**Task对象:**

**构造函数：**

Task(Action action) - 方法无参数，无返回值

Task(Action<object> action, object stateObj) - 方法只带一个object参数，无返回值

Task<int>(Func<int> func) - 方法无参数，返回值int

Task<int>(Func<int, object> func, object stateObj) - 方法只带一个object参数，返回值int

Debug.WriteLine(string.Format("Start: thread={0} - Time={1}", Thread.CurrentThread.ManagedThreadId, DateTime.Now));

Task tsk1 = new Task<int>(a => {

int sum = 0;

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

{

Thread.Sleep(600);

Debug.WriteLine("Thread={0} i={1}", Thread.CurrentThread.ManagedThreadId,i);

sum += i;

}

return sum;

}, 10);

tsk1.Start();

Debug.WriteLine(string.Format("End: thread={0} - Time={1}", Thread.CurrentThread.ManagedThreadId, DateTime.Now));

return;

Start: thread=9 - Time=8/26/2016 4:47:54 PM - 线程9

End: thread=9 - Time=8/26/2016 4:47:54 PM - 线程9

Thread=10 i=1 - 线程 10

Thread=10 i=2

Thread=10 i=3

Thread=10 i=4

Thread=10 i=5

Thread=10 i=6

Thread=10 i=7

Thread=10 i=8

Thread=10 i=9

Thread=10 i=10

Debug.WriteLine(string.Format("Start: thread={0} - Time={1}", Thread.CurrentThread.ManagedThreadId, DateTime.Now));

Task tsk1 = new Task<int>(a => {

int sum = 0;

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

{

Thread.Sleep(600);

Debug.WriteLine("Thread={0} i={1}", Thread.CurrentThread.ManagedThreadId,i);

sum += i;

}

return sum;

}, 10);

tsk1.RunSynchronously(); - 同步执行

Debug.WriteLine(string.Format("End: thread={0} - Time={1}", Thread.CurrentThread.ManagedThreadId, DateTime.Now));

return;

Start: thread=9 - Time=8/26/2016 4:50:57 PM

Thread=9 i=1

Thread=9 i=2

Thread=9 i=3

Thread=9 i=4

Thread=9 i=5

Thread=9 i=6

Thread=9 i=7

Thread=9 i=8

Thread=9 i=9

Thread=9 i=10

End: thread=9 - Time=8/26/2016 4:51:03 PM

**Task 的各种属性和方法：  
Start()**

**Wait()**

**RunSynchronously()**

**GetAwaiter() – 用于设置回调函数 call back function**

**ContinueWith() – 继续执行新的任务，也可以用作回调函数**

Debug.WriteLine(string.Format("Start: thread={0} - Time={1}", Thread.CurrentThread.ManagedThreadId, DateTime.Now));

Task<int> tsk1 = new Task<int>(a => {

int sum = 0;

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

{

Thread.Sleep(600);

Debug.WriteLine("Thread={0} i={1}", Thread.CurrentThread.ManagedThreadId, i);

sum += i;

}

return sum;

}, 10);

**GetAwaiter() – 的用法**

var tAwaitor = tsk1.GetAwaiter(); - 设置回调函数

tAwaitor.OnCompleted( ()=> {

Debug.WriteLine(

string.Format("Thead={0} Task({1})={2} has been completed",

Thread.CurrentThread.ManagedThreadId,

tsk1.AsyncState, - 获取传入参数object

tAwaitor.GetResult() - Task<int> 获取返回值 int, Task 获得 void

)

);

});

Debug.WriteLine(string.Format("Thread:Status={0} - isCompleted={1}", tsk1.Status, tsk1.IsCompleted));

tsk1.Start();

Debug.WriteLine(string.Format("Thread:Status={0} - isCompleted={1}", tsk1.Status, tsk1.IsCompleted));

tsk1.Wait();

Debug.WriteLine(string.Format("Thread:Status={0} - isCompleted={1}", tsk1.Status, tsk1.IsCompleted));

int res = await tsk1;

Debug.WriteLine("Result: {0}", res);

Debug.WriteLine(string.Format("Thread:Status={0} - isCompleted={1}", tsk1.Status, tsk1.IsCompleted));

Debug.WriteLine(string.Format("End: thread={0} - Time={1}", Thread.CurrentThread.ManagedThreadId, DateTime.Now));

return;

Start: thread=8 - Time=8/26/2016 5:13:38 PM

Thread:Status=Created - isCompleted=False

Thread:Status=WaitingToRun - isCompleted=False

Thread=9 i=1

Thread=9 i=2

Thread=9 i=3

Thread=9 i=4

Thread=9 i=5

Thread=9 i=6

Thread=9 i=7

Thread=9 i=8

Thread=9 i=9

Thread=9 i=10

Thread:Status=RanToCompletion - isCompleted=True

Result: 55

Thread:Status=RanToCompletion - isCompleted=True

End: thread=8 - Time=8/26/2016 5:13:44 PM

Thead=8 Task(10)=55 has been completed

**使用函数传递：**

Task<int> tsk1 = new Task<int>(taskfunc, 10); 如果想获取值必须 Task<int> tsk1, 不能Task tsk1

var tAwaitor = tsk1.GetAwaiter();

tAwaitor.OnCompleted( () => taskComplete<int>( tAwaitor, tsk1 ) );

private int taskfunc(object a)

{

int sum = 0;

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

{

Thread.Sleep(200);

Debug.WriteLine("Thread={0} i={1}", Thread.CurrentThread.ManagedThreadId, i);

sum += i;

}

return sum;

}

private void taskComplete<T>( TaskAwaiter<T> tskAwaitor, Task<T> tsk)

{

T t = default(T);

t = tskAwaitor.GetResult();

object input = tsk.AsyncState;

Debug.WriteLine(string.Format("CallBack Complete: {0} - {1}", input, t));

}

**无参数， 无返回值：**

Task tsk1 = new Task(taskfunc);

var tAwaitor = tsk1.GetAwaiter();

tAwaitor.OnCompleted( () => taskComplete( tAwaitor, tsk1 ) );

tsk1.Start();

await tsk1;

private void taskfunc()

{

int a = 8;

int sum = 0;

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

{

Thread.Sleep(200);

Debug.WriteLine("Thread={0} i={1}", Thread.CurrentThread.ManagedThreadId, i);

sum += i;

}

}

private void taskComplete( TaskAwaiter tskAwaitor, Task tsk)

{

object t = null;

tskAwaitor.GetResult(); - void, 没有放回值， 此处只能调用而已，没有实际意义

object input = tsk.AsyncState; - 没有参数，则为 null

Debug.WriteLine(string.Format("CallBack Complete: {0} - {1}",

input==null?"null":"none",

t==null?"null":"good"));

}

**ContinueWith() – 会自动多出一个参数，用来传递Task**

Task<int> tsk1 = new Task<int>(a => {

int sum = 0;

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

{

Thread.Sleep(600);

Debug.WriteLine("Thread={0} i={1}", Thread.CurrentThread.ManagedThreadId, i);

sum += i;

}

return sum;

}, 10);

var tAwaitor = tsk1.GetAwaiter();

tAwaitor.OnCompleted(() => {

Debug.WriteLine(

string.Format("Thead={0} Task({1})={2} has been completed",

Thread.CurrentThread.ManagedThreadId,

tsk1.AsyncState,

tAwaitor.GetResult()

)

);

});

tsk1.Start();

await tsk1.ContinueWith((tsk, m) => {

Debug.WriteLine(string.Format("Thread:{0} - Continues:{1} - LastResult: {2}",

Thread.CurrentThread.ManagedThreadId,

m, tsk.Result));

return (int)m;

}, 399);

