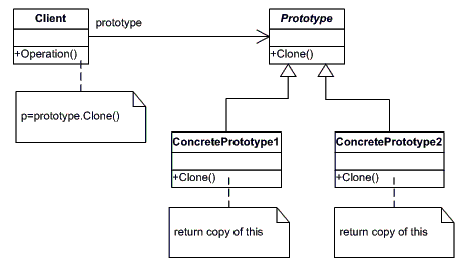
# 1，简介

原型模式：

用原型实例来指向创建对象的种类，通过拷贝原型来创建新的对象。使用原型模式可以尽可能减少客户所知道的类的数目，独立于产品的创建，构成和表示，使运行时动态配置应用成为可能。上面这两点我认为是倒出了原型模式的精髓，具体下面慢慢道来。

# 2，详细说明

## 2.1 图示说明



如上图所示，Client作为客户存储着具体原型的指针，当它需要创建对象的时候，它调用prototype.Clone方法进行对象的创建。它不关心prototype到底是哪一个具体原型，它不关心prototype是如何创建和表示，它唯一关心的就是prototype原型可以通过Clone方法返回一个与prototype本身一致的对象就可以了。

## 2.2 精髓解释

对上面“原型模式精髓”一说的例子解释：prototype指针指向**具体的原型实例**，它可能指向Concrete1，也可能指向Concrete2，通过调用它的Clone方法就可以创建新的对象，至于**如何创建的，它不知晓，至于有多少个具体原型，它也不知晓**，这就是说它大大减少了Client需要知道的原型类的数目，它不知道是不是有Concrete1和Concrete2，亦或者是Concrete3，甚至有可能更具体的子类，这**降低Client类与Prototype类层次的耦合**。**动态配置**又是如何说起呢？prototype原型指针指向哪一个具体的对象，这个可以在Client初始化的时候硬编码，当需要修改它的时候，更改prototype的指针就可以了，这当然需要修改代码，不算是真正的对修改关闭。动态配置可以采取运行时候动态读取配置文件来的办法，通过使用一个原型管理器，它读取配置文件来保存原型实例的一个注册表，当需要创建对象的时候，客户向此注册表来请求原型。

## 2.3 举例说明

有一个需要音频编解码的应用，它需要能够动态的配置是采用PCM，MP3，G729的一种来进行编解码。现有一个Code的类层次，Code父类，CodePCM， CodeMp3，CodeG729子类， 以及PrototypeManger原型管理器(可以以单例模式实现)， Client客户类持有一个Code\*指针。

具体操作：用户配置文件可以配置Code=MP3,PCM,G79,UseCode=Mp3然后PrototypeManage读取用户配置文件配置的Code，从而调用RegisterCodeForKey(“MP3”, new CodeMp3())方法来将CodeMP3注册到PrototypeManager中，对于另外的两种编码采用类似的操作。Client在需要具体Code\*实例的时候，首先读取文件获取UseCode对应的为MP3，然后通过p = GetCodeByKey(“MP3”)来获得具体的MP3原型指针，对于Client来说，它并不知道具体的编码是什么，它仅仅持有一个Code\*的指针，而具体的Code编码类型是通过读取配置文件得到的。这里面的动态配置一个是Code的编码的类型可以动态的配置，可以运行时添加，删除，更改编码类型，同时对于客户的使用的编码，同样可以通过配置文件中读取，然后请求原型管理器返回具体的实例，这样Code和Client类就实现了很大程序的解耦，同时Code本身又可以动态的添加和删除，这个是通过原型管理器实现的，原型管理器可以是一个Map的封装，它存储关键字key和关键字对应的具体原型实例，客户可以检索相应的原型。

核心代码：

#include <iostream>

#include <map>

using namespace std;

//抽象编解码类

class Code

{

virtual Code\* Clone() const = 0;

