**Сборка мусора**

Жизненный цикл любого объекта можно представить следующим образом:

1) выделение памяти для типа;

2) инициализация выделенной памяти (установка объекта в начальное значение, вызов конструктора);

3) использование объекта в программе;

4) разрушение состояния объекта;

5) освобождение занятой памяти.

Сбор мусора освобождает программиста от необходимости постоянно следить за использованием и своевременным освобождением памяти. Специальный код, осуществляющий сбор мусора называется «сборщик мусора» (**Garbage Collector**).

Сборка мусора состоит из следующих шагов:

1. Сборщик мусора осуществляет поиск управляемых объектов, на которые есть ссылки в управляемом коде.
2. Сборщик мусора пытается завершить объекты, на которые нет ссылок.
3. Сборщик мусора освобождает объекты, на которые нет ссылок, и высвобождает выделенную им память.

**Деструктор и метод Finalize**

**Пример 1**

using System;

namespace FinalizeExample

{

class Program

{

static void Main(string[] args)

{

MyClass mc = new MyClass();

}

}

class MyClass

{

public MyClass()

{

Console.WriteLine("Создание объекта");

}

~MyClass()

{

// Здесь уничтожаем ресурсы, которые занял данный объект.

// Данный метод вызывается непосредственно

// перед уничтожением объекта

Console.WriteLine("Уничтожение объекта");

}

}

}

При очистке сборщик мусора вызывает не деструктор, а метод **Finalize** класса. Все потому, что компилятор C# компилирует деструктор в конструкцию, которая эквивалентна следующей:

protected override void Finalize()

{

try

{

// здесь идут инструкции деструктора

}

finally

{

base.Finalize();

}

}

**Пример 2**

using System;

namespace ConsoleApplication1

{

class FinalizeObject

{

public int id { get; set; }

public FinalizeObject(int id)

{

this.id = id;

}

// Создадим специальный деструктор

~FinalizeObject()

{

Console.ForegroundColor = ConsoleColor.Yellow;

Console.WriteLine("Объект №{0} уничтожен", id);

Console.Beep();

}

}

class Program

{

static void Main(string[] args)

{

Console.Read();

// После того как будет нажата клавиша Enter (выход из программы)

// все последующие объекты будут уничтожены

FinalizeObject[] obj = new FinalizeObject[100];

for (int i = 0; i < 100; i++)

obj[i] = new FinalizeObject(i);

}

}

}

**Метод Dispose и интерфейс IDisposable**

public interface IDisposable

{

void Dispose();

}

**Пример 3**

using System;

namespace ConsoleApplication1

{

// Данный класс реализует интерейс IDisposable

class FinalizeObject : IDisposable

{

public int id { get; set; }

public FinalizeObject(int id)

{

this.id = id;

}

// Реализуем метод Dispose()

public void Dispose()

{

Console.WriteLine("Высвобождение объекта!");

}

}

class Program

{

static void Main(string[] args)

{

FinalizeObject obj = new FinalizeObject(4);

obj.Dispose();

Console.Read();

}

}

}

**Пример 4**

using System;

using System.Collections.Generic;

using System.Text;

using System.IO;

namespace IDisposableExample

{

class Program

{

static void Main(string[] args)

{

try

{

// Инициализация потока для передачи в

// DisposableResource

Console.Write("Введите путь к файлу: ");

string fileSpec = Console.ReadLine();

FileStream fs = File.OpenRead(fileSpec);

DisposableResource testObj = new DisposableResource(fs);

// Использование ресурса.

testObj.DoSomethingWithResource();

// Освобождение ресурса

testObj.Dispose();

}

catch (FileNotFoundException e)

{

Console.WriteLine(e.Message);

}

}

}

/// <summary>

/// Демонстрационный класс, наследующий

/// и реализующий интерфейс IDisposable.

/// </summary>

public class DisposableResource : IDisposable

{

#region Fields

private Stream \_resource;

private bool \_disposed;

#endregion

/// <summary>

/// Поток, передаваемый в конструктор

/// должен быть читаем и не null

/// </summary>

/// <param name="stream"></param>

public DisposableResource(Stream stream)

{

if (stream == null)

throw new ArgumentNullException("Поток null.");

if (!stream.CanRead)

throw new ArgumentException("Поток должен быть"+"доступен для чтения.");

this.\_resource = stream;

this.\_disposed = false;

}

/// <summary>

/// Метод демонстрирует использование ресурса.

