What is KBEngine?

KBEngine is an open source game server engine. The client can communicate with the server through a simple protocol. KBEngine plugins can be quickly combined with (Unity3D, OGRE, Cocos2d-x, HTML5, etc.) technology to form a complete game client. The server-side low-level framework is written in C++, and the game logic layer uses Python (supports hot-fixing). Developers do not need to repeatedly implement the underlying technologies common to some game servers, and instead can focus their efforts on the game development level, to quickly build various multiplayer games.

(Because it is often asked what the upper limit of the load is that KBEngine can handle, the underlying architecture has been designed as a multi-process distributed dynamic load balancing solution. Theoretically, by continuously expanding the hardware, the upper limit of the load can also be continuously increased. The upper limit of the capacity of a single machine depends on the complexity of the game logic itself.)

Homepage **http://www.kbengine.org** Releases sources : **https://github.com/kbengine/kbengine/releases/latest** binarys : **https://sourceforge.net/projects/kbengine/files/** Demo sources unity3d : **https://github.com/kbengine/kbengine_unity3d_demo/releases/latest** unity3d : **https://github.com/kbengine/kbengine_unity3d_warring/releases/latest** ogre : **https://github.com/kbengine/kbengine_ogre_demo/releases/latest** html5 : **https://github.com/kbengine/kbengine_html5_demo/releases/latest** Docs docs : **http://www.kbengine.org/docs/** API : **https://github.com/kbengine/kbengine/tree/master/docs** Support Email : **kbesrv@gmail.com** Maillist : **https://groups.google.com/d/forum/kbengine_maillist**

Start

You can look at the **installation guide** first. After learning about the **common tools**, you can go to **GitHub** to join our open source team.
I believe that after you have enough knowledge of KBEngine, you will like it.

Want to modify and improve the documentation?

If you think there is something in the documentation that needs to be changed, you can **fork kbengine_docs** and submit a pull request.
Again, if you find anything in the documentation that you don't understand, please submit an **issue**, and I will adjust the document to help everyone have a better understanding.

Keywords Used

EntityCall:

This is the conventional means of remote interaction between entities in the script layer. (other references: **allClients**, **otherClients**, **clientEntity**).

The EntityCall object is very simple to implement in the underlying C++, it contains only the ID of the entity, the address of the destination, the entity type, and type of EntityCall. When a remote call is requested by the user, the engine first finds a description of the entity definition through the entity type, and checks the data input by the user against the description of the entity definition. If the check is legal, then the data is packaged and sent to the destination according to the protocol.

Note: An EntityCall can only be used to call methods declared in its corresponding def file. It cannot call any of KBEngine's basic Entity class functions or access entity attributes.

An Entity can contain up to three parts:
Client: When an entity includes a client part (usually a player), the entity's client (EntityCall) property can be accessed on the server side.
Base: When an entity includes a baseapp part, the base (EntityCall) attribute of the entity can be accessed in a non-current baseapp.
Cell: When an entity includes a cellapp part, the cell (EntityCall) attribute of the entity can be accessed in a non-current cellapp.

Example:

Client remote method defined in Avatar.def:

```
<ClientMethods>
        <hello>
        </hello>
</ClientMethods>
```

client\Avatar.py

```
class Avatar:
        def hello(self):
                print("hello")
```

Enter in the Debug page input box of the GUIConsole tool (check the process to be debugged first in the list on the left):
First find the ID of the player entity (Avatar) in the log of the server's Baseapp, and then get the player entity (Avatar) or EntityCall through the entity ID:

>>> KBEngine.entities[Player ID].client.hello()

At this point, the client log file will output "hello" and a remote call process will be completed.

KBE_ROOT:

This is a KBEngine environment variable that describes the root directory where KBEngine is located.

KBE_RES_PATH:

This is a KBEngine environment variable that describes the resource directory that KBEngine engine can read.

KBE_HYBRID_PATH:

This is a KBEngine environment variable that describes the directory where the KBEngine engine executable file is located.

entities.xml:

All valid entity types on the server must be registered here. When the engine is initialized, the description of the entity is loaded according to the order.

kbengine_defaults.xml:

Server-side default configuration, where users can modify all component configurations such as **cellapp**, **baseapp**, and **loginapp**.

Note: You may often need to upgrade the engine. Modifying directly may cause conflicts during the upgrade, and it is not suitable for multiple projects in the same KBEngine environment.

It is recommended that you modify the overload in **kbengine.xml**. You only need to rewrite the parts you want to modify according to the format in xml.

kbengine.xml:

Server configuration file, where users can modify all component configurations such as **cellapp**, **baseapp**, and **loginapp**.
For details, please refer to **kbengine_defaults.xml**

entity

Entity is defined as the most basic object of the server, similar to Python's base object.
When and how should you define an entity? See
http://www.kbengine.org/docs/programming/entitydef.html

View

Each client entity connected to the server will have a View. It allows the client entity to communicate events in its View to its own client.
View is related to space, and each View can be set to an independent size range.

Note: The space described here is an abstract concept and does not necessarily need to be bound to the concept of physical space (except for MMORPGs). For the core gameplay of a card game, players in a room can also be considered to be in a logical space.

Events include: entity movement, property change of client broadcast type, destruction on death, and so on.

Witness

Eyewitness.
Only Witnesses bound to **cell** entity Views take effect. In other words, Witness is a **cell** proxy of the client. The cellapp continuously synchronizes the information in the View to the client through the Witness.
When an NPC on the server is seen by a witness, it call the onWitness callback of the entity. The server can rely on this feature to reduce CPU consumption. When an entity is not witnessed, the server can stop any of its behavior.

Space

A space KBEngine allocates on the cellapp, which is isolated from other spaces. Views, traps, entity collisions, etc. only interact with each other in the current space. The space is defined by the user. It can be a scene, copy, room...

cell

There are two different meanings of cell in this documentation. Usually, when referring to the **Entity.cell** attribute, you are actually describing the entity's **CellEntityCall**.

If a cell is described as part of a **space**, it refers to **cellapp**'s load balancing technique. A space in the cellapp may be divided into n parts, each called a cell, and each cell is maintained by a different process.

base

Usually refers to the Base entity on the **baseapp** or a **BaseEntityCall** that points to the Base entity.
For example: **Entity.base**

client

Usually refers to the client or an **EntityCall** that points to the client entity.
For example: **Entity.client**

cellapp

The Cellapp process in mainly responsible for position-related game logic, View, AI, scene rooms, and so on.
See also: **cellapp**

baseapp

The Baseapp process is mainly responsible for communication with the client, position-independent game logic (guild manager, chat system, game lobby, leaderboard, etc.), archiving, backup, and so on.
See also: **baseapp**

real

Refers to an entity in a cell that is actually present in the cell at that time. (As opposed to a **ghost** entity broadcast there by another cell)

ghost

This kind of entity is a projected copy generated by cellapp's dynamic load balancing mechanism which divides a space into N shares and splits the cells between different processes.
Space is divided into multiple regions. To make the client unable to perceive the existence of the boundaries between them, we synchronize a certain range of entities in each cell's boundary to an adjacent cell's boundary. The entity has a part of its data synchronized over (CELL_PUBLIC, cell broadcast types of attributes) to a ghost entity. In this way the entity can interact seamlessly with the other side of the boundary and both cells simultaneously.

Non-ghost entities are called **real** entities.

vector3

Describe and manage 3D space vectors.
There are three properties of x, y, and z that represent different axial directions.

Example in script: import Math v = Math.Vector3()

Basic data types

[Name]	[Bytes]
UINT8	1
UINT16	2
UINT32	4
UINT64	8
INT8	1
INT16	2
INT32	4
INT64	8
FLOAT	4
DOUBLE	8
VECTOR2	12
VECTOR3	16
VECTOR4	20
STRING	N
UNICODE	N
PYTHON	N
PY_DICT	N
PY_TUPLE	N
PY_LIST	N
ENTITYCALL	N
BLOB	N

KBEngine

KBEngine module

This **KBEngine** module provides a part of the logic script layer access to **entity**, as well as current space data and so on.

Class

Entity

Member function

def **login**(username, password):
def **createAccount**(username, password):
def **reloginBaseapp**():
def **player**( ):
def **resetPassword**(username):
def **bindAccountEmail**(emailaddress):
def **newPassword**(oldpassword, newpassword):
def **findEntity**(entityID):
def **getSpaceData**(key):

Callback

Attributes

component	Read only **string**
entities	**Entities**
entity_uuid	uint64
entity_id	int32
spaceID	int32

Member functions documentation

def login(*username, password*):

Function description:
Login account to KBEngine server.
Note: If the plug-in and the UI layer use event interaction mode, do not call directly from the UI layer. Please trigger a "login" event to the plug-in. The event is accompanied by the data username and password.

parameter

username string, username.

password string, password.

def createAccount(*username, password*):

Function description

Request to create a login account on the KBEngine server.
Note: If the plug-in and the UI layer use the event interaction mode, do not call directly from the UI layer. Please trigger a "createAccount" event to the plug-in. The event is accompanied by the data username and password.

parameters:

username string, username.

password string, password.

def reloginBaseapp( ):

Function description

Requests to re-login to the KBEngine server (usually used after a dropped connection in order to connect to the server more quickly and continue to control the server role).
Note: If the plug-in and the UI layer use event interaction mode, do not call directly from the UI layer, please trigger a "reloginBaseapp" event to the plug-in, and the incidental data is empty.

def player( ):

Function description

Gets the entity that the current client controls.

return:

Entity, return controlled entity, if it does not exist (e.g.: failed to connect to the server) returns null.

def resetPassword(*username*):

Function description

Asks loginapp to reset the password of the account. The server will send a password reset email (usually the forgotten password function) to the email address to which the account is bound.

parameters:

username string, username.

def bindAccountEmail(*emailaddress*):

Function description

Requests Baseapp to bind the email address of the account.

parameters:

emailaddress string, email address.

def newPassword(*oldpassword, newpassword*):

Function description

Requests to set a new password for the account.

parameters:

oldpassword string, old password

newpassword string, new password

def findEntity(*entityID*):

Function description

Finds an instance object of an entity by its ID.

parameters:

entityID int32, entity ID.

returns:

Entity An entity instance is returned. There can be no return null.

def getSpaceData(*key*):

Function description

Gets the space data for the specified key.
The space data is set by the user on the server through **setSpaceData**.

parameters:

key string, a keyword

returns:

string, specifies the value at the key

Callback function documentation

Attributes documentation

component

Description:

This is the component that is running in the current scripting environment. (So far) Possible values are 'cell', 'base', 'client', 'database', 'bot' and 'editor'.

entities

Description:

entities is a dictionary object that contains all the entities in the current process.

Types:
Entities

entity_uuid

Description:

The uuid of the entity. Change the ID and entity to bind to this login. When using the heavy login function, the server compares this ID and determines the validity.

entity_id

Description:

The ID of the entity controlled by the current client.

spaceID

Description:

The ID of the Space where the entity controlled by the current client is located (also can be understood as the corresponding scene, room, and copy).

KBEngine

Entity class

[KBEngine module]

Entity is part of the **KBEngine** module. **More...**

```
import KBEngine
```

Member functions

def **baseCall**(self, methodName, methodArgs):
def **cellCall**(self, methodName, methodArgs):

Callbacks

def **onDestroy**(self):
def **onEnterWorld**(self):
def **onLeaveWorld**(self):
def **onEnterSpace**(self):
def **onLeaveSpace**(self):

Attributes

direction	Tuple of 3 floats as (roll, pitch, yaw)
id	Read-only Integer
position	**Vector3**
spaceID	Read-only uint32
isOnGround	Read-only bool
inWorld	Read-only bool
className	Read-only string

A detailed description

Instances of class **Entity** represent game objects on the client.

An **Entity** can call methods on its equivalent entity in the base and cell applications via **ENTITYCALL**. This requires a set of remotely-invoked functions (specified in the entity's .def file). It also works the other way around, and a Client can have its functions remotely invoked by the entity's base and cell parts (must be specified in the <ClientMethods> section of the entity's .def file).

Client entities can have **cell** attribute changes broadcast to them by using any of the *_CLIENT* broadcast flags on properties in the entity's def file on the server side. If a property is set to be broadcast, set_<property>() is called on the client entity when a **cell** attribute is changed. See **http://kbengine.org/docs/programming/entitydef.html** for more info.

Member function documentation

def baseCall(*self, methodName, methodArgs*):

Function description:

The method to call the base part of the entity.
Note: the entity must have a base part on the server side. Only client entities controlled by the client can access this method.

Example:
js plugin: entity.baseCall("reqCreateAvatar", roleType, name);
c# plugin: entity.baseCall("reqCreateAvatar", new object[]{roleType, name});

parameters:

methodName string, method name.

methodArgs objects, method parameter list.

return:

Because it is a remote call, it is not possible to block waiting for a return, so there is no return value.

def cellCall(*self, methodName, methodArgs*):

Function description:

The method to call the cell part of this entity.
Note: The entity must have a cell part on the server. Only client entities controlled by the client can access this method.

Example:
js plugin: entity.cellCall("xxx", roleType, name);
c# plugin: entity.cellCall("xxx", new object[]{roleType, name});

parameters:

methodName string, method name.

methodArgs objects, method parameter list.

return:

Because it is a remote call, it is not possible to block waiting for a return, so there is no return value.

Callback function documentation

def onDestroy(*self*):

Called when the entity is destroyed

def onEnterWorld(*self*):

If the entity is not client-controlled, it indicates that the entity has entered the view scope of the client-controlled entity on the server, at which point the client can see the entity.
If the entity is client controlled, it indicates that the entity has created a cell on the server and entered the Space.

def onLeaveWorld(*self*):

If the entity is not client-controlled, it indicates that the entity has left the view scope of the client-controlled entity on the server side, and the client cannot see this entity at this time.
If the entity is client controlled, it indicates that the entity has already destroyed the cell on the server and left the Space.

def onEnterSpace(*self*):

The client-controlled entity enters a new space.

def onLeaveSpace(*self*):

The client-controlled entity leaves the current space.

Attribute documentation

className

The class name of the entity.

Type:
Read-only, string

position

The coordinates (x,y,z) of this entity in world space. The data is synchronized from the server to the client.

Type:
Vector3

direction

This attribute describes the orientation of the **Entity** in world space. Data is synchronized from the server to the client.

Type:
Vector3, which contains (roll, pitch, yaw) in radians.

isOnGround

If the value of this attribute is True, the **Entity** is on the ground, otherwise it is False.
If it is a client-controlled entity, this attribute will be synchronized to the server when changed, and other entities will be synchronized to the client by the server. The client can determine this value to avoid the overhead of accuracy.

Type:
Read-write, bool

KBEngine

KBEngine module

This **KBEngine** module provides the Python script access to the **entity**'s cell part, in particular it provides the registration and removal of timers, as well as the creation of **entities**.

Classes

Entity

Member functions

def **addSpaceGeometryMapping**(spaceID, mapper, path, shouldLoadOnServer, params):

def **addWatcher**(path, dataType, getFunction):

def **address**( ):

def **MemoryStream**( ):

def **createEntity**(entityType, spaceID, position, direction, params):

def **debugTracing**( ):

def **delSpaceData**(spaceID, key):

def **delWatcher**(path):

def **deregisterReadFileDescriptor**(fileDescriptor):

def **deregisterWriteFileDescriptor**(fileDescriptor):

def **executeRawDatabaseCommand**(command, callback, threadID, dbInterfaceName):

def **genUUID64**( ):

def **getResFullPath**(res):

def **getSpaceData**(spaceID, key):

def **getSpaceGeometryMapping**(spaceID):

def **getWatcher**(path):

def **getWatcherDir**(path):

def **getAppFlags**( ):

def **hasRes**(res):

def **isShuttingDown**( ):

def **listPathRes**(path, extension):

def **matchPath**(res):

def **open**(res, mode):

def **publish**( ):

def **registerReadFileDescriptor**(fileDescriptor, callback):

def **registerWriteFileDescriptor**(fileDescriptor, callback):

def **raycast**(spaceID, layer, src, dst):

def **reloadScript**(fullReload):

def **scriptLogType**(logType):

def **setAppFlags**(flags):
def **setSpaceData**(spaceID, key, value):
def **time**( ):

Callback

def **onCellAppData**(key, value):
def **onCellAppDataDel**(key):
def **onGlobalData**(key, value):
def **onGlobalDataDel**(key):
def **onInit**(isReload):
def **onSpaceData**(spaceID, key, value):
def **onSpaceGeometryLoaded**(spaceID, mapping):
def **onAllSpaceGeometryLoaded**(spaceID, isBootstrap, mapping):

Attributes

LOG_TYPE_DBG	
LOG_TYPE_ERR	
LOG_TYPE_INFO	
LOG_TYPE_NORMAL	
LOG_TYPE_WAR	
NEXT_ONLY	
cellAppData	
component	Read-only **string**
entities	**Entities**
globalData	GlobalDataClient

Member functions documentation

def addSpaceGeometryMapping(*spaceID, mapper, path, shouldLoadOnServer, params*):

Function description:
Associate a geometric mapping of a given space. After the function is called, the server and client will load the corresponding geometry data.

On the server, all geometry data is loaded from the given directory into the specified space. These data may be divided into many blocks. Different blocks are loaded asynchronously. The following callback methods are called when all the geometry data is loaded:

```
def onAllSpaceGeometryLoaded( self, spaceID, mappingName ):
```

The server only loads the geometric data of the scene for use by the navigation and collision functions. In addition to the geometric data, the client also loads data such as textures.
3D scenes currently use the data exported by the recastnavigation plugin-in by default. 2D scenes currently use the data exported by MapEditor by default.

There is a possibility that onAllSpaceGeometryLoaded() will not be invoked, that is, if multiple Cellapps call this method at the same time to add geometry to the same space, cellappmgr crashes.

parameters:

spaceID	uint32, ID of the space, specifies in which space to operate
mapper	Not yet implemented
path	Directory path containing geometry data
shouldLoadOnServer	Optional boolean parameter that specifies whether to load geometry on the server. Default is True.
params	Optional PyDict parameter, specifies the navmesh used by different layers, for example: KBEngine.addSpaceGeometryMapping(self.spaceID,

None, resPath, True, {0 :
"srv_xinshoucun_1.navmesh", 1 :
"srv_xinshoucun.navmesh"})

def addWatcher(*path, dataType, getFunction*):

Function description:

Interacts with the debug monitoring system to allow users to register a monitoring variable with the monitoring system.

Example:

```
>>> def countPlayers( ):
>>>     i = 0
>>>     for e in KBEngine.entities.values():
>>>             if e.__class__.__name__ == "Avatar":
>>>                     i += 1
>>>     return i
>>>
>>> KBEngine.addWatcher( "players", "UINT32", countPlayers )
```

This function adds a watch variable under the "scripts/players" watch path. The function countPlayers is called when the watcher observes a change.

parameters:

path	The path to create a watcher.
dataType	The value type of the monitored variable. Reference: **Basic data types**
getFunction	This function is called when the observer retrieves the variable. This function returns a value representing a watch variable without arguments.

def address( ):

Function description:

Returns the address of the internal network interface.

def MemoryStream( ):

Function description:

Return a new MemoryStream object.

