Using the Work Queue Manager

Version 2018.1
2024-04-03

InterSystems Corporation 1 Memorial Drive Cambridge MA 02142 www.intersystems.com

Using the Work Queue Manager
Caché Version 2018.1 2024-04-03

For Support questions about any InterSystems products, contact:

InterSystems Worldwide Response Center (WRC)
Tel: +1-617-621-0700
Tel: +44 (0) 844 854 2917
Email: support@InterSystems.com

Table of Contents

Using the Work Queue Manager

The work queue manager enables you to distribute work to multiple concurrent processes in order to improve performance. InterSystems code uses the work queue manager internally in several places, and you can use it for your own needs as well. This article discusses the following topics:

Also see the class reference for the %SYSTEM.WorkMgr class.

1 Basics

In order to use this feature, you need to divide the work into units that can be processed independently. Once you have identified such units of work, there are three basic steps:

1. Initialize the worker jobs. To do this, call the **Initialize()** method of %SYSTEM.WorkMgr, which returns a *work queue* — an instance of %SYSTEM.WorkMgr.

   You can specify the number of parallel jobs to use, or you can use the default, which depends on the machine and operating system.

2. Add work queue items. To do this, call the **Queue()** method of the work queue. As arguments, pass the name of a class method (or a routine), as well as any arguments.

   When you add a work queue item, the work queue immediately begins work on that item, if possible. If the number of work items is larger than the number of jobs for a work queue, the work queue divides the work items into *groups*, and processes the groups one at a time. For example, if there are 100 work items and four jobs, the work queue processes the work items four at a time.

   When the work item is run, it uses the security context of the caller.

3. Wait for the work to be completed. To do this, call the **WaitForComplete()** method of the work queue.

4. Continue processing as appropriate for your application.

The following example shows the steps:

ObjectScript

```
Set queue=##class(%SYSTEM.WorkMgr).Initialize(,.sc)
If $$$ISERR(sc) {
   Return sc
}
For i = 1:1:filelist.Count() {
    Set sc=queue.Queue("..Load",filelist.GetAt(i))
    If $$$ISERR(sc) {
        Return sc
    }
}
Set sc=queue.WaitForComplete()
If $$$ISERR(sc) {
    Return sc
}
```

In this example, the code initializes the work queue manager and then iterates through a list of files. For each file, the code adds a work queue item that loads the file. After adding all the work queue items, the code waits for the work to be completed.

1.1 Details on the Basic Methods

This section provides reference information on the basic methods shown in the previous section. These methods are available in %SYSTEM.WorkMgr.

Initialize()

```
classmethod Initialize(qspec As %String = "", ByRef sc As %Status, numberjobs As %Integer) as
WorkMgr
```

Creates, initializes, and returns a *work queue*, that is, an instance of %SYSTEM.WorkMgr that you can use to perform parallel processing. The arguments are as follows:

- *qspec* is a string of compiler flags and qualifiers that affect code running within this work queue. See "Viewing Class Compiler Flags and Qualifiers" in the chapter "Defining and Compiling Classes" in *Defining and Using Classes*.
- *sc*, which is returned by reference, is a %Status value that indicates whether the system was successful when it created and initialized this work queue.
- *numberjobs* is the number of parallel workers to use within this work queue. The default depends on the characteristics of the machine and operating system.

You can also specify the number of parallel workers by including the `/multicompile=num` qualifier within the *qspec* string, with the following exceptions:

- If `/multicompile=1`, the work queue uses the default number of parallel workers, which depends on the machine and operating system.
- If `/multicompile=-1`, the work queue uses only one worker.

If you specify `/multicompile=num` and you specify *numberjobs*, *numberjobs* takes precedence.

Queue()

```
method Queue(work As %String, args... As %String) as %Status
```

Adds a work unit to a work queue. The arguments are as follows:

- *work* specifies the code to execute. The code should return a %Status value to indicate success or failure. In this case:

  - To call a class method, use the syntax `##class(Classname).ClassMethod` where *Classname* is the fully qualified name of the class and *ClassMethod* is the name of the method. If the method is in the same class, you can use the syntax `..ClassMethod` as shown in the example.
  - To call a subroutine, use the syntax `$$entry^rtn` where *entry* is the name of the subroutine and *rtn* is the name of the routine.

  If the code does not return a %Status value, then:

  - To call a class method, use the syntax `=##class(Classname).ClassMethod` (or `=..ClassMethod` if the method is in the same class)
  - To call a subroutine, use the syntax `entry^rtn`

  Also see the "Requirements" subsection.

