Версионность

Nonvirtual interface

Основнаязадачаэтогопаттерна—разделениепредставленияинтерфейсаиегореализации. Виртуальныефункцииобъявляютсякакprotected,аихвызовпроисходитвнутриобычных функций,которыеипредоставляютсяпользователю.

Вопервых,внутриневиртуальнойфункциивыможетеосуществлятьразличныепроверки,какдо вызовавиртуальнойфункции,такипосленего.

Вовторых,интерфейсывиртуальнойиневиртуальнойфункциинеобязанысовпадать.

Открытаяфункцияможетиметьинтерфейс,которыйбудетнаиболееудобенпользователю,а виртуальная—которыйбудетнаиболееудобендлязамещенияфункциональности.Болеетого,набори интерфейсвиртуальныхфункцийможетбытьсовершенноиным,нежелинабориинтерфейсоткрытых функций.Никтонемешаетвнутриоткрытойфункциивызватьпятьвиртуальных.

Паттернпозволяетчеткоразделитьобязанности—вэтомегодостоинствоиосновноеназначение. Открытыефункциипредоставляютинтерфейс,которыйвпервуюочередьудобенпользователю. Виртуальныефункциипредоставляютинтерфейс,которыйвпервуюочередьудобендлязамещения функциональности.

\*\*\*

Не изменяя ни одной строки кода в библиотека мы заставляем работать библиотеку правильно

public class Base

{

public void DoWork()

{

PreDoWork();

CoreDoWork();

}

void PreDoWork()

{

Console.WriteLine("Base.PreDoWork();");

}

protected virtual void CoreDoWork()

{

Console.WriteLine("Base.DoWork()");

}

public class Derived : Base

{

public int a;

protected override void CoreDoWork()

{

Console.WriteLine("Derived.DoWork()");

}

}

class Program

{

static void Main()

{

Base instance = new Derived();

instance.DoWork(); // = "Derived.DoWork()"

// Задержка.

Console.ReadKey();

}

}

\*\*\*

Перекрытие и переопределение

Оператор sealed не даст переопределяться потомкам

class BaseClass

{

// Виртуальная функция.

public virtual void SomeMetod1()

{

Console.WriteLine("1");

}

public virtual void SomeMetod2()

{

Console.WriteLine("2");

}

}

class DerivedClass : BaseClass

{

// Не является переопределением.

public new void SomeMetod1() // NEW метод - (перекрытие) не отработает

{

Console.WriteLine("3");

}

public sealed override void SomeMetod2()

{

Console.WriteLine("4");

}

}

class Program

{

static void Main()

{

BaseClass instance = new DerivedClass();

instance.SomeMetod1();

instance.SomeMetod2();

Console.ReadKey();

}

}

\*\*\*

Всегда вызывается переопределенный метод

class BaseClass

{

public void BaseClassMethod()

{

SomeMetod1();

SomeMetod2();

}

public virtual void SomeMetod1()

{

Console.WriteLine("1");

}

public virtual void SomeMetod2()

{

Console.WriteLine("2");

}

}

class DerivedClass : BaseClass

{

public new void SomeMetod1()

{

Console.WriteLine("3");

}

public override void SomeMetod2()

{

Console.WriteLine("4");

}

}

class Program

{

static void Main()

{

BaseClass instance = new DerivedClass();

instance.SomeMetod1();

instance.SomeMetod2();

instance.BaseClassMethod();

1,4 – на вывод

Console.ReadKey();

}

\*\*\*

class BaseClass

{

public virtual void SomeMethods()

{

SomeMetod1();

SomeMetod2();

}

public virtual void SomeMetod1()

{

Console.Write("1.");

}

public virtual void SomeMetod2()

{

Console.Write("2.");

}

}

class DerivedClass : BaseClass

{

public override void SomeMethods()

{

this.SomeMetod1();

this.SomeMetod2();

}

public new void SomeMetod1()

{

Console.Write("3.");

}

public override void SomeMetod2()

{

Console.Write("4.");

}

}

class Program

{

static void Main()

{

while (true)

{

Console.WriteLine("Введите ключ: или 0 или 1 или 2");

string s = Console.ReadLine();

if (s == "0") // Версия: 1.2

new BaseClass().SomeMethods();

if (s == "1") // Версия: 3.4

(new DerivedClass() as BaseClass).SomeMethods();

if (s == "2") // Версия: 3.4

new DerivedClass().SomeMethods();

Console.WriteLine("\n");

}

}

\*\*\*

AdHoc

public class Class1

{

public void Method()

{

Console.WriteLine("Class 1");

}

}

interface IInterface

{

void Method();

}

//Just for fun!

class MyClass1 : Class1, IInterface { }

class MyClass2 : Class2, IInterface { }

class MyClass3 : Class3, IInterface { }

class Program

{

static void Main()

{

IInterface[] array = { new MyClass1(), new MyClass2(), new MyClass3() };

for (var i = 0; i < 3; i++)

array[i].Method();

// Delay.

Console.ReadKey();

}

}

\*\*\*

Dynamic

DLR найти что-то с именем Method();

static void Main()

{

dynamic[] array = { new Class1(), new Class2(), new Class3() };

foreach (var item in array)

item.Method();

// Delay.

Console.ReadKey();

}

\*\*\*

public abstract class AbsBaseClass2

{

// (1)

public abstract void AbstractMethod();

// (2)

// Виртуальный метод.

public virtual void VirtualMethod()

{

Console.WriteLine("Виртуальный метод Абстрактного класса AbsBaseClass2");

}

// (3)

// Обычный метод.

public void SimpleMethod()

{

Console.WriteLine("Обычный метод Абстрактного класса AbsBaseClass2");

}

}

public class Class2 : AbsBaseClass2

{

public override void AbstractMethod()

{

Console.WriteLine("Class 2");

}

public new void SimpleMethod()

{

Console.WriteLine("Замещенный метод класса Class2");

}

object[] array = { new Class1(), new Class2(), new Class3() };

foreach (dynamic i in array)

{

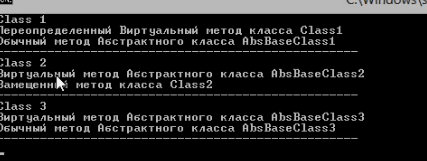
i.AbstractMethod();

i.VirtualMethod();

i.SimpleMethod();

Console.WriteLine(new string('-', 50));

}



\*\*\*

Не бывает абстрактных и виртуальных полей, а свойства и события бывают.

