前序

[**探究osg中的程序设计模式---开篇**](http://www.3wwang.cn/html/article_81.html)

探究osg中的程序设计模式---创造性模式

### [探究osg中的程序设计模式---创造型模式---Factory(工厂)模式](http://www.3wwang.cn/html/article_83.html)

### [探究osg中的程序设计模式---创造型模式---AbstractFactory(抽象工厂)模式](http://www.3wwang.cn/html/article_84.html)

### [探究osg中的程序设计模式---创造型模式---Singleton(单例)模式](http://www.3wwang.cn/html/article_86.html)

### [探究osg中的程序设计模式---创造型模式---Builder(创造者)模式](http://www.3wwang.cn/html/article_87.html)

### [探究osg中的程序设计模式---创造型模式---Prototype(原型)模式](http://www.3wwang.cn/html/article_89.html)

### [总结：创造型设计模式](http://www.3wwang.cn/html/article_90.html)

探究osg中的程序设计模式---结构性模式

### [探究osg中的程序设计模式---结构性模式---Bridge(桥接)模式](http://www.3wwang.cn/html/article_91.html)

### [探究osg中的程序设计模式---结构性模式---Adapter(适配器)模式](http://www.3wwang.cn/html/article_92.html)

### [探究osg中的程序设计模式---结构型设计模式--- Decorator (装饰器)模式](http://www.3wwang.cn/html/article_94.html)

### [探究osg中的程序设计模式---结构型设计模式--- Composite(组合)模式](http://www.3wwang.cn/html/article_95.html)

### [探究osg中的程序设计模式---结构型设计模式---Flyweigh(享元)模式](http://www.3wwang.cn/html/article_96.html)

### [探究osg中的程序设计模式---结构型设计模式---Facade(外观)模式](http://www.3wwang.cn/html/article_97.html)

### [探究osg中的程序设计模式---结构型设计模式---Proxy(代理)模式](http://www.3wwang.cn/html/article_98.html)

[总结：结构型设计模式](http://www.3wwang.cn/html/article_99.html)

探究osg中的程序设计模式---行为性模式

我们通过一个月的时间完成了对osg每一帧所要执行动作的探究，使得我们从整体上了解了这个强大的三维渲染引擎。但是要想真正的用好这个渲染引擎我们还有一个步骤也是不可或缺的--- osg中的设计模式。它保证了osg的灵活多变（多种平台支持windows，linux，MacOS，Android等等）以及稳定运行。

什么是设计模式

鲁迅曾经说过“世上本没有路，走的人多了便成了路”。这句话就是所有的设计模式的来源。也就是说设计模式是前辈们一代代的实践，然后总结出来的一套增加代码的可复用性，可维护性的程序设计经验。

为什么学设计模式

设计模式可以说是软件设计的基石、脉络，类似osg这种代码框架，从根本上就是定义了一组可以复用的相互协作的类，这样使得我们通常使用的时候可以专注于应用本身去实现所需的特定功能。但是我们想要达到随用随取，来去自如的状态必须要理解它内容的实现机制。所以我们必须要了解设计模式。

怎么学设计模式

设计模式是通用的(面向对象语言)，如果只是看类似《GoF 23种设计模式》的书籍，会显得很枯燥乏味，并且缺乏实践性的理解。正好我们目前在研究osg渲染引擎，它既然可以被全世界的三维爱好者推崇，那么它肯定完美的把设计模式的思想融入其中。

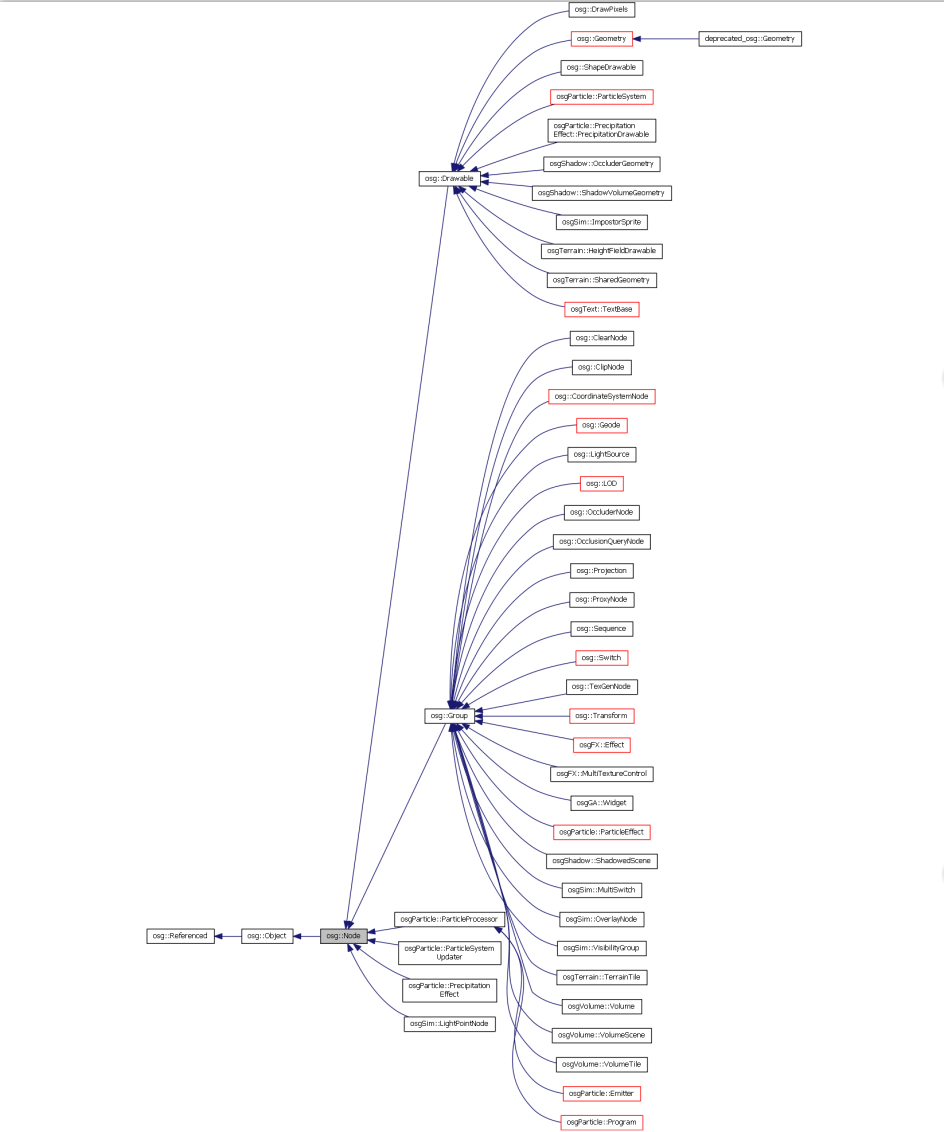
所以就有了这么一段旅程，我本身也研读过一段时间的《java设计模式》，但是只对经常用到的一些比较熟悉（例如：单例模式、工厂模式、抽象工厂模式、原型模式等）。如果您对程序员的发展路线有过研究，就会了解要想走上架构师之路，必须要有一个好的程序设计的思想，了解到这些当时就有了要深入研究设计模式的想法。正好趁着这个机会去深入的探究一下设计模式在软件生产过程的应用。

设计模式六大原则

我们这里所要讲解的设计模式特指面向对象语言。所以我们先来简单的介绍一下设计模式的六大基本原则。我们的先知们就是在这六大基本原则的基础上总结出了23中设计模式。基本原则只是‘原则’不是‘法则’，所以我们要灵活的应用，不要生搬硬套。想要更加完美的运用这些，唯一高效的方法就是研究一个成名的、成熟的框架。

1、单一职责原则

一个类只负责一个具体功能。我们现在可以随意的打开一个OSG Core中的类。例如我们经常用到的osg::Node类。



从上面的osg::Node的类架构图，我们可以看出，Node类是一个中间类，他集成自osg：：Object赋予智能指针的特性，同时它又是osg中所有生命体的父节点。虽然它的作用非常的重要，但是他的类的功能很简单就是提供一些公用的接口为他的所有的孩子。试想一下如果我们把所有的osg中的生命体的功能都集中在osg::Node类中，他会是多么的庞大、冗杂，大大减小了其复用性。

2、开闭原则

应用程序中的一个模块，应当对扩展开放，而对修改关闭。还是用osg::Node类为基础进行介绍。

例如我们在开发一款针对天气模拟的三维设计软件，为了在场景中实现”雨”这种生命体。有一种解决办法就是重新设计一个osgWeather::Rain类继承自osg::Node，在其中创建一个特定的Drawable，附加到其上。这就是我们要遵循的开闭原则。

3、里氏替换原则

如果程序中调用父类可以实现的功能，调用子类同样也可以完成。也就是子类重写virtual方法时，子类中方法的访问权限不能比父类的小。同样的我们还是用osg::Node来介绍里氏替换原则。

例如我们需要通过setUpdateCallback(Callback\* nc)函数来设定这个node的更新回调类TempUpdateCallBack，在osg的更新遍历的时候去执行TempUpdateCallBack中operator()函数。这里的TempUpdateCallBack就是osg::NodeCallback子类，而osg::NodeCallback又是osg:: Callback的子类。所以要保证TempUpdateCallBack的operator()函数的访问类型必须是public的，这样node的setUpdateCallback函数使用TempUpdateCallBack才是可以完成的。

4、依赖倒转原则

我们在传递参数或者关联关系的时候，应当尽量使用最高层次的抽象接口进行变量类型声明、参数类型声明、方法返回类型声明，以及数据类型的转换等，而不要用具体类来做这些事情。并且类中声明方法的时候，仅且只有一个使用接口或抽象类声明的方法。还是在osg::Node来介绍依赖倒转原则。

setUpdateCallback(Callback\* nc)函数，osg在定义这个函数的时候就是使用了最上层的，且不影响函数运行的抽象类osg::Callback。

5、接口隔离原则

使用多个专门的接口，而不是一个总的大的接口。也就是随着程序的不断壮大，接口中的内容也会变得非常的庞大。这时我们需要将它分割成一些更细小的接口，每个接口都要对应一种并且只有一种角色。虽然c++支持多继承(父类)，但是我们不提倡多继承，应当替换成多实(接口)。

这个我们就不进行代码讲解，因为osg中存在很多多继承的情况。

6、迪米特原则

一个软件实体应当尽可能少地与其他实体发生相互作用。对于一个对象可以发生作用的包括以下几种“朋友”：

(1) 当前对象本身(this)；

(2) 以参数形式传入到当前对象方法中的对象；

(3) 当前对象的成员对象；

(4) 如果当前对象的成员对象是一个集合，那么集合中的元素也都是朋友；

(5) 当前对象所创建的对象。

这个原则告诉我们应该尽量减少对象之间的交互，如果两个对象之间不必彼此直接通信，那么这两个对象就不应当发生任何直接的相互作用，如果其中的一个对象需要调用另一个对象的某一个方法的话，可以通过第三者转发这个调用。简言之，就是通过引入一个合理的第三者来降低现有对象之间的耦合度。

今天我们就开始介绍osg在框架设计的使用用到的设计模式，探索到每个设计模式的时候我都会尽量多的去从osg中找到对应的代码，这样方便大家理解设计模式的思想，以及osg在设计框架时候的思想。

创造型模式---Factory(工厂)模式

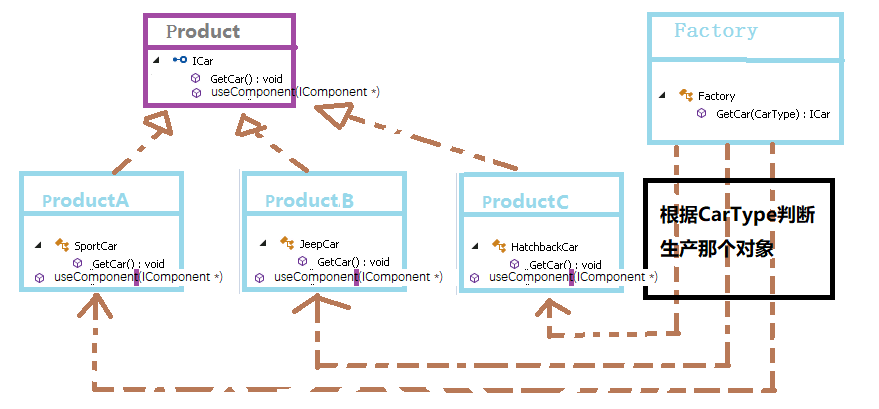
什么情况下会需要Factory模式

情景1、我们开发系统的时候经常会写一些大体功能相同但是内容有一部分处理方法不同的类，这时为了实现整体代码的高内聚低耦合的原则，就需要提取出这些类的共有方法组成一个抽象接口或者基类。在使用的时候需要先声明一个指向基类的指针，而这个指针的内容是需要通过(new xxx())的方法来构造一个子类。虽然这样符合面向对象中多态的特性，但是他也会带来一些问题。(程序员必须知道实际的子类的名称，才可以初始化一个子类对象。但是子类的名称会根据不同程序员的不同习惯来命名。)

情景2、我们在复杂系统的设计过程中，会遇到在一个父类A中用到一个抽象类B指针，来实现一些特定的功能。但是按照上面的new b()的方法来初始化一个抽象类B的实现类(子类b)赋给这个抽象类指针B，就要求父类A必须知道到底要调用那个抽象类B的实现类(子类有b1，b2。。。)。而实际的环境是父类A不清楚具体使用那个那个抽象类B的实现类，但是父类A的子类a知道类b是要调用的那个抽象类B的实现类。

由于上面两种情况，我们才会使用到Factory设计模式。(由于工厂模式在osg中基本没有用到，不过在osgEarth中是有所体现的GeometryFactory。所以在这里我们就不使用osg中的代码讲解工厂模式)

针对情景1、可以用以下代码实现方式



所要生产的对象的抽象接口(汽车接口)

namespace 3WWANG

{

/// <summary>

/// 抽象产品类： 汽车

/// </summary>

public interface ICar

{

void GetCar();

//void UseComponent(IComponent \*);

