Data Toolkit Quick Tutorial

1. A brief introduction

The data toolkit focuses on solving the problems of data form and accessibility for data-driven development. It includes the following functions to provide a simple one-stop service for game development:

- Database CRUD operation utils based on SQLite.
- Common local file I/O interface in a cross-platform way.
- Analysis and generator of standard CSV format.
- Provide Unity JSON serialization extension for dictionary.

The data toolkit will focus on engine changes and provide long-term support to ensure the project stability of users. And more functions will be also updated for free in the subsequent versions.

If you want to keep up with updates, please follow our GitHub official website. You can also find the latest version of this document on our online documents page.

2. Prerequisite of the Data Toolkit package

By downloading and then importing the Data Toolkit package from the unity package manager, you have completed its installation just like importing other packages.

This package is applicable to all versions after Unity 2020, and is also applicable to Android/iOS/Windows and Mac OS platforms.

Please note that if you want to use the **SQLite Toolkit** in the package, please check the additional settings to ensure that it behaves correctly at runtime:

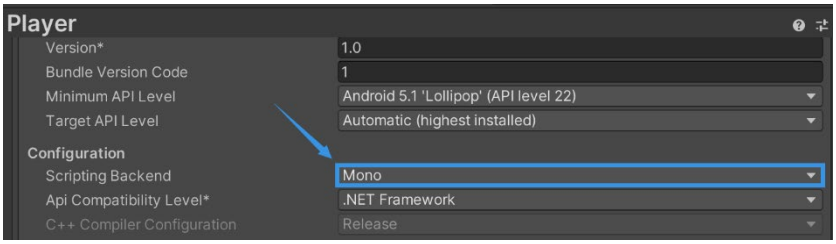

Figure 1. Additional Setting

That setting item is located in Project Settings - Player - Other Settings - Scripting - Backend, Choose **Mono** for SQLite Toolkit.

If you don't need to use the functionality in the SQLite toolkit, you don't need to change any options, and you don't have to worry about throwing exceptions or crash -- it's safe all the time.

3. SQLite Toolkit tutorial

SQLite Toolkit provides a type for abstraction of SQL data connection: SQLiteConnection. To write your own SQL logic code, just use the following two namespaces:

```
using Arcspark.DataToolkit;
using Mono.Data.Sqlite;
```

3.1 Create SQLConnection

Create an SQLConnection object using SQL connection string parameters to start operating the corresponding SQLite database.

A connection string is used to specify how to connect to the database.
On Windows, iOS and Mac OS platforms, you can access database files In StreamingAssets path and Persistent data path. But on Android platform, Persistent data path is the only data source for SQLiteConnection.

For a simple example, if you want to connect a database file, and its path is Assets/StreamingAssets/test.db, you should use the following code to build a correct connection:

```
string connectionStr = Application.streamingAssetsPath+ "/test.db";
SQLiteConnection db = new SQLiteConnection(connectionStr);
```

Similarly, if you want to access a database file “test.db” that located under persistent data path (in Windows systems, it is usually located under “C:\Users\user name\AppData\LocalLow\your company name\your game name” folder), you should use the following code to build a correct connection:

```
string connectionStr = Application.persistentDataPath + "/test.db";
SQLiteConnection db = new SQLiteConnection(connectionStr);
```

Don't forget to use **Close** function to close the database link when completing the transaction operation on the database:

```
db.Close();
```

3.2 Insert Operation

Use the **InsertValues** method to complete the field insertion of a table:

```
db.InsertValues("TableName",
    new string[] { "10000", "'String Data'", "0.1" });
```

The sample code deletes all data rows with amount less than 0. Note that the number of value lists should be the same as the number of columns in the data table(the excess part will be ignored).

The syntax for values here always needs to comply with the SQL syntax specification. For example, when we want to insert a string, we need to wrap it in a pair of single quotes.

3.3 Delete Operation

Use the **DeleteValues** method to delete data in a table:

```
db.DeleteValues("TableName", "Amount < 0");
```

A conditional SQL clause as the second parameter to indicate what data should be deleted. The sample code deleta all data rows with amount less than 0.

You can also use "**DeleteValuesOR**" or "**DeleteValuesAND**" to simplify the operation process under multiple conditions. For more details, please refer to API reference.

3.4 Modify Operation

Use the **UpdateValues** method to update data in a table:

```
db.UpdateValues("TableName",
    new string[] { "Amount", "Name", "Rate" },
    new string[] { "1", "'Value2'", "1.2" },
    "Amount < 0");
```

A conditional SQL clause as the second parameter to indicate what data should be updated.The sample code update all data rows with amount less than 0.

The “Amount” field of these rows will become 1, “KeyName2” field will become "Value2", and “KeyName3” field will become "Value3".