**注意： tsk.Result 获取值， 如果改成这样就是返回结果值：**

int res0 = await tsk1.ContinueWith((tsk, m) => {

Debug.WriteLine(string.Format("Thread:{0} - Continues:{1} - LastResult: {2}",

Thread.CurrentThread.ManagedThreadId,

m, tsk.Result));

return tsk.Result;

}, 399);

**通过回调函数来返回值，可以在返回值之前，添加自己的代码来完成业务逻辑**

**使用**

int res0 = await tsk1.ContinueWith((tsk, m) => {

Debug.WriteLine(string.Format("Thread:{0} - Continues:{1} - LastResult: {2}",

Thread.CurrentThread.ManagedThreadId,

m, tsk.Result));

return tsk.Result;

}, 399); - 回调函数返回值

int res = await tsk1; - 阻塞并等待异步返回值

tsk1.Result – 阻塞并等待直接返回值，

Debug.WriteLine("Result: {0} - Continue: {1} result: {2}", res, res0, tsk1.Result);

**注意： ContinueWith() 必须是放在线程启动以后， 否则造成死等待：**

Task<int> tsk1 = new Task<int>(a => {

int sum = 0;

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

{

Thread.Sleep(600);

Debug.WriteLine("Thread={0} i={1}", Thread.CurrentThread.ManagedThreadId, i);

sum += i;

}

return sum;

}, 10);

int res0 = await tsk1.ContinueWith((tsk, m) => {

Debug.WriteLine(string.Format("Thread:{0} - Continues:{1} - LastResult: {2}",

Thread.CurrentThread.ManagedThreadId,

m, tsk.Result));

return tsk.Result;

}, 399); - 一直在等待没有启动的线程执行完毕。所以死等待

tsk1.Start();

**为什么要使用 Task**

1. 我们知道一般多线程异步，传入的参数只能有一个参数object
2. 通过 Task 我们很容易实现多参数异步

Task<int> tsk1 = new Task<int>( () => add(100, 299) );

tsk1.Start();

int res = await tsk1;

private int add(int a, int b)

{

Thread.Sleep(2000);

Debug.WriteLine(string.Format("Add({0},{1}) Thread:{2}", a, b, Thread.CurrentThread.ManagedThreadId));

return a + b;

}

**正确的用法：**

private int sum(int a, int b)

{

int rs = 0;

for (int i = a; i <= b; i++)

{

Thread.Sleep(600);

Debug.WriteLine(string.Format("Sum({0},{1}) Thread:{2}", a, b, Thread.CurrentThread.ManagedThreadId));

rs += i;

}

return rs;

}

private async Task<int> sumAsync(int a, int b)

{

Task<int> tsk = Task.Run(() => sum(a, b));

return await tsk;

}

private async Task<int> sumAsync(int a, int b, Action<int>callback)

{

Task<int> tsk = Task.Run(() => sum(a, b));

TaskAwaiter<int> twait = tsk.GetAwaiter();

twait.OnCompleted( () => callback( twait.GetResult() ));

return await tsk;

}

**Main Function**

Debug.WriteLine(string.Format("Start: thread={0} - Time={1}", Thread.CurrentThread.ManagedThreadId, DateTime.Now));

int res = await sumAsync(20, 30, sumDone);

Debug.WriteLine(string.Format("Result: {0}", res));

Debug.WriteLine(string.Format("End: thread={0} - Time={1}", Thread.CurrentThread.ManagedThreadId, DateTime.Now));

**CallBack**

private void sumDone(int total)

{

Debug.WriteLine(string.Format("Thread:{0} - CallBack: {1}",

Thread.CurrentThread.ManagedThreadId,

total

));

}

Start: thread=9 - Time=8/26/2016 6:48:57 PM

Sum(20,30) Thread:10

Sum(20,30) Thread:10

Sum(20,30) Thread:10

Sum(20,30) Thread:10

Sum(20,30) Thread:10

Sum(20,30) Thread:10

Sum(20,30) Thread:10

Sum(20,30) Thread:10

Sum(20,30) Thread:10

Sum(20,30) Thread:10

Sum(20,30) Thread:10

Thread:9 - CallBack: 275

Result: 275

End: thread=9 - Time=8/26/2016 6:49:03 PM

**看看主程序不阻塞的情况：**

Debug.WriteLine(string.Format("Start: thread={0} - Time={1}", Thread.CurrentThread.ManagedThreadId, DateTime.Now));

sumAsync(20, 30, sumDone); - 注意这里没有 await

Debug.WriteLine(string.Format("Result: {0}", 0)); - 不可能有返回值，所以写上 0

Debug.WriteLine(string.Format("End: thread={0} - Time={1}", Thread.CurrentThread.ManagedThreadId, DateTime.Now));

Start: thread=9 - Time=8/26/2016 6:54:39 PM - 主线程9

Result: 0 - 主线程9 并没有等待

End: thread=9 - Time=8/26/2016 6:54:39 PM - 主线程9 并没有等待

Sum(20,30) Thread:10

Sum(20,30) Thread:10

Sum(20,30) Thread:10

Sum(20,30) Thread:10

Sum(20,30) Thread:10

Sum(20,30) Thread:10

Sum(20,30) Thread:10

Sum(20,30) Thread:10

Sum(20,30) Thread:10

Sum(20,30) Thread:10

Sum(20,30) Thread:10

Thread:9 - CallBack: 275 - 最后回调函数被呼叫，返回值

**容易误解的地方：**

private async Task<int> sumAsync(int a, int b)

{

Task<int> tsk = Task.Run(() => sum(a, b));

return tsk;

}

当方法上加了 async 以后不能返回Task<int>， 只需要返回 int

**---------------------------------------------------------------------------------------------**

private async Task<int> sum(int a, int b, Action<int> Callback )

{

Task<int> tsk = Task.Run(() => {

Thread.Sleep(500);

Debug.WriteLine(string.Format("Thread:{0} a={1} b={2}",

Thread.CurrentThread.ManagedThreadId, a, b));

return a + b;

});

//int r = await tsk.ContinueWith<int>(t => { Callback(t.Result); return t.Result; });

return await tsk;

}

Debug.WriteLine(string.Format("Start Thread:{0}", Thread.CurrentThread.ManagedThreadId));

Task<int> res = sum(20, 35, acb);

Debug.WriteLine(string.Format("result = {0}", res.Result));

Debug.WriteLine(string.Format("End Thread:{0}", Thread.CurrentThread.ManagedThreadId));

Start Thread:9

Thread:10 a=20 b=35

-死等待

private Task<int> sum(int a, int b, Action<int> Callback )

{

Task<int> tsk = Task.Run(() => {

Thread.Sleep(500);

Debug.WriteLine(string.Format("Thread:{0} a={1} b={2}",

Thread.CurrentThread.ManagedThreadId, a, b));

return a + b;

});

//int r = await tsk.ContinueWith<int>(t => { Callback(t.Result); return t.Result; });

return tsk;

}

函数里没有async 和 await, 执行同样代码， 则不会造成死等待

Start Thread:9

Thread:10 a=20 b=35

result = 55

End Thread:9

Task.Run()  **和** Task.Factory.StartNew()

**Run –** 不允许有输入的参数， 但是可以有返回值， 即 Action , Func<T>

**StartNew –** 允许有一个object参数, 允许有一个返回值。即 Action, Action<object>, Func<T> Func<object, T>等

**其他：**

Debug.WriteLine(string.Format("Start: thread={0} - Time={1}", Thread.CurrentThread.ManagedThreadId, DateTime.Now));

sumAsync(20, 30);

Debug.WriteLine(string.Format("Result: {0}", 0));

Debug.WriteLine(string.Format("End: thread={0} - Time={1}", Thread.CurrentThread.ManagedThreadId, DateTime.Now));

private void sumAsync(int a, int b)

