基于 cocos2d-x 引擎的游戏框架设计

2011-12-20

李成（关中刀客）&郑鑫

一：前言

时下，移动互联网浪潮的到来，正在彻底改变人们日常的生活习惯和方式，相应的，基于移动终端和感应交互的游戏，也为人们带来了全新的游戏体验和感受。下面，我们将结合目前流行的 cocos2d-x 引擎，使用 C++语言，基于 IOS 平台，和大家分享一下 IPhone，IPad 上游戏客户端的构架与实现。

二：游戏架构与实现

目前很多基于 cocos2d-x 的代码，基本上仅是对引擎功能的使用而已，完全不能按照游戏项目的标准来参考。作为游戏项目的代码，我们不仅仅需要实现游戏的诸多功能，还需要从架构层面，从模块设计的角度来思考和设计，使我们的代码具有更好的复用性和拓展性。

对于游戏客户端，按照功能模块的划分，一般可以分为以下几个独立的模块：引擎封装层模块；游戏数据管理模块；应用程序配置模块；日志记录模块；网络管理模块；消息事件机制模块；输入输出控制模块；音效管理模块；UI 系统模块；逻辑系统处理模块；强大的调试器控制模块等，整体的框架如下图所示：

针对不同类型的游戏，通常我们只需要单独实现最上层的游戏逻辑系统，而剩余的模块，完全可以复用。下面，我们将详细讲解一下各个模块的职能与实现（暂不包含游戏逻辑系统）。

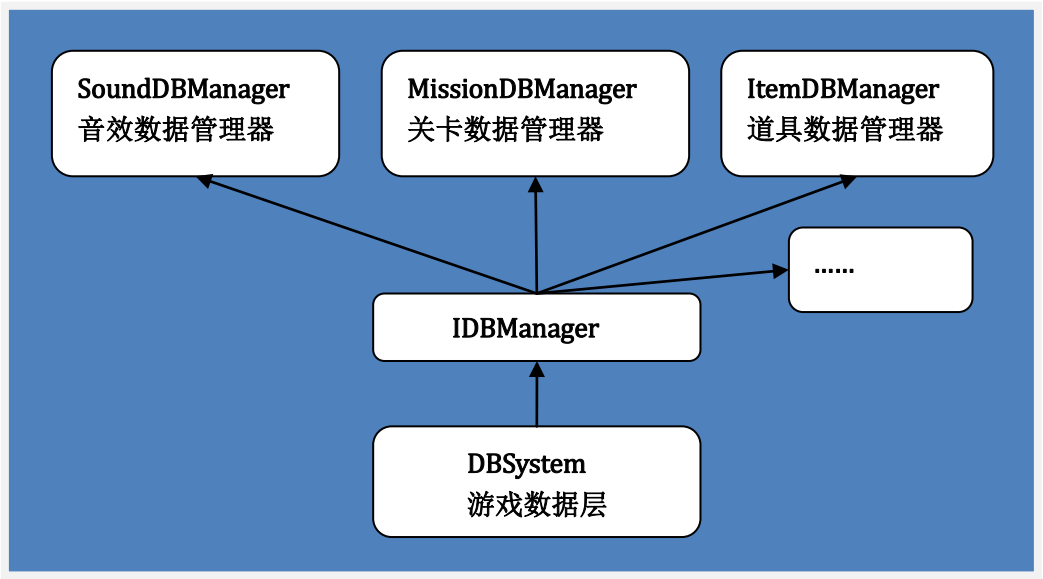
2.1 引擎封装模块（EngineSystem）

为了减少客户端代码对 cocos2d-x 引擎的依赖程度，降低耦合度，我们建立了引擎封装层模块，将引擎必要的初始化，逻辑更新，渲染和资源管理等操作全部交给引擎封装层来处理，使得客户端其他模块不需要过于依赖引擎层。同时，为了避免客户端代码中频繁，直接的调用平台相关的诸多功能，我们还将一些平台相关的功能全部封装在引擎封装层模块里面。

cocos2d-x 功能很多很强大，但是在实际开发时，我们需要根据项目需要有条件选择引擎功能（当然，cocos2d-x 本身设计实现的很好），例如，在引擎封装层内部，我们自始至终仅使用了一个 CCScene 对象，在设计之初就刻意避免处理多个 CCScene 之间的初始化，跳转，销毁，更新等操作，极大简化了我们逻辑层代码，降低了复杂度，且到目前为止，表现效果上没有什么影响。

