Життєвий цикл об’єктів у середовищі .NET

Ми вже знаємо, що всі типи даних у світі .NET діляться на дві категорії: типи значення і типи посилання. Типи значення займають місце в стеку. Розміщуються вони там автоматично після оголошення-ініціалізації. Звільняються значення зі стека також автоматично, коли виконання програми полишить блок, у якому дані було створено. Все так само, як в мові C++.

Типи посилання займають місце в керованій купі. Їх створюють оператором *new*. (Пригадайте, як оголошують звичайний масив.) Проте ніхто ніколи не використовує оператор *delete*. Його навіть нема в мові. Купа тому і називається керованою, що її очищенням займається середовище виконання. Програміст може не турбуватися про очищення посилань. Замість нього цим займається системний збирач сміття (garbage collector).

Збирач сміття

Як би Ви організували очищення купи від непотрібних об’єктів? Пригадайте генеральне прибирання вдома. Всі справи відкладено, кожен предмет в кімнаті піддається перевірці: чи він ще потрібен. Всі списані та поламані ручки, використані чорновики, упакування від їжі йдуть у сміття, розкидані книги та одяг займають місця на полицях. Після прибирання кімната сяє, вона готова до роботи впродовж наступного тижня (чи року), але розплатою за порядок стала зупинка всіх інших справ. Ви не можете дозволити собі такі зупинки часто через напружений ритм навчання і роботи. Системний збирач сміття також не може. На час прибирання керованої купи всі інші процеси застосунку зупиняються, тому збирач мусить працювати не довго і не надто часто, щоб не гальмувати виконання програми.

То ж як організувати очищення пам’яті, щоб воно відбувалося швидко і ефективно.

Розміщення посилань у купі

Розміщення всіх об’єктів відбувається підряд за вказівником вільного місця купи. Після розміщення вказівник посувається далі на розмір розміщеного об’єкта. Розміщення швидке, оскільки не витрачає час на пошуки місця.

Пошук «жертви»

Керована купа вміє будувати граф досяжності об’єктів, який починається з кореневих об’єктів. Об’єкт підлягає очищенню, коли стає недосяжним. Проте збирач сміття не поспішає знищувати непотрібний об’єкт, поки є місце в купі. Момент запуску збирача вибирає середовище виконання недерміновано: неможливо передбачити, коли воно почнеться.

Механізм пошуку сміття не позбавлений недоліків: об’єкт може «застрягнути» в купі, якщо посилання на нього випадково збережеться у якійсь частині іншого об’єкта. Іноді, неусвідомлено з боку програміста. Це критично, якщо об’єкт захопив дорогий системний ресурс.

Покоління об’єктів

У купі є три покоління об’єктів. Покоління потрібні, щоб скоротити побудову графа досяжності. Усі новостворені об’єкти потрапляють в покоління 0. Коли настає час очищувати купу, збирач сміття переглядає покоління 0 і вилучає з нього всі недосяжні об’єкти. У результаті зайнята пам’ять може зазнати фрагментації. Щоб позбутися дірок у зайнятій частині, збирач переміщує всіх, хто вижив, на нове неперервне місце – в покоління 1. Усі об’єкти покоління 1, які пережили очистку, переходять в покоління 2. Старші покоління піддаються чистці лише за крайнього браку пам’яті чи за вказівкою програміста. Поділ на покоління суттєво пришвидшує очистку, оскільки ті дані, які пережили відбір зазвичай потрібні тривалий час, часто – до кінця роботи програми.

Звільнення системних ресурсів

Програми .NET все ще залежать від старої інфраструктури. Їм доводиться використовувати ресурси операційної системи з-поза меж світу .NET. Наприклад, дескриптор файла, приєднання до бази даних, мережевий сокет не є керованими ресурсами, тому збирач сміття

не може їх звільнити, а звільняти треба обов'язково. Якщо екземпляри класу захоплюють некеровані ресурси, програміст мусить додатково потурбуватися про їхнє звільнення. Для цього можна оголосити деструктор, або (і/або) реалізувати інтерфейс *IDisposable*.

Деструктори

Відомі, як методи-фіналізатори. Програміст пише таке:

```
class MyClass
{
	~MyClass()
	{
		// звільнення захоплених системних ресурсів
	}
}
```

А компілятор створює в збірці таке (в коді класу):

```
protected override void Finalize()
{
	try
	{
		// звільнення захоплених системних ресурсів
	}
	finally
	{
		base.Finalize();
	}
}
```

Таким чином середовище турбується, щоб екземпляр підкласу не забув звільнити успадковані ресурси. Також суттєво спрощується написання правильного деструктора: у програміста голова не болить про поля батьківського класу.

Деструктор чи фіналізатор викликати явно нема ніякої змоги. Їх викликає збирач сміття середовища виконання тоді, коли середовище вирішить, що настав час прибирання купи. Тому покладатися на вчасний виклик деструктора, чи на певну послідовність викликів деструкторів різних об'єктів не доводиться. Якщо об'єкт мав би звільнити критично важливі ресурси, причому, на вимогу програміста, доведеться застосувати інший підхід.

Ще один недолік деструкторів – затримка виконання. Звичайні об'єкти збирач сміття звільняє за один прохід, а об'єкт з деструктором – за два. Спочатку середовище викликає для всіх таких об'єктів метод *Finalize*, а потім об'єкти звільняють остаточно. Для виконання фіналізаторів середовище має лише один потік, що також може сповільнити програму.

