软件哲学

我从 2005 年开始设计 Creature3D 游戏引擎，历时 4 年，积累了一些软件架构方面的经验。Creature3D 游戏引擎是一个功能全面、庞大的次世代 3D 游戏引擎，具备多线程优化、人工智能、物理仿真（ODE 动力学引擎）、以及次世代实时渲染技术等。在引擎设计过程中，我深深的感受到软件架构的重要性。曾经多次因为想到了更好的设计方案，或者应用过程中发现原先架构上的漏洞，而反复修改引擎核心。Creature3D 引擎的设计目标是扩展性、复用性、易用性、通用性和无限增益性，做到可以满足 3D 游戏乃至任何 3D 可视化视觉软件需求；做到基于 SDK 就可以具备完整的应用程序开发能力；做到游戏等应用开发与引擎底层无关；做到二次开发扩展的代码可被复用至使用 Creature3D 引擎的其它系统。于是我常常在思考软件的本质。在写这篇文章的时候，我对软件哲学有了更深的理解，我相信这将能帮助 Creature3D 引擎实现上述设计目标。

1.软件的本质：

软件的本质是数据与操作。任何软件都可以被分解为数据与操作的集合。数据与操作结合的紧密程度将直接影响到软件的扩展性和复用性。由于我们在做软件设计的时候大都都是从需求出发进行设计，为了具有好的可扩展性，我们通常需要很努力的去寻找需求的本质，抽象需求。但是不管我们设计的有多好，软件只能适应这一类需求的扩展。当需求彻底改变的时候，我们的软件也就无法适应了。需求的本质对于软件的本质来讲还是太具体化了。我们无法用 A 需求的本质来满足 B 需求的要求。但是如果我们从软件的本质进行设计，那么从理论上，软件的本质将能满足各种软件的需求。我们知道软件的本质就是数据与操作，软件本身就是数据和操作的集合。那么在设计过程中，我们该如何应用呢。

2.关于数据：

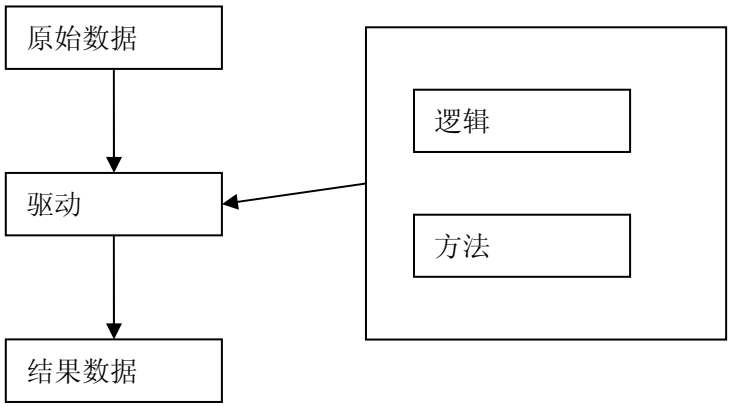
对于数据，按照数据的类型分，可以分为原始数据（输入数据）、过程数据（中间数据）、结果数据（输出数据）。我们需要关心的是原始数据和结果数据，原始数据通常需要被设计为文件保存或者录入数据库。而结果数据则是我们需要得到的数据。可以看出原始数据和结果数据是可以做到与操作分离的。而过程数据则是与操作紧密相连的。但是过程数据却是我们并不关心的数据。所以我们可以抽象原始数据和结果数据。使之从操作体里完全独立出来。原始数据的独立首先就能让我们感受到数据共享的好处，并且很多时候为了获得我们需要的结果数据，我们只需要通过修改原始数据，而无须修改软件本身。原始数据的分离，是非常重要的，也是目前我们在设计里面经常应用的。

过程数据（中间数据）则是与操作体本身密不可分的。也就是说，中间数据寄存器将是操作体不可分离的一部分。

3.关于操作：

对于操作的分析是最为困难的。软件体的操作异常多变与复杂。我们以 C++为例，任何成员函数、静态函数、全局函数、以及预定义函数等都可以归类为操作。但就其本质而言，操作可以分为三类，就是方法、逻辑和驱动。方法是具体的操作；逻辑是抽象的操作；驱动是输入输出的操作，是操作的发起者，也是数据与逻辑以及方法的桥接器。从概念上可以看出方法和逻辑是可以与原始数据以及结果数据完全分离的，只有驱动才是与原始数据以及结果数据相关的。

