任务：任务一旦完成就不能重新启动它。因此，除非重新创建任务，否则没有方法重新运行任务。

任务ID：Task的属性Id。Id为只读属性，是唯一的，无序的。

CurrentId属性获取当前执行的任务的ID。

例：程序task\_test2

// Copyright 2016.刘珅珅

// author：刘珅珅

// 任务ID

using *System*;

using *System*.*Collections*.*Generic*;

using *System*.*Linq*;

using *System*.*Text*;

using *System*.*Threading*;

using *System*.*Threading*.*Tasks*;

namespace task\_test2

{

class TaskTest

{

// 任务函数

static void MyTask()

{

*Console*.*WriteLine*("MyTask() #" + *Task*.*CurrentId* + " starting.");

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

{

*Thread*.*Sleep*(500);

*Console*.*WriteLine*("In MyTask() #" + *Task*.*CurrentId* + ", count is " + i);

}

*Console*.*WriteLine*("MyTask #" + *Task*.*CurrentId* + " terminating");

}

static void Main(string[] args)

{

*Console*.*WriteLine*("Main thread starting.");

*Task* task1 = new *Task*(MyTask);

*Task* task2 = new *Task*(MyTask);

task1.*Start*();

task2.*Start*();

*Console*.*WriteLine*("Task ID for task1 is " + task1.*Id*);

*Console*.*WriteLine*("Task ID for task2 is " + task2.*Id*);

// 任务执行期间，保持Main()运行不退出

for (int i = 0; i < 60; ++i)

{

*Console*.*Write*(".");

*Thread*.*Sleep*(100);

}

*Console*.*WriteLine*("Main thread ending.");

}

}

}

输出结果：

MyTask() #2 starting.

.....In MyTask() #2, count is 0

In MyTask() #1, count is 0

....In MyTask() #2, count is 1

.In MyTask() #1, count is 1

....In MyTask() #2, count is 2

.In MyTask() #1, count is 2

....In MyTask() #2, count is 3

.In MyTask() #1, count is 3

....In MyTask() #2, count is 4

.In MyTask() #1, count is 4

....In MyTask() #2, count is 5

In MyTask() #1, count is 5

.....In MyTask() #2, count is 6

In MyTask() #1, count is 6

.....In MyTask() #2, count is 7

In MyTask() #1, count is 7

.....In MyTask() #2, count is 8

In MyTask() #1, count is 8

.....In MyTask() #1, count is 9

MyTask #1 terminating

.In MyTask() #2, count is 9

MyTask #2 terminating

.........Main thread ending.

从结果中可以看出，主线程，task1和task2是并行执行，互不干扰。

任务的等待

例：程序task\_test3

// Copyright 2016.刘珅珅

// author：刘珅珅

// 任务的等待

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading;

using System.Threading.Tasks;

namespace task\_test3

{

class TaskTestcs

{

// 任务函数

static void MyTask()

{

Console.WriteLine("MyTask() #" + Task.CurrentId + " starting.");

for (int i = 0; i < 5; ++i)

{

Thread.Sleep(500);

Console.WriteLine("In MyTask() #" + Task.CurrentId + ", count is " + i);

}

Console.WriteLine("MyTask #" + Task.CurrentId + " terminating");

}

static void Main(string[] args)

{

Console.WriteLine("Main thread starting.");

Task task1 = new Task(MyTask);

Task task2 = new Task(MyTask);

// 运行任务

task1.Start();

task2.Start();

Console.WriteLine("Task ID for task1 is " + task1.Id);

Console.WriteLine("Task ID for task2 is " + task2.Id);

// 暂停Main()直到task1和task2完成

task1.Wait();

task2.Wait();

Console.WriteLine("Main thread ending.");

}

}

}

输出结果：

Main thread starting.

Task ID for task1 is 1

Task ID for task2 is 2

MyTask() #2 starting.

MyTask() #1 starting.

In MyTask() #2, count is 0

In MyTask() #1, count is 0

In MyTask() #2, count is 1

In MyTask() #1, count is 1

In MyTask() #2, count is 2

In MyTask() #1, count is 2

In MyTask() #2, count is 3

In MyTask() #1, count is 3

In MyTask() #2, count is 4

MyTask #2 terminating

In MyTask() #1, count is 4

MyTask #1 terminating

Main thread ending.

