# 运行时环境(runtime)

底层使用cocos2d-x,同时我们也根据需要扩展一部分功能。上层是cocos2d-x lua binding它提供一个统一的Lua接口。参见下表

最终目标是构造一个方便的编辑器用于构造游戏，一个播放器用于玩游戏。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 编辑器 | | | 播放器 | | |
| 场景(scene) | 入场方式 | 层(layer) | 界面 | 对象(node) | 粒子系统 |
| 对象创建层(将文件中的对象描述，载入到系统) | | | | | |
| Lua-binding | | | | | |
| Cocos2d-x | | | 我们的扩展 | | |
| android | | ios | | Windows | |

# 对象的文件系统(object file system)

以下说的对象其实是指对象的类

系统里面出现的诸多对象，例如场景，场景里的精灵对象，都以一种统一的方式被组织。

每个对象都有自己的专有目录，名称是一个全局唯一的ID( uuid)。该对象目录里放置对象的描述文件，和源代码，以及资源文件。

对象的描述文件是一个desc.json文件，格式举例如下:

desc.json

{

classid:” 6acf084856854760ab1c3919d2784f76”,

superid:” 5ec0b375a0dd27b5aa230d094266639b”,

name:”滚动场景”,

icon:”scrollscrene.png”,

script:”scrollscrene.lua”,

pedigree:{

“7bk154854584760ab2e3913d2485d42”,

“34195ce9c6d0209dfbe0c1ccfdfa772f”,

}

version:1

}

|  |  |
| --- | --- |
| classid | 对象类ID,和目录名称相同 |
| superid | 父类的ID |
| name | 名称 |
| icon | 对象类的图标,用于编辑器，或者其他的对象类列表。 |
| script | 脚本文件，描述对象的逻辑。 |
| depends | 脚本中如果用到一些对象需要将其classid加入到这个列表. |
| pedigree | 继续关系列表,第一个为父类的父类依次往下。 |
| progressid | 类在被加载时使用的加载界面。(用在那些可以独立启动的对象) |
| splashid | 加载过程中的splash阶段。 |
| version | 版本 |
| comment | 备注描述对象的用处等信息。 |

version.json用来描述对象的版本,当对象的任何文件改变了版本都需要改变

{

version:1

}

filelist.json用来描述对象包含的全部文件与文件的md5校验码

[

{name:” cache.lua”,md5=” 030b49475bf4c97edbfa1564e31faacb”},

{name: "uikits.lua", md5: "e92892ed42b71bd093c2fa63f8cb17ba"}

]

depends.json用来描述和对象相关的全部相关对象的列表。只有可以独立启动游戏、应用，有这个文件。

[

“4ba836dc8112b7cfa4747a7eee521fd9”,

“0c350b7fbf4b0e1edec9764a0593093c”

]

对象都被存放在class目录下。

目录结构大概如下:

root

-src

-res

-class-“4ba836dc8112b7cfa4747a7eee521fd9”

-desc.json

-version.json

-filelist.json

-depends.json

….

# 对象的继承

对象只能有一个父类，也就单继承模式。另外所有对象都有唯一的根父类。这个类叫作root

在base.lua中有关于它的定义。Id为7c3064bb858e619b9f02fef85432f162。

Root有如下函数:

getR(res) 所有的资源都使用该函数实现替换。

isInstanceOf(A) 用来判断对象是不是属于某一个类，只要在对象的继承链上发现A就返回true,否则返回nil

# 脚本文件(script)

脚本文件使用lua,对象的逻辑都将在此脚本文件中处理。文件在结尾返回一个表。举例如下:

写法可以多样

local cat = {

“init” = function( self,data )

self.super.init( self,data )

end,

“release” = function( self,b )

self.super.release(self,b)

end,

“enter” = function( self,scene )

self.super.enter( self,scene )

end,

“exit” = function( self,scene )

end,

“collide” = function( self,objs )

end,

}

return cat

或者可以这么写

local cat = {}

function cat:init( data )

self.super.init( self,data ) –写法有点怪异

self.hp = 100

end

function cat:release( b )

end

function cat:enter()

local b = create(ball) –创建一个球

self.scene:add( b ) –将球加入到创建中去

end

function cat:exit()

end

function cat:collide( objs )

end

return cat

描述文件被命名为desc.json,同时增加depends.json依赖类表。该文件用来描述类的依赖关系，在加载时

# 对象工厂(object factory)

对象工厂从对象类加上数据产生对象实例。它由一组函数组成。

create\_object(data)

delete\_object( obj )

get\_object( id )