2.2 数据管理模块（DBSystem）

在开发过程中，我们经常会面临存储和读取大量的游戏静/动态数据，针对这部分，我们设计了 DBSystem 模块，专门进行整个游戏数据层的管理。每一种类型的游戏数据，都会派生出一个具体的类，如音效数据管理器，图片数据管理器，剧情数据管理器等，这些数据管理器，都在 DBSystem 内部统一的进行初始化，更新和销毁，并且各自使用单例模式，外层使用时，直接通过其类进行数据读取即可，无需关心其初始化，逻辑更新，销毁等操作。同时，为了时刻对游戏的静态数据进行监控，我们所有的数据模块，都暴露了获取其所包含数据的接口，这样我们就可以在游戏中随时的获取数据层的信息，方便进行统计和监控，如图所示：



2.3 应用配置系统（VariableSystem）

从产品角度来讲，一般我们都需要对产品的应用属性进行可配置化处理：一方面可以方便开发者快速开启/屏蔽某些功能；另一方面也是为了更人性化的支持用户偏好设置。目前，根据类型的不同，我们建立了账号，网络，日志三种配置文件，分别对游戏账号，游戏功能信息，游戏网络配置信息，游戏日志配置信息进行动态设置，其全部使用 XML 进行数据存储和读取。

在开发过程中，我们通常都需要保存大量的内存临时数据，而这些数据，往往被放在各个模块内部，如果其他模块需要使用，就造成两个模块之间强行的依赖，增加了耦合度。所以，针对这部分，我们将所有临时需要的数据，统一定位为内存配置数据，也放在我们的应用配置系统中，其和账号，网络，日志配置文件的区别就在于：基于文件配置的属性数据，都需要在程序退出时强行写回文件，而基于内存配置的属性数据，无需进行保存。具体的应用如下图所示：

```
// 设置本地UDP服务器的IP和端口
char buffer[1024] = "127.0.0.1";
VARIABLESYSTEM->SetVarible(VariableSystem::_TYPED_CONFIG_MEM_, "NET", "LocalUDPIPAddr", buffer);
int port = 9999;
VARIABLESYSTEM->SetVarible(VariableSystem::_TYPED_CONFIG_MEM_, "NET", "LocalUDPPort", port);

// 设置远程UDP服务器的IP和端口
char buffer2[1024] = "127.0.0.1";
VARIABLESYSTEM->SetVarible(VariableSystem::_TYPED_CONFIG_MEM_, "NET", "ServerUDPIPAddr", buffer2);
int port2 = 10000;
VARIABLESYSTEM->SetVarible(VariableSystem::_TYPED_CONFIG_MEM_, "NET", "ServerUDPPort", port2);

do
{
    // 初始化本地UDP服务器套接字
    m_hSocket = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
    if (m_hSocket == -1)
        break;

    // 从VariableSystem中获取本地UDP服务器的IP和端口
    char buffer[1024] = "";
    VARIABLESYSTEM->GetVarible(VariableSystem::_TYPED_CONFIG_MEM_, "NET", "LocalUDPIPAddr", buffer, sizeof(buffer));
    int port = 0;
    VARIABLESYSTEM->GetVarible(VariableSystem::_TYPED_CONFIG_MEM_, "NET", "LocalUDPPort", port);

    struct sockaddr_in clientAddress;
    memset(&clientAddress, 0, sizeof(sockaddr_in));
    clientAddress.sin_family = AF_INET;
    clientAddress.sin_port   = htons(port);
    clientAddress.sin_addr.s_addr = inet_addr(buffer);
    if (bind(m_hSocket, (struct sockaddr*)&clientAddress, sizeof(sockaddr)) == -1)
        break;

    // 设置可复用性
    Socket::set_socket_reuse(m_hSocket);
    // 设置为非阻塞式
    Socket::set_socket_unblock(m_hSocket);

    m_State = _TYPED_STATE_RUNING_;
    return true;
}
while (false);
```

2.4 音效控制模块（SoundSystem）

对于音效处理，IOS 平台下并没有十分完善的音效引擎，而一般自行实现的音效库都难以进行拓展和支持跨平台，所以，我们直接选取流行的 FmodEx 引擎，进行游戏音效的播放和管理。同时，结合 FmodEx 提供的强大接口，我们可以很方便的实现声音大小设置，暂停，循环，3D 音效等操作，完全满足一般游戏的需求。