}

}

所要生产的对象的具体实现(汽车子类)

namespace 3WWANG

{

public enum CarType

{

SportCarType = 0,

JeepCarType = 1,

HatchbackCarType = 2

}

/// <summary>

/// 具体产品类： 跑车

/// </summary>

public class SportCar : ICar

{

public void GetCar()

{

Console.WriteLine("跑车");

}

//public void UseComponent(IComponent \*)

//{

//Console.WriteLine("跑车车窗");

//}

}

/// <summary>

/// 具体产品类： 越野车

/// </summary>

public class JeepCar : ICar

{

public void GetCar()

{

Console.WriteLine("越野车");

}

//public void UseComponent(IComponent \*)

//{

//Console.WriteLine("越野车车窗");

//}

}

/// <summary>

/// 具体产品类： 两箱车

/// </summary>

public class HatchbackCar : ICar

{

public void GetCar()

{

Console.WriteLine("两箱车");

}

//public void UseComponent(IComponent \*)

//{

//Console.WriteLine("两箱车车窗");

//}

}

}

生产环境

namespace 3WWANG

{

public class Factory

{

public ICar GetCar(CarType carType)

{

switch (carType)

{

case CarType.SportCarType:

return new SportCar();

case CarType.JeepCarType:

return new JeepCar();

case CarType.HatchbackCarType:

return new HatchbackCar();

default:

throw new Exception("牛车");

}

}

}

}

工厂模式的使用

namespace 3WWANG

{

using System;

class Program

{

static void Main(string[] args)

{

ICar car;

try

{

Factory factory = new Factory();

car = factory.GetCar(CarType.SportCarType);

car.GetCar();

}

catch (Exception ex)

{

Console.WriteLine(ex.Message);

}

}

}

}

针对情景2、根据上面代码中的产品和工厂的代码。我们来假设一个应用场景。汽车的零件分配。汽车的零件肯定是可以抽取成一个基类的。在基类中需要指定共有的方法就是这个零件用于那辆汽车。但是零件的基类是不清楚她下面的子类的零件到底应该用于那辆汽车。这时就需要用到工厂模式。

把上面代码中的useComponent的注释解开

零件基类

namespace 3WWANG

{

public interface IComponent

{

void UseCar(Factory\* factory);

}

}

车窗零件类

namespace 3WWANG

{

/// <summary>

/// 零件： 车窗

/// </summary>

public class ComponentWindow : IComponent

{

public void UseCar(Factory\* factory)

{

ICar\* iCar\* = factory.GetCar(CarType.SportCarType);

iCar->useComponent(this);

}

}

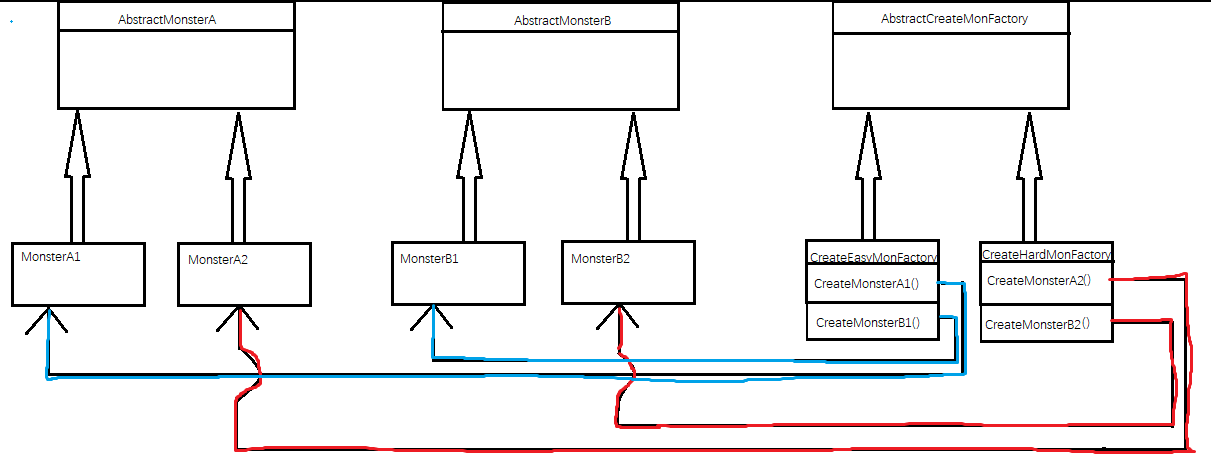
}

可以看出，Factory 模式对于对象的创建给予开发人员提供了很好的实现策略，但是Factory 模式仅仅局限于一类类（就是说 Product 是一类，有一个共同的基类），如果我们要为不同类的类提供一个对象创建的接口，那就要用 AbstractFactory 了

对于初学设计模式来说，Factory(工厂)模式和AbstractFactory(抽象工厂)模式比较难以区分。我们来举个栗子说明一下什么时候会用到抽象工厂模式。

当我们在使用osg创建一个简单的单机游戏，游戏的内容就是打怪升级。游戏里存在各种各样的怪物，当然怪物的难易程度会直接影响玩家的体验。所以我们设定了三种玩家等级：初级、中级和高级，每个玩家等级就会对应不同等级的怪物。我们在编写这个游戏程序的时候首先会提取相似怪物类(人形怪物)为一个基类（接口），然后根据玩家是初级、中级还是高级分别创建不同等级的怪物。这种情况下我们一般会创建很多很多的怪物对象，这样特别不方便管理，一不小心就会犯错（把高级的怪物付给了初级的玩家）。

针对上面提出的场景，先知们总结出了AbstractFactory 模式，AbstractFactory 模式就是用来解决这类问题---要创建一组相关或者相互依赖的对象。



AbstractFactory 模式关键就是将这一组对象的创建封装到一个用于创建对象的类（ConcreteFactory）中，维护这样一个创建类总比维护 n 多相关对象的创建过程要简单

AbstractFactory代码展示

namespace 3WWANG

{

/// <summary>

/// 抽象怪物A类

/// </summary>

public interface AbstactMonsterA

{

}

}

namespace 3WWANG

{

/// <summary>

/// 抽象怪物B类

/// </summary>

public interface AbstactMonsterB

{

}

}

namespace 3WWANG

{

/// <summary>

/// 怪物A1类

/// </summary>

public class MonsterA1 : AbstactMonsterA

{

MonsterA1()

{

Console.WriteLine("普通怪物A");

}

}

/// <summary>

/// 怪物A2类

/// </summary>

public class MonsterA2 : AbstactMonsterA

{

MonsterA2()

{

Console.WriteLine("高级怪物A");

}

}

}

namespace 3WWANG

{

/// <summary>

/// 怪物B1类

/// </summary>

public class MonsterB1 : AbstactMonsterB

{

MonsterB1()

{

Console.WriteLine("普通怪物B");

}

}

/// <summary>

/// 怪物B2类

/// </summary>

public class MonsterB2 : AbstactMonsterB

{

MonsterB2()

{

Console.WriteLine("高级怪物B");

}

}

}

namespace 3WWANG

{

/// <summary>

/// 抽象工厂

/// </summary>

public class AbstractCreateMonFactory

{

public:

virtual ~AbstractCreateMonFactory();

virtual AbstactMonsterA\* CreateMonsterA() = 0;

virtual AbstactMonsterB\* CreateMonsterB() = 0;

protected:

AbstractCreateMonFactory();

}

/// <summary>

/// 抽象工厂:创建简单模式的怪物

/// </summary>

class ConcreteEasyMonFactory:public AbstractCreateMonFactory

{

public:

ConcreteEasyMonFactory(){}

~ConcreteEasyMonFactory(){}

AbstactMonsterA\* CreateMonsterA()

{

return new MonsterA1();

}

AbstactMonsterB\* CreateMonsterB()

{

return new MonsterB1();

}

protected:

private:

};

/// <summary>

/// 抽象工厂:创建困难模式的怪物

/// </summary>

class ConcreteHardMonFactory:public AbstractCreateMonFactory

{

public:

ConcreteEasyMonFactory(){}

~ConcreteEasyMonFactory(){}

AbstactMonsterA\* CreateMonsterA()

{

return new MonsterA2();

}

AbstactMonsterB\* CreateMonsterB()

{

return new MonsterB2();

}

protected:

private:

};

}

/// <summary>

/// 抽象工厂使用

/// </summary>

namespace 3WWANG

{

using System;

int main(int argc,char\* argv[])

{

AbstractCreateMonFactory\* cf1 = new ConcreteEasyMonFactory();

cf1->CreateMonsterA();

cf1->CreateMonsterB();

AbstractCreateMonFactory\* cf2 = new ConcreteHardMonFactory();

cf2->CreateMonsterA();

cf2->CreateMonsterB();

return 0;

}

}

那我们结合上一节的内容，就可以清楚的明白工厂模式的创建过程，以及什么使用抽象工厂模式。实际上，AbstractFactory 模式是为创建 一组 （有多类）相关或依赖的对象提供创建接口，而 Factory 模式正如我在相应的文档中分析的是为 一类对象提供创建接口或延迟对象的创建到子类中实现。

Singleton(单例)模式是设计模式中普及最广的，因为它是面试官(特别是java面试官)最喜欢提问的问题；并且它所解决的问题十分常见就是常见一个并且是整个应用唯一的对象。一般情况下我们是怎么创建一个唯一的对象(变量)呢？分为一下三种情况。

1、类似c语言这种面向过程的语言设计中，我们可以通过创建一个唯一的全局变量，使用的时候包含这个头文件，然后声明一下就可以使用了。

2、类似java语言这种纯粹的面向对象的语言设计中，我们只能通过单例设计模式来创建一个唯一的全局变量。

3、类似c++语言这种面向对象和面向过程结合的语言设计中，我们可以使用上面两种方式来创建一个唯一的全局变量。

讲解工厂模式(上两节)时，没有使用到osg内部的类来说明(提醒一下工厂模式在osgearthSymbology:: GeometryFactory中使用到)，单例模式在osg原生代码中肯定会用到。我们这就进入osg的src文件查找一下。

osgDB::Registry类就是一个正宗的单例类。我可以在帮助回忆一下registry类是用来做什么，osgDB::Registry是用来注册osgdb插件到osg环境中，并且储存所有链接的osgdb(读写器)插件。所以这个类所产生对象得是在整个osg运行环境中唯一存在的，否则整个的osg读写机制就会非常的混乱。正是这样osg设定使用osgDB::Registry::instance()函数来产生这个唯一的osgdb插件管理对象。我们把他涉及到的代码抽取出来供大家理解。

//所有osgDB读写插件的管理器---单例模式的创建

namespace 3wwang{

class Registry

{

public:

static Registry\* instance(bool erase = false)

{

static ref\_ptr<Registry> s\_registry = new Registry;

if (erase)

{

s\_registry->destruct();

s\_registry = 0;

}

return s\_registry.get(); // will return NULL on erase

}

//改造了addReaderWriter，便于理解

void addReaderWriter(const std::string& rw)

{

std::cout<<rw<<std::endl;

rwList.push\_back(rw);

}

protected:

Registry(){}

~Registry(){}

protected:

std::vector<std::string> rwList;

}

}

//Registry注册一个插件---单例模式的使用

namespace 3wwang{

int main(int argc,char\* argv[])

{

if (Registry::instance())

{

std::string rw = "fbxRead";

Registry::instance()->addReaderWriter(rw);

}

}

}

上面这种是饿汉式单例模式，这种模式的声明过程就是

1、需要声明一个static的函数instance，在函数外部来创建一个唯一的对象实例。（虽然代码显示的是内部创建的唯一的对象，但是因为static的声明也在函数内容，依据static对象的属性，第二次的时候就不会再进行对象的创建，直接使用。）

2、Singleton 不可以被实例化，因此我们将其构造函数声明为 protected 或者直接声明为 private。

注意：饿汉式单例模式最主要的特色，它是线程安全的。

懒汉式单例模式：

// 懒汉式单例

class CSingleton

{

private:

CSingleton() //构造函数是私有的

{

}

public:

static CSingleton \* GetInstance()

{

static CSingleton \*m\_pInstance;

if(m\_pInstance == NULL) //判断是否第一次调用

m\_pInstance = new CSingleton();

return m\_pInstance;

}

};

这种饿汉式单例模式的声明过程和饿汉式的区别就是那个唯一的对象的创建是在函数的内部完成的。这种在多线程模式下会出现创建多个单例对象的情况（上一个线程卡在了if(single == null)后创建对象前，这时第二个线程也进入了单例的getInstance函数。当再次唤醒上一个线程时，就不会判断直接创建一个对象。）。

线程安全的懒汉式单例模式

class Lock

{

private:

CCriticalSection m\_cs;

public:

Lock(CCriticalSection cs) : m\_cs(cs)

{

m\_cs.Lock();

}

~Lock()

{

m\_cs.Unlock();

}

};

class CSingleton

{

private:

CSingleton();

CSingleton(const CSingleton &);

CSingleton& operator = (const CSingleton &);

public:

static CSingleton \*Getinstance();

static CSingleton \*m\_Instance;

static CCriticalSection cs;

};

CSingleton\* CSingleton::m\_Instance = 0;

CSingleton\* CSingleton::Getinstance()

{

if(m\_Instance == NULL)

{ //double check

Lock lock(cs); //用lock实现线程安全，用资源管理类，实现异常安全

//使用资源管理类，在抛出异常的时候，资源管理类对象会被析构，析构总是发生的无论是因为异常抛出还是语句块结束。

if(m\_Instance == NULL)

{

m\_Instance = new Singleton();

}

}

return m\_Instance;

