现代软件设计技术

潘爱民

http://www.icst.pku.edu.cn/compcourse

内容

- 复习
- Generic Programming
- 补充一些patterns
- 构造框架(framework)的技术
- 构造可重用类库的技术

复习：creational patters

- Factory Method
  - 本质：用一个virtual method完成创建过程
- Abstract Factory
  - 一个product族的factory method构成了一个factory接口
- Prototype
  - 通过product原型来构造product，Clone+prototype manager
- Builder
  - 通过一个构造算法和builder接口把构造过程与客户隔离开
- Singleton
  - 单实例类型，如何构造这单个实例？如何访问这单个实例？
- Finder
  - 把对象的获取过程与客户隔离开

复习：Structural patterns

- Adapter 、bridge、facade
  - adapter用于两个不兼容接口之间的转接
  - bridge用于将一个抽象与多个可能的实现连接起来
  - facade用于为复杂的子系统定义一个新的简单易用的接口
- composite、decorator和proxy
  - composite用于构造对象组合结构
  - decorator用于为对象增加新的职责
  - proxy为目标对象提供一个替代者
- flyweight
  - 针对细粒度对象的一种全局控制手段

复习：Behavioral Patterns

- Command
  - 用对象封装命令，使得命令可以被传递、记录、排队等
- Iterator
  - 把对聚合体对象的访问封装起来
- Observer
  - 建立起一对多的通信模型，特别适合于更新通知和事件模型
- Strategy
  - 把一个对象或者类的某些行为封装到另一个单独的对象中
- Visitor
  - 把对一个结构模型的操作单独组织到一个类中

复习：Behavioral Patterns(续一)

- Chain of Responsibility
  - 请求的处理过程，沿着链传递，decouple发送和接收方
- Interpreter
  - 在类层次结构中，在特定环境的“interpret”过程
- Mediator
  - 用一个mediator来decouple各同等单元
- Memento
  - 在对象之外保存对象的内部状态
- State
  - 把一个对象的状态独立出来，动态可变换状态对象的类型
- Template Method
  - 在基类中定义算法的骨架，把某些细节延迟到子类中

复习：Behavioral Patterns(续二)

- Strategy、Iterator、Mediator、State、command
  - 用一个对象来封装某些特性，比如变化、交互、状态、行为、命令
- Mediator、Observer
  - Observer建立起subject和observer之间的松耦合连接
  - mediator把约束限制集中起来，-〉中心控制
- command、Chain of Responsibility、interpreter
  - command模式侧重于命令的总体管理
  - Chain of Responsibility侧重于命令被正确处理
  - interpreter用于复合结构中操作的执行过程

Generic programming

- 思想：
  - “通过参数化技术达到重用的目的(reuse through parameterization)”，使得组件更容易被定制。实现静态配置代码，从而获得很高的效率。 Generic Programming可以消除类型和算法之间本来不必要的相依性
- STL是成功的generic programming典型
- generic programming限制代码产生的方式：用实体类型代替类型参数，使这些预先编制好的代码模型成为真正的代码，无法产生完全新的代码。

相关的programming

- Aspect-Oriented Programming (AOP)
  - aspect-oriented decomposition and composition techniques
  - 把问题分解为功能单元(组件)和aspect
  - 在AOP系统中，组件和aspects(交织)组合起来得到一个系统的具体实现。交织组合既可以在compile-time，也可以在runtime
  - 参考：http://www.parc.xerox.com/csl/projects/aop/
- generative programming
  - Generative Programming ：建立起一族软件系统的模型，在特定的要求下，利用一些基本的、可重用的组件，通过配置，能够自动根据需要产生一个高度定制和优化的软件系统实例。
  - Generative Programming涉及到软件开发过程的各个阶段。
  - Domain Engineering

C++ Generic Programming

- Template技术
  - 使C++成为two-level language
    - metaprogram
  - 从科学计算用途-〉一般性的抽象，即generic programming
  - 对于编译器
    - 代码产生、优化
    - 在编译时刻，实现静态绑定 —— partial evaluation
  - 对于库开发人员
    - Active libraries，提供一种抽象的功能，并且控制优化过程
    - 许多技术，如policy(traits)、编译时刻计算功能
  - 对于应用开发人员
    - 定制template class或者template function