You can also use " UpdateValuesOR" or " UpdateValuesAND" to simplify the operation process under multiple conditions. For more details, please refer to API reference.

All transaction operations will return a Mono.Data.Sqlite.SqliteDataReader. It’s a Mono version for Microsoft.Data.Sqlite.SqliteDataReader.

3.4 Access Data

You can traverse a SqliteDataReader in the following simple way.

```
SqliteDataReader r = db.SelectTable("TableName");
do
{
    while (reader.Read())
    {
        //...
    }
}
while (reader.NextResult());
```

The behavior of stepping the SqliteDataReader each time in the loop will get the next row of data in the data collection.

We also provide some methods in the data toolkit to quickly obtain the data of specific fields in the current row:

```
SqliteDataReader r = db.SelectTable("TableName");
do
{
    while (reader.Read())
    {
        int? id = reader.GetInt32("Amount");
        string name = reader.GetString("Name");
        double? attackDamage = reader.GetDouble("Rate");
    }
}
while (reader.NextResult());
```

The GetXXX methods cover all the basic value types of C#. For more details, please refer to API reference.

3.5 Other Features

SQLiteConnection.CreateTable: Create a table in correct database connection.
SQLiteConnection.DropTable: Drop a exist table in correct database connection.
SQLiteConnection.SelectTable: Select all data in a specified table.
SQLiteConnection.ExecuteQuery: Execute a SQL query statement directly.
SQLUtil: A SQL related auxiliary tool, you can use SQLUtil.CreateSQLiteDatabase method to create a database file in local.
For more details, please refer to API reference.

4. File Util tutorial

File Util provides some unified cross platform file access interfaces, both synchronous and asynchronous are provided. The class FileUtil is a singleton class and it is used without pre-constructing an instance.

To write your own File I/O logic code, just use the following one namespace:

```
using Arcspark.DataToolkit;
```

4.1 Get Local File

GetLocalFile method help you get local files synchronously as the name suggests. It can access file under StreamingAssets path and Persistent data path. You can use it with just one statement:

```
FileUtil.Instance.GetLocalFile(
    FileUtil.LocalSourceType.STREAMING_ASSETS,
    "testfile.txt",
    out byte[] bytes,
    out string text
);
```

The above code gets a file named testfile.txt under the Streaming Assets Path. The bytes and text output parameters contains the content of the obtained file.

And the only parameters you need to pay attention to are “type” and “filePath”.
The value of parameter “type” is **FileUtil.LocalSourceType.STREAMING_ASSETS**(for File in Streaming Assets Path) or **FileUtil.LocalSourceType.PERSISTENT_DATA_PATH**(for File in Persistent Data Path).
Then the “filePath” is a relative path about StreamingAssets path or Persistent data path.

The local load function does not provide an asynchronous version (because it is unnecessary). But it provides two variants: **GetLocalFileText** and **GetLocalFileBytes**, which let you focus only on text or file bytes.

4.2 Get Remote File & AssetBundle

GeRemoteFile method helps you get the files in the remote server through URL.

```
FileUtil.Instance.GetRemoteFile C:\Users\Kiro\Desktop\TC.cs (
    FileUtil.RemoteSourceType.FILE,
    "http://192.168.100.1:8181/testfile.txt",
    out byte[] bytes,
    out string text
);
```

The above code gets a file named testfile.txt from the 192.168.100.1. The bytes and text output parameters contains the content of the obtained file.

The “type” parameter of GetRemoteFile method can be **FileUtil.RemoteSourceType.File**(for remote file) or **FileUtil.RemoteSourceType.ASSET_BUNDLE**(for remote AssetBundle).

For remote file reading of AssetBundle type, you can also set the specific version and CRC values of AssetBundle in parameters.

It also provides an asynchronous version: **GetRemoteFileAsync**. You can provide a callback function as a parameter, which will be called automatically after the download is completed.

The method GetRemoteFileAsync returns a DownloadRequest type to view the progress of your download request. You can also interrupt the download in the returned instance:

```
// Get progress of dowdload request
request.Progress;
// Abort this download request
request.Abort();
```

For more details, please refer to API reference.

4.3 Set Local File

FileUtil allows you to write the contents of the file under the Persistent data path in local. Just use the **SetFile** method like this:

```
FileUtil.Instance.SetFile("Data/testfile.txt", "Content text");
```

The above function will write the "Content text" string into the testfile.txt file under the %Persistent data path%/Data. Folders and files that do not exist will be created after SetLocalFile.

5. CSV Util tutorial

CSV Util provides a type for abstraction of CSV table data: CSV Table. To write your own CSV logic code, just use the following one namespaces:

```
using Arcspark.DataToolkit;
```

A typical example of CSV is as follows:

```
[string]Name,[int]Age,[double]Height
Bill,16,1.6
Joe,17,1.7
Bruce,18,1.8
```

The first row generally represents the table header declaration, identifying the data name and data type of each column. The remaining lines are data.