{

Task tsk = Task.Run(() => sum(a, b));

}

Start: thread=9 - Time=8/26/2016 7:09:30 PM

Result: 0

End: thread=9 - Time=8/26/2016 7:09:30 PM

Sum(20,30) Thread:10

Sum(20,30) Thread:10

Sum(20,30) Thread:10

Sum(20,30) Thread:10

Sum(20,30) Thread:10

Sum(20,30) Thread:10

Sum(20,30) Thread:10

Sum(20,30) Thread:10

Sum(20,30) Thread:10

Sum(20,30) Thread:10

Sum(20,30) Thread:10

private async void sumAsync(int a, int b) - 异步执行，但是有warning

{

Task tsk = Task.Run(() => sum(a, b));

}

private async void sumAsync(int a, int b) - 异步执行，正确

{

Task tsk = Task.Run(() => sum(a, b));

await tsk;

}

private async Task sumAsync(int a, int b) - 出错， 因为有 return

{

Task tsk = Task.Run(() => sum(a, b));

return await tsk;

}

private async Task sumAsync(int a, int b) - 异步执行，正确

{

Task tsk = Task.Run(() => sum(a, b));

await tsk;

}

Async 和 await 是一对搭档

**同步延时和异步延时：**

Thread.Sleep(1000);

Task.Delay(1000);

必须使用 await Task.Delay(1000);

Debug.WriteLine("Start: {0} - {1}", Thread.CurrentThread.ManagedThreadId, DateTime.Now);

Task.Delay(5000);

Debug.WriteLine("After: {0} - {1}", Thread.CurrentThread.ManagedThreadId, DateTime.Now);

Start: 10 - 08/26/2016 16:38:36

After: 10 - 08/26/2016 16:38:36

结果是：并没有等待５秒钟，　因为是异步等待，是在新的线程里等待，当前线程并不需要等待

Debug.WriteLine("Start: {0} - {1}", Thread.CurrentThread.ManagedThreadId, DateTime.Now);

await Task.Delay(5000);

Debug.WriteLine("After: {0} - {1}", Thread.CurrentThread.ManagedThreadId, DateTime.Now);

Start: 9 - 08/26/2016 16:41:54

After: 9 - 08/26/2016 16:41:59

await Task.Delay(5000); - 阻塞当前直到等待其他线程执行完毕，所以等待了5秒钟

**--------------------------------------------------------------------------------------------------------------------------------------**

简单应用：

Task tsk = new Task(taskcall);

tsk.Start();

static void taskcall()

{

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

{

Thread.Sleep(1000);

Console.WriteLine("Task Call: {0} - {1}", i, DateTime.Now);

}

}

如果需要同步运行： tsk.RunSynchronously();

可以等待两秒： tsk.Wait(2000);

一直等到结束： tsk.Wait();

Task tsk = new Task(taskcall);

tsk.Start();

tsk.Wait();

取消Task:

token 取自于 tokenSource

CancellationTokenSource tokenSource = new CancellationTokenSource();

CancellationToken token = tokenSource.Token;

Task tsk = new Task(taskcall, token);

tsk.Start();

tokenSource.CancelAfter(3500); - 3.5 秒以后取消

tokenSource.Cancel(); - 即刻取消

Thread.Sleep(4000);

Console.WriteLine("In Cancel: {0}", tokenSource.IsCancellationRequested);

传递参数给 Task : 支持无参数Action , 只带一个参数 Action<object> 必须是 object

Task tsk = new Task(taskcall, 500, token);

tsk.Start();

Console.WriteLine("Object: {0}", tsk.AsyncState);

static void taskcall(object state)

{

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

{

Thread.Sleep(1000);

Console.WriteLine("Task Call: {0} - {1}", i, state);

}

}

延时：

await Task.Delay(3000); 必须放在async的方法里

Thread.Sleep(3000); 作用于任何的当前 Thread

使用匿名：

相当于 Action<object>

Task tsk = new Task( a=>Console.WriteLine("Task= {0}",a), 500, token);

tsk.Start();

Console.WriteLine("Object: {0}", tsk.AsyncState);

相当于 Action

Task tsk = new Task( ()=>Console.WriteLine("Task=Action"), token);

tsk.Start();

Console.WriteLine("Object: {0}", tsk.AsyncState);

使用 Task.Factory 启动异步

Task.Factory.StartNew(call1, 399);

Task Task.Run( Action ) -

Task Task.Run( Func<Task> ) -

Task<T> Task.Run( Func<T> ) -

Task<T> Task.Run( Func<Task<T>> ) -

public event dele1 onAdd;

public async void callAdd(object b)

{

int result = await Task.Run<int>(() => {

Thread.Sleep(500);

if (onAdd != null) onAdd(88);

return 100;

}); -- 异步执行，线程号： 9

if(onAdd!=null) onAdd(result); -- 回到主线程： 8

}

private void button1\_Click(object sender, EventArgs e)

{

ix uu = new ix();

uu.onAdd += Uu\_onAdd;

uu.callAdd(20);

Debug.WriteLine(string.Format("\nFinished : {0}", Thread.CurrentThread.ManagedThreadId));

* 主线程：8 最后的语句， 最先输出

}

private void Uu\_onAdd(int a)

{

Debug.WriteLine(string.Format("\nResult: {0} - {1}", a, Thread.CurrentThread.ManagedThreadId));

}

Finished : 8

Result: 88 - 9

Result: 100 - 8

public event dele1 onAdd;

public async void callAdd(object b)

{

//Func<Task<int>> myfunc = add;

int result = await Task.Run(add);

if(onAdd!=null) onAdd(result);

}

public async Task<int> add()

{

return await

Task.Factory.StartNew<int>(() => {

int sum = 0;

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

{

Thread.Sleep(500);

sum += i;

if (onAdd != null) onAdd(sum);

}

return sum;

});

