Реализация параллелизма с использованием «эффективных объектов»

Решение задач организации параллелизма приложения происходит традиционно, применяя вытесняющую многозадачность. Такая схема целесообразна, когда параллельность касается отдельных процессов операционной системы (однако, однозначно переносить понятие вытесняющей многозадачности на уровень отдельного взятого процесса-приложения нельзя).

Ввиду специфики традиционной схемы возникает ряд недостатков:

1. Необходимость синхронизации и взаимоисключающего доступа. При увеличении нитей и их интенсивном взаимодействии друг с другом очевидна проблема синхронизации.
2. Высокие требования к ресурсам. Что влечет существенное ограничение числа создаваемых нитей, поэтому возникает низкая масштабируемость проекта и невозможность декомпозиции при десятках и сотнях тысяч параллельных процессов.
3. Плохая переносимость на различные платформы и языки программирования.

С учетом недостатков традиционной схемы предполагается новый способ организации параллелизма, достоинства которого:

1. Нет привязки к языку и платформе;
2. Хорошо масштабируется (от одного процесса до нескольких миллионов);
3. Требует минимальных ресурсов времени;
4. Обеспечивает простые механизмы взаимодействия параллельных процессов.

Рассмотрим новый способ организации параллелизма в рамках одного приложения, который основан на кооперативной многозадачности, но без использования сопрограмм. Отказ от сопрограмм связан с тем, что их поддержка возможна только на уровне языка. Возможно использовать для реализации сопрограмм внеязыковые механизмы операционной системы (2),

но это вносит существенную зависимость от платформы. Сопрограммы на основе волокон (fibers) требуют меньше ресурсов, чем нити, но стек сопрограммы остается достаточно большим.

Для достижения указанных выше достоинств, требуется принять условия кооперативной многозадачности и сопрограмм:

1. Необходимость указания в программном коде параллельного процесса точек прерывания выполнения и передачи управления другим параллельным процессам;
2. Необходимость самостоятельно обеспечивать справедливость распределения процессорного времени между всеми параллельными процессами.

Под параллелизмом будем понимать только программно реализованный параллелизм, а фактически его имитация на процессоре. Традиционно параллелизм основан на декомпозиции по функционалу, т.е. единицей параллельности является процедура. Рассматриваемый пример основан на объектной декомпозиции, - «эффективном объекте»- конечный автомат, имеющий набор состояний. Суть «эффективного объекта» в том, что он, с одной стороны, является полноценным объектом в терминах объектно-ориентированного подхода, а с другой стороны он является активным независимо от активности других объектов. Если в последовательной программе выделяется только один поток управления, то в параллельной программе их множество, причем каждый «эффективный объект» самостоятелен.

Кооперативная многозадачность предполагает, диспетчеризацию программистом, который ответственен:

1. за справедливое распределение процессорного времени, т.е. за организацию короткого шага «эффективного объекта»;
2. после каждого шага объект должен находиться в непротиворечивом состоянии и полностью завершать изменение своего состояния.

С учетом решения поставленных выше задач программисту исключены следующие проблемы:

1. проблема взаимоисключающего доступа, т.е. объекты могут взаимодействовать между собой так же просто, как и в последовательной программе.
2. проблема неопределенного размера стека эффективного объекта, поскольку каждый шаг активного объекта сводится к вызову процедуры и возврату из нее, то все активные объекты используют один и тот же стек.

   Пример, скелет активного объекта:

```
public class ActiveObject
{protected virtual void Step() { }
public bool Active { get{} set{} }
public bool Sleeping { get{} }
public void Sleep(int time) { }
public void Sleep() { }
public void WakeUp() { }
public static void Open(ThreadPriority priority) { }
public static void Close() { }}
```

   В приведенном примере выделяются три группы методов:

   1. описание жизненного цикла «эффективного объекта»;
   2. изменение состояния «эффективного объекта»;
   3. управление жизнью всех «эффективных объектов» в границе отдельной нити.

Объект может быть активным или неактивным, в активном состоянии он определяется методом Step. Установка Active изменяет активность объекта. Методы Sleep приостанавливают объект на указанный интервал времени или до вызова Wake. Свойство Sleeping возвращает значение true, если объект находится в состоянии ожидания. Статический метод Open

инициирует циклический обход всех «эффективных объектов» и вызов у каждого объекта метода Step.

Так как активный объект реализуется как конечный автомат, то в C# можно использовать удобный метод реализации итераторов (5). Оператор yield return выполняет возврат очередного значения итератора, подобно тому, как это происходит в сопрограммах, но реализуется он через конечный автомат. Ознакомившись с кодом, генерируемым компилятором, можно увидеть, что создается скрытый класс, целочисленное поле state и метод MoveNext, содержащий оператор switch. Реализация yield в точности соответствует простому конечному автомату. Поскольку оператор yield return можно применить только в итераторах, то сигнатура метода Step будет выглядеть следующим образом:

```
public class SampleActiveObject : ActiveObject
{protected override IEnumerator Step()
{while (true)
{// действие 0
yield return null;
// действие N
yield return null;
}}}
```

Так как yield return не используется для возврата текущего состояния итератора, не важно возвращаемое значение. Оператор yield return завершает шаг и приводит к выходу из Step. Получилось, что предоставили компилятору реализацию конечного автомата.