The MemoryStream object stores binary information. This type is provided to allow the user to easily serialize and deserialize the Python base types following the same KBEngine underlying serialization rules.

For example, you can use this object to construct a network packet that KBEngine can parse.

Usage:

```
>>> s = KBEngine.MemoryStream()
>>> s
>>> b''
>>> s.append("UINT32", 1)
>>> s.pop("UINT32")
>>> 1
```

The types that MemoryStream currently supports are only basic data types.
Reference: **Basic data types**

def createEntity(*entityType, spaceID, position, direction, params*):

Function description:

createEntity creates a new entity in the specified space of the current process. When calling this function you must specifiy the type, location, and direction of the entity to be created. Optionally, any attribute of the entity can be set with the params Python dictionary parameter. (the attributes are described in the entity's .def file).

Example:

```
# Create an open Door entity in the same space as the "thing" entity
```

```
direction = ( 0, 0, thing.yaw )
properties = { "open":1 }
KBEngine.createEntity( "Door", thing.space, thing.position, directio
                       properties )
```

parameters:

entityType	string, the name of the entity to create, declared in the **/scripts/entities.xml** file.
spaceID	int32, the ID of the space to place the entity
position	A sequence of 3 floats that specify the creation point of the new entity, in world coordinates.
direction	A sequence of 3 floats that specify the initial orientation (roll, pitch, yaw) of the new entity in world coordinates.
params	Optional parameters, a Python dictionary object. If a specified key is an **Entity** attribute, its value will be used to initialize the properties of the new **Entity**.

returns:

The new **Entity**.

def debugTracing( ):

Function description:

Outputs the Python extension object counter that outputs KBEngine trace.
Extended objects include: fixed dictionary, fixed array, Entity, EntityCall...
If the counter is not zero when the server is shut down normally, it means that the leak already exists and the log will output an error message.

ERROR cellapp [0x0000cd64] [2014-11-12 00:38:07,300] - PyGC::debugTracing(): FixedArray : leaked(128)
ERROR cellapp [0x0000cd64] [2014-11-12 00:38:07,300] - PyGC::debugTracing(): EntityCall : leaked(8)

def delSpaceData(*spaceID, key*):

Function description:

Deletes the space data of the specified key (if space is divided into multiple

parts, it will be deleted synchronously).
The space data is set by the user via **setSpaceData**.

parameters:
spaceID int32, the ID of the space
key string, a string keyword

def delWatcher(*path*):

Function description:

Interacts with the debug monitoring system, allowing users to delete watcher variables in the script.

parameters:
path The path of the variable to delete.

def deregisterReadFileDescriptor(*fileDescriptor*):

Function description:

Deregisters the callback registered with **KBEngine.registerReadFileDescriptor**.

Example:
http://www.kbengine.org/assets/other/py/Poller.py

parameters:
fileDescriptor Socket descriptor/file descriptor

def deregisterWriteFileDescriptor(*fileDescriptor*):

Function description:

Deregisters the callback registered with **KBEngine.registerWriteFileDescriptor**.

Example:
http://www.kbengine.org/assets/other/py/Poller.py

parameters:

fileDescriptor Socket descriptor/file descriptor

def executeRawDatabaseCommand(*command, callback, threadID, dbInterfaceName*):

Function description:

This script function executes a database command on the database, which will be directly parsed by the relevant database.

Please note that using this function to modify the entity data may not be effective because if the entity has been checked out, the modified entity data will still be archived by the entity and cause overwriting.
This function is strongly not recommended for reading or modifying entity data.

parameters:

command This database command will be different for different database configurations. For MySQL databases it will be an SQL query statement.

Optional parameter, callbacks object (for example, a function) with the command's execution result. This callback has 4 parameters: result set, number of rows affected, auto value, and error message.
Example:

```
def sqlcallback(result, rows, insertid, error):
    print(result, rows, insertid, error)
```

As the above example shows, the result parameter corresponds to the "result set", and the result set parameter is a row. List. Each line is a list of strings containing field values.
The command execution does not return a result set (for example, a DELETE command), or the result set is None if the command execution has an error.

The rows parameter is the "number of rows affected", which is an integer indicating the number of rows affected

callback	by the command execution. This parameter is only relevant for commands that do not return results (such as DELETE). This parameter is None if there is a result set return or if there is an error in the command execution. The insertid corresponds to a "long value", similar to the entity's databaseID. When successfully inserting data into a table with an auto long type field, it returns the data at the time of insertion. Assigned value. More information can be found in mysql's mysql_insert_id() method. In addition, this parameter is only meaningful when the database type is mysql. error corresponds to the "error message", when the command execution error, this parameter is a string describing the error. This parameter is None when the command execution has not occurred.
threadID	int32, optional parameter, specifies a thread to process this command. Users can use this parameter to control the execution order of certain commands (dbmgr is multi-threaded). The default is not specified. If threadID is the ID of the entity, it will be added to the entity's archive queue and written by the thread one by one.
dbInterfaceName	string, optional parameter, specified by a database interface, defaults to "default" interface. The database interface is defined by kbengine_defaults.xml->dbmgr->databaseInterfaces.

def genUUID64( ):

Function description:

This function generates a 64-bit unique ID.
Note: This function is dependent on the 'gus' startup argument of the Cellapp service process. Please set the startup arguments to be unique.
In addition, if gus exceeds 65535, this function can only remain unique on the

current process.

Usage
A unique item ID is generated on multiple service processes and there is no conflict when combined.
A room ID is generated on multiple service processes and no uniqueness verification is required.

returns:
64-bit integer

def getResFullPath(*res*):

Function description:

Get the absolute path of the resource.
Note: Resoures must be accessible under **KBE_RES_PATH**.

parameters:
res string, if there is an absolute path to return the resource, otherwise it returns null.

returns:
string, the absolute path to the resource.

def getSpaceData(*spaceID, key*):

Function description:

Get the space data of the specified key.
The space data is set by the user via **setSpaceData**.

parameters:
spaceID int32, the ID of the space
key string, a string keyword

returns:
string, string data for the given key

def getSpaceGeometryMapping(*spaceID*):

Function description:

Returns the geometry map name of a specified space.

parameters:

spaceID The ID of the space to be queried

returns:

string, the name of the geometry map.

def getWatcher(*path*):

Function description:

Gets the value of a watch variable from the KBEngine debug system.

Example: In the baseapp1 Python console, enter:
>>>KBEngine.getWatcher("/root/stats/runningTime")
12673648533

>>>KBEngine.getWatcher("/root/scripts/players")
32133

parameters:

path string, the absolute path of the variable including the variable name (can be viewed on the GUIConsole watcher page).

returns:

The value of the variable.

def getWatcherDir(*path*):

Function description:

Get a list of elements (directories, variable names) under the watch directory from the KBEngine debugging system.

Example: In baseapp1 Python console, enter:
>>>KBEngine.getWatcher("/root")
('stats', 'objectPools', 'network', 'syspaths', 'ThreadPool', 'cprofiles', 'scripts', 'numProxices', 'componentID', 'componentType', 'uid', 'numClients', 'globalOrder', 'username', 'load', 'gametime', 'entitiesSize', 'groupOrder')

parameters:

path string, the absolute path of the variable including the variable name (can be viewed on the GUIConsole watcher page).

returns:

The list of elements (directory, variable name) under the Watch directory.

def getAppFlags( ):

Function description:

Get the flags of the current engine APP, Reference:**KBEngine.setAppFlags**¡£

returns:

KBEngine.APP_FLAGS_*

def hasRes(*res*):

Function description:

Use this interface to determine if a relative path exists.
Note: Resources must be accessible under **KBE_RES_PATH**.

Example:

>>>KBEngine.hasRes("**scripts/entities.xml**")
True

parameters:

res string, the relative path of the resource

returns:

BOOL, if it exists return True, otherwise False.

def isShuttingDown( ):

Function description:

Returns whether the server is shutting down. After the onBaseAppShuttingDown callback function is called, this function returns True.

returns:

BOOL, if the server is shutting down True, otherwise False.

def listPathRes(*path, extension*):

Function description:

Get a list of resources in a resource directory
Note: Resources must be accesible under **KBE_RES_PATH**.

Example:

>>>KBEngine.listPathRes("scripts/cell/interfaces")
('/home/kbe/kbengine/demo/res/scripts/cell/interfaces/AI.py',
'/home/kbe/kbengine/demo/res/scripts/cell/interfaces/New Text Document.txt')

>>>KBEngine.listPathRes("scripts/cell/interfaces", "txt")
('/home/kbe/kbengine/demo/res/scripts/cell/interfaces/New Text Document.txt')

>>>KBEngine.listPathRes("scripts/cell/interfaces", "txt|py")
('/home/kbe/kbengine/demo/res/scripts/cell/interfaces/AI.py',
'/home/kbe/kbengine/demo/res/scripts/cell/interfaces/New Text Document.txt')

>>>KBEngine.listPathRes("scripts/cell/interfaces", ("txt", "py"))
('/home/kbe/kbengine/demo/res/scripts/cell/interfaces/AI.py',
'/home/kbe/kbengine/demo/res/scripts/cell/interfaces/New Text Document.txt')

parameters:

res string, the relative path of the resource.
extension string, optional parameter, file extension.

returns:
Tuple, resource list.

def matchPath(*res*):

Function description:

Use the relative path of the resource to get its absolute path.
Note: Resources must be accessible under **KBE_RES_PATH**.

Example:

>>>KBEngine.matchPath("scripts/entities.xml")
'/home/kbe/kbengine/demo/res/scripts/entities.xml'

parameters:
res string, the relative path to the resource (including the resource name)

returns:
string, the absolute path of the resource.

def open(*res, mode*):

Function description:

Use this interface to open resources using relative paths. Note: Resources must be accessible under **KBE_RES_PATH**.

parameters:
res string, the relative path of the resource.
string, file operation mode:
w Open in write mode,
a Open in append mode (Start from EOF, create new file if necessary)
r+ Open
w+ in read/write mode Open in read/write mode (see w)
a+ Open in read/write mode (See a)
mode rb Opens
wb in binary read mode Opens in binary write mode (see w)

ab Opens in binary append mode (see a)
rb+ Opens in binary read and write mode (see r+)
wb+ Opens in binary read and write mode (see w+)
ab+ Open in binary read/write mode (see a+)

def publish( ):

Function description:

This interface returns the current server release mode.

returns:

int8, 0: debug, 1: release, others can be customized.

def raycast(*spaceID, layer, src, dst*):

Function description:

In the specified layer of the specified space, a ray is emitted from the source coordinates to the destination coordinates, and the collided coordinate point is returned.

Note: Space must load geometry using **addSpaceGeometryMapping**.

Below is an example:

```
>>> KBEngine.raycast( spaceID, entity.layer, (0, 10, 0), (0,
((0.0000, 0.0000, 0.0000), ( (0.0000, 0.0000, 0.0000),
(4.0000, 0.0000, 0.0000), (4.0000, 0.0000, 4.0000)), 0)
```

parameters:

spaceID int32, space ID

layer int8, geometric layer. A space can load multiple navmesh data at the same time. Different navmesh can be in different layers. Different layers can be abstracted into the ground, the water surface and so on.

returns:

list, list of coordinate points collided

def registerReadFileDescriptor(*fileDescriptor, callback*):

Function description:

Registers a callback function that is called when the file descriptor is readable.

Example:
http://www.kbengine.org/assets/other/py/Poller.py

parameters:

fileDescriptor	Socket descriptor/file descriptor
callback	A callback function with the socket descriptor/file descriptor as its only parameter.

def registerWriteFileDescriptor(*fileDescriptor, callback*):

Function description:

Registers a callback function that is called when the socket descriptor/file descriptor is writable.

Example:
http://www.kbengine.org/assets/other/py/Poller.py

parameters:

fileDescriptor	Socket descriptor/file descriptor
callback	A callback function with the socket descriptor/file descriptor as its only parameter.

def reloadScript(*fullReload*):

Function description:

Reloads Python modules related to entity and custom data types. The current entity class is set to the newly loaded class. This method should only be used for development mode and not for product mode. The following points should be noted:

1) The overloaded script can only be executed on **Cellapp**. The user should

ensure that all server components are loaded.

2) The custom type should ensure that the objects already instantiated in memory are updated after the script is reloaded. Here is an example:

```
for e in KBEngine.entities.values():
   if type( e ) is Avatar.Avatar:
      e.customData.__class__ = CustomClass
```

When this method completes **KBEngine.onInit**(True) is called.

parameters:

fullReload Optional boolean parameter that specifies whether to reload entity definitions at the same time. If this parameter is False, the entity definition will not be reloaded. The default is True.

returns:

True if the reload succeeds, False otherwise.

def scriptLogType(*logType*):

Function description:

Set the type of information output by the current Python.print (Reference: KBEngine.LOG_TYPE_*).

def setAppFlags(*flags*):

Function description:

Set the flags of the current engine APP.

KBEngine.APP_FLAGS_NONE // Default (not set)
KBEngine.APP_FLAGS_NOT_PARTCIPATING_LOAD_BALANCING // Do not participate in load balancing

Example:
KBEngine.setAppFlags(KBEngine.APP_FLAGS_NOT_PARTCIPATING_LOAI | KBEngine.APP_FLAGS_*)

def setSpaceData(*spaceID, key, value*):

Function description:

Sets the space data for the specified key.
The space data can be obtained via **getSpaceData**.

parameters:

spaceID int32, the ID of the space.
key string, a string keyword
value string, the string value.

def time( ):

Function description:

This method returns the current game time (number of cycles).

returns:

uint32, the current time of the game. This refers to the number of cycles. The period is affected by the frequency. The frequency is determined by the configuration file **kbengine.xml** or **kbengine_defaults.xml**-> gameUpdateHertz.

Callback functions documentation

def onCellAppData(*key, value*):

Function description:

This function is called back when KBEngine.cellAppData changes.
Note: This callback interface must be implemented in the portal module (**kbengine_defaults.xml**->entryScriptFile).

parameters:

key The key of the changed data.

value The value of the changed data.

def onCellAppDataDel(*key*):

Function description:

This function is called back when KBEngine.cellAppData is deleted.
Note: This callback interface must be implemented in the portal module (**kbengine_defaults.xml**->entryScriptFile).

parameters:

key Deleted data key.

def onGlobalData(*key, value*):

Function description:

This function is called back when KBEngine.globalData changes.
Note: This callback interface must be implemented in the portal moodule (**kbengine_defaults.xml**->entryScriptFile).

parameters:

key The key of the changed data.

value The value of the changed data.

def onGlobalDataDel(*key*):

Function description:

This function is called back when KBEngine.globalData is deleted.
Note: This callback interface must be implemented in the portal module (**kbengine_defaults.xml**->entryScriptFile).

parameters:

key Deleted data key.

def onInit(*isReload*):

Function description:

This interface is called after all scripts have been initialized since the engine started.
Note: This callback interface must be implemented in the portal module (**kbengine_defaults.xml**->entryScriptFile).

parameters:

isReload Bool, whether it was triggered after reloading the loading the script.

def onSpaceData(*spaceID, key, value*):

Function description:

Called when there is a change in the space data.
The space data is set by the user via **setSpaceData**.

parameters:

spaceID The ID of the space.
key The key of the changed data.
value The value of the changed data.

def onSpaceGeometryLoaded(*spaceID, mapping*):

Function description:

The space required by the grid collision data is loaded.
Set by user through **addSpaceGeometryMapping**.

parameters:

spaceID The ID of the space.

mapping The map value of the grid collision data.

def onAllSpaceGeometryLoaded(*spaceID, isBootstrap, mapping*):

Function description:

The space required for grid collision and other data is completely loaded.
Set by user through **addSpaceGeometryMapping**.

parameters:

spaceID	The ID of the space.
isBootstrap	If a space is partitioned by multiple cells, isBootstrap describes whether it is the originating cell of the loading request.
mapping	The map value of grid collision data.

Attribute documentation

LOG_TYPE_DBG

Description:

The log output type is debug.
Set by **scriptLogType**.

LOG_TYPE_ERR

Description:

The log output type is error.
Set by **scriptLogType**.

LOG_TYPE_INFO

Description:

The log output type is general information.
Set by **scriptLogType**.

LOG_TYPE_NORMAL

Description:

The log output type is normal.
Set by **scriptLogType**.

LOG_TYPE_WAR

Description:

The log output type is warning.
Set by **scriptLogType**.

NEXT_ONLY

Description:

This constant is currently unused in Cellap.

cellAppData

Description:

This property contains a dictionary-like object that is automatically synchronized across all CellApps. When a value in the dictionary is modified, this change is broadcast to all Cellapps.

Example:

```
KBEngine.cellAppData[ "hello" ] = "there"
```

The rest of Cellap can access the following:

```
print KBEngine.cellAppData[ "hello" ]
```

Keys and values can be of any type, but these types must be encapsulated and unpacked on all target components.

When a value is changed or deleted, a callback function is called on all components. See: **KBEngine.onCellAppData** and **KBEngine.onDelCellAppData**.

Note: Only the top-level value will be broadcast. If you have a value (such as a list) that changes the internal value (such as just changing a number), this information will not be broadcast.

Do not do the following:

```
KBEngine.cellAppData[ "list" ] = [1, 2, 3]
KBEngine.cellAppData[ "list" ][1] = 7
```

This will cause the local access to read [1, 7, 3] and the remote [1, 2, 3]

component

Description:

This is the component that is running in the current Python environment. (So far) Possible values are 'cell', 'base', 'client', 'database', 'bot' and 'editor'.

entities

Description:

entities is a dictinary object that contains all entities in the current process, including **ghost** entities.

Debugging leaked entities: (instances that call destroy without releasing memory, usually as a result of the reference not being released)

```
>>> KBEngine.entities.garbages.items()
[(1025, Avatar object at 0x7f92431ceae8.)]


>>> e = _[0][1]
>>> import gc
>>> gc.get_referents(e)
[{'spacesIsOk': True, 'bootstrapIdx': 1}, ]
```

Debugging a leaked KBEngine-encapsulated Python object:
KBEngine.debugTracing

Types:

Entities

globalData

Description:

This attribute contains a dictionary-like object that is automatically copied between all BaseApps and CellApps. When a value in a dictionary is modified, this change is broadcast to all BaseApps and CellApps. CellAppMgr solves completion conditions and ensures the authority of information replication.

Example:

```
KBEngine.globalData[ "hello" ] = "there"
```

The rest of **Cellapp** or **Baseapp** can access the following:

```
print KBEngine.globalData[ "hello" ]
```

Keys and values can be of any type, but these types must be encapsulated and unpacked on all target components.

When a value is changed or deleted, a callback function is called on all components. See: **KBEngine.onGlobalData** and **KBEngine.onGlobalDataDel**.

Note: Only the top-level value will be broadcast. If you have a volatile value (such as a list) that changes the internal value (such as just changing a number), this information will not be broadcast.