- *args* is a comma-separated list of arguments for the class method or subroutine. To pass a multidimensional array as an argument, precede that argument with a period as usual so that it is passe/d by reference.

  Note that the size of the data passed in these arguments should be kept relatively small. If there is a large amount of information that needs to be provided, then use a global instead of passing arguments.

The security context of the caller is recorded, and each work item runs within that security context.

WaitForComplete()

```
method WaitForComplete(qspec As %String, errorlog As %String) as %Status
```

Waits for the work queue to complete all the items and then returns a %Status value to indicate success or failure. The %Status value contains information from all %Status values returned by the work items. The arguments for **WorkForComplete()** are as follows:

- *qspec* is a string of compiler flags and qualifiers. See "Viewing Class Compiler Flags and Qualifiers" in the chapter "Defining and Compiling Classes" in *Defining and Using Classes*.
- *errorlog*, which is returned as output, is a string of error information (if any).

1.2 Requirements

The following requirements apply to the units of work that you pass into a work queue:

- As noted previously, by default, all units of work are expected to return a %Status value to indicate success or failure so that the **WaitForComplete()** method can return a %Status value to indicate overall success or failure. A unit of work can also throw an exception; in this case, the exception is trapped and converted to a %Status value to be returned in the master process.
- All the units of work must be totally independent and must not rely on each other. You cannot rely on the order in which the units of work are processed. For example, one unit of work cannot rely on output from another unit of work.
- If the units of work change the same global, be sure to add locking to ensure one worker cannot change the global while another worker is reading it.
- The units of work should not use exclusive news, kills or unlocks, because these will interfere with the framework.
- If you use process-private globals to store data during the processing, note that because multiple jobs will be processing each chunk, you cannot rely on accessing these process-private globals from the master process (or even from another chunk).

- The size of each unit of work should be on the order of thousands of lines of ObjectScript code to ensure the overhead of the framework is not a significant factor. Also, rather than a having than a few very large units of work (for example, 4) it is better to have a fairly large number (for example, 100) of units of work, because this permits scaling up when there are more CPU cores.

2 Including Callbacks for Work Items

You can specify callbacks for work items — code that the work queue manager should execute after completing the work item. To do this, instead of calling the **Queue()** method, call the **QueueCallback()** method:

```
method QueueCallback(work As %String, callback As %String, args... As %String) as %Status
```

The *work* and *args* methods are the same as for the **Queue()** method.

The *callback* argument specifies the callback code to execute. For this argument:

- To call a class method, use the syntax `##class(Classname).ClassMethod`
- To call a subroutine, use the syntax `$$entry^rtn`.

The class method or subroutine must accept the same arguments, in the same order, as the main work item. The master process passes the same arguments to the main work item and to the callback code.

The callback code can use the following public variables:

- *%job*, which contains the job ID of the process that actually did the work.
- *%status*, which contains the %Status value returned by the work unit.
- *%workqueue*, which is the OREF of the work queue instance.

These public variables are available within the callbacks but not within the work items.

3 Using Callbacks to Determine Completion

The basic technique uses the **WaitForComplete()** method to wait until all work items are complete. You can instead use callbacks to indicate that work is complete. To do this:

- Instead of using **Queue()** to add work items, use **QueueCallback()**.
- In the callback code, when the work is complete for all work items, set the public variable *%exit* to 1.
- Instead of using **WaitForComplete()**, use **Wait()**.

The **Wait()** method is as follows:

```
method Wait(qspec As %String, byRef AtEnd As %Boolean) as %Status
```

This method waits for a signal from a callback to exit back to the caller. Specifically, it waits for the callback code to set the public variable *%exit* equal to 1. This method returns *AtEnd* by reference. If *AtEnd* is 1, all the work is complete. If *AtEnd* is 0, there are work items that did not get done.

4 Controlling Output to the Current Device

By default, if work items generate output (**WRITE** statements) to the current device, the work queue saves the output in a buffer until the end of **WaitForComplete()** or **Wait()**. If you want a work item to generate output earlier, have that work item call the **Flush()** class method of %SYSTEM.WorkMgr.

```
ClassMethod Flush() as %Status
```

When the work item calls this method, that causes the parent work queue to write all saved output for the work item.