}

Finished : 9

Result: 0 - 11

Result: 1 - 11

Result: 3 - 11

Result: 6 - 11

Result: 10 - 11

Result: 15 - 11

Result: 21 - 11

Result: 28 - 11

Result: 36 - 11

Result: 45 - 11

Result: 55 - 11

Result: 55 – 9

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | Task 结合 await async | | |  |  |  | | --- | --- | --- | |  |  |  | |  | | | |  |  |  |  |  |  |  | | --- | |  |   1) await 必须是在 带有 async 的方法里使用， 否则会编译出错  2）在 async 方法的返回值只能是： 只能是 void, Task, Task<T> 三种， 不能是其他  3）在 async 方法里如果没有 await 不出错，会报警， 此时方法是以同步方式执行。  4）Task<T> 只能返回一个值，如果想要返回多个值， 可以考虑使用 Tuple<> 类型。  5）async 方法里有 await，表示此方法从await语句开始是以异步的方式执行。  6）如果我们是使用 await 调用 async 的函数，这表示我要等到函数执行完毕，即同步的意思。  7）另外一个概念 await 某个函数时, 这个函数必须返回是 Task, Task<T> 。  static void Main(string[] args)  {  Task.Factory.StartNew(call1, 399); 异步运行    Console.WriteLine("Thread is starting...");  Task<string> res = callWCF("Motion Metrics MetricsManager"); 此处是异步执行  Console.WriteLine("AsyncState: {0} - {1}", res.AsyncState, res.Result); 注意此处是同步执行；  Console.WriteLine("Console is end!");  Console.ReadLine();  }  static void call1(object ts)  {  for (int i = 0; i < 4; i++)  {  Thread.Sleep(1000);  Console.WriteLine("Call1: {0} - {1} - {2}", i, DateTime.Now, ts);  }  }  static async Task<string> callWCF(string msg)  {  HttpClient htp\_client = new HttpClient();  htp\_client.BaseAddress = new Uri("http://192.168.1.182/MetricsManager/MMService/LogProcessor.svc/");  string send\_content = string.Format("sayHello/msg={0}", msg);  HttpResponseMessage rep = await htp\_client.GetAsync(send\_content);  string ret\_str = await rep.Content.ReadAsStringAsync();  Thread.Sleep(2000);  return ret\_str;  }  这种情况是同步执行：  async 方法里没有 await , 不出错但是会报警， 此时调用方法是同步执行  static void Main(string[] args)  {  Task.Factory.StartNew(call1, 399);    Console.WriteLine("Thread is starting...");  Task<string> res = callWCF("Motion Metrics MetricsManager");  Console.WriteLine("Console is end!");  Console.ReadLine();  }    static async Task<string> callWCF(string msg)  {  HttpClient htp\_client = new HttpClient();  htp\_client.BaseAddress = new Uri("http://192.168.1.182/MetricsManager/MMService/LogProcessor.svc/");  string send\_content = string.Format("sayHello/msg={0}", msg);  HttpResponseMessage rep = htp\_client.GetAsync(send\_content).Result;  string ret\_str = rep.Content.ReadAsStringAsync().Result;  Console.WriteLine("before sleep wcf return: {0}", ret\_str);  Thread.Sleep(2000);  Console.WriteLine("after sleep wcf return: {0}", ret\_str);  return ret\_str;  }  再看这种情况：  static void Main(string[] args)  {  Task.Factory.StartNew(call1, 399);    Console.WriteLine("Thread is starting...");  AsyncCall("Motion Metrics MetricsManager"); 同步执行， 在这里不能使用 await 因为 Main 没有 async, 而且Main做为入口不允许有  Console.WriteLine("Console is end!");  Console.ReadLine();  }  情况一  static async void AsyncCall(string msg)  {  await callWCF(msg); 同步执行，虽然有 await , 但是asnyc函数里没有await  Console.WriteLine("AsyncCall: {0}", msg);  }  static async Task<string> callWCF(string msg)  {  HttpClient htp\_client = new HttpClient();  htp\_client.BaseAddress = new Uri("http://192.168.1.182/MetricsManager/MMService/LogProcessor.svc/");  string send\_content = string.Format("sayHello/msg={0}", msg);  HttpResponseMessage rep = htp\_client.GetAsync(send\_content).Result;  string ret\_str = rep.Content.ReadAsStringAsync().Result;  Console.WriteLine("before sleep wcf return: {0}", ret\_str);  Thread.Sleep(2000);  Console.WriteLine("after sleep wcf return: {0}", ret\_str);  return ret\_str;  }  情况二  static async void AsyncCall(string msg)  {  await callWCF(msg); 同步执行到 await Task.Delay(1000);  Console.WriteLine("AsyncCall: {0}", msg);  }  static async Task<string> callWCF(string msg)  {  HttpClient htp\_client = new HttpClient();  htp\_client.BaseAddress = new Uri("http://192.168.1.182/MetricsManager/MMService/LogProcessor.svc/");  string send\_content = string.Format("sayHello/msg={0}", msg);  HttpResponseMessage rep = htp\_client.GetAsync(send\_content).Result;  Console.WriteLine("before sleep wcf return: {0}", rep);  await Task.Delay(1000); - 从此处开始是以异步方式执行。以上都是同步方式执行。  string ret\_str = rep.Content.ReadAsStringAsync().Result;  //Thread.Sleep(2000);  Console.WriteLine("after sleep wcf return: {0}", ret\_str);  return ret\_str;  }  await 某个函数时, 这个函数必须返回是 Task, Task<T> 。  static void Main(string[] args)  {  Task.Factory.StartNew(call1, 399);    Console.WriteLine("Thread is starting...");  AsyncCall("Motion Metrics MetricsManager"); 同步执行  Console.WriteLine("Console is end!");  Console.ReadLine();  }  情况一  static Task call2(string ts)  {  for (int i = 0; i < 4; i++)  {  //await Task.Delay(1000);  Thread.Sleep(500);  Console.WriteLine("Call2: {0} - {1} - {2}", i, DateTime.Now, ts);  }  return new Task(() => Console.WriteLine("call2 return")); - 此Console.WriteLine 没有被执行  }  static async void AsyncCall(string msg)  {  await call2(msg); 同步执行  Console.WriteLine("AsyncCall: {0}", msg); - 这一句也没有被执行？？  }  情况二  static Task call2(string ts)  {  for (int i = 0; i < 4; i++)  {  //await Task.Delay(1000);  Thread.Sleep(500);  Console.WriteLine("Call2: {0} - {1} - {2}", i, DateTime.Now, ts);  }  return new Task(() => Console.WriteLine("call2 return")); 同步被执行  }  static async void AsyncCall(string msg)  {  Task tt = call2(msg);  tt.RunSynchronously();  Console.WriteLine("AsyncCall: {0}", msg); 同步被执行  }  情况三  static async Task call2(string ts)  {  for (int i = 0; i < 4; i++)  {  await Task.Delay(1000); -- 这样是异步运行  //Thread.Sleep(500); -- 这样是同步运行  Console.WriteLine("Call2: {0} - {1} - {2}", i, DateTime.Now, ts);  }  //return new Task(() => Console.WriteLine("call2 return")); - 不能有返回值  }  static async void AsyncCall(string msg)  {  Task tt = call2(msg); 这样是异步, 是否是异步取决运函数体里是否有 await  //tt.RunSynchronously(); - 不可以这样运行， 说没有函数体delegate  Console.WriteLine("AsyncCall: {0}", msg);  }  情况四  static async void AsyncCall(string msg)  {  await call2(msg); 这样还是取决于运函数体里是否有 await  //tt.RunSynchronously();  Console.WriteLine("AsyncCall: {0}", msg);  }  情况五  static async Task call2(string ts)  {  for (int i = 0; i < 4; i++)  {  await Task.Delay(1000);  // Thread.Sleep(500);  Console.WriteLine("Call2: {0} - {1} - {2}", i, DateTime.Now, ts);  }  //return new Task(() => Console.WriteLine("call2 return"));  }  static void AsyncCall(string msg)  {  call2(msg); 是异步与否取决运函数体里是否有 await， await call2(msg) 不可以，因为没有async  //tt.RunSynchronously();  Console.WriteLine("AsyncCall: {0}", msg);  }  await 的函数 async Task<TResult> 的情况：  注意返回时，不是 return Task<string> 而是 return string. 