Do not do the following:

```
KBEngine.globalData[ "list" ] = [1, 2, 3]
KBEngine.globalData[ "list" ][1] = 7
```

This will cause the local access to read [1, 7, 3] and the remote [1, 2, 3]

KBEngine

Entity

[KBEngine module]

Entity is part of the **KBEngine** module. **More...**

```
import KBEngine
```

Member functions

def **accelerate**(self, accelerateType, acceleration):
def **addYawRotator**(self, targetYaw, velocity, userArg):
def **addProximity**(self, range, userArg):
def **addTimer**(self, start, interval=0.0, userData=0):
def **cancelController**(self, controllerID):
def **clientEntity**(self, destID):
def **canNavigate**(self):
def **debugView**(self):
def **delTimer**(self, id):
def **destroy**(self):
def **destroySpace**(self):
def **entitiesInView**(self):
def **entitiesInRange**(self, range, entityType=None, position=None):
def **isReal**(self):
def **moveToEntity**(self, destEntityID, velocity, distance, userData, faceMovement, moveVertically, offsetPos):
def **moveToPoint**(self, destination, velocity, distance, userData, faceMovement, moveVertically):
def **getViewRadius**(self):
def **getViewHystArea**(self):
def **getRandomPoints**(self, centerPos, maxRadius, maxPoints, layer):
def **navigate**(self, destination, velocity, distance, maxMoveDistance, maxSearchDistance, faceMovement, layer, userData):
def **navigatePathPoints**(self, destination, maxSearchDistance, layer):
def **setViewRadius**(self, radius, hyst=5):
def **teleport**(self, nearbyMBRef, position, direction):
def **writeToDB**(self, shouldAutoLoad, dbInterfaceName):

Callbacks

def **onDestroy**(self):
def **onEnterTrap**(self, entity, rangeXZ, rangeY, controllerID, userArg):
def **onEnteredView**(self, entity):
def **onGetWitness**(self):
def **onLeaveTrap**(self, entity, rangeXZ, rangeY, controllerID, userArg):
def **onLoseControlledBy**(self, id):
def **onLoseWitness**(self):
def **onMove**(self, controllerID, userData):
def **onMoveOver**(self, controllerID, userData):
def **onMoveFailure**(self, controllerID, userData):
def **onRestore**(self):
def **onSpaceGone**(self):
def **onTurn**(self, controllerID, userData):
def **onTeleport**(self):
def **onTeleportFailure**(self):
def **onTeleportSuccess**(self, nearbyEntity):
def **onTimer**(self, timerHandle, userData):
def **onUpdateBegin**(self):
def **onUpdateEnd**(self):
def **onWitnessed**(self, isWitnessed):
def **onWriteToDB**(self):

Attributes

allClients	Read-only PyClient
base	Read-only **BaseEntityCall**
client	Read-only **ClientEntityCall**
controlledBy	**BaseEntityCall**
className	Read-only **string**
direction	Tuple of 3 floats as (roll, pitch, yaw)
hasWitness	boolean
id	Read-only Integer
isDestroyed	Read-only bool
isWitnessed	Read-only bool
layer	int8
otherClients	Read-only PyClient
position	**Vector3**
spaceID	Read-only uint32
topSpeed	float
topSpeedY	float
volatileInfo	float

A detailed description

Instances of class **Entity** represent game objects in the cell. An **Entity** can either be "real" or "ghosted". A "ghost" **Entity** is a copy of a "real" **Entity** living on a neighboring cell. There is a unique "real" **Entity** instance for each entity, and 0 or more "ghost" **Entity** instances.

An **Entity** instance controls the location data of the entity, including its position and rotation. It also controls how often this data is sent to the client (if possible). The location data can be updated by a unique client and modified by the controller object using the teleport member function. Controllers are non-python objects that can be used to change the location data over time on cell entities. They are added to **Entity** through member functions such as "trackEntity" and "turnToYaw" and can be removed via "cancelController".

Area of Interest or "View" is an important concept for all **KBEngine** entities that belong to a client. The view of an entity is the area that the client (if it has a client) can perceive around this entity. This is used to select the amount of data sent to the client. The actual shape of the View is defined by the range of distances on the x-axis and around the z-axis, and there is a lag region that extends outward like a shape. An **Entity** enters another **Entity**'s view, but does not leave it until it leaves the lag area. An **Entity** can modify its View size via "setViewRadius". You can find all entities within a specific distance with "entitiesInRange" and set traps to capture all entities that enter the trap with "addProximity".

The new **Entity** on cellapp can be created using **KBEngine**.**createEntity**. An entity can also be created by the baseapp remote calls to the **KBEngine**.**createCellEntity** function.

An **Entity** can access it equivalent entities on the base and client applications via **ENTITYCALL**. This requires a set of remotely-invoked functions (specified in the entity's .def file)

Member function documentation

def accelerate(*self, accelerateType, acceleration*):

Function description:

Accelerate the current movement of the entity.

The activities that can be accelerated include:

Entity.**moveToEntity**
Entity.**moveToPoint**
Entity.**navigate**
Entity.**addYawRotator**

parameters:

accelerateType	string, the type of movement affected such as: "Movement", "Turn".
velocity	float, acceleration per second, use negative to decelerate

returns:

The current speed of the affected entity.

def addYawRotator(*self, targetYaw, velocity, userArg*):

Function description:

The control entity rotates around yaw. **Entity**.**onTurn** is called when the rotation completes.

To remove it, use **Entity**.**cancelController** with the controller ID or use **Entity**.**cancelController("Movement")** to remove it.

See:

Entity.**cancelController**

parameters:

targetYaw float, the given target yaw radians.

velocity float, the arc per second when rotated.

userArg Optional integer that is common to all controllers. If this value is not 0, it is passed to the callback function. It is recommended to set the default value to 0 in the callback prototype.

def addProximity(*self, rangeXZ, rangeY, userArg*):

Function description:

Create an area trigger that will notify the **Entity** when other entities enter or leave the trigger area. This area is a square (for efficiency).

If another entity is within a given distance on the x-axis and z-axis, it is considered to be within the range. This **Entity** is notified via the onEnterTrap and onLeaveTrap functions, which can be defined as follows:

```
def onEnterTrap( self, entityEntering, rangeXZ, rangeY, cont
def onLeaveTrap( self, entityLeaving, rangeXZ, rangeY, contr
```

Because the scope trigger is a controller, use **Entity**.**cancelController** with the controller ID to delete it.

It should be noted that the callback may be triggered immediately, even before the call to addProximity() returns.

See:

Entity.**cancelController**

parameters:

rangeXZ float, the size of the xz axis area of the trigger, must be greater than or equal to zero.

float, the height of the y-axis of the trigger, must be greater than or equal to zero.
It should be noted that for this parameter to take effect **kbengine_defaults.xml**->cellapp->coordinate_system->rangemgr_y must be set to true.

rangeY Open y-axis management will increase CPU consumption, because some games have a large number of entities at the same y-axis height or all on the ground which is almost completely flat. Because of this, the collision becomes very dense.
3D space games or small room-type games are more suitable for this option.

userArg Optional integer that is common to all controllers. If this value is not 0, it is passed to the callback function. It is recommended to set the default value to 0 in the callback prototype.

returns:

The ID of the created controller.

def addTimer(*self, start, interval=0.0, userData=0*):

Function description:

Register a timer. The timer triggers the callback function ***onTimer*** The callback function will be executed the first time after "initialOffset" seconds, and then it will be executed once every "repeatOffset" seconds with the "userArg" parameter. (integer only)

The ***onTimer*** function must be defined in the cell part of the entity with two parameters. The first is an integer, the timer's ID (which can be used to remove the timer's "**delTimer**" function), and the second is the user parameter "userArg".

Example:

```
# Here is an example of using addTimer
import KBEngine

class MyCellEntity( KBEngine.Entity ):

    def __init__( self ):
        KBEngine.Entity.__init__( self )

        # Add a timer, perform the first time after 5 seconds, and e
        self.addTimer( 5, 1, 9 )

        # Add a timer and execute it after 1 second. The default use
```

```
        self.addTimer( 1 )

    # Entity timer callback "onTimer" is called
    def onTimer( self, id, userArg ):
        print "MyCellEntity.onTimer called: id %i, userArg: %i" % (
        # if this is a repeated timer, when it is no longer needed,
        #     self.delTimer( id )
```

parameters:

initialOffset	float, specifies the time interval (in seconds) for the timer to execute the first callback.
repeatOffset	float, specifies the time interval (in seconds) between each callback after the first callback. The timer must be removed with the function **delTimer**, otherwise it will be repeated. Values less than or equal to 0 will be ignored.
userArg	integer, specifies the value of the "userArg" parameter when calling "**onTimer**".

returns:

integer, returns the internal ID of the timer. This ID can be used to remove the timer using **delTimer**.

def cancelController(*self, controllerID*):

Function description:

The function cancelController stops the effect of a controller on **Entity**. It can only be called on a **real** entity.

parameters:

controllerID	integer, the index of the controller to cancel. A special controller type string can also be used as its type. For example, only one mobile/navigation controller can be activated at a time. This can be cancelled with entity.cancelController("Movement").

def clientEntity(*self, destID*):

Function description:

This method can access the method of an entity in its own client (the current

entity must be bound to the client). Only the entities in the View scope will be synchronized to the client. It can only be called on a **real** entity.

parameters:

destID integer, the ID of the target entity.

def canNavigate(*self*):

Function description:

This method determines whether the current entity can use the navigation (**Entity**.**navigate**) feature. It can only be called on a **real** entity.
Usually it can use navigation when the entity's Space uses **KBENgine**.**addSpaceGeometryMapping** to load valid navigation collision data (Navmesh or 2D tile data) and the entity is available in the effective navigation area.

returns:

bool, returns True if the entity can use the Navigate function in the current space, otherwise it returns False.

def debugView(*self*):

Function description:

debugView outputs the **Entity**'s View details to the cell's debug log. A description of the workings of the View system can be found in the **Entity** class documentation.

A sample of information is as follows:

```
INFO cellapp [0x00001a1c] [2014-11-04 00:28:41,409] - Avatar::debu
INFO cellapp [0x00001a1c] [2014-11-04 00:28:41,409] - Avatar::debu
INFO cellapp [0x00001a1c] [2014-11-04 00:28:41,409] - Avatar::debu
INFO cellapp [0x00001a1c] [2014-11-04 00:28:41,409] - Avatar::debu
INFO cellapp [0x00001a1c] [2014-11-04 00:28:41,409] - Avatar::debu
```

The first line of information tells us:

- It is entity #1000's data.
- There are 4 entities in its View area and all have been synchronized to the

client.
- There are 0 entities in its view Area that are waiting to be synchronized to the client.
- The radius of the View is 50.000
- The lag area of the View extends 5.000 outward.

def delTimer(*self, id*):

Function description:

The delTimer function is used to remove a registered timer. The removed timer is no longer executed. Single shot timers are automatically removed after the callback is executed, and it is not necessary to use the delTimer to remove it. If the delTimer function uses an invalid ID (for example, it has been removed), an error will be generated.

parameters:

id integer, which specifies the timer ID to remove.

def destroy(*self*):

Function description:

This function destroys its local **Entity** instance. If the entity has a **ghost** part on other processes, it will also notify for their destruction. This function is best called by the entity itself, and throws an exception if the entity is a **ghost**. If the callback function onDestroy() is implemented, it is executed.

def destroySpace(*self*):

Function description:

Destroys the **space** this entity is in.

def entitiesInView(*self*):

Function description:

Get a list of entities in the **View** scope of this entity.

def entitiesInRange(*self, range, entityType=None, position=None*):

Function description:

Search for **entities** within a given distance. This is a spherical search. The distances of the three axes must be measured. This can find **entities** that are outside the **View** scope of this entity, but cannot find entities in other **cells**.

Example:

```
self.entitiesInRange( 100, 'Creature', (100, 0, 100) )
```

Searches for a list of entities of type 'Creature' (an instantiated entity of a subclass of 'Creature'). The center point is (100, 0, 100) and the search radius is 100 meters.

```
[ e for e in self.entitiesInRange( 100, None, (100,0,100) ) if isi
```

Gives a list of entities instantiated from subclasses of 'BaseType'.

parameters:

range	Search distance around this entity, float type
entityType	An optional string parameter, the entity's type name, used to match entities. If the entity type is a valid class name (valid entities are ones listed in **/scripts/entities.xml**) only this type of entity will be returned, otherwise all entities in this range will be returned.
position	Optional **Vector3** type parameter, which is the center of the search radius is centered on the entity itself by default.

returns:

A list of **Entity** objects in a given range.

def isReal(*self*):

Function description:

This function returns whether the **Entity** is **real** or a **ghost**.

This function is rarely used but is useful for debugging.

returns:

bool, True if real, otherwise False.

def moveToEntity(*self, destEntityID, velocity, distance, userData, faceMovement, moveVertically, offsetPos*):

Function description:

Moves the **Entity** straight to another **Entity** position.
Any **Entity** can only have one motion controller at any time. Repeatedly calling any move function will terminate the previous move controller.
This function will return a controller ID that can be used to cancel this move.

For example, **Entity**.**cancelController**(movementID). You can also cancel the move using **Entity**.**cancelController**("Movement"). The callback function will not be called if the move is cancelled.

```
def onMove( self, controllerID, userData ):
def onMoveOver( self, controllerID, userData ):
def onMoveFailure( self, controllerID, userData ):
```

References:

Entity.**cancelController**

parameters:

destEntityID	int, the ID of the target **Entity**
velocity	float, speed of the **Entity** move, in m/s
distance	float, distance target that when reached the entity will stop moving, if the value is 0, it moves to the target position.
userData	object, optional parameter, when the callback function is invoked the userData parameter will be this value.
faceMovement	bool, optional parameter, True if the entity faces the direction of the move. If it is other mechanism, it is False.
moveVertically	bool, optional parameter, set to True to move in a straight line, set to False means to move in a straight line parallel to the ground.
	Vector3, optional parameter, Set a certain offset value, such as

offsetPos	moving the target position to the left of the entity.

returns:

int, newly created controller ID.

def moveToPoint(*self, destination, velocity, distance, userData, faceMovement, moveVertically*):

Function description:

Move the **Entity** to the given coordinate point in a straight line. The callback function is invoked on success or failure.
Any **Entity** can only have one motion controller at any time. Repeatedly calling any move function will terminate the previous move controller.
Returns a controller ID that can be used to cancel this move.

For example:
Entity.**cancelController**(movementID). You can also cancel the move with **Entity**.**cancelController**("Movement"). The callback function will not be called if the move is cancelled.

The callback function is defined as follows:

```
def onMove( self, controllerID, userData ):
def onMoveOver( self, controllerID, userData ):
def onMoveFailure( self, controllerID, userData ):
```

See:

Entity.**cancelController**

parameters:

destination	Vector3, the target point to which the **Entity** is to be moved
velocity	float, **Entity**'s moving speed, in m/s
distance	float, distance target that when reached the entity will stop moving, if the value is 0, it moves to the target position.
userData	object, data passed to the callback function
faceMovement	bool, True if the entity faces the direction of the move. If it is other mechanism, it is false.

moveVertically bool, set to True to move in a straight line, set to False means to move in a straight line parallel to the ground.

returns:

int, newly created controller ID.

def getViewRadius(*self*):

Function description:

This function returns the current View radius value of this **Entity**.

Data can be set via **Entity**.**setViewRadius**(radius, hyst).

returns:

float, View radius

def getViewHystArea(*self*):

Function description:

This function returns the current lag area value of this **Entity** View.

Data can be set via **Entity**.**setViewRadius**(radius, hyst).

returns:

float, The current lag area value of this Entity's View.

def getRandomPoints(*self, centerPos, maxRadius, maxPoints, layer*):

Function description:

This function is used to get an array of random coordinate point that **Entity**.**navigate** can reach in a certain area centered on a certain coordinate point.

parameters:

centerPos Vector3, **Entity** center coordinates

maxRadius float, the maximum search radius

maxPoints	uint32, the maximimum number of random coordinate points returned.
layer	int8, layer of navmesh to search.

returns:

tuple, an array of one or more coordinates.

def navigate(*self, destination, velocity, distance, maxMoveDistance, maxSearchDistance, faceMovement, layer, userData*):

Function description:

Use the navigation system to move this **Entity** to a target point. A callback will be invoked on success or failure.
KBEngine can have several pre-generated navigation meshes with different mesh sizes (leading to different navigation paths).
Any **Entity** can only have one motion controller at any time. Repeatedly calling any move function will terminate the previous move controller.
Returns a controller ID that can be used to cancel this move.

For example:
Entity.**cancelController**(movementID). You can also cancel the movement controller with **Entity**.**cancelController**("Movement"). The callback function will not be called if the move is cancelled.

The callback functions are defined as follows:

```
def onMove( self, controllerID, userData ):
def onMoveOver( self, controllerID, userData ):
def onMoveFailure( self, controllerID, userData ):
```

See:

Entity.**cancelController**

parameters:

destination	Vector3, the target point where the **Entity** moves.
velocity	float, **Entity**'s move speed, in m/s
distance	float, distance target that when reached the entity will stop moving, if the value is 0, it moves to the target

	position.
maxMoveDistance	float, the maximum move distance
maxSearchDistance	float, the maximum search distance from the navigatio ndata.
faceMovement	bool, True if the entity faces the direction of the move (default). Otherwise False.
layer	int8, navmesh layer to search
userData	object, the data passed to the callback function

returns:

int, the newly created controller ID.

def navigatePathPoints(*self, destination, maxSearchDistance, layer*):

Function description:

This functions returns a list of path points from the current **Entity** location to the destination.

parameters:

destination	Vector3, target point where the **Entity** moves
maxSearchDistance	float, the maximum search distance
layer	int8, navmesh layer to search for a path on.

def setViewRadius(*self, radius, hyst=5*):

Function description:

Specifies the size of the **Entity**'s **View**.

This function can only be used by **Witness** related entities.

Note: You can set the default View radius by setting the **kbengine.xml** configuration option 'cellapp/defaultViewRadius'.

Data can be obtained with **Entity**.**getViewRadius**( ) and **Entity**.**getViewHystArea**( ).

parameters:

radius	float, specifies the radius of the View area
hyst	float, specifies the size of the lag area of the View. A reasonable setting of the lag area will reduce the sensitivity of View collisions and reduce CPU consumption. Views where one entity enters another entity must span the View radius area, but entities that leave the View area need to move out of the View radius area including the lag area.

returns:

None

def teleport(*self, nearbyMBRef, position, direction*):

Function description:

Instantly move an **Entity** to a specified space. This function allows you to specify the position and orientation of the entity after is has been moved.
If you need to jump in different **spaces** (usually for different scene or room jumps), you can pass a **CellEntityCall** to this function (the entity corresponding to the entityCall must be in the destination **Space**).

This function can only be called on **real** entities.

parameters:

nearbyMBRef	A **CellEntityCall** (the entity corresponding to this entityCall must be in the destination **Space**) that determines which **Space** an **Entity** is to jump to. It is considered to be the transfer destination. This can be set to None, in which case it will teleport on the current cell.
position	A sequence of 3 floats (x, y, z), the coordinates of where to teleport the **Entity** .
direction	A sequence of 3 floats (roll, pitch, yaw), the orientation of the **Entity** after teleportation.

def writeToDB(*self, shouldAutoLoad, dbInterfaceName*):

Function description:

This function saves the data related to this entity to the database, including the data of the base entity. The onWriteToDB function of the base entity is called before the data is passed to the database.

