Les tables SQL de Trinity

Galathil

Important : Ce tutoriel a été rédigé par Nobodie pour le site wow-emu.fr. Je le recopie ici à l'identique, Il se peut que certaines informations soient obsolètes à ce jour. Bonne lecture !

Les tables SQL de Trinity

Prélude : Ce sujet n'est pas original, il a été transféré d'un précédent forum d'émulation en date d'Octobre 2013. J'en suis l'auteur, je le transfère donc pour les besoins de ce forum. Bien qu'il soit en partie ré-écrit, il se peut que certains termes soient inadaptés ou ne soient plus d'actualité.
Ce tutoriel n'a pas été directement écrit par moi-même, mais par un des membres de l'équipe Mist-Eria. ce sujet est posté avec l'accord de son auteur. Des modifications peuvent y être apportées dans un but de clarification.

Bonjour tous le monde,

Suite au tutoriel sur l'apprentissage des bases du SQL, on continue la série avec une explication sur les tables TrinityCore. Pour la suite de ce post, nous partirons sur une base 4.3.4.

1) Un serveur WoW, 3 bases de données

Pour fonctionner, un serveur WoW (ici TrinityCore) a besoin de très nombreuses tables qui permettent de stocker de nombreuses informations comme la liste des joueurs, leur inventaire, leurs spells, mais aussi la liste des NPCs, leur niveau et des milliers d'autres informations.

Pour cela, les concepteurs des émulateurs ont divisé les tables dans trois DBs :

- **characters** : qui contient la liste des personnages et toutes les informations les concernant.
- **auth** : qui gère l'authentification des comptes (notamment sur la page de connexion).
- **world** : qui contient toutes les données concernant le monde.

Nous pourrions détailler les trois DBs, mais l'intérêt serait limité pour la table auth et characters qui ne sont pas tellement débuguables, en revanche la base world est celle sur laquelle les devs SQL ont l'habitude de travailler, donc c'est parti pour la détailler et surtout pour la comprendre.

Les tables *_template et associés

Comme son nom l'indique, cette DB décrit le contenu du monde. Que ce soit pour une quête, un objet du jeu, une créature ou bien d'autres choses, c'est la DB où vous travaillez. J'espère que cette introduction bien peu utile vous aura distrait parce que c'est parti pour bosser :niark: .

Lorsque vous lirez les lignes qui suivent, ne perdez pas de vue que tout ce que vous faites dans les tables SQL est relié au Core (c'est à dire au système gérant le serveur). Ainsi si vous créez une ligne pour un nouveau NPC, cela signifiera que le Core lira à un moment ou à un autre votre ligne.

Les premières tables seront celles qui permettent d'interagir rapidement sur le monde (histoire de vous mettre l'eau à la bouche).
Les tables dont nous parlons sont :

- creature_template / creature
- gameobject_template / gameobject
- item_template
- quest_template

En dehors de quest_template et item_template, vous remarquerez que les tables fonctionnent par paire. Template signifie modèle. Donc dans les tables *_template, on trouve une définition de l'objet. Par exemple si on est dans creature_template on trouvera le nom du NPC, son

niveau, sa vie. Alors que dans les tables sans _template on trouvera l'instanciation du template. Pour creature, cela signifiera qu'on trouvera l'endroit où est spawn le NPC (son altitude, la zone, la coordonnée X et Y ainsi que quelques petites informations comme le temps de spawn ou ce genre de chose).

Alors dit comme ça c'est bien beau, mais concrètement, ça ressemble à quoi en DB ?
Et bien les couples de tables sont relié par... un id !

Ainsi dans les tables *_template vous avez un champ nommé entry ou id (selon la version de votre DB). Cet identifiant permet dans la table sans le _template d'identifier le template.
Plus clairement :
Si je crée le NPC 1 dans la table creature_template, je pourrai spawn un NPC correspondant au template 1 en spécifiant l'id 1 dans la table creature.

