée.

## Opérations sur les objets

Considérons la classe suivante :

class Dummy<T> {

public T variable ;

public T Function() { }

public static T StaticFunction() { }

public constructor Dummy New() { }

}

Dummy<int> dum ;

### Création d’une instance

dum = Dummy<int>.new()

### Appel de fonction

Un appel de fonction est considéré comme une expression.

dum.Function() ;

### Appel de fonction statique

Un appel de fonction statique est considéré comme une expression.

Dummy<int>.StaticFunction() ;

### Accès à une variable

dum.variable = dum.variable+1 ;

## Structures de contrôle

### If/else/elsif

Syntaxe :

if(boolexpr) { }

elsif(boolexpr) { }

### else { }

### while

Syntaxe :

while(boolexpr) { }

## Types built-in

Les types suivants sont disponibles dans le langage :

String, int, float, bool, void, Type, State.

State : correspond au type contenant les déclarations contenues dans le block state.

Type : Représente un type en langage clank. Les référence au type sont typées avec ce type. Ex : dans ClassName.New(), ClassName est de type Type.

## Préprocesseur

Le langage inclut aussi un préprocesseur, fonctionnant de manière analogue au préprocesseur C.

### Directives #include

Syntaxe : #include filename

Le préprocesseur va utiliser un module de chargement afin de charger le script dont l’URI est « filename ». Par défaut, cela va charger le fichier « filename » à partir du dossier d’exécution du compilateur Clank.

Il est aussi possible de charger des fichiers dans la mémoire (en modifiant le IncludeLoader du Preprocessor du Générateur, dans le code servant à la compilation) :

string generationLog;

GenerationTarget serverTarget = new GenerationTarget("CS", "Serveur.cs");

List<GenerationTarget> clientTargets = new List<GenerationTarget>() { new GenerationTarget("CS", "Client.cs"), new GenerationTarget("Python", "Client.py") };

ProjectGenerator generator = new ProjectGenerator();

// Permet de charger les scripts depuis la mémoire

MemoryIncludeLoader loader = new MemoryIncludeLoader();

// #include myScript va être remplacer par le string s1.

loader.AddFile("myScript", s1);

loader.AddFile("myScript2", s2);

generator.Preprocessor.ScriptIncludeLoader = loader;

List<OutputFile> files = generator.Generate(script, serverTarget, clientTargets, out generationLog);

# Blocs nommés

### Généralités

Plusieurs ensembles de code peuvent être générés à l'aide de Clank.

Ces ensembles de code sont sont :

- header : contient des informations sur les types autorisés, et le modèle de communication (détaillé après).

- macros : Représentent des types et fonctions considérées built-in des langages cibles. Par exemple, les classes telles que les listes, ou autres sont à renseigner dans les macros.

- state : Contient uniquement des classes et des variables. Les variables contenues dans un bloc state peuvent utiliser les classes déclarées dans ce même bloc ainsi que les classes définis dans les blocs "macros".

- access : Contient uniquement des fonctions permettant l'accès à des variables de State. Dans ce bloc, le mot clef state est réservé et est une instance d'une classe State contenant les variables définies dans les blocs "state". La variable client\_id est réservée et contient le numéro du client.

- write : Contient uniquement des fonctions permettant la modification de variables de State. Dans ce bloc, le mot clef state est réservé et se comporte comme dans le bloc access. La variable client\_id est réservée et contient le numéro du client.

Ces différents blocs seront détaillés dans leur propre section.

Les modifications de variables ou l'accès à des variables du mot clef state sont faits avec la syntaxe suivante :

state.machin # Accès à la variable machin

state.machin = valeur; # Modification de la variable machin

En réalité, ces opérations passent par le réseau ou se font directement en mémoire, mais le modèle en fait abstraction.

### Block state

Les variables / fonctions / classes contenues dans le block state sont imbriqués dans une classe State disponible à la fois chez le client et le serveur.

Toutefois : les fonctions contenues dans le block state n’ont pas accès aux variables contenues dans ce même block.

Seules les classes publiques seront exportées chez le client.

Les variables du block state ne seront pas exportées chez le client (seulement sur le serveur).

### Block macros

Les classes contenues dans le block macro sont un peu particulières.

Elles concernent des classes built-in dans les langages cibles, ayant des fonctionnalités similaires, mais un nom différent, et des fonctions ayant des noms différents entre langages, mais fournissant les mêmes fonctionnalités.

Elles permettent de pouvoir utiliser de telles classes (pour pouvoir générer du code dans tous les langages cibles) sans en connaître le code.

Pour cela il faut :

1. Créer la classe normalement.

2. Créer une fonction string Name() contenant une variable de type string du nom de chaque langage cible, et contenant le nom de la classe à utiliser dans le langage cible (pour les déclarations etc...).

3. Créer des fonctions "normalement" : c'est à dire, écrire leur signature correcte. En lieu et place du corps, pour chaque langage cible, créer une variable de type string du nom du langage cible contenant le code à exécuter dans le langage cible.

Pour que cela marche correctement, il faut remplacer les noms des arguments de la fonction par $(argument), et si besoin, le nom de la variable sur laquelle la fonction est appelée par @self. Par défaut, sera appelé variableName.functionName(arg1, arg2…).

# Représente une "liste".

# Ce conteneur est à utiliser en lieu et place des array, et autres,

# car il est compatible avec la plupart des langages et facile à utiliser.

# Cependant, l'implémentation ne sera pas forcément optimale, mais ce n'est

# pas très important.

public class List<T>

{

# Nom du type "List" dans les différents langages cibles.

string Name()

{

string cs = "List";

string java = "ArrayList";

string cpp = "std::vector";

string python = "array";

}

# Obtient un élément de la liste à l'index donné.

T get(int index)

{

string cs = "[$(index)]";

string java = "get($(index))";

string cpp = "at($(index))";

string python = "[$(index)]";

### }

….

}

### Blocks write/access

Les blocks write et access sont très similaires. Ils contiennent des fonctions qui permettent de lire ou d’écrire dans l’état. Ces fonctions sont traduites en code « natif » côté serveur, et en envoi/réception de données côté client. Le générateur de code génère aussi une fonction côté serveur qui se charge de recevoir les messages (typiquement JSON envoyés par TCP), de les décoder, et d’appeler la fonction adéquate avec les arguments encodés en JSON.

Les fonctions de ces blocks peuvent prendre en argument des types publics (ou dérivés de ces types) du block state, ainsi que des types de base. **Les types macro ne sont pas autorisés à être passés en argument ou retournés par ces fonctions** !!

Les fonctions dans ces blocks contiennent deux mot-clefs :

**clientId** : id du client qui a demandé l’appel à la fonction.

**state** : contient l’instance de la classe State utilisée par le serveur.

# Documentation développeurs

Doc pour les gens qui doivent utiliser le code.

### Créer un générateur pour un nouveau langage

TODO

### Modifier le langage / ajouter des fonctionnalités (avancé)

### Compilation