这和普通函数不一样。  static async Task<string> call2(string a, int b, int c)  {  string str = string.Empty;  for (int i = 0; i < 4; i++)  {  await Task.Delay(5000); - 到这里开始， 以异步方式执行， 开辟第二线程处理以下的语句和返回值。  //Thread.Sleep(5000);  str = string.Format("Task<string>: {0} - {1} - {2}", a, b, c);  Console.WriteLine("Async Task: {0} - {1} - {2} - {3}", i, a, b, c);  }  return str;  }    static async void AsyncCall(string msg)  {  for (int i = 0; i < 3; i++)  {  Task<string> result = call2(msg, 300, 200); - 这里也是同步执行。  Console.WriteLine("AsyncCall End: {0} - {1}", i, result);  }  }  static void Main(string[] args)  {  Task.Factory.StartNew(call1, 399);    Console.WriteLine("Thread is starting...");  AsyncCall("Motion Metrics MetricsManager"); - 对于主线程来说， 是同步执行。  Console.WriteLine("Console is end!");  Console.ReadLine();  }    情况二：  static async void AsyncCall(string msg)  {  for (int i = 0; i < 3; i++)  {  await Task.Delay(1); - 从这里开始开辟第二线程， 以异步方式执行  //Thread.Sleep(2000);  Task<string> result = call2(msg, 300, 200);  Console.WriteLine("AsyncCall End: {0} - {1}", i, result);  }  }    情况三：  static async Task<string> call2(string a, int b, int c)  {  string str = string.Empty;  for (int i = 0; i < 4; i++)  {  //await Task.Delay(2000);  Thread.Sleep(2000);  str = string.Format("Return Task<string> Call2: {0} - {1} - {2}", a, b, c);  Console.WriteLine("Func Call2: {0} - {1} - {2} - {3}", i, a, b, c);  }  return str;  }  static async void AsyncCall(string msg)  {  for (int i = 0; i < 3; i++)  {  await Task.Delay(1); - 此处开始以异步方式执行（相对于主线程来说）， 开辟第二线程处理， 此处并不会开辟三个独立的线程来处理  //Thread.Sleep(2000);  Task<string> result = call2(msg, 300, 200);  Console.WriteLine("AsyncCall End: {0} - {1}", i, result);  }  }    情况四：  static async Task<string> call2(string a, int b, int c)  {  string str = string.Empty;  for (int i = 0; i < 4; i++)  {  //await Task.Delay(2000);  Thread.Sleep(2000);  str = string.Format("Return Task<string> Call2: {0} - {1} - {2}", a, b, c);  Console.WriteLine("Func Call2: {0} - {1} - {2} - {3}", i, a, b, c);  }  return str;  }  static async void AsyncCall(string msg)  {  for (int i = 0; i < 3; i++)  {  //await Task.Delay(1);  //Thread.Sleep(2000);  string result = await call2(msg, 300, 200); - 由于方法没有 await; 此时 async 和 await 的结合,这里是同步执行。  Console.WriteLine("AsyncCall End: {0} - {1}", i, result);  }  }    情况五：  static async Task<string> call2(string a, int b, int c)  {  string str = string.Empty;  for (int i = 0; i < 4; i++)  {  await Task.Delay(2000); - 异步执行从这里开始  //Thread.Sleep(2000);  str = string.Format("Return Task<string> Call2: {0} - {1} - {2}", a, b, c);  Console.WriteLine("Func Call2: {0} - {1} - {2} - {3}", i, a, b, c);  }  return str;  }  static async void AsyncCall(string msg)  {  for (int i = 0; i < 3; i++)  {  //await Task.Delay(1);  //Thread.Sleep(2000);  string result = await call2(msg, 300, 200); - 等待异步执行完成，返回结果  Console.WriteLine("AsyncCall End: {0} - {1}", i, result);  }  }    情况六：  static async Task<string> call2(string a, int b, int c)  {  string str = string.Empty;  for (int i = 0; i < 4; i++)  {  await Task.Delay(2000); - 这里开始异步，  //Thread.Sleep(2000);  str = string.Format("Return Task<string> Call2: {0} - {1} - {2}", a, b, c);  Console.WriteLine("Func Call2: {0} - {1} - {2} - {3}", i, a, b, c);  }  return str;  }  static async void AsyncCall(string msg)  {  for (int i = 0; i < 3; i++)  {  //await Task.Delay(1);  //Thread.Sleep(2000);  Task<string> result = call2(msg, 300, 200); - 这里同步，  string str = await result; - 取值  string str = result.Result; - 取值    string result = await call2(msg, 300, 200); - 等待异步执行完成， 直接取值  Console.WriteLine("AsyncCall End: {0} - {1}", i, result); 同步执行到这里  }  } | | Async 和 await， Task<> , Task<string> HttpClient 的结合使用：  public class person  {  public int age;  public string name;  }  static async Task<string> callWCF(HttpClient htp\_client, person msg)  {  string send\_content = string.Format("sayHello/msg={0}", string.Format("age:{0}, name:{1}", msg.age, msg.name));  Console.WriteLine("Before Get: {0}", "Before");  HttpResponseMessage rep = await htp\_client.GetAsync(send\_content); - 从这里开始异步执行  Console.WriteLine("wcf GetAsync: {0}", rep);  string ret\_str = await rep.Content.ReadAsStringAsync();  Console.WriteLine("Return ReadString: {0}", ret\_str);    msg.name += "--Good Man"; - class object 是引用传递，即传引用非传值。 所以修改将改变全局。    return ret\_str;  }  static async void AsyncCall(HttpClient htp\_client, person msg)  {  string ret\_str = await callWCF(htp\_client, msg); - 等待异步执行完成， 异步的启动不在这里。 而在函数体里遇到第一个 await 开始异步  Console.WriteLine("WCF Return:{0} - {1}", ret\_str, msg.name); - 同步等待上面语句执行完毕  }  static void Main(string[] args)  {  HttpClient htp\_client = new HttpClient();  htp\_client.BaseAddress = new Uri("http://192.168.1.187/MetricsManager/MMService/LogProcessor.svc/");  htp\_client.Timeout = TimeSpan.FromSeconds(3);  // 在主程序创建对象， 可以传递给异步方法， 由此可以共享此对象。    Task.Factory.StartNew(call1, 399);    Console.WriteLine("Thread is starting...");  HttpClient htp\_client = new HttpClient();  htp\_client.BaseAddress = new Uri("http://192.168.1.187/MetricsManager/MMService/LogProcessor.svc/");  htp\_client.Timeout = TimeSpan.FromSeconds(3);  person son = new person() { age = 36, name = "Willson" };  // 在主程序创建对象， 可以传递给异步方法， 由此可以共享此对象。 取决于业务逻辑是否需要共享对象。  