C++ Generic Programming(续)

- 设计思想
  - 编译时刻程序设计，类似于logic-programming即，在compile-time让编译器完成一些功能，例如
    - 静态的计算功能
    - if/else，loop，switch
    - 对type进行一些基本的逻辑操作(编程)
  - 保证类型安全
    - 宁可要compile-time error，也不要runtime error
  - 尽可能地泛化 —— 抽象性
  - 跟传统的设计方法结合起来，比如利用patterns

Template基础 —— class template

```
template<typename T, size_t MAX_ELEMS = 8 >
class Array {
    public:
      T& operator[](size_t n) {
          if (n < 0 || n>=MAX_ELEMS) throw "Out of bounds! ";
          return m_rg[n];
      }
      protected:
      T m_rg[MAX_ELEMS];
};
```

使用：

```
Array<long, 8> a1;
Array<char, 200> a2;
```

Template基础 —— 模板特化 template specialization

```
template<>
class Array<char, 256> {
public:
    char& operator[](size_t n) {
        if (n < 0 || n>=256) throw "Out of bounds!";
        return m_sz[n];
    }
    bool operator== (const Array<char, 256>& rhs) {
        return strcmp(m_sz, rhs.m_sz) == 0;
    }
protected:
    char m_sz[256];
};
```

Template基础 —— 部分模板特化 partial template specialization

```
template<size_t MAX_ELEMS>
class Array<char , MAX_ELEMS> {
public:
    T& operator[](size_t n) {
        if (n < 0 || n>=MAX_ELEMS) throw "Out of bounds!";
        return m_sz[n];
    }
    bool operator== (const Array<char, MAX_ELEMS>& rhs) {
        return strcmp(m_sz, rhs.m_sz) == 0;
    }
protected:
    T m_sz[MAX_ELEMS];
};
```

*Visual C++目前不支持部分模板特化

Template基础 —— 函数模板 function template

```
template < class T >
void Swap(T &a, T&b)
{
    T temp = a ;
    a = b;
    b = temp;
}

template <class T>
T& min(T& a, T& b)
{
    return a < b ? a : b;
}
```

模板参数，compile-time起作用

函数参数，runtime起作用

Template基础 —— 函数对象泛化 generalized functors(function objects)

- functor：重载了operator()的C++对象，扩展了函数指针的概念，可以增加内部状态，可以被当作对象来传递，具有值语义(value semantic)。它可以是template class，也可以不是

```
template < typename ResultType >
class Functor
{
public :
    ResultType operator()();
    // other member function
private :
    // implementation
};
```

用法：

```
Functor<int> MyFunctor(val1);
int Result = MyFunctor();
```

Template技术 —— 代替runtime的if/else

```
if (condition)
  statement1;
else
  statement2;
```



```
// Class declarations
template<bool C>
class ConditionProcess { };

class ConditionProcess <true> {
public:
   static inline void f()
   { statement1; }        // true case
};

class ConditionProcess <false> {
public:
   static inline void f()
   { statement2; }        // false case
};

// Replacement for 'if/else' statement:
ConditionProcess <condition>::f();
```

Compile-time能够确定condition的值

Template技术 —— 代替runtime的switch

```
int i;
switch(i)
{
  case value1:
    statement1;
    break;
  case value2:
    statement2;
    break;

  default:
    default-statement;
    break;
}
```



```
// Class declarations
template<int I>
class SwitchProcess {
public:
   static inline void f() { default-statement; }
};

class SwitchProcess <value1> {
public:
   static inline void f()  { statement1; }
};

class SwitchProcess <value2> {
public:
   static inline void f() { statement2; }
};

// Replacement for switch(i) statement
SwitchProcess <I>::f();
```