}

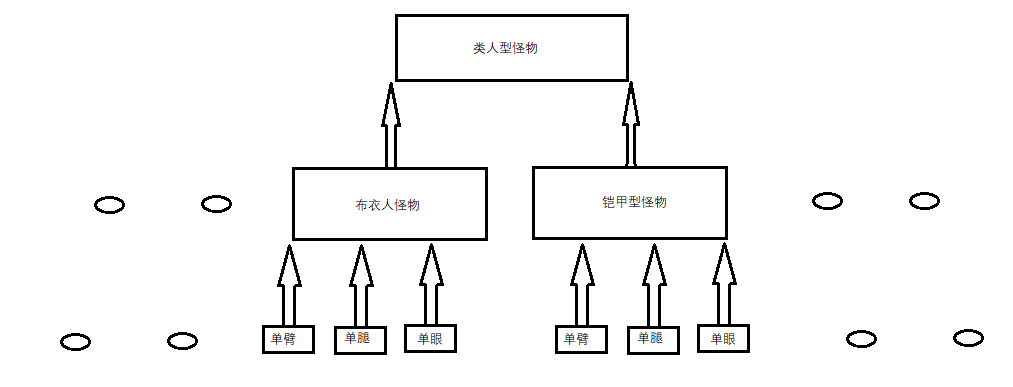
Builder(创造者)模式，是创造型模式中比较难以理解的一种设计模式，虽然在osg的源代码中有所体现，但是会增加理解负担。所以这一种设计模式我们也不使用osg中源代码进行说明。

什么时候会用Builder模式？

假如我们还是在编写一个打怪升级的游戏，在工厂模式的讲解中我们说明了怎么初始化不同等级的怪物。这一次我们遇到了新的问题，在创建类人型怪物(独臂侠，独眼龙，三头六臂怪等)的时候，理论上可以抽取这几种怪物共同的属性组成一个类人型怪物的基类。试想一下，如果这样来解决这个问题的话，那我们就会出现这种情况：

1、有多少条胳膊就要创建多少个新的对象，有多少条腿也要创建新的对象。

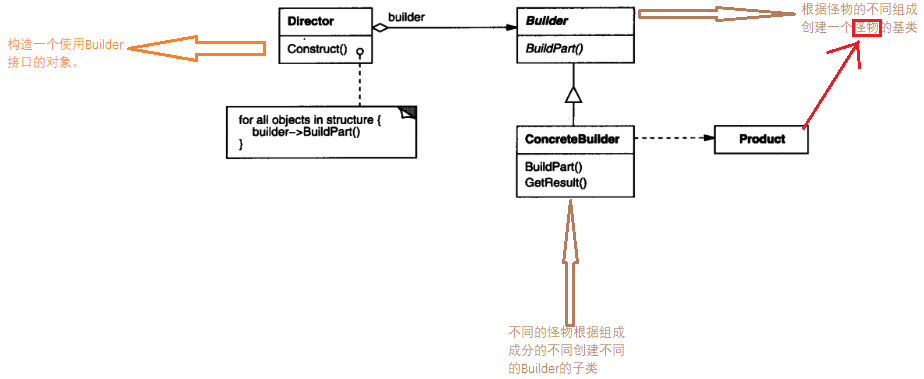
2、由于各种各样类人型怪物的差别很大，所以抽取的基类中共有的属性会非常的少，结果就是稍微有一点不同就要创建一个新的对象。



出现上面这种情况就是Builder模式的应用场景。

builder模式提出的目的就是将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。

Builder模式的大体结构图



Builder: 创建一个Product怪物对象的各个部件指定抽象接口。

ConcreteBuilder:Builder的子类(实现其接口的类)，构造指定的部件组成一个怪物。

Product:一个怪物实例。

Director：构造使用不同组成成分生产怪物的使用Builder接口的对象。

代码展示：

namespace 3wwang{

///

///怪物的基类

///

class Monster

{

public:

Monster(){}

~Monster(){}

void createMonsterArms(const std::string& arm)

{

m\_arms.push\_back(arm);

}

void createMonsterLegs(const std::string& leg)

{

m\_legs.push\_back(leg);

}

void show()

{

std::cout<<m\_monsterType<<"::";

for(int i=0;i<m\_arms.size();i++)

std::cout<<"arm"<<i<<":"<<m\_arms[i]<<",";

for(int i=0;i<m\_legs.size();i++)

std::cout<<"leg"<<i<<":"<<m\_legs[i]<<",";

std::cout<<std::endl;

}

protected:

virtual void monsterType() = 0;

protected:

std::vector<std::string> m\_arms;

std::vector<std::string> m\_legs;

std::string m\_monsterType;

}

///

///布衣怪物

///

class clothMonster : public Monster

{

public:

clothMonster()

{

monsterType();

}

~clothMonster(){}

protected:

virtual void monsterType()

{

m\_monsterType = "cloth";

}

}

///

///铠甲怪物

///

class metalMonster : public Monster

{

public:

metalMonster()

{

monsterType();

}

~metalMonster(){}

protected:

virtual void monsterType()

{

m\_monsterType = "metal";

}

}

}

namespace 3wwang{

///

///创造者的基类

///

class MonsterBuilder

{

public:

MonsterBuilder()

{

m\_monster = NULL;

}

Monster\* GetMonster()

{

return m\_monster;

}

MonsterBuilder(){}

protected:

virtual void buildArms()=0;

virtual void buildLegs()=0;

protected:

Monster\* m\_monster;

}

///

///创造两个胳膊一条腿的布衣怪物

///

class TwoArmsAndOneLegClothMonsterBuilder : public MonsterBuilder

{

public:

TwoArmsClothMonsterBuilder()

{

m\_monster = new clothMonster();

}

~TwoArmsAndOneLegClothMonsterBuilder()

{

delete\* m\_monster;

m\_monster = NULL;

}

protected:

virtual void buildArms()

{

m\_monster->createMonsterArms("towArms");

}

virtual void buildLegs()

{

m\_monster->createMonsterLegs("oneLeg");

}

}

///

///创造一个胳膊两条腿的铠甲怪物

///

class OneArmAndTwoLegsMetalMonsterBuilder : public MonsterBuilder

{

public:

OneArmAndTwoLegsMetalMonsterBuilder()

{

m\_monster = new MetalMonster();

}

~OneArmAndTwoLegsMetalMonsterBuilder()

{

delete\* m\_monster;

m\_monster = NULL;

}

protected:

virtual void buildArms()

{

m\_monster->createMonsterArms("oneArm");

}

virtual void buildLegs()

{

m\_monster->createMonsterLegs("towLegs");

}

}

}

namespace 3wwang{

///

///构造builder对象

///

class MonsterDirector

{

public:

MonsterDirector(MonsterBuilder\* monsterBuilder)

{

m\_monsterBuilder = monsterBuilder;

}

~MonsterDirector(){}

Monster\* ConstructMonster()

{

m\_monsterBuilder->buildArms();

m\_monsterBuilder->buildLegs();

}

void show()

{

m\_monsterBuilder->GetMonster()->show();

}

public:

MonsterBuilder\* m\_monsterBuilder;

}

}

///

///构造者模式的调用

///

namespace 3wwang{

int main(int argc,char\* argv[])

{

MonsterBuilder\* builder1 = new OneArmAndTwoLegsMetalMonsterBuilder();

MonsterDirector\* direct1 = new MonsterDirector(builder1);

direct1->ConstructMonster();

direct1->show();

MonsterBuilder\* builder2 = new TwoArmsAndOneLegClothMonsterBuilder();

MonsterDirector\* direct2 = new MonsterDirector(builder2);

direct2->ConstructMonster();

direct2->show();

delete\* builder1;

delete\* builder2;

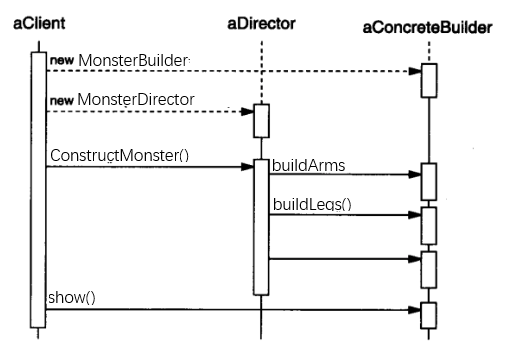
delete\* direct1;

delete\* direct2;

}

}

总结一下创造者整个的调用过程，如下图

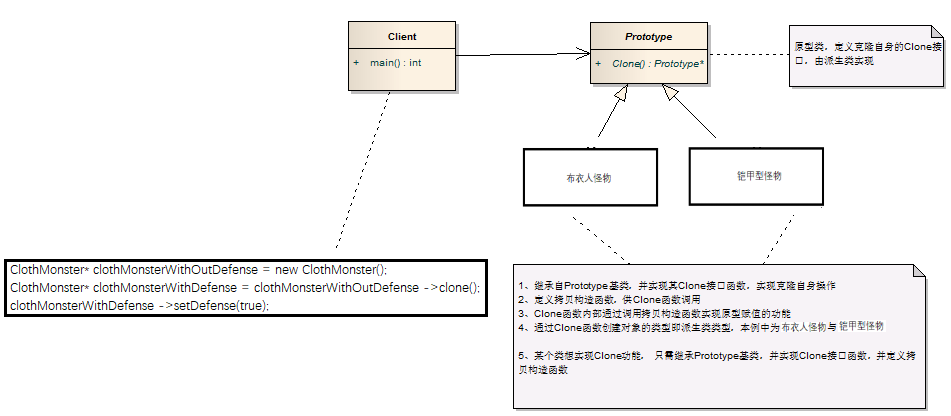


Prototype(原型)模式的起源：

我们现在还是在制作一款打怪升级的游戏，我们通过上面的工厂模式或者创造者模式来创建出来了一个人形怪物，它有自己的血条、经验以及衣着等属性，其中血条和经验是值类型(也就是说可以直接被赋值)，衣着是一个特有的对象(不能直接赋值)。这时候我们需要在一个怪物洞穴内放置100个相同的怪物，并且可能有些怪物带有防御光圈。遇到这样的问题我们在游戏制作的时候首先应该想到c++的拷贝构造函数（在此场景中应该是深拷贝），但是当遇到带有防御光圈的怪物时，拷贝构造函数就会失去作用。为了解决类似的问题就需要用到Prototype(原型)模式。

Prototype模式的作用就是：提供了自我复制的功能，就是说新对象的创建可以通过对象本身进行创建，而c++的拷贝构造函数只是复制一个已有的对象，并且原型模式生成的新对象可能是一个派生类。拷贝构造函数生成的新对象只能是类本身。

Prototype模式的架构简图



用代码实现：

///

///原型模式的基类

///

namespace 3wwang

{

class Prototype

{

protected:

Prototype(){}

~Prototype(){}

virtual Prototype\* Clone() const=0;

}

}

namespace 3wwang{

///

///怪物的基类

///

class Monster

{

public:

Monster(){}

~Monster(){}

void createMonsterArms(const std::string& arm)

{

m\_arms.push\_back(arm);

}

void createMonsterLegs(const std::string& leg)

{

m\_legs.push\_back(leg);

}

void show()

{

std::cout<<m\_monsterType<<"::";

for(int i=0;i<m\_arms.size();i++)

std::cout<<"arm"<<i<<":"<<m\_arms[i]<<",";

for(int i=0;i<m\_legs.size();i++)

std::cout<<"leg"<<i<<":"<<m\_legs[i]<<",";

std::cout<<std::endl;

}

protected:

virtual void monsterType() = 0;

protected:

std::vector m\_arms;

std::vector m\_legs;

std::string m\_monsterType;

}

///

///布衣怪物

///

class clothMonster : public Monster, public Prototype

{

public:

clothMonster()

{

monsterType();

}

clothMonster(const clothMonster& mon)

{

this.m\_arms = mon.m\_arms;

this.m\_legs = mon.m\_legs;

this.m\_monsterType = mon.m\_monsterType;

}

~clothMonster(){}

protected:

virtual void monsterType()

{

m\_monsterType = "cloth";

}

virtual Prototype\* Clone() const

{

return new clothMonster(\*this);

}

}

///

///铠甲怪物

///

class metalMonster : public Monster

{

public:

metalMonster()

{

monsterType();

}

clothMonster(const metalMonster& mon)

{

this.m\_arms = mon.m\_arms;

this.m\_legs = mon.m\_legs;

this.m\_monsterType = mon.m\_monsterType;

}

~metalMonster(){}

protected:

virtual void monsterType()

{

m\_monsterType = "metal";

}

virtual Prototype\* Clone() const

{

return new metalMonster(\*this);

}

}

}

namespace 3wwang{

///

///创造者的基类

///

class MonsterBuilder

{

public:

MonsterBuilder()

{

m\_monster = NULL;

}

Monster\* GetMonster()

{

return m\_monster;

}

MonsterBuilder(){}

protected:

virtual void buildArms()=0;

virtual void buildLegs()=0;

protected:

Monster\* m\_monster;

}

///

///创造两个胳膊一条腿的布衣怪物

///

class TwoArmsAndOneLegClothMonsterBuilder : public MonsterBuilder

{

public:

TwoArmsClothMonsterBuilder()

{

m\_monster = new clothMonster();

}

~TwoArmsAndOneLegClothMonsterBuilder()

{

delete\* m\_monster;

m\_monster = NULL;

}

protected:

virtual void buildArms()

{

m\_monster->createMonsterArms("towArms");

}

virtual void buildLegs()

{

m\_monster->createMonsterLegs("oneLeg");

}

}

///

///创造一个胳膊两条腿的铠甲怪物

///

class OneArmAndTwoLegsMetalMonsterBuilder : public MonsterBuilder

{

public:

OneArmAndTwoLegsMetalMonsterBuilder()

{

m\_monster = new MetalMonster();

}

~OneArmAndTwoLegsMetalMonsterBuilder()

{

delete\* m\_monster;

m\_monster = NULL;

}

protected:

virtual void buildArms()

{

m\_monster->createMonsterArms("oneArm");

}

virtual void buildLegs()

{

m\_monster->createMonsterLegs("towLegs");

}

}

}

namespace 3wwang{

///

///构造builder对象

///

class MonsterDirector

{

public:

MonsterDirector(MonsterBuilder\* monsterBuilder)

{

m\_monsterBuilder = monsterBuilder;

}

~MonsterDirector(){}

Monster\* ConstructMonster()

{

m\_monsterBuilder->buildArms();

m\_monsterBuilder->buildLegs();

}

void show()

{

m\_monsterBuilder->GetMonster()->show();

}

public:

MonsterBuilder\* m\_monsterBuilder;

}

}

///

///构造者模式和原型模式联合的调用

///

namespace 3wwang{

int main(int argc,char\* argv[])