Les _template définissent un objet / creature / quête / item alors que les autres tables les placent dans le monde.
Alors pourquoi item et quest n'ont pas de table sans _template ? Tout simplement parce qu'une quête ne peut pas être matérialisé à un endroit précis tout comme un item. Les quêtes sont un regroupement d'informations, et les items ont leur existence propre uniquement lorsqu'ils sont possédés par des joueurs.

J'insiste beaucoup sur cette différence *_template / sans _template car c'est un des points fondamentales de la logique TrinityCore (et sûrement Mangos&co).

Maintenant si vous souhaitez comprendre toutes ces tables, il ne vous reste plus qu'à visiter le wiki Trinity (je ne tiens pas vraiment à lister tous les champs pour vos beaux yeux).

Les tables locales_*

Ces tables là ne sont vraiment pas compliquées mais sont très nombreuses, j'explique donc leur fonctionnement une fois et comme ça, vous n'y pensez plus.
Les tables locales servent simplement à définir les traductions des noms de différentes choses jusqu'à dans 9 langues. Ainsi locales_creature, on vous propose de traduire le nom et et le "sous-nom" d'une créature : spécifiez l'entry de la créature et ensuite traduisez. Vous n'êtes évidemment pas obligés de traduire dans toutes les langues, mais juste celle qui vous intéresse ^^.
Pour savoir à quelle langue correspond name_loc1, il faut voir sur le wiki trinity à quoi correspond la langue 1.

C'est tout pour cette sous-partie .

Creature_questrelation, creature_involvedrelation et les équivalents gameobject

J'ai parlé plus haut des games objects sans pour autant les définir. La traduction française de ces termes est : objet du jeu. Les objets du jeu peuvent être un banc, une affiche, une table, et même Hurlevent (si si hurlevent doit se placer sur la map comme une pauvre chaise).

Plus haut, nous avons vu qu'il y avait des templates de quêtes. Vous vous doutez bien qu'on y définit les objectifs, les récompenses et ce genre de chose. En revanche, vous ne savez pas encore comment on sait si un NPC doit donner une quête. Et bien dans quelques minutes vous le saurez grâce à ces quatre tables SQL.
La table creature_questrelation dispose de deux champs : id et quest. ID correspond à l'identifiant du NPC qui donnera la quête et quest à la quête qui sera donnée. Creature_involvedrelation fait exactement la même chose avec les mêmes champs, mais l'objectif est de définir le NPC qui vous permettra de prendre vos récompenses.

Vous l'avez sûrement remarqué : les panneaux dans les grandes capitales donnent des quêtes, et pourtant ce ne sont pas des NPCs : c'est un gameobject (ou GOB) . Pour qu'un GOB puisse donner une quête il existe deux tables : gameobject_questrelation et gameobject_involvedrelation. Je ne décrirais pas le fonctionnement de ces tables, mais vous aurez deviné que c'est exactement comme pour les créatures ^^.

Avec l'ensemble de ces tables, vous pouvez déjà bien vous amuser. L'important c'est avant tout de pratiquer et de prendre des automatismes. C'est notamment ce qui vous permettra peu à peu de comprendre la logique de TrinityCore .

Les gossips

Lorsqu'un NPC ouvre une fenêtre de discussion avec vous, il ouvre un gossip (gossip signifiant discussion ^^). Leur gestion s'effectue à l'aide de deux tables : gossip_menu qui contient la liste des gossip en reliant l'ID du NPC au texte qui sera affiché en haut de la fenêtre (et qui est stocké dans npc_text) et gossip_menu_option qui gère les différentes possibilités offerte par le gossip.
Pour créer un gossip sur un NPC on commence tout d'abord par attribué un flag au NPC dans le creature_template, c'est à dire qu'on va lui

ajouter selon le principe des bitmasks une valeur qui sera géré par le core et qui lui indiquera que le NPC doit ouvrir un gossip lorsqu'on clic sur lui. Le flag correspondant au gossip est le flag 1. Pour l'ajouter à un flag déjà existant, il suffit de faire flag | 1, s'il n'y a aucun flag, on met simplement une valeur de 1. Je vous invite à aller lire le tutoriel sur les bitmasks pour mieux comprendre leur fonctionnement.