AsyncCall(htp\_client, son);  Console.WriteLine("Console is end!");  Console.ReadLine();  }      public class MMStream\_Service : MMStream\_Service\_Contract  {  public MMStream\_Service()  { }  public string sayHello(string str)  {  WebOperationContext ctx = WebOperationContext.Current;  ctx.OutgoingResponse.StatusCode = System.Net.HttpStatusCode.Conflict; 人为返回 conflict  return string.Format("From Server : {0}", str);  }  }  static async Task<string> callWCF(HttpClient htp\_client, person msg)  {  string ret\_str = string.Empty;  try  {  string send\_content = string.Format("sayHello/msg={0}", string.Format("age:{0}, name:{1}", msg.age, msg.name));  Console.WriteLine("Before Get: {0}", "Before");  HttpResponseMessage rep = await htp\_client.GetAsync(send\_content);  if (rep.IsSuccessStatusCode || rep.StatusCode == HttpStatusCode.Conflict) - 增加判断条件，确保返回值的正确性  {  Console.WriteLine("wcf GetAsync: {0}", rep);  ret\_str = await rep.Content.ReadAsStringAsync();  Console.WriteLine("Return ReadString: {0}", ret\_str);  Console.WriteLine("Response Error: {0}", rep.StatusCode);  }  else  {  Console.WriteLine("Response Error: {0}", rep.StatusCode);  }  msg.name += "--Good Man";  }  catch (Exception err)  {  ret\_str = string.Format("Error:{0}-{1}", err.Message, err.StackTrace);  }  return ret\_str;  }  对于普通的函数， 我们想使用异步：  public class person  {  public int age;  public string name;  public person() { }  public async Task<int> sum(int a)  {  await Task.Delay(1); - 只需要在函数开始时， 执行一个非常短时间的 await Task.Delay(1) , 从这里开始即开创第二个线程执行以下语句  int sm = 0;  for (int i = 0; i <= a; i++)  {  Thread.Sleep(100);  sm += i;  Console.WriteLine("processing: {0} - {1}", i, sm);  }  return sm;  }  public async void call(int a) - 我们需要此函数来充当 callback 函数。在此执行必要的后续操作。  {  int sss = await sum(a);  Console.WriteLine("sum:{0} - {1}", a, sss);  }  }  static void Main(string[] args)  {  Task.Factory.StartNew(call1, 399);    Console.WriteLine("Thread is starting...");  person son = new person() { age = 36, name = "Willson" };  son.call(10);  Console.WriteLine("Console is end!");  Console.ReadLine();  }  也可以如此异步执行，如果不需要返回结果：  static void Main(string[] args)  {  person son = new person() { age = 36, name = "Willson" };  Task<int> sss = son.sum(10); 异步执行  需要返回结果：  Console.WriteLine("sss:{0}", sss.Result); - 此时堵塞等待异步执行结束， 并返回结果。  }  总结 await , async, Task, Task<TResult> 使用异步：  1）如果不需要 callback 来处理异步执行完成的结果： 可以使用一对 await & async void, Task来完成异步：  public class person  {  public int age;  public string name;  public person() { }  public async void sum(int a)  {  await Task.Delay(500); - 异步从这里开始，开创新的线程  int sm = 0;  for (int i = 0; i <= a; i++)  {  Thread.Sleep(500);  sm += i;  Console.WriteLine("processing: {0} - {1}", i, sm);  }  }  public void call(int a)  {  sum(a);  Console.WriteLine("Call:{0}", a);  }  }    static void Main(string[] args)  {  Task.Factory.StartNew(call1, 399);    Console.WriteLine("Thread is starting...");  person son = new person() { age = 36, name = "Willson" };  son.sum(10);  son.call(10);  一样的效果  Console.WriteLine("Console is end!");  Console.ReadLine();  }  同上是一样的效果： 方法 async Task  public class person  {  public int age;  public string name;  public person() { }  public async Task sum(int a)  {  Console.WriteLine("sum start: {0} ", a);  await Task.Delay(500); - 这里开始异步执行  int sm = 0;  for (int i = 0; i <= a; i++)  {  Thread.Sleep(500);  sm += i;  Console.WriteLine("processing: {0} - {1}", i, sm);  }  }  public async void call(int a)  {  await sum(a); - 等待异步执行完成  Console.WriteLine("Person Call:{0}", a);  }  }  person son = new person() { age = 36, name = "Willson" };  son.sum(10); - 异步执行  son.call(10); - 异步执行  Task tt = son.sum(10);  tt.Wait(2000); - 阻塞2秒。  tt.Wait(); - 阻塞直到完成， 相当于同步  public async void call(int a)  {  sum(a); - 异步执行，不等待，直接执行  Console.WriteLine("Person Call:{0}", a);  }  son.call(10); - 异步执行 | | public async void call(int a)  {  await sum(a); - 异步执行，阻塞等待异步完成  Console.WriteLine("Person Call:{0}", a);  }  son.call(10); - 异步执行 | | 2) 如果需要 callback 来处理异步执行完成的结果： 可以使用一对 await & async void, Task<TResult> 来完成异步：  public class person  {  public int age;  public string name;  public person() { }  public async Task<int> sum(int a)  {  Console.WriteLine("sum start: {0} ", a);  await Task.Delay(500); - 异步从这里开始  int sm = 0;  for (int i = 0; i <= a; i++)  {  Thread.Sleep(500);  sm += i;  Console.WriteLine("processing: {0} - {1}", i, sm);  }  return sm; //string.Format("sum return: {0}", sm);  }  public async void call(int a) - 相当于异步回调函数  {  int total = await sum(a); 等待异步执行完成，并返回结果  Console.WriteLine("Person Call:{0}", total);  }  }  person son = new person() { age = 36, name = "Willson" };  son.call(10); |   快速异步执行： |
| Task sss = Task.Run(new Action(call3));  static void call3()  {  Thread.Sleep(2000);  Console.WriteLine("Call3 is here");  }   |  | | --- | | Task<int> sss1 = Task.Run<int>(() => { Thread.Sleep(2000); Console.WriteLine("Task Run"); return 100; });  Console.WriteLine("Sum: {0} - {1}",10, sss.Result);  Task<int> sss = Task.Run<int>(new Func<int>(call3) );  static int call3()  {  return 1000;  }  AsyncCall() 异步方法  static async void AsyncCall()  {  await Task.Run<int>(() => { Thread.Sleep(2000); Console.WriteLine("Task Run"); return 100; });  }  AsyncCall() - 异步方法，返回数值  static async Task<int> call3()  {  Thread.Sleep(2000);  Console.WriteLine("Call3 is here");  return 1000;  }  static async void AsyncCall()  {  int total = await Task.Run<int>(new Func<Task<int>>(call3));  Console.WriteLine("Total: {0}", total);  } | | |  | | --- | | 由于Framework 4.0和Framework 4.5对Task类稍微有些不同，此处声明以下代码都是基于Framework 4.5  Task类和Task<TResult>类，后者是前者的泛型版本。TResult类型为Task所调用方法的返回值。  主要区别在于Task构造函数接受的参数是Action委托，而Task<TResult>接受的是Func<TResult>委托。   1. Task(Action) 2. Task<TResult>(Func<TResult>)   **启动一个任务**   1. static void Main(string[] args) 2. { 3. Task Task1 = new Task(() => Console.WriteLine("Task1")); 4. Task1.Start(); 5. Console.ReadLine(); 6. }   通过实例化一个Task对象，然后Start，这种方式中规中矩，但是实践中，通常采用更方便快捷的方式  Task.Run(() => Console.WriteLine("Foo"));  这种方式直接运行了Task，不像上面的方法还需要调用Start();  Task.Run方法是Task类中的静态方法，接受的参数是委托。返回值是为该Task对象。  Task.Run(Action)  Task.Run<TResult>(Func<Task<TResult>>)  Task构造方法还有一个重载函数如下：  Task 构造函数 (Action, TaskCreationOptions)，对应的Task泛型版本也有类似构造函数。TaskCreationOptions参数指示Task创建和执行的可选行为。常用的参数LongRunning。  默认情况下，Task任务是由线程池线程异步执行的。如果是运行时间很长的操作，使用LongRunning 参数暗示任务调度器，将这个任务放在非线程池上运行。通常不需要用这个参数，除非通过性能测试觉得使用该参数能有更好的性能，才使用。  **任务等待**  默认情况下，Task任务是由线程池线程异步执行。要知道Task任务的是否完成，可以通过task.IsCompleted属性获得，也可以使用task.Wait来等待Task完成。Wait会阻塞当前线程。   1. static void Main(string[] args) 2. { 3. Task Task1=Task.Run(() => { Thread.Sleep(5000); 4. Console.WriteLine("Foo"); 5. Thread.Sleep(5000); 6. }); 7. Console.WriteLine(Task1.IsCompleted); 8. Task1.Wait();//阻塞当前线程 9. Console.WriteLine(Task1.IsCompleted); 10. }   Wait方法有个重构方法，签名如下：public bool Wait(int millisecondsTimeout)，接受一个时间。如果在设定时间内完成就返回true，否则返回false。如下的代码：   1. static void Main(string[] args) 2. { 3. Task Task1=Task.Run(() => { Thread.Sleep(5000); 4. Console.WriteLine("Foo"); 5. Thread.Sleep(5000); 6. }); 8. Console.WriteLine("Task1.IsCompleted:{0}",Task1.IsCompleted); 9. bool b=Task1.Wait(2000); 10. Console.WriteLine("Task1.IsCompleted:{0}", Task1.IsCompleted); 11. Console.WriteLine(b); 12. Thread.Sleep(9000); 13. Console.WriteLine("Task1.IsCompleted:{0}", Task1.IsCompleted); 14. }   运行结果为：  **获得返回值**  要获得返回值，就要用到Task的泛型版本了。   