**.NET程序集高级-多线程进阶**

**之**

**并行编程**

* **什么是并行编程？**

.NET提供Task Parallel Library(TPL)任务并行库中的类System.Threading.Tasks可以将应用程序的大量任务（如迭代处理大文件的读写操作，即数据并行）分配给应用程序的线程池里面线程处理。

* **并行和并发有什么不同？**

一旦进行并行操作，CRL会将任务分配给线程池里面尽量多的线程处理，也就是说，并行操作至少会使用线程池里面一个线程。而并发中，每一个线程的使用，都必须进行手动的创建。

* **并行有什么好处？**
* 使用尽可能多的线程处理任务，提高CPU使用效率，特别是在多核CPU中使用，会更加明显。
* TPL可以处理工作分区、线程调度、状态管理和低级别细节操作。
* 并行处理有哪些？
* 数据并行
* 任务并行
* LINQ查询并行
* **如何使用数据并行？**

在迭代处理大文件数据时，使用数据并行是最合适不过的了。类System.Threading.Tasks.Parallel提供了静态方法ForEach()和For()并行数据处理。

private void btStartParallel\_Click(object sender, EventArgs e)

{

//可迭代对象

//获取指定目录的所有.jpg文件名称

string[] files = Directory.GetFiles(

Directory.GetCurrentDirectory() + @"\pictures",

"\*.jpg",

SearchOption.AllDirectories

);

//并行数据处理会阻塞调用线程

//若不通过Task.Factory.StartNew或Thread创建线程，则UI主线程会阻塞

//也就是说，Parallel.ForEach()或Parallel.For()会阻塞调用线程

Task.Factory.StartNew(() =>

{

Parallel.For(0, files.Count(), (i) =>

{

//模拟耗时任务

Thread.Sleep(1000);

MessageBox.Show(Path. GetFileName(files[i]));

});

});

/\*

Thread newThread = new Thread(() =>

{

Parallel.ForEach(files, (f)=>

{

//模拟耗时任务

Thread.Sleep(1000);

MessageBox.Show(Path. GetFileName(f));

});

});

newThread.Start();

\*/

}

以上的数据并行处理由CRL线程池里面的可支配线程负责，负责的线程执行迭代体里面的完整任务后，如果迭代还没有结束，则该线程可能执行下一个迭代体。其中的调用线程也可以用Thread创建。

* **如何中途终止数据并行？**
* 创建并行取消监控对象CancellationTokenSource

//WF程序

public partial class MainForm : Form

{

//Form级别的线程取消监控对象

private CancellationTokenSource cancelToken =

new CancellationTokenSource();

//……

}

* 创建并行配置对象ParallelOptions，并接收并行取消对象CancellationTokenSource的监控属性。

//创建任务并行配置类的实例对象，并保存cancelToken

ParallelOptions parOpts = new ParallelOptions();

//配置取消监控对象

parOpts.CancellationToken = cancelToken.Token;

//配置线程池允许处理的的最大线程数（=计算机的处理器数）

//parOpts.MaxDegreeOfParallelism = System.Environment.ProcessorCount;

* 将配置对象ParallelOptions作为Parallel.ForEach()或者Parallel.For()的实参。

try

{

Parallel.For(0, files.Count(),parOpts ,(i) =>

{

//确保取消有效

parOpts.CancellationToken.ThrowIfCancellationRequested();

//模拟耗时任务

});

}

catch(OperationCanceledException ex)

{

MessageBox.Show("并行数据处理已取消终止");

};

或

try

{

Parallel.ForEach(files, parOpts, (f)=>

{

//确保取消有效

parOpts.CancellationToken.ThrowIfCancellationRequested();

//模拟耗时任务

});

}

catch(OperationCanceledException ex)

{

MessageBox.Show("并行数据处理已取消终止");

};

每次终止并行时，剩余等待并行数据处理的线程任务将被取消，同时，CRL等待正在执行的线程完成后，会自动抛出相应异常。因此，需要将并行迭代函数体包含在try{}catch{}中。

* 现在，可以在任意时刻终止并行数据处理，如通过取消按钮。

//取消并行

private void btCancelParallel\_Click(object sender, EventArgs e)

{

//取消剩余等待并行数据处理的线程

//正在执行的线程不受到干扰

cancelToken.Cancel();

//创建新CancellationTokenSource对象，解除取消

cancelToken = new CancellationTokenSource();

}

注意，当CancellationTokenSource对象执行一次取消并行的方法Cancel()后，若再次尝试执行并行操作，会发现无法执行且会抛出并行取消异常，这是因为传给并行配置ParallelOptions对象的并行取消项CancellationToken已经被永久设置为true，因此，需要创建新CancellationTokenSource对象，解除取消。

* **如何使用任务并行？**

类Parallel的静态方法Invoke()可以执行任务并行，其定义如下：

private void Parallel.Invoke(ParallelOptions parallelOptions = null, params Action[] actions);

以下是Parallel.lnvoke()的使用方法：

//定义任务

private void parallelAffair()

{

//模拟耗时任务

}

//并行任务组

Action[] actions = new Action[5];

//任务

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

{

actions[i] = parallelAffair;

}

//并行任务

//同样，会阻塞调用线程

Task.Factory.StartNew(() =>

{

//捕获并行取消异常

try

{

Parallel.Invoke(parOpts,actions);

//或者

/\*

Parallel.Invoke(parOpts,

()=>{

},

()=>{

});

\*/

}

catch (OperationCanceledException ex)

{

//异常处理

}

});

并行任务的并行取消编程和数据并行的相同，故不再赘述。

* **如何使用并行LINQ?**

**并行LING查询的编程比较简单：**

//并行LINQ查询

//同样，会阻塞调用线程

Task.Factory.StartNew(() =>

{

//查询结果

var query = new object();

//捕获并行LINQ取消异常

try

{

query = from i in arr.AsParallel().WithCancellation(

cancelToken.Token)

select i;

}

catch (OperationCanceledException ex)

{

//异常处理

}

});

并行LINQ查询只需要在查询对象后面添加. AsParallel()方法即可，同时，如要支持并行LINQ查询取消，只需要WithCancellation()方法中传入并行取消监控对象的Token。同理，并行取消编程和数据并行的相同，故不再赘述。

注意，并行LINQ无法自定义任务体。

* **并行编程有哪些注意点?**
* 并行开始执行时，有延迟停顿现象。
* 调用线程被阻塞，阻塞的是其执行的任务，但调用线程并不是不可用。通过编程发现，调用线程被利用起来执行并行里面的任务。
* 通过Thread创建的线程是新线程，而且是前台线程（可以通过线程的IsBackground属性验证），而由TPL创建的并行任务所使用的线程都是线程池里面的已存在线程，它们都是后台线程。