The data of the cell entity is also backed up in the base entity to ensure that the data is up-to-date when crash recovery data is encountered

This function can only be called on **real** entities, and the entity must exist in the base section.

parameters:

shouldAutoLoad	bool, optional parameter, specifies whether this entity needs to be loaded from the database when the service starts. Note: The entity is automatically loaded when the server starts. The default is to call createEntityAnywhereFromDBID to create an entity to a minimally loaded baseapp. The entire process will be completed before the first started baseapp calls onBaseAppReady. The script layer can reimplement the entity creation method in a customized script (kbengine_defaults.xml->baseapp->entryScriptFile definition), for example: def onAutoLoadEntityCreate(entityType, dbid): KBEngine.createEntityFromDBID(entityType, dbid)
dbInterfaceName	string, optional parameter, specified by a database interface, uses the interface name "default" by default. The database interface is defined in kbengine_defaults.xml->dbmgr->databaseInterfaces.

Callback functions documentation

def onDestroy(*self*):

If this function is implemented in a script, it is called after **Entity**.**destroy**() destroys this entity. This function has no parameters.

def onEnterTrap(*self, entity, rangeXZ, rangeY, controllerID, userArg*):

When a scope trigger is registered using **Entity**.**addProximity** and another entity enters the trigger, this callback function is called.

parameters:

entity	Entity that has entered the area
rangeXZ	float, the size of the xz axis of the trigger, must be greater than or equal to zero.
rangeY	float, the size of the y-axis height of the trigger, must be greater than or equal to zero. It should be noted that for this parameter to take effect you must enable **kbengine_defaults.xml**->cellapp->coordinate_system->rangemgr_y Opening y-axis management will increase CPU consumption, because some games have a large number of entities at the same y-axis height or on the ground at nearly the same height. Because of this, the collision becomes very dense. 3D space games or small room-type games are more suitable for enabling this option.
controllerID	The controller id of this trigger.
userArg	The value of the parameter given by the user when calling **addProximity**, the user can decide how to use this parameter.

def onEnteredView(*self, entity*):

If this function is implemented in a script, when an entity enters the **View** scope of the current entity, this callback is triggered.

parameters:

entity The entity which has entered the View scope.

def onGetWitness(*self*):

If this function is implemented in a script, it is called when the entity has a **Witness** bound to it.
You can also access the entity property **Entity.hasWitness** to get the current state of the entity.

def onLeaveTrap(*self, entity, rangeXZ, rangeY, controllerID, userArg*):

If this function is implemented in a script, it is triggered when an entity leaves the trigger area registered by the current entity. The scope trigger is registered with **Entity**.**addProximity**.

parameters:

entity	The entity that has left the trigger area.
rangeXZ	float, the size of the xz axis of the trigger, must be greater than or equal to zero.
rangeY	float, the size of the y-axis height of the trigger, must be greater than or equal to zero. It should be noted that for this parameter to take effect you must enable **kbengine_defaults.xml**->cellapp->coordinate_system->rangemgr_y Opening y-axis management will increase CPU consumption, because some games have a large number of entities at the same y-axis height or on the ground at nearly the same height. Because of this, the collision becomes very dense. 3D space games or small room-type games are more suitable for enabling this option.
controllerID	The controller ID of this trigger.
userArg	The value of the parameter given by the user when calling **addProximity**, the user can decide how to use this parameter.

def onLoseControlledBy(*self, id*):

If this function is implemented in a script, this callback is triggered when this entity loses the **Entity**.**controlledBy** entity.

parameters:

id ID of the controlledBy entity.

def onLoseWitness(*self*):

If this function is implemented in a script, the callback is triggered whe this entity loses a **Witness**.
You can also access that **Entity.hasWitness** property to get the current state.

def onMove(*self, controllerID, userData*):

If this function is implemented in the script, the callback is invoked each frame when moved after a call to **Entity**.**moveToPoint**, **Entity**.**moveToEntity**, or **Entity**.**navigate**.

parameters:

controllerID The controller ID associated with the move.

userData The parameter given by the user when requesting to move the entity.

def onMoveOver(*self, controllerID, userData*):

If this callback function is implemented in a script, it is invoked after a call to **Entity**.**moveToPoint**, **Entity**.**moveToEntity**, or **Entity**.**navigate** when this entity reaches the target point.

parameters:

controllerID The controller ID associated with the move.

userData This parameter value is given by the user when requesting to move an entity.

def onMoveFailure(*self, controllerID, userData*):

If this function is implemented in the script, this callback is invoked after a call to **Entity**.**moveToPoint**, **Entity**.**moveToEntity**, or **Entity**.**navigate** if the movement has failed.

parameters:

controllerID The controller ID associated with the move.

userData This parameter value is given by the user when requesting to move an entity.

def onRestore(*self*):

If this callback function is implemented in a script, it is invoked when the Cell application crashes and recreates the entity on another cellapp. This function has no arguments.

def onSpaceGone(*self*):

If this callback function is implemented in the script, it will be called when the current entity's **Space** is destroyed. This function has no parameters.

def onTurn(*self, controllerID, userData*):

If this callback function is implemented in a script, it will be called after reaching the specified yaw. (related to **Entity**.**addYawRotator**)

parameters:

controllerID The controller ID returned by **Entity**.**addYawRotator**.

userData This parameter value is given by user when requesting to move an entity.

def onTeleport(*self*):

If this callback function is implemented in a script, it will be called at the moment before the (Real) entity is transmitted in the entity transfer that occurs through the baseapp's Entity.teleport call.
Note: Calling teleport on the entity's cell section does not trigger this callback, if you need this feature please invoke this callback after a call to **Entity**.**teleport**.

def onTeleportFailure(*self*):

If this callback function is implemented in a script, it will be called after a call to **Entity**.**teleport** if the teleport has failed.

def onTeleportSuccess(*self, nearbyEntity*):

If this callback function is implemented in a script, it is invoked after a succesful

call to **Entity.teleport**

parameters:

nearbyEntity This parameter is given by the user when calling **Entity.teleport**. This is a real entity.

def onTimer(*self, timerHandle, userData*):

Function description:

This function is called when a timer associated with this entity is triggered. A timer can be added using the **Entity.addTimer** function.

parameters:

timerHandle The ID of the timer.

userData integer, given by the user when calling **Entity.addTimer**.

def onUpdateBegin(*self*):

Invoked when a synchronization frame begins.

def onUpdateEnd(*self*):

Invoked after a synchronization frame has completed.

def onWitnessed(*self, isWitnessed*):

If this callback function is implemented in a script, it is called when this entity enters the View area of another entity bound to a Witness (also can be understood as when this entity is observed by a client). This function can be used to activate the entity's AI when it is observed, and stopping AI execution when the entity ceases to be observed, thus reducing CPU consumption of the server to increase efficiency.

parameters:

isWitnessed bool, True if the entity is observed and False when the entity is not observed. You can also access the entity property **Entity.isWitnessed** to get

the current state of the entity.

def onWriteToDB(*self*):

If this callback function is implemented in a script, it is called when the entity is about to be archived into the database.

Attributes documentation

allClients

By calling the entity's remote client methods through this attribute, the engine broadcasts the message to all other entities bound to a client that are within this entity's View area (including its own client, and the entity bound to the client is usually the player)

Example:
Avatar has player A, player B, and monster C in the View range.
avatar.allClients.attack(monsterID��skillID, damage)

At this point, the player himself, player A's, and player B's clients will all call the entity's attack method, and their client can invoke the specified skill's attack action to perform.

Other references:

Entity.**clientEntity**
Entity.**otherClients**

base

base is the entityCall used to contact the base **Entity**. This attribute is read-only and is None if the entity has no associated base **Entity**.

Other references:

Entity.**clientEntity**
Entity.**allClients**
Entity.**otherClients**

Type:

Read-only, **ENTITYCALL**

className

The class name of the entity.

Type:
Read-only, string

client

client is the entityCall used to contact associated client. This attribute is read-only, and is None if this entity does not have an associated client.

Other references:
Entity.clientEntity
Entity.allClients
Entity.otherClients

Type:
Read-only, **ENTITYCALL**

controlledBy

If this attribute is set to the BaseEntityCall of the server-side entity associated with a client, this entity is controlled by the corresponding client to move. If the attribute is None, the entity is moved by the server. When the client logs in and calls giveClientTo on this entity, this attribute is automatically set to its own BaseEntityCall.
Scripts can flexibly control the movement of the entity by the server or by the client (its own client or give control to other clients).

Other references:
Entity.onLoseControlledBy

Type:
BaseEntityCall

direction

This attribute describes the orientation of the **Entity** in world space. Users can change this attribute and the data will be synchronized to the client.
Example: self.direction.y = 1.0 self.direction.z = 1.0

Type:

Vector3, which contains (roll, pitch, yaw) in radians.

hasWitness

If this read-only attribute is True, it means that the entity has already bound a **Witness**. If the entity is bound to **Witness**, the client can obtain information from the entity's view scope. Otherwise, False.

Type:
Read-only, bool

id

id is the id of the **Entity** object. This id is an integer that is the same between base, cell, and client associated entities. This attribute is read-only.

Type:
Read-only, int32

isDestroyed

If this attribute is True, this **Entity** has already been destroyed.

Type:
Read-only, bool

isOnGround

If the value of this attribute is True, the **Entity** is on the ground, otherwise it is False.

Type:
Read-only, bool

isWitnessed

If the current entity is in the View scope of another entity bound to Witness (can also be understood as an entity observed by a client), this property is True, otherwise it is False.

Other references:

Entity.onWitnessed

Type:

Read-only, bool

layer

A space can load multiple navmesh data at the same time. Different navmesh can be in different layers. Different layers can be abstracted into the ground, the water surface, and so on. This attribute determines which layer an entity exists in.

Reference:
KBEngine.addSpaceGeometryMapping

Type:

int8

otherClients

By calling the entity's remote client methods through this property, the engine broadcasts the message to all other entities bound to the cliend within this entity's View scope (Not including its own client. The entity bound to the client is usually the player.).

Example:
avatar has player A, player B, and monster C in the View range.
avatar.otherClients.attack(monsterID, skillID, damage)

At this point, player A's and player B's client will call the entity attack method, and their client can invoke the specified skill's attack action to perform.

Other references:

Entity.clientEntity
Entity.otherClients

position

The coordinates of this entity in world space (x, y, z). This attribute can be changed by the user and will be synchronized to the client after the change. It is important to note that this attribute should not be referenced. Referencing this attribute is likely to incorrectly modify the real coordinates of the entity.

Example:

```
self.position.y = 10.0
```

If you want to copy this attribute value you can do the following:

```
import Math
self.copyPosition = Math.Vector3( self.position )
```

Type:
Vector3

spaceID

This attribute is the ID of the space in which the entity is located. The cell and client ids are the same.

Type:
Read-only, Integer

topSpeed

The maximum xz movement speed of the entity (m/s). This attribute is usually larger than the actualy movement speed. The server checks the client's movement legality through this attribute. If the movement distance excedes the speed limit, it is forced back to the previous position.

Other references:
Entity.topSpeedY

Type:
float

topSpeedY

The maximum y-axis movement speed of the entity (m/s). This attribute is usually larger than the actualy movement speed. The server checks the client's movement legality through this attribute. If the movement distance excedes the speed limit, it is forced back to the previous position.

Other references:

Entity.**topSpeed**

Type:

float

volatileInfo

This attribute specifies the **Entity**'s volatile data synchronization policy.
Volatile data includes the coordinate position of the entity and the orientation of the entity. Since volatile data is easily changed, the engine uses a set of optimized solutions to synchronize it to the client.
This attribute is four floats (position, yaw, pitch, roll) that represents the distance value, and the server synchronizes the relevant data to it when an entity reaches a close distance. If the distance value is larger than the View radius, it means that it is always synchronized

There is also a special bool attribute that is optimized. Its role is to control whether or not the server is optimized for synchronization. The current main optimization is the Y axis.
If true, the server does not synchronize the y-axis coordinates of the entity when some actions (e.g., navigate) cause the server to determine the entity is on the ground. This can save a lot of bandwidth when synchronizing a large number of entities. The default is true.

Users can also set the synchronization policies for different entities in .def:

```
<Volatile>
    <position/>           <!-- always synchronize -->
    <yaw/>                <!-- always synchronize -->
    <pitch>20</pitch>     <!-- synchronize within 20m or less -->
    <optimized> true </optimized>
```

```
</Volatile>                <!-- roll is always synchronized if not sp
```

Type:

sequence, four floats (float, float, float, float)

KBEngine

KBEngine module

This **KBEngine** module provides the Python script access to the **entity**'s base part, in particular it provides the registration and removal of timers, as well as the creation of **entities**.

Classes

Entity
Proxy

Member functions

def **addWatcher**(path, dataType, getFunction):
def **address**( ):
def **MemoryStream**( ):
def **charge**(ordersID, dbID, byteDatas, pycallback):
def **createEntity**( ):
def **createEntityAnywhere**(entityType, *params, callback):
def **createEntityRemotely**(entityType, baseMB, *params, callback):
def **createEntityFromDBID**(entityType, dbID, callback, dbInterfaceName):
def **createEntityAnywhereFromDBID**(entityType, dbID, callback, dbInterfaceName):
def **createEntityRemotelyFromDBID**(entityType, dbID, baseMB, callback, dbInterfaceName):
def **createEntityLocally**(entityType, *params):
def **debugTracing**( ):
def **delWatcher**(path):
def **deleteEntityByDBID**(entityType, dbID, callback, dbInterfaceName):
def **deregisterReadFileDescriptor**(fileDescriptor):
def **deregisterWriteFileDescriptor**(fileDescriptor):
def **executeRawDatabaseCommand**(command, callback, threadID, dbInterfaceName):
def **genUUID64**( ):
def **getResFullPath**(res):
def **getWatcher**(path):
def **getWatcherDir**(path):
def **getAppFlags**( ):
def **hasRes**(res):
def **isShuttingDown**( ):
def **listPathRes**(path, extension):
def **lookUpEntityByDBID**(entityType, dbID, callback, dbInterfaceName):
def **matchPath**(res):
def **open**(res, mode):

```
def publish( ):
def quantumPassedPercent( ):
def registerReadFileDescriptor( fileDescriptor, callback ):
def registerWriteFileDescriptor( fileDescriptor, callback ):
def reloadScript( fullReload ):
def scriptLogType( logType ):
def setAppFlags( flags ):
def time( ):
```

Callback functions

def **onBaseAppReady**(isBootstrap):
def **onBaseAppShutDown**(state):
def **onCellAppDeath**(addr):
def **onFini**( ):
def **onBaseAppData**(key, value):
def **onBaseAppDataDel**(key):
def **onGlobalData**(key, value):
def **onGlobalDataDel**(key):
def **onInit**(isReload):
def **onLoseChargeCB**(orderID, dbID, success, datas):
def **onReadyForLogin**(isBootstrap):
def **onReadyForShutDown**( ):
def **onAutoLoadEntityCreate**(entityType, dbID):

Attributes

LOG_ON_ACCEPT	
LOG_ON_REJECT	
LOG_ON_WAIT_FOR_DESTROY	
LOG_TYPE_DBG	
LOG_TYPE_ERR	
LOG_TYPE_INFO	
LOG_TYPE_NORMAL	
LOG_TYPE_WAR	
NEXT_ONLY	
component	Read-only **string**
entities	**Entities**
baseAppData	**GlobalDataClient**
globalData	**GlobalDataClient**

Member functions documentation

def addWatcher(*path, dataType, getFunction*):

Function description:
Interacts with the debug monitoring system, allowing the user to register a monitoring variable with the monitoring system.

Example:

```
>>> def countPlayers( ):
>>>     i = 0
>>>     for e in KBEngine.entities.values():
>>>             if e.__class__.__name__ == "Avatar":
>>>                     i += 1
>>>     return i
>>>
>>> KBEngine.addWatcher( "players", "UINT32", countPlayers )
```

This function adds a watch variable under the "scripts/players" watch path. The function countPlayers is called when the watcher observes.

parameters:

path	Create a monitored path.
dataType	The value type of the monitor variable. Reference: **Basic data types**
getFunction	This function is called when the observer retrieves the variable. This function returns a value representing a watch variable without arguments.

def address( ):

Function description:

Returns the address of the internal network interface.

def MemoryStream( ):

Function description:

Returns a new MemoryStream object.

The MemoryStreamobject stores binary information. This type is provided to allow the user to easily serialize and deserialize the Python base types following KBEngine underlying serialization rules.

For example, you can use this object to construct a network packet that KBEngine can parse.

Usage:

```
>>> s = KBEngine.MemoryStream()
>>> s
>>> b''
>>> s.append("UINT32", 1)
>>> s.pop("UINT32")
>>> 1
```

The types that MemoryStream currently supports are only basic data types.
Reference: **Basic data types**

def charge(*ordersID, dbID, byteDatas, pycallback*):

Function description:

Billing interface.

parameters:

ordersID	string, order ID.
dbID	uint64, the **databaseID** of the entity.
byteDatas	bytes, with data, which is parsed and defined by the developer.
pycallback	Billing callback. Billing callback prototype: (When calling KBEngine.chargeResponse in interfaces, the callback is called if an order is set to callback) def on*ChargeCB(self, orderID, dbID, success, datas):

ordersID: string, OrderID
dbID: uint64, usually the **databaseID** of the entity.
success: bool, whether the order succeeded datas: bytes, with data, parsed and defined by the developer.

def createEntity( ):

Function description:

KBEngine.createEntityLocally alias.

def createEntityAnywhere(*entityType, params, callback*):

Function description:

Create a new base **Entity**. The server can choose any **Baseapp** to create an **Entity**.

This method should be preferred over **KBEngine.createEntityLocally** so the server has the flexibility to choose a suitable **Baseapp** to create an entity.

The function parameters need to provide the type of entity created, and there is also a Python dictionary as a parameter to initialize the entities value.

The Python dictionary does not require the user to provide all of the properties, and the default values provided by the entity definition file ".def" are defaults.

Example:

```
params = {
        "name" : "kbe", # base, BASE_AND_CLIENT
        "HP" : 100,     # cell, ALL_CLIENT, in cellData
        "tmp" : "tmp"   # baseEntity.tmp
}

def onCreateEntityCallback(entity)
        print(entity)

createEntityAnywhere("Avatar", params, onCreateEntityCallback)
```

parameters:

entityType	string, specifies the type of **Entity** to create. Valid entity types are listed in **/scripts/entities.xml**.
params	optional parameter, a Python dictionary object. If a specified key is an **Entity** attribute, its value will be used to initialize the properties of this **Entity**. If the key is a **Cell** attribute, it will be added to the '**cellData**' attribute of the **Entity**. This **cellData**' attribute is a Python dictionary and will be used later to initialize the attributes of the cell entity.
callback	An optional callback function that is called when the entity is created. The callback function takes one argument, when the **Entity** is created successfully it is the entity's **entityCall**, on failure it is None.

returns:

Returns the **entityCall** of the **Entity** through the callback.

def createEntityRemotely(*entityType, baseMB, params, callback*):

Function description:

Create a new **Entity** on the specified baseapp through the baseMB parameter.

