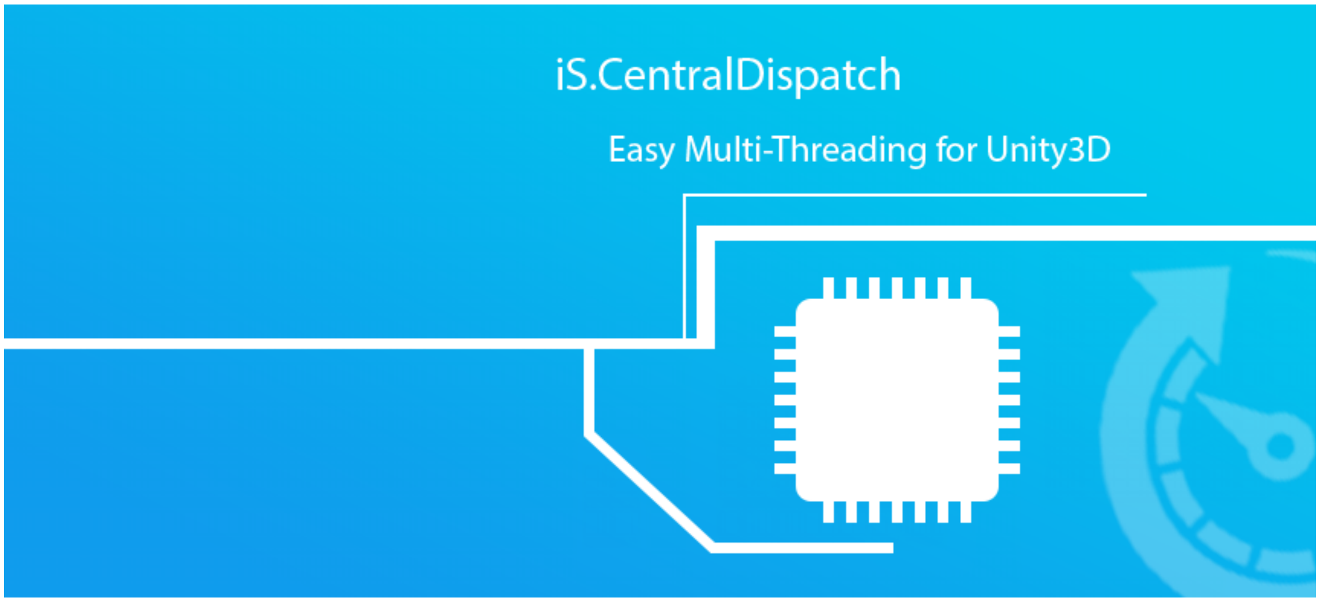


iS.CentralDispatch Documentation

July 27, 2016
Last Edit : July 27, 2016

GETTING START

Thanks for purchasing iS.CentralDispatch, we hope our product will help you with your projects.

IS.CentralDispatch is very easy to setup, we will discuss all the features in this document. Another faster way to get start is to play each example and read the example's source code. The main script(The one which call iS.CentralDispatch APIs) is named Ex*DemoController. We recommend you to play and understand each example then read the script. This is the best way we think to getting start.

This document will guide you to get start and then describe some details of iS.CentralDispatch to prevent incorrect usage.

Please e-mail us for support if you have any questions or needs help.

NEW FEATURES IN iS.CentralDispatch 1.1

Thanks for all of your supports, now iS.CentralDispatch was shipped with many cool new features !

Better Debugger

Using iSCDDebug.Log will have less delay when huge number of logs is issued.

Logs from other thread will be displayed in another color in console.

Known Issues :

1. Using iSCDDebug.Log may block your thread for a while at some certain situations. (Workaround : call iSCDDebug.Log less or remove them when you decided to publish your game)
2. Stack trace from iSCDDebug.Log is not correct, and will not open caller script.

We are working on a smarter debugger system, and planned to ship in next version. :)