Une fois cela fait, et toujours dans le creature_template, on va lier le template à la table gossip_menu. Pour cela on donne à la colonne gossip_menu_id avec l'id qu'on donnera dans gossip_menu : l'objectif est de lier les deux grâce à un identifiant unique.

Vous l'aurez deviné une fois cela fait on crée dans la table gossip_menu une ligne liant le champ gossip_menu_id avec le texte qui sera écrit en haut. Après avoir un peu observé cette table, vous verrez quelque chose de monstrueux ! : il n'y a pas de varchar où l'on peut écrire. Ne vous inquiétez pas, tout est normal. Il faut en faites remplir dans la seconde colonne un id vers la table npc_text (ça devient compliqué). Choisissez donc un id pour le npc_text et inscrivez le dans le gossip_menu.
Pour remplir rien de bien compliqué : le champ text0_0 correspond au texte pour un personnage homme et text0_1 au texte pour un personnage femme.

Si vous redémarrez votre serveur, vous pourrez cliquer sur votre NPC et voir s'afficher le text0_0/1 .

Passons maintenant à la seconde partie, les options. Pour cela rendez-vous dans la table gossip_menu_option. Pour définir une option, on entre le menu_id (l'id que vous avez définit avec le gossip_menu_id dans creature_template ensuite rentrez un id qui identifiera l'option pour le gossip d'id gossip_menu_id. Ainsi si vous avez deux options, vous aurez l'id 1 et 2.

Pour l'option_icon, vous pouvez choisir un chiffre dans la liste (sur le wiki). Certains émulateurs ne fonctionnent pas lorsqu'on met une autre valeur que le un. Le champ option_text correspond au texte de l'option, pour les champs option_id et npc_option_npcflag laissez la valeur à 1 (sauf cas très particulier qui ne mérite pas forcément quelques paragraphes).
Vous pouvez laisser vide les derniers champs.
Si vous avez une table nommée gossip_scripts, vous pouvez mettre dans action_menu_id l'id du script lancé par l'option du gossip (vous verrez plus bas les tables en *_scripts).

Les waypoints et waypoints_data

C'est bien joli d'avoir des NPCs fonctionnels un peu partout, mais c'est pas forcément très beau. D'où l'intérêt des waypoints : faire marcher des NPCs d'un point A à un point B (et même jusqu'à un point Z ou 30000). Le système marche avec deux tables : waypoints et waypoints_data. La seconde table est facultative mais permet d'ajouter certaines informations.

Dans waypoints, vous entrez le GUID du NPC. Je ne vous en ai pas encore parlé, mais les GUID sont des identifiants uniques (Global Unique IDentifier). Si vous avez un template (oui on en revient toujours à ces tables), vous pouvez spawner plusieurs fois le NPC du template. Comment identifier chacun des NPCs spawné ? Eh bien avec un GUID. Vous pouvez avoir 36 000 NPCs avec un template = 1, ils auront tous un GUID différent (un champ auto-incrémenté dans la table creature ou gameobject).

Reprenons donc à nos waypoints : entrez le GUID du NPC que vous souhaitez faire bouger, l'id du point (si vous devez faire aller un NPC de A à B, il faut donner un ID au point : le point 1 sera le premier, le 2 ensuite, etc), puis les coordonnées de chaque point et enfin si vous le souhaitez un commentaire (point de départ, point d'arrivé, etc...).

Si nous devions remplir cette table nous même, nous serions encore plus fou que ce que nous sommes déjà, mais heureusement pour nous, il existe la commande .wp que je ne vous décrirai pas ici.