\*\*\*

class MyClass

{

public void SomeMethod()

{

Console.WriteLine("MyClass.SomeMethod()");

}

}

class Program

{

static void Main()

{

object instance1 = new MyClass();

// instance.SomeMethod(); Нет доступа

dynamic instance = new MyClass();

instance.SomeMethod();

// Delay.

Console.ReadKey();

}

\*\*\*



\*\*\*

Dynamic

dynamic dyn = 42;

object obj = (object)42;

dyn++;

// obj++;

obj = (int)obj + 1;

Console.WriteLine("dyn = {0}, obj = {1}", dyn, obj);

\*\*\*

class MyClass

{

static public void Method(double dbl, dynamic dyn)

{

Console.WriteLine("void Method( double dbl, dynamic dyn )");

}

static public void Method(int i, string str)

{

Console.WriteLine("void Method( int i, string str )");

}

static public void Method(double i, string str)

{

Console.WriteLine("void Method( double i, string str )");

}

}

class Program

{

static void Main()

{

dynamic d = "I'm dynamic";

dynamic d1 = new object();

MyClass.Method(3.1415, d); // перегрузка времени компиляции?

MyClass.Method(42, d); // перегрузка времени выполнения

MyClass.Method(42, d1); // перегрузка времени выполнения

//А это не компилируется!

// MyClass.Method("Hello!", d);

// Delay.

Console.ReadKey();

}

\*\*\*

Dynamic

Нельзя наследоваться от dynamic

Нельзя использовать в обобщении в интерфейсах

class Base<T>

{

public T Value { get; set; }

}

class DerivedDynamic : Base<dynamic>

{

}

class DerivedObject : Base<object>

{

}

class Program

{

static void Main()

{

DerivedDynamic instance1 = new DerivedDynamic();

instance1.Value = "Some string...";//конкатенация

instance1.Value = 20;//сложение

Console.WriteLine(42 + instance1.Value);

DerivedObject instance2 = new DerivedObject();

instance2.Value = "Some string...";

Console.WriteLine(42 + instance2.Value.ToString());

\*\*\*

Многопоточность

Join

Тот поток в котором был вызван метод Join будет ожидать завершение вторичного потока

class Program

{

// Метод выполяняющийся во Вторичном потоке.

Private static void ThreadFunc()

{

Console.WriteLine(“ID Вторичного потока: {0}”, Thread.CurrentThread.ManagedThreadId);

Console.ForegroundColor = ConsoleColor.Yellow;

for (int i = 0; i < 160; i++ )

{

Thread.Sleep(20);

Console.Write(“.”);

}

Console.ForegroundColor = ConsoleColor.Gray;

Console.WriteLine(«Вторичный поток завершился.»);

}

static void Main()

{

// Создание нового потока.

Var thread = new Thread(new ThreadStart(ThreadFunc));

Console.WriteLine(«ID Первичного потока: {0} \n», Thread.CurrentThread.GetHashCode());

Console.WriteLine(«Запуск нового потока…»);

thread.Start();

Console.WriteLine(Thread.CurrentThread.GetHashCode());

// Ожидание первичным потоком, завершения работы вторичного потока.

Thread.Join(); //TODO Снять комментарий.

Console.ForegroundColor = ConsoleColor.Green;

for (int i = 0; i < 160; i++)

{

Thread.Sleep(20);

Console.Write(“-“);

}

Console.ForegroundColor = ConsoleColor.Gray;

Console.WriteLine(«\nПервичный поток завершился.»);

Console.WriteLine(«ID Первичного потока: {0} \n», Thread.CurrentThread.GetHashCode());

// Задержка.

Console.ReadKey();

}

\*\*\*

class Program

{

private static void WriteChar(char chr, int count, ConsoleColor color)

{

Console.ForegroundColor = color;

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

{

Thread.Sleep(20);

Console.Write(chr);

}

Console.ForegroundColor = ConsoleColor.Gray;

}

// Метод выполняющийся в третичном потоке.(Запуск из вторичного потока.)

public static void Method3()

{

Console.WriteLine("Третичный поток # {0}", Thread.CurrentThread.GetHashCode());

WriteChar('3',160, ConsoleColor.Yellow);

Console.WriteLine("Третичный поток завершился.");

}

// Метод выполняющийся во вторичном потоке.(Запуск из первичного потока.)

public static void Method2()

{

Console.WriteLine("Вторичный поток # {0}", Thread.CurrentThread.GetHashCode());

WriteChar('2',80, ConsoleColor.Blue);

// Создание третичного потока.

var thread = new Thread(Method3);

thread.Start();

thread.Join();

WriteChar('2',80, ConsoleColor.Blue);

Console.WriteLine("Вторичный поток завершился.");

}

// Метод выполняющийся в первичном потоке.

static void Main()

{

Console.WriteLine("Первичный поток # {0}", Thread.CurrentThread.GetHashCode());

WriteChar('1',80, ConsoleColor.Green);

// Создание вторичного потока.

var thread = new Thread(Method2);

thread.Start();

thread.Join();

WriteChar('1', 80, ConsoleColor.Green);

Console.WriteLine("Первичный поток завершился.");

// Задержка.

Console.ReadKey();

}

}

\*\*\*

Атрибут ThreadStatic

Атрибут делает копию для каждого потока

[ThreadStatic] //TODO Снять комментарий

public static int counter;

// Рекурсивный запуск потоков.

public static void Method()

{

if (counter < 100)

{

counter++; // Увеличение счетчика вызваных методов.

Console.WriteLine(counter+" - СТАРТ --- " + Thread.CurrentThread.GetHashCode());

var thread = new Thread(Method);

thread.Start();

thread.Join(); // Закомментировать.

}

\*\*\*

Thread.Aboart

private static void Function()

{

Console.ForegroundColor = ConsoleColor.Green;

while (true)

{

try

{

Thread.Sleep(10);

Console.Write(".");

}

catch (ThreadAbortException ex)

{

Console.ForegroundColor = ConsoleColor.Red;

// Попытка 'проглотить' исключение и продолжиться выполняться в данном цикле.

// То есть вернуться в цикл и продолжить выводить counter.

