Le format DBC

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 !

Le format DBC

Bonsoir à tous,

suite à différents sujets et demandes en interne au sujet des DBC, j'écris donc ce tutoriel aujourd'hui afin de comprendre exactement ce qu'il en est. Nous verrons donc d'abord ce qu'est ce format de fichier, à quoi il ressemble et comment le lire, puis nous verrons un petit script python capable de générer un fichier texte représentant les données qu'il contient. Bonne lecture à vous

Ce tutoriel est basé sur mes travaux personnels et sur une page qui m'a beaucoup aidé à l'époque ou je n'y connaissais rien : ~~http://www.pxr.dk/wowdev/wiki/index.php?title=DBC~~ Une portion de ce sujet sera copiée ici et traduite pour la communauté française.

Partie théorique : les fichiers DBC

Qu'est-ce qu'un fichier DBC ?

Un fichier .DBC n'est ni plus ni moins que l'équivalent en SQL d'une table. Ce format contient donc des informations numériques ou textuelles sous forme binaire. Exit donc les modèles 3D, les cinématiques ou que sais-je encore.

Le format DBC est utilisé par Blizzard depuis la première version de WoW. Jusqu'à l'extension Warlords of Draenor, celles-ci étaient accessibles via leurs fichiers propriétaires MoPaQs (les incontournables MPQs), et sont maintenant contenues sous une forme légèrement différentes dans leur nouveau système appelé CASC (dont je ne parlerais pas ici).

Il existe une autre version "évoluée" des DBC que l'on appelle les DB2. Leur format est quasiment identique, à ceci près que toutes les données présentes dans les fichiers DB2 peuvent être overwritées par le serveur de jeu de façon invisible pour les joueurs. Il n'est donc pas nécessaire à Blizzard d'effectuer une mise à jour pour corriger une donnée dans celles-ci, par exemple. Nous ne traiterons pas les fichiers .DB2 ici, sachez simplement que le format est quasiment identique.

Pourquoi ces fichiers se retrouvent-ils sur un serveur privé ?

Si vous avez lu mon tutoriel sur le fonctionnement d'un serveur privé (disponible ici : **http://wow-emu.fr/showthread.php?tid=167**), vous savez que le serveur a besoin d'un certain nombre d'informations en provenance du client. Les DBCs en font partie. Le principal challenge d'un serveur privé est de se confronter à la masse phénoménale de contenu disponible sur officiel. Ainsi, au lieu de créer tout serveur à partir de zéro, les développeurs ont choisi d'extraire un certain nombre d'informations à partir du client afin d'avoir une base "saine" à partir de laquelle ils peuvent extraire un certain nombre d'informations. C'est entre autres grâce à ces bases qu'on possède les informations sur les sorts, les zones, les maîtres de vol, etc etc.

Cependant, si vous avez bien suivi, les DBCs ne sont qu'un fichier de stockage, il n'y a aucune information sur ce qu'elles contiennent ni comment elles s'articulent les unes par rapport aux autres. Ce travail reste donc à la charge des développeurs de serveurs privés qui doivent alors identifier chaque colonne pour en comprendre la signification et l'utilité.

A quoi ressemble un fichier DBC ?

Un fichier DBC est un fichier binaire. Si vous l'ouvrez avec Notepad++ par exemple, vous verrez un tas de caractères qui ne vous diront rien.

Si vous l'ouvrez avec un éditeur hexadécimal, vous y verrez tout un tas de chiffres, qui ne vous diront rien non plus

Pourtant, l'ensemble des chiffres que vous voyez devant votre écran fait pourtant partie de la matière pure des DBCs que vous convoitez. Ces

chiffres là, une fois bien découpés vous permettrons d'obtenir n'importe quelle information contenue dans la DBC et vous pourrez alors en fait ce que vous voulez - en générer un fichier texte ou un ensemble de requêtes SQL par exemple.

Quelle est la structure des fichiers DBC ?

Pour cela, je vais me baser sur le wiki dont j'ai posté le lien au début.
Voici basiquement à quoi ressemble la structure d'une DBC :

```
struct dbc_header
{
uint32_t magic;
uint32_t record_count;
uint32_t field_count;
uint32_t record_size;
uint32_t string_block_size;
};

template<typename record_type>
struct dbc_file
{
dbc_header header;
record_type records[header.record_count];
char string_block[header.string_block_size];
};
```

Cette structure est somme toute assez simple. Analysons-là ensemble

La première partie constitue ce qu'on appelle les DbcHeaders. Ils permettent d'obtenir un certain nombre d'informations sur la DBC et contient 5 variables numériques.

```
struct dbc_header
{
uint32_t magic;
uint32_t record_count;
uint32_t field_count;
uint32_t record_size;
uint32_t string_block_size;
};
```

La première valeur contient, pour un fichier DBC, la valeur 1464091203. C'est ce qui permet d'identifier le fichier et de savoir qu'il s'agit d'un fichier DBC. Cette valeur est de prime abord assez barbare, mais quand on y regarde de plus près elle prend tout son sens.