En revanche, la table waypoints_data elle est à faire à la main (mais heureusement, elle est juste facultative). Elle peut vous servir par exemple si vous souhaitez que votre NPC attende 10 secondes avant d'aller au point suivant). Créez une ligne dans la table waypoints_data donnez lui le GUID du NPC et le point sur lequel vous souhaitez influer. Ensuite rentrez ce que vous souhaitez : delay pour changer la durée avant le nouveau départ, move_flag pour faire marcher le NPC plutôt que de le faire courir, action pour joindre le waypoints à... waypoints_script (les tables *_scripts seront vues plus bas).

*_scripts et la magie s'opéra

Il arrive que l'on veuille faire des choses compliquées et bien souvent on pense tout de suite au C++, cependant bien des choses sont faisables avec le SQL à l'aide de certaines tables comme les tables en *_scripts.

Selon votre version de Trinity (voir de SkyFire) les tables en *_scripts disponibles ne sont pas les mêmes, mais le système est toujours similaire : sur un événement -> faire une action. On pourrait presque parler de micro-programmation événementielle ^^.

Selon les tables liées aux scripts, l'événement se déclenche de manière différente. Dans le cas d'un waypoint, l'événement se déclenche à l'arrivé du NPC sur le point lié au script. Une fois cela fait, il faut donner la durée avant le déclenchement de l'action (bien souvent 0 milliseconde) puis le type d'action (la commande), enfin on donne les paramètres correspondant à la commande : pour téléporter un joueur

millisecondes) puis le type d'action (la commande), enfin on donne les paramètres correspondant à la commande : pour téléporter un joueur on ne donnera pas les mêmes paramètres que pour ouvrir une porte ou tuer un mob.

Toutes ces commandes et leurs paramètres sont trouvables sur le wiki de Trinity.

En faites j'ai écrit les tables *_scripts, mais la table smart_scripts ne rentre pas dedans. Pourquoi ? Parce que c'est une table bien plus complète qui mérite sa propre partie ^^.

Smart_scripts ou mal de tête express

Le titre vous a sûrement intrigué, mais il est très proche de la réalité. La complexité de la table smart_scripts permet de faire beaucoup de choses (pour du SQL).

Cette page vous sera indispensable pour faire du smart_scripts, vous y trouverez tous les tableaux de correspondances

Tout comme pour les tables en _scripts, smart_scripts déclenche une action sur un événement.
Tout d'abord on définit l'élément qu'on va vérifier pour valider un événement. Ce champ est le source_type. Ensuite, donnez l'entryorguid : si vous souhaitez indiquer un entry écrivez le, si c'est un GUID, écrivez le en version négative : pour le guid 3165, il faudra écrire : -3165.
Vient ensuite l'id, comme souvent dans les tables Trinity, l'id est l'id pour l'association entryorguid/source_type. Ainsi si vous avez plusieurs smart_scripts pour un même élément, il y aura plusieurs id : 1, 2, 3... Le champ link permet de lier entre eux des scripts : Une fois le script d'id A exécuté, il lancera celui qui est dans le champ link. Pour que le script B se lance, il faut obligatoirement que l'événement B soit de type : SMART_EVENT_LINK.

Il vous faut ensuite définir le type de l'événement (la liste se trouve toujours sur le lien plus haut) et si vous le souhaitez la probabilité qu'il se produise (en pourcentage). Rentrez ensuite les paramètres demandé sur la page du wiki puis choisissez le type d'action et rentrez les paramètres correspondants.
Pour définir la cible de l'action rentrez les champs de target (type + paramètres). Dans certains cas les événements ont aussi besoin d'une cible au quel cas, tout se complique. Vous ne pouvez rentrer qu'une seule target. La solution est donc de faire deux smart_scripts lié par le link et l'évent SMART_EVENT_LINK.

Pour expliquer votre smart_scripts vous pouvez aussi insérer un commentaire dans le dernier champ.

Vu comme ça, les smart_scripts ne donnent pas vraiment envie, et c'est tout à fait normal. Cependant il existe un outils révolutionnaire pour les rédiger qui vous permettra de gagner énormément de temps et de vous rendre les smart_scripts presque... agréable . Ce logiciel se nomme Event Horizon et donne une interface graphique pour générer vos codes SQL.