Or you can use a pure CSV without header declaration row:

```
Name,Age,Height
Bill,16,1.6
Joe,17,1.7
Bruce,18,1.8
```

Both format of CSV can be parsed and serialized correctly. CSVTable will distinguish, parse and serialize them automatically.

After explaining the difference between the two forms in the first line, let's quickly preview the format rules of CSV:

CSV elements separated by commas. If there are commas, double quotes, spaces or line breaks in one data element, the data will be wrapped in a pair of double quotes. A single double quotes mark need to be replaced by two consecutive double quotes marks.

The above is the simple format rule of a CSV format. CSVTable can handle this kind of data, including parsing, serialization and CRUD operations on CSV table content.

5.1 Parse & Serialize

Parsing CSV with CSVTable is very easy:

```
CSVTable ct = new CSVTable("TableName", csvStr);
```

In this constructor, you can also customize the delimiter and escaper you use.
The above code is equivalent as follows:

```
CSVTable ct = new CSVTable("TableName", csvStr, ',', '"');
```

(Here, the separator is a comma and the escaper is a double quotes mark.)

It is also very simple to convert an instance of CSVTable into a string in CSV format:

```
string csvStr = ct.Serialize();
```

This will cause the CSVTable object “ct” to be re-serialized into a CSV string.

5.2 Access Element in CSVTable

CSVTable provides a variety of ways to access elements in it.

Indexed access is the most direct access method, which treats CSVTable as second-order data to access its content.

```
// Access the row at 0 index.
CSVRow r = ct[0]
// Access a element at (0,0).
CSVElem e = ct[0][0]
```

A CSVRow represents a row in the table (and a CSVColumn represents a column in the table). You can use **CSVRow.GetElemByName ("Key Name")** to further obtain the specific element value in this data row.

You can alse use **GetRow** / **GetColumnByIdx** / **GetColumnByName** / **GetElem** method group to access your csv table data. For more details, please refer to API reference.

Basically, in the end, you will get a **CSVElem** object representing a specific CSV element. You can convert a CSVElem to any C# standard value type (or its primitive string type) by implicit conversion:

```
// Get the first column whose header is "Age".
CSVColumn cc = ct.GetColumnByName("Age");
// Get the element with index 0 in the "Age" column.
CSVElem age = cc[0];
// Get the value in age by implicit conversion.
int a0 = age;
// Or use the ExplainAsInt method directly to get its value.
int a1 = age.ExplainAsInt();
```

5.3 Change CSVTable

You can use the **SetElemValue** method to modify the value of an element in the CSV table:

```
ct.SetElemValue(0, 0, "20");
// The same as above.
ct[0][0] = "20";
```

And the **AppendRow** / **RemoveRow** / **AppendColumn** / **RemoveColumnByIdx** / **RemoveColumnByName** methods are also able to change the content of the CSV table. For more details, please refer to API reference.

6. Archive Manager

SerializeDictionary is a extension for serialize. It allows user seriallize Dictionary just like other serializable field types as such as List. Use SerializeDictionary instead for Dictionary container to make Dictory serializable:

```
// A serializable class
[Serializable]
public class MyArchiveData
{
    public int A = 0;
    public List<int> B = new List<int>();
    // C & D are serializable dictionaries.
    public SerializeableDictionary<string, int> C = new
SerializeableDictionary<string, int> ();
    public SerializeableDictionary<string, double> D = new
SerializeableDictionary<string, double>();
}
```

And the Archive Manager class used this to complete a data storage function, which can serialize your user data as a JSON string and store it in local files.

And using the generic class ArchiveManager to complete the archiving function requires only two steps:

1. Define your user data Class
   The user data type has some requirements:
   1. it must have the [Serializable] attribute.
   2. it must have a default constructor.
   3. it's field must be public or have a [SerializeField] attribute.
   4. If there are Dictionary fields, replace their types with SerializeableDictionary.
2. Define your Archive Manager class:

```
// A serializable class.
[Serializable]
public class MyArchiveData
{
    // ...
}

// A archive manager for the class MyArchiveData
public class MyArchiveManager : ArchiveManager<MyArchiveData>{}
```

Now, you can use the Save and Load methods to read and write the archive:

```
// Create a archive manager instance.
MyArchiveManager m = new MyArchiveManager();
// Try load the save file.
m.Load();
```

```
// Change user record value.
m.Archive.i = 1;
// Save it in local file.
m.Save();
```

7. More APIs

The above only lists some common APIs and points for attention. See the help file “Data Toolkit API Reference.chm” for more API information.