{

MonsterBuilder\* builder1 = new OneArmAndTwoLegsMetalMonsterBuilder();

MonsterDirector\* direct1 = new MonsterDirector(builder1);

direct1->ConstructMonster();

direct1->show();

Prototype\* cloneMonster1 = direct1->m\_monsterBuilder->GetMonster()->clone();

cloneMonster1->show();

MonsterBuilder\* builder2 = new TwoArmsAndOneLegClothMonsterBuilder();

MonsterDirector\* direct2 = new MonsterDirector(builder2);

direct2->ConstructMonster();

direct2->show();

Prototype\* cloneMonster2 = direct2->m\_monsterBuilder->GetMonster()->clone();

cloneMonster2->show();

delete\* builder1;

delete\* builder2;

delete\* direct1;

delete\* direct2;

delete\* cloneMonster1;

delete\* cloneMonster2;

}

}

### 在osg中也有用到原型模式的地方，就是osg::Object类中的clone方法。特别是大家可以注意一下在使用osg::clone(object, copyop)这个方法的时候

Singleton(单例)模式是设计模式中普及最广的，因为它是面试官(特别是java面试官)最喜欢提问的问题；并且它所解决的问题十分常见就是常见一个并且是整个应用唯一的对象。一般情况下我们是怎么创建一个唯一的对象(变量)呢？分为一下三种情况。

1、类似c语言这种面向过程的语言设计中，我们可以通过创建一个唯一的全局变量，使用的时候包含这个头文件，然后声明一下就可以使用了。

2、类似java语言这种纯粹的面向对象的语言设计中，我们只能通过单例设计模式来创建一个唯一的全局变量。

3、类似c++语言这种面向对象和面向过程结合的语言设计中，我们可以使用上面两种方式来创建一个唯一的全局变量。

讲解工厂模式(上两节)时，没有使用到osg内部的类来说明(提醒一下工厂模式在osgearthSymbology:: GeometryFactory中使用到)，单例模式在osg原生代码中肯定会用到。我们这就进入osg的src文件查找一下。

osgDB::Registry类就是一个正宗的单例类。我可以在帮助回忆一下registry类是用来做什么，osgDB::Registry是用来注册osgdb插件到osg环境中，并且储存所有链接的osgdb(读写器)插件。所以这个类所产生对象得是在整个osg运行环境中唯一存在的，否则整个的osg读写机制就会非常的混乱。正是这样osg设定使用osgDB::Registry::instance()函数来产生这个唯一的osgdb插件管理对象。我们把他涉及到的代码抽取出来供大家理解。

//所有osgDB读写插件的管理器---单例模式的创建

namespace 3wwang{

class Registry

{

public:

static Registry\* instance(bool erase = false)

{

static ref\_ptr<Registry> s\_registry = new Registry;

if (erase)

{

s\_registry->destruct();

s\_registry = 0;

}

return s\_registry.get(); // will return NULL on erase

}

//改造了addReaderWriter，便于理解

void addReaderWriter(const std::string& rw)

{

std::cout<<rw<<std::endl;

rwList.push\_back(rw);

}

protected:

Registry(){}

~Registry(){}

protected:

std::vector<std::string> rwList;

}

}

//Registry注册一个插件---单例模式的使用

namespace 3wwang{

int main(int argc,char\* argv[])

{

if (Registry::instance())

{

std::string rw = "fbxRead";

Registry::instance()->addReaderWriter(rw);

}

}

}

上面这种是饿汉式单例模式，这种模式的声明过程就是

1、需要声明一个static的函数instance，在函数外部来创建一个唯一的对象实例。（虽然代码显示的是内部创建的唯一的对象，但是因为static的声明也在函数内容，依据static对象的属性，第二次的时候就不会再进行对象的创建，直接使用。）

2、Singleton 不可以被实例化，因此我们将其构造函数声明为 protected 或者直接声明为 private。

注意：饿汉式单例模式最主要的特色，它是线程安全的。

懒汉式单例模式：

// 懒汉式单例

class CSingleton

{

private:

CSingleton() //构造函数是私有的

{

}

public:

static CSingleton \* GetInstance()

{

static CSingleton \*m\_pInstance;

if(m\_pInstance == NULL) //判断是否第一次调用

m\_pInstance = new CSingleton();

return m\_pInstance;

}

};

这种饿汉式单例模式的声明过程和饿汉式的区别就是那个唯一的对象的创建是在函数的内部完成的。这种在多线程模式下会出现创建多个单例对象的情况（上一个线程卡在了if(single == null)后创建对象前，这时第二个线程也进入了单例的getInstance函数。当再次唤醒上一个线程时，就不会判断直接创建一个对象。）。

线程安全的懒汉式单例模式

class Lock

{

private:

CCriticalSection m\_cs;

public:

Lock(CCriticalSection cs) : m\_cs(cs)

{

m\_cs.Lock();

}

~Lock()

{

m\_cs.Unlock();

}

};

class CSingleton

{

private:

CSingleton();

CSingleton(const CSingleton &);

CSingleton& operator = (const CSingleton &);

public:

static CSingleton \*Getinstance();

static CSingleton \*m\_Instance;

static CCriticalSection cs;

};

CSingleton\* CSingleton::m\_Instance = 0;

CSingleton\* CSingleton::Getinstance()

{

if(m\_Instance == NULL)

{ //double check

Lock lock(cs); //用lock实现线程安全，用资源管理类，实现异常安全

//使用资源管理类，在抛出异常的时候，资源管理类对象会被析构，析构总是发生的无论是因为异常抛出还是语句块结束。

if(m\_Instance == NULL)

{

m\_Instance = new Singleton();

}

}

return m\_Instance;

}

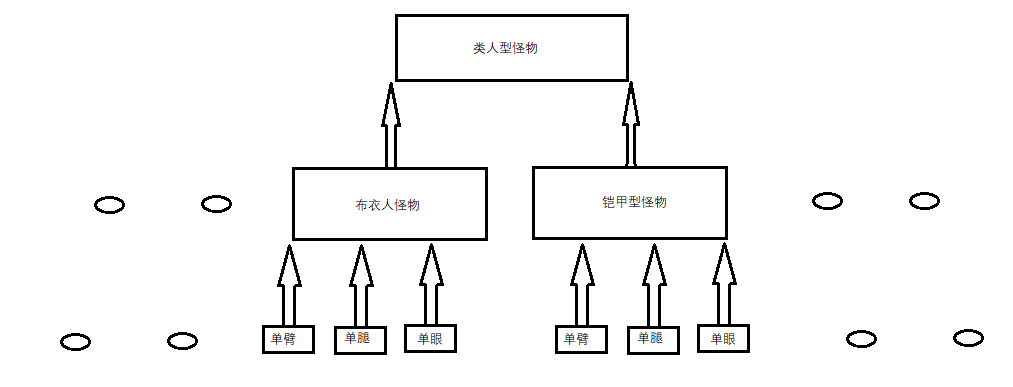
Builder(创造者)模式，是创造型模式中比较难以理解的一种设计模式，虽然在osg的源代码中有所体现，但是会增加理解负担。所以这一种设计模式我们也不使用osg中源代码进行说明。

什么时候会用Builder模式？

假如我们还是在编写一个打怪升级的游戏，在工厂模式的讲解中我们说明了怎么初始化不同等级的怪物。这一次我们遇到了新的问题，在创建类人型怪物(独臂侠，独眼龙，三头六臂怪等)的时候，理论上可以抽取这几种怪物共同的属性组成一个类人型怪物的基类。试想一下，如果这样来解决这个问题的话，那我们就会出现这种情况：

1、有多少条胳膊就要创建多少个新的对象，有多少条腿也要创建新的对象。

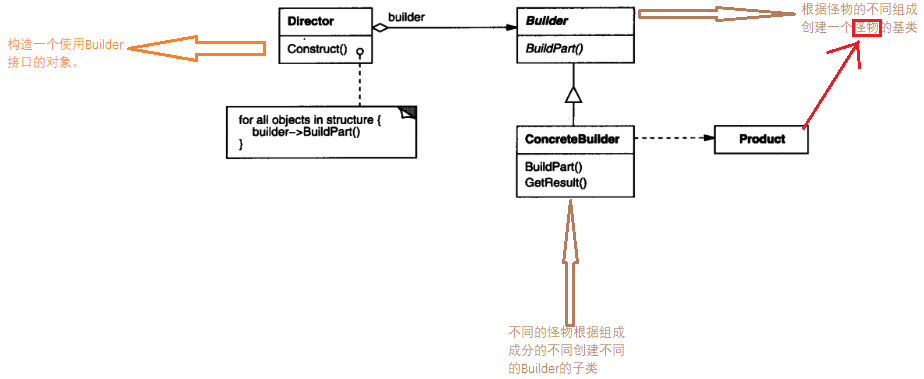
2、由于各种各样类人型怪物的差别很大，所以抽取的基类中共有的属性会非常的少，结果就是稍微有一点不同就要创建一个新的对象。



出现上面这种情况就是Builder模式的应用场景。

builder模式提出的目的就是将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。

Builder模式的大体结构图



Builder: 创建一个Product怪物对象的各个部件指定抽象接口。

ConcreteBuilder:Builder的子类(实现其接口的类)，构造指定的部件组成一个怪物。

Product:一个怪物实例。

Director：构造使用不同组成成分生产怪物的使用Builder接口的对象。

代码展示：

namespace 3wwang{

///

///怪物的基类

///

class Monster

{

public:

Monster(){}

~Monster(){}

void createMonsterArms(const std::string& arm)

{

m\_arms.push\_back(arm);

}

void createMonsterLegs(const std::string& leg)

{

m\_legs.push\_back(leg);

}

void show()

{

std::cout<<m\_monsterType<<"::";

for(int i=0;i<m\_arms.size();i++)

std::cout<<"arm"<<i<<":"<<m\_arms[i]<<",";

for(int i=0;i<m\_legs.size();i++)

std::cout<<"leg"<<i<<":"<<m\_legs[i]<<",";

std::cout<<std::endl;

}

protected:

virtual void monsterType() = 0;

protected:

std::vector<std::string> m\_arms;

std::vector<std::string> m\_legs;

std::string m\_monsterType;

}

///

///布衣怪物

///

class clothMonster : public Monster

{

public:

clothMonster()

{

monsterType();

}

~clothMonster(){}

protected:

virtual void monsterType()

{

m\_monsterType = "cloth";

}

}

///

///铠甲怪物

///

class metalMonster : public Monster

{

public:

metalMonster()

{

monsterType();

}

~metalMonster(){}

protected:

virtual void monsterType()

{

m\_monsterType = "metal";

}

}

}

namespace 3wwang{

///

///创造者的基类

///

class MonsterBuilder

{

public:

MonsterBuilder()

{

m\_monster = NULL;

}

Monster\* GetMonster()

{

return m\_monster;

}

MonsterBuilder(){}

protected:

virtual void buildArms()=0;

virtual void buildLegs()=0;

protected:

Monster\* m\_monster;

}

///

///创造两个胳膊一条腿的布衣怪物

///

class TwoArmsAndOneLegClothMonsterBuilder : public MonsterBuilder

{

public:

TwoArmsClothMonsterBuilder()

{

m\_monster = new clothMonster();

}

~TwoArmsAndOneLegClothMonsterBuilder()

{

delete\* m\_monster;

m\_monster = NULL;

}

protected:

virtual void buildArms()

{

m\_monster->createMonsterArms("towArms");

}

virtual void buildLegs()

{

m\_monster->createMonsterLegs("oneLeg");

}

}

///

///创造一个胳膊两条腿的铠甲怪物

///

class OneArmAndTwoLegsMetalMonsterBuilder : public MonsterBuilder

{

public:

OneArmAndTwoLegsMetalMonsterBuilder()

{

m\_monster = new MetalMonster();

}

~OneArmAndTwoLegsMetalMonsterBuilder()

{

delete\* m\_monster;

m\_monster = NULL;

}

protected:

virtual void buildArms()

{

m\_monster->createMonsterArms("oneArm");

}

virtual void buildLegs()

{

m\_monster->createMonsterLegs("towLegs");

}

}

}

namespace 3wwang{

///

///构造builder对象

///

class MonsterDirector

{

public:

MonsterDirector(MonsterBuilder\* monsterBuilder)

{

m\_monsterBuilder = monsterBuilder;

}

~MonsterDirector(){}

Monster\* ConstructMonster()

{

m\_monsterBuilder->buildArms();

m\_monsterBuilder->buildLegs();

}

void show()

{

m\_monsterBuilder->GetMonster()->show();

}

public:

MonsterBuilder\* m\_monsterBuilder;

}

}

///

///构造者模式的调用

///

namespace 3wwang{

int main(int argc,char\* argv[])

{

MonsterBuilder\* builder1 = new OneArmAndTwoLegsMetalMonsterBuilder();

MonsterDirector\* direct1 = new MonsterDirector(builder1);

direct1->ConstructMonster();

direct1->show();

MonsterBuilder\* builder2 = new TwoArmsAndOneLegClothMonsterBuilder();

MonsterDirector\* direct2 = new MonsterDirector(builder2);

direct2->ConstructMonster();

direct2->show();

delete\* builder1;

delete\* builder2;

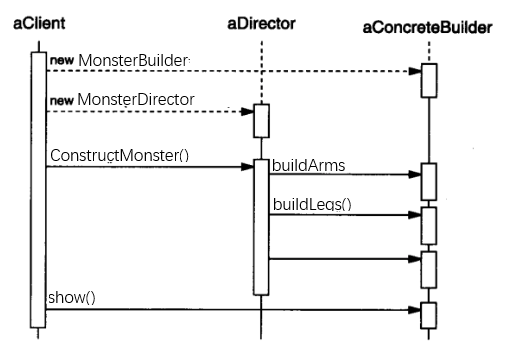
delete\* direct1;

delete\* direct2;

}

}

总结一下创造者整个的调用过程，如下图



结构性模式-Bridge模式

问题提出：我们在开发一款游戏的时候，对于特指的一种布衣人形怪物。

1、我们在刚开始编码的时候就会简单的用一个类来实现这个怪物所具备的所有功能。

2、某一天整个游戏有了”皮肤”的概念，那么我们就需要改变以前设计的布衣人形怪物类，从中抽象出共有属性和方法组成一个基类，然后每个皮肤的怪物都继承自这个基类去实现自己特有的打击效果。