Пример части класса, который управляет активностью объекта:

```
private static Dispatcher dispatcher;
private ActiveObject prev;
private ActiveObject next;
private int timeout;
private bool sleeping;
```

```
public bool Active
{get
{return next != null;}
set
{if (value)
dispatcher.Add(this);
else
dispatcher.Remove(this);
}}
public bool Sleeping
{get
{return sleeping;
}}
public void Sleep(int interval)
{dispatcher.Sleep(this, Environment.TickCount + interval);
}
public void Sleep()
{dispatcher.Sleep(this);
}
public void WakeUp()
{dispatcher.WakeUp(this);
}
```

Класс активного объекта имеет скрытый статический объект – диспетчер, который содержит все объекты, находящиеся в активном состоянии. Каждый объект имеет два поля – ссылку на предыдущий активный объект (prev) и ссылку на следующий (next). Если объект не содержится в списке диспетчера, он находится в неактивном состоянии. Активация объекта заключается в его добавлении в конец списка диспетчера, а деактивация – в удалении из списка диспетчера.

Пример методов Open и Close:

```
private static Activator activator;
private static Timer timer;
private static Log log;
public static void Open(ThreadPriority priority)
{activator.Start(priority);}
public static void Close()
{activator.Stop();
dispatcher.Clear();
timer.Clear();
log.Clear();}
```

Метод Open создает и стартует нить, вызывая метод Start у скрытого статического объекта activator, а метод Close останавливает и уничтожает нить, а также очищает содержимое остальных скрытых статических объектов. Рассмотрим скрытые статические объекты: диспетчер, активатора и таймера. Все эти объекты создаются в статическом конструкторе класса активного объекта.

```
static ActiveObject()
{dispatcher = new Dispatcher();
timer = new Timer();
activator = new Activator();
log = new Log();
}
```

Реализация диспетчера основана на двусвязном списке с явно выделенными граничными узлами first и last. Связи активного объекта с предыдущим и последующим хранятся в полях prev и next. Некоторая сложность списка обусловлена тем, что активный объект может во время своего выполнения создать новый активный объект, активировать или деактивировать другой объект, или деактивировать самого себя. То есть, состояние списка может измениться при его обходе.

Использование граничных узлов более эффективно при добавлении и удалении элементов, так как эти операции всегда происходят в предположении, что элемент имеет как предыдущего, так и последующего соседа. В целом реализация списка достаточно проста.

Назначение активатора – циклическая итерация по всем активным объектам и вызов их метода Step. Активатор создает собственную нить, в контексте которой и выполняются все активные объекты.

При Активации объекта выполняется три проверки:

1. контроль неактивности объекта (неактивность определяется тем, что поле next объекта равно null, то есть у него еще нет соседа);
2. контроль отсутствия фатальной ошибки (если объект содержит непустую ссылку на исключительную ситуацию, то он был завершен с фатальной ошибкой и не может быть активирован вновь);
3. контроль отсутствия состояния ожидания (если поле sleeping объекта имеет значение true, то объект стал неактивным в результате вызова одного из методов Sleep и может быть активирован только с помощью Wake). При деактивации выполняется проверка - является ли объект действительно активным. Таким образом, можно безопасно активировать уже активные объекты и деактивировать неактивные объекты.

Деактивация объекта на некоторый интервал времени возможна в том случае, если объект не стоит в очереди таймера, не находится в состоянии ожидания и не завершен по фатальной ошибке. Если все эти условия выполняются, то объект удаляется из списка диспетчера и добавляется в список таймера.

Изменение списка выполняется в критических секциях (lock), что позволяет активировать и деактивировать объекты из других нитей приложения.

Таймер представляет собой список временно неактивных объектов. Объекты попадают к таймеру при выполнении метода Sleep. Список сортируется по убыванию времени, и, таким образом, объекты с меньшими значениями времени ожидания (timeout) размещаются в конце списка.

```
private class Timer : : IComparer<ActiveObject>
{private List<ActiveObject> list;
public Timer()
{list = new List<ActiveObject>();
}
public int Compare(ActiveObject obj1, ActiveObject obj2)
{return obj2.timeout - obj1.timeout;
}
public void Add(ActiveObject obj)
{lock (this)
{int i = list.BinarySearch(obj, this);
if (i < 0)
i = ~i;
list.Insert(i, obj);
}}
public void Clear()
{lock (this)
{list.Clear();
}}
public ActiveObject GetReady()
{lock (this)
{int H = list.Count - 1;
if (H >= 0)
{ActiveObject obj = list[H];
if (Environment.TickCount >= obj.timeout)
{list.RemoveAt(H);
```

```
return obj;
}}}
return null;
}}
```

Метод Add выполняет добавление объекта в список на основе его поля timeout, используя компаратор с обратным порядком сортировки. Метод GetReady извлекает из конца списка объект, если его время ожидания истекло. Если же такого объекта нет, то метод возвращает null.

Метод Start создает нить и активирует ее. Метод Stop устанавливает признак завершения нити (terminate) и ожидает ее завершения. Основная работа нити происходит в методе Execute. Метод выполняет бесконечный цикл, в ходе которого вначале контролируется наличие активных объектов, и, если таковые имеются, то выполняется итерация по всем активным объектам. В ходе итерации для каждого объекта вызывается его метод Step.

Поскольку все активные объекты работают в контексте одной нити, они могут использовать для взаимодействия между собой обычные, не многопоточные контейнеры. Например, для создания очереди сообщений, можно применить объект класса Queue без его блокировки (lock). Также нет необходимости во взаимоисключающем доступе, если активный объект выполняет составное действие «проверить и установить». Для реализации взаимодействий имеет смысл создавать либо контейнеры для буферизации обмена, либо объекты-события для небуферизованного взаимодействия (недетерминированного ожидания или рандеву(7)).