}

//三个具体的编解码类

class CodeMP3 : public Code

{

Code\* Clone() {return new CodeMP3(\*this);}

};

class CodePCM : public Code

{

Code\* Clone() {return new CodePCM(\*this);}

};

class CodeG729 : public Code

{

Code\* Clone() {return new CodeG729(\*this);}

};

class PrototypeManager

{

public:

bool RegisterCodeForKey(string key, Code\* pCode){

m\_codeMap[key] = pCode;

return true;

}

Code\* GetCodeByKey(string key){

return m\_codeMap[key];

}

void Initialize(const char\* filename)

{

//Code = PCM, MP3, G729

//截取字符串，然后根据字符串的值注册到系统中。

while(1)

{

string strCode;

//begin to read strCode,如果到达文件尾部，终止循环

if(strCode.compare("PCM") == 0 )

RegisterCodeForKey("PCM", new CodePCM());

else if(strCode.compare("MP3") == 0)

RegisterCodeForKey("MP3", new CodeMP3());

else if(strCode.compare("G729") == 0)

RegisterCodeForKey("G729", new CodeG729());

}

}

static PrototypeManager\* GetInstance(){

static PrototypeManager instanceManager;

return &instanceManager;

}

private:

map<string, Code\*> m\_codeMap;

};

class client

{

public:

void Initialize(const char\* filename){

//初始化原型管理器，从配置文件读取并动态配置原型

PrototypeManager.GetInstance()->Initialize(filename);

string strCode;

//read file to get UserCode's value "MP3" //UseCode=MP3

m\_pUseCode = PrototypeManager.GetInstance()->GetCodeByKey("strCode")

}

//如果需要运行时修改，可以修改配置文件后，调用Reload操作

void Reload(const char\* filename)

{

this->Initialize(filename);

}

private:

Code\* m\_pUseCode;

};

# 主要难点

## 3.1 Clone方法的具体实现问题

Clone是进行深复制还是浅复制，这个需要仔细的斟酌，一般需要浅复制就可以了，但是有些特殊情况，需要注意

## 3.2用户需要定制Clone

在Clone完对象之后，客户可能需要进行一些额外的初始化操作，来设定对象的工作状态以及其他对象内部参数。毫无疑问，在Clone方法中直接携带参数不是一个明智的选择，因为对于参数的可能数目以及类型都没有一个统一的标准，因此具体原型类一个可能需要额外的Initialize的初始化操作来设定对象的内部工作状态。这里需要考虑深copy下的对象是不是进行的重复创建。

后记：

网上很多原型模式的博客，很多都是一个抽象原型类，然后两个具体原型类，一个main函数，然后打印一段输出就完了，看着总感觉自己抓不住它的精髓(过去我也是这样子)。今天重读了GOF的原型模式，感觉过去很多没有看懂的地方现在又懂了不少，于是按照自己的思考写了上文。本文主要是自己对原型模式的看法和自己的理解。语言看起来不算那么正式，却又处处是自己对它的理解的精髓所在。关于原型模式和抽象工厂模式，没有时间写了，关键一点就是原型模式省略了抽象工厂模式所需要的与产品一致的工厂的继承层次结构。原型模式可以仅仅只有一个工厂类，它维持着Prototype\*的指针，可以通过原型工厂的构造函数中传入具体的原型，从而可以实现不同的工厂。

例如：

CodeFactory\* mp3Factory = new CodeFactory(new CodeMP3());

CodeFactory\* g729Factory = new CodeFactory(new CodeG729());

而不需要继承两个子类工厂来完成这样的事情。不过您可能新一看这个不太像抽象工厂的东东，那是因为我的举的例子就只有一种产品，抽象工厂是一个产品组，您可以在加入一种视频编码就可以了。

例如：

CodeFactory\* MPFactory = new MPFactory(new CodeAudioMP3(), new CodeVideoMP4);

CodeFactory保留着CodeAudio\*和CodeVideo\*的编解码指针来指向具体的音视频编解码。

音频和视频编解码就是一个产品族了。通过传入不同的具体产品实现了不同的原型工厂。如果您问它与抽象工厂的本质区别，那我会告诉你请关注组合和继承。明天有空我会分析一下装饰者模式，让您更容易看到组合的灵活性。