3、某一天我们的游戏火了，不仅仅在Android上运行，大批的Iphone的粉丝也在期待这款游戏。所以我们需要再抽象一个基类，为Android和Iphone平台各自实现一个子类，来满足日益增加的需求。

我们可以回头看看，使用上面的方法虽然可以编写出一款游戏，但是面向对象的松耦合（Coupling），高内聚（Cohesion）两个最基本的思想被我们抛弃了。而面向对象系统追求的目标就是尽可能地提高系统模块内部的内聚（Cohesion）、尽可能降低模块间的耦合（Coupling），于是总结出了Bridge模式来解决这个问题。

Bridge模式是面向对象设计模式中非常难以理解的几个之一。难以理解的地方在于它的定义” Bridge 模式将抽象部分与它的实现部分分离，使得它们可以独立地变化”，我们一定要区分清楚这里的“抽象”和“实现”不是我们写代码时所了解的---“实现”就是“抽象”的具体子类的实现。而这里的“实现”的含义指的是怎么去实现用户的需求，并且指的是通过组合（委托）的方式实现的，因此这里的实现不是指的继承基类、实现基类接口，而是指的是通过对象组合实现用户的需求。

代码实现：

///

///实现

///

namespace 3wwang

{

class AbstractMonster

{

public:

AbstractMonster();

~AbstractMonster();

virtual void Operation() = 0;

}

class ClothMonster : public AbstractMonster

{

public:

ClothMonster();

~ClothMonster();

virtual void Operation()

{

std::cout<<"ClothMonster"<<std::endl;

}

}

class MetalMonster : public AbstractMonster

{

public:

MetalMonster();

~MetalMonster();

virtual void Operation()

{

std::cout<<"MetalMonster"<<std::endl;

}

}

}

///

///抽象

///

namespace 3wwang

{

class AbstractOperator

{

public:

AbstractOperator();

~AbstractOperator();

virtual void Operation() = 0;

}

class AndroidOprator : public AbstractOperator

{

public:

AndroidOprator(AbstractMonster\* monster)

{

\_monster = monster;

}

~AndroidOprator()

{

delete\* \_monster;

}

virtual void Operation()

{

\_monster->Operation();

}

private:

AbstractMonster\* \_monster;

}

class IphoneOprator : public AbstractOperator

{

public:

IphoneOprator(AbstractMonster\* monster)

{

\_monster = monster;

}

~IphoneOprator()

{

delete\* \_monster;

}

virtual void Operation()

{

\_monster->Operation();

}

private:

AbstractMonster\* \_monster;

}

}

namespace 3wwang

{

int main(int argc,char\* argv[])

{

AbstractMonster\* androidMonster = new MetalMonster();

AbstractOperator\* androidOperator = new AndroidOprator(androidMonster);

androidOperator->Operation();

AbstractMonster\* iphoneMonster = new ClothMonster();

AbstractOperator\* iphoneOperator = new IphoneOprator(iphoneMonster);

iphoneOperator->Operation();

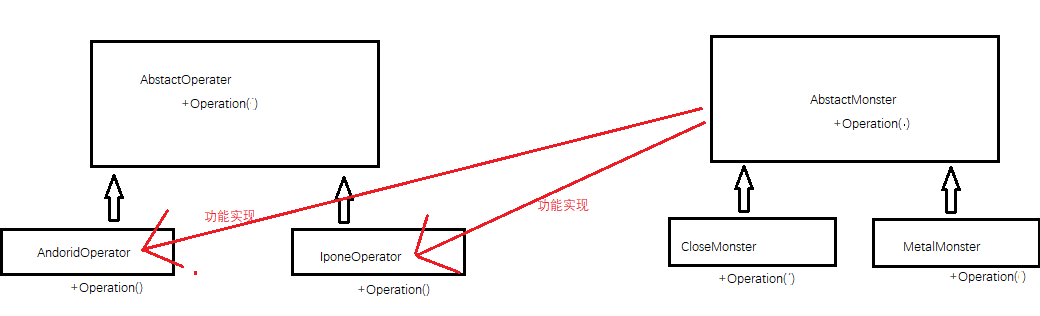
delete\* androidOperator;

delete\* iphoneOperator;

}

}

Bridge模式整个调用过程，如下图：



总结：

是否使用Bridge模式，根本的矛盾在于是使用组合来实现一个功能，还是使用继承(实现)的方式去实现一个功能。随着我们不断的开发，这个矛盾就会被我们不断的化解。

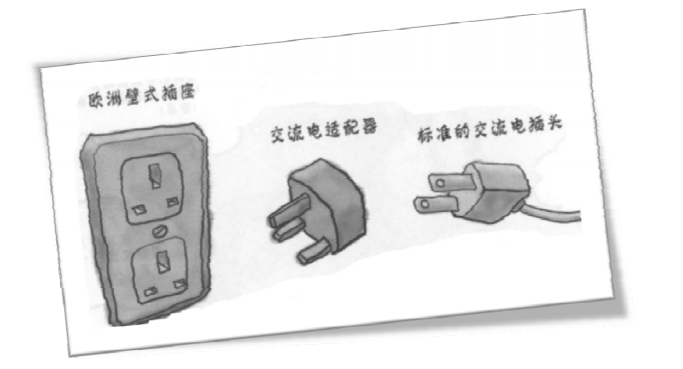
结构型设计模式---Adapter(适配器)模式

问题提出

假设我们还是在开发一款打怪升级的游戏，接到老板的要求要在现有基础上加上一个宠物功能，宠物的可升级并且它的外观、技能等在游戏的怪物中都有所体现。我们就想如果重写写一个宠物的基类，然后依次开发的他的子类，虽然也可以实现功能。但是代码出现了很多冗余（和游戏怪物相同的外观和技能）。所以这时我们就想在宠物类中复用怪物类的一些代码。出现类似的需求（将一个类的接口和另一个类的接口匹配起来，而无须修改原来的适配者接口和抽象目标类接口）。这时就会用到Adapter（适配器）模式。

Adapter(适配器)模式的作用

适配器就是一种适配中间件，它存在于不匹配的二者之间，用于连接二者，将不匹配变得匹配，简单点理解就是平常所见的转接头，转换器之类的存在。也就是适配器就像电源插座转换器，当我们需要使用二头插座但是只能找到三头插孔时，我们就需要使用电源插座转换器确保我们的机器可以运行。



适配器主要分为两种：类适配器和对象适配器。两者从根本上没有什么区别，但是在实现上有些许的不同。

代码实现

///

///宠物类

///

namespace 3wwang

{

class IPet

{

public:

virtual void play() = 0;

~IPet(){}

protected:

IPet(){}

}

}

///

///怪物类

///

namespace 3wwang

{

class IMonster

{

public:

virtual void hit() = 0;

~IMonster(){}

protected:

IMonster(){}

}

class MatelMonster : public IMonster

{

public:

MatelMonster(){}

~MatelMonster(){}

virtual void hit()

{

std::cout<<"MatelMonster->hit()"<<std::endl;

}

}

}

///

///类适配器，继承自IPer抽象基类以及MatelMonster类

///

namespace 3wwang

{

class PetToMonsterAdapter : public IPet, public MatelMonster

{

public:

PetToMonsterAdapter(){}

~PetToMonsterAdapter(){}

virtual void play()

{

this->hit();

}

}

}

///

///对象适配器

///

namespace 3wwang

{

class PetToMonsterAdapter : public IPet

{

public:

PetToMonsterAdapter(IMonster\* monster)

{

m\_monster = monster;

}

~PetToMonsterAdapter(){}

virtual void play()

{

m\_monster->hit();

}

private:

IMonster\* m\_monster;

}

}

namespace 3wwang

{

int main(int argc, char\* argv[])

{

///类适配器的调用

PetToMonsterAdapter\* adpter1 = new PetToMonsterAdapter();

adpter->play();

///对象适配器的调用

MatelMonster\* monster = new MatelMonster();

PetToMonsterAdapter\* adpter2 = new PetToMonsterAdapter(monster);

adpter2->play();

delete\* adpter1;

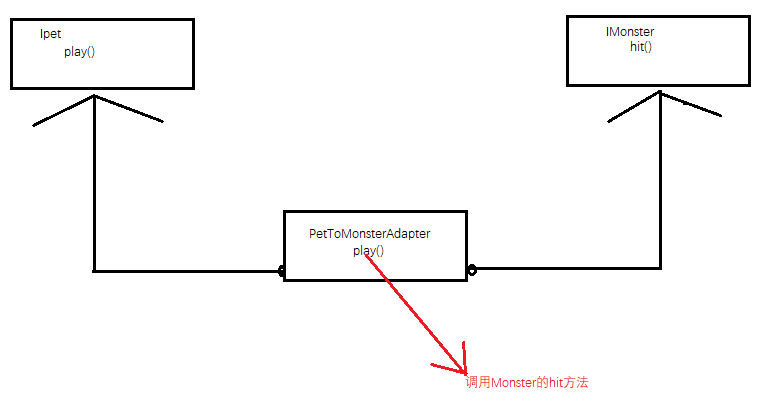
delete\* monster;

delete\* adpter2;

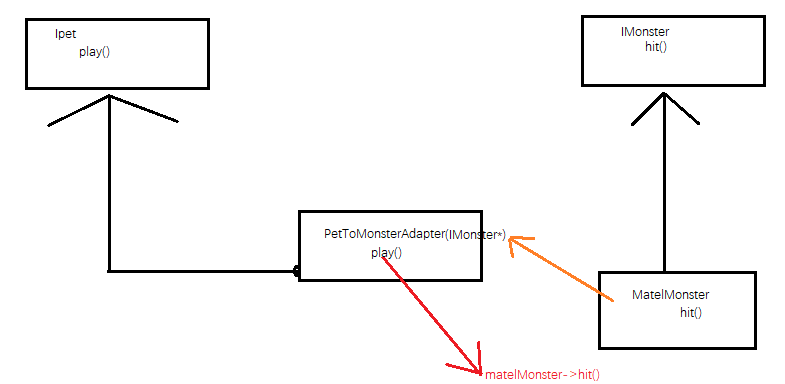
}

}

适配器模式的整个调用过程图



类适配器

  
 对象适配器

osg中的适配器的调用

osg中的事件适配器(osgGA::GUIEventAdapter)，就是在用户和操作系统事件之间放置了一个适配器，来适配用户的输入。

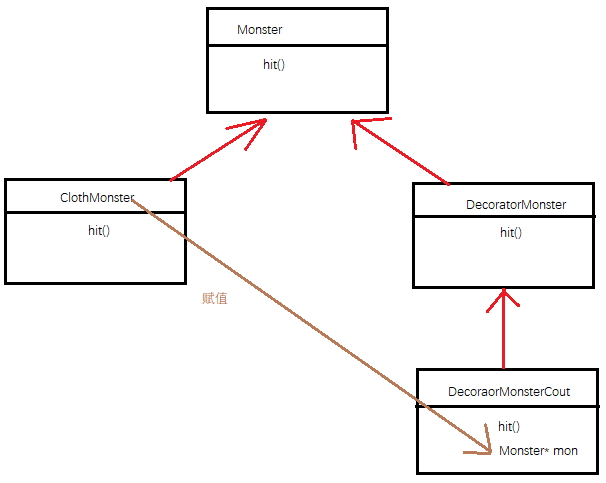
结构型设计模式--- Decorator (装饰器)模式

问题提出：

我们还是在开发这一款打怪升级的游戏，随着开发的逐渐深入，系统的代码量在不断的增加，出现错误的时候就会越多，通常解决错误的方法是debug到出现错误的地方，查看他的上下文，特别是输入和输出是否符合要求。但是那就意味着我们得把游戏玩一遍，如果玩一遍调试出问题我们也可以接受但是通常是需要无数遍的调试才能发现问题。这种方式肯定不太现实。所以我们想到了另一种办法就是在主要的函数前后加上输入和输出的功能，不仅可以定位到错误的位置，还可以形成日志，有助于以后的系统维护。但是这种方法也需要很大的工作量就是在添加类似std::cout的输出。所以这个时候我们就需要扩张类的函数的功能，当然类的主体功能不能受到影响。Decorator模式就是为了解决这种问题而出现的。

Decorator (装饰器)模式作用

就是对已经存在的某些类进行装饰，以此来扩展一些功能。其结构图如下



Monster为统一接口，也是装饰类和被装饰类的基本类型。

ClothMonster为具体实现类，也是被装饰类，他本身是个具有一些功能的完整的类。

DecoratorMonster是装饰类，实现了Component接口的，Decorator本身，通常采用默认实现，他的存在仅仅是一个声明：我要生产出一些用于装饰的子类了。而其子类才是赋有具体装饰效果的装饰产品类。

DecoratorMonsterCout是具体的装饰产品类，每一种装饰产品都具有特定的装饰效果---在hit函数的前后加上输入和输出。同时还在内部维护了一个Monster的实例，并可以通过构造器声明装饰哪种类型的ConcreteComponent，从而对其进行装饰。

代码实现

///

///基础接口类

///

namespace 3wwang

{

class Monster

{

public:

~Monster(){}

virtual int hit(int enemyNum) = 0;

protected:

Monster(){}

}

}

///

///被装饰的具体类

///

namespace 3wwang

{

class ClothMonster:public Monster

{

public:

ClothMonster(){}

~ClothMonster(){}

virtual int hit(int enemyNum)

{

return enemyNum-1;

}

}

}

///

///装饰类

///

namespace 3wwang

{

class DecoratorMonster:public Monster

{

public:

DecoratorMonster(){}

~DecoratorMonster(){}

virtual int hit(int enemyNum){return -1}

}

///

///具体装饰类

///

class DecoratorMonsterCout:public DecoratorMonster

{

public:

DecoratorMonster(Monster\* monster)

{

m\_monster = monster;

}

~DecoratorMonster(){}

virtual int hit(int enemyNum)