Console.WriteLine("\nThreadAbortException");

for (int i = 0; i < 160; i++)

{

Thread.Sleep(20);

Console.Write(".");

}

Console.ForegroundColor = ConsoleColor.Green;

// Если не вызывать Thread.ResetAbort() - исключение повторно сгенерируется после выхода из catch{}

// Предотвращение повторной генерации ThreadAbortException!

Thread.ResetAbort();

}

}

}

static void Main()

{

Thread thread = new Thread(new ThreadStart(Function));

thread.Start();

// Даем время запуститься вторичному потоку и поработать.

Thread.Sleep(2000);

// Прервать поток. Генерируется исключение ThreadAbortException.

thread.Abort(); // Закоментировать!

// Ждать завершения потока.

thread.Join();

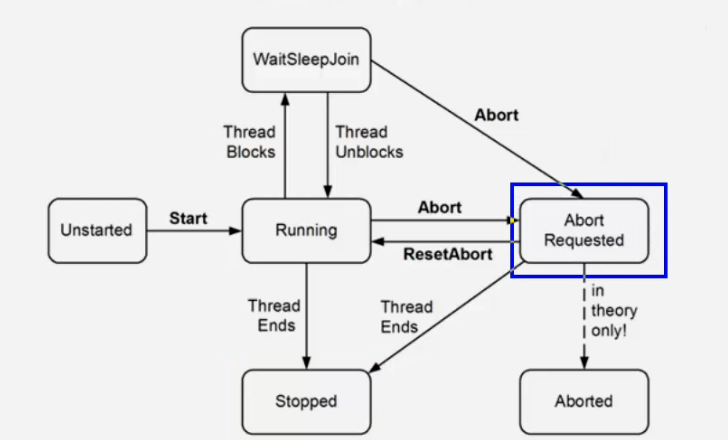
Console.ForegroundColor = ConsoleColor.White;

Console.WriteLine("\n" + new string('\_', 80));

// Задержка.

//Console.ReadKey();

}



\*\*\*

IsBackground

ЕСЛИ IsBackground = true, то завершаем работу вторичного потока по завершению первичного

static void Main()

{

var thread = new Thread(Function);

// IsBackground - устанавливает поток как фоновый. Не ждем завершения вторичного потока в данном случае.

// По умолчанию - thread.IsBackground = false;

thread.IsBackground = true; // Закомментировать!

thread.Start();

Thread.Sleep(500);

Console.WriteLine("\nЗавершение главного потока.");

}

\*\*\*

Thread.Priority

// Установить приоритеты для потоков.

work1.Priority = ThreadPriority.Highest;

work2.Priority = ThreadPriority.Lowest;

\*\*\*

Interlocked

class Kvas

{

public static int a;

public static void Met2()

{

for (int i = 0; i < 10000000; i++)

{

Interlocked.Increment(ref a);

}

}

}

class Program

{

public static void Main()

{

var t = new Thread(Kvas.Met2);

var t2 = new Thread(Kvas.Met2);

var t3 = new Thread(Kvas.Met2);

t.Start();

t2.Start();

t3.Start();

t.Join();

t2.Join();

t3.Join();

Console.WriteLine(Kvas.a);

Console.ReadKey();

}

}

\*\*\*

Monitor

class Program

{

// Объект для блокировки.

static private readonly object block = new object();

// Счетчик потоков.

static private int counter;

static private readonly Random random = new Random();

// Выполняется в отдельном потоке.

private static void Function()

{

// Управляющий поток увеличивает счетчик и ожидает

// произвольный период времени от 1 до 12 секунд.

try

{

Monitor.Enter(block); // Начало блокировки.

++counter;

}

finally

{

Monitor.Exit(block); // Конец блокировки.

}

int time = random.Next(1000, 12000);

Thread.Sleep(time);

try

{

Monitor.Enter(block); // Начало блокировки.

--counter;

}

finally

{

Monitor.Exit(block); // Конец блокировки.

}

}

// Мониторинг количества запущеных потоков.

private static void Report()

{

while (true)

{

int count;

try

{

Monitor.Enter(block);// Начало блокировки.

count = counter;

}

finally

{

Monitor.Exit(block);

}

Console.WriteLine("{0} поток(ов) активно", count);

Thread.Sleep(100);

}

}

static void Main()

{

var reporter = new Thread(Report) {IsBackground = true};

reporter.Start();

var threads = new Thread[150];

for (uint i = 0; i < 150; ++i)

{

threads[i] = new Thread(Function);

threads[i].Start();

}

Thread.Sleep(15000);

}

\*\*\*

Lock

private static void Function()

{

// Управляющий поток увеличивает счетчик и ожидает

// произвольный период времени от 1 до 12 секунд.

lock (block)

{

++counter;

}

int time = random.Next(1000, 12000);

Thread.Sleep(time);

lock (block)

{

--counter;

}

}

\*\*\*

Потоки 2. Синхронизация

Пулпотоков—этоколлекцияпотоков,которыемогутиспользоватьсядлявыполнения несколькихзадачвфоновомрежиме.Пулпотоковпозволяетразгрузитьглавныйпотокдля асинхронноговыполнениядругихзадач.

**Чтотакое«объектыядра»Windows?**

Вчисло«объектовядра»входятсамипотокиипроцессы.Онипереходятв сигнализирующеесостояниесразужепозавершении.Этооченьважнаяособенность, посколькучастотребуетсяотслеживатьмоментзавершенияпотокаилипроцесса.

**Например:**

Требуетсязавершениеработысерверногоприложенияснаборомрабочихпотоков. Управляющийпотокприэтомдолженкаким-либоспособомпроинформировать рабочиепотокиотом,чтопоразаканчиватьработу(например,установивглобальный флаг),послечегодождаться,завершениявсехрабочихпотоков.

Рабочиепотокивсвоюочередь,должнывыполнитьвсенеобходимыедействиядля своегокорректногозавершения:освободитьресурсы,проинформироватьклиентово завершенииработы,закрытьсетевыесоединенияит.п.

\*\*\*

Volatile

\*

Ключевое слово volatile можно применять к полям следующих типов:

1. Ссылочные типы.

2. Типы, такие как sbyte, byte, short, ushort, int, uint, char, float и bool.

3. Тип перечисления с одним из следующих базовых типов: byte, sbyte, short, ushort, int или uint.

4. Параметры универсальных типов, являющиеся ссылочными типами.

Ключевое слово volatile можно применить только к полям класса или структуры.

