НЕКОТОРЫЕ ПРИЕМЫ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОЕКТИРОВАНИЯ ПРИЛОЖЕНИЙ РАСПРЕДЕЛЕННОГО МОДЕЛИРОВАНИЯ НА C++

В. А. Репин (Санкт-Петербург)

Введение

В последние годы для решения задач моделирования все чаще применяются распределенные вычислительные системы. Популярность технологии распределенных вычислений, в обиходе называемой GRID, связана как с экономическими аспектами – вычислительный кластер можно собрать практически в любой лаборатории, отталкиваясь от потребностей в вычислительной мощности и доступного бюджета [6], так и с принципиальной возможностью масштабировать алгоритм, отлаженный в лабораторных условиях, на более производительный вычислительный кластер, вплоть до метакомпьютера.

Несмотря на всю привлекательность GRID-технологий для заказчиков, перед разработчиками программного обеспечения (ПО) распределенной вычислительной системы встает ряд проблем, вызванных в первую очередь отсутствием общепризнанных, «стандартных» подходов к разработке ПО для GRID-систем. В то же время, существует несколько общеизвестных методологий проектирования сложных программных систем. Одной из наиболее часто применяемых при проектировании является объектно-ориентированная технология (ООП) [6]. Применение объектно-ориентированного подхода при разработке ПО распределенного моделирования позволяет не только задействовать богатый арсенал шаблонов проектирования, накопленный за долгие годы развития ООП, но и, отладив выполнение алгоритма моделирования на одном компьютере (используя многопотоковую или многопроцессную модель выполнения), перенести его («масштабировать» алгоритм) на вычислительный кластер.

Одним из наиболее распространенных языков объектно-ориентированного программирования является C++ [2]. Именно его популярность обуславливает повышенное внимание к различным средствам разработки приложений для распределенного моделирования, ориентированным на C++. Далее предполагается, что читатель знаком с этим языком. В статье предложен подход к объектно-ориентированному проектированию приложений распределенного моделирования, проиллюстрированный рядом примеров.

Абстракции распределенного ПО на основе стратегий

Чрезвычайно плодотворным методом обобщенного программирования является понятие класса-стратегии. Подробное описание этой идеи можно найти в [3]. Здесь приводится лишь краткое описание этого понятия, необходимое для понимания дальнейших разделов статьи.

Механизм стратегии основан на комбинации шаблонов и множественного наследования. Класс, использующий стратегии, представляет собой шаблон с шаблонными шаблонными параметрами, каждый из которых является стратегией. Главный класс «переадресовывает» части функциональных возможностей стратегиям и действует в качестве хранилища нескольких согласованных между собой стратегий [3].

Класс, использующий стратегии (главный класс), может быть не только шаблоном, но и обычным классом, наследующим от стратегий.

Для пояснения вышеизложенных положений рассмотрим пример шаблона стратегии и главного класса. Определим стратегию реализации распределенных вычислений как шаблон с одним шаблонным параметром `T`:

```
template <typename T> class ThreadingModel { ... };
```

Члены этого шаблона будут определять интерфейс доступа к сущностям, инкапсулирующим различные аспекты взаимодействия с объектами и функциями системы, обеспечивающими поддержку параллелизма. Различные реализации стратегии будут отличаться названиями этого шаблона, сохраняя его сигнатуру. Например, реализация стратегии распределенных вычислений для кластера, состоящего из одного компьютера (вырожденный случай), может быть объявлена так:

```
template <typename T> class SingleThreadingModel { ... };
```

А реализация данной стратегии для лабораторного вычислительного кластера так:

```
template <typename T> class LabThreadingModel { ... };
```

Аналогичным образом можно объявить произвольное количество стратегий.

Главный класс может быть как шаблоном с шаблонными параметрами:

```
template
<
  class T,
  template <class> class ThreadingModel
>
class Host : public ThreadingModel<T> { ... };
```

так и обычным классом:

```
class Host : public LabThreadingModel<Host> { ... };
```

или

```
class Host : public LabThreadingModel<EmptyType> { ... };
```

Здесь `EmptyType` – это «пустой» класс, объявленный следующим образом[1]:

```
struct EmptyType {};
```

Отметим, что в последних случаях приходится явно указывать используемую стратегию поддержки распределенных вычислений, что ведет к увеличению количества изменений, которые необходимо внести в код приложения при ее замене.

Главный класс `Host` наследует интерфейс класса-стратегии; этот интерфейс как бы «подмешивается» в интерфейс главного класса. Причем «сборка» главного класса (который, очевидно, может наследовать от нескольких стратегий, описывающих различные аспекты его поведения) происходит на этапе компиляции.