/// Ресурс не должен быть освобожден к моменту вызова метода

/// </summary>

public void DoSomethingWithResource()

{

if (\_disposed)

throw new ObjectDisposedException("Ресурс был освобожден.");

//Показать количество байт

int numBytes = (int)\_resource.Length;

Console.WriteLine("Количество байт: {0}", numBytes.ToString());

}

#region IDisposable Members

public void Dispose()

{

this.Dispose(true);

// Необходимо использовать SupressFinalize,

// чтобы не выполнять финализацию после

// явного освобождения ресурсов

GC.SuppressFinalize(this);

}

#endregion

/// <summary>

/// Финализатор

/// </summary>

~DisposableResource()

{

this.Dispose(false);

}

protected virtual void Dispose(bool disposing)

{

if (!\_disposed)

{

if (disposing)

{

if (\_resource != null)

\_resource.Dispose();

Console.WriteLine("Ресурс освобожден.");

}

// Индикация, что ресурс уже был освобожден

this.\_resource = null;

this.\_disposed = true;

}

}

}

}

Синтаксис C# также предлагает синонимичную конструкцию для автоматического вызова метод Dispose - конструкцию **using**:

using (Person p = new Person())

{

}

**Класс System.GC**

|  |  |
| --- | --- |
| **Метод** | **Описание** |
| **AddMemoryPressure(), RemoveMemoryPressure()** | Позволяют указывать числовое значение, отражающее "уровень срочности", который вызывающий объект применяет в отношении к сборке мусора. Следует иметь в виду, что эти методы должны изменять уровень давления в тандеме и, следовательно, никогда не устранять больше давления, чем было добавлено |
| **CancelFullGCNotification()** | Отменяет уведомление о "сборке мусора" |
| **Collect()** | Заставляет сборщик мусора провести сборку мусора. Должен быть перегружен так, чтобы указывать, объекты какого поколения подлежат сборке, а также какой режим сборки использовать (с помощью перечисления GCCollectionMode) |
| **CollectionCount()** | Возвращает числовое значение, показывающее, сколько раз объектам данного поколения удалось переживать процесс сборки мусора |
| **GetGeneration()** | Возвращает информацию о том, к какому поколению в настоящий момент относится объект |
| **GetTotalMemory()** | Возвращает информацию о том, какой объем памяти (в байтах) в настоящий момент занят в управляемой куче. Булевский параметр указывает, должен ли вызов сначала дождаться выполнения сборки мусора, прежде чем возвращать результат |
| **KeepAlive()** | Создает ссылку на объект, защищая его от "сборки мусора". Действие этой ссылки оканчивается после выполнения метода KeepAlive() |
| **MaxGeneration** | Возвращает информацию о том, сколько максимум поколений поддерживается в целевой системе. В .NET 4.0 поддерживается всего три поколения: 0, 1 и 2 |
| **RegisterForFullGCNotification()** | Разрешает уведомление о "сборке мусора". |
| **RemoveMemoryPressure()** | Задает в качестве параметра количество байтов, освобождаемых в неуправляемой области памяти |
| **SuppressFinalize()** | Позволяет устанавливать флаг, указывающий, что для данного объекта не должен вызываться его метод Finalize() |
| **WaitForPendingFinalizers()** | Позволяет приостанавливать выполнение текущего потока до тех пор, пока не будут финализированы все объекты, предусматривающие финализацию. Обычно вызывается сразу же после вызова метода GC.Collect() |

**Пример 3**

namespace CSharp\_GC

{

class UserInfo

{

public string Name { set; get; }

public int Age { set; get; }

public UserInfo(string Name, int Age)

{

this.Name = Name;

this.Age = Age;

}

}

class Program

{

static void Main()

{

Console.ForegroundColor = ConsoleColor.Green;

Console.WriteLine("БАЗОВАЯ ИНФОРМАЦИЯ О СИСТЕМЕ: \n" +

"-----------------------------\n");

Console.ForegroundColor = ConsoleColor.White;

Console.WriteLine(@"Операционная система: {0}

Версия .NET Framework: {1}", Environment.OSVersion, Environment.Version);

Console.ForegroundColor = ConsoleColor.Green;

Console.WriteLine("\nИНФОРМАЦИЯ О СБОРКЕ МУСОРА: \n" +

"---------------------------\n");

Console.ForegroundColor = ConsoleColor.White;

Console.WriteLine(@"Количество байт в куче: {0}

Максимальное количество поддерживаемых поколений объектов: {1}"

, GC.GetTotalMemory(false), GC.MaxGeneration + 1);

UserInfo user1 = new UserInfo("Alex", 26);

Console.WriteLine("Поколение объекта user1: " + GC.GetGeneration(user1));

for (int i = 0; i < 50000; i++)

{

UserInfo user = new UserInfo("Dm", 27);

}

// Намеренно вызовем сборку мусора

GC.Collect(0, GCCollectionMode.Forced);

GC.WaitForPendingFinalizers();

Console.WriteLine("\nсборка мусора ...\n");

Console.WriteLine("Теперь поколение объекта user1: " + GC.GetGeneration(user1));

Console.ReadLine();

}

}

}

**Понятие поколений при сборке мусора**

Сборщик мусора после очистки памяти, чтобы не допустить фрагментации, «сжимает» ее, другими словами, он «сдвигает» все объекты так, чтобы они располагались последовательно

****

Как показали исследования, большая часть объектов, которые созданы недавно, быстро становятся недостижимыми. Поэтому управляемая куча была разделена на 3 поколения: 0, 1 и 2. Размер памяти под каждое из поколений разный – он увеличивается с ростом номера поколения. Все вновь созданные объекты попадают в первое (0) поколение. Сбор мусора автоматически вызывается по окончанию памяти в одном из поколений. Объекты, которые пережили сбор мусора, переходят в следующее поколение.