Локальные переменные не могут быть объявлены как volatile.

\*/

namespace VolatileSample

{

class Program

{

// Поля, объявленные как volatile, не проходят оптимизацию компилятором, которая предусматривает доступ посредством отдельного потока.

// Это гарантирует наличие наиболее актуального значения в поле в любое время.

// Ключевое слово гарантирует что при чтении и записи манипуляция будет происходить непосредственно с памятью а не со значениями,

// которые кэшированы в регистры процессора.

static volatile bool stop; // Без JIT оптимизации.

//static bool stop; // С JIT оптимизацией.

static void Main()

{

Console.WriteLine("Main: запускается поток на 2 секунды.");

var thread = new Thread(Worker);

thread.Start();

Thread.Sleep(2000);

stop = true;

Console.WriteLine("Main: ожидание завершения потока");

thread.Join();

}

private static void Worker(Object o)

{

// При компиляции данного кода JIT компилятор обнаружит, что переменная stop не меняется в методе.

// JIT Компилятор может создать код, заранее проверяющий состояние переменной stop

// и если она true сразу вывести результат "Worker: остановлен при x = 0"

// В противном случае JIT компилятор создает код входящий в бесконечный цикл и бесконечно увеличивающий переменную x.

// ВНИМАНИЕ! Оптимизация не производиться в режиме отладки (DEBUG).

int x = 0;

while (!stop)

{

// checked

{

x++;

}

}

// Код после JIT оптимизации, если stop не voatile:

// (Если stop - volatile - то оптимизация JIT компилятором производиться не будет.)

// int x = 0;

// if (stop != true)

// {

// while (true)

// {

// x++;

// }

// }

Console.WriteLine("Worker: остановлен при x = {0}.", x);

Console.ReadKey();

}

}

\*\*\*

VolatileRead and Write

// static volatile int stop = 0;

static int stop;

static void Main()

{

Console.WriteLine("Main: запускается поток на 2 секунды.");

var t = new Thread(Worker);

t.Start();

Thread.Sleep(2000);

Thread.VolatileWrite(ref stop, 1);

Console.WriteLine("Main: ожидание завершения потока.");

t.Join();

}

private static void Worker(Object o)

{

int x = 0;

while (Thread.VolatileRead(ref stop) != 1)

{

x++;

}

Console.WriteLine("Worker: остановлен при x = {0}.", x);

}

\*\*\*

Пул потоков

Поток заходит в очередь и выполняется, после выходит и ждет новою задачу (поток не уничтожается)

class Program

{

static void Main()

{

Console.WriteLine("Начало работы программы");

ShowThreadInfo();

Console.WriteLine("Запускаем Task1 в потоке из пула потоков");

ThreadPool.QueueUserWorkItem(new WaitCallback(Task1));

ShowThreadInfo();

Console.WriteLine("Запускаем Task2 в потоке из пула потоков");

Thread.Sleep(1000);

ThreadPool.QueueUserWorkItem(Task2);

ShowThreadInfo();

Console.WriteLine("Главный поток.");

Thread.Sleep(1000);

Console.WriteLine("Главный поток завершен.\n");

// Delay.

Console.WriteLine("Task1 и Task2 закончили работу");

ShowThreadInfo();

Console.ReadKey();

}

static void Task1(Object state)

{

Thread.CurrentThread.Name = "1";

Console.WriteLine("Поток {0}:{1}", Thread.CurrentThread.Name, Thread.CurrentThread.ManagedThreadId);

Console.WriteLine();

Thread.Sleep(200);

}

static void Task2(Object state)

{

Thread.CurrentThread.Name = "2";

Console.WriteLine("Поток {0}:{1}", Thread.CurrentThread.Name, Thread.CurrentThread.ManagedThreadId);

Thread.Sleep(200);

}

//Когда метод GetAvailableThreads возвращает значение, переменная, указанная параметром workerThreads,

//содержит число дополнительных рабочих потоков, которые могут быть запущены, а переменная, указанная параметром

//completionPortThreads, содержит число дополнительных потоков асинхронного ввода/вывода, которые могут быть запущены.

//Если доступные потоки отсутствуют, дополнительные запросы к пулу потоков будут оставаться в очереди до тех пор,

//пока в пуле потоков не появятся доступные потоки.

static void ShowThreadInfo()

{

int availableWorkThreads, availableIOThreads, maxWorkThreads, maxIOThreads;

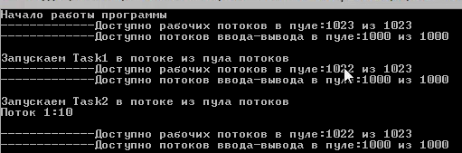
ThreadPool.GetAvailableThreads(out availableWorkThreads, out availableIOThreads);

ThreadPool.GetMaxThreads(out maxWorkThreads, out maxIOThreads);

Console.WriteLine("-------------Доступно рабочих потоков в пуле:{0} из {1}", availableWorkThreads, maxWorkThreads);

Console.WriteLine("-------------Доступно потоков ввода-вывода в пуле:{0} из {1}\n", availableIOThreads, maxIOThreads);

}



\*\*\*

Mutex

КлассWaitHandleобычноиспользуетсявкачестве базовогодляобъектовсинхронизации.

// Mutex - Примитив синхронизации, который также может использоваться в межпроцессной и междоменной синхронизации.

// MutEx - Mutual Exclusion (Взаимное Исключение).

namespace MutexSample

{

class Program

{

// Mutex - Примитив синхронизации, который также может использоваться в межпроцессорной синхронизации.

// функционирует аналогично AutoResetEvent но снабжен дополнительной логикой:

// 1. Запоминает какой поток им владеет. ReleaseMutex не может вызвать поток, который не владеет мьютексом.

// 2. Управляет рекурсивным счетчиком, указывающим, сколько раз поток-владелец уже владел объектом.

private static readonly Mutex Mutex1 = new Mutex(false, "MutexSample:AAED7056-380D-412E-9608-763495211EA8");

static void Main()

{

var threads = new Thread[5];

for (int i = 0; i < threads.Length; i++)

{

threads[i] = new Thread(new ThreadStart(Function))

{

Name = i.ToString()

};

threads[i].Start();

}

// Delay.