Template技术 —— 代替runtime的do循环

```
int i = N;

do {
   statement;
} while (--i > 0);
```



```
template<int I>
class DoProcess {
private:
   enum { go = (I-1) != 0 };
public:
   static inline void f() {
        statement;
        DoProcess <go ? (I-1) : 0>::f();
   }
};

// Specialization provides base case for recursion
class DoProcess <0> {
public:
   static inline void f() { }
};

// Equivalent loop code
DoProcess <N>::f();
```

Template技术 —— 代替runtime的临时变量

```
int countBits(int N)
{
   int bit3 = (N & 0x08) ? 1 : 0,
      bit2 = (N & 0x04) ? 1 : 0,
      bit1 = (N & 0x02) ? 1 : 0,
      bit0 = (N & 0x01) ? 1 : 0;

   return bit0+bit1+bit2+bit3;
}

int i = countBits(13);
```



```
template<int N>
class countBits {
   enum {
        bit3 = (N & 0x08) ? 1 : 0,
        bit2 = (N & 0x04) ? 1 : 0,
        bit1 = (N & 0x02) ? 1 : 0,
        bit0 = (N & 0x01) ? 1 : 0 };

public:
   enum { nbits = bit0+bit1+bit2+bit3 };
};

int i = countBits<13>::nbits;
```

Template技术 —— 计算 Compile-time functions

- 一般原则：
  - 局部变量用enum类型
  - 循环转化为递归，结束条件为一个特化版本
    - 也可以是多重循环，需要用到部分特化特性
  - 条件分支用模板特化解决
  - 效果：以类型为基础，实现各种操作
- 例如
  - sin x = x - x^3/3! + x^5/5! - x^7/7! + …
  - 在编译时刻求pow(x,y)，即x的y次方

Template技术 —— 计算pow(x,y)

```
template<int X, int Y>
struct ctime_pow {
    enum { result = X*ctime_pow<X, Y-1>::result };
};

template<int X>
struct ctime_pow<X, 0>
{
    enum { result = 1};
};
```

用法：

```
const int z = ctime_pow<5,3>::result;
```

Template技术 —— 模板类作为基类

- 某种程度上可以代替模板特化

```
template<size_t MAX_LEN>
class String : public Array<char, MAX_LEN+1> {
public :
    // additional functionality
    bool operator==(const String<MAX_LEN>& rhs)
    {
        return strcmp(m_rg, rhs.m_rg) ==0;
     }
}
```

Template技术 —— 以模板参数作为基类

- 允许用户把自己的类插入到类层次的中间
- 用户提供基类，类库使用基类

```
template<typename Base, typename Policy1>
class Deriving : public Base<Policy1> {

……

}
```

Template技术 —— typelist

- 以类型作为元素构成链

```
template <class T, class U>
  struct Typelist
  {
    typedef T Head;
    typedef U Tail;
  };
```

- typelist各种操作
  - Length
  - TypeAt
  - IndexOf
  - Append
  - Erase
  - Replace
  - MostDerived
  - ……

- 例如Length操作

```
template <class TList> struct Length;
    template <> struct Length<NullType>
    {
      enum { value = 0 };
    };

    template <class T, class U>
    struct Length< Typelist<T, U> >
    {
      enum { value = 1 + Length<U>::value };
    };
```

Template技术 —— typelist(续一)

- typelist用法

```
#define TYPELIST_1(T1) Typelist<T1, NullType>
#define TYPELIST_2(T1, T2) Typelist<T1, TYPELIST_1(T2) >
#define TYPELIST_3(T1, T2, T3) Typelist<T1, TYPELIST_2(T2, T3) >

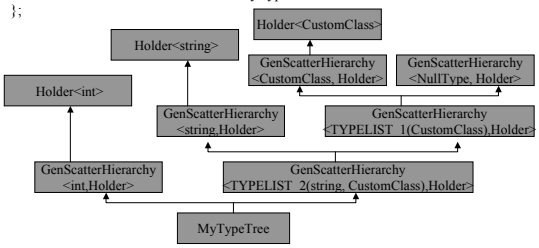
template <class T1, class T2, template <class> class Unit>
  class GenScatterHierarchy<Typelist<T1, T2>, Unit>
    : public GenScatterHierarchy<T1, Unit>
    , public GenScatterHierarchy<T2, Unit>
{
};
template <class AtomicType, template <class> class Unit>
  class GenScatterHierarchy : public Unit<AtomicType>
{
};
template <template <class> class Unit>
  class GenScatterHierarchy<NullType, Unit>
{
};
```