Одним из приемов, упрощающих построение кросс-платформенных приложений, является объявление шаблона DefaultThreadingModel, являющегося синонимом одного из шаблонов классов-стратегий, и наследование всех классов проекта, нуждающихся в поддержке параллелизма, от него. В этом случае при переносе кода на другую платформу достаточно будет изменить декларацию данного шаблона, никаких изменений в объявлениях главных классов не потребуется. Однако следует отметить, что для объявления такого шаблона невозможно использовать ключевое слово typedef, поскольку шаблон превращается в тип только после инстанцирования. Поэтому для задания стратегии распределенных вычислений «по умолчанию» приходится использовать средства препроцессора. Или вводить еще один уровень иерархии, определяя шаблон DefaultTheadingModel как потомка какого-либо шаблона класса стратегии.

[1] Если Вы используете библиотеку *Loki*, то можете использовать тождественное определение класса `EmptyType` из пространства имен *Loki*. Для этого достаточно подключить заголовочный файл `EmptyType.h`.

Абстракция блокировки

Практически во всех нетривиальных параллельных программах необходимо обеспечивать доступ в режиме чтения/записи к данным, разделяемым несколькими потоками выполнения. Буч отмечает, что как только в систему введен параллелизм, сразу возникает вопрос о том, как синхронизировать отношения активных объектов друг с другом, а также с остальными объектами, действующими последовательно. Например, если два объекта посылают сообщение третьему, должен быть какой-то механизм, гарантирующий, что объект, на который направлено действие, не разрушится при одновременной попытке двух активных объектов изменить его состояние. В параллельных системах недостаточно определить поведение объекта, надо еще принять меры, гарантирующие, что он не будет растерзан на части несколькими независимыми процессами [1].

Одним из механизмов, без которых практически невозможно обойтись в распределенных приложениях, является блокировка доступа к общим ресурсам (данным, устройствам и т.п.). Для этих целей используются такие конструкции как семафор, мьютекс и критическая секция [4].

Поставим задачу выделения абстракции блокировки. Какие операции должен предоставлять объект, отвечающий за блокировку? Основными его функциями будут захват и освобождение ресурса. Также нельзя забыть про запросы на создание и уничтожение самой блокировки. Моделью ресурса в объектно-ориентированном проектировании является объект. Таким образом, мы приходим к стратегии блокировки, обеспечивающей конструирование и уничтожение объекта блокировки, а также операции захвата и освобождения объекта главного класса, наследующего от этой стратегии.

Библиотека *Loki* предлагает следующий интерфейс стратегии блокировки объекта:

```
template < class Host >
class ObjectLevelLockable {
public:
        ObjectLevelLOckable();
        ~ObjectLevelLockable();
        class Lock; friend class Lock;
        class Lock {
        public:
                explicit Lock(ObjectLevelLockable& host);
                ~Lock();
        private:
                 Lock(const Lock&);
                 Lock& operator=(const Lock&);
                 ObjectLevelLockable& host;
        };
};
```

В процессе создания объекта класса, использующего стратегию блокировки (см. п. 2), будет вызван конструктор класса-стратегии, в котором у низкоуровневого механизма синхронизации (некоего сетевого механизма в случае применения GRID-технологии, операционной системы в случае выполнения приложения на одном компьютере в нескольких потоках выполнения и т.д.) запрашивается объект блокировки. Аналогичным образом в деструкторе класса-стратегии происходит возвращение объекта блокировки. Дескриптор объекта блокировки хранится в закрытой части объекта главного класса и используется объектами класса *Lock*, поэтому этот класс объявлен другом класса-стратегии.

Обратите внимание на класс *Lock*, определяемый стратегией блокировки. В его конструкторе происходит захват ресурса (выполняется операция *p*), а в деструкторе его освобождение (операция *v*). Таким образом, для того, чтобы обеспечить выполнение фрагмента кода <code> между операциями *p* и *v*, в методе главного класса достаточно написать:

```
{
  Lock(*this);
  <code>
}
```

Преимуществом данного подхода в сравнении с традиционным определением функций-членов для выполнения операций захвата и освобождения ресурса является перенос бремени контроля соответствия вызовов *p* и *v* с плеч программиста на компилятор. Этот шаблон проектирования известен под названием *scoped locking* и описан в работе [5]. Он тесно связан с общим принципом «выделение ресурса есть инициализация», который часто используют при программировании на C++. Копирующий конструктор класса *Lock* и его оператор присваивания объявлены в закрытой секции (и не определены) для того, чтобы запретить копирование объектов этого класса.