{

std::cout<<"输入的敌人的数量："<<enemyNum<<std::endl;

int ememyDiedNum = m\_monster->hit();

std::cout<<"死亡的敌人的数量："<<ememyDiedNum<<std::endl;

return ememyDiedNum;

}

private:

Monster\* m\_monster;

}

}

namespace 3wwang

{

int main(int argc,char\* argv[])

{

ClothMonster\* clothMonster = new ClothMonster();

DecoratorMonster\* monster = new DecoratorMonsterCout(clothMonster);

monster->hit(3);

delete\* clothMonster;

delete\* monster;

}

}

装饰器的整个调用过程

请参照上图

Osg中装饰器的调用

osgSim:: Impostor替代节点也是一种布告板，它是通过从当前视点将一个复杂物体对象绘制到一副图像纹理上来创建的，其中渲染的图像纹理映射的布告板上。值得注意的是，绘制过程与替代物在屏幕上覆盖的像素点数不是顶点数或者物体的复杂度成正比。替代节点可用于物体的一些实例或图形的一些画面，可以加速图像的绘制及渲染。替代节点的另一个优点是可以对纹理图像进行低通滤波，从而生成一张可用于景深效果的模糊图像。在实际运用中，绘制一个替代节点应该比绘制起所表示的物体要快的多，而且替代物应该和标识的物体非常像。同样重要的是，替代节点应该可以重复用于挨在一起的的多个视点，从而可以有效地利用帧与帧之间的相关性。随着视点距离的加大，物体的投影图像会随之变小，所以通常这种情形意味着可以使用替代节点方法来处理距离视点远的缓慢运动的物体。就是距离视点比较近，而且运动时有一面总是面向视点的物体也比较适合用替代节点。

但很多时候替代节点对当前几何形状的近似不是很好。当错误超过这个范围时，替代节点就会变得无效。通常，对替代物的有效限制就是它的分辨率，当一个远处的替代节点慢慢靠近时，纹理中的单个像素将变得越来越明显，原来的视觉效果就会慢慢的消失，这时就需要根据视点和替代节点的当前位置生成新的替代物。这种情况下，图形纹理的分辨率绝对不能超过屏幕的当前分辨率。

在这个类中使用到了装饰模式，请大家仔细阅读。

结构型设计模式---Composite(组合)模式

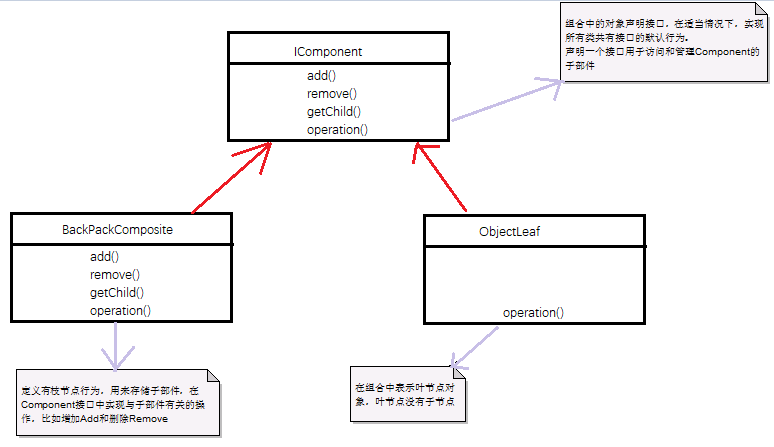
问题提出

我们还是在开发这一款打怪升级的游戏，现在我们需要给人物附加上背包，一般的游戏中背包会有很多的格子，每个格子可以放无数个相同类型的物品，每个物品的使用方式是不尽相同的(有的可以吃，有的可以用，有的可以穿等等)。如果我们编写这么一个背包的功能，可能会用到无数种if判断，并且背包的格子和物品调用的接口也不统一，非常不方便管理。Composite(组合)模式就是为了把整体和部分组合起来进行统一性的管理。

Composite(组合)模式的作用

首先将各种对象组成树形结构，枝干表示“格子”，叶子表示“物品”。并且所有的对象都有共同的基类，使得对枝干和叶子的使用具有统一性。

组合模式管理上面的”背包功能”，就会方便很多。下图表示了组合模式的设计思想



IComponent声明了所有用来管理子对象的方法，但是ObjectLeaf没有子对象，所以它不需要重写这些管理方法：

1)Component::Add添加一个子组件

2)Component::Remove::删除一个子组件.

3)Component::GetChild:获得子组件的指针.

无论是枝干(BackPackComposite)还是叶子(ObjectLeaf)，都应该重写operation()方法，从而实现他们特有的功能。并且叶子可以有很多中就想上面提到的(有的可以吃，有的可以用，有的可以穿等等)。也就是说：Component::Operatation:定义了各个组件共有的行为接口，由各个组件的具体实现。

特别是当我们需要遍历所有的物品时，这种设计模式可以大大的提高编码效率。

代码展示

///

///组件接口定义

///

namespace 3wwang{

class IComponent

{

public:

virtual ~IComponent(){}

virtual void add(IComponent\*){}

virtual void remove(IComponent\*){}

virtual IComponent\* getChild(int index){}

virtual void operation() = 0;

protected:

IComponent(){}

}

}

///

///枝干定义

///

namespace 3wwang{

class BackPackComposite: public IComponent

{

public:

BackPackComposite(){}

~BackPackComposite(){}

virtual void add(IComponent\* component)

{

m\_componentList.push\_back(component);

}

virtual void remove(IComponent\* component)

{

m\_componentList.erase(component);

}

virtual IComponent\* getChild(int index)

{

if(index < 0 || index > this->m\_componentList.size())

{

return NULL;

}

return this->m\_componentList[index];

}

virtual void operation()

{

std::cout<<"BackPackComposite"<<std::endl;

}

protected:

std::vector<IComponent\*> m\_componentList;

}

}

///

///叶子定义

///

namespace 3wwang{

class ObjectLeaf: public IComponent

{

public:

ObjectLeaf(){}

~ObjectLeaf(){}

virtual void operation()

{

std::cout<<"ObjectLeaf"<<std::endl;

}

}

}

///

///背包的组织方式以及遍历方法

///

namespace 3wwang{

int main(int argc,char\* argv[])

{

BackPackComposite\* backPackComposite = new BackPackComposite();

BackPackComposite\* cellComposite = new BackPackComposite();

ObjectLeaf\* objectLeaf = new ObjectLeaf();

cellComposite.add(objectLeaf);

backPackComposite.add(cellComposite);

run(backPackComposite);

}

void run(BackPackComposite\* backPackComposite)

{

for(int i=0;i<backPackComposite->m\_componentList.size();i++)

{

BackPackComposite\* tempCell = dynamic\_cast<BackPackComposite\*>(backPackComposite->m\_componentList[i]);

if(!!tempCell)

run(tempCell);

backPackComposite->m\_componentList[i]->operation();

}

}

}

Osg中组合模式

Osg中的渲染树就是完美应用了Composite组合模式，如果大家对渲染树、状态树以及节点树不是很了解，请翻看我的文章 探索未知种族之osg类生物---状态树与渲染树以及节点树之间的关系。渲染树中的RenderBin和RenderStage都集成自RenderBin，并且他们之间相互组合形成了树结构类型。请大家仔细阅读相关内容。

结构型设计模式--- Flyweigh(享元)模式

问题提出

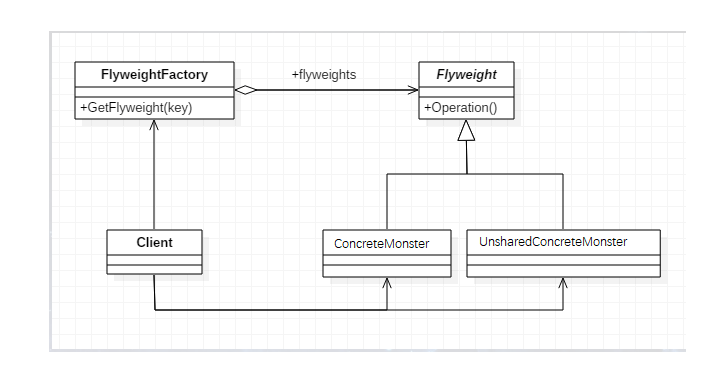
我们还是继续开发打怪升级游戏，这一次我们遇到了新的问题，就是可以联网组队刷怪。由于每一个副本都存在大量的怪物，这样如果多人联机就会出现必须为各一个队友初始化一遍所有的怪物，这样会导致大量的内存的损耗，服务器的性能就至关重要。但是每个怪物对于每个队员都是同样的，所以就提出了享元模式来解决这种大量的相对不变的对象结构。

Flyweigh(享元)模式的作用

一提到享元模式，我们就应该想到各种池(数据库连接池，String常量池，缓存数据池等等)，这种池技术就是应用到了享元模式。

享元模式运用共享技术有效地支持大量细粒度的对象。每一个所用共享的元素一般都包含内部状态和外部状态。内部状态是共享元素特有的属性，只要对象生成了，内部状态就不会发生变化。而外部状态是被外界控制的共享元素内的属性，外部状态可以被修改。

享元模式的使用一般都会结合工厂模式来生成或者取出共享的元素(有点类似单例模式)。下图是享元模式的整体结构图。



代码实现

namespace 3wwang

{

class Flyweight

{

public:

virtual ~Flyweight(){};

virtual void operate() = 0;

std::string getExtrinsic()

{

return m\_extrinsic

}

void setExstrinsic(const std::string& extrinsic)

{

m\_extrinsic = extrinsic;

}

protected:

Flyweight(){};

protected:

std::string m\_instrinsic;//内部状态

std::string m\_extrinsic;//外部状态

}

}

namespace 3wwang

{

class ConcreteMonster: public Flyweight

{

public:

ConcreteMonster(const std::string& instrinsic)

{

m\_instrinsic = instrinsic;

m\_extrinsic = "0";

}

~ConcreteMonster(){}

virtual void operate()

{

std::cout<<"ConcreteMonster."<<m\_instrinsic<<","<<m\_extrinsic<<std::endl;

}

}

}

namespace 3wwang

{

class FlyweightFactory

{

public:

FlyweightFactory(){};

~FlyweightFactory(){};

Flyweight\* getFlyweight(const std::string& instrinsic)

{

Flyweight\* tempMonster = NULL;

std::map<std::string,Flyweight\*>::iterator iter;

iter = m\_MonsterPool.find(instrinsic);

if(iter != mapStudent.end())

{

tempMonster = \*(iter->second);

}

else

{

tempMonster = new ConcreteMonster(instrinsic);

//放入池中

pool.put(instrinsic, tempMonster);

}

return tempMonster;

}

protected:

std::map<std::string,Flyweight\*> m\_MonsterPool;

}

}

namespace 3wwang

{

int main(int argc,char\* argv[])

{

std::string monsterName = "clothMonster";

FlyweightFactory\* monFactory = new FlyweightFactory();

Flyweight\* clothMonster = monFactory->getFlyweight(monsterName);

clothMonster->operate();

clothMonster->setExstrinsic("1");

clothMonster->operate();

delete\* clothMonster;

delete\* monFactory;

}

}

Osg中的享元模式应用

暂时还没有找到osg中对应的应用场景，但是在我们实际开发中一定会碰到一下几种情况

1、如果一个系统中存在大量的相同或者相似的对象，由于这类对象的大量使用，会造成系统内存的耗费，可以使用享元模式来减少系统中对象的数量。

2、对象的大部分状态都可以外部化实例化，可以将这些外部状态传入对象中。

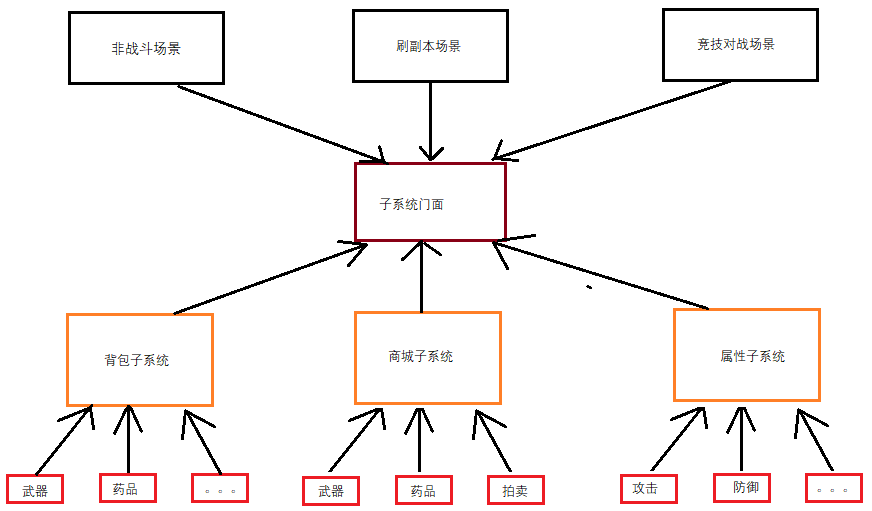
结构型设计模式---facade (外观)模式

问题提出

我们还是在开发一款打怪升级的游戏，无论角色处于那种场景(刷副本、竞技对战、非战斗场景等等)中，都要与游戏中的一些子系统(背包子系统、商城子系统、属性子系统等等)进行交互，并且每个子系统都是由无数个类协作组成的。这样当我们初始化场景的时候就需要遍历所有的子系统，以及所有子系统中的类来进行他们的初始化，增加整个系统的耦合性。为了解耦整个系统，需要把各个子系统封装成模块，然后给每个模块制定一个“门面”，通过这个“门面”可以进行整个系统的初始化等操作。这种解决模式就是facade (外观)模式。

facade (外观)模式的作用

整个应用场景示意图：



通过子系统的外观模式，可以降低各个相对独立的子系统之间的通信和依赖，实现松耦合的特性。外部调用时不必理会各个子系统内部的实现和组织方式，降低应用系统的复杂性，便于维护。

并且整个游戏程序中，不只存在这一种情况需要”门面”来调用各个子系统，所以必要的时候也要抽取门面类的基类，统一实现这个功能。

并且每个门面类都必须是一个单例类，也就是整个程序中只会存在某种门面的唯一的一个对象，来调用各个子系统。

代码实现

namespace 3wwang

{

class EnvScene

{

public:

~EnvScene()

{

delete\* m\_bs;

delete\* m\_ps;

delete\* m\_gs;

}

static EnvScene\* getInstance()

{

static EnvScene\* m\_envScene = new EnvScene();

return m\_envScene;

}

void initSubSystem()

{

m\_bs.Init();

m\_ps.Init();

m\_gs.Init();

}

BattleSystem\* getBattleSystem()

{

return m\_bs;

}

PacketSystem\* getPacketSystem()

{

return m\_ps;

}

MallSystem\* getMallSystem()

{

return m\_gs;

}

priavte:

EnvScene()

{

m\_bs = new BattleSystem();

m\_ps = new PacketSystem();

m\_gs = new MallSystem();

}

private:

static BattleSystem\* m\_bs;

static PacketSystem\* m\_ps;

static MallSystem\* m\_gs;

}

}

namespace 3wwang

{

class BattleSystem

{

public:

BattleSystem(){}

~BattleSystem(){}

void Init()

{

std::cout<<"BattleSystem"<<std::endl;

}

}

class PacketSystem

{

public:

PacketSystem(){}

~PacketSystem(){}

void Init()

{

std::cout<<"PacketSystem"<<std::endl;

}

}

class MallSystem

{

public:

MallSystem(){}

~MallSystem(){}

void Init()

{

std::cout<<"MallSystem"<<std::endl;

}

}

}

namespace 3wwang

{

int main(int argc, char\* argv[])

{

EnvScene\* envScene = EnvScene::getInstance();

envScene->initSubSystem();

delete\* envScene;

}

}

Osg中façade(门面)模式应用

在osg中暂时没有找到具体的façade(门面)模式的应用场景，但是随着osgEarth，ODE，osgOcean等的深入探究，我们总会发现门面模式的应用场景。如果您有所新的发现请留言通知，我们共同进步。

结构型设计模式---Proxy(代理)模式

我们依然是在开发一款打怪升级的游戏，随着游戏慢慢的进入人们的生活，人们发现在玩游戏的时候不能做其他任何事情，比如你正在刷一个特别的副本(进入一次成本较高)，这时突然来了外卖，你不能一边取着外卖一边使用神级的操作。所以就有一种诉求---挂机模式。作为游戏的开发者想要实现这个挂机模式，就得使游戏中存在一个’玩家的替身’帮助玩家自动完成一些动作，这个时候Proxy(代理)模式就会排上用场。代理类和玩家在游戏内的人物必须继承自同一个基类，然后在proxy类中保存一个玩家人物类的对象。这样就可以自动的执行一些人物的动作，实现挂机功能。

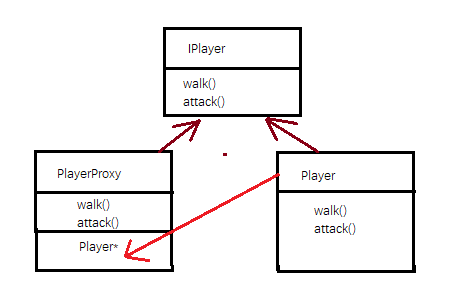
Proxy(代理)模式的作用

为其他对象提供一种代理以控制对这个对象的访问。

使用代理模式的情景1、无法直接操作某个对象，例如要调用的对象来网络的另一台机器上，无法直接那个这个对象，这个时候我们就需要在客户端保存一个远程对象的代理，通过它和远程对象间的联系来调用实际的对象。

使用代理模式的情景2、通过调用实际的对象不能满足我们的要求，我们需要补充一些功能。例如我们访问服务器上的一些资源的时候需要权限限制，通过代理类可以实现只有符合权限要求的请求才可以使用实际对象。这里可能有疑问，为什么不直接在实际的类内实现这个功能，原因一、违反单一原则，原因二、对修改关闭扩展开放。

Proxy(代理)模式的框架图



普通代理模式

代码实现

//基类

namespace 3wwang

{

class IPlayer

{

public:

virtual ~IPlayer(){}

virtual void walk() = 0;

virtual void attack() = 0;

protected:

IPlayer();

}

}

//真实类

namespace 3wwang

{

class Player: public IPlayer

{

public:

Player()[]

~Player(){}

virtual void walk()

{

std::cout<<"Player::walk"<<std::endl;

}

virtual void attack()

{

std::cout<<"Player::attack"<<std::endl;

}

}

}

//代理类

namespace 3wwang

{

class PlayerProxy : public IPlayer

{

public:

PlayerProxy()

{

m\_player = new Player();

}

~PlayerProxy()

{

delete\* m\_player

}

virtual void walk()

{

std::cout<<"PlayerProxy::walk"<<std::endl;

m\_player->walk();

}

virtual void attack()

{

std::cout<<"PlayerProxy::attack"<<std::endl;

m\_player->attack();

}

private:

Player\* m\_player;

}

}

namespace 3wwang

{

int main(int argc, char\* argv[])

{

PlayerProxy\* proxy = new PlayerProxy();

proxy->walk();

proxy->attack();

}

}

Osg中Proxy(代理)模式

我们一定会想到osg::ProxyNode这个类，但是这个类并不是使用了代理模式，osg::ProxyNode的主要作用就是为了实现延时加载一些模型等操作，具体的使用过程应该是和PagedLOD使用的时机相同是在分页数据库DatabasePager中使用，如有疑问移步到 osg探究补充：DatabasePager类简介 进行查看和提问。

补充：在java中还有动态代理模式，这是一种java内置的对象。

结构型设计模式总结

1. Adapte适配器模式：Adapter模式通过类的继承(类适配器)或者对象的组合(对象适配器)来实现，侧重于转换已有的接口，也就是说我们清楚的知道这个类就是我们需要的，只是调用方式有冲突。类适配器采用“多继承”的实现方式，带来了不良的高耦合。对象适配器采用“对象组合”的方式，更符合松耦合精神。

  例如：每种手机的充电接口不一样，我们只要usb类型的充电线，但是手机需要的是type-c的，所以想要使用与之匹配的数据线，就要用转接头可以使不同的接口连接在一起。

2. Bridge桥接模式：将抽象部分与实现部分分离，使它们都可以独立的变化。减少因变化带来的代码的修改量。注意这里的抽象和实现并非是父和子的关系，实质功能上的抽象，和功能的实现。

 例如：手机上的某个app既要支持android也要支持iphone，我们要把操作系统和软件功能实现相分离，然后把软件和操作系统相互组合，使其可以在任意的操作系统上运行。

3. Composite组合模式：将对象组合成树形结构以表示“部分-整体”的层次结构。Composite模式使得客户对单个对象和组合对象的使用具有一致性。从而解决了解决客户程序与复杂对象容器的解耦，即：通过继承统一的接口，我们可以将容器对象及其子对象看成同一类对象使用，以减少对象使用中的复杂度。

 例如：手机操作系统上的文件系统中的folder和file，他们实现了一个公共的接口，只是打开方法的实现不同。

4. Decorator装饰模式：动态地给一个对象添加一些额外的职责。就增加功能来说，Decorator模式相比生成子类更为灵活。Decorator模式采用对象组合而非继承的手法，实现了在运行时动态的扩展对象功能的能力，而且可以根据需要扩展多个功能，避免了单独使用继承带来的“灵活性差”和“多子类衍生问题”。同时它很好地符合面向对象设计原则中“优先使用对象组合而非继承”和“开放-封闭”原则。

例如：手机我们想要竖着放到桌子上，方便看视频。我们可以给手机上加一个装饰---带有手指环的手机套。这样就完美的解决了问题，并且没有改变手机本身。

5. Facade外观模式：为子系统中的一组接口提供一个一致的界面，可以通过界面上的一个函数来调用这个子系统中这一组接口的方法，简化接口。

例如：我们通过手机拨打10086，来办理流量，短信，宽带等业务，10086的客服会转到相应子系统的负责人哪里来取得信息，然后在返回给客户，这样10086就相当于为所有的子系统提供了统一的一致界面，方便用户的使用。

6. Flyweight享元模式：运用共享技术有效地支持大量细粒度的对象。解决：　面向对象的思想很好地解决了抽象性的问题，一般也不会出现性能上的问题。但是在某些情况下，对象的数量可能会太多，从而导致了运行时的代价。那么我们如何去避免大量细粒度的对象，同时又不影响客户程序使用面向对象的方式进行操作，享元模式的出现恰好解决了该问题。

　例如：公共交换电话网（PSTN）是享元的一个例子。有一些资源例如拨号音发生器、振铃发生器和拨号接收器是必须由所有用户共享的。当一个用户拿起听筒打电话时，他不需要知道使用了多少资源。对于用户而言所有的事情就是有拨号音，拨打号码，拨通电话。

7. Proxy代理模式：为其他对象提供一种代理以控制这个对象的访问。解决直接访问某些对象是出现的问题。

例如：中介是我们最常见的一个代理！

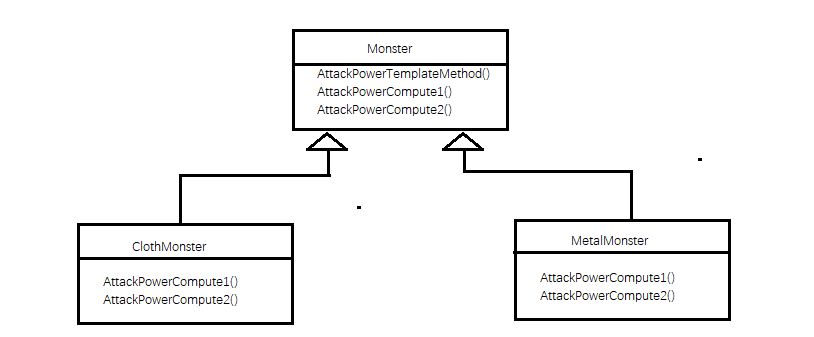
行为型设计模式----Template(模板)模式

我们依然在开发一款打怪升级的游戏，我们知道怪物的攻击力和它们自身的属性十分相关，攻击力越强造成的伤害就会越大。所以每个怪物都得有自己一套的算法机制来计算攻击力。为了实现所有怪物攻击力算法接口，一般我们就需要在每个怪物类中实现一套算法。由于怪物的种类很多，所以可能实现的算法的接口也会有所不同。这样就增加了我们外部调用计算攻击力算法的难度和复杂度。而为了使所有的怪物的子类都有一个统一的计算攻击力的算法接口，而又可以为每个不同怪物的攻击力的计算赋予它独有的算法。我们就需要用到Template模式。

Template(模板)模式的作用

采用继承的方式来实现将算法框架放在抽象基类中，并定义好细节的接口，然后这些接口函数在子类中进行实现。

Template(模板)模式的大体结构图



Monster中的AttackPowerTemplateMethod()是统一的计算攻击力算法的入口，而AttackPowerCompute1()和AttackPowerCompute2()是要根据每个子类怪物种类的不同而赋予不同的变化。

代码实现

///

///基类：提供统一的算法接口

///

namespace 3wwang

{

class Monster

{

public:

~Monster(){}

AttackPowerTemplateMethod()

{

std::cout<<"AttackPowerTemplateMethod"<<std::endl;

AttackPowerCompute1();

AttackPowerCompute2();

}

virtual void AttackPowerCompute1() = 0;

virtual void AttackPowerCompute2() = 0;

protected:

Monster(){}

}

}

///

///布衣怪物的攻击力算法

///

namespace 3wwang

{

class ClothMonster : public Monster

{

public:

ClothMonster(){};

~ClothMonster(){};

virtual void AttackPowerCompute1()

{

std::cout<<"ClothMonster::AttackPowerCompute1"<<std::endl;

}

virtual void AttackPowerCompute2()

{

std::cout<<"ClothMonster::AttackPowerCompute2"<<std::endl;

}

}

}

///

///金属怪物的攻击力算法

///

namespace 3wwang

{

class MetalMonster : public Monster

{

public:

MetalMonster(){};

~MetalMonster(){};

virtual void AttackPowerCompute1()

{

std::cout<<"MetalMonster::AttackPowerCompute1"<<std::endl;

}

virtual void AttackPowerCompute2()

{

std::cout<<"MetalMonster::AttackPowerCompute2"<<std::endl;

}

}

}

namespace 3wwang

{

int main(int argv,char\* argc[])

{

Monster\* clothMonster = new ClothMonster();

Monster\* metalMonster = new MetalMonster();

clothMonster->AttackPowerTemplateMethod();

metalMonster->AttackPowerTemplateMethod();

delete\* clothMonster;

delete\* metalMonster;

}

}

Osg中template模式实现

Osg中有很多用到template模式的地方，这里我就举一个例子，就是Dragger(拖拽器)以及他所有的子类。拖拽器的主要算法实现是在两个handle中实现。

virtual bool handle(const osgGA::GUIEventAdapter &ea, osgGA::GUIActionAdapter &aa)

virtual bool handle(const PointerInfo& pi, const osgGA::GUIEventAdapter& ea, osgGA::GUIActionAdapter& aa);

每个Dragger的子类都是在其中实现自己特有的拖拽算法，实现不同的场景拖拽效果的。

行为型设计模式---- Strategy (策略)模式

问题提出

针对上一节的Template模式，我们来总结一下他的优缺点：

优点：1、简单、易用的通过继承的方式实现了算法的异构。

2、Template模式可以实现面向对象的”依赖倒置”原则。也就是父类调用子类的操作子类实现父类声明的接口。这样控制权在父类，子类是要依赖于父类的逻辑。

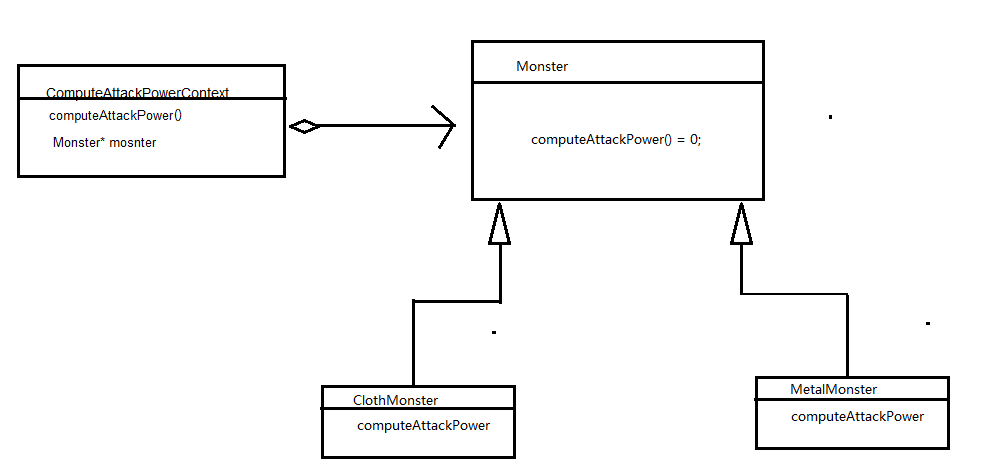
缺点：1、由于继承的强制性约束关系，我们想要复用子类中某一种算法的实现，就会变得非常的困难。