使用TaskFactory类启动任务，将Lambda表达式用作任务

当方法只是作为单一用途的任务时，Lambda表达式就特别有用。

例：程序task\_test4

// Copyright 2016.刘珅珅

// author：刘珅珅

// TaskFactory与Lambda表达式

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading;

using System.Threading.Tasks;

namespace task\_test4

{

class TaskTest

{

static void Main(string[] args)

{

Console.WriteLine("Main thread starting.");

// 使用TaskFactory任务工厂启动任务

// 使用Lambda表达式定义一个任务入口方法

Task task = Task.Factory.StartNew(() =>

{

Console.WriteLine("Task starting.");

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

{

Thread.Sleep(500);

Console.WriteLine("Task count is " + i);

}

Console.WriteLine("Task terminating");

});

// 等待直到任务完成

task.Wait();

// 任务的清除

// Dispose()方法必须在Wait()或类似的表示

// 任务完成的方法调用完成之后才能调用

// Dispose()释放任务相关的资源，一般情况下

// 任务的资源由垃圾回收自动释放，Dispose()

// 可以在此之前释放资源

// Dispose()方法主要用于创建并放弃许多任务的程序

task.Dispose();

Console.WriteLine("Main thread ending.");

}

}

}

输出结果：

Main thread starting.

Task starting.

Task count is 0

Task count is 1

Task count is 2

Task count is 3

Task count is 4

Task count is 5

Task count is 6

Task count is 7

Task count is 8

Task count is 9

Task terminating

Main thread ending.

任务延续：在一个任务完成时自动开始的任务

public Task ContinueWith(Action<Task> continuationAction);

其中，continuationAction指定在主调任务完成后将运行的任务。这个委托有一个Task类型的参数，其使用的Action委托版本为：

public delegate void Action<in T>(T obj) // 是一个逆变类型的委托

例：程序task\_test5

// Copyright 2016.刘珅珅

// author：刘珅珅

// 任务延续

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading;

using System.Threading.Tasks;

namespace task\_test5

{

class TaskTest

{

static void MyTask()

{

Console.WriteLine("MyTask() starting.");

for (int i = 0; i < 5; ++i)

{

Thread.Sleep(500);

Console.WriteLine("In MyTask() count is " + i);

}

Console.WriteLine("MyTask terminating.");

}

// 延续任务入口点方法

static void ContTask(Task t)

{

Console.WriteLine("Continuation starting.");

for (int i = 0; i < 5; ++i)

{

Thread.Sleep(500);

Console.WriteLine("Continuation count is " + i);

}

Console.WriteLine("Continuation terminating.");

}

static void Main(string[] args)

{

Console.WriteLine("Main thread starting.");

// 起始任务

Task task = new Task(MyTask);

// 延续任务

Task taskCont = task.ContinueWith(ContTask);

task.Start();

// 延续任务等待

// 起始任务不需要等待

// 它肯定在延续任务结束之前结束

taskCont.Wait();

task.Dispose();

taskCont.Dispose();

Console.WriteLine("Main thread ending.");

}

}

}

输出结果：

Main thread starting.

MyTask() starting.

In MyTask() count is 0

In MyTask() count is 1

In MyTask() count is 2

In MyTask() count is 3

In MyTask() count is 4

MyTask terminating.

Continuation starting.

Continuation count is 0

Continuation count is 1

Continuation count is 2

Continuation count is 3

Continuation count is 4

Continuation terminating.

Main thread ending.

任务返回值：

任务可以返回值，可以使用任务计算某种结果，支持并行计算。主调线程将阻塞，直到结果准备就绪，不需要执行特殊的同步。

要返回结果，需要使用Task类的泛型，即Task<TResult>

两个常用的构造函数：

public Task(Func<TResult> function)

public Task(Func<Object, TResult> function, Object state)

其中，function是要运行的委托，它是Func类型，不再是Action类型。第二种类型创建一个带有Object类型实参（在state中传递）的任务

任务工厂泛型TaskFactory<TResult>与上面两种构造函数对应的StartNew()方法为：

public Task<TResult> StartNew(Func<TResult> function)

public Task<TResult> StartNew(Func<Object, TResult> function, Object state)

所有情况下，任务返回的值都通过Task类的Result属性获得：

public TResult Result {get; internal set;}

set存取器是内部存取器，Result属性相对于外部代码是只读的。

例：程序task\_test6

// Copyright 2016.刘珅珅

// author：刘珅珅

// 任务返回值

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading;

using System.Threading.Tasks;

namespace task\_test6

{

class TaskTest

{

// 任务方法：返回bool值

static bool MyTask()