Async Instantiating

In 1.1, we provided the following new APIs for you to do Async Instantiate :

```
/// <summary>
/// Instantiate the specified prefab at certain position and rotation.
///
/// This function will block caller thread until the prefab is instantiated and return.
/// </summary>
/// <param name="prefab">Prefab to instantiate.</param>
/// <param name="position">Position.</param>
/// <param name="rotation">Rotation.</param>
static public UnityEngine.Object Instantiate(GameObject prefab, Vector3 position, Quaternion rotation){
    if (!IsRuntimeStarted()) return null;

    return GetRuntime ().ThreadedInstantiate (prefab, position, rotation);
}

/// <summary>
/// Instantiate the specified prefab at the Transform's position.
///
/// This function will block caller thread until the prefab is instantiated and return.
/// </summary>
/// <param name="prefab">Prefab.</param>
/// <param name="place">Place.</param>
/// <param name="setChild">If set to <c>true</c> set child.</param>
static public UnityEngine.Object Instantiate(GameObject prefab, Transform place, bool setChild){
    if (!IsRuntimeStarted()) return null;

    return GetRuntime ().ThreadedInstantiate (prefab, place, setChild);
}

/// <summary>
/// Instantiates the specified prefab at the Transform's position.
///
/// This function will not block the caller thread and returns nothing.
/// When your prefab is spawned, callback will be issued with instance of the prefab.
/// </summary>
/// <param name="prefab">Prefab.</param>
/// <param name="place">Place.</param>
/// <param name="setChild">If set to <c>true</c> set child.</param>
/// <param name="callback">Callback.</param>
static public void InstantiateAsync(GameObject prefab, Transform place, bool setChild, Action<UnityEngine.Object> callback){
    if (!IsRuntimeStarted()) return;

    GetRuntime ().ThreadedInstantiateTasked (prefab, place, setChild, callback);
}
```

Using Async Instantiating will reduce the cost of spawning new object from your thread in a smarter way and give you more power to write your multi-threaded code in Unity.

To know more about Async Instantiating, please refer to the following chapter “Async Instantiating”

Main Thread Loads Balancer

In this version we provided a new way to deals with massive main thread task called “Main Thread Loads Balancer”, which will makes your game runs at your targeted frame rate even when you calling a huge number of main thread dispatch.

You don’t have to make any changes to your current scripts, every will be faster and safer after you upgrade.

WRITE YOUR FIRST MULTI-THREADED SCRIPT

Using SPINACH.iSCentralDispatch

To use features of iS.CentralDispatch, you must import the iS.CentralDispatch's namespace first. Type this line of code at the top of your script:

```
using SPINACH.iSCentralDispatch;
```

Start a New Thread

After you import iS.CentralDispatch into your script, you're ready to start a new thread at anytime. You can call :

```
iSCentralDispatch.DispatchNewThread ("name",MyThread);
```

to start a new thread. The type of second parameter is "System.Action" which refer to the lines of codes that your thread would execute. If you don't know what does System.Action means, just think if as a function, or method.

So you can define a new function now :

```
void MyThread () {
     while(true){
          iSCDDebug.Log("Hello World!");
     }
}
```

So this is basically a function that returns nothing with a while loop inside. You may already notice that the while loop will never be finish since the condition is always true. If you call MyThread from main thread, such as call in Update, Start, your game will be stuck forever because the dead loop will block your application. This will be a great chance to test threads.

Instead of call it directly, we start a new thread to execute it :

```
void Start () {
     iSCentralDispatch.DispatchNewThread ("Hello World", MyThread);
}
```

Run this code then you will see “Hello World” rush into your console

Booooooo! You just wrote your first Multi-Thread application.

ISCDDebug.Log vs Debug.Log

In the example code, we use ICDDebug.Log to log instead of Debug.Log, what is the differences between them?

in fact, Debug.Log is not a initially thread-safe API, but you still can call it from your thread, why ?

When you call Debug.Log from your thread, Unity will block your thread to wait for main thread finished Updates, which will heavily drop the performance if you have too many things to log.

ISCDDebug.Log is a completely thread-safe API and will not drop any performance. Also, if you log via ISCDDebug.Log, threads information will be included automatically in the log.

CALL UNITY API FROM YOUR THREAD

Since Unity APIs is not thread-safe, you can not call them from your own thread instead of main thread. Without calling Unity APIs, multi-threaded design in Unity became hard to use or even useless. But don't worry, iS.CentralDispatch will helps here.

Dispatch Main Thread

In order to do something in main thread, you can call :

```
iSCentralDispatch.DispatchMainThread (task);
```

The parameter "task" is also a System.Action, which means you can pass some functions to it. A simpler way to do here is use a "Lambda Expression".

Using Lambda Expression

So what is a lambda expression? It is a C# syntax for defining a block of codes. It basically looks like this :

```
(input parameters here) => {
     // some codes here.
} ;
```

Think about a short hand of defining function. We don't need anything as input value this time so we simply leave the "()" empty. The final code will looks like this :

```
void MyThread () {
     while(true){

          float speed = 0;

          iSCentralDispatch.DispatchMainThread (() => {
               speed = GetComponent<Rigidbody>().velocity.magnitude;
          } );

          //A lot of complex computations here for the speed;

          iSCentralDispatch.DispatchMainThread (() => {
               transform.Translate(Vector3.forward * speed
                    * Time.deltaTime
                    ,Space.Self);
          } );

     }
}
```

The example code firstly get the current speed of the GameObject and do a huge calculation of the wanted speed. Then move it in main thread.

