専門単語：

policyポリシー　策

動作　どうさ

柔軟　じゅうなん

Variation バリエーション

eventイベント

objectオブジェクト

observerオブザーバー

callbackコールバック

Virtual function仮想関数

templateテンプレート

多重 たじゅう

継承　けいしょう

シンタックス syntax 句法

セマンティック semantics 意味/ ロジック？

カプセル　capsule 封装

プリミティブ　primitive　原函数

可搬性　portability　異種のデータ処理システム上で実行できるプログラムの能力

デフォルト　default

コンストラクタ　constructor

フレームワーク　framework　枠組み

カスタマイズ　customize　注文に応じて製作する

リファレンス　reference

コンポーネント　component　構成要素

メカニズム　mechanism　機構

スケーラビリティ　scalability　拡張性

トレードオフ　trade-off　权衡

変数 へんすう

判る　わかる

課す　加える？

解釈 かいしゃく

高々　たかだか　せいぜい

第一義　大事な事柄

多重継承の欠点：（組み合わせだけ）

1.単純なメカニズム：各メンバーにアクセスできるだけ

2.型情報が足りない： 判っていない型のオブジェクトを関数の中で生成✖NG

3.状態の操作？：基底のクラスを仮想継承しなければならない

テンプレート：コンパイル時点で生成する

部分特殊化

template<typename T, typename U> class smart\_ptr{};

template<typename U>class smart\_ptr<Widget, U>{};

欠点：

1.構造を特殊化✖NG

2.スケーラビリティが低い：複数のパラメータがあるとき、関数を部分特殊化は✖NG  
3.デフォルト関数は高々１つ

////////////////////////////////////////////////////////////////////////////

Policies（继承+模板）：定义class 或 template的接口

将class能多种方式实现的行为提取出来成为policy，例如（内存分配，指针的安全性鉴别）

且policy间没有互相影响

**policy class**

template<class T>

struct op\_new\_creator

{

static T\* create(){ return new T;}

//... set() get() 或其他规范

protected: //防止非派生类（指针）调用析构

~op\_new\_creator() {}

};

//... 其他都具备create()的policy class

1.创造出来的class主要用于继承或被内含于其他实例中

2.规定了规范 Ex:必须有create()  
3.用户可以继承这个策略添加其他成员函数

4.策略上下转型最好显性控制，控制成员变量和转换的合法性

template<

class T, //策略1用到的类型也需要额外设置

template<class> class creation\_policy = op\_new\_creator,

class read\_policy = reader>

class widget\_manager : public creation\_policy<T>, public read\_policy

{

public:

//不同的policy对应不同的create()方法

inline void create()

{

creation\_policy<T>::create();

};

void do\_something()

{

//host class内部也可以利用policy class直接产生所需的class

widget\* wid = creation\_policy<widget>().create();

reader().read();

}

//...

};

policy class都具备create()的策略，因此可组合成host class

1. 用户想构建class时，需要在host class 的template中采用一个或多个policy class
2. 简单的多重继承时，虚函数是动态连接，而policy 是静态连接则类型安全(不存在指针找不到成员函数行为)
3. 不允许运用不支持的policy（编译期报错，Ex：缺乏create()）
4. 用户一般看不到获取不到host class内部细节，只负责构建和调用行为

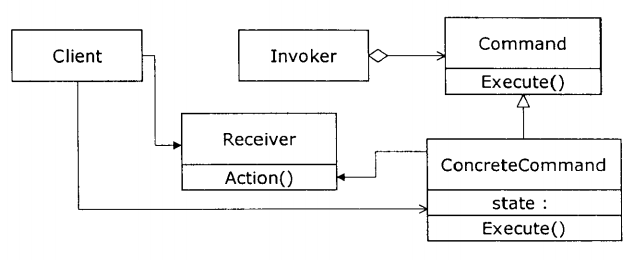
用户使用时，用策略组合出host class再予以调用

typedef widget\_manager<widget,op\_new\_creator, reader> normal\_widget\_mag;

//....

/////////////////////////////////////////////////////////////////////////////////////////

**Command Class**



Command 中间层，分离调用者invoker和被调用者receiver

Invoker 某时刻获取 Command的Execute()函数

Receiver 被Command调用

1. 接口分离（调用者和被调用者）
2. 时间分离（填充参数时间 和 执行时间 不同）

环境：填充参数就是构建环境

ConcreteCommand负责保存各种环境，和execute()时提供环境

第一种class：（转发式命令）（可考虑泛化仿函数 functor）直接调用receiver对象的函数

第二种class：（主动式命令）（必定和程序强关联）调用其他对象的函数，以及更多的逻辑

/\*保留\*/

/////////////////////////////////////////////////////////////////////////////////////////

**Singleton Class**

全局性，唯一

负责自身生成和销毁

静态的问题：

1. 静态函数不能是虚函数
2. 无法控制初始化和清理

构造和复制和赋值函数要显式为private

析构也要private （防止无意删除）

Static singleton& Instance()获取时生成 动态 （编译时无法保证已生成）

或cpp中定义 singleton singleton::instance\_; 静态生成

或者！ 直接Instance()内的static局部变量（程序关闭时自动删除）

/\*保留\*/

/////////////////////////////////////////////////////////////////////////////////////////

**Object Factories**

单继承的缺点：

多态利用new 产生对象时，更改类型时必须修改后面的对象类型。（对象生成是静态绑定的）

对象工厂使用情况：

1. 程序库要产生和操作用户的自定义对象
2. 利用字符串等标识符生成对应C++类型 object-type-object

/////////////////////////////////////////////////////////////////////////////////////////