**Пример 4**

using System;

namespace GCExample

{

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Демонстрация System.GC");

Console.WriteLine("Максимальное поколение: {0}", GC.MaxGeneration);

GarbageHelper hlp = new GarbageHelper();

// Узнаем поколение, в котором находится объект

Console.WriteLine("Поколение объекта: {0}", GC.GetGeneration(hlp));

// Количество занятой памяти

Console.WriteLine("Занято памяти (байт): {0}", GC.GetTotalMemory(false));

// Создаем мусор

hlp.MakeGarbage();

Console.WriteLine("Занято памяти (байт): {0}", GC.GetTotalMemory(false));

// Вызываем явный сбор мусора в первом поколении

GC.Collect(0);

// Количество занятой памяти

Console.WriteLine("Занято памяти (байт): {0}", GC.GetTotalMemory(false));

// Узнаем поколение, в котором находится объект

Console.WriteLine("Поколение объекта: {0}",

GC.GetGeneration(hlp));

// Вызываем явный сбор мусора во всех поколениях

GC.Collect();

// Количество занятой памяти

Console.WriteLine("Занято памяти (байт): {0}", GC.GetTotalMemory(false));

// Узнаем поколение, в котором находится объект

Console.WriteLine("Поколение объекта: {0}", GC.GetGeneration(hlp));

Console.Read();

}

}

/// Вспомогательный класс для создания мусора

class GarbageHelper

{

/// <summary>

/// Метод, создающий мусор

/// </summary>

public void MakeGarbage()

{

for (int i = 0; i < 1000; i++)

{

var p = new Person();

}

}

class Person

{

private string \_name;

private string \_surname;

private byte \_age;

public Person(string name, string surname, byte age)

{

this.\_age = age;

this.\_name = name;

this.\_surname = surname;

}

public Person() : this("", "", 0)

{

}

}

}

}

**Пример 5**

using System;

namespace GCGenerationsExample

{

class Program

{

static void Main(string[] args)

{

// Создадим несколько объектов класса Garbage

Garbage g1 = new Garbage();

Garbage g2 = new Garbage();

Garbage g3 = new Garbage(); // Выведем номер поколения,

// в котором находится кажды

// После создания все объекты попадают

// в поколение №0

Console.WriteLine("Номер поколения g1: {0}",

GC.GetGeneration(g1));

Console.WriteLine("Номер поколения g2: {0}",

GC.GetGeneration(g2));

Console.WriteLine(" Номер поколения g3: {0}", GC.GetGeneration(g3));

// Проведем сбор мусора и больше не

// ссылаться на объект g1, чтобы его уничтожил

// сборщик мусора

Console.WriteLine("Количество занятой памяти " + "до сбора мусора: {0} байт", GC.GetTotalMemory(false));

// Проводим сбор мусора в поколении 0

// Объекты, на которые

// уже нет ссылок (g1) уничтожаются, а те, на

// которые еще есть - переходят в

// более старшее поколение (1)

//g1 = null;

GC.Collect(0);

Console.WriteLine("Количество занятой памяти " + "после сбора мусора: {0} байт", GC.GetTotalMemory(false));

// Узнаем номера поколений, в котором находятся

// объекты g2 и g3

Console.WriteLine("Номер поколения g2: {0}",

GC.GetGeneration(g2));

Console.WriteLine("Номер поколения g3: {0}", GC.GetGeneration(g3));

// Проведем сбор мусора и больше не будем в коде

// ссылаться на объект g2, чтобы его уничтожил

// сборщик мусора

Console.WriteLine("Количество занятой памяти до сбора мусора: {0} байт", GC.GetTotalMemory(false));

// Проводим сбор мусора в поколении 0.

// Все объекты в нем должны быть уничтожены

GC.Collect(0);

// Но объект g2 не был уничтожен,

// ведь он находится в поколении 1. Кроме того,

// объкт g3 не перешел в старшее поколение:

g2 = null;

Console.WriteLine("Номер поколения g3: {0}",

GC.GetGeneration(g3));

// Проведем сбор мусора в поколении 1 , сравнивая

// объем занятой памяти до и после сбора мусора:

Console.WriteLine("Количество занятой памяти " + "до сбора мусора: {0} байт", GC.GetTotalMemory(false));

GC.Collect(1);

Console.WriteLine("Количество занятой памяти " + "после сбора мусора: {0} байт", GC.GetTotalMemory(false));

GC.GetTotalMemory(false);

Console.WriteLine("Номер поколения g3: {0}", GC.GetGeneration(g3));

}

}

/// Класс, создающий мусор

class Garbage

{

private int[] \_garbage = null;

public Garbage()

{

// Создаем мусор

Random rnd = new Random();

this.\_garbage = new int[rnd.Next(1000, 10000)];

for (int i = 0; i < this.\_garbage.Length; i++)

{

this.\_garbage[i] = rnd.Next();

}

}

} }