# 背景

在软件设计中，我们经常需要向某些对象发送请求，但是并不知道请求的接收者是谁，也不知道被请求的操作是哪个，我们只需在程序运行时指定具体的请求接收者即可，此时，可以使用命令模式来进行设计，使得请求发送者与请求接收者消除彼此之间的耦合，让对象之间的调用关系更加灵活。

命令模式可以对发送者和接收者完全解耦，发送者与接收者之间没有直接引用关系，发送请求的对象只需要知道如何发送请求，而不必知道如何完成请求。这就是命令模式的模式动机。

# 定义

命令模式(Command Pattern)：将一个请求封装为一个对象，从而使我们可用不同的请求对客户进行参数化；对请求排队或者记录请求日志，以及支持可撤销的操作。

命令模式是一种对象行为型模式，其别名为动作(Action)模式或事务(Transaction)模式。

命令模式可以将请求发送者和接收者完全解耦，发送者与接收者之间没有直接引用关系，发送请求的对象只需要知道如何发送请求，而不必知道如何完成请求。

# 模式结构

命令模式包含如下角色：

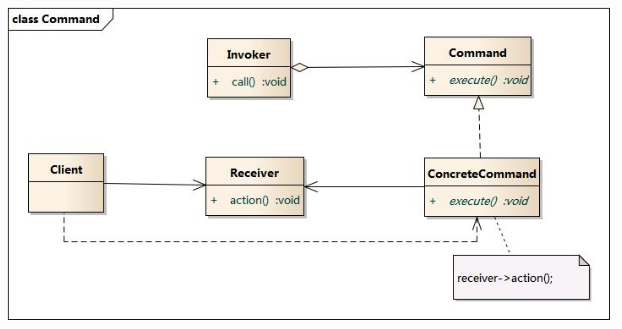
1、Command: 抽象命令类

2、ConcreteCommand: 具体命令类

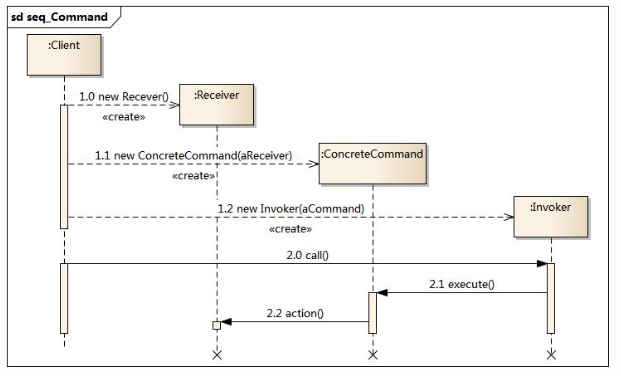
3、Invoker: 调用者

4、Receiver: 接收者

5、Client:客户类



# 时序图



# 代码

#include <iostream>

#include "ConcreteCommand.h"

#include "Invoker.h"

#include "Receiver.h"

using namespace std;

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

{

Receiver \* pReceiver = new Receiver();

ConcreteCommand \* pCommand = new ConcreteCommand(pReceiver);

Invoker \* pInvoker = new Invoker(pCommand);

pInvoker->call();

delete pReceiver;

delete pCommand;

delete pInvoker;

return 0;

}

// Receiver.h

// Implementation of the Class Receiver

#if !defined(EA\_8E5430BB\_0904\_4a7d\_9A3B\_7169586237C8\_\_INCLUDED\_)

#define EA\_8E5430BB\_0904\_4a7d\_9A3B\_7169586237C8\_\_INCLUDED\_

class Receiver

{

public:

Receiver();

virtual ~Receiver();

void action();

};

#endif // !defined(EA\_8E5430BB\_0904\_4a7d\_9A3B\_7169586237C8\_\_INCLUDED\_)

// Receiver.cpp

// Implementation of the Class Receiver

#include "Receiver.h"

#include <iostream>

using namespace std;

Receiver::Receiver(){

}

Receiver::~Receiver(){

}

void Receiver::action(){

cout << "receiver action." << endl;

}

// ConcreteCommand.h

// Implementation of the Class ConcreteCommand

#if !defined(EA\_1AE70D53\_4868\_4e81\_A1B8\_1088DA355C23\_\_INCLUDED\_)

#define EA\_1AE70D53\_4868\_4e81\_A1B8\_1088DA355C23\_\_INCLUDED\_

#include "Command.h"