1464091203 vaut en héxadécimal 0x57444243. Cela nous donne donc 4 octets, 0x57, 0x44, 0x42 et 0x43. Ces quatre valeurs correspondent en fait aux quatre lettres W, D, B et C. Vous pouvez le vérifier en allant sur ce site **http://www.dolcevie.com/js/converter.html** par exemple.

Quand vous voyez un fichier .wdb, c'est en fait un fichier dbc. En réalité, lorsque le fichier ne contient aucune extension, la plupart des logiciels tentent de trouver leur nom et par convention, beaucoup de formats "consomment" les trois premiers octets du fichier pour indiquer l'extension. C'est pour cette raison qu'un dezippeur renomme les fichiers en .wdb par exemple

Les trois valeurs suivantes sont plus simples. Nous allons prendre le cas d'une table SQL.

- Record_count contient le nombre d'enregistrements (autrement dit, le nombre de "lignes" dans une table SQL) que contient le fichier DBC.
- Field_count contient le nombre de champs (autrement dit, le nombre de "colonnes" dans une table SQL) que contient le fichier DBC.
- Record_size contient la taille en octets d'un enregistrement (autrement dit, la taille en mémoire d'une "ligne" dans une table SQL). Il faut savoir que dans la plupart des cas (on va dire 99% pour se laisser une légère marge) une donnée dans les DBCs fait 4 octets (la taille d'un uint32). le Record_size vaut donc très souvent 4*Field_count. Cependant, dans de rares cas cela diffère et c'est à ça que sert cette valeur.

La dernière valeur est un peu différente. Nous venons de dire que chaque donnée dans une DBC est stockée sur 4 octets. Or, nous avons vu aussi que pour écrire WDBC (une chaîne de caractères), il nous fallait également 4 octets. Que faire alors si dans une DBC, Blizzard souhaite enregistrer une chaîne de caractères de plus de quatre octets ? Il y a en réalité une seconde zone de données (un second tableau, plus précisément) qui contiendra absolument toutes les chaînes de caractères. Et au lieu d'enregistrer la chaîne dans le premier tableau, on enregistre une variable qui contient l'offset de la chaîne de caractère dans le second tableau. Cela parait un peu complexe, mais ce n'est ni plus ni moins qu'un pointeur vers une case en mémoire, l'avantage étant que ce pointeur-là peut être stocké dans une variable de 4 octets. Nous verrons lors de la mise en pratique que cela va nous poser quelques soucis si nous voulons identifier automatiquement le type des champs.

La dernière variable contient donc ni plus ni moins que la taille en mémoire du tableau de chaînes de caractères.

Prenons un exemple avec la DBC Achievement, dont voici les premiers octets :

```
...
57444243F70C00000F0000003C000000FC8C04000600000FFFFFFFFFFFFFFFF000000000100000000B000000
...
```

Et une fois découpés en groupes de 4 octets :

```
...
57 44 42 43 F7 0C 00 00 0F 00 00 00 3C 00 00 00 FC 8C 04 00 06 00 00 00 FF FF FF FF FF FF FF FF 00 00 00 00 01 00 00 00 0E
...
```

On a donc :

- uint32_t magic = 57 44 42 43 = "WDBC" => on est bien dans un fichier de type DBC.
- uint32_t record_count = F7 0C 00 00 = 3199 => il y a 3199 enregistrements
- uint32_t field_count = 0F 00 00 00 = 15 => il y a 15 colonnes par enregistrement
- uint32_t record_size = 3C 00 00 00 = 60 => chaque enregistrement fait 60 octets en tout
- uint32_t string_block_size = FC 8C 04 00 = 298236 => la taille totale du tableau de caractères fait 298236 (soit le même nombre de caractères)

Pour faire un lecteur de DBC, il faut donc en priorité lire ces données. Quatre autres "colonnes" suivent, nous ne nous en préoccuperons pas dans le cadre de ce tutoriel. Vient ensuite le bloc de données (de taille Record_count*Record_size) que nous parcourrons pour récupérer tous les champs. Enfin vient le bloc de caractères (de taille String_block_size) qui vient compléter les données.