1. static void Main(string[] args) 2. { 3. Task<int> Task1 = Task.Run<int>(() => { Thread.Sleep(5000); return Enumerable.Range(1, 100).Sum(); }); 4. Console.WriteLine("Task1.IsCompleted:{0}",Task1.IsCompleted); 5. Console.WriteLine("Task1.IsCompleted:{0}", Task1.Result);//如果方法未完成，则会等待直到计算完成，得到返回值才运行下去。 6. Console.WriteLine("Task1.IsCompleted:{0}", Task1.IsCompleted); 7. }   结果如下：  **异常抛出**  和线程不同，Task中抛出的异常可以捕获，但是也不是直接捕获，而是由调用Wait()方法或者访问Result属性的时候，由他们获得异常，将这个异常包装成AggregateException类型，再重新抛出捕获。   1. static void Main(string[] args) 2. { 3. try 4. { 5. Task<int> Task1 = Task.Run<int>(() => { throw new Exception("xxxxxx"); return 1; }); 6. Task1.Wait(); 7. } 8. catch (Exception ex)//error的类型为System.AggregateException 9. { 10. Console.WriteLine(ex.StackTrace); 11. Console.WriteLine("-----------------"); 12. Console.WriteLine(ex.InnerException.StackTrace); 13. } 14. }   如上代码，运行结果如下：  可以看到异常真正发生的地方。  对于某些匿名的Task（通过 Task.Run方法生成的，不调用wait，也不关心是否运行完成），某些情况下，记录它们的异常错误也是有必要的。这些异常称作未观察到的异常（unobserved exceptions）。可以通过订阅一个全局的静态事件TaskScheduler.UnobservedTaskException来处理这些异常。只要当一个Task有异常，并且在被垃圾回收的时候，才会触发这一个事件。如果Task还处于被引用状态，或者只要GC不回收这个Task，这个UnobservedTaskException事件就不会被触发  例子：   1. static void Main(string[] args) 2. { 3. TaskScheduler.UnobservedTaskException += UnobservedTaskException; 4. Task.Run<int>(() => { throw new Exception("xxxxxx"); return 1; }); 5. Thread.Sleep(1000); 6. } 8. static void UnobservedTaskException(object sender, UnobservedTaskExceptionEventArgs e) 9. { 10. Console.WriteLine(e.Exception.Message); 11. Console.WriteLine(e.Exception.InnerException.Message); 12. }   这样的代码直到程序运行完成也为未能触发UnobservedTaskException，因为GC没有开始做垃圾回收。  在代码中加入 GC.Collect();   1. static void Main(string[] args) 2. { 3. TaskScheduler.UnobservedTaskException += UnobservedTaskException; 4. Task.Run<int>(() => { throw new Exception("xxxxxx"); return 1; }); 5. Thread.Sleep(1000); 6. GC.Collect(); 7. GC.WaitForPendingFinalizers(); 8. }   运行后得到如下：  **延续任务**  延续任务就是说当一个Task完成后，继续运行下一个任务。通常有2种方法实现。  一种是使用GetAwaiter方法。GetAwaiter方法返回一个TaskAwaiter结构，该结构有一个OnCompleted事件，只需对OnCompleted事件赋值，即可在完成后调用该事件。   1. static void Main(string[] args) 2. { 3. Task<int> Task1 = Task.Run<int>(() => { return Enumerable.Range(1, 100).Sum(); }); 4. var awaiter = Task1.GetAwaiter(); 5. awaiter.OnCompleted(() => 6. { 7. Console.WriteLine("Task1 finished"); 8. int result = awaiter.GetResult(); 9. Console.WriteLine(result); // Writes result 10. }); 11. Thread.Sleep(1000); 12. }   运行结果如下：  此处调用GetResult()的好处在于，一旦先前的Task有异常，就会抛出该异常。而且该异常和之前演示的异常不同，它不需要经过AggregateException再包装了。  另一种延续任务的方法是调用ContinueWith方法。 ContinueWith返回的任然是一个Task类型。ContinueWith方法有很多重载，算上泛型版本，差不多40个左右的。其中最常用的，就 是接受一个Action或者Func委托，而且，这些委托的第一个传入参数都是Task类型，即可以访问先前的Task对象。示例：   1. static void Main(string[] args) 2. { 3. Task<int> Task1 = Task.Run<int>(() => {return Enumerable.Range(1, 100).Sum(); }); 4. Task1.ContinueWith(antecedent => { 5. Console.WriteLine(antecedent.Result); 6. Console.WriteLine("Runing Continue Task"); 7. }); 8. Thread.Sleep(1000); 9. }   使用这种ContinueWith方法和GetAwaiter都能实现相同的效果，有点小区别就是ContinueWith如果获取Result的时候有异常，抛出的异常类型是经过AggregateException包裹的，而GetAwaiter()后的OnCompleted所调用的方法中，如果出错，直接抛出异常。  **生成Task的另一种方法，TaskCompletionSource**  使用TaskCompletionSource很简单，只需要实例化它即可。TaskCompletionSource有一个Task属性，你可以 对该属性暴露的task做操作，比如让它wait或者ContinueWith等操作。当然，这个task由TaskCompletionSource完 全控制。TaskCompletionSource<TResult>类中有一些成员方法如下：   1. public class TaskCompletionSource<TResult> 2. { 3. public void SetResult (TResult result); 4. public void SetException (Exception exception); 5. public void SetCanceled(); 6. public bool TrySetResult (TResult result); 7. public bool TrySetException (Exception exception); 8. public bool TrySetCanceled(); 9. ... 10. }   调用以上方法意味着对Task做状态的改变，将状态设成completed，faulted或者 canceled。这些方法只能调用一次，不然会有异常。Try的方法可以调多次，只不过返回false而已。  通过一些技巧性的编码，将线程和Task协调起来，通过Task获得线程运行的结果。  示例代码：   1. static void Main(string[] args) 2. { 3. var tcs = new TaskCompletionSource<int>(); 4. new Thread(() => { 5. Thread.Sleep(5000); 6. int i = Enumerable.Range(1, 100).Sum(); 7. tcs.SetResult(i); }).Start();//线程把运行计算结果，设为tcs的Result。 8. Task<int> task = tcs.Task; 9. Console.WriteLine(task.Result); //此处会阻塞，直到匿名线程调用tcs.SetResult(i)完毕 10. }   说明一下以上代码：  tcs是TaskCompletionSource<int>的一个实例，即这个Task返回的肯定是一个int类型。  获得tcs的Task属性，读取并打印该属性的值。那么 Console.WriteLine(task.Result);其实是会阻塞的，直到task的result被赋值之后，才会取消阻塞。而对task.result的赋值正在一个匿名线程中做的。也就是说，一直等到匿名线程运行结束，把运行结果赋值给tcs后，task.Result的值才会被获得。这正是变相的实现了线程同步的功能，并且可以获得线程的运行值。而此时的线程并不是运行在线程池上的。  我们可以定义一个泛型方法，来实现一个Task对象，并且运行Task的线程不是线程池线程：   1. Task<TResult> Run<TResult>(Func<TResult> function) 2. { 3. var tcs = new TaskCompletionSource<TResult>(); 4. Thread t = new Thread(() => 5. { 6. try { tcs.SetResult(function()); } 7. catch (Exception ex) { tcs.SetException(ex); } 8. }); 9. t.IsBackground = true; 10. t.Start();//启动线程 11. return tcs.Task; 12. }   比如什么一个泛型方法，接受的参数是Func委托，返回的是Task类型。  该方法中启动一个线程t，把t设为后台线程，该线程运行的内容就是传入的Func委托，并将Func委托的运行后的返回值通过 tcs.SetResult赋给某个task。同时，如果有异常的话，就把异常赋给，某个task，然后将这个task返回。这样，直到线程运行完毕，才 能得到task.Result的值。调用的时候：   1. Task<int> task = Run(() => { Thread.Sleep(5000); return Enumerable.Range(1, 100).Sum(); }); 2. Console.Write(task.Result);//这句会阻塞当前线程，直到task的result值被赋值才行。   TaskCompletionSource的另一个强大用处，是可以创建Task，而不绑定任何线程，比如，我们可以通过TaskCompletionSource实现对某一个方法的延迟调用。  代码示例：   1. static Task<int> delayFunc() 2. { 3. var tcs = new TaskCompletionSource<int>(); 4. var timer = new System.Timers.Timer(5000) { AutoReset = false }; 5. timer.Elapsed += (sender, e) => { 6. timer.Dispose(); 7. int i = Enumerable.Range(1, 100).Sum(); 8. tcs.SetResult(i); 9. }; 11. timer.Start(); 12. return tcs.Task; 13. }   说明：  delayFunc()方法使用了一个定时器，5秒后，定时器事件触发，将i的值赋给某个task的result。返回的是tcs.Task属性，调用方式:   1. var task = delayFunc(); 2. Console.Write(task.Result);   task变量得到赋值后，要读取Result值，必须等到tcs.SetResult(i);运行完成才行。这就相当于实现了延迟某个方法。  当然Task自身提供了Delay方法，使用方法如下：   1. Task.Delay (5000).GetAwaiter().OnCompleted (() => Console.WriteLine (42)); 2. 或者: 3. Task.Delay (5000).ContinueWith (ant => Console.WriteLine (42));   Delay方法是相当于异步的Thread.Sleep(); | | |  | |