Also, as usual, you can use the `-d` flag to suppress all output to the current device. In this case, the **Flush()** method does nothing, because there is no output.

5 Pausing and Resuming a Work Queue

The %SYSTEM.WorkMgr class provides methods you can use to pause and resume work within a work queue. These methods do not affect the work items that are currently in progress, but instead affect items that have not yet been started. (For information on halting work completely, including work in progress, see the next section.)

Pause()

```
method Pause() as %Status
```

Prevents the work queue processes from accepting *additional* items from this specific work queue. Any work items currently in progress are completed as usual.

Resume

```
method Resume() as %Status
```

Resumes work in this work queue, if it had previously been paused via **Pause()**. Specifically, this method enables the work queue processes to accept and start additional items in the work queue, if any.

6 Stopping a Work Queue and Removing Work Items

You can stop a work queue, interrupting any work items in progress and removing any queued work items. To do this, call the **Clear()** method of the work queue.

```
method Clear() as %Status
```

Any work items are immediately stopped. The system removes and then recreates the work queue, with no attached work items. Processing is now considered done, so the system returns immediately from **Wait()** or **WaitForComplete()**.

7 Specifying Setup and Teardown Processing

Each work queue typically has multiple worker jobs. If there are more work items than worker jobs, then a worker job will perform multiple work items, one at a time. It is useful to identify any setup steps needed before these work items start, and invoke all such logic before adding the work items to the queue.

The %SYSTEM.WorkMgr class provides methods, **Setup()** and **TearDown()**, that you can use to define the setup activity and the cleanup activity for the worker jobs. For example, use **Setup()** to set public variables for use within the worker job, and use **TearDown()** to kill those variables. You can also use **Setup()** to take out locks and to set process-private globals, and you would use **TearDown()** to release those locks and remove those globals.

In either case, you must call **Setup()**, **TearDown()**, or both before calling **Queue** or **QueueCallback**. The **Setup()** and **TearDown()** methods save information in internal globals used only by the work queue manager. When any worker job starts its first work item from this queue, that worker job first checks the work manager queue globals to see if there is any setup logic. If so, the worker job executes that logic and then starts the work item. The worker job does not execute the setup logic again. Similarly, after any worker job finishes its last work item from the queue, that worker job checks to see if there is any teardown logic. If so, the worker job executes that logic.

The following provides details for these methods:

Setup()

```
method Setup(work As %String, args... As %String) as %Status
```

Specifies the code for a worker process to call before processing its first item from the queue. If you use this method, call it before calling **Queue** or **QueueCallback**.

The arguments are as follows:

- *work* specifies the setup code to execute. See the comments for **Queue()**, described earlier.
- *args* is a comma-separated list of arguments for this code. To pass a multidimensional array as an argument, precede that argument with a period as usual so that it is passed by reference.

  Note that the size of the data passed in these arguments should be kept relatively small. If there is a large amount of information that needs to be provided, then use a global instead of passing arguments.

TearDown

```
method TearDown(work As %String, args... As %String) as %Status
```

Specifies the code for a worker process to call to restore the process to its previous state, after processing its last item from a queue. If you use this method, call it before calling **Queue** or **QueueCallback**.

The arguments are the same as for **Setup()**, except that *work* specifies the teardown code to execute.

8 Notes on the Worker Jobs

The worker jobs are separate processes and can viewed, managed, and monitored like other processes. Note the following points:

- When you call **WaitForComplete()** or **Wait()**, for a given queue, that queue is moved to the top of the priority list so the background workers will process work from this queue before any other existing queue.

- When a queue is deleted or cleared, if there is a worker job actively processing something for this queue, the system waits (by default) up to 5 seconds for the job to finish. If the job has not finished in that period of time, the system forces this job to exit and then start up additional worker jobs to replace it.
- After a worker job is no longer used, it remains available — for a span of time — for use by other work manager queues. After a long enough period of inactivity, the job is removed. This timeout period is subject to change and is deliberately not documented.
- The superserver starts the worker jobs, which means that they run under the name of the operating system user used by the superserver process. This username may be different from the currently logged-in operating system user.

If you need to need to know whether a given process is a worker job, call **$system.WorkMgr.IsWorkerJob()** from within that process (call the **IsWorkerJob()** method of the class %SYSTEM.WorkMgr).

The work queue (instance of %SYSTEM.WorkMgr) provides the properties NumWorkers (the number of worker jobs assigned to this queue) and NumActiveWorkers (number of currently active worker jobs).