Console.ReadKey();

}

static void Function()

{

bool myMutex = Mutex1.WaitOne();

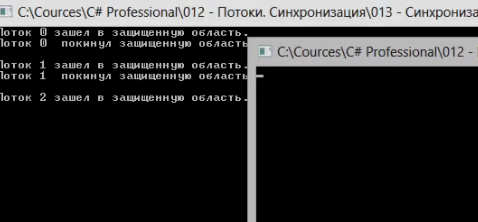
Console.WriteLine("Поток {0} зашел в защищенную область.", Thread.CurrentThread.Name);

Thread.Sleep(2000);

Console.WriteLine("Поток {0} покинул защищенную область.\n", Thread.CurrentThread.Name);

Mutex1.ReleaseMutex();

}



\*\*\*

Рекурсивное запирание

class Program

{

static void Main()

{

var instance = new SomeClass();

var thread = new Thread(instance.Method1);

thread.Start();

var thread2 = new Thread(instance.Method1);

thread2.Start();

// Delay.

thread.Join();

thread2.Join();

Console.ReadKey();

}

}

// Рекурсивное запирание.

public class SomeClass : IDisposable

{

private Mutex mutex = new Mutex();

public void Method1()

{

mutex.WaitOne();

Console.WriteLine("Method1 Start "+Thread.CurrentThread.ManagedThreadId);

Method2();

mutex.ReleaseMutex();

Console.WriteLine("Method1 End " + Thread.CurrentThread.ManagedThreadId);

}

public void Method2()

{

mutex.WaitOne();

Console.WriteLine("Method2 Start " + Thread.CurrentThread.ManagedThreadId);

Thread.Sleep(1000);

mutex.ReleaseMutex();

Console.WriteLine("Method2 End " + Thread.CurrentThread.ManagedThreadId);

}

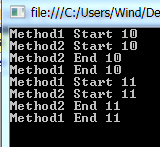
public void Dispose()

{

mutex.Dispose();

}

}



\*\*\*

Семафор

Когда есть 4 разделяемых ресурсов и нужно обрабатывать их 4 потоками

// Класс Semaphore - используется для управления доступом к пулу ресурсов.

// Потоки занимают слот семафора, вызывая метод WaitOne(), и освобождают занятый слот вызовом метода Release().

namespace MyNamespace

{

public class Program

{

private static Semaphore pool;

private static void Work(object number)

{

pool.WaitOne();

Console.WriteLine("Поток {0} занял слот семафора.", number);

Thread.Sleep(1000);

Console.WriteLine("Поток {0} -----> освободил слот.", number);

pool.Release();

}

public static void Main()

{

// Первый аргумент:

// Задаем количество слотов для использования в данный момент (не более максимального клоличества).

// Второй аргумент:

// Задаем максимальное количество слотов для данного семафора.

pool = new Semaphore(2, 4, "MySemafore65487563487");

for (int i = 1; i <= 8; i++)

{

Thread thread = new Thread(new ParameterizedThreadStart(Work));

thread.Start(i);

}

Thread.Sleep(2000);

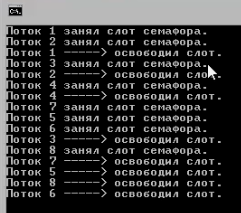
pool.Release(2);

// Задержка.

Console.ReadKey();

}

}



\*\*\*

SemaphoreSlim

class Program

{

// SemaphoreSlim - легковесный класс-семафор, который не использует объекты синхронизации ядра.

static readonly SemaphoreSlim slim = new SemaphoreSlim(1, 2);

static void Main()

{

Thread[] threads = { new Thread(Function), new Thread(Function), new Thread(Function) };

for (int i = 0; i < threads.Length; i++)

{

threads[i].Name = i.ToString();

threads[i].Start();

}

Thread.Sleep(1000);

slim.Release(); // Возможен принудительный сброс из потока владельца семафора.

// Delay.

Console.ReadKey();

}

static void Function()

{

slim.Wait();

Console.WriteLine("Поток {0} начал работу.", Thread.CurrentThread.Name);

Thread.Sleep(5000);

Console.WriteLine("Поток {0} закончил работу.\n", Thread.CurrentThread.Name);

slim.Release();

}

}

\*\*\*

AutoResetEvent

Позволяет разблокировать поток по событию

AutoResetEventпозволяетпотокамвзаимодействоватьдругсдругомпутем передачисигналов.Какправило,этотклассиспользуется,когдапотокам требуетсяисключительныйдоступкресурсу.

ВызовSetсигнализируетсобытиюAutoResetEventонеобходимости освобожденияожидающегопотока.СобытиеAutoResetEventостаетсяв сигнальномсостояниидоосвобожденияодногоожидающегопотока,азатем возвращаетсявнесигнальноесостояние.Еслинетожидающихпотоков, состояниеостаетсясигнальнымбесконечно

// AutoResetEvent - Уведомляет ожидающий поток о том, что произошло событие.

static readonly AutoResetEvent auto = new AutoResetEvent(false);

static void Main()

{

Console.WriteLine("Нажмите на любую клавишу для перевода AutoResetEvent в сигнальное состояние.\n");

var thread = new Thread(Function1);

thread.Start();

Console.ReadKey();

auto.Set(); // Послать сигнал первому потоку

Console.ReadKey();

auto.Set(); // Послать сигнал второму потоку

// Delay.

Console.ReadKey();

}

static void Function1()

{

Console.WriteLine("Красный свет");

auto.WaitOne(); // после завершения WaitOne() AutoResetEvent автоматически переходит в несигнальное состояние.

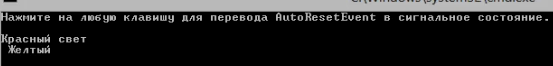
Console.WriteLine("Желтый");

auto.WaitOne(); // после завершения WaitOne() AutoResetEvent автоматически переходит в несигнальное состояние.

Console.WriteLine("Зеленый");

}

}



\*\*\*

ManualResetEvent

Посылает всем потокам сигнал о разблокировании

ManualResetEventостаетсявсигнальномсостояниидотогомомента,как онобудетсноваустановленовручную.Тоесть,вызовыкWaitOneнемедленно возвращаются.

ManualResetEventпозволяетпотокамвзаимодействоватьдругсдругом путемпередачисигналов.Обычноэтовзаимодействиекасаетсязадачи, которуюодинпотокдолжензавершитьдотого,какдругойпродолжитработу.