这组全局函数用于创建对象。

对象工厂首先找到对象类存放的位置（通过id,id直接和对象类存放的目录同名）。然后读取类的描述，递归的将父类都加载到内存中，对象以lua 表的形式存在。

并且调用新创建的对象表的init函数。init函数可以附加数据data,它可以是一个表,值或者字符串,由函数init对其进行解释。如果对象有父类还要向上传递初始化事件。

function init( self,data )

self.super.init( self,data)

end

父类的函数表被放入对象表的元表和字段super中，这样子类没有被重载的函数会自动调用父类的。同时子类也可以直接通过super字段来访问父类函数。

这里涉及任何对象都必须有的几个函数

function init( self,data )

end

data可以有可以没有，没有就初始化为默认值。有的话就按data提供的数据初始化对象。

成功返回true,失败返回false,errmsg

function save( self )

end

函数返回保存的数据，可以是任何数据。

每个对象都具有一个唯一的变量id,并且id为只读变量。

并且有一个classid,也是只读变量。

函数返回对象的id

function getR( self,r) 该函数将一个资源名称r,映射为资源的绝对名称。

end

成功返回资源的绝对路径,失败返回nil

create\_object(data)

首先data中是一个固定格式的表。

举例:

{

classid:” fdfc8b206aab2cbf1dc9e88c784c745b”,

objected:” 34195ce9c6d0209dfbe0c1ccfdfa772f”,

data:{

hp:100,

mp:32,

…

}

}

create\_object 首先取得id,然后根据id找到对象类存放的目录。然后创建该对象，最后调用对象的init函数同时传入data。

成功返回对象，失败nil。

如果没有对象数据可以通过构造一个表来创建默认值得对象。

create\_object{ classid = “6acf084856854760ab1c3919d2784f76” }

get\_object( id ) 函数通过给出对象id取得对象,系统中被创建的对象都被放入一个表中。

# 实例数据(object instance)

实例数据使用对象save的数据再次重建对象，通过创建对象然后调用init实现。

对于一般的树状结构实例结构可以顺序加载不需要额外出来。

{

classid:”scroll\_game\_classid”,

objected:” 5ec0b375a0dd27b5aa230d094266639b”,

data:{

scene:[

{

classid:”scroll\_screne\_classid”,

objected:” c5aba3fdf5e123f1c0d9b7c106362949”,

data:{

layer:[

{

classid:”scroll\_layer\_classid”,

objected:” d081b23850c5d30517a72cd349b3a2c7”,

data:

….

}

]

}

},

],

start\_scene:1

….

}

}

上面的实例数据描述，标准的树状结构。一个游戏可以有多个场景，场景可以有多个层，层可以有很多的对象。

game—scene—layer—object1

这种树状结构的加载是比较简单的，game通过读取scene表就可以知道怎么创建场景。然后依次类推。

如果是一个网状结构的数据加载。或者简单点两个对象互相的关联，一个对象引用另一个对象。

例如有这样的任务，一个对象要跟随另外一个对象。对象A跟随对象B

对象A有一个变量followObject = B

在保存的时候这么写

function A:save()

t = self.super.save(self)

t.followObjectId = self.followObject.id

return t

end

初始化函数这么写

function A:init(data)

self.super.init(data)

ref( data.followObjectId,function(obj)

self.followObject = obj

end )

end

因此这里就要求create\_object 需要一个上下文，并且在加载过程中处理引用这种情况。

# 资源的继承与重载(resource)

一个对象在加载资源的时候，必须有一个固定的查找顺序。首先它在自己的目录里面查找，如果没找到它顺着继承树向上查找。

举例说明:

比如有对象类A,和对象类B,A是基类。

A目录下有如下文件A/images/1.png,A/images/2.png

B目录下有如下文件B/images/2.png

那么在对象类B的实例中

self:getR(‘images/1.png’) 返回 A/images/1.png

self:getR(‘images/2.png’) 返回 B/images/2.png

显然B可以通过增加和A中的同名资源来重载A的资源

# 游戏(game)

游戏中可能有很多场景，包括游戏的存档，游戏的方式是单人游戏还是多人的。

有对象提供一组函数，用于创建场景，切换场景，退出游戏，等等。

game:launch() –启动游戏

game:addScene( data ) –加入一个场景

game:removeScene( scene ) –删除场景

game:scenes() –枚举全部场景

游戏对象同时维护一个游戏中派生的对象类的目录

# 场景(scene)

场景中包括一系列层，比如背景层，UI层等等。层和层之间通过z值来确定上下关系。

scene:run() –开始运行场景

scene:pause() –暂停场景

scene:addLayer(data) –加入场景

scene:layers() –返回场景中的全部层对象

scene:removeLayer(layer) –删除一个层对象

# 层(layer)

# 场景对象(node)

# 游戏的加载与版本控制

在加载时首先检测本地代码缓存如果有就使用,没有从网络上下载到本地缓冲然后使用。如果发现版本变化做差异性跟新。关于对象的版本遵循兼容原则，如果一个对象的修改造成不兼容问题，它应该做继承操作创建一个新对象，而不是修改原来的对象。