Conditions

Encore une table compliquée qui permet de faire des merveilles. Elle permet à l'aide de conditions prédéfinies de faire apparaître ou disparaître des éléments du jeu : que ce soit un NPC, une quête, un gossip, une vente, un smart_scripts.
Bref une table permettant de faire des choses compliquées est forcément très ennuyante.

Etant donnée sa complexité, je ne vais pas vous la détailler, mais sachez que le wiki est très clair sur cette table (à condition que vous descendiez vers le bas de la page).

Des tables en vracs

Il existe des dizaines de tables, et faire le descriptif détaillé de chacune serait très long sans pour autant vous assurer de bien les retenir. Voici donc une liste de quelques tables avec une description :

- **autobroadcast** : cette table sert à définir les messages qui sont automatiquement publiés par le serveur.
- **command** : l'aide des commandes en jeu. Lorsque vous tapez .help commande, le jeu vous affiche la ligne correspondant à la commande.
- **creature_addon / creature_template_addon** : permet d'ajouter des montures ou des smileys sur des guid ou des id de créature.
- **creature_formation** : permet de définir des groupes de NPCs qui bougent ensemble (notamment utilisé pour gérer les armées devant les capitales) ou pour créer des groupes de NPCs hostiles qui s'entraident en cas d'attaque.
- **disables** : permet de désactiver certains éléments du jeu (quête, NPC, etc) sans pour autant les supprimer des tables (pour pouvoir les réactiver plus tard). En général, on évite de supprimer une ligne du world, on préfère utiliser cette table.
- **game_event / game_event_*** : ces tables sont utilisés pour gérer les événements en jeu et ainsi relier de nombreuses informations entre elle avec la possibilité de les désactiver facilement. Un petit tutoriel sera dédié à ces quelques tables.
- **game_graveyard_zone** : liste les zones de téléportation des joueurs à leur mort selon leur zone
- **game_tele** : table contenant tous les points de téléportation (commande : .tele xxx)

- **game_tele** : table contenant tous les points de téléportation (commande : .tele xxx)
- **game_weather** : ici sont stockés les probabilités de chaque météo par zone (orage, soleil, pluie, neige)
- **mail_level_reward** : contient les mails envoyé automatiquement à certains niveaux.
- **npc_vendor** : contient les items vendu par un NPC vendeur (attention le NPC doit avoir le flag 128)
- **player_levelstats** : une table très problématique lorsqu'on ne connait pas la formule mathématiques utilisés par blizzard qui définit les stats minimum des joueurs lorsqu'ils sont nu à chaque niveau.
- **player_xp_for_level** : contient le nombre de points d'expérience nécessaire pour passer au niveau supérieur
- **playercreateinfo**_* : les données qui sont générés lors de la création d'un personnage (par exemple lorsque vous commencer un personnage vous parlez déjà certaines langues -qui sont des spells-, vous avez des vêtements -des items- et bien d'autres choses
- **pool**_* : ces tables ci sont faites pour faire apparaître un des éléments contenu avec une certaine probabilité. Par exemple, un drakonide rare pourra être placé dedans et il spawnera au différents emplacements définies de manière aléatoire.
- *_**loot_template** : ces tables définissent le taux de loot d'items lorsqu'un mob meurt, mais aussi lorsqu'il est fouillé par un voleur ou ce genre de chose.

Il existe évidemment d'autres tables, mais le but de ce cour n'était pas de réécrire le wiki mais bien d'essayer de vous faire comprendre quelles tables étaient les plus importantes.

Si vous pensez que certaines tables évoquées uniquement dans la dernière partie voir pas du tout évoquées méritent une place dans les tables que je décris plus volontiers, vous pouvez le spécifier dans les réponses, et je réfléchirai à leur intégration dans le cour .

N'oubliez pas de pratiquer car c'est la clé du développement SQL sous TrinityCore.
Bon développement.