#include "Receiver.h"

class ConcreteCommand : public Command

{

public:

ConcreteCommand(Receiver \* pReceiver);

virtual ~ConcreteCommand();

virtual void execute();

private:

Receiver \*m\_pReceiver;

};

#endif // !defined(EA\_1AE70D53\_4868\_4e81\_A1B8\_1088DA355C23\_\_INCLUDED\_)

// ConcreteCommand.cpp

// Implementation of the Class ConcreteCommand

#include "ConcreteCommand.h"

#include <iostream>

using namespace std;

ConcreteCommand::ConcreteCommand(Receiver \*pReceiver){

m\_pReceiver = pReceiver;

}

ConcreteCommand::~ConcreteCommand(){

}

void ConcreteCommand::execute(){

cout << "ConcreteCommand::execute" << endl;

m\_pReceiver->action();

}

// Invoker.h

// Implementation of the Class Invoker

#if !defined(EA\_3DACB62A\_0813\_4d11\_8A82\_10BF1FB00D9A\_\_INCLUDED\_)

#define EA\_3DACB62A\_0813\_4d11\_8A82\_10BF1FB00D9A\_\_INCLUDED\_

#include "Command.h"

class Invoker

{

public:

Invoker(Command \* pCommand);

virtual ~Invoker();

void call();

private:

Command \*m\_pCommand;

};

#endif // !defined(EA\_3DACB62A\_0813\_4d11\_8A82\_10BF1FB00D9A\_\_INCLUDED\_)

// Invoker.cpp

// Implementation of the Class Invoker

#include "Invoker.h"

#include <iostream>

using namespace std;

Invoker::Invoker(Command \* pCommand){

m\_pCommand = pCommand;

}

Invoker::~Invoker(){

}

void Invoker::call(){

cout << "invoker calling" << endl;

m\_pCommand->execute();

}

**另外一个案例：**

#define CRT SECURE NOWARVINGS

#include <iostream>

#include<vector>

#include<queue>

#include<algorithm>

#include<Windows.h>

using namespace std;

//发送协议类

class HandleClientProtocol{

public:

//处理增加金币

void AddMoney(){

cout << "给玩家增加金币!" << endl;

}

//处理增加钻石

void AddDiamond(){

cout << "给玩家增加钻石!" << endl;

}

//处理玩家装备

void AddEquipment(){

cout << "给玩家穿装备!" << endl;

}

//处理玩家等级

void addLevel(){

cout << "给玩家增加等级!" << endl;

}

};

// 命令接口

class AbstractCommand{

public:

virtual void handle() = 0;//处理客户端请求的接口

};

//处理增加金币请求

class AddMoneyCommand :public AbstractCommand{

public:

HandleClientProtocol\*pProtocol;

public:

AddMoneyCommand(HandleClientProtocol\*pProtocol){

this->pProtocol = pProtocol;

}

virtual void handle(){

this->pProtocol->AddMoney();

}

};

//处理增加钻石的请求

class AddDiamondCommand :public AbstractCommand{

public:

AddDiamondCommand(HandleClientProtocol\*pProtocol){

this->pProtocol = pProtocol;

}

virtual void handle(){

this->pProtocol->AddDiamond();

}

public:

HandleClientProtocol\*pProtocol;

};

//处理增加装备的请求

class AddEquipmentCommand : public AbstractCommand {

public:

HandleClientProtocol\* pProtocol;

public:

AddEquipmentCommand(HandleClientProtocol\* pProtocol){

this->pProtocol = pProtocol;

}

virtual void handle(){

this->pProtocol->AddEquipment();

}

};

//处理玩家升级的请求

class AddLevelCommand :public AbstractCommand{

public:

HandleClientProtocol\*pProtocol;

public:

AddLevelCommand(HandleClientProtocol\*pProtocol){

this->pProtocol = pProtocol;

}

virtual void handle(){

this->pProtocol->addLevel();

}

};

//服务器程序

class Serser{

public:

queue<AbstractCommand\* > mCommands;

public:

void startHandle(){

while (!mCommands.empty()) {

Sleep(2000);

AbstractCommand\* command = mCommands.front();

command->handle();

mCommands.pop();

}

}

void addRequest(AbstractCommand\* command) {

mCommands.push(command);