KBEngine.createEntityAnywhere should be preferred over this method to allow the server to decide which is the most suitable **Baseapp** to create the entity on for load balancing purposes.

The function parameters need to provide the type of the created entity, and there is also a Python dictionary as a parameter to initialize the entity's value.

This Python dictionary does not require the user to provide all of the properties, and the default values provided by the entity definition file ". def" are defaults.

Example:

```
params = {
        "name" : "kbe", # base, BASE_AND_CLIENT
        "HP" : 100,     # cell, ALL_CLIENT, in cellData
        "tmp" : "tmp"   # baseEntity.tmp
}
```

```
def onCreateEntityCallback(entity)
        print(entity)

createEntityRemotely("Avatar", baseEntityCall, params, onCreateEntit
```

parameters:

entityType	string, specifies the type of **Entity** to create. Valid entity types are listed in **/scripts/entities.xml**.
baseMB	BaseEntityCall which is a base **Entity** EntityCall. The entity will be created on the baseapp process corresponding to this entity.
params	Optional parameters, a Python dictionary object. If a specified key is an **Entity** attribute, its value will be used to initialize the properties of this **Entity**. If this key is a **Cell** attribute, it will be added to the **Entity**'s '**cellData**' attribute. This '**cellData**' attribute is a Python dictionary and will be used later to initialize the attributes of the cell entity.
callback	An optional callback function that is called when the entity is created. The callback takes one argument, on success it is an **Entity**实体的**entityCall**, on failure it is None.

returns:

Returns the **Entity**'s **entityCall** through the callback.

def createEntityFromDBID(*entityType, dbID, callback, dbInterfaceName*):

Function description:

Create an **Entity** by loading data from the database. The new **Entity** will be created on the **Baseapp** that called this function. If the **Entity** has been checked out from the database, a reference to this existing entity will be returned.

parameters:

entityType	string, specifies the **Entity** type to load. Valid entity types are listed in **/scripts/entities.xml**.
dbID	Specifies the database ID of the entity to create. The database ID of this entity is stored in the entity's **databaseID** attribute.
	An optional callback function that is called when the

callback	operation is complete. The callback function has three parameters: baseRef, databaseID, and wasActive. If the operation is successful, baseRef will be an **entityCall** or a direct reference to the newly created **Entity**. The databaseID will be the database ID of the entity. wasActive will be True if baseRef is a reference to an already existing entity (checked out from the database). If the operation fails the three parameters will be baseRef - None, databaseID - 0, wasActive - False. The most common reason for failure is that the entity does not exist in the database, but occasionally other errors such as timeouts or ID allocation failure.
dbInterfaceName	string, optional parameter, specified by a database interface, and the "default" interface is used by default. Database interfaces are defined in kbengine_defaults.xml->dbmgr->databaseInterfaces.

def createEntityAnywhereFromDBID(*entityType, dbID, callback, dbInterfaceName*):

Function description:

Create an **Entity** by loading data from the database. The server may choose any **Baseapp** to create the **Entity**.

Using this function will help BaseApps load balance.

If the entity has been checked out from the database, a reference to the existing **Entity** will be returned.

parameters:

entityType	string, specifies the **Entity** type to load. Valid entity types are listed in **/scripts/entities.xml**.
dbID	Specifies the database ID of the entity to create. The database ID of this entity is stored in the entity's **databaseID** attribute.
	An optional callback function that is called when the operation is complete. The callback function has three

callback	parameters: baseRef, databaseID, and wasActive. If the operation is successful, baseRef will be an **entityCall** or a direct reference to the newly created **Entity**. The databaseID will be the database ID of the entity. wasActive will be True if baseRef is a reference to an already existing entity (checked out from the database). If the operation fails the three parameters will be baseRef - None, databaseID - 0, wasActive - False. The most common reason for failure is that the entity does not exist in the database, but occasionally other errors such as timeouts or ID allocation failure.
dbInterfaceName	string, optional parameter, specified by a database interface, and the "default" interface is used by default. Database interfaces are defined in kbengine_defaults.xml->dbmgr->databaseInterfaces.

returns:

The **Entity**'s **entityCall** through the callback.

def createEntityRemotelyFromDBID(*entityType, dbID, baseMB, callback, dbInterfaceName*):

Function description:

Load data from the database and create an **Entity** on the baseapp specified via the baseMB parameter.

If the entity has been checked out from the database, a reference to the existing **Entity** will be returned.

parameters:

entityType	string, specifies the **Entity** type to load. Valid entity types are listed in **/scripts/entities.xml**.
dbID	Specifies the database ID of the entity to create. The database ID of this entity is stored in the entity's **databaseID** attribute.
	An optional callback function that is called when the operation is complete. The callback function has three

callback	parameters: baseRef, databaseID, and wasActive. If the operation is successful, baseRef will be an **entityCall** or a direct reference to the newly created **Entity**. The databaseID will be the database ID of the entity. wasActive will be True if baseRef is a reference to an already existing entity (checked out from the database). If the operation fails the three parameters will be baseRef - None, databaseID - 0, wasActive - False. The most common reason for failure is that the entity does not exist in the database, but occasionally other errors such as timeouts or ID allocation failure.
dbInterfaceName	string, optional parameter, specified by a database interface, and the "default" interface is used by default. Database interfaces are defined in kbengine_defaults.xml->dbmgr->databaseInterfaces.

returns:

Returns the **Entity**'s **entityCall** through the callback.

def createEntityLocally(*entityType, params*):

Function description:

Create a new **Entity**. The function parameters need to provide the type of the created entity, and there is also an optional Python dictionary as paramater to initialize the entity's values.

The Python dictionary does not require the user to provide all of the properties, and the default values provided by the entity definition file ".def" are defaults.

KBEngine.**createEntityAnywhere** should be preferred over this method to allow the server to decide which is the most suitable **Baseapp** to create the entity on for load balancing purposes.

It should be noted that this method returns the entity instantly without a callback, and is also guaranteed to return a direct reference to the **Entity** object, rather than its **EntityCall**. It is suitable to use this method over **KBEngine**.**createEntityAnywhere** when you need to manage the entities life

cycle (such as control when destroy is called on the entity) or access the entities attributes from the creating entity, because as described in the **EntityCall** documentation, it is not possible to access attributes or call methods not listed in the entity's def file using the **EntityCall**. This method is also necessary to use when you need a direct reference to an entity (as it's not possible to get one on a different baseapp). Many functions take an EntityCall as a parameter, but some require a direct reference to the entity (such as **Proxy**.**giveClientTo**).

Example:

```
params = {
        "name" : "kbe", # base, BASE_AND_CLIENT
        "HP" : 100,     # cell, ALL_CLIENT, in cellData
        "tmp" : "tmp"   # baseEntity.tmp
}

baseEntity = createEntityLocally("Avatar", params)
```

parameters:

entityType	string, specifies the type of entity to create. Valid entity types are listed in **/scripts/entities.xml**.
params	optional parameter, a Python dictionary object. If a specified key is an **Entity** attribute, its value will be used to initialize the properties of this **Entity**. If the key is a **Cell** attribute, it will be added to the '**cellData**' attribute of the **Entity**. This **cellData**' attribute is a Python dictionary and will be used later to initialize the attributes of the cell entity.

returns:

The newly created **Entity**.

def debugTracing( ):

Function description:

Outputs the Python extended object counter currently tracked by KBEngine.
Extended objects include: fixed dictionary, fixed array, Entity, EntityCall...
If the counter is not zero when the server is shut down normally, it means that the leak already exists and the log will output an error message.

ERROR cellapp [0x0000cd64] [2014-11-12 00:38:07,300] - PyGC::debugTracing(): FixedArray : leaked(128)
ERROR cellapp [0x0000cd64] [2014-11-12 00:38:07,300] - PyGC::debugTracing(): EntityCall : leaked(8)

def delWatcher(*path*):

Function description:

Interacts with the debug monitoring system, allowing users to delete monitored variables in the script.

parameters:

path The path to the variable to delete.

def deleteEntityByDBID(*entityType, dbID, callback, dbInterfaceName*):

Function description:

Deletes the specified entity (including the child table data generated by the attribute) from the database. If the entity is not checked out from the database, the deletion is successful. If the entity has been checked out from the database, **KBEngine** will fail to delete and return the **Entity**'s **entityCall** in the callback.

parameters:

entityType	string, specifies the type of Entity to delete. Valid entity types are listed in **/scripts/entities.xml**.
dbID	Specifies the database ID of the entity to delete. The database ID of the entity is stored in the entity's databaseID attribute.**databaseID**属性。
callback	An optional callback, with only one parameter. When the entity has not been checked out from the database it will be deleted successfully and the parameter will be True. If the entity has been checked out from the database then the parameter is the **Entity**'s **entityCall**.
dbInterfaceName	String, optional parameter, specifies a database interface. By default it uses the "default" interface. Database interfaces are defined by kbengine_defaults.xml->dbmgr->databaseInterfaces.

def deregisterReadFileDescriptor(*fileDescriptor*):

Function description:

Unregisters the callback registered with
KBEngine.registerReadFileDescriptor.

Example:
http://www.kbengine.org/assets/other/py/Poller.py

parameters:
fileDescriptor socket descriptor/file descriptor

def deregisterWriteFileDescriptor(*fileDescriptor*):

Function description:

Unregisters the callback registered with
KBEngine.registerWriteFileDescriptor.

Example:
http://www.kbengine.org/assets/other/py/Poller.py

parameters:
fileDescriptor socket descriptor/file descriptor.

def executeRawDatabaseCommand(*command, callback, threadID, dbInterfaceName*):

Function description:

This script function executes a database command on the database, which is directly parsed by the relevant database.

Please note that using this function to modify entity data may not be effective because if the entity has been checked out, the modified data will still be archived by the entity and cause overwriting.
This function is strongly not recommended for reading or modifying entity data.

parameters:

command

This database command will be different for different database configuration scenarios. For a MySQL database it is an SQL query.

callback

Optional parameter, callback object (for example, a function) called back with the command execution result. This callback has 4 parameters: result set, number of rows affected, auto value, error message.
Example:
def sqlcallback(result, rows, insertid, error): print(result, rows, insertid, error)

As the above example shows, the result parameter corresponds to the "result set", and the result set parameter is a row List. Each line is a list of strings containing field values. If the command execution does not return a result set (for example, a DELETE command), or the command execution encounters an error, the result set is None.

The rows parameter is the "number of rows affected", which is an integer indicating the number of rows affected by the command execution. This parameter is only relevant for commands that do not return results (such as DELETE).
This parameter is None if there is a result set return or if there is an error in the command execution.

The insertid is a long value, similar to an entity's databaseID. When successfully inserting data int a table with an auto long type field, it returns the data at the time of insertion.
More information can be found in mysql's mysql_insert_id() method. In addition, this parameter is only meaningful when the database type is mysql.

Error corresponds to the "error message", when the command execution encounters an error, this parameter is

	a string describing the error. This parameter is None when the command execution has not occurred.
threadID	int32, optional parameter, specifies a thread to process this command. Users can use this parameter to control the execution order of certain commands (dbmgr is multi-threaded). The default is not specified. If threadId is the ID of an entity, it will be added to the entity's archive queue and written by the thread one by one.
dbInterfaceName	string, optional parameter, specifies a database interface. By default it uses the "default" interface. Database interfaces are defined by kbengine_defaults.xml->dbmgr->databaseInterfaces.

def genUUID64( ):

Function description:

This function generates a 64-bit unique ID.
Note: This function depends on the baseapp server process startup parameter 'gus'. Please set the startup parameters to be unique.
In addition, if gus exceeds 65535, this function can only remain unique for the current process.

Usage:
Unique IDs can be generated on multiple service processes and do not conflict.
A room ID can be generated on multiple service processes and no uniqueness verification is required.

returns:

Returns a 64-bit integer.

def getResFullPath(*res*):

Function description:

Get the absolute path of a resource.
Note: Resource must be accessible under **KBE_RES_PATH**.

parameters:

res string, the relative path of the resource

returns:

string, if there is an absolute path to the given resource, otherwise returns null.

def getWatcher(*path*):

Function description:

Gets the value of a watch variable from the KBEngine debugging system.

Example: In the Python console of baseapp1:
>>>KBEngine.getWatcher("/root/stats/runningTime")
12673648533

>>>KBEngine.getWatcher("/root/scripts/players")
32133

parameters:

path string, the absolute path of the variable including the variable name (can be viewed on the GUIConsole watcher page).

returns:

The value of the variable.

def getWatcherDir(*path*):

Function description:

Get a list of elements (directories, variable names) under the watch directory from the KBEngine debugging system.

Example: In the Python console of baseapp1 enter:
>>>KBEngine.getWatcher("/root")
('stats', 'objectPools', 'network', 'syspaths', 'ThreadPool', 'cprofiles', 'scripts', 'numProxices', 'componentID', 'componentType', 'uid', 'numClients', 'globalOrder', 'username', 'load', 'gametime', 'entitiesSize', 'groupOrder')

parameters:

path string, the absolute path to this variable (can be viewd on the GUIConsole watcher page).

returns:

Monitors the list of elements in the directory (directory, variable name).

def getAppFlags( ):

Function description:

Get the flags of the current engine APP, Reference: **KBEngine.setAppFlags**.

returns:

KBEngine.APP_FLAGS_*

def hasRes(*res*):

Function description:

Use this interface to determine if a relative path exists.
Note: Resource must be accessible under **KBE_RES_PATH**.

Example:

>>>KBEngine.hasRes("scripts/entities.xml")
True

parameters:

res string, the relative path of the resource

returns:

bool, True if relative path exists, otherwise False.

def isShuttingDown( ):

Function description:

Returns whether the server is shutting down.
After the **onBaseAppShutDown(state=0)** is called, this function returns True.

returns:
True if the server is shutting down, otherwise False.

def listPathRes(*path, extension*):

Function description:

Get a list of resources in a resource directory
Note: Resources must be accessible under **KBE_RES_PATH**.

Example:

>>>KBEngine.listPathRes("scripts/cell/interfaces")
('/home/kbe/kbengine/demo/res/scripts/cell/interfaces/AI.py',
'/home/kbe/kbengine/demo/res/scripts/cell/interfaces/New Text Document.txt')

>>>KBEngine.listPathRes("scripts/cell/interfaces", "txt")
('/home/kbe/kbengine/demo/res/scripts/cell/interfaces/New Text Document.txt')

>>>KBEngine.listPathRes("scripts/cell/interfaces", "txt|py")
('/home/kbe/kbengine/demo/res/scripts/cell/interfaces/AI.py',
'/home/kbe/kbengine/demo/res/scripts/cell/interfaces/New Text Document.txt')

>>>KBEngine.listPathRes("scripts/cell/interfaces", ("txt", "py"))
('/home/kbe/kbengine/demo/res/scripts/cell/interfaces/AI.py',
'/home/kbe/kbengine/demo/res/scripts/cell/interfaces/New Text Document.txt')

parameters:
res string, the relative path of the resource directory
extension string, optional parameter, file extension to filter by

returns:
Tuple, resource list.

def lookUpEntityByDBID(*entityType, dbID, callback, dbInterfaceName*):

Function description:

Queries whether an entity is checked out of the database, and if the entity has been checked out of the database, **KBEngine** will return the **Entity**'s **entityCall** in the callback.

parameters:

entityType	string, specifies the type of Entity to query. Valid entity types are listed in **/scripts/entities.xml**.
dbID	Specifies the database ID of the **Entity** to be queried. The database ID is stored in the entity's databaseID attribute.
callback	A callback with one parameter, True when the entity is not checked out from the database, if it is checked out then it is the **Entity**'s **entityCall**. False in any other case.
dbInterfaceName	string, optional parameter, specifies a database interface. Uses the "default" interface by default. Database interfaces are defined in kbengine_defaults.xml->dbmgr->databaseInterfaces.

def matchPath(*res*):

Function description:

Get the absolute path of a resource from its relative path.
Note: Resources must be accessible under **KBE_RES_PATH**.

Examples:

>>>KBEngine.matchPath("scripts/entities.xml")
'/home/kbe/kbengine/demo/res/scripts/**entities.xml**'

parameters:

res string, the relative path of the resource (including its name).

returns:

string, the absolute path of the resource.

def open(*res, mode*):

Function description:

Use this function to open resources with their relative paths.
Note: Resource must be accessible under **KBE_RES_PATH**.

parameters:

res string, the relative path of the resource.

mode string, file operation mode:
w Open in write mode,
a Open in append mode (Start from EOF, create new file if necessary)
r+ Open
w+ in read/write mode Open in read/write mode (see w)
a+ Open in read/write mode (See a)
rb Opens
wb in binary read mode Opens in binary write mode (see w)
ab Opens in binary append mode (see a)
rb+ Opens in binary read and write mode (see r+)
wb+ Opens in binary read and write mode (see w+)
ab+ opens in binary read/write mode (see a+)

def publish( ):

Function description:

This function returns the server's current release mode.

returns:

int8, 0: debug, 1: release, others can be customized.

def quantumPassedPercent( ):

Function description:

Returns the percentage of the current tick that takes one clock cycle.

returns:

Returns the percentage of the current tick that takes one clock cycle.

def registerReadFileDescriptor(*fileDescriptor, callback*):

Function description:

Registers a callback function that is called when the file descriptor is readable.

Example:
http://www.kbengine.org/assets/other/py/Poller.py

parameters:

fileDescriptor	socket descriptor/file descriptor.
callback	A callback function with the socket descriptor/file descriptor as its only parameter.

def registerWriteFileDescriptor(*fileDescriptor, callback*):

Function description:

Registers a callback function that is called when the socket descriptor/file descriptor is writable.

Example:
http://www.kbengine.org/assets/other/py/Poller.py

parameters:

fileDescriptor	socket descriptor/file descriptor
callback	A callback function with the socket descriptor/file descriptor as its only parameter.

def reloadScript(*fullReload*):

Function description:

Reloads Python modules related to entity and custom data types. The current entity's class is set to the newly loaded class. This method should only be used for development mode and not for product mode. The following points should be noted:

1) The overloaded script can only be executed on **Baseapp**, and the user should ensure that all server components are loaded.

2) The custom type should ensure that the objects already instantiated in memory are updated after the script is reloaded. Here is an example:

```
for e in KBEngine.entities.values():
   if type( e ) is Avatar.Avatar:
      e.customData.__class__ = CustomClass
```

When this mehod completes, **KBEngine.onInit**(True) is called.

parameters:

fullReload bool, optional parameter that specifies whether to reload entity definitions at the same time. If this parameter is False, the entity definition will not be reloaded. The default is True.

returns:

True if the reload succeeds, otherwise False.

def scriptLogType(*logType*):

Function description:

Set the type of information output by the current Python.print (Reference: KBEngine.LOG_TYPE_*).

def setAppFlags(*flags*):

Function description:

Set the flags of the current engine APP.

KBEngine.APP_FLAGS_NONE // Default (not set)
KBEngine.APP_FLAGS_NOT_PARTCIPATING_LOAD_BALANCING //Do not participate in load balancing

Example:
KBEngine.setAppFlags(KBEngine.APP_FLAGS_NOT_PARTCIPATING_LOAI | KBEngine.APP_FLAGS_*)

def time( ):

Function description:

This method returns the current game time (number of cycles).

returns:

uint32, the time of the current game. This refers to the number of cycles. The period is affected by the frequency. The frequency is determined by the configuration file **kbengine.xml** or **kbengine_defaults.xml**->gameUpdateHertz.