游戏过程中，一般都需要频繁的播放多个音效，为了提高效率和节省内存，在逻辑层，我们对每一个音效文件都使用了引用计数技术，同一种音效文件，仅需通过计数的方式维持一份实例即可，同时播放的多个相同音效，实际上都使用了同样的一份实例而已，无需单独创建音效实例；另外，通过引用计数，我们也很好的解决了音效资源回收的问题，当音效资源计数为零时，即表示其可以被回收，对应的资源，占用的内存也将被释放。

2.5 日志系统模块（LoggerSystem）

为了方便在开发，运营期对出现的问题及时进行定位和排查，对于游戏中关键的处理流程，我们都需要进行日志记录。

在客户端，我们仿照 Log4J 的方式，实现了分级（Trace 级，Info 级，Error 级等），分文件，分输出方式的强大日志管理。游戏的日志模块，结合我们的应用配置系统，完全实现了动态化配置，通过对日志配置文件进行设

置修改，开发者可以很方便的设置日志的开启等级，输出方式，大小拆分，输出名称等。另外，对于客户端日志模块来说，我们无需过多的考虑其性能问题，所以，我们的日志模块，完全是简单的在主线程里面进行文件写入，没有多开线程进行文件操作。

2.6 消息事件系统模块（EventSystem）

考虑到客户端框架总体的拓展性，所以，我们完全使用事件驱动模型（Event-driven）来设计和开发。使用事件驱动模式，将客户端中事件的触发时机和具体的处理逻辑彻底分隔开，游戏的各个模块，仅需要注册，监听和实现其关心的消息事件，而无需关心事件何时被触发，降低了总体的耦合度。目前为止，游戏中所有 UI 面板的隐藏/显示，事件响应；音效的播放/停止；游戏流程的切换；游戏角色状态迁移等，完全通过事件驱动的方式开发；同时，这种基于事件的处理方式，为项目使用动态脚本拓展提供了支持：脚本层省去对逻辑代码的大量直接调用，通过消息事件，完成脚本层和逻辑层的交互调度，大大简化了开发的复杂度。

2.7 UI 系统（GUISystem）

熟悉 cocos2d，cocos2d-x 的朋友一定都知道，这两种引擎本身并没有提供太多的 UI 控件，仅提供了按钮，进度条等基础控件，如果想使用更多的 UI 控件，需要开发者借鉴或使用其他成熟的 GUI 引擎，如 CEGUI 等。

在我们的 UI 系统中，充分借鉴了 CEGUI 的设计思想，整体上，将游戏中有关联关系的 UI 控件集中到一个个单独的 CCLayer 上面，组成多个独立的 Layout，也就是我们在代码中定义的 IWindow 类。每一个 IWindow 类，都包含其自身的一个根面板（CCLayer）和众多依附在其之上的子 UI 控件。通过 IWindow，我们实现了对所有的 UI 布局 Layout 进行统一的接口调用和处理，如初始化加载，消息注册与响应，隐藏/显示，销毁等。

在之前多个项目开发中，虽然我们提倡将窗口的逻辑实现全部交给动态语言（Python&Lua）来实现，但是，对于某一些 UI 功能，并不适合使用脚本来进行逻辑拓展，对于这类难度比较大，复杂度特别高的 UI 处理，使用原生的 C++开发可能更为合适，所以，在 UI 系统中，我们针对这两种不同的需求，对 IWindow 进行了拓展：对于需要使用脚本来拓展逻辑的 Layout，我们派生出 UIWindowByScript 类，其内部主要就是通过消息事件机制，将对应窗口的初始化，加载，逻辑更新，事件处理，销毁等操作传递给对应的脚本逻辑来处理；对于需要使用 C++来进行处理的 Layout，我们直接根据功能需要，从 IWindow 上派生出各个具体的实现类。

目前我们所有的 UI 布局，还没有完全达到通过外部配置文件来动态的实现，这部分，接下来我们将借鉴 CEGUI 的处理方式，将所有 UI 的布局信息，控件属性与事件响应处理等全部使用外层配置文件实现，从而将这些静态信息和程序分隔开，达到动态配置的目的。

2.8 网络管理模块（NetSystem）

对于网络模块，考虑到传统类型游戏和即时竞技类游戏的差异性，所以，在设计之初，我们就同时支持了 UDP 和 TCP 两种通讯方式。

为了支持 UDP 通讯方式，我们使用监听器模型，在客户端中设计了 UDPAcceptor 管理器，专门进行 UDP 通讯方式的初始化，操作和销毁。同时，考虑到 UDP 通讯方式的特殊性，在 UDPAcceptor 内部，我们对收到的数据进行了杂乱包的过滤，并且使用序列号技术（Sequence Number），实现了 UDP 数据包先后顺序的管理和纠正，确保最终交给逻辑层处理的数据包，都是完整可靠有序的。