If you call iSCentralDispatch.DispatchMainThread in you thread, your thread will be blocked until main thread finished your task.

Optimize Performance

Since iSCentralDispatch.DispatchMainThread is a blocking function, it is recommended to use it as less as possible to enhance performances.

A useful way to do is try first get all informations your need from main thread before you start the calculation, then set them back after that. For example, if your thread's mission is to smooth a Mesh(Mesh is a Unity API and can not be used outside main thread), you can get all informations using iSCentralDispatch.DispatchMainThread and store them into your own struct, or variables. After all calculation is completed in you thread, you can set them back to Mesh via iSCentralDispatch.DispatchMainThread again. So the whole process only do main thread dispatch twice.

Another important point is : If the part of code below a main thread task does not depends on it, for example, when all the main thread task do is to set some values back to a component, you can use :

```
iSCentralDispatch.DispatchTaskToMainThread (task, finishCallBack);
```

to establish a task to main thread without blocking the current thread. The parameter task is the lines of code you want to execute in main thread. finishCallBack will be called when the task has been executed in main thread. finish callback is a function with a int input parameter indicates the ID of thread of where the task from. If you doesn’t need any callback, just pass null to it.

Life Reporting

iS.CentralDispatch will monitor your thread to make sure everything goes well and apply control to it.

You must call :

```
iSCentralDispatch.LifeReport ();
```

at the start of each loop of your thread to help iS.CentralDispatch monitor your thread and make everything safe.

The corrected code of the last example should looks like this :

```
using UnityEngine;
using System.Collections;
using SPINACH.iSCentralDispatch;

public class Example : MonoBehaviour {

	void Start () {
		iSCentralDispatch.DispatchNewThread ("Hello World", MyThread);
	}

	void MyThread () {
		while(true){

			iSCentralDispatch.LifeReport ();

			float speed = 0;

			iSCentralDispatch.DispatchMainThread (() => {
				speed = GetComponent<Rigidbody>().velocity.magnitude;
			} );

			//A lot of complex computations here for the speed;

			iSCentralDispatch.DispatchTaskToMainThread (() => {
				GetComponent<Rigidbody>().velocity = transform.forward * speed;
			} );
		}
	}
}
```

CONTROLLING THREADS

Identifying Threads

The API iSCentralDispatch.DispatchMainThread returns an int value which is the ID of started thread. You can store the ID to control your thread.

```
int threadID = iSCentralDispatch.DispatchNewThread ("Hello World", MyThread);
```

Pause and Resuming

Use the following API to pause of resume threads :

```
iSCentralDispatch.PauseThread (threadID);
iSCentralDispatch.ResumeThread (threadID);
```

Aborting Thread

Use the following API to abort thread :

```
iSCentralDispatch.AbortThread (threadID);
```

Setting Priority for Thread

Use the following API to set priority for thread :

```
iSCentralDispatch.SetPriorityForThread (threadID, iSCDThreadPriority.Normal);
```

ASYNC INSTANTIATING

We introduced a new feature in 1.1 version called “Async Instantiating”, which allows you to instantiate objects directly from your thread.

Simple Instantiate

You can call :

iSCentralDispatch.Instantiate (prefab, position, rotation)

directly from your thread to let iSCentralDispatch instantiate an object for you, which uses exactly same as the Unity’s Instantiate.

If you are using a Transform component to store spawning informations such as position and rotation, you may prefer :

iSCentralDispatch.Instantiate (prefab, place, setchild)

While “place” is the Transform component you are using as the placeholder. If you want to make your new GameObject become a child of the placeholder, please set the “setchild” flag to true.

Instantiating Massive Objects

Use the method we talked above to instantiating is good for a small amount of object since both of them are block function. They will block your thread until the object finished spawning and able to return.

If you’re spawning a huge number of objects you may like to do asynchronous instantiate.

Call :

iSCentralDispatch.InstantiateAsync(prefab, place,setchild, finishedCallback);

to request an async instantiation. A callback will be fired when the object is finished and able to return.

An example usage can be found in “Example 6” :

```
void MyThread () {
    iSCDDebug.Log ("I am an alternative thread, I am preparing to spawn some big things.");
    iSCDDebug.Log ("Before I start, I would like to have a rest, so I pause my self for 2 seconds and use Enumerator to wait.");

    iSCentralDispatch.DispatchMainThread (() => {
        StartCoroutine(Timer());
    });
    iSCentralDispatch.PauseThread (threadID);

    iSCDDebug.Log ("Now I will start spawning...");

    for (int i = 0; i < 2000; i++) {
        iSCentralDispatch.InstantiateAsync (objects [0], spawnPoint, false, (UnityEngine.Object obj) => {
            //callback will be execute in main thread !
            GameObject gobj = obj as GameObject;
        });
    }

    //iSCentralDispatch.PauseThread (threadID);

    iSCDDebug.Log ("I have spawned 2000 of objects without blocking the gameplay, isn't that cool ?");
}

IEnumerator Timer(){
    yield return new WaitForSeconds (2);
    iSCentralDispatch.ResumeThread (threadID);
}
```

MAIN THREAD LOADS BALANCER

In this version we provided a new way to deals with massive main thread task called “Main Thread Loads Balancer”, which will makes your game runs at your targeted frame rate even when you calling a huge number of main thread dispatch.