\*

ManualResetEvent

ManualResetEvent позволяет потокам взаимодействовать друг с другом путем передачи сигналов.

Обычно это взаимодействие касается задачи, которую один поток должен завершить до того, как другой продолжит работу.

Когда поток начинает работу, которая должна быть завершена до продолжения работы других потоков,

он вызывает метод Reset для того, чтобы поместить ManualResetEvent в несигнальное состояние.

Этот поток можно понимать как контролирующий ManualResetEvent.

Потоки, которые вызывают метод WaitOne в ManualResetEvent, будут заблокированы, ожидая сигнала.

Когда контролирующий поток завершит работу, он вызовет метод Set для сообщения о том, что ожидающие потоки могут продолжить работу.

Все ожидающие потоки освобождаются.

\*/

namespace ManualResetEventNs

{

class Program

{

// ManualResetEvent - Уведомляет один или более ожидающих потоков о том, что произошло событие.

private static ManualResetEvent manual = new ManualResetEvent(false);

static void Main()

{

Console.WriteLine("Нажмите на любую клавишу для перевода ManualResetEvent в сигнальное состояние.\n");

Thread[] threads = { new Thread(Function1), new Thread(Function2) };

foreach (Thread thread in threads)

thread.Start();

Console.ReadKey();

manual.Set(); // Просылает сигнал всем потокам.

// Delay.

Console.ReadKey();

}

static void Function1()

{

Console.WriteLine("Поток 1 запущен и ожидает сигнала.");

manual.WaitOne(); // после завершения WaitOne() ManualResetEvent остаеться в сигнальном сотоянии.

Console.WriteLine("Поток 1 завершается.");

}

static void Function2()

{

Console.WriteLine("Поток 2 запущен и ожидает сигнала.");

manual.WaitOne(); // после завершения WaitOne() ManualResetEvent остаеться в сигнальном сотоянии.

Console.WriteLine("Поток 2 завершается.");

}

}

\*\*\*

class Work

{

readonly ManualResetEvent manual;

readonly Thread thread;

public Work(string name, ManualResetEvent manual)

{

this.thread = new Thread(this.Run) {Name = name};

this.manual = manual;

this.thread.Start();

}

void Run()

{

Console.WriteLine("Запущен поток " + thread.Name);

for (int i = 0; i < 10; i++)

{

Console.Write(". ");

Thread.Sleep(200);

}

Console.WriteLine("\nПоток " + thread.Name + " завершен");

// Уведомление о событии происходит при помощи вызова метода Set()

manual.Set();

}

}

class ManualEventDemo

{

static void Main()

{

// false - установка несигнального состояния.

var manual = new ManualResetEvent(false);

var thread = new Work("1", manual);

Console.WriteLine("Основной поток ожидает событие.\n");

manual.WaitOne(); // Ожидать уведомления о событии.

Console.WriteLine("\nОсновной поток получил уведомление о событии от первого потока.\n");

manual.Reset(); // Сбрасывает в несигнальное состояние.

thread = new Work("2", manual);

Console.WriteLine("Основной поток ожидает событие.\n");

manual.WaitOne(); // Ожидать уведомления о событии.

Console.WriteLine("\nОсновной поток получил уведомление о событии от второго потока.");

// Delay.

Console.ReadKey();

}

\*\*\*

WaitOrTimerCallback

// Блокировка потоков, бесконечно ожидающих доступ к объекту ядра - не рациональная трата памяти.

// Пул потока предлагает механизм вызова метода.

class Program

{

static void Main()

{

AutoResetEvent auto = new AutoResetEvent(false);

WaitOrTimerCallback callback = new WaitOrTimerCallback(CallbackMethod);

// auto - от кого ждать сингнал

// callback - что выполнять

// null - 1-й аргумент Callback метода

// 1000 - интервал между вызовами Callback метода

// если true - вызвать Callback метод один раз. Если false - вызывать Callback метод с интервалом.

// ThreadPool.RegisterWaitForSingleObject(auto, callback, null, Timeout.Infinite, true);

var waitHandle = ThreadPool.RegisterWaitForSingleObject(auto, callback, null, 1000, false);

Console.WriteLine("S - сигнал, Q - выход");

while (true)

{

string operation = Console.ReadKey(true).KeyChar.ToString().ToUpper();

if (operation == "S")

{

auto.Set();

}

if (operation == "Q")

{

waitHandle.Unregister(auto);

break;

}

}

Console.ReadKey();

}

static void CallbackMethod(object state, bool timedOut)

{

Thread.Sleep(5000);

Console.WriteLine("Signal");

}

}

\*\*\*

GlobalEvent

3 проги жду сигнала

static EventWaitHandle manual = null;

static void Main()

{

// Если объект ядра с именем GlobalEvent уже существует будет получена ссылка на него.

// false - несигнальное состояние.

// ManualReset - тип событиия.

// GlobalEvent - имя по которому все приложения будут слушать событие.

manual = new EventWaitHandle(false, EventResetMode.ManualReset, "GlobalEvent::GUID");

Thread thread = new Thread(Function);

thread.IsBackground = true;

thread.Start();

Console.WriteLine("Нажмите любую клавишу для начала работы потока.");

Console.ReadKey();

// Перевод события в сигнальное состояние.

// Все приложения, которые используют событие с именем GlobalEvent, получат оповещение о переходе в сигнальное состояние.

manual.Set();

// Delay.

Console.ReadKey();

}

static void Function()

{

manual.WaitOne();

while (true)

{

Console.WriteLine("Hello world!");

Thread.Sleep(300);

}

}

\*\*\*

WaitHandle

class Program

{

static WaitHandle[] events = new WaitHandle[] { new AutoResetEvent(false), new AutoResetEvent(false) };

static Random random = new Random();

static void Main()

{

DateTime dateTime = DateTime.Now;

Console.WriteLine("Главный поток ожидает завершения ОБЕИХ задач.\n");

// Очередь для двух задач в двух разных потоках.

ThreadPool.QueueUserWorkItem(new WaitCallback(Task1), events[0]);

ThreadPool.QueueUserWorkItem(Task2, events[1]);

// Ожидание пока все задачи завершаться.

WaitHandle.WaitAll(events);

// Время отображаемое ниже, должно совпадать с продолжительностью выполнения самой длинной задачи.