}

};

void test01(){

HandleClientProtocol\*pProtocol = new HandleClientProtocol;

//客户端增加金市的请求

AbstractCommand\*addmoney = new AddMoneyCommand(pProtocol);

//客户端增加钻石的请求

AbstractCommand\*adddiamond = new AddDiamondCommand(pProtocol);

//客户端穿装备的请求

AbstractCommand\*addequpment = new AddEquipmentCommand(pProtocol);

//客户端升级请求

AbstractCommand\*addlevel = new AddLevelCommand(pProtocol);

Serser\* server = new Serser;

//将客户端请求加入到处理的队列中

server->addRequest(addmoney);

server->addRequest(adddiamond);

server->addRequest(addequpment);

server->addRequest(addlevel);

//服务器开始处理请求

server->startHandle();

}

int main()

{

test01();

return 0;

}

# 分析

命令模式的本质是对命令进行封装，将发出命令的责任和执行命令的责任分割开。

每一个命令都是一个操作：请求的一方发出请求，要求执行一个操作；接收的一方收到请求，并执行操作。

命令模式允许请求的一方和接收的一方独立开来，使得请求的一方不必知道接收请求的一方的接口，更不必知道请求是怎么被接收，以及操作是否被执行、何时被执行，以及是怎么被执行的。

命令模式使请求本身成为一个对象，这个对象和其他对象一样可以被存储和传递。

命令模式的关键在于引入了抽象命令接口，且发送者针对抽象命令接口编程，只有实现了抽象命令接口的具体命令才能与接收者相关联。

# 特点

## 优点

降低系统的耦合度。

新的命令可以很容易地加入到系统中。

可以比较容易地设计一个命令队列和宏命令（组合命令）。

可以方便地实现对请求的Undo和Redo。

## 缺点

使用命令模式可能会导致某些系统有过多的具体命令类。因为针对每一个命令都需要设计一个具体命令类，因此某些系统可能需要大量具体命令类，这将影响命令模式的使用。

# 适用环境

在以下情况下可以使用命令模式：

系统需要将请求调用者和请求接收者解耦，使得调用者和接收者不直接交互。

系统需要在不同的时间指定请求、将请求排队和执行请求。

系统需要支持命令的撤销(Undo)操作和恢复(Redo)操作。

系统需要将一组操作组合在一起，即支持宏命令

# 模式应用

很多系统都提供了宏命令功能，如UNIX平台下的Shell编程，可以将多条命令封装在一个命令对象中，只需要一条简单的命令即可执行一个命令序列，这也是命令模式的应用实例之一。

# 模式扩展

宏命令又称为组合命令，它是命令模式和组合模式联用的产物。

宏命令也是一个具体命令，不过它包含了对其他命令对象的引用，在调用宏命令的execute()方法时，将递归调用它所包含的每个成员命令的execute()方法，一个宏命令的成员对象可以是简单命令，还可以继续是宏命令。执行一个宏命令将执行多个具体命令，从而实现对命令的批处理。

# 总结

在命令模式中，将一个请求封装为一个对象，从而使我们可用不同的请求对客户进行参数化；对请求排队或者记录请求日志，以及支持可撤销的操作。命令模式是一种对象行为型模式，其别名为动作模式或事务模式。

命令模式包含四个角色：抽象命令类中声明了用于执行请求的execute()等方法，通过这些方法可以调用请求接收者的相关操作；具体命令类是抽象命令类的子类，实现了在抽象命令类中声明的方法，它对应具体的接收者对象，将接收者对象的动作绑定其中；调用者即请求的发送者，又称为请求者，它通过命令对象来执行请求；接收者执行与请求相关的操作，它具体实现对请求的业务处理。

命令模式的本质是对命令进行封装，将发出命令的责任和执行命令的责任分割开。命令模式使请求本身成为一个对象，这个对象和其他对象一样可以被存储和传递。

命令模式的主要优点在于降低系统的耦合度，增加新的命令很方便，而且可以比较容易地设计一个命令队列和宏命令，并方便地实现对请求的撤销和恢复；其主要缺点在于可能会导致某些系统有过多的具体命令类。

命令模式适用情况包括：需要将请求调用者和请求接收者解耦，使得调用者和接收者不直接交互；需要在不同的时间指定请求、将请求排队和执行请求；需要支持命令的撤销操作和恢复操作，需要将一组操作组合在一起，即支持宏命令。