You don’t have to make any changes to your current scripts, every will be faster and safer after you upgrade

Setting Target Frame rate

The default target frame rate is 60 fps, you can call :

```
iSCentralDispatch.SetTargetFramerate (30);
```

to change.

USING DEBUGGER

The Debugger

You already have some experience on using the iS.CentralDispatch debugger :

```
iSCDDebug.Log("Hello World!”);
```

When you call this function, a Debugger instance will attach to the current runtime automatically. To attach a Debugger manually, call :

```
iSCDDebug.StartDebugging ();
```

While Debugger is attach, the performance of you game will be affected. You need to disable Debugger if it no longer useful :

```
iSCDDebug.StopDebugging ();
```

Debugger will not be attached by default.

Logging

There are three API available for you to log :

```
iSCDDebug.Log ("Hello");
iSCDDebug.LogWarning ("Hello");
iSCDDebug.LogError (“Hello");
```

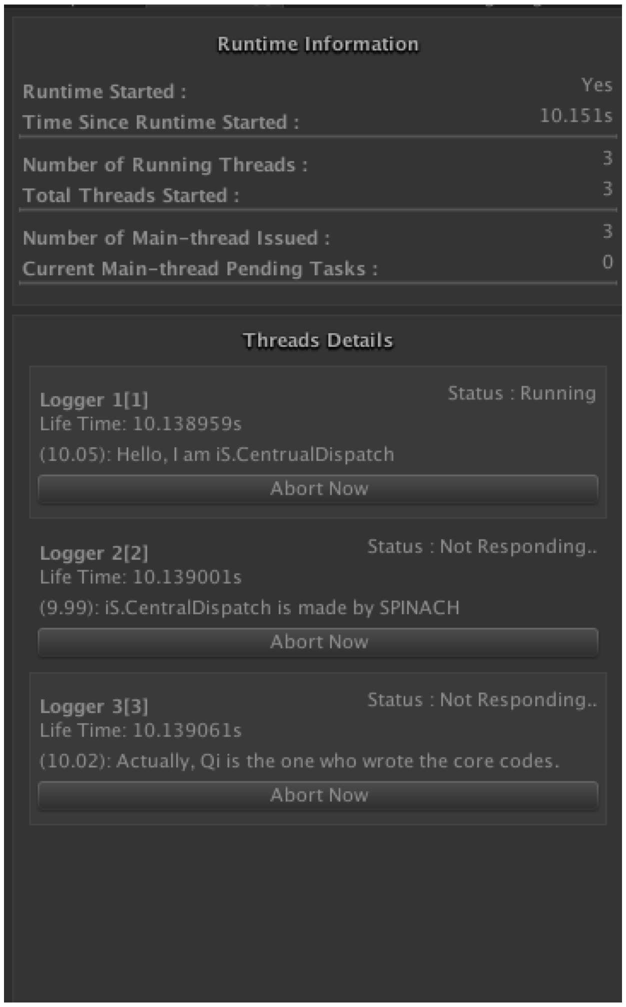
LogWarning and LogError will change to look of your log like when using UnityEngine.Debug.

The Debugger Window

iS.CentralDispatch comes with a powerful debugger window to view running threads.

To open debugger window, go to menu bar and click : Window -> iSCD Debugger

Once you start the debugger window, a debugger will be attached to runtime.



Runtime Information

Runtime Started :	Yes
Time Since Runtime Started :	10.151s
Number of Running Threads :	3
Total Threads Started :	3
Number of Main-thread Issued :	3
Current Main-thread Pending Tasks :	0

Threads Details

Logger 1[1] Status : Running
Life Time: 10.138959s
(10.05): Hello, I am iS.CentrualDispatch

Logger 2[2] Status : Not Responding..
Life Time: 10.139001s
(9.99): iS.CentralDispatch is made by SPINACH

Logger 3[3] Status : Not Responding..
Life Time: 10.139061s
(10.02): Actually, Qi is the one who wrote the core codes.

MORE INFORMATIONS

Please contact us if you have any questions or need support, we will reply every support e-mail in 24 hours.

E-mail : support@spinachelectr.com

Please use English or Simplified Chinese.

Our official website : https://www.spinachelectr.com.