Foundation Tasks (v4.0) 3/6/2015

Nicholas Ventimiglia | AvariceOnline.com

Unity3d Async Task Library

A utility library inspired by the Task Parallelism Library, but made especially for Unity3d. Supports running and waiting on actions running in background threads. Supports running and waiting on coroutines. Supports coroutines with return results and exception messages !

- Tasks support running on the main thread, background thread or as coroutines.
- Tasks support return results.
- Wait for your tasks to complete in a coroutine or as a thread blocking call
- Graceful exception handling (IsFaulted ?)
- Works on Windows, Mac, Ios, Android
- May have issues on WebGL as it does not support threading (but coroutine should work fine.)

Structure

- The "Foundation Tasks" folder contains a very basic example
- The "Foundation.Tasks" folder contains the plugin source code

Setup

Drop the Foundation.Tasks.dll into your plug in folder

Use

Run

Tasks have a static factory "Run" method which will create a new task in that started state. This method has overrides for every conscionable situation. You may also construct a task yourself and start it yourself.

Strategy

Tasks have a number of strategies you can choose from.

- Run in Background thread
- Run in a coroutine via the task manager
- Run in the current thread
- Run on the main thread
- Run a custom strategy. This is useful if you want to manually set the task's state, result and exception.

ContinueWith

ContinueWith is a extension method which allows you to execute a piece of code after the task is complete. This is useful with the coroutine strategy as a way to populate the Result property. You may chain multiple continue with's

Wait

- Wait will hault the thread until the task is complete. Only call this from a background thread. DO NO CALL THIS IN THE MAIN THREAD.
- WaitRoutine is a Coroutine that you may start. This routine will continue until the task is complete. Use this in the main thread.

TaskManager

The task manager is a monobehaviours which interfaces the task's with unity. It is responsible for executing on the main thread and running coroutines. You dont need to add this object to your scene, it is added automatically.

Exceptions

To set the task to the faulted state in an action simply throw an exception. The exception will be saved in the Task.Exception property. For coroutines you will need to set the task state and exception manually (as exception handeling in coroutines is limmited.

```
//Pass in an action, function, method or coroutine
var task = Task.Run(() =>
{
    throw new Exception("I am failure");
});
```

Debugging

I have a static flag to disable background threads. This will cause Unity to act funny (pausing the main thread), but, you will get a complete stack trace.

```
    /// <summary>
    /// Forces use of a single thread for debugging
    /// </summary>
    public static bool DisableMultiThread = false;

    /// <summary>
    /// Logs Exceptions
    /// </summary>
    public static bool LogErrors = false;
```

Scenario Example

Tasks are long running processes, so you should use tasks from a coroutine somewhere in your code. Just like the WWW class.

For example lets take a login task

```
  public class AccountMenu : Monobehaviour    {

    // run on instance startup
    IEnumerator Start(){
        // returns a Task<bool>
        var task = AccountService.Login();

        // wait for the task to finish
        yield return StartCoroutine(task.WaitRoutine());

        if(task.IsFaulted)
            // handle fault
        else
            // handle success

    }
  }
```

In the above example the AccountService would be returning a Task of some sort. Internally, it could be a action running in a background thread or a coroutine on the unity thread.

```
public class AccountService     {

    public Task<bool> Login(){
        // run in background thread
        return Task.Run(LoginInternal);
        // or run in unity thread as a coroutine
        return Task<.RunCoroutine<bool>(LoginInternal2);
    }

    bool LoginInternal(){
        // do work
    }

    IEnumerator LoginInternal2(Task<bool> task){
        // do work
        // manually set result / state
    }
}
```

Custom Strategy Example

The Custom strategy is used when you want to return a task without wrapping a action or coroutine. Here are two examples

Failed Sanity Check

For instance if the method fails a sanity check I will return a custom task in the faulted state and set the exception message manually. I figure this is less overhead than spinning up a background thread and throwing it.

```
void Login(string username){
    return new Task(TaskStrategy.Custom) {
        Status = TaskStatus.Faulted,
        Exception = new Exception("Invalid Username")
    };
    //or extension method
    return Task.FailedTask("Invalid Username");
}
```

Returning a Task without wrapping a method

I also use the custom strategy when I need to return a task but the internal method uses an arbitrary callback - such as in the case of UnityNetworking.

```
Task<bool> ConnectTask;
void Awake(){
    ConnectTask = new Task<bool>(TaskStrategy.Custom);
}
Task<bool> ConnectToServer(HostData username){
        ConnectTask.State = TaskState.Running;
    // Do Unity Connect Logic here
    // Consumer will 'wait' untill this server fails/successes the task
    return ConnectTask;
}

void OnConnectedToServer(){
    ConnectTask.Result = true;
    ConnectTask.State = TaskState.Success;
}

// todo fail and timeout for the connect task
```

Kitchen Sink Examples

```
// Assume running from a coroutine

//Turn an action into a waitable background task
var task = Task.Run(() =>
{
    //Debug.Log does not work in
    Debug.Log("Sleeping...");
    Task.Delay(2000);
    Debug.Log("Slept");
});
```

```
//wait for it
yield return StartCoroutine(task.WaitRoutine());

// check exceptions
if(task.IsFaulted)
    Debug.LogException(task.Exception)

//Valid if this method returned something
//var result = task.Result;

// Run a Task on the main thread (great for networking!)
Task.RunOnMain(() =>
{
    Debug.Log("Sleeping...");
    Task.Delay(2000);
    Debug.Log("Slept");
});

// Run a coroutine as a tasks
Task.RunCoroutine(RoutineFunction());

IEnumerator RoutineFunction(){
    Debug.LogOutput("Sleeping...");
    yield return new WaitForSeconds(2);
    Debug.LogOutput("Slept");
}

// Run a background task that then runs a task on the main thread
Task.Run(() =>
{
    Debug.Log("Thread A Sleep");
    Task.Delay(2000);
    Debug.Log("Thread A Awake");
    Task.RunOnMain(() =>
    {
        Debug.Log("Thread B Sleeping...");
        Task.Delay(2000);
        Debug.Log("Thread B Slept");
    });

    Debug.Log("Thread A Done");
});

// Run a coroutine with a result
Task.RunCoroutine<string>(RoutineFunction());
```

```
IEnumerator RoutineFunction(Task<string> task){
    //manually set State / Exception / Result
}
```