4. 依赖关系图：



5. 实验程序：

我们以 1+2=3 为例。其中 1 和 2 是输入数据，3 是输出数据。+和=是具体的操作，归为方法。而 1+2=这个表达式，则是抽象的操作，归为逻辑，这个逻辑用到了中间数据。驱动则是执行这个表达式的操作，现在我们尝试将这个思想用程序语言来描述。

```
/*************************************************************************
*    HandleTest
*
*    CREATED BY:     吴财华2009.4.3
*
*    DESCRIPTION: 用数据、方法、逻辑、驱动概念编写+2=3
*
*    HISTORY:      吴财华2009.4.3
*
*************************************************************************/
#include <stdio.h>
#include <map>
#include <string>
#define HandleClass(Lib,Class) virtual const char *className() { return #Class; }\
     virtual const char *libName(){ return #Lib; }\
     virtual Handle *clone() { return new Lib::Class; }
#define DataClass(Lib,Class) virtual const char *className() { return #Class; }\
     virtual const char *libName(){ return #Lib; }\
     virtual Data *clone() { return new Lib::Class; }
namespace Core {

     class Data;
     class Handle
     {
     public:
```

```
        Handle(){}
          virtual const char *className() { return "Handle"; }
          virtual const char *libName(){ return "Core"; }
          virtual Handle *clone() { return NULL; }

          virtual void operator()(Data &data)=0;
          virtual void operator()(Handle &handle)=0;
          virtual void inputParam(int i, void *param)=0;
          virtual void outputParam(int i, void *param)=0;
          virtual void getOutputParam(int i, void*& param)=0;

          virtual void addParam(int i, const std::string& str)=0;
    };
    class HandleManager
    {
    public:
          static HandleManager *getInstance(){ static HandleManager s_handleManager; return
&s_handleManager; }
          typedef std::map<std::string, Handle*> HandleMap;
          void RegisterHandle(Handle *handle)
          {
               m_handleMap.insert(std::make_pair(handle->className(),handle));
          }
          Handle *getHandle(const std::string &name) { HandleMap::iterator itr =
m_handleMap.find(name); return itr!=m_handleMap.end()?itr->second->clone():NULL; }
          ~HandleManager()
          {
               for( HandleMap::iterator itr = m_handleMap.begin();
                    itr != m_handleMap.end();
                    ++itr )
               {
                    delete itr->second;
               }
               m_handleMap.clear();
          }
    protected:
          HandleMap m_handleMap;
    };

    class Data
    {
    public:
          virtual const char *className() { return "Data"; }
          virtual const char *libName(){ return "Core"; }
```

```
        virtual Data *clone() { return NULL; }

        typedef std::map<int, Handle*> HandleMap;

        void insertHandle(int msg,Handle *handle)
        {
            m_handleMap.insert(std::make_pair(msg,handle));
        }
        Handle *getHandle(int msg) { HandleMap::iterator itr = m_handleMap.find(msg); return
itr!=m_handleMap.end()?itr->second:NULL; }
        void excHandle(int msg){ excHandle(getHandle(msg)); }
        void excHandle(Handle *handle)
        {
          if(handle)
            {
                (*handle)(*this);
            }
        }

        virtual void addParam(int i, const std::string& str)=0;

        ~Data()
        {
            for( HandleMap::iterator itr = m_handleMap.begin();
                itr != m_handleMap.end();
                ++itr )
            {
                delete itr->second;
            }
            m_handleMap.clear();
        }
    protected:
        HandleMap m_handleMap;
    };

    class Method : public Handle
    {
    public:
        virtual const char *className() { return "Method"; }
        virtual const char *libName(){ return "Core"; }
        virtual Handle *clone() { return NULL; }

        virtual void operator()(Handle &handle)=0;
        virtual void inputParam(int i, void *param) = 0;
```

```
        virtual void addParam(int i, const std::string& str){}
    private:
        virtual void operator()(Data &data){}
        virtual void outputParam(int i, void *param){}
        virtual void getOutputParam(int i, void*& param){}
    };
    class Logic : public Handle
    {
    public:
        virtual const char *className() { return "Logic"; }
        virtual const char *libName(){ return "Core"; }
        virtual Handle *clone() { return NULL; }

        virtual void operator()(Handle &handle)=0;
        virtual void inputParam(int i, void *param) = 0;
        virtual void outputParam(int i, void *param) = 0;
        virtual void addParam(int i, const std::string& str){}
    private:
        virtual void operator()(Data &data){}