Console.WriteLine("Обе задачи завершены (время ожидания = {0})", (DateTime.Now - dateTime).TotalMilliseconds);

dateTime = DateTime.Now;

Console.WriteLine("\nОжидание завершения одной из задач.");

ThreadPool.QueueUserWorkItem(new WaitCallback(Task1), events[0]);

ThreadPool.QueueUserWorkItem(Task2, events[1]);

// Ожидание пока одна из задач не завершится.

int index = WaitHandle.WaitAny(events);

// Время отображаемое ниже, должно совпадать с продолжительностью выполнения самой короткой задачи.

Console.WriteLine("Задача {0} завершилась первой (время ожидания = {1}).", index + 1, (DateTime.Now - dateTime).TotalMilliseconds);

// Задержка.

Console.ReadKey();

}

static void Task1(Object state)

{

var auto = (AutoResetEvent)state;

int time = 1000 \* random.Next(2, 10);

Thread.Sleep(time);

Console.WriteLine("Задача 1 выполнена за {0} миллисенунд.", time);

auto.Set();

}

static void Task2(Object state)

{

var auto = (AutoResetEvent)state;

int time = 1000 \* random.Next(2, 10);

Thread.Sleep(time);

Console.WriteLine("Задача 2 выполнена за {0} миллисенунд.", time);

auto.Set();

}

}

\*\*\*

АССИНХРОННАЯ МОДЕЛЬ

Async

class Program

{

static void Main()

{

Console.WriteLine("Первичный поток: Id {0}", Thread.CurrentThread.ManagedThreadId);

var myDelegate = new Action(Method);

// Выполнение метода Method в отдельном потоке, взятом из пула потоков.

myDelegate.BeginInvoke(null, null);

Console.WriteLine("Main");

// Delay.

for (int i = 0; i < 80; i++)

{

Thread.Sleep(50);

Console.Write(".");

}

Console.ReadKey();

}

// Метод для выполнения в отдельном потоке.

static void Method()

{

Console.ForegroundColor = ConsoleColor.Green;

Console.WriteLine("\nАсинхронный метод запущен.");

Console.WriteLine("\nВторичный поток: Id {0}", Thread.CurrentThread.ManagedThreadId);

for (int i = 0; i < 80; i++)

{

Thread.Sleep(60);

Console.Write("A");

}

Console.WriteLine("Асинхронная операция завершена.\n");

Console.ForegroundColor = ConsoleColor.Gray;

}

}

\*\*\*

EndInvoke

static void Main()

{

Console.WriteLine("Первичный поток: Id {0}", Thread.CurrentThread.ManagedThreadId);

var myDelegate = new Action(Method);

// BeginInvoke - выполняем метод Method в отдельном потоке, взятом из пула потоков.

// IAsyncResult - представляет состояние асинхронной операции.

IAsyncResult asyncResult = myDelegate.BeginInvoke(null, null);

Console.WriteLine("Первичный поток продолжает работать.");

// Ожидание завершения асинхронной операции.

myDelegate.EndInvoke(asyncResult);

Console.WriteLine("Первичный поток завершил работу.");

// Delay.

Console.ReadKey();

}

// Метод для выполнения в отдельном потоке.

static void Method()

{

Console.ForegroundColor = ConsoleColor.Green;

Console.WriteLine("\nАсинхронный метод запущен.");

Console.WriteLine("\nВторичный поток: Id {0}", Thread.CurrentThread.ManagedThreadId);

for (int i = 0; i < 80; i++)

{

Thread.Sleep(50);

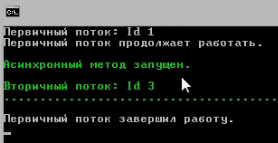
Console.Write(".");

}

Console.WriteLine("Асинхронная операция завершена.\n");

Console.ForegroundColor = ConsoleColor.Gray;

}



\*\*\*

static void Main()

{

var myDelegate = new Func<int,int,int>(Add);

// Так как класс делегата сообщается с методами, которые принимают два целочисленных параметра, метод BeginInvoke также

// начинает принимать два дополнительных параметра, кроме двух последних постоянных аргументов.

IAsyncResult asyncResult = myDelegate.BeginInvoke(1, 2, null, null);

// Ожидание завершения асинхронной операции и получение результата работы метода.

int result = myDelegate.EndInvoke(asyncResult);

Console.WriteLine("Результат = " + result);

// Delay.

Console.ReadKey();

}

// Метод для выполнения в отдельном потоке.

static int Add(int a, int b)

{

Thread.Sleep(2000);

return a + b;

}

}

\*\*\*

static void Main()

{

var myDelegate = new Func<int,int,int>(Add);

// Запуск асинхронной задачи.

IAsyncResult asyncResult = myDelegate.BeginInvoke(1, 2, null, null);

Console.WriteLine("Асинхронный метод запущен. Метод Main продолжает работать.");

Console.WriteLine("Метод Main ожидает завершения асинхронной задачи.");

Console.WriteLine(asyncResult.AsyncWaitHandle.GetType());

// AsyncWaitHandle типа WaitHandle, переходит в сигнальное состояние при завершении асинхронной операции.

asyncResult.AsyncWaitHandle.WaitOne();

Console.WriteLine("Асинхронный метод завершен.");

// Получение результата асинхронной операции.

int result = myDelegate.EndInvoke(asyncResult);

// Закрываем WaitHandle.

asyncResult.AsyncWaitHandle.Close();

Console.WriteLine("Результат = " + result);

// Delay.

Console.ReadKey();

}

// Метод для выполнения в отдельном потоке.

static int Add(int a, int b)

{

Thread.Sleep(3000);

return a + b;

}

}

\*\*\*

IsCopmlete

class Program

{

static void Main()

{

var myDelegate = new Func<int,int,int>(Add);

// Запуск асинхронной задачи.

IAsyncResult asyncResult = myDelegate.BeginInvoke(1, 2, null, null);

Console.WriteLine("Асинхронный метод запущен. Метод Main продолжает работать.");

// Выполнение цикла до тех пор, пока работает асинхронная операция.

while (!asyncResult.IsCompleted)

{

Thread.Sleep(100);

Console.Write(".");

}

// Получение результата асинхронной операции.

int result = myDelegate.EndInvoke(asyncResult);

Console.WriteLine("\nРезультат = " + result);

// Delay.