Template技术 —— typelist(续二)

- GenScatterHierarchy用法

```
template <class T>
class Holder
{
    T m_value;
};
```

```
typedef GenScatterHierarchy
<
    TYPELIST_3(int, string, CustomClass),
    Holder
> MyTypeTree
```



Template技术：动态绑定 —— 模拟虚函数多态性

```
template<typename T>
class Array {
public :
    ……
    virtual int Compare(const Array<T>&rhs)=0;
    bool operator<(const Array<T>& rhs)
    { return this->Compare(rhs) < 0; }

    bool operator>(const Array<T>& rhs)
    { return this->Compare(rhs)> 0; }

    bool operator==(const Array<T>& rhs)
    { return this->Compare(rhs) == 0; }
}
```

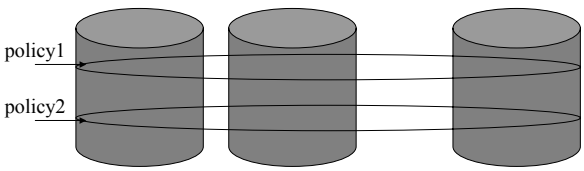
```
template<typename T, typename Deriving>
class Array {
public :
    ……
    bool operator<(const Array<T>& rhs)
    { return static_cast<Deriving *>(this)->
                Compare(rhs) < 0;
    }

    bool operator>(const Array<T>& rhs)
    { return static_cast<Deriving *>(this)->
                Compare(rhs) > 0;
    }

    bool operator==(const Array<T>& rhs)
    { return static_cast<Deriving *>(this)->
                Compare(rhs) == 0;
    }
}
```

policy

- Aliases ： strategy, behavior class, trait, aspect
- 在设计供重用的类或者组件的时候，尽可能地把细节抽象出来，虽然这些细节遍布各处，但它们本身并不构成耦合
- 对问题的垂直分解和水平分解



Policy(续一)

- 想法：用policy来配置一个类或者组件
- 把影响问题的因素分解成几个不相关的方面，并且用policy class来表达，例如
  - 创建策略
  - 线程模型
- policy之间尽可能不相关
  - 一旦有关联，就可能产生一些制约因素，例如引用计数策略(RefCountPolicy)和存储策略(StoragePolicy)
- policy类与host类之间可以在runtime进行组合，也可以在compile-time进行组合

Policy(续二)

- 用模板参数作为policy class
  - 编译器在compile-time对host class进行配置
- 例如：

```
template< typename ThreadingModel=SingleThreaded >
class Widget
……

template<template<class Created> class  CreationPoly>
class WidgetManager
……
```

Policy(续三)

- Policy组合，用m+n个类组合成n*m种可能，例如：

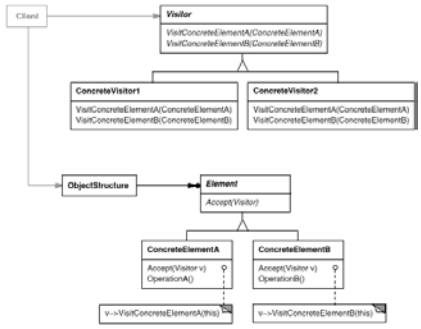
```
template
<
    class T,
    template <class> class CheckingPolicy,
    template<class> ThreadingModel
>
class SmartPtr;
……
typedef SmartPtr<Widget, EnsureNotNull, SingleThreaded> SafeWidgetPtr;

template<class T>struct EnsureNotNull
{
    static void Check(T*& ptr)
    { if (!ptr) ptr = GetDefaultValue(); }
}
```