        virtual void getOutputParam(int i, void*& param){}
    };
    class Drive : public Handle
    {
    public:
        virtual const char *className() { return "Drive"; }
        virtual const char *libName(){ return "Core"; }
        virtual Handle *clone() { return NULL; }

        virtual void operator()(Data &data)=0;
        virtual void outputParam(int i, void *param) = 0;
        virtual void getOutputParam(int i, void*& param) = 0;
        virtual void inputParam(int i, void *param){}
        virtual void addParam(int i, const std::string& str){}
    private:
        virtual void operator()(Handle &handle){}
    };
    class Add : public Method
    {
    public:
        HandleClass(Core,Add)
        virtual void operator()(Handle &handle)
        {
            int result = m_param1+m_param2;
            handle.outputParam(0,&result);
```

```
        }
        virtual void inputParam(int i, void *param)
        {
            switch (i)
            {
            case 0:
                m_param1 = *((int*)param);
                break;
            case 1:
                m_param2 = *((int*)param);
                break;
            }
        }
    protected:
        int m_param1;
        int m_param2;
    };
    class Equal : public Method
    {
    public:
        HandleClass(Core,Equal)
        virtual void operator()(Handle &handle)
        {
            handle.outputParam(0,&m_param1);
        }
        virtual void inputParam(int i, void *param)
        {
            if(i == 0) m_param1 = *((int*)param);
        }
    protected:
        int m_param1;
    };

    class DataTest : public Data
    {
    public:
        DataClass(Core,DataTest)
        virtual void addParam(int i, const std::string& str)
        {
            switch (i)
            {
            case 0:
                m_value1 = atoi(str.c_str());
                break;
```

```
            case 1:
                m_value2 = atoi(str.c_str());
                break;
            }
        }
        int getValue1() { return m_value1; }
        int getValue2() { return m_value2; }
    protected:
        int m_value1;
        int m_value2;
    };

    class LogicTest : public Logic
    {
    public:
        ~LogicTest()
        {
            if(m_add) delete m_add;
            if(m_equal) delete m_equal;
        }
        HandleClass(Core,LogicTest)
        virtual void addParam(int i, const std::string &str)
        {
            switch (i)
            {
            case 0:
                m_add = HandleManager::getInstance()->getHandle(str);
                break;
            case 1:
                m_equal = HandleManager::getInstance()->getHandle(str);
                break;
            }
        }
        virtual void inputParam(int i, void *param)
        {
            switch (i)
            {
            case 0:
                m_input1 = *((int*)param);
                break;
            case 1:
                m_input2 = *((int*)param);
                break;
            }
```

```
        }
        virtual void outputParam(int i, void *param)
        {
            switch (i)
            {
            case 0:
                m_result = *((int*)param);
                break;
            }
        }
        virtual void operator()(Handle &handle)
        {
            if(m_add && m_equal)
            {
                m_add->inputParam(0,&m_input1);
                m_add->inputParam(1,&m_input2);
                (*m_add)(*this);
                m_equal->inputParam(0,&m_result);
                (*m_equal)(*this);

                handle.outputParam(0,&m_result);
            }
        }
    protected:
        int m_input1;
        int m_input2;
        Handle *m_add;
        Handle *m_equal;

        int m_result;
    };

    class DriveTest : public Drive
    {
    public:
        HandleClass(Core,DriveTest)

        virtual void inputParam(int i, void *param)
        {
            switch (i)
            {
            case 0:
                m_handle = (Handle*)param;
                break;
```

```
            }
        }
        virtual void operator()(Data &data)
        {
            DataTest *dataTest = dynamic_cast<DataTest *>(&data);
          if(m_handle && dataTest)
            {
                int value1 = dataTest->getValue1();
                int value2 = dataTest->getValue2();
                m_handle->inputParam(0,&value1);
                m_handle->inputParam(1,&value2);
                (*m_handle)(*this);
            }
        }

        virtual void outputParam(int i, void *param)