{

Console.WriteLine("Task starting.");

Thread.Sleep(2000);

Console.WriteLine("Task terminating.");

return true;

}

// 任务方法：带有参数，返回int

static int SumIt(object v)

{

int x = (int)v;

int sum = 0;

for (; x > 0; --x)

sum += x;

return sum;

}

static void Main(string[] args)

{

Console.WriteLine("Main thread starting.");

Task<bool> task = Task<bool>.Factory.StartNew(MyTask);

// 检索任务的返回结果，主线程会阻塞，直到任务运行完毕并返回结果

// 如果不检索任务的返回结果，则主线程不会阻塞，任务有可能不会执行完

Console.WriteLine("After running MyTask. The result is " + task.Result);

Task<int> task2 = Task<int>.Factory.StartNew(SumIt, 5);

Console.WriteLine("After running SumIt. The result is " + task2.Result);

// 如果不检索结果就调用Dispose()方法，

// 有可能会抛出异常，因为任务没有执行完毕

task.Dispose();

task2.Dispose();

Console.WriteLine("Main thread terminating.");

}

}

}

输出结果：

Main thread starting.

Task starting.

Task terminating.

After running MyTask. The result is True

After running SumIt. The result is 15

Main thread terminating.

并行任务类：Parallel

Parallel类定义了For()，ForEach()和Invoke()，这是一个静态类。

For()执行并行for循环，ForEach()执行并行foreach循环，支持数据并行性。Invoke()方法支持两个或多个方法的并行执行，支持任务并行性。

Invoke()并行运行方法

例：程序task\_test7

// Copyright 2016.刘珅珅

// author：刘珅珅

// 并行方法

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading;

using System.Threading.Tasks;

namespace task\_test7

{

class TaskTest

{

static void Main(string[] args)

{

Console.WriteLine("Main thread starting.");

// Invoke()将启动传递给它的所有方法，

// 并等待所有方法完成。不需要Wait()这样的方法

// 也无法保证方法执行的顺序

// 使用语句Lambda传递方法

Parallel.Invoke(() =>

{

Console.WriteLine("Expression #1 starting.");

for (int i = 0; i < 5; ++i)

{

Thread.Sleep(500);

Console.WriteLine("Expression #1 count is " + i);

}

Console.WriteLine("Expression #1 terminating.");

},

() => {

Console.WriteLine("Expression #2 starting.");

for (int i = 0; i < 5; ++i)

{

Thread.Sleep(500);

Console.WriteLine("Expression #2 count is " + i);

}

Console.WriteLine("Expression #2 terminating.");

}

);

Console.WriteLine("Main thread ending.");

}

}

}

PLINQ：并行版本的LINQ

例：程序task\_test8

// Copyright 2016.刘珅珅

// author：刘珅珅

// PLINQ：并行查询

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace task\_test8

{

class TaskTest

{

static void Main(string[] args)

{

int[] data = new int[10000000];

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

data[i] = i;

data[1000] = -1;

data[14000] = -2;

data[15000] = -3;

data[676000] = -4;

data[8024540] = -5;

data[9908000] = -6;

// 使用PLNQ查询负值

var negatives = from val in data.AsParallel()

where val < 0

select val;

// 执行PLNQ查询

// 每次执行查询，得到的结果的顺序不尽相同

// 并行查询，任务负载，可以的处理器不同，

// 会导致结果的顺序不同

foreach (var v in negatives)

Console.Write(v + " ");

Console.WriteLine();

}

}

}

有序的并行查询：AsOrdered()

例：程序task\_test9

// Copyright 2016.刘珅珅

// author：刘珅珅

// 有序的PLNQ

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace task\_test9

{

class TaskTest

{

static void Main(string[] args)

{

int[] data = new int[10000000];

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

data[i] = i;

data[1000] = -2;

data[14000] = -1;

data[15000] = -3;

data[676000] = -4;

data[8024540] = -5;

data[9908000] = -6;

// 使用PLNQ查询负值

// 有序查询，反映源序列的顺序

var negatives = from val in data.AsParallel().AsOrdered()

where val < 0

select val;

// 执行PLNQ查询

//查找的结果顺序与源数据

// 中的顺序一致

foreach (var v in negatives)

Console.Write(v + " ");

Console.WriteLine();

}

}

}