为了解决这种继承所带来的’通病’，我们可以通过组合(委托)的方式重现实现怪物攻击力相关的算法计算问题。也就是Strategy (策略)模式

Strategy (策略)模式的作用

Strategy (策略)模式和 Template（模板）模式要解决的问题是类似的，都是为了给算法的具体实现和抽象接口之间进行解耦。Strategy 模式将算法（computeAttackPower）封装到一个类（ComputeAttackPowerContext）里面，通过组合的方式将具体算法的实现在组合对象中实现，再通过委托的方式将抽象接口的实现委托给组合对象实现。

Strategy (策略)模式的大体结构图



ComputeAttackPowerContext：封装了算法的调用逻辑

Monster及其子类：实现了算法的具体的实现方法。

代码实现

namespace 3wwang

{

class ComputeAttackPowerContext

{

public:

ComputeAttackPowerContext(Monster\* monster)

{

m\_monster = monster;

}

~ComputeAttackPowerContext(){}

computeAttackPower()

{

std::cout<<"ComputeAttackPowerContext::computeAttackPower"<<std::endl;

if(!!m\_monster)

m\_monster->computeAttackPower();

}

private:

Monster\* m\_monster;

}

}

namespace 3wwang

{

class Monster

{

public:

virtual ~Monster(){}

virtual void computeAttackPower() = 0;

protected:

Monster(){}

}

class ClothMonster:public Monster

{

public:

ClothMonster(){};

~ClothMonster(){};

void computeAttackPower()

{

std::cout<<"ClothMonster::computeAttackPower"<<std::endl;

}

}

class MetalMonster:public Monster

{

public:

MetalMonster(){};

~MetalMonster(){};

void computeAttackPower()

{

std::cout<<"MetalMonster::computeAttackPower"<<std::endl;

}

}

}

namespace 3wwang

{

int main(int argv,char\* argc[])

{

Monster\* clothMonster = new ClothMonster();

ComputeAttackPowerContext\* context = new ComputeAttackPowerContext(clothMonster);

context->computeAttackPower();

delete\* clothMonster;

delete\* context;

}

}

Osg中Strategy (策略)模式的实现

osg::osgParticle::SinkOperator，粒子的下落计算方式，就是根据不同的\_sinkTarget，来计算出不同的结果。

注：从上面对比中我们可以看出，组合相比继承可以取得更好的效果，因此在面向对象的设计中的有一条很重要的原则就是： 优先使用（对象）组合，而非（类）继承（FavorComposition Over Inheritance）

行为型设计模式 —— State(状态)模式

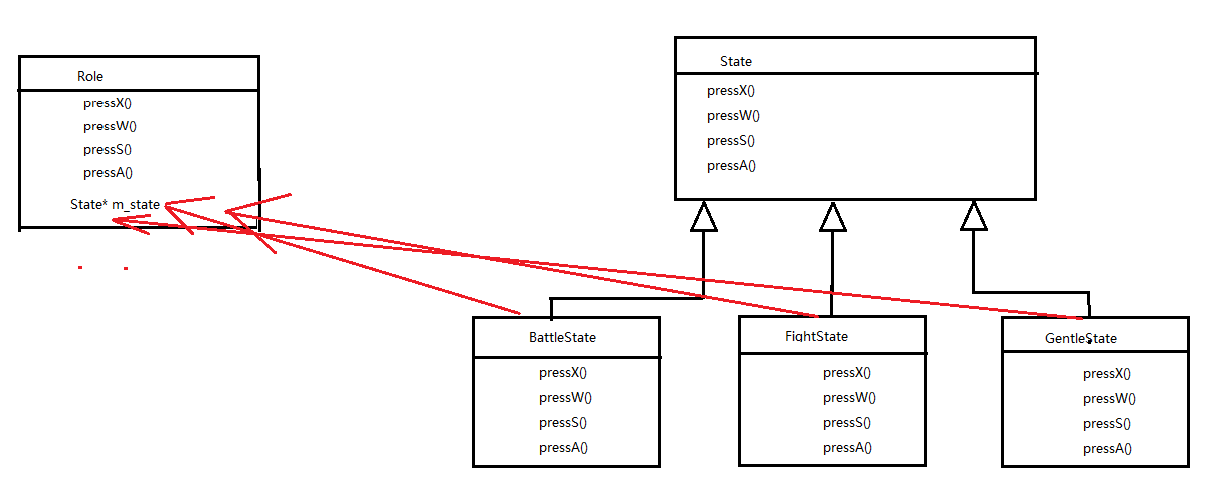
问题提出

我们打怪升级的游戏中的人物存在各种“状态”---打怪状态、决斗状态以及主城区状态。由于处于各个状态时处理事件的方式会有所不同。例如打怪状态按x键挥刀，决斗状态按x键挥刀，主城区状态按x键打开聊天室。一般的开发步骤是通过使用多个if或者switch语句来判断角色所处的状态，然后在确定键盘事件发生时所应当触发的动作。这样固然可以解决我们的问题，但是这种代码不但枯燥无趣，而且难以维护，如果需要添加一个新的状态(死亡状态)是不是就意味着需要修改每个if语句。基于这种情况提出了State模式。

State(状态)模式的作用

我们将可能需要用到的状态封装为一个类，在每个类的内部来处理相应的键盘事件，外部会有一个角色 (Role)类，来确定所处的状态并调用相应的函数。这样思路就非常的清晰，如果再增加一种状态，代码需要修改的地方会非常的少，对于状态非常多的情景来说非常的方便。

State(状态)模式的大体结构图



State是所有可能用到的状态的基类，子类BattleState\FightState\GentleState中实现了四个特有方法。

Role是角色类，会根据不同的state来确定使用具体的那个state来调用相应的事件。

代码实现

///

///状态的基类

///

namespace 3wwang

{

class State

{

public:

virtual ~State(){}

virtual void PressX() = 0;

virtual void PressW() = 0;

virtual void PressS() = 0;

virtual void PressA() = 0;

protected:

State(){}

}

}

///

///状态的具体实现类(子类)

///

namespace 3wwang

{

//决斗状态

class BattleState : public State

{

public:

BattleState(){}

~BattleState(){}

virtual void PressX()

{

std::cout<<"BattleState::PressX"<<std::endl;

}

virtual void PressW()

{

std::cout<<"BattleState::PressW"<<std::endl;

}

virtual void PressS()

{

std::cout<<"BattleState::PressS"<<std::endl;

}

virtual void PressA()

{

std::cout<<"BattleState::PressA"<<std::endl;

}

}

//打怪状态

class FightState : public State

{

public:

FightState(){}

~FightState(){}

virtual void PressX()

{

std::cout<<"FightState::PressX"<<std::endl;

}

virtual void PressW()

{

std::cout<<"FightState::PressW"<<std::endl;

}

virtual void PressS()

{

std::cout<<"FightState::PressS"<<std::endl;

}

virtual void PressA()

{

std::cout<<"FightState::PressA"<<std::endl;

}

}

//平和状态

class GentleState : public State

{

public:

GentleState(){}

~GentleState(){}

virtual void PressX()

{

std::cout<<"GentleState::PressX"<<std::endl;

}

virtual void PressW()

{

std::cout<<"GentleState::PressW"<<std::endl;

}

virtual void PressS()

{

std::cout<<"GentleState::PressS"<<std::endl;

}

virtual void PressA()

{

std::cout<<"GentleState::PressA"<<std::endl;

}

}

}

///

///角色类(使用状态)

///

namespace 3wwang

{

class Role

{

public:

Role(){}

~Role(){}

setState(State\* state)

{

if(!!m\_state)

delete m\_state;

m\_state = state;

}

void PressX()

{

if(!!m\_state)

m\_state->PressX();

}

void PressW()

{

if(!!m\_state)

m\_state->PressW();

}

void PressS()

{

if(!!m\_state)

m\_state->PressS();

}

void PressA()

{

if(!!m\_state)

m\_state->PressA();

}

private:

State\* m\_state;

}

}

namespace 3wwang

{

int main(int argc, char\* argv[])

{

Role\* role = new Role();

State\* state = new GentleState();

role->setState(state)

role->PressX();

delete\* state;

state = NULL;

state = new FightState();

role->setState(state)

role->PressX();

delete\* state;

state = NULL;

state = new BattleState();

role->setState(state)

role->PressX();

}

}

Osg中state(状态)模式的实现

Opengl整个的函数库就是一个状态集，也就是认为整个三维环境都是有状态组成的，前一时刻到下一时刻，整个三维环境的空间发生了什么样的变化，因此状态可以非常的大也可以非常的小，光照情况是状态，纹理单元也是状态，三维世界的组成等等都是状态，因此状态无处不在。Osg就是对opengl进行了一层封装，使其接口更加的易用，各种状态实现统一的管理等。所以osg中控制渲染模式的整个流程都是使用到了状态模式。

行为型设计模式----Observer(观察者)模式

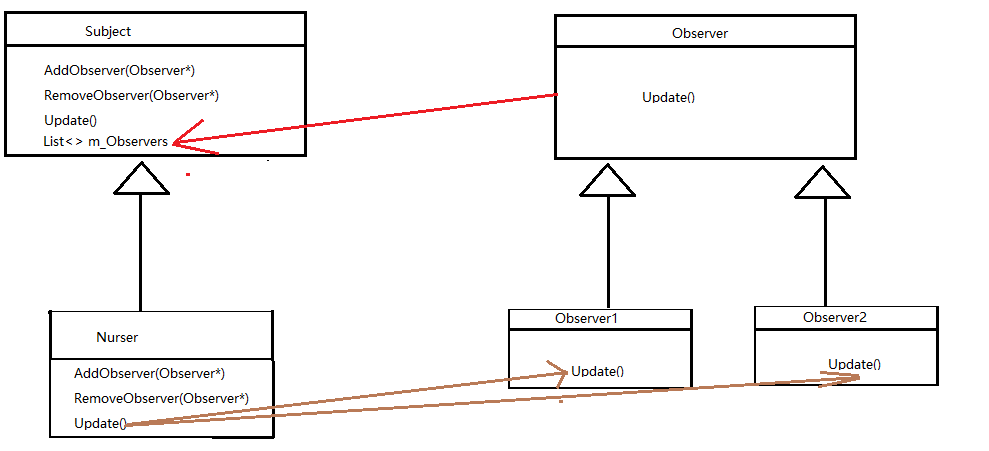
问题提出

打怪升级的游戏原来越完善，距离上线的日子不远了，今天我们遇到了团队作战的问题，当我们组团刷怪的时候，必须要有奶妈(奶爸)在旁边。当奶妈(奶爸)增加各种buff的时候，队员们的攻击力和防御力都会有相应的提升，也就是其他成员就是观察者，时刻观察奶妈(奶爸)身上的变化。我们在编写这部分代码的时候为了减少代码的耦合性就会用到观察者模式。

Observer(观察者)模式

当对象之间是一对多的依赖关系，且当这个被依赖的对象发生变化的时候，所有依赖他的其他对象得到通知并被自动更新。这种更新是每个观察者通过查询自身的状态，在此基础上进行相应的同步。

Observer(观察者)模式简单结构图



Subject：就是被观察者(奶妈)的基类，提供了添加观察者、移除观察者以及更新观察者的方法。保存了所有观察者的对象。

Observer: 观察者的基类。

代码展示

namespace 3wwang

{

class Observer

{

public:

virtual ~Observer(){}

virtual void Update() = 0;

protected:

Observer(){}

}

class Observer1: public Observer

{

public:

Observer1(){}

~Observer1(){}

virtual void Update()

{

std::cout<<"Observer1::Update"<<std::endl;

}

}

class Observer2: public Observer

{

public:

Observer2(){}

~Observer2(){}

virtual void Update()

{

std::cout<<"Observer2::Update"<<std::endl;

}

}

}

namespace 3wwang

{

class Subject

{

public:

virtual ~Subject(){}

virtual void AddObserver(Observer\* observer) = 0;

virtual void RemoveObserver(Observer\* observer) = 0;

virtual void Update() = 0;

protected:

Subject(){}

}

class Nurser: public Subject

{

public:

Nurser(){}

~Nurser(){}

virtual void AddObserver(Observer\* observer)

{

m\_Observers->push\_back(observer);

}

virtual void RemoveObserver(Observer\* observer)

{

if (m\_Observers->size() > 0)

m\_Observers->remove(observer);

}

virtual void Update()

{

std::list<Observer\*>::iterator iter = m\_Observers->begin();

for ( ;iter != m\_Observers->end(); iter++ )

(\*iter)->Update();

}

}

protected:

std::List<Observer\*> m\_Observers;

}

}

namespace 3wwang

{

int main(int argv, char\* argc[])

{

Nurser\* nurser = new Nurser();

Observer1\* obj1 = new Observer1();

Observer2\* obj2 = new Observer2();

nurser->AddObserver(obj1);

nurser->AddObserver(obj2);

nurser->Update();

delete\* obj1;

delete\* obj2;

delete\* nurser;

return 0;

}

}

Osg中观察者模式的应用

我们都知道osg中用到了智能指针，而osg::ref\_ptr就是用到了Observer模式来检测对象是否被引用，如果场景中没有对他的引用则释放相应的内存。Osg::Observer就是这个模式最好的应用。