Callback functions documentation

def onBaseAppReady(*isBootstrap*):

Function description:

This callback function is called when the current **Baseapp** process is ready.
Note: This callback function must be implemented in the portal module (**kbengine_defaults.xml**->entryScriptFile).

parameters:

isBootstrap bool, True if this is the first **Baseapp** started

def onBaseAppShutDown(*state*):

Function description:

The **Baseapp** shutdown procedure will call this function.
Note: This callback function must be implemented in the portal module (**kbengine_defaults.xml**->entryScriptFile).

parameters:

state If state is 0, it means that it is before all clients are disconnected, if state is 1, it means that it is before all entities are written to the database, if state is 2, it mean all entities have been written to the database.

def onCellAppDeath(*addr*):

Function description:

This callback function will be called on the death of a cellapp.
Note: This callback function must be implemented in the portal module (**kbengine_defaults.xml**->entryScriptFile).

parameters:

addr Dead cellapp address.
tuple:(ip, port) Network byte order

def onFini( ):

Function description:

This callback function is called after the engine is officially shut down.
Note: This callback function must be implemented in the portal module (**kbengine_defaults.xml**->entryScriptFile).

def onBaseAppData(*key, value*):

Function description:

This function is called back when KBEngine.baseAppData changes.
Note: This callback function must be implemented in the portal module (**kbengine_defaults.xml**->entryScriptFile).

parameters:

key The key of the changed data.

value The value of the changed data.

def onBaseAppDataDel(*key*):

Function description:

This function is called back when KBEngine.baseAppData is deleted.
Note: This callback function must be implemented in the portal module (**kbengine_defaults.xml**->entryScriptFile).

parameters:

key Deleted data key

def onGlobalData(*key, value*):

Function description:

This function is called back when KBEngine.globalData changes.
Note: This callback function must be implemented in the portal module (**kbengine_defaults.xml**->entryScriptFile).

parameters:

key The key of the changed data

value The value of the changed data

def onGlobalDataDel(*key*):

Function description:

This function is called back when KBEngine.globalData is deleted.
Note: This callback function must be implemented in the portal module (**kbengine_defaults.xml**->entryScriptFile).

parameters:

key Deleted data key.

def onInit(*isReload*):

Function description:

This function is called back after all scripts have been initialized after the engine started.
Note: This callback function must be implemented in the portal module (**kbengine_defaults.xml**->entryScriptFile).

parameters:

isReload bool, whether it was triggered after rewriting the loading script.

def onLoseChargeCB(*orderID, dbID, success, datas*):

Function description:

This function is called back when **KBEngine.chargeResponse** is called in and the order is lost or unknown.
Note: This callback function must be implemented in the portal module (**kbengine_defaults.xml**->entryScriptFile).

parameters:

ordersID string, order ID.

dbID uint64, the database ID of the entity, see: **Entity.databaseID**.

success bool, is it successful?
datas bytes, with information

def onReadyForLogin(*isBootstrap*):

Function description:

When the engine is started and initialized, it will always call this function to ask whether the script layer is ready. If the script layer is ready, loginapp allows the client to log in.
Note: This callback function must be implemented in the portal module (**kbengine_defaults.xml**->entryScriptFile).

parameters:
isBootstrap bool, True if this is the first **Baseapp** started.

returns:
If the return value is greater than or equal to 1.0, the script layer is ready; otherwise, return a value from 0 to less than 1.0.

def onReadyForShutDown( ):

Function description:

If this callback function is implemented in the script, it is called when the process is ready to exit.

You can use this callback to control when the process exits.

Note: This callback function must be implemented in the portal module (**kbengine_defaults.xml**->entryScriptFile).

returns:
bool if it returns True, it allows the process to exit. Returning other values will cause the process to ask again after a period of time.

def onAutoLoadEntityCreate(*entityType, dbID*):

Function description:

Called when an automatically loaded entity is created. If the script layer implements this callback, the entity is created by the script layer, otherwise the engine defaults to create the entity using createEntityAnywhereFromDBID.

This callback is envoked because **Entity.writeToDB** was set to automatically load the entity.
Note: this callback takes precedence over onBaseAppReady execution and can be checked for onBaseAppReady when the entity is loaded.

parameters:

entityType	string, specifies the type of entity to query. Valid entity types are listed in **/scripts/entities.xml**.
dbID	specifies the database ID of the **Entity** to be queried. The database ID of this entity is stored in the entity's **databaseID** attribute.

Attributes documentation

LOG_ON_ACCEPT

Description:

This constant is returned by **Proxy**.**onLogOnAttempt**, and means that the new client is allowed to bind to a **Proxy entity**.
If the **Proxy entity** already has a client binding, the previous client will be kicked out.

LOG_ON_REJECT

Description:

This constant is returned by **Proxy**.**onLogOnAttempt**, which means that the current client is bound to the **Proxy entity.**

LOG_ON_WAIT_FOR_DESTROY

Description:

This constant is returned by **Proxy**.**onLogOnAttempt**. The current requesting client will wait until the **Proxy entity** is completely destroyed and the underlying layer will complete the subsequent binding process. Before this returns, **Proxy**.**destroy** or **Proxy**.**destroyCellEntity** should be invoked.

LOG_TYPE_DBG

Description:

The log output type is debug.
Set by **scriptLogType**.

LOG_TYPE_ERR

Description:

The log output type is error.

Set by**scriptLogType**.

LOG_TYPE_INFO

Description:

The log output type is general information.
Set by **scriptLogType**.

LOG_TYPE_NORMAL

Description:

The log output type is normal.
Set by **scriptLogType**.

LOG_TYPE_WAR

Description:

The log output type is warning.
Set by **scriptLogType**.

NEXT_ONLY

Description:

This constant is used for the **Entity.shouldAutoBackup** and **Entity.shouldAutoArchive** attributes and means that the entity is backed up automatically next time it is deemed acceptable, and then the attribute is automatically set to false (0).

component

Description:

This is the component that is running in the current Python environment. (So far) Possible values are 'cell', 'base', 'client', 'database', 'bot', and 'editor'.

entities

Description:

entities is a dictionary object that contains all the entities in the current process. Debugging leaked entities (instances that call destroy without releasing memory, usually due to being referenced):

```
>>> KBEngine.entities.garbages.items()
[(1025, Avatar object at 0x7f92431ceae8.)]


>>> e = _[0][1]
>>> import gc
>>> gc.get_referents(e)
[{'spacesIsOk': True, 'bootstrapIdx': 1}, ]
```

Debugging a leaked KBEngine-encapsulated Python object:
KBEngine.debugTracing

Types:
Entities

baseAppData

Description:

This attribute contains a dictionary-like object that is automatically synchronized across all BaseApps. When a value in the dictionary is modified, the change is broadcast to all BaseApps.

Example:

```
KBEngine.baseAppData[ "hello" ] = "there"
```

The rest of the BaseApps can access the following:

```
print KBEngine.baseAppData[ "hello" ]
```

Keys and values can be of any type, but these types must be encapsulated and unpacked on all target components.

When a value is changed or deleted, a callback function is called on all components. See: **KBEngine.onBaseAppData** and **KBEngine.onDelBaseAppData**.

Note: Only top-level value changes will be broadcast. If you have a value (such as a list) that changes an internal value (such as just changing a number), this information will not be broadcast.

Do not do the following:

```
KBEngine.baseAppData[ "list" ] = [1, 2, 3]
KBEngine.baseAppData[ "list" ][1] = 7
```

The local access is [1, 7, 3] and the remote access is [1, 2, 3].

globalData

Description:

This attribute contains a dictionary-like object that is automatically synchronized across all BaseApps and CellApps. When a value in the dictionary is modified, the change is broadcast to all BaseApps and CellApps.

example:

```
KBEngine.globalData[ "hello" ] = "there"
```

The other **Baseapps** and **Cellapps** can access the following:

```
print KBEngine.globalData[ "hello" ]
```

Keys and values can be of any type, but these types must be encapsulated and unpacked on all target components.

When a value is changed or deleted, a callback function is called on all components. See: **KBEngine.onGlobalData** and **KBEngine.onGlobalDataDel**.

Note: Only top-level value changes will be broadcast. If you have a value (such as a list) that changes an internal value (such as just changing a number), this information will not be broadcast.

Do not do the following:

```
KBEngine.globalData[ "list" ] = [1, 2, 3]
KBEngine.globalData[ "list" ][1] = 7
```

The local access is [1, 7, 3] and the remote access is [1, 2, 3].

KBEngine

Entity class

[KBEngine module]

Entity is part of the **KBEngine** module. **More...**

```
import KBEngine
```

Member functions

def **addTimer**(self, initialOffset, repeatOffset=0, userArg=0):
def **createCellEntity**(self, cellEntityMB):
def **createCellEntityInNewSpace**(self, cellappIndex):
def **delTimer**(self, id):
def **destroy**(self, deleteFromDB, writeToDB):
def **destroyCellEntity**(self):
def **teleport**(self, baseEntityMB):
def **writeToDB**(self, callback, shouldAutoLoad, dbInterfaceName):

Callback functions

def **onCreateCellFailure**(self):
def **onDestroy**(self):
def **onGetCell**(self):
def **onLoseCell**(self):
def **onPreArchive**(self):
def **onRestore**(self):
def **onTimer**(self, timerHandle, userData):
def **onWriteToDB**(self, cellData):

Attributes

cell	Read-only **CellEntityCall**
cellData	**CELLDATADICT**
className	Read-only **string**
client	Read-only **ClientEntityCall**
databaseID	Read-only **int64**
databaseInterfaceName	Read-only **string**
id	Read-only **int32**
isDestroyed	**bool**
shouldAutoArchive	True, False or **KBEngine**.**NEXT_ONLY**
shouldAutoBackup	True, False or **KBEngine**.**NEXT_ONLY**

A detailed description

This **Entity** class represents an **Entity** that resides on the **Baseapp**. **Entities** can be created using the **KBEngine**.**createEntity** function (as well as functions prefixed by createEntity). An **Entity** can also be remotely created using the **Cellapp** function **KBEngine**.**createEntityOnBaseApp**.

A base **Entity** can be linked to an active **cell** entity and can be used to create an associated **cell** entity. This class allows you to create and destroy **cell** entities, register a timer on the base entity, or access the contact information for this object.
You can also access a **CellEntityCall** through which this base entity can communicate with its **cell** entity (the associated **cell** entity can be moved to a different **cell** when KBEngine performs load balancing as a result of the movement of the **cell** entity).

Member function documentation

def addTimer(*self, initialOffset, repeatOffset=0, userArg=0*):

Function description:

Register a timer. The timer is triggered by the callback function ***onTimer***, which will be executed the first time after "initialOffset" seconds, and then executed once every "repeatOffset" seconds. A "userArg" parameter can be set (integer only).

The ***onTimer*** function must be defined in the base part of the entity with two parameters. The first is an integer, the timer id (which can be used to remove the timer's "**delTimer**" function), and the second is the user parameter "userArg".

Example:

```
# Here is an example of using addTimer
import KBEngine

class MyBaseEntity( KBEngine.Entity ):

    def __init__( self ):
        KBEngine.Entity.__init__( self )

        # Add a timer, trigger for the first time after 5 seconds, a
        self.addTimer( 5, 1, 9 )

        # Add a timer and execute it once after 1 second. The defaul
        self.addTimer( 1 )

    # Entity timer callback "onTimer" is called
    def onTimer( self, id, userArg ):
        print "MyBaseEntity.onTimer called: id %i, userArg: %i" % (
        # If this is a repeated timer, when the timer is no longer n
        #     self.delTimer( id )
```

parameters:

initialOffset	float, specifies the time interval in seconds for the timer to trigger the first callback.
	float, specifies the time interval (in seconds) after each

repeatOffset execution of the first callback execution. You must remove the timer with the function **delTimer**, otherwise it will continue to repeat. Values less than or equal to 0 will be ignored.

userArg integer, specifies the value of the userArg parameter when invoking the "**onTimer**" callback.

returns:Z

integer, the internal id of the timer. This id can be used to remove the timer using **delTimer**.

def createCellEntity(*self, cellEntityMB*):

Function description:

Requests to create an associated entity in a **cell**.

The information used to create the **cell** entity is stored in the entity's cellData attribute. The cellData attribute is a dictionary that corresponds to the default value in the entity's .def file, as well as the "position", "direction", and "spaceID" used to represent the entity's position and orientation (roll, pitch, yaw).

parameters:

cellEntityMB **CellEntityCall** parameter that specifies which space to create this cell entity in.

Only a direct **CellEntityCall** may be used. If you have an entity **BaseEntityCall**, you cannot pass its *baseEntityCall.cell* to this function. Instead, you must create a new function on the current entity's base that accepts a direct **CellEntityCall** as a parameter and then calls this function using it.

E.g.

```
baseEntityCallOfNearbyEntity.createCellNearSelf( self
```

On the nearby entity's base:

```
def createCellNearSelf( self, baseEntityCall ):
    baseEntityCall.createCellNearHere( self.cell )
```

On the current entity's base:

```
def createCellNearHere( self, cellEntityCall ):
    self.createCellEntity( cellEntityCall )
```

def createCellEntityInNewSpace(*self, cellappIndex*):

Function description:

Create a space on the cellapp and create the **cell** of this entity into the new space. It requests to complete through cellappmgr.

The information used to create the cell entity is stored in the entity's cellData attribute. This property is a dictionary. The default values in the corresponding entity's .def file also include "position", "direction", and "spaceID" for representing the entity's position and orientation (roll, pitch, yaw).

parameters:

cellappIndex integer, if it is either None or 0, a cellapp is dynamically selected engine load balancer. If it is greater than 0, a space is created in t specified cellapp
Example: If you expect to open four cellapps, then the cellappIn to specify the index can be
1, 2, 3, 4,
if the actual running cellapp is less than 4, for example, only 3, t cellappIndex input 4 due to the number of overflow 4 1, 5 2.

Tip: This feature can be used in conjunction with KBEngine.setA (KBEngine.APP_FLAGS_NOT_PARTCIPATING_LOAD_BAL for example: placing large map spaces in several fixed cellapps a these cellapps to not participate in load balancing, and other cell place copy space. When the copyspace is created and the cellapp to 0 or None, the consumption of the copy map will not affect th process, thus ensuring the smoothness of the main scene.

def delTimer(*self, id*):

Function description:

The function delTimer is used to remove a registered timer. The removed timer is no longer executed. Single-shot timers are automatically removed after the callback is executed, and it is not necessary to use the delTimer to remove it. If the delTimer function uses an invalid id (for example, has been removed), it will generate an error.

A usage example is with the **Entity**.**addTimer** function.

parameters:

id integer, which specifies the timer id to remove.

def destroy(*self, deleteFromDB, writeToDB*):

Function description:

This function destroys the base parts of the entity. If the entity has a cell part, then the user must first destroy the cell part, otherwise it will generate an error. To destroy the cell part of the entity, call a **Entity**.**destroyCellEntity**.

It may be more appropriate to call self.destroy in the onLoseCell callback. This ensures that the base part of the entity is destroyed.

parameters:

deleteFromDB	If True, the entry associated with this entity in the database will be deleted. This parameter defaults to False.
writeToDB	If True, the archived attributes associated with this entity will be written to the database. Only if this entity is read for the database or uses **Entity**.**writeToDB** will it be written to the database. This parameter is True by default, but will be ignore when deleteFromDB is True.

def destroyCellEntity(*self*):

Function description:

destroyCellEntity requests destruction of the associated cell entity. This method will generate an error if there is no associated cell entity.

def teleport(*self, baseEntityMB*):

Function description:

teleport will teleport the cell part of this entity to the space where the entity specified by the parameter is located.

After arriving at the new space, Entity.onTeleportSuccess is called. This can be used to move the entity to a suitable location in the new space.

parameters:

baseEntityMB	The **EntityCall** of the entity that is in the space this entity will be teleported. When successful, the cell entity associated with this parameter is passed to the Entity.onTeleportSuccess function.

def writeToDB(*self, callback, shouldAutoLoad, dbInterfaceName*):

Function description:

This function saves the entity's archive attributes to the database so that it can be loaded again when needed.

Entities can also be marked as automatically loaded so that the entity will be re-created when the service is started.

parameters:

callback	This optional parameter is a callback function when the database operation is complete. It has two parameters. The first is a success or failure boolean flag, and the second is the base entity.
shouldAutoLoad	This optional parameter specifies whether this entity needs to be loaded from the database when the service is started. Note: The entity is automatically loaded when the server starts. The default is to call createEntityAnywhereFromDBID to create an entity to a minimally loaded baseapp. The entire process will be completed before the first started baseapp calls onBaseAppReady. The script layer can reimplement the entity creation

	method in the personalization script (kbengine_defaults.xml->baseapp->entryScriptFile definition), for example: def onAutoLoadEntityCreate(entityType, dbid): KBEngine.createEntityFromDBID(entityType, dbid)
dbInterfaceName	string, optional parameter, specified by a database interface, default is to use the "default" interface. Database interfaces are defined in kbengine_defaults.xml->dbmgr->databaseInterfaces.

Callback functions documentation

def onCreateCellFailure(*self*):

Function description:

If this function is implemented in the script, this function is called when the cell entity fails to create. This function has no parameters.

def onDestroy(*self*):

Function description:

If this callback function is implemented in a script, it is called after **Entity**.**destroy**() actually destroys the entity. This function has no parameters.

def onGetCell(*self*):

Function description:

If this function is implemented in the script, this function is called when it gets a cell entity. This function has no parameters.

def onLoseCell(*self*):

Function description:

If this function is implemented in the script, this function is called after its associated cell entity is destroyed. This function has no parameters.

def onPreArchive(*self*):

Function description:

If this function is implemented in a script, it is called before the entity is automatically written to the database. This callback is called before the **Entity**.**onWriteToDB** callback. If the callback returns False, the archive operation is aborted. This callback should return True to continue the operation. If this callback does not exist, the archiving operation continues.

def onRestore(*self*):

Function description:

If this function is implemented in a script, it is called when this **Entity**'s application crashes and the **Entity** is recreated on other applications. This function has no parameters.

def onTimer(*self, timerHandle, userData*):

Function description:

This function is called when a timer associated with this entity is triggered. A timer can be added using the **Entity**.**addTimer** function.

parameters:

timerHandle The id of the timer.

userData integer, User data passed in on **Entity**.**addTimer**.

def onWriteToDB(*self, cellData*):

Function description:

If this function is implemented in the script, this function is called when the entity data is to be written into the database.

Note that calling writeToDB in this callback will result in an infinite loop.

parameters:

cellData Contains the cell properties that will be stored in the database. **cellData** is a dictionary.

Attributes documentation

cell

Description:

cell is the **ENTITYCALL** used to contact the cell entity. This property is read-only, and the property is set to None if this base entity has no associated cell.

Type:

Read-only **ENTITYCALL**

cellData

Description:

cellData is a dictionary property. Whenever the base entity does not create its cell entity, the attributes of the cell entity are stored here.