对于 TCP 通讯方式，我们使用连接器模型，在客户端中封装了 TCPConnector 管理器，为了很好的解决粘包，拼接，临时数据拷贝等问题，我们内部设计了特殊的 MemNode 存储结构，将读取的数据，全部存储到 MemNode 里面，其内部根据当前数据的读，写指针位置来进行有效数据定位，确保最终交给上层逻辑使用的数据包，都是完整独立的数据。

网络模块，面临着频繁的数据接受和发送，如果不进行控制，频繁的 new/delete，对性能会有一定的影响，所以，在网络系统内部，我们使用链表式内存池技术，对于接收和发送的数据包，都是通过该内存池进行统一分配，回收和管理，从根本上解决了频繁的 new/delete。另外，NetSystem 内部，我们使用网络编程中传统的 Selector 模式，进行网络连接的监听，轮训，读取和发送。目前市面上很多游戏对于网络模块，都是单开线程进行处理，而在我们开发中，考虑到多线程同步，数据串行化等问题，所以我们尽量避免了多线程的方式，使用非阻塞式 IO，全部在逻辑主线程里面进行网络控制管理。

2.9 输入控制系统（InputSystem）

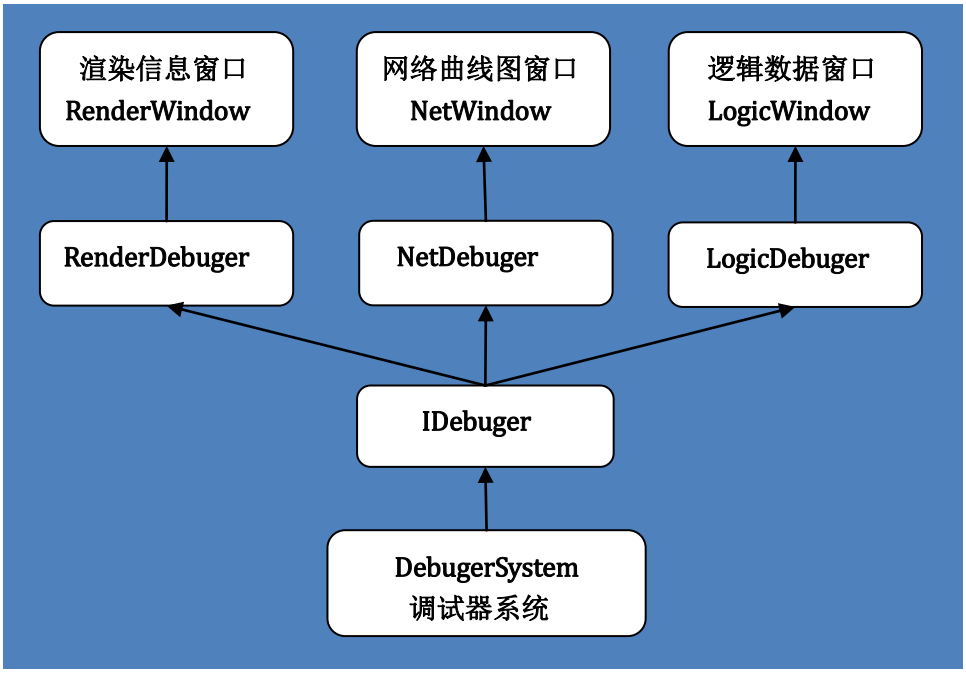
对于游戏的输入控制系统，因为 cocos2d-x 本身提供的 UI 控件都有其自身的输入响应机制，这部分我们很难直接去修改。所以，此处我们讨论的输入控制系统，主要就是针对 CCLayer 进行的 Touch 和 Accelerate 事件控制与管理。

在游戏中，为了对 Touch 和 Accelerate 事件进行统一的管理和处理，在整个客户端的窗口上，我们特意设计了一个最底层的 UILayer（也就是所有 UI 布局 Layout 的根窗口），整个游戏中，仅这个根窗口的 CCLayer 监听了 Touch 和 Accelerate 事件，其内部对两种事件进行捕获和处理，然后将对应的事件，存储在内部的输入消息队列中，通过消息事件，通知当前各个游戏管理器调度和处理。