        {
            switch (i)
            {
            case 0:
                m_result = *((int*)param);
                break;
            }
        }
        virtual void getOutputParam(int i, void*& param)
        {
            if(i==0)
            {
                param = &m_result;
            }
        }
    protected:
        int m_result;
        Handle *m_handle;
    };

    class DriveTest2 : public Drive
    {
    public:
        DriveTest2():m_inputData(0){}
        HandleClass(Core,DriveTest2)

        virtual void operator()(Data &data)
        {
```

```
        if(m_handle)
        {
            int result1 = 0;
            int result2 = 0;
            DataTest *dataTest = dynamic_cast<DataTest *>(&data);
            if(dataTest)
            {
                int value1 = dataTest->getValue1();
                int value2 = dataTest->getValue2();
                m_handle->inputParam(0,&value1);
                m_handle->inputParam(1,&value2);
                (*m_handle)(*this);
                result1 = m_result;
            }
            if(m_inputData)
            {
                int value1 = m_inputData->getValue1();
                int value2 = m_inputData->getValue2();
                m_handle->inputParam(0,&value1);
                m_handle->inputParam(1,&value2);
                (*m_handle)(*this);
                result2 = m_result;
            }

            m_handle->inputParam(0,&result1);
            m_handle->inputParam(1,&result2);
            (*m_handle)(*this);
        }
    }

    virtual void outputParam(int i, void *param)
    {
        switch (i)
        {
        case 0:
            m_result = *((int*)param);
            break;
        }
    }
    virtual void getOutputParam(int i, void*& param)
    {
        if(i==0)
        {
            param = &m_result;
```

```
            }
        }
        virtual void inputParam(int i, void *param)
        {
            switch (i)
            {
            case 0:
                m_handle = (Handle*)param;
                break;
            case 1:
                m_inputData = (DataTest*)param;
                break;
            }
        }
    protected:
        DataTest* m_inputData;
        int m_result;
        Handle *m_handle;
    };

}

using namespace Core;

int main()
{//1+2=3
    HandleManager::getInstance()->RegisterHandle(new Add);
    HandleManager::getInstance()->RegisterHandle(new Equal);
    HandleManager::getInstance()->RegisterHandle(new LogicTest);
    HandleManager::getInstance()->RegisterHandle(new DriveTest);
    HandleManager::getInstance()->RegisterHandle(new DriveTest2);

    DataTest myData;
    myData.addParam(0,"1");
    myData.addParam(1,"2");

    Handle *myDrive = HandleManager::getInstance()->getHandle("DriveTest");
    Handle *myLogic = HandleManager::getInstance()->getHandle("LogicTest");
    myLogic->addParam(0,"Add");
    myLogic->addParam(1,"Equal");
    myDrive->inputParam(0,myLogic);
    int msg = 1;
    myData.insertHandle(msg,myDrive);
    myData.excHandle(msg);
```

```
    void *_result;
    myDrive->getOutputParam(0,_result);
    int result = *((int*)_result);

    DataTest myData2;
    myData2.addParam(0,"3");
    myData2.addParam(1,"4");
    Handle *myDrive2 = HandleManager::getInstance()->getHandle("DriveTest2");
    myDrive2->inputParam(0,myLogic);
   myDrive2->inputParam(1,&myData2);
    myData.excHandle(myDrive2);

    myDrive2->getOutputParam(0,_result);
    result = *((int*)_result);

    delete myDrive2;
}
////////////////////////////////////////////////////////////////////////
/*
结论：
    1.软件最本质的定义是数据和操作。
    2.对于数据，我们可以分为输入数据、中间数据、输出数据，也可以分为原始数据、过程数据、结果
      数据。我们需要关心的是原始数据和结果数据。我们用Data类抽象原始数据。
    3.对于操作，我们将其分解为方法、逻辑和驱动。
    4.方法是具体的操作，我们用Method类描述，程序里的例子有Add、Equal，它们都是具体的处理单元。
    5.逻辑是抽象的操作，我们用Logic类描述，程序里的例子有LogicTest，它们封装了逻辑层的操作，
      我们看到LogicTest通过组合Add、Equal这两个方法来实现我们需要的逻辑。
    6.驱动是输入输出的操作，我们用Driver类描述，程序里的例子有DriveTest，DriveTest2。它们直接
      与源数据和输出数据联系。我们可以看到通过驱动层我们将逻辑层与数据层进行了分离。
    7.从以上的设计我们可以做到数据层、逻辑层、方法层具有很好的扩展性、复用性以及通用性和易用
      性。例如当数据层或者逻辑层接口有所变化的时候，我们只需要更新驱动层就可以适应。
    8.原始数据层、驱动层、逻辑层、方法层都是由相对独立的个体组成，局部个体的增加、修改或着删
      除，都不会影响到系统的其它部分。它们在后续的开发过程中皆可跟随需求而任意扩充。并且新增
      的元素皆是对系统的增益，可实现不同需求间的复用。所以我们需要一些管理器来负责管理这些增
      益单元。例如HandleManager，并且我们还可以增加一个DataManager。考虑到系统无限增益的可能。
      我们需要注意HandleManager、DataManager的Map数据查找速度。避免在对速度要求高的情况下从
      HandleManager或DataManager创建实例，在无法回避的情况下，可以设计一些专用的高速对象缓存
      区来实现。
*/
////////////////////////////////////////////////////////////////////////
```