If the cell entity is created, these used values and **cellData** attributes will be deleted. In addition to the attributes that the cell entity specifies in the entity definition file, it also contains position, direction, and spaceID.

Type:

CELLDATADICT

className

Description:

The class name of the entity.

Type:

Read-only string

client

Description:

client is the EntityCall used to contact the client. This attribute is read-only and is set to None if this base entity has no associated client.

Type:
Read-only **ENTITYCALL**

databaseID

Description:

databaseID is the entity's permanent ID (database id). This id is of type uint64 and is greater than 0. If it is 0 then the entity is not permanent.

Type:
Read-only int64

databaseInterfaceName

Description:

databaseInterfaceName is the database interface name where the entity persists. The interface name is configured in kbengine_defaults->dbmgr. The entity must be persistent (databaseID>0) for this attribute to be available, otherwise an empty string is returned.

Type:
Read-only string

id

Description:

id is the object id of the entity. This id is an integer that is the same between base, cell, and client associated entities. This attribute is read-only.

Type:
Read-only int32

isDestroyed

Description:

This attribute is True if the **Entity** has been destroyed.

Type:
bool

shouldAutoArchive

Description:

This attribute determines the automatic archiving strategy. If set to True, AutoArchive will be available, if set to False AutoArchive will not be available. If set to **KBEngine**.**NEXT_ONLY**, automatic archiving will be available at the next scheduled time. This attribute will be set to false after the next archiving.

Type:
True, False or **KBEngine**.**NEXT_ONLY**

shouldAutoBackup

Description:

This attribute determines the automatic bacup strategy. If set to True, automatic backup will be available, if set to False, automatic backup will not be available. If set to **KBEngine**.**NEXT_ONLY**, automatic backup will be done at the next available predetermined time. After the next backup, this attribute will be set to False.

Type:
True, False or **KBEngine**.**NEXT_ONLY**

KBEngine

Proxy class

[KBEngine module]

Proxy is part of the **KBEngine** module. **More...**

```
import KBEngine
```

Parent

Entity

Member function

def **disconnect**(self):
def **getClientType**(self):
def **getClientDatas**(self):
def **giveClientTo**(self, proxy):
def **streamFileToClient**(self, resourceName, desc="", id=-1):
def **streamStringToClient**(self, data, desc="", id=-1):

Callbacks

def **onClientDeath**(self):
def **onClientGetCell**(self):
def **onClientEnabled**(self):
def **onGiveClientToFailure**(self):
def **onLogOnAttempt**(self, ip, port, password):
def **onStreamComplete**(self, id, success):

Attributes

__ACCOUNT_NAME__	Read-only **string**
__ACCOUNT_PASSWORD__	Read-only **string**
clientAddr	Read-only
clientEnabled	Read-only **bool**
hasClient	Read-only **bool**
roundTripTime	Read-only
timeSinceHeardFromClient	Read-only

A detailed description

A **Proxy** is a special type of **Entity**. It inherits from **Entity** and has an associated client. By itself, it is a proxy client entity that handles all server-to-client updates. Cannot create **Proxy** class objects directly script.

Member functions documentation

def disconnect(*self*):

Disconnect the client.

def getClientType(*self*):

Function description:

This function returns the client type.

returns:

UNKNOWN_CLIENT_COMPONENT_TYPE = 0,
CLIENT_TYPE_MOBILE = 1, // Mobile phone
CLIENT_TYPE_WIN = 2, // PC, typically EXE clients
CLIENT_TYPE_LINUX = 3 // Linux Application program
CLIENT_TYPE_MAC = 4 // Mac Application program
CLIENT_TYPE_BROWSER = 5, // Web applications, HTML5, Flash
CLIENT_TYPE_BOTS = 6, // bots
CLIENT_TYPE_MINI = 7, // Mini-Client
CLIENT_TYPE_END = 8 // end

def getClientDatas(*self*):

Function description:

This function returns the data attached to the client when logging in and registering.
This data can be used to expand the operating system. If a third-party account service is connected, this data is sent to the third-party service system through the interfaces process.

returns:

tuple, a tuple of 2 elements (login data bytes, registration data bytes), the first element is the datas parameter passed in when the client invokes the login, and the second element is passed in when the client registers. Since they can store arbitrary binary data, they all exist as bytes.

def giveClientTo(*self, proxy*):

Function description:

The client's controller is transferred to another Proxy, the current Proxy must have a client and the target Proxy must have no associated client, otherwise it will cause an error.

See also:

Proxy.onGiveClientToFailure

parameters:

proxy Control will be transferred to this entity.

def streamFileToClient(*self, resourceName, desc="", id=-1*):

Function description:

This function is similar to **streamStringToClient()** and sends a resource file to the client. The sending process operates on different threads so it does not compromise the main thread.

See also:

Proxy.onStreamComplete

parameters:

resourceName	The name of the resource to send, including the path.
desc	An optional string that describes the resource sent to the client.
id	A 16-bit id whose value depends entirely on the caller. If the incoming -1 system will select an unused id in the queue. The client can make resource judgments based on this id.

returns:

The id associated with this download.

def streamStringToClient(*self, data, desc="", id=-1*):

Function description:

Sends some data to the client bound to the current entity. If the client port data is cleared, this function can only be called when the client binds to the entity again. The 16-bit id is entirely up to the caller.
If the caller does not specify this ID then the system will allocate an unused id. The client can make resource judgments based on this id.

You can define a callback function (onStreamComplete) in a Proxy-derived class. This callback function is called when all data is successfully sent to the client or when the download fails.

See also: **Proxy**.**onStreamComplete**, client **Entity**.**onStreamDataStarted**, **Entity**.**onStreamDataRecv**, and **Entity**.**onStreamDataCompleted**.

parameters:

data The string to send

desc An optional description string sent.

id A 16-bit id whose value depends entirely on the caller. If the incoming -1 system will select an unused id in the queue.

returns:

The id associated with this download.

Callback functions documentation

def onClientDeath(*self*):

If this callback is implemented in a script, this method will be called when the client disconnects. This method has no parameters.

def onClientGetCell(*self*):

If this callback is implemented in a script, the callback is called when the client can call the entity's **cell** attribute

def onClientEnabled(*self*):

If this callback is implemented in the script, it is invoked when the entity is available (various initializations and communication with the client). This method has no parameters.
Note: giveClientTo also assigns control to the entity and causes the callback to be called.

def onGiveClientToFailure(*self*):

If this callback is implemented in a script, it is called when the entity fails to call **giveClientTo**. This method has no parameters.

def onLogOnAttempt(*self, ip, port, password*):

If this callback is implemented in a script, it is invoked when a client attempts to log in using the current account entity.
This situation usually happens when the entity that exists in memory is in a valid state, the most obvious example is user A logs in with this account, and user B tries to use the same account to log in, triggering this callback.

This callback function can return the following constant values:
KBEngine.LOG_ON_ACCEPT: Allows the new client to bind to the entity. If the entity has bound a client, the previous client will be kicked out.
KBEngine.LOG_ON_REJECT: Reject new client entity binding.

KBEngine.LOG_ON_WAIT_FOR_DESTROY: Wait for the entity to be

destroyed before the client binds.

parameters:

ip	The IP address of the client trying to log in.
port	The port to which the client attempted to log in.
password	The MD5 password used when the user logs in.

def onStreamComplete(*self, id, success*):

If you implement this callback in a script, when a user uses **Proxy**.**streamStringToClient**() or **Proxy**.**streamFileToClient**() and is completed, this callback is invoked.

parameters:

id	The id associated with the download.
success	Success or failure

Attributes documentation

__ACCOUNT_NAME__

Note:

If the proxy is an account, you can access __ACCOUNT_NAME__ to get the account name.

__ACCOUNT_PASSWORD__

Note:

If the proxy is an account, you can access __ACCOUNT_PASSWORD__ to get the MD5 password.

clientAddr

This is a tuple object that contains the client's ip and port.

clientEnabled

Whether the entity is already available. The script cannot communicate with the client until the entity is available.

hasClient

Proxy is bound to a client connection.

roundTripTime

The average round-trip time for client communication between the server and this Proxy over a period of time. This property only takes effect under Linux.

timeSinceHeardFromClient

The time (in seconds) that has passed since the client packet was last received.

KBEngine

KBEngine module

This **KBEngine** module provides Python scripts control over the loginapp process to handle entity login registration.

Member functions

def **addTimer**(initialOffset, repeatOffset=0, callbackObj=None):
def **delTimer**(id):

Callback functions

def **onLoginAppReady**( ):
def **onLoginAppShutDown**( ):
def **onRequestLogin**(loginName, password, clientType, datas):
def **onLoginCallbackFromDB**(loginName, accountName, errorno, datas):
def **onRequestCreateAccount**(accountName, password, datas):
def **onCreateAccountCallbackFromDB**(accountName, errorno, datas):

Member functions documentation

def addTimer(*initialOffset, repeatOffset=0, callbackObj=None*):

Function description:
Register a timer. The timer is triggered by the callback function callbackObj.
The callback function will be executed the first time after "initialOffset" seconds, and then will be executed once every "repeatOffset" seconds.

Example:

```
# Here is an example of using addTimer
        import KBEngine

        # Add a timer, perform the first time after 5 seconds, and e
        KBEngine.addTimer( 5, 1, onTimer_Callbackfun )

        # Add a timer and execute it after 1 second. The default use
        KBEngine.addTimer( 1, onTimer_Callbackfun )

    def onTimer_Callbackfun( id ):
        print "onTimer_Callbackfun called: id %i" % ( id )
        # If this is a repeated timer, it is no longer needed, call
        #     KBEngine.delTimer( id )
```

parameters:

initialOffset float, specifies the time interval in seconds for the timer to register from the first callback.

repeatOffset float, specifies the time interval (in seconds) between each execution after the first callback execution. You must remove the timer with the function **delTimer**, otherwise it will continue to repeat. Values less than or equal to 0 will be ignored.

callbackObj function, the specified callback function object

returns:

integer, the internal id of the timer. This id can be used to remove the timer using **delTimer**

def delTimer(*id*):

Function description:

The function delTimer is used to remove a registered timer. The removed timer is no longer executed. Single-shot timers are automatically removed after the callback is executed, and it is not necessary to use delTimer to remove it. If the delTimer function uses an invalid id (for example, has been removed), it will generate an error

A use case for the KBEngine.addTimer reference timer.

parameters:

id integer, timer id to remove

Callback functions documentation

def onLoginAppReady( ):

Function description:

This function is called back when the current process is ready.
Note: This callback interface must be implemented in the portal module (kbengine_defaults.xml ->entryScriptFile).

def onLoginAppShutDown( ):

Function description:

Process shutdown calls this function back.
Note: This callback interface must be implemented in the portal module (kbengine_defaults.xml ->entryScriptFile).

def onRequestLogin(*loginName, password, clientType, datas*):

Function description:

Called back when the client requests the server login account.

Here you can do some administrative control on user login. For example:
Use this interface to truncate the user's login here, record the request and queue it, and return an error code to tell the client the queue status.

Note: This callback interface must be implemented in the portal module (kbengine_defaults.xml ->entryScriptFile).

parameters:

loginName	string, the name of the account submitted when logging in.
password	string, MD5 password.
clientType	integer, client type, given when the client logs in.
datas	bytes, the data attached to the client request, can forward data to a third-party platform.

returns:

Tuple, the return value is (error code, real account name, password, client type, data data submitted by the client), if there is no need to extend the modification, the return value is usually to destroy the incoming value (KBEngine.SERVER_SUCCESS , loginName, password, clientType, datas).

def onLoginAppReady( ):

Function description:

This function is called back when the current process is ready.
Note: This callback interface must be implemented in the portal module (kbengine_defaults.xml ->entryScriptFile).

def onLoginAppShutDown( ):

Function description:

Process shutdown calls this function back.
Note: This callback interface must be implemented in the portal module (kbengine_defaults.xml ->entryScriptFile).

def onLoginCallbackFromDB(*loginName, accountName, errorno, datas*):

Function description:

The callback returned by dbmgr after the client requests the server login account.

Note: This callback interface must be implemented in the portal module (kbengine_defaults.xml ->entryScriptFile).

parameters:

loginName	string, the name of the account submitted when logging in.
accountName	string, the real account name (obtained from the the query at dbmgr)
errorno	integer, error code, if it is not KBEngine.SERVER_SUCCESS, login failed.
	bytes, which may be any data, such as data returned by a third-

datas party platform or data returned by dbmgr and interfaces when processing the login.

def onRequestCreateAccount(*accountName, password, data*):

Function description:

Callback when the client requests the server to create an account.

Note: This callback interface must be implemented in the portal module (kbengine_defaults.xml ->entryScriptFile).

parameters:

accountName string, the name of the account submitted by the client.

password string, MD5 password.

datas bytes, the data attached to the client request, can forward data to a third-party platform.

returns:

Tuple, the return value is (error code, real account name, password, data data submitted by the client), if there is no need to extend the modified value is usually returned to destroy the incoming value (KBEngine.SERVER_SUCCESS, loginName, password , datas).

def onCreateAccountCallbackFromDB(*accountName, errorno, datas*):

Function description:

The callback returned by dbmgr after the client requests the server to create an account.

Note: This callback interface must be implemented in the portal module (kbengine_defaults.xml ->entryScriptFile).

parameters:

accountName string, the name of the account submitted by the client.

errorno	integer, error code, if it is not KBEngine.SERVER_SUCCESS, login failed.
datas	bytes, which may be any data, such as data returned by a third-party platform or data returned by dbmgr and interfaces when processing the login.

KBEngine

KBEngine module

This **KBEngine** module provides Python scripts to control the dbmgr process to handle entity login queries and data access.

Member functions

def **addTimer**(initialOffset, repeatOffset=0, callbackObj=None):
def **delTimer**(id):

Callbacks

def **onDBMgrReady**( ):
def **onDBMgrShutDown**( ):
def **onReadyForShutDown**( ):
def **onSelectAccountDBInterface**(accountName):

Member functions documentation

def addTimer(*initialOffset, repeatOffset=0, callbackObj=None*):

Function description:
Registers a timer. The timer triggers the callback function specified by callbackObj. The callback will be executed the first time after "initialOffset" seconds, and then executed once every "repeatOffset" seconds.

Example:

```
# Here is an example of using addTimer
        import KBEngine

        # Add a timer, perform the first time after 5 seconds, and e
        KBEngine.addTimer( 5, 1, onTimer_Callbackfun )

        # Add a timer and execute it after 1 second. The default use
        KBEngine.addTimer( 1, onTimer_Callbackfun )

    def onTimer_Callbackfun( id ):
        print "onTimer_Callbackfun called: id %i" % ( id )
        # If this is a repeated timer, it is no longer needed, call
        #     KBEngine.delTimer( id )
```

parameters:

initialOffset float, specifies the time interval in seconds for the timer to register the first callback.

repeatOffset float, specifies the time interval (in seconds) after each execution of the first callback execution. You must remove the timer with the function **delTimer**, otherwise it will continue to repeat. Values less than or equal to 0 will be ignored.

callbackObj function, the specified callback function object.

returns:

integer, this function returns the internal id of the timer. This id can be used to remove the timer using **delTimer**.

def delTimer(*id*):

Function description:

The delTimer function is used to remove a registered timer. The removed timer is no longer executed. Single-shot timers are automatically removed after the callback is executed, and it is not necessary to use delTimer to remove it. If the delTimer function receives an invalid id (for example, it was removed), it will generate an error.

A use case is in the KBEngine.**addTimer** example.

parameters:

id integer, specifies the timer id to remove.

Callback functions documentation

def onDBMgrReady( ):

Function description:

This function is called back when the current process is ready.
Note: This callback interface must be implemented in the portal module (kbengine_defaults.xml ->entryScriptFile).

def onDBMgrShutDown( ):

Function description:

This function is called when the process shuts down.
Note: This callback interface must be implemented in the portal module (kbengine_defaults.xml ->entryScriptFile).

def onReadyForShutDown( ):

Function description:

If this function is implemented in a script, the callback function is called when the process is ready to exit.

You can use this callback to control when the process exits.

Note: This callback interface must be implemented in the portal module (kbengine_defaults.xml ->entryScriptFile).

returns:

bool, if it returns True, it allows the process to exit. Returning other values will cause the process to ask again after a period of time.

def onSelectAccountDBInterface(*accountName*):

Function description:

When implemented in a script, this callback returns the database interface corresponding to an account. After the interface is selected, the dbmgr operations related to this account are completed by the corresponding database interface.

Database interfaces are defined in **kbengine_defaults.xml**->dbmgr->databaseInterfaces.
Use this function to determine which database the account should be stored in based on accountName.

Note: This callback interface must be implemented in the portal module (kbengine_defaults.xml ->entryScriptFile).

parameters:

accountName string, the name of the account.

returns:

string, the database interface name (database interfaces are defined in **kbengine_defaults.xml**->dbmgr->databaseInterfaces).

KBEngine

KBEngine module

This **KBEngine** module provides the parts of the logical script layer access to **Entity**, as well as the data of other clients in the current process, and so on.

Classes

Entity
PyClientApp

Member functions

def **addBots**(reqCreateAndLoginTotalCount, reqCreateAndLoginTickCount=0, reqCreateAndLoginTickTime=0):

def **callback**(initialOffset, callbackObj):

def **cancelCallback**(id):

def **genUUID64**( ):

def **getWatcher**(path):

def **getWatcherDir**(path):

def **scriptLogType**(logType):

Callbacks

def **onInit**(isReload):
def **onFinish**( ):

Attributes

bots	**bots**
component	Read-only **string**

Member functions documentation

def addBots(*reqCreateAndLoginTotalCount, reqCreateAndLoginTickCount=0, reqCreateAndLoginTickTime=0*):

Function description:
Add a bot to the server.

Example:

```
# Here is an example of using addBots
        import KBEngine

        # Add 5 robots to the server at one time (instantaneously).
        KBEngine.addBots( 5 )

        # Add a total of 1000 robots to the server, 5 at a time, at
        KBEngine.addBots( 1000, 5, 10 )
```

parameters:

reqCreateAndLoginTotalCount	integer, integer, the total number of bots to add to the server.
reqCreateAndLoginTickCount	integer, the number of bots added to the server each interval
reqCreateAndLoginTickTime	integer, the interval of time (in seconds) between adding bots.

def callback(*initialOffset, callbackObj*):

Function description:

Registers a callback, on the callbackObj function, which will be executed once after "initialOffset" seconds.

Example:

```
# Here is an example of using callback
        import KBEngine

        # Add a timer and execute it after 1 second
```

```
        KBEngine.callback( 1, onCallbackfun )

    def onCallbackfun( ):
        print "onCallbackfun called"
```

parameters:

initialOffset float, time, in seconds, to wait before triggering the callback.

callbackObj function, the specified callback function object.

returns:

integer, this function returns the internal id of the callback. This id can be used with **cancelCallback** to remove the callback.

def cancelCallback(*id*):

Function description:

This function is used to remove a registered but not yet triggered callback. The removed callback will not be executed. If this function is passed an invalid id (for example, timer was removed), an error will be generated.