Console.ReadKey();

}

// Метод для выполнения в отдельном потоке.

static int Add(int a, int b)

{

Thread.Sleep(3000);

return a + b;

}

}

\*\*\*

AsyncCallback

class Program

{

static void Main()

{

Console.WriteLine("Первичный поток: Id {0}", Thread.CurrentThread.ManagedThreadId);

var myDelegate = new Action(Method);

// Делегат, метод которого будет запущен по завершению асинхронной операции.

var callback = new AsyncCallback(HandleCompletion);

// Первый параметр:

// Принимает метод обратного вызова, который должен сработать по завершению асинхронной операции.

// Второй параметр:

// Дополнительный объект хранящий состояние, который будет доступен в методе обратного вызова.

myDelegate.BeginInvoke(callback, "Hello world");

Console.WriteLine("Первичный поток продолжает работать.");

// Delay.

Console.ReadKey();

}

// Метод для выполнения в отдельном потоке.

static void Method()

{

Console.ForegroundColor = ConsoleColor.Green;

Console.WriteLine("\nАсинхронный метод запущен.");

Console.WriteLine("\nВторичный поток: Id {0}", Thread.CurrentThread.ManagedThreadId);

for (int i = 0; i < 80; i++)

{

Thread.Sleep(50);

Console.Write(".");

}

Console.WriteLine("Асинхронная операция завершена.\n");

Console.ForegroundColor = ConsoleColor.Gray;

}

// Callback метод для обработки завершения асинхронной операции.

static void HandleCompletion(IAsyncResult asyncResult)

{

Console.ForegroundColor = ConsoleColor.Yellow;

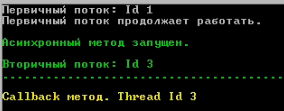
Console.WriteLine("Callback метод. Thread Id {0}", Thread.CurrentThread.ManagedThreadId);

Console.WriteLine("Информация связанная с асинхронной операцией - " + asyncResult.AsyncState);

Console.ForegroundColor = ConsoleColor.Gray;

}

}



\*\*\*

AsyncState

static void Main()

{

Console.WriteLine("Первичный поток: Id {0}", Thread.CurrentThread.ManagedThreadId);

var myDelegate = new Action(Method);

// Делегат, метод которого будет запущен по завершению асинхронной операции.

var callback = new AsyncCallback(HandleCompletion);

// Первый параметр:

// Принимает метод обратного вызова, который должен сработать по завершению асинхронной операции.

// Второй параметр:

// Дополнительный объект хранящий состояние, который будет доступен в методе обратного вызова.

myDelegate.BeginInvoke(callback, "Hello world");

Console.WriteLine("Первичный поток продолжает работать.");

// Delay.

Console.ReadKey();

}

// Метод для выполнения в отдельном потоке.

static void Method()

{

Console.ForegroundColor = ConsoleColor.Green;

Console.WriteLine("\nАсинхронный метод запущен.");

Console.WriteLine("\nВторичный поток: Id {0}", Thread.CurrentThread.ManagedThreadId);

for (int i = 0; i < 80; i++)

{

Thread.Sleep(50);

Console.Write(".");

}

Console.WriteLine("Асинхронная операция завершена.\n");

Console.ForegroundColor = ConsoleColor.Gray;

}

// Callback метод для обработки завершения асинхронной операции.

static void HandleCompletion(IAsyncResult asyncResult)

{

Console.ForegroundColor = ConsoleColor.Yellow;

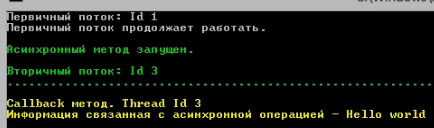
Console.WriteLine("Callback метод. Thread Id {0}", Thread.CurrentThread.ManagedThreadId);

Console.WriteLine("Информация связанная с асинхронной операцией - " + asyncResult.AsyncState);

Console.ForegroundColor = ConsoleColor.Gray;

}

}



\*\*\*

AsyncCallBack

class Program

{

// Метод для выполнения в отдельном потоке.

static int Method(int a, int b)

{

// Thread.CurrentThread.IsBackground = false;

Console.WriteLine("Вторичный поток: Id {0}", Thread.CurrentThread.ManagedThreadId);

Thread.Sleep(3000);

return a + b;

}

// Метод обработки завершения асинхронной операции.

static void CallBack(IAsyncResult iAsyncResult)

{

// Получение экземпляра делегата, на котором была вызвана асинхронная операция.

var asyncResult = iAsyncResult as AsyncResult;

var caller = (Func<int,int,int>)asyncResult.AsyncDelegate;

// Получение результатов асинхронной операции.

int sum = caller.EndInvoke(iAsyncResult);

string result = string.Format(iAsyncResult.AsyncState.ToString(), sum);

Console.WriteLine("Результат асинхронной операции: " + result);

}

static void Main()

{

Console.WriteLine("Первичный поток: Id {0}", Thread.CurrentThread.ManagedThreadId);

var myDelegate = new Func<int,int,int>(Method);

myDelegate.BeginInvoke(1, 2, CallBack, "a + b = {0}");

Console.WriteLine("Первичный поток завершил работу.");

// Delay.

Console.ReadKey();

}

}

\*\*\*

Асинхронное чтение файла

static void Main()

{

var stream = new FileStream("file.txt", FileMode.Open, FileAccess.Read);

var array = new byte[stream.Length];

// Синхронный вызов метода, который считает все байты файла file.txt в массив.

// stream.Read(array, 0, array.Length);

// Асинхронный вызов метода чтения байтов файла.

IAsyncResult asyncResult = stream.BeginRead(array, 0, array.Length, null, null);

Console.WriteLine("Чтение файла ...");

// Ожидание завершения чтения файла.

stream.EndRead(asyncResult);

foreach (byte item in array)

Console.Write(item+" ");

stream.Close();

// Delay.

Console.ReadKey();

}

\*\*\*

static void Main()

{

var stream = new FileStream("file.txt", FileMode.OpenOrCreate, FileAccess.ReadWrite);

byte[] array = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20 };

stream.BeginWrite(array, 0, array.Length, new AsyncCallback(CleanUp), stream);

// Delay.

Console.ReadKey();

}

static void CleanUp(IAsyncResult asyncResult)

{

Console.WriteLine("Файл записан.");

var stream = asyncResult.AsyncState as FileStream;

if (stream != null)

stream.Close();

}

}

\*\*\*

TPL