Comme vous le voyez, un fichier DBC n'est pas si complexe une fois qu'on en a saisi le fonctionnement, et nous allons voir dans la partie suivante comment développer un "simple" lecteur de fichier DBC qui générera un fichier texte contenant les informations des DBCs.

Partie pratique : extraire les données et générer un fichier texte

Rappel de la structure

Comme je vous l'ai expliqué juste au dessus, la structure est divisée en plusieurs sections. Nous allons donc devoir découper notre programme en plusieurs fonctions. La première sera chargée de lire le header du fichier. La seconde aura pour tâche d'extraire toutes les données contenues dans la table, et la troisième récupèrera un tableau de chaînes de caractères identifiées par leur index (afin de pouvoir faire le lien simplement avec l'index).

Je ne détaillerais pas vraiment le code source, je pense qu'il est suffisamment propre pour être lu

Identifier le type de chaque variable

Jusqu'à présent, nous sommes partis du principe que nous savions ce que contenait chaque colonne dans la DBC. Or, je suis allé un peu vite, car si, effectivement, Blizzard sait comment il a structuré chacune de ces DBCs, nous, nous n'en savons rien. Comment savoir que telle colonne est un entier ? Comment savoir que telle autre contient un offset vers une chaîne de caractères ? Comment savoir que ce chiffre là n'est pas en réalité un nombre décimal ? Voilà ce qu'il nous faut trouver.

Je ne vais pas rentrer dans les détails parce que ce n'est pas l'objet du tutoriel et que c'est un poil trop complexe (principalement pour les nombres décimaux), mais je vais tout de même vous donner l'algorithme. Si vous souhaitez voir la version en code, regardez la fonction

"identifyColumns".

Donc, dans les grandes lignes, il faut déjà se rappeler une chose, c'est que nous travaillons dans une base de données. A ce titre, tous les champs d'une colonne donnée sont du même type. L'idée de l'algorithme est donc de parcourir chaque colonne de la DBC et de trouver de quel type il s'agit.

Pour commencer, je pars du principe que chaque colonne est un chiffre entier (le plus simple). De prime abord, si le chiffre est égal à 0, je ne l'utilise pas pour mes tests (car 0 existe dans un entier, dans un float et dans une chaîne de caractères (quand aucune chaîne n'est assignée). Ensuite, je vérifie certains bits du nombre afin de vérifier s'il pourrait potentiellement être un flottant. Si c'est le cas, je le considère directement comme flottant (ça va se vérifier très très vite avec les valeurs, et la possibilité de tomber sur un nombre entier qui se code de la même manière qu'un flottant est très faible).
Enfin, je vérifie si la valeur de l'entier peut correspondre à une chaîne de caractères (autrement dit, si l'index existe dans mon tableau de chaînes de caractères) après avoir évité mes valeurs égales à 1 (car la probabilité de tomber sur une erreur est grande). Si, à la fin de toute ma boucle, je n'ai pas su identifier de contre-exemple au fait que ce soit possiblement une chaîne de caractères, alors je la considère comme une chaîne de caractères.

Cet algorithme est loin d'être parfait et cause parfois des erreurs d'appréciation (principalement des entiers identifiés comme des flottants par exemple). Cependant, ce sont très souvent des champs qui ne vous serviront pas pour comprendre et trouver les informations qui vous sont nécessaires.

Je vous fournis le code tel quel et ne le commenterais pas, c'est somme toute un programme assez basique. Je vous donne juste les paramètres :

```
./dbcreader.py [-hfv -s <semicolon> -o <outputfile>] -i <inputfile>
```

- **-h** : vous permet d'afficher l'aide (la même chose que ce vous voyez ci-dessus).
- **-f** : vous permet d'afficher les informations contenues dans le header ainsi que le format de la dbc (le type de chaque colonne) identifié par le programme.
- **-v** : vous permet d'activer le mode verbose et d'afficher le résultat dans la console en plus du fichier de sortie (utile pour rediriger la sortie sous linux, par exemple).
- **-s <semicolon>** : vous permet de définir le caractère de séparation dans le fichier de sortie. Par défaut ; (point virgule).
- **-o <outputfile>** : vous permet de définir le nom du fichier de sortie. Par défaut, nom_fichier_entree.csv.
- **-i <inputfile>** : seul paramètre obligatoire, vous permet de définir le fichier d'entrée du programme. Doit être un fichier de type dbc.