2.10 调试器系统（DebugerSystem）

游戏过程中，我们需要时刻对游戏内的各项性能指标进行监控，判断各个模块，各个环节是否存在着重大的性能问题。所以，在我们客户端中，彻

底的将调试，监控作为一个核心模块来设计开发，目前主要分为网络模块调试器（NetDebuger），渲染模块调试器（RenderDebuger），逻辑模块调试器（LogicDebuger）三大模块（可根据需要动态增加）。对于 NetDebuger，主要监控当前网络上下行数据量，接受和发送的数据包个数，网络接受和发送的耗时信息等，并且通过曲线图展示；对于 RenderDebuger，主要用于监控当前客户端渲染的具体信息，例如 DIP 数量，像素填充率，渲染顶点数量等，开发者可以通过这些渲染引擎判断当前渲染是否存在性能问题；对于 LogicDebuger，管理和统计所有逻辑对象个数，大小，逻辑处理耗时等。具体如下图所示：



实际开发中，通过我们自行开发的游戏逻辑层调试器系统，在结合 XCODE 本身提供的强大监控工具，可以非常完善仔细的监控各个模块的详细数据，内存，处理耗时等信息，完全满足一般的开发监控需求。

三：引擎改进与公共代码库

游戏中的 UI 面板，难免会使用到模态窗口，但是 cocos2d-x 引擎并没有提供类似的功能，为了避免在逻辑层编写大量冗余代码来实现该功能，我们需要对 cocos2d-x 引擎进行必要的修改：在 cocos2d-x 消息事件处理（CCTouchDispatcher）内部，存在一个事件响应队列，对于所有关心 Touch 事件的对象，按照优先级从小到大排序，优先级越小，则越优先调度。所以，如果我们需要实现模态窗口的功能，需要自己管理所有 UI 控件的消息响应级别，并且按照控件之间的父子依赖关系，实现一个类似树的优先级结构，每次需要实现模态窗口时，只要确保其对应的事件优先级在最顶层，并且在处理完消息之后，屏蔽掉该消息，避免其继续传递到下面的窗口。

考虑到不同开发者对操作系统，底层接口等熟悉程度不同，为了降低开发成本，我们开发了一套基于 IOS 平台的基础代码库 cobra_ios，其内部封装了开发中常用的各种接口：整套完善的线程安全容器，如数组，单链表，双向循环列表，队列，二叉树等；各种线程控制模型和串行化接口；各种内存管理技术；数据解

包器 DPacket 和数据组包器 EPacket 等，在开发中，所有开发者都统一使用该基础库，使用相同的接口处理，方便开发并提升了开发效率。

四：拓展与其他

4.1 cocos2d-x VS Cocoa

上面也提到了，cocos2d-x 引擎本身没有提供太多的 UI 控件，除了开发者自行实现之外，我们还可以使用 IOS 标准的 UI 控件。熟悉 Win32，MFC 的朋友都知道，Win32 标准控件很难与 DirectX 结合，因为两者完全不是相同的渲染机制，但是在 IOS 平台下，cocos2d-x 与 Cocoa 自带的 UI 控件完全兼容，并且可以相互的调用，例如在游戏登陆界面，我们就可以使用 Cocoa 自带的 NSTextField 控件来实现账号和密码输入框。所以，学好 Cocoa，对于游戏开发者来说，很有必要。

4.2 其他

对于游戏脚本语言的选择，由于 IOS 平台不支持以动态链接库的方式使用第三方库，所以，我们不可能选择 boost+python 的方式进行脚本拓展，不过，我们还可以选择 Lua 等其他可行性比较高的动态语言。

“工欲善其事，必先利其器”，对于中大型的项目，除了开发客户端之外，一般还需要单独开发编辑器进行关卡，场景等编辑处理。对于编辑器的开发，由于引擎 cocos2d-x 本身完全跨平台，所以我们完全可以使用自己熟悉的语言和平台来开发，选择使用 Cocoa，MFC，C#等熟悉的语言，只要确保所有最终的关卡场景数据可被跨平台读取即可。

五：总结

移动时代，游戏的操作方式，已不仅仅限于我们传统的鼠标键盘模式，随着触摸，摄像，语音，重力感应等更多操作形式的普及，也为游戏带来了前所未有的机遇和挑战。本文仅以简单的游戏框架原型为例，结合目前流行的 cocos2d-x 引擎，讲述了一般游戏客户端的框架和具体实现。由于篇幅有限，无法涉及客户端开发中的方方面面，如果大家有更好的建议和想法，欢迎通过 guanzhongdaoke@gmail.com 和我联系。