[**C#异步编程之：（一）Task对象和lamda表达式探究**](http://blog.csdn.net/puncha/article/details/8316427)

基于TPL的异步编程，最简单的就是使用Task对象，而创建一个Task对象，最简单的就是使用lamda表达式：

1. static void Main(string[] args) {
3. // create the cancellation token source
4. CancellationTokenSource tokenSource = new CancellationTokenSource();
5. // create the cancellation token
6. CancellationToken token = tokenSource.Token;
7. int i = 0;
8. int value = 10;
10. // create the first task, which we will let run fully
11. Task task1 = new Task(() => {
12. for (i = 0; i < Int32.MaxValue; i++)
13. {
14. value++;
15. bool cancelled = token.WaitHandle.WaitOne(10000);
17. if (cancelled) {
18. throw new OperationCanceledException(token);
19. }
20. }
21. }, token);
23. // start task
24. task1.Start();
26. // cancel the token
27. tokenSource.Cancel();
29. // wait for input before exiting
30. Console.WriteLine("Main method complete. Press enter to finish with value = " + value.ToString());
31. Console.ReadLine();
32. }

# [说说C#的async和await](http://blog.csdn.net/tianmuxia/article/details/17675681)

2013-12-30 10:36 67113人阅读 [评论](http://blog.csdn.net/tianmuxia/article/details/17675681/#comments)(18) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/tianmuxia/article/details/17675681/#report)

http://static.blog.csdn.net/images/category_icon.jpg分类：

编程技术（6） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg

版权声明：本文为博主原创文章，未经博主允许不得转载。

C# 5.0中引入了async 和 await。这两个关键字可以让你更方便的写出异步代码。

看个例子：

**[csharp]** [view plain](http://blog.csdn.net/tianmuxia/article/details/17675681/) [copy](http://blog.csdn.net/tianmuxia/article/details/17675681/)

[在CODE上查看代码片](https://code.csdn.net/snippets/134105)

1. public class MyClass
2. {
3. public MyClass()
4. {
5. DisplayValue(); //这里不会阻塞
6. System.Diagnostics.Debug.WriteLine("MyClass() End.");
7. }
8. public Task<double> GetValueAsync(double num1, double num2)
9. {
10. return Task.Run(() =>
11. {
12. for (int i = 0; i < 1000000; i++)
13. {
14. num1 = num1 / num2;
15. }
16. return num1;
17. });
18. }
19. public async void DisplayValue()
20. {
21. double result = await GetValueAsync(1234.5, 1.01);//此处会开新线程处理GetValueAsync任务，然后方法马上返回
22. //这之后的所有代码都会被封装成委托，在GetValueAsync任务完成时调用
23. System.Diagnostics.Debug.WriteLine("Value is : " + result);
24. }
25. }

上面在MyClass的构造函数里调用了async关键字标记的异步方法DisplayValue()，DisplayValue()方法里执行了 一个await关键字标记的异步任务GetValueAsync()，这个异步任务必须是以Task或者Task<TResult>作为返回 值的，而我们也看到，异步任务执行完成时实际返回的类型是void或者TResult，DisplayValue()方法里 await GetValueAsync()之后的所有代码都会在异步任务完成时才会执行。

DisplayValue()方法实际执行的代码如下：

**[csharp]** [view plain](http://blog.csdn.net/tianmuxia/article/details/17675681/) [copy](http://blog.csdn.net/tianmuxia/article/details/17675681/)

[在CODE上查看代码片](https://code.csdn.net/snippets/134105)

1. public void DisplayValue()
2. {
3. System.Runtime.CompilerServices.TaskAwaiter<double> awaiter = GetValueAsync(1234.5, 1.01).GetAwaiter();
4. awaiter.OnCompleted(() =>
5. {
6. double result = awaiter.GetResult();
7. System.Diagnostics.Debug.WriteLine("Value is : " + result);
8. });
9. }

可以看到，async和await关键字只是把上面的代码变得更简单易懂而已。

程序的输出如下：

MyClass() End.

Value is : 2.47032822920623E-322

以下是我写的一个静态类，可以方便将一个普通Function执行异步调用：

**[csharp]** [view plain](http://blog.csdn.net/tianmuxia/article/details/17675681/) [copy](http://blog.csdn.net/tianmuxia/article/details/17675681/)

[在CODE上查看代码片](https://code.csdn.net/snippets/134105)

1. public static class TaskAsyncHelper
2. {
3. /// <summary>
4. /// 将一个方法function异步运行，在执行完毕时执行回调callback
5. /// </summary>
6. /// <param name="function">异步方法，该方法没有参数，返回类型必须是void</param>
7. /// <param name="callback">异步方法执行完毕时执行的回调方法，该方法没有参数，返回类型必须是void</param>
8. public static async void RunAsync(Action function, Action callback)
9. {
10. Func<System.Threading.Tasks.Task> taskFunc = () =>
11. {
12. return System.Threading.Tasks.Task.Run(() =>
13. {
14. function();
15. });
16. };
17. await taskFunc();
18. if (callback != null)
19. callback();
20. }
22. /// <summary>
23. /// 将一个方法function异步运行，在执行完毕时执行回调callback
24. /// </summary>
25. /// <typeparam name="TResult">异步方法的返回类型</typeparam>
26. /// <param name="function">异步方法，该方法没有参数，返回类型必须是TResult</param>
27. /// <param name="callback">异步方法执行完毕时执行的回调方法，该方法参数为TResult，返回类型必须是void</param>
28. public static async void RunAsync<TResult>(Func<TResult> function, Action<TResult> callback)
29. {
30. Func<System.Threading.Tasks.Task<TResult>> taskFunc = ()=>
31. {
32. return System.Threading.Tasks.Task.Run(()=>
33. {
34. return function();
35. });
36. };
37. TResult rlt = await taskFunc();
38. if(callback != null)
39. callback(rlt);
40. }
41. }

使用很简单，将方法名作为参数传进去就行了，最常用的是把很耗时的序列化函数传进去，以免阻塞UI进程，造成卡顿现象，影响用户体验。