Et voici le fichier en lui-même. Si des soucis d'antislashs viennent poser problème, cela vient du forum et il vous faudra alors les enlever manuellement.

```
#!/usr/bin/python
import struct
import sys, getopt

def main(filename, argv):
inputfilename = ''
outputfilename = ''
global semicolon
semicolon = ';'
global verbose
verbose = False
showFormat = False
helper = filename+" [-hfv -s <semicolon> -o <outputfile>] -i <inputfile>"+"\n"

try:
opts, args = getopt.getopt(argv,"hfvs:o:i:",["semicolon=", "ifile=", "ofile="])
except getopt.GetoptError:
sys.stdout.write(helper)
sys.exit(1)
for opt, arg in opts:
if opt == '-h':
sys.stdout.write(helper)
```

```
sys.exit()
elif opt == "-v":
verbose = True
elif opt in ("-s", "--semicolon"):
semicolon = arg
elif opt in ("-i", "--ifile"):
inputfilename = arg
elif opt in ("-o", "--ofile"):
outputfilename = arg
elif opt == "-f":
showFormat = True

if not inputfilename:
sys.stdout.write(helper)
sys.exit(1)

if not outputfilename:
outputfilename = inputfilename[:-4]+'.csv'

inputfile = open(inputfilename, "rb")
global outputfile
outputfile = open(outputfilename, "w")

try:
WDBC = inputfile.read(4)
if WDBC != "WDBC":
sys.stderr.write("Not supported format\n")
sys.exit(2)

headers = readHeaders(inputfile)
datas = readDatas(inputfile, headers)
skipBytes(inputfile, 1)
strings = readStrings(inputfile, headers)
columns = identifyColumns(headers, datas, strings)
if showFormat:
sys.stdout.write("Headers : "+str(headers)+"\n")
sys.stdout.write("Format : "+''.join(columns)+"\n")
sys.exit(0)
writeData(headers, datas, strings, columns)
finally:
inputfile.close()
outputfile.close()

def write(string):
outputfile.write(string+"\n")
if verbose:
sys.stdout.write(string+"\n")


def writeData(headers, datas, strings, columns):

for rec in xrange(headers["nbrec"]):
recstr = ""
for field in xrange(headers["nbfields"]):
col = columns[field]
if col == 'i':
recstr += str(struct.unpack('i', datas[rec][field])[0]) + semicolon
elif col == 'f':
recstr += str(struct.unpack('f', datas[rec][field])[0]) + semicolon
elif col == 's':
```

```
s = struct.unpack('i', datas[rec][field])[0]
if not s:
recstr += "NULL" + semicolon
else:
recstr = recstr + '"'+str(strings[s])+'"' + semicolon
write(recstr)


def identifyColumns(headers, datas,strings):
columns = ['i']*headers["nbfields"]
for field in xrange(headers["nbfields"]):
possiblyAString = True
for rec in xrange(headers["nbrec"]):
value = struct.unpack('i', datas[rec][field])[0]
if not value:
continue
if not (not(value & 0xFF800000) or (value & 0xFF800000 == 0xFF800000)):
columns[field] = 'f'
possiblyAString = False
break
if value != 1 and value not in strings:
possiblyAString = False

if possiblyAString:
columns[field] = 's'
return columns


def readHeaders(inputfile):
headers = {}
headers["nbrec"] = struct.unpack('i', inputfile.read(4))[0]
headers["nbfields"] = struct.unpack('i', inputfile.read(4))[0]
headers["recsize"] = struct.unpack('i', inputfile.read(4))[0]
headers["ssize"] = struct.unpack('i', inputfile.read(4))[0]
return headers

def readDatas(inputfile, headers):
data = []
for irec in xrange(headers["nbrec"]):
rec = []
for jfield in xrange(headers["nbfields"]):
bytes = inputfile.read(4)
rec.append(bytes)
data.append(rec)
return data

def skipBytes(inputfile, count):
inputfile.read(count)

def readStrings(inputfile, headers):
strings = {}
counter = 1
while counter < headers["ssize"]:
s = extractString(inputfile)
strings[counter] = s
counter += len(s)+1
return strings

def extractString(inputfile):
```

```
s = ""
car = inputfile.read(1)
while ord(car) != 0:
s = s+car
car = inputfile.read(1)
return s


if __name__ == "__main__":
main(sys.argv[0], sys.argv[1:])
```

J'espère que ce tutoriel vous aura plu et que vous serez un peu plus curieux quant aux fichiers que vous utilisez (et pourquoi pas voir un tutoriel sur les fichiers utilisés dans les maps et les vmaps par exemple ?) et que vous pourrez à présent développer vos propres outils de lecture ou d'écriture des DBCs au lieu d'utiliser des logiciels existants, fermés et faisant la plupart du temps la moitié du travail que vous souhaitez

~~Merci à vous de m'avoir lu, comme toujours, je suis ouvert aux critiques constructives ~~