**Abstract Factory**

1. 将对象分类，简单类别的怪由简单工厂生成，困难类别的怪由困难工厂生成
2. 两类工厂派生至抽象工厂，并必须所有重载怪物生成函数

设计模式

10

多线程

11

指针

7

内存

4

序列化

3

2

编译期检查错误

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

//编译器执行expression表达式，成功为1，失败则为0，0导致uname的大小为0，编译器出错

#define STATIC\_CHECK(expression){ char unname[(expression)?1:0];}

或

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

//false时，编译器输出CompileTimeError<false>未定义

template<bool> struct CompileTimeError;//只声明

template<> struct CompileTimeError<true> {};//只定义了true

#define STATIC\_CHECK(expression) \

(CompileTimeError<(expression) != 0>)

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

输出错误信息：

template<bool> struct CompileTimeChecker//隐式定义true

{

CompileTimeChecker(...); //非定量任意参数表

};

// ## ：ERROR\_后面加入msg的变量名

// 显示定义false导致无法强制转换，触发编译器输出

template<> struct CompileTimeChecker<false>{};

#define STATIC\_CHECK(expr,msg) \

{\

class ERROR\_##msg {};\

(void)sizeof(CompileTimeChecker<(expr)>(ERROR\_##msg()));\

}

使用方式

STATIC\_CHECK(表达式, 错误时要输出的信息);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Partial Template Specialization

模板偏特化：执行最匹配的显示定义

template<class button\_arg>

class widget<button<button\_arg>, controller>{/\*\*/ };

装配了conteroller和可设置button\_arg参数button策略的widget的显示定义。

不能偏特化函数

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Local class

不能写static，不能被继承

可以使用作用域内的泛型

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

常整数映射为类型

//编译期的switch / if

//编译期的switch

template<int v>

struct int\_to\_type

{

enum {value = v};

};

使用Ex:一个带新建函数的类 和 一个带深拷贝函数类 都被允许装进 模板类 中同一个模板参数时，

可以控制编译该参数的 深拷贝函数 还是 新建函数

template <typename T, int type>

class use\_class

{

private:

void do\_s(T\* obj, int\_to\_type<0>) { obj->copy(); };

void do\_s(T\* obj, int\_to\_type<1>) { obj->create(); };

public:

void do\_s(T\* obj) { do\_s(obj, int\_to\_type<type>); };

};

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

类型映射类型

//解决函数无法偏特化机制，用于代替其他不偏特化的模板参数

template<typename T>

struct type\_to\_type

{

typedef T original\_type;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

类型选择（可用于静态选择数组存储哪个类型的变量）

//true 为T类型

template<bool flag, typename T, typename U>

struct select

{

typedef T result;

};

//false 为U类型

template<typename T, typename U>

struct select<false, T, U>

{

typedef U result;

};

使用

std::vector<select<false, void\*, int>> my\_group;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

编译时检测T和U的可转换性(代替 dynamic\_cast())

template<typename T, typename U>

class conversion

{

typedef char small;

class big { char dummy[2]; };

//sizeof()不执行内部的表达式，所以只需要声明

static small test(U); //只有u作参才会返回small

static big test(...); //其他类型都返回 big

static T make\_T(); //只声明

public:

enum //存在T到U的转换exists = 0 或 1

{exists = sizeof(test(make\_T())) == sizeof(small)};

enum//存在双向转换

{exists\_2\_way = exists && conversion<U, T>::exists};

enum { same\_type = false}; //默认T U不同类型

};

//偏特化定义T U相同类型

template<class T>

class conversion<T, T>

{

public:

enum{ exists = 1, exists\_2\_way = 1, same\_type = 1};

};

使用

bool can\_convert = conversion<double, int>::exists;

//检测继承，U是不是T的派生类

//const 以可以加以检测const的指针

//void\* 防止 T 是void

#define SUPER\_SUBCLASS(T,U) \

{conversion<const U\*, const T\*>::exists && \

!conversion<const T\*, const void\*>::same\_type}

//更严格，防止 T U是同类型

#define SUPER\_SUBCLASS\_STRICT(T,U) \

{SUPER\_SUBCLASS(T,U) && \

!conversion<const T, const U>::same\_type}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

std::type\_info

const std::type\_info& a = typeid();//查询对象类型

const char\* std::type\_info::name()//输出 class ...<...,...>

特性：

1. 没有拷贝和构建函数
2. Typeid()返回的都是static变量，但也不一定是同一个

需要派生重载一下

/\*保留\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

class null\_type; //无类型，无定义 让报错

struct empty\_type; //空类型，可作为模板的空参数

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Type traits 编译期根据类型判断

//检测T是不是指针

template<typename T>

class type\_traits

{

private:

template <class U> struct ptr\_traits

{

enum {result = false};//不是为false

typedef null\_type ptr\_type;

};

//显示定义 T\*的模板

template <class U> struct ptr\_traits<U\*>

{

enum { result = true };//是为true

typedef U ptr\_type;

};

public:

enum { is\_ptr = ptr\_traits<T>::result };

typedef ptr\_traits<T>::ptr\_type ptr\_type;//重命名

};

/\*保留\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

C++设计准则：尽可能静态模型，必要时才动态模型

面向对象： 通过继承，虚函数，使得在执行期才决定用那哪个函数。因此多态得以提高二进制码的复用性和扩充性（系统自动将虚成员函数分派给合适的派生对象）

C++ class是程序员产生的，object是程序产生的，执行期无法做新的class，编译期没有object

是因为静态类型是优化的关键

内存泄漏 memory leak ，new malloc分配 并丢失所有ptr reference

资源泄漏 resource leak 网络连接 OS互斥体 通信 COMreference等

无命名namespace{} = final