System.IDisposable

Інтерфейс є рекомендованою альтернативою, яка має підтримку на рівні мови.

```
class MyClass: IDisposable
{
	public void Dispose ()
	{
		// звільнення вкладених об'єктів
		if (nestedObj is IDisposable) nestedObj.Dispose();
		// звільнення захоплених системних ресурсів
	}
}
```

Використовувати екземпляри таких класів потрібно за певною схемою, щоб гарантувати очищення.

```
MyClass theInstance = null;
try {
	theInstance = new MyClass();
	// виконати роботу
```

```
}
finally {
	if (theInstance != null) {
		theInstance.Dispose ();
	}
}
```

Мова надає підтримку з автоматичного виклику очищення. Зручний аналог наведеного вище фрагменту:

```
using (MyClass theInstance = new MyClass())
{
	// виконати роботу
}
```

Виклик *Dispose* відбудеться автоматично і незалежно від винятків. Пригадайте, в яких програмах ви вже бачили таку інструкцію.

Об’єднання двох підходів

А якщо, все таки, хтось забуде викликати *Dispose* для екземпляра, для якого це дуже треба? Добре було б підстрахуватися, і додатково оголосити фіналізатор. За такого підходу потрібно узгодити роботу обох, а для цього потрібні будуть: метод, що власне очищує, прапорець стану і спеціальне влаштування методів.

```
public class ResourceHolder: IDisposable
{
	private bool isDisposed = false; // прапорець стану

	// метод очистки, що виконує всю роботу
	protected virtual void Dispose(bool disposing)
	{
		if (!isDisposed)		// якщо ще не очистили
		{
			if (disposing)	// і метод викликано з інтерфейсу, а не з деструктора
			{
				// Очищення керованих об'єктів
				if (nestedObj is IDisposable) nestedObj.Dispose();
			}
			// Очищення некерованих об'єктів – потрібне в обох випадках
		}
		isDisposed = true;
	}

	public void Dispose ()		// метод інтерфейсу
	{
		Dispose(true);
		GC.SuppressFinalize(this); // повідомляє збирачу, що деструктор кликати не треба
	}

	~ResourceHolder ()		// деструктор покладається на метод очищення
	{
		Dispose(false);
	}

	public void SomeMethod () // кожен метод влаштовано особливим чином
	{
		// перед виконанням потрібно впевнитися, що об'єкт ще не звільнили
		if (isDisposed)
		{
			throw new ObjectDisposedException("ResourceHolder");
		}
		// реалізація методу...
	}
}
```

Garbage Collector

Тип System.GC надає доступ до збирача сміття. За допомогою методів цього класу програміст може додатково впливати на його роботу. Методи наведено в таблиці нижче.

Метод	Значення
AddMemoryPressure(), RemoveMemoryPressure()	Дозволяють налаштовувати «рівень терміновості» звільнення об'єкта з купи
Collect ()	Примушує збирача сміття виконати свою роботу терміново. Можна вказати покоління об'єктів.
CollectionCount()	Повідомляє, скільки разів виконували збирання сміття для вказаного покоління
GetGeneration ()	Повідомляє покоління об'єкта
GetTotalMemory()	Приблизний розмір купи
MaxGeneration	Скільки поколінь підтримує середовище виконання
SuppressFinalize()	Повідомляє збирачеві, що об'єкт не повинен викликати свій метод *Finalize*
WaitForPendingFinalizers()	Призупиняє потік виконання поки не завершиться фіналізація всіх об'єктів. Зазвичай викликають після *Collect*

Ліниве створення об'єктів

Припустимо, екземпляр деякого класу інкапсулює значну за обсягом колекцію об'єктів і повинен на запит користувача повертати її. Чи розумно буде створювати таку колекцію відразу в конструкторі? Не дуже, адже користувач може і не звернутися по неї. Тоді робота буде виконана даремно. Краще застосувати відкладене створення: об'єкт готовий створити колекцію за першою вимогою, але не робить цього, поки не виникне потреба. Відкладене створення об'єктів реалізує узагальнений тип *Lazy<>*.

```
class Song              // Одна пісня
{
      public string Artist { get; set; }
      public string TrackName { get; set; }
      public double TrackLength { get; set; }
}

class AllTracks        // Пісенна колекція
{
      // Програвач може пам'ятати щонайбільше 10 000 композицій.
      private Song[] allSongs = new Song[10000];

      // Заповнення колекції пісень
      public AllTracks()
      {
            for (int i = 0; i < allSongs.Length(); ++i)
                  Song[i] = getNextSongFromMusicLibrary();
      }
}

// Клас MediaPlayer містить об'єкт Lazy<AllTracks> заради відкладеної ініціалізації
// змінної allSongs.
class MediaPlayer
{
      public void Play() {  /* Відтворення композиції */ }
      public void Pause(){  /* Пауза у відтворенні */    }
      public void Stop() {  /* Зупинка відтворення */    }

      // відкладене створення
      private Lazy<AllTracks> allSongs = new Lazy<AllTracks>();
```

```
        public AllTracks GetAllTracks()
        {
                // Повернути всі композиції означає створити AllTracks
                return allSongs.Value;
        }
}
```

При першому звертанні до властивості *Value* буде викликано конструктор за замовчуванням інкапсульованого класу. Якщо треба викликати щось інше, можна скористатися таким способом:

```
private Lazy<AllTracks> allSongs = new Lazy<AllTracks>(
        () => { Console.WriteLine("Creating AllTracks object!");
                    return new AllTracks(); } );
```

Тут спосіб створення об’єкта задано лямбда-виразом.

Приклади використання описаних інструментів можна знайти в
[1] Эндрю Троелсен Язык программирования C#5.0 и платформа .Net 4.5
[2] Ватсон Б. C# 4.0 на примерах - 2011