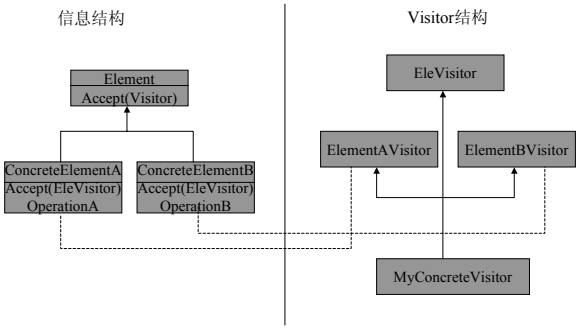
Policy(续四)

- 用模板实现Policy的意义
  - 在compile-time配置host class，有助于产生更为高效的代码，而generic更加general
  - 以类为基础配置一个类，所以policy往往包含静态方法，即class-level的方法
- 在设计可重用类或者组件的时候，根据需要提取出policy，并确定policy的接口
  - 实现类似hook的思想，由用户提供具体的policy class，从而把有些行为往后延迟
- 缺点：
  - 要求使用者对于policy有非常的了解，知道每一种策略会影响到问题的哪些方面

Visitor模式(GoF)



Visitor模式改进



Lazy技术

- Lazy Initialization，第一次被访问时初始化

```
Singleton& Singleton::Instance()
{
    if (!pInstance)
    {
        pInstance = new Singleton;
    }
    return *pInstance;
}
```

- COW(Copy on write)，第一次被写入时才拷贝
- Lazy Protocol：DCOM协议

Double-Checked Locking Pattern

- 假设mutex是一个Mutex对象，现在需要对pInstance的访问进行同步，一种方案是

```
Singleton& Singleton::Instance()
{
    Lock guard(mutex);
    if (!pInstance)
    {
        pInstance = new Singleton;
    }
    return *pInstance;
}
```

Double-Checked Locking Pattern(续一)

- 前述的方案效率比较低，只有一次new操作需要保护，其他的只读访问不必同步保护，改进：

```
Singleton& Singleton::Instance()
{
    if (!pInstance)
    {
      Lock guard(mutex);
      pInstance = new Singleton;
    }
    return *pInstance;
}
```

- 问题：race condition又出现了

Double-Checked Locking Pattern(续二)

- Double-Checked Locking Pattern ：

```
Singleton& Singleton::Instance()
{
    if (!pInstance)
    {
      Lock guard(mutex);
      if (!pInstance)
      {
          pInstance = new Singleton;
      }
    }
    return *pInstance;
}
```

Table-driven pattern

- 代替runtime switch
- 问题：从文件中读入一组以CShape为基类的对象，传统的做法：

```
ReadWord(stream, &typeid);
switch (typeid) {
case Line_ID:
      pObj = (CShape *)new CLine;
      pObj->Load(stream);
      break;
case Poly_ID:
    ……
}
pDoc.Add(pObj);
```

Table-driven pattern(续)

- 维护一张表(typeid, fnCreator)
- 根据typeid查找到创建函数，然后创建对象，示例代码如下：

```
ReadWord(stream, &typeid);
FnCreator fn = Table::Instance().Lookup(typeid);
pObj = fn();
pObj->Load(stream);
pDoc.Add(pObj);
```

- 在MFC/ATL大量用到table-driven的技术
- 用多维表可以实现double-dispatch或者multi-dispatch技术

framework

- 领域工程
  - 单个系统 ——〉一类系统
  - 有较强的抽象能力
- 组件库
- 提供定制功能，允许开发人员对于框架主体部分进行修改
- 不同层次上的framework
  - 基于二进制代码的framework，例如MMC
  - 基于源代码的framework，例如MFC

基于二进制的framework

- 接口：
  - 为应用中的组件提供二进制接口
    - 以对象形式封装
    - 以功能为单位
  - 粒度
    - 大而全的接口
    - 小型接口，允许动态发现新的接口
- 通信模型
  - 用户组件与框架进行通信
  - 用户组件之间如何通信？
    - 通过框架传递信息
    - 通过框架建立直通模型