6.关于方法、逻辑与驱动：

在这里有两点问题需要解答，尽管它们非常抽象难以解释。第一点是方法、逻辑、驱

动三者是否能充分表示操作，第二点是将操作分解为方法、逻辑、驱动的意义。

从上文的依赖关系图和实验程序，我们可以了解到。将驱动从操作体里分离出来有助于数据和操作体其它部分的分离。原始数据和结果数据直接与驱动层联系，而操作体其它部分只与过程数据联系，它们只需要具备过程数据寄存器就可以完成工作。所以驱动层的分离对于软件结构来说是很有意义的。在脱离了驱动层后，剩下的操作体就是非常独立的数据处理器了。如果将它们当成是硬件集成电路。那么驱动层就是它们的 IO 单元。那么我们如何来思考集成电路呢？首先，数据需要经过一系列具体的算法来处理。所以集成电路内应当包含具体的算法处理单元，比如实验程序里面的 Add 单元以及 Equal 单元。然后，有了这些基本处理单元后，我们还需要有一层特殊的硬件体对其进行有组织有顺序的调度，以达到我们需要的数据处理结果。而这一层特殊硬件体，我们将其归纳为逻辑。也就是说逻辑具备组织和调度方法的能力。于是，剩余的操作体我们将其分类为逻辑和方法。在做数据处理的时候中间数据是普遍存在的。所以操作体里必然涉及到中间数据寄存器的问题。从上文的讨论以及实验程序上，这些中间数据寄存器普遍存在与驱动、逻辑以及方法体内，而并没有独立设计一个寄存器单元。独立的中间数据寄存器单元对于软件来讲，只会增加复杂度，降低执行效率，如果真的要这么做，那么我们还需要为其制定索引，这是完全没有必要的。

至此我们说明了分离出驱动层的意义，以及除去驱动剩余的操作体就是方法和逻辑。也就是说，驱动、方法、逻辑可以充分表示操作。那么接下来我们来分析一下方法与逻辑分离的意义。我们知道方法是具体的算法处理单元，而逻辑则是负责方法的组织和调度的。逻辑和方法的分离则可以给程序带来更大的灵活性。而从实验程序上，我们还可以看出：

1. 驱动、方法、逻辑具有很大的相通性，在使用过程中可以根据需要进行互相转换。
2. 每个操作体都可以作为一个独立的处理单元使用，它们具备的完整的输入输出以及执行接口。
3. 驱动、方法、逻辑从操作继承，但它们并没有特殊的功能扩展，它们都可以作为一个完整的独立的操作单元。它们只是在逻辑意义上被分派为负责不同功能，但它们的本质描述仍然等同于操作。
4. 我们可以通过组合多种通用操作来实现一个复杂的任务，而无须增加一个专用的复杂操作。

7. 后记：

我们现在对软件的本质有了清晰的认识，而且上文的实验程序，也给我们提供了设计方法。然而在真正应用过程中，我们还需要把握好设计的粒度问题。从本质处抽象的设计方法粒度过细，在没有必要的情况下应用反而会成为系统的累赘。我们需要认识到并不是所有操作都需要做到和数据分离，原始数据对象里也可以包含一些必要的操作。其中软件设计模式以及其他设计经验都给我们提供了很多很好的软件设计方法。理解了软件的本质，在设计过程中，我们需要做到不拘泥于方法，做到灵活应用，合理应用。

8. 关于我：

吴财华，2005 年至 2009 年设计完成 Creature3D（灵兽）游戏引擎，并取得著作权。
我的博客：http://blog.sina.com.cn/creature3d