Стратегия блокировки на уровне объекта, при которой объект главного класса владеет ресурсом блокировки, не всегда достаточно эффективна. Если объектов главного класса в программе очень много и они нечасто прибегают к услугам класса *Lock*, то возможно более эффективной будет стратегия блокировки на уровне класса, ClassLevelLockable [3]. Интерфейс класса, реализующего такую стратегию, совпадает с вышеописанным. Но один ресурс блокировки уже соответствует одному главному классу, а не объекту главного класса, что достигается использованием статических членов шаблонного класса, реализующего стратегию ClassLevelLockable.

Итак, в ходе анализа было выявлено две стратегии блокировки – на уровне класса и на уровне объекта. Число шаблонных классов, их реализующих, зависит от количества механизмов блокировки, которыми пользуется разработчик приложения.

При реализации стратегии блокировки на уровне объекта возникает один нюанс, приводящий к необходимости внесения дополнений в интерфейс стратегии блокировки в некоторых приложениях. Принадлежность дескриптора блокировки объекту главного класса заставляет обращать особое внимание на операции, приводящие к копированию объектов главного класса (т.е. к вызову копирующего конструктора или оператора присваивания главного класса). Предположим, что разработчик создал два объекта класса H_t, использующего стратегию ObjectLevelLocking, реализованную при помощи семафоров *SysV* [4], – H1 и H2 (рассматриваем реализацию параллельных вычислений на основе нескольких потоков выполнения в рамках одной ЭВМ). При этом H2 создан из H1 при помощи копирующего конструктора, сгенерированного компилятором. Тогда в программе будут одновременно существовать две копии дескриптора набора семафоров *SysV*. Удаление же одного из объектов H1, H2 приведет к появлению некорректного дескриптора в другом объекте. Способ решения этой проблемы зависит от того, как разработчик приложения планирует использовать объекты класса H_t. Возможны следующие варианты:

– объекты класса H_t *не* предполагается создавать при помощи копирующих конструкторов и присваивать их друг другу; в этом случае достаточно поместить копирующий конструктор и оператор присваивания в закрытую часть описания класса H_t и не определять их[1];

[1] Если Вы используете библиотеку *Boost* [7], то достаточно включить в иерархию наследования класса H_t класс noncopyable, определенный в пространстве имен Boost.

– создание объектов класса H_t при помощи копирующего конструктора приводит к порождению объекта блокировки ОС, связанного с новообразовавшимся объектом класса H_t; присваивание объектов этого класса друг другу не приводит к копированию дескриптора блокировки ОС;

– все объекты класса H_t, созданные при помощи копирующего конструктора или измененные при помощи оператора присваивания, содержат ссылку на один и тот же «первичный» дескриптор блокировки ОС; в этом случае для корректного освобождения блокировки ОС (возврата дескриптора) придется вести подсчет ссылок на нее и возвращать объект блокировки операционной системе при достижении этим счетчиком нуля.

Для реализации последних двух вариантов необходимо добавить в стратегию блокировки специальные методы, выполняющие функции копирующего конструктора и оператора присваивания. Необходимы именно отдельные методы, а не определение копирующего конструктора и оператора присваивания в стратегии, поскольку они не наследуются – так, если не определен оператор копирующего присваивания, он будет сгенерирован компилятором [2].

Литература

1. **Буч Г**. Объектно-ориентированный анализ и проектирование с примерами приложений на C++, 2-е изд./Пер с англ. – М.: Бином, СПб.: Невский диалект, 1998 – 560 с.: ил.
2. **Страуструп Б.** «Язык программирования C++, 3-е изд./Пер. с англ. – М.: Бином, СПб: Невский диалект, 1999 – 991 с., ил.
3. **Александреску А.** Современное проектирование на C++. Пер. с англ. – М.: Издательский дом «Вильямс», 2002. – 336 с.: ил.
4. **Хэвиленд К., Грэй Д., Салама Б.** Системное программирование в UNIX. Руководство программиста по разработке ПО./Пер. с англ. – М.: ДМК Пресс, 2000. – 368 с.: ил.
5. **Douglas Schmidt, Michael Stal, Hans Rohnert** и др. Pattern-Oriented Software Architecture, Volume 2, Patterns for Concurrent and Networked Objects. – *John Wiley & Sons, 2000*. – 666 с.: ил.
6. **Воеводин В.В., Воеводин Вл. В.** Параллельные вычисления/СПб.: БХВ-Петербург, 2002. – 608 с.: ил.
7. http://www.boost.org