A use case is in the KBEngine.**callback** example.

parameters:

id integer, specifies the callback id to remove.

def genUUID64( ):

Function description:

This function generates a 64-bit unique ID.
Note: This function is dependent on the startup argument 'gus' of the Cellapps service process. Please set the startup arguments to be unique.
In addition, if gus exceeds 65535, the function can only remain unique on the current process.

Usage:
Unique item IDs are generated on multiple service processes and do not conflict

when combined.
A room ID is generated on multiple service process and no uniqueness verification is required.

returns:
A 64-bit integer.

def getWatcher(*path*):

Function description:

Gets the value of a watch variable from the KBEngine debug system.

Example: In the Python console of baseapp1 enter:
>>>KBEngine.getWatcher("/root/stats/runningTime")
12673648533

>>>KBEngine.getWatcher("/root/scripts/players")
32133

parameters:
path string, the absolute path of the variable including the variable name (can be viewed on the GUIConsole watcher page).

returns:
The value of the variable.

def getWatcherDir(*path*):

Function description:

Get a list of elements (directories, variable names) under the watch directory from the KBEngine debugging system.

Example: In the Python console of baseapp1 enter::
>>>KBEngine.getWatcher("/root")
('stats', 'objectPools', 'network', 'syspaths', 'ThreadPool', 'cprofiles', 'scripts', 'numProxices', 'componentID', 'componentType', 'uid', 'numClients', 'globalOrder', 'username', 'load', 'gametime', 'entitiesSize', 'groupOrder')

parameters:

path string, the absolute path of the variable including the variable name (can be viewed on the GUIConsole watcher page).

returns:

A list of elements in the directory (directories, variable names).

def scriptLogType(*logType*):

Function description:

Sets the type of information output by the current Python.print (Reference: KBEngine.LOG_TYPE_*)

Callback functions documentation

def onInit(*isReload*):

Function description:

This function is called after all scripts have been initialized since the engine started.
Note: This callback function must be implemented in the portal module (**kbengine_defaults.xml**->entryScriptFile).

parameters:

isReload bool, whether it was triggered after rewriting the loading script.

def onFinish( ):

Function description:

This function is called back when the process shuts down.
Note: This callback function must be implemented in the portal module (**kbengine_defaults.xml**->entryScriptFile).

Attributes documentation

bots

Description:

bots is a dictionary object that contains all client objects on the current process.

Types:

PyBots

component

Description:

This is the component that is running in the current scripting environment. (So far) Possible values are 'cell', 'base', 'client', 'database', 'bot', and 'editor'.

KBEngine

PyClientApp class

[KBEngine module]

PyClientApp is a part of the **KBEngine** Module. It is is client object created when a client is simulated from the bottom of C++. It cannot be created in the script layer directly.

Member functions

def **getSpaceData**(key):

Callbacks

def **onDestroy**(self):
def **onEnterWorld**(self):
def **onLeaveWorld**(self):
def **onEnterSpace**(self):
def **onLeaveSpace**(self):

Attributes

id	Read-only Integer
entities	**Entities**

A detailed description

Instances of class **Entity** represent game objects on the client.

An **Entity** can access its equivalent entities in the base and cell applications via **ENTITYCALL**. This requires a set of remotely-invoked functions (specified in the entity's .def file).

Member functions documentation

def getSpaceData(*key*):

Function description:

Get the space data of the specified key.
The space data is set by the user on the server through **setSpaceData**.

parameters:

key string, a string keyword.

returns:

string, string data for the key

Callback functions documentation

def onDestroy(*self*):

Called when the entity is destroyed.

def onEnterWorld(*self*):

If the entity is not a client-controlled entity, it indicates that the entity has entered the View scope of the entity controlled by the client on the server side. At this time, the client can see the entity.
If this entity is a client-controlled entity, it indicates that the entity has created a cell on the server and entered space.

def onLeaveWorld(*self*):

If the entity is not a client-controlled entity, it indicates that the entity has entered the View scope of the entity controlled by the client on the server side. At this time, the client can see this entity.
If the entity is a client-controlled entity, it indicates that the entity has created a cell on the server and entered space.

def onEnterSpace(*self*):

The client-controlled entity enters a new space.

def onLeaveSpace(*self*):

The client-controlled entity leaves the current space.

Attributes documentation

entities

Description:

entities is a dictionary object that contains all the entities in the current process.

Types:

Entities

KBEngine

Entity class

[KBEngine module]

Entity is part of the **KBEngine** module. **More...**

```
import KBEngine
```

Member functions

def **moveToPoint**(self, destination, velocity, distance, userData, faceMovement, moveVertically):

def **cancelController**(self, controllerID):

Callbacks

def **onEnterWorld**(self):
def **onLeaveWorld**(self):
def **onEnterSpace**(self):
def **onLeaveSpace**(self):

Attributes

base	Read-only **ENTITYCALL**
cell	Read-only **ENTITYCALL**
className	Read-only **string**
clientapp	Read-only **PyClientApp**
direction	Tuple of 3 floats as (roll, pitch, yaw)
id	Read-only Integer
position	**Vector3**
spaceID	Read-only uint32
isOnGround	Read-only bool

A detailed description

Instances of the class **Entity** represent game objects on the client.

An **Entity** can access its equivalent entities in the base and cell applications via **ENTITYCALL**. This requires a set of remotely-invoked functions (specified in the entity's .def file).

Member functions documentation

def moveToPoint(*self, destination, velocity, distance, userData, faceMovement, moveVertically*):

Function description:

Moves the **Entity** to the given coordinate point in a straight line. The callback will be called on success or failure.
Any **Entity** can only have one motion controller at any time. Repeatedly calling any move function will terminate the previous move controller.
Returns a controller ID that can be used to cancel this move.

Example:
You can use **Entity**.**cancelController**(movementID) or **Entity**.**cancelController**("Movement") to cancel the move. The callback will not be called if the move is cancelled.

The callback functions are defined as follows:

```
def onMove( self, controllerID, userData ):
def onMoveOver( self, controllerID, userData ):
def onMoveFailure( self, controllerID, userData ):
```

See also:

Entity.**cancelController**

parameters:

destination	Vector3, the target point to which the **Entity** is to be moved.
velocity	float, the speed to move the **Entity** (in m/s).
distance	float, the distance target which if it is within, movement is stopped. If the value is 0, it moves to the target position.
userData	object, user data passed to the callback function.
faceMovement	bool, True if the entity faces the direction of the move. If it is other mechanism, it is False.
moveVertically	bool, set to True means to move in a straight line directly to the point, and False means to move in a straight line paralell to the ground.

returns:

int, newly created controller ID.

def cancelController(*self, controllerID*):

Function description:

The cancelController function stops the effect of a controller on the **Entity**. It can only be called on a **real** entity.

parameters:

controllerID	integer, the index of the controller to cancel. A special controller type string can also be used. For example, only one movement/navigation controller can be activated at a time. This can be cancelled with entity.cancelController("Movement").

Callback functions documentation

def onEnterWorld(*self*):

If the entity is not a client-controlled entity, it indicates that the entity has entered the View scope of the entity controlled by the client on the server side. At this time, the client can see this entity.
If the entity is a client-controlled entity, it indicates that the entity has created a cell on the server and entered space.

def onLeaveWorld(*self*):

If the entity is not a client-side control entity, it indicates that the entity has left the view scope of the client-controlled entity on the server side, and the client cannot see this entity at this time.
If the entity is a client-controlled entity, it indicates that the entity has destroyed the cell on the server and left space.

def onEnterSpace(*self*):

The client-controlled entity enters a new space.

def onLeaveSpace(*self*):

The client-controlled entity leaves the current space.

Attributes documentation

base

base is the entityCall used to contact the base **Entity**. This attribute is read-only, and is None if this entity does not have an associated base **Entity**.

Other references:

Entity.**clientEntity**
Entity.**allClients**
Entity.**otherClients**

Types:

Read-only, **ENTITYCALL**

cell

Description:

cell is the **ENTITYCALL** used to contact the cell entity. This attribute is Read-only, and is None if the base entity has no associated cell.

Types:

Read-only **ENTITYCALL**

cellData

Description:

cellData is a dictionary property. Whenever the base entity has not created its cell entity, the attributes of the cell entity are stored here.

If the cell entity is created, the values and **cellData** attributes will be deleted. In addition to the attributes that the cell entity specifies in the entity definition file, it also contains position, direction, and spaceID.

Types:

CELLDATADICT

className

Description:

The class name of the entity.

Types:
Read-only, string

clientapp

Description:

The client (object) to which the current entity belongs.

Types:
Read-only, PyClientApp

position

The coordinates (x, y, z) of this entity in world space. The data is synchronized from the server to the client.

Types:
Vector3

direction

This attribute describes the orientation of the Entity in world space. Data is synchronized from the server to the client.

Types:
Vector3, which contains (roll, pitch, yaw) in radians.

isOnGround

If the value of this attribute is True, the **Entity** is on the ground, otherwise it is False.
If it is a client-controlled entity, this attribute will be synchronizd to the server at the time of change, and other entities will be synchronized to the client by the server. The client can determine this value to reduce the cost of accuracy.

Types:

Read-write, bool

KBEngine

KBEngine module

This **KBEngine** module mainly handles access of third-party platforms for the KBEngine server.

Member functions

def **addTimer**(initialOffset, repeatOffset=0, callbackObj=None):

def **accountLoginResponse**(commitName, realAccountName, extraDatas, errorCode):

def **createAccountResponse**(commitName, realAccountName, extraDatas, errorCode):

def **chargeResponse**(orderID, extraDatas, errorCode):

def **delTimer**(id):

Callbacks

def **onInterfaceAppReady**( ):
def **onInterfaceAppShutDown**( ):
def **onRequestCreateAccount**(registerName, password, datas):
def **onRequestAccountLogin**(loginName, password, datas):
def **onRequestCharge**(ordersID, entityDBID, datas):

Member functions documentation

def addTimer(*initialOffset, repeatOffset=0, callbackObj=None*):

Function description:
Registers a timer. The timer triggers the callback function callbackObj. The callback function will be executed the first time after "initialOffset" seconds, and then will be executed once every "repeatOffset" seconds.

Example:

```
# Here is an example of using addTimer
        import KBEngine

        # Add a timer, perform the first time after 5 seconds, and e
        KBEngine.addTimer( 5, 1, onTimer_Callbackfun )

        # Add a timer and execute it after 1 second. The default use
        KBEngine.addTimer( 1, onTimer_Callbackfun )

    def onTimer_Callbackfun( id ):
        print "onTimer_Callbackfun called: id %i" % ( id )
        # If this is a repeated timer, it is no longer needed, call
        #     KBEngine.delTimer( id )
```

parameters:

initialOffset	float, specifies the time interval in seconds for the timer to register from the first callback.
repeatOffset	float, specifies the time interval (in seconds) between each execution after the first callback execution. You must remove the timer with the function **delTimer**, otherwise it will continue to repeat. Values less than or equal to 0 will be ignored.
callbackObj	function, the specified callback function object

returns:

integer, the internal id of the timer. This id can be used to remove the timer from **delTimer**.

def accountLoginResponse(*commitName, realAccountName, extraDatas, errorCode*):

Function description:

After onRequestAccountLogin is called back, the script needs to call this function to give the result of the login processing.

parameters:

commitName	string, the name submitted by the client when requested.
realAccountName	string, returns the real account name (if there are no special requirements it is ually commitName, this is available when logging in with various alias accounts).
extraDatas	bytes, the data attached to the client's request. Can forward the data to a third-party platform and provide an opportunity to modify it. This parameter can be read in the script via the getClientDatas interface of the base entity.
errorCode	integer, error code. If you need to interrupt the user's behavior, you can set the error code here. The error code can be referenced (KBEngine.SERVER_ERROR_*, described in kbengine/kbe/res/server/server_errors.xml), otherwise submitting KBEngine.SERVER_SUCCESS represents permitting the login.

def createAccountResponse(*commitName, realAccountName, extraDatas, errorCode*):

Function description:

After onRequestCreateAccount is called back, the script needs to call this function to give an account creation processing result.

parameters:

commitName	string, the name submitted by the client when requested.
realAccountName	string, returns the real account name (if there are no special requirements it is ually commitName, this is available when logging in with various alias accounts).
extraDatas	bytes, the data attached to the client's request. Can forward the data to a third-party platform and provide an opportunity to modify it. This parameter can be read in the script via the getClientDatas interface of the base entity.

errorCode	integer, error code. If you need to interrupt the user's behavior, you can set the error code here. The error code can be referenced (KBEngine.SERVER_ERROR_*, described in kbengine/kbe/res/server/server_errors.xml), otherwise submitting KBEngine.SERVER_SUCCESS represents permitting the login.

def chargeResponse(*orderID, extraDatas, errorCode*):

Function description:

After onRequestCharge is called back, the script needs to call this function to give the billing result.

parameters:

ordersID	string, the ID of the order
extraDatas	bytes, the data attached to the client's request. Can forward the data to a third-party platform and provide an opportunity to modify it. This parameter can be read in the script via the getClientDatas interface of the base entity.
errorCode	integer, error code. If you need to interrupt the user's behavior, you can set the error code here. The error code can be referenced (KBEngine.SERVER_ERROR_*, described in kbengine/kbe/res/server/server_errors.xml), otherwise submitting KBEngine.SERVER_SUCCESS represents permitting the login.

def delTimer(*id*):

Function description:

The delTimer function is used to remove a registered timer. The removed timer is no longer executed. Single-shot timers are automatically removed after the callback is executed, and it is not necessary to use the delTimer function to remove it. If the delTimer function is passed an invalid id (for example, timer was removed), it will generate an error.

A use case is shown in the **KBEngine.addTimer** example.

parameters:

id integer, which specifies the timer id to remove.

Callback functions documentation

def onInterfaceAppReady( ):

Function description:

This function is called back when the current process is ready.
Note: This callback interface must be implemented in the portal module (kbengine_defaults.xml ->entryScriptFile).

def onInterfaceAppShutDown( ):

Function description:

This function is called back when the process shuts down.
Note: This callback interface must be implemented in the portal module (kbengine_defaults.xml ->entryScriptFile).

def onRequestCreateAccount(*registerName, password, datas*):

Function description:

This callback is called when the client requests the server to create an account.

The data can be checked and modified within this function, and the final result is submitted to the engine through KBEngine.createAccountResponse.

Note: This callback interface must be implemented in the portal module (kbengine_defaults.xml ->entryScriptFile).

parameters:

registerName	string, the name submitted by the client when requested.
password	string, password
datas	bytes, the data attached to the client's request, can forward data to a third-party platform.

def onRequestAccountLogin(*loginName, password, datas*):

Function description:

This callback is called when the client requests the server to login an account.

The data can be checked and modified within this function, and the final result is submitted to the engine through KBEngine.accountLoginResponse.

Note: This callback interface must be implemented in the portal module (kbengine_defaults.xml ->entryScriptFile).

parameters:

loginName	string, the name submitted by the client when requested.
password	string, password.
datas	bytes, the data attached to the client request, can forward data to a third-party platform.

def onRequestCharge(*ordersID, entityDBID, datas*):

Function description:

This callback is invoked when billing is requested (usually KBEngine.charge is called on baseapp).

Data can be checked and modified within this function, and the final result is submitted to the engine via KBEngine.chargeResponse.

Note: This callback interface must be implemented in the portal module (kbengine_defaults.xml ->entryScriptFile).

parameters:

ordersID	uint64, the ID of the order.
entityDBID	uint64, the entity DBID of the submitted order.
datas	bytes, the data attached to the client request, can forward data to a third-party platform.

KBEngine

KBEngine module

This **KBEngine** module provides Python scripts the ability to control, analyze, and dump certain types of logs.

Member functions

def **addTimer**(initialOffset, repeatOffset=0, callbackObj=None):
def **delTimer**(id):

Callbacks

def **onLoggerAppReady**( ):
def **onLoggerAppShutDown**( ):
def **onLogWrote**(datas):
def **onReadyForShutDown**( ):

Member functions documentation

def addTimer(*initialOffset, repeatOffset=0, callbackObj=None*):

Function description:
Registers a timer. The timer triggers the callback function callbackObj. The callback function will be executed the first time after "initialOffset" seconds, and then will be executed once every "repeatOffset" seconds.

Example:

```
# Here is an example of using addTimer
        import KBEngine

        # Add a timer, perform the first time after 5 seconds, and e
        KBEngine.addTimer( 5, 1, onTimer_Callbackfun )

        # Add a timer and execute it after 1 second. The default use
        KBEngine.addTimer( 1, onTimer_Callbackfun )

    def onTimer_Callbackfun( id ):
        print "onTimer_Callbackfun called: id %i" % ( id )
        # If this is a repeated timer, it is no longer needed, call
        #     KBEngine.delTimer( id )
```

parameters:

initialOffset	float, specifies the time interval in seconds for the timer to register from the first callback.
repeatOffset	float, specifies the time interval (in seconds) between each execution after the first callback execution. You must remove the timer with the function **delTimer**, otherwise it will continue to repeat. Values less than or equal to 0 will be ignored.
callbackObj	function, the specified callback function object

returns:

integer, the internal id of the timer. This id can be used to remove the timer from **delTimer**.

def delTimer(*id*):

Function description:

The delTimer function is used to remove a registered timer. The removed timer is no longer executed. Single-shot timers are automatically removed after the callback is executed, and it is not necessary to use the delTimer function to remove it. If the delTimer function is passed an invalid id (for example, timer was removed), it will generate an error.

A use case is shown in the KBEngine.**addTimer** example.

parameters:

id integer, which specifies the timer id to remove.

Callback functions documentation

def onLoggerAppReady( ):

Function description:

This function is called back when the current process is ready.
Note: This callback interface must be implemented in the portal module (kbengine_defaults.xml ->entryScriptFile).

def onLoggerAppShutDown( ):

Function description:

This function is called back when the process shuts down.
Note: This callback interface must be implemented in the portal module (kbengine_defaults.xml ->entryScriptFile).

def onLogWrote(*datas*):

Function description:

If this function is implemented in the script, it is invoked when the logger process obtains a new log.

The database interface is defined in **kbengine_defaults.xml**->dbmgr->databaseInterfaces.

Note: This callback interface must be implemented in the portal module (kbengine_defaults.xml ->entryScriptFile).

parameters:

datas bytes, log data.

def onReadyForShutDown( ):

Function description:

If this function is implemented in the script, it is called when the process is ready to exit.

You can use this callback to control when the process exits.

Note: This callback interface must be implemented in the portal module (kbengine_defaults.xml ->entryScriptFile).

returns:

bool, if it returns True, it allows the process to exit. Returning other values will cause the process to ask again after a period of time.

Cellapp process

The Cellapp process is primarily responsible for space-related game logic, providing players on different baseapps a real-time interaction in one space. The Cellapp can usually implement scene-related logic such as NPCs/monsters, battles, and checkpoint rooms.

Baseapp process

The Baseapp process is mainly responsible for communication with the client, location independent game logic (guild manager, chat system, leaderboard, etc.), archiving, backup, and so on.

Loginapp process

The Loginapp process is primarily responsible for handling entity registration and login requests.
Note: This process script is implemented in the scripts/login directory.