基于源代码的framework

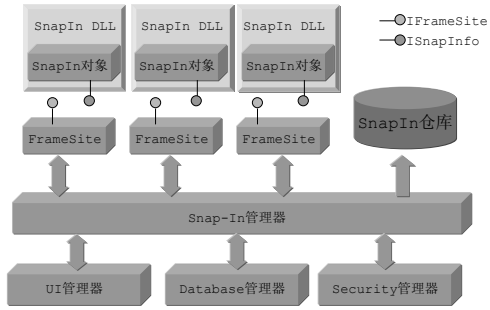
- 接口：
  - 一般为抽象类，用户提供虚函数的实现，并注册到主框架中
  - 用户定制的余地比较大
- 通信模型
  - 用户组件与框架进行通信
  - 用户组件之间容易建立起直通途径

从派生类传播类型到基类的一种模式

- 意图：
  - 基类有时需要根据子类的类型执行一些功能，而基类又不可能直接得到子类的静态类型
  - 这对于generic programming非常重要，因为编译器要靠静态类型来实例化模板(函数或者类)
- 解决方案
  - 用虚函数不能解决问题 —— runtime多态性
  - 在子类中插入一个函数，由该函数调用模板函数或者模板类，或者该函数调用基类中的模板成员函数
  - 仅对基于源代码的framework适用

Framework举例

- 为报社提供一套framework



可重用类库的设计(一)

- 在所有的系统设计中，可重用类库的设计是难度比较大的，要做到：
  - 使用：灵活性和易用性
  - 功能：广泛性和效率
- 经验非常重要
  - 实现同样的功能会有许多不同的道路，如何选择？效果怎么样？
- 类库的基础
  - 是否使用其他的类库？是否使用特殊的平台和编译环境？
- 参考成功的类库
  - 起点要高

可重用类库的设计(二)

- 接口的设计
  - 这是类库的关键，会影响到类库的使用
    - 接口的类型：C/C++
  - 大而全的接口并不理想
  - 接口的语义一定要清晰
  - facade模式
- 内存管理
  - 保证内存分配和释放的一致性
  - 使用要方便
    - [out]参数的资源由谁来申请？谁知道size？
    - 是否使用自定义的内存分配器，例如针对小对象的分配器

可重用类库的设计(三)

- 使用各种模式
  - 模式是经验，成功的典范
  - policy模式允许使用者定制类
  - 结构型模式有助于建立起更加合理的结构模型，而不至于层次错综复杂
  - 行为型模式有助于各个类之间有更好的协作模型
  - 创建型模式可以提供各种合理的创建机制
- 模板类库的特殊性
  - 利用模板类型实现compile-time的预处理
  - 熟悉编译器的特性
  - 控制模板生成代码

可重用类库的设计(四)

- 行为前置和延后
  - 在基类中提供缺省的实现
  - 纯虚函数 —— 强制子类提供实现
  - 利用functor或者函数指针
  - 要求(必须)子类调用父类的实现
- 用宏来封装代码
- 代码风格
- 类库的优化
  - 优化需要用到内部知识，是否暴露这些知识
  - 允许使用者用policy进行配置，用不同的实现配置类
  - 类似于policy的思想，在细节点上用开关进行控制

可重用类库的设计(五)

- 类库的调试
  - 类库内部调试，使用assert支持
- 类库的测试
  - 比应用系统的测试更加严格
- 类库的发行
  - 是否提供源代码？
  - 文档。
  - 编译设置

可重用类库的设计(六)

- 举例：MFC/ATL
- MFC
  - 同时也具有源代码框架的结构
  - 传统意义上的C++类库，对Win32进行了封装
  - 以便于使用为主要目标，优化较少
  - 用到了许多patterns，吻合Windows应用模式
  - 涉及到许多类库设计技术
  - 与Wizard结合产生基本代码
- ATL
  - 用到了generic programming中许多新的技术
  - 模板技术
  - 优化比较突出

总结

- C++基础
- COM基础
- COM扩展
- COM+
- Design Patterns
- 软件设计技术