InterSystems™
IRIS Data Platform

Using the Work Queue Manager

Version 2024.1
2024-07-02

Using the Work Queue Manager
InterSystems IRIS Data Platform   Version 2024.1   2024-07-02
Copyright © 2024 InterSystems Corporation
All rights reserved.

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

1

Introduction to the Work Queue Manager

The Work Queue Manager is a feature of InterSystems IRIS® data platform that enables you to improve performance by distributing work to multiple concurrent processes programmatically. It provides an efficient and straightforward API that enables you to off-load process management.

InterSystems code uses the Work Queue Manager internally in several places. Wherever you have code that meets the requirements, you can use the Work Queue Manager to perform parallel processing.

In addition to reducing the time needed to process large workloads, the Work Queue Manager provides you with a high level of control over how the CPU resources on your system are used. For example, you can create categories of work and define the number of worker jobs assigned to the categories. Additionally, the Work Queue Manager provides work load metrics so that you can monitor the load on your system in real time.

The following pages provide details on using this feature. Also see %SYSTEM.WorkMgr and %SYSTEM.ShardWorkMgr, and %SYSTEM.AbstractWorkMgr in the class reference.

2

Code Requirements

The Work Queue Manager processes *units of work* (also called *work items*), which are ObjectScript class methods or subroutines that meet the following requirements:

- The class method or subroutine can be processed independently. For example, a unit of work cannot rely on output from a different unit of work. Independence is required because units of work may be processed in any order. (However, you can use callbacks to execute work sequentially if needed. For more information, see Using Callbacks.)

- The class method or subroutine is on the order of thousands of lines of ObjectScript code in size. This requirement ensures that the overhead of the framework is not a significant factor.

  Furthermore, it is preferable to use a large number (for example, 100) of smaller units of work rather than a small number of very large units of work (for example, 4). Distributing the work in such a way permits the system to scale up when more CPU cores are available.

- The code returns a %Status value to indicate success or failure so that the **WaitForComplete()** method can return a %Status value to indicate overall success or failure. Alternatively, the unit of work can throw an exception that is trapped, converted to a %Status value, and returned in the master process.

- If the code changes the same global as a different unit of work, you must employ a locking strategy to ensure that one worker job cannot change the global while another worker is reading it.

- The code does not include exclusive NEWs, KILLs, or unLOCKs since these interfere with the framework.

- If the code includes process-private globals for storing data, these process-private globals are not accessed from the master process or from any other chunk. This requirement is necessary since multiple jobs process each chunk.

- Any logic called as part of the class method or subroutine is correctly cleaned up such that no variables, locks, process-private globals, or other artifacts remain in the partition. This requirement is important since the same process will be used to subsequently process completely separate work items.

3

Using the Work Queue Manager

This section discusses %SYSTEM.WorkMgr specifically. Also see %SYSTEM.ShardWorkMgr in the class reference; both classes share a common API.

To use the Work Queue Manager to perform parallel processing:

1. Identify the ObjectScript code that you want to process in parallel. See Code Requirements.

2. Divide your code into *units of work*.

3. Create a *work queue*, which is an instance of the %SYSTEM.WorkMgr class. To do so, call the **%New()** method of the %SYSTEM.WorkMgr class. The method returns a work queue.

   You can specify the number of parallel worker jobs to use, or you can use the default, which depends on your machine and operating system. Additionally, if you have created categories, you can specify the category that the jobs should be taken from.

   When you create a work queue, the Work Queue Manager creates the following artifacts:

   • A global that contains information about the work queue such as what namespace the work queue runs in

   • A location and an event queue for the serialized units of work that the work queue must process

   • A location and an event queue for completion events that are created as the work queue finishes processing units of work

4. Add units of work (also called work items) to the work queue. To do so, you can call the **Queue()** or **QueueCallback()** method. As arguments, you pass the name of a class method (or subroutine) and any corresponding arguments.

   Processing begins immediately on items added to the queue.

   If there are more items in the queue than there are worker jobs available to the queue, then the jobs compete to empty the queue. For example, if there are 100 items and four jobs, each job removes an item from the head of the queue, processes it, and then returns to the head of the queue to remove and process another item. This pattern continues until the queue is empty.

   The Work Queue Manager uses the security context of the caller when running a work item.

   When you queue work items, the Work Queue Manager performs the following tasks:

   • Serializes the arguments, security context, and class method or subroutine that comprises the unit of work, and then inserts the serialized data into the global that lists the units of work associated with the work queue.

   • Signals an event on the work queue.

   • If additional worker jobs are required and available to process the units of work, causes a worker job to attach to the work queue and decrements the number of available worker jobs.

5. Wait for the work to be completed. To do so, you can call the **WaitForComplete()** method of the work queue.

    The Work Queue Manager then performs the following tasks:

    •   Waits for a completion event

    •   Displays output such as workload metrics to the terminal

    •   Collects any errors related to the unit of work

    •   If you added units of work to the work queue using the **QueueCallback()** method, runs the callback code

6. Continue processing as appropriate for your application.

3.1 Example

The following example shows these basic steps:

ObjectScript

```
Set queue=##class(%SYSTEM.WorkMgr).%New()
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
}
```

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 a file. After adding all the work queue items, the code waits for the work to be completed.

Note: The %SYSTEM.WorkMgr class supports more complex workflows with the methods described in other topics.

3.2 Methods for Creating Work Queues

To create work queues, add items, and check for completion, use the following methods of the %SYSTEM.WorkMgr class:

%New()

> classmethod %New(qspec As %String = "", numberjobs As %Integer, category) as WorkMgr

Creates, initializes, and returns a *work queue*, which is an instance of the %SYSTEM.WorkMgr class that you can use to perform parallel processing. The method accepts the following arguments:

> ### *qspec*
>
> > A string of compiler flags and qualifiers that affect code running within this work queue. See Flags and Qualifiers.

numberjobs

The maximum number of parallel worker jobs to use in this work queue. The default depends on the characteristics of the machine and operating system.

category

The name of the category that supplies the worker jobs to use in this work queue. For more information, see Managing Categories.

The system does not allocate any worker jobs to the queue upon creation. Worker jobs are allocated only after you add a unit of work to the work queue.

Queue()

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

Adds a unit of work to a work queue. The method accepts the following arguments:

work

The code to execute. In general, the code should return a %Status value to indicate success or failure.

If the code returns a %Status value, you can use the following syntax:

- `##class(Classname).ClassMethod` for a class method, 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.

- `$$entry^rtn` for a subroutine, 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, use the following syntax instead:

- `=##class(Classname).ClassMethod` for a class method (or `=..ClassMethod` if the method is in the same class)

- `entry^rtn` for a subroutine

See About Units of Work for information about the requirements for units of work.

args

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 passed by reference.

The size of the data passed in these arguments should be relatively small to make the most of the framework. To pass a large amount of information, use a global instead of an argument.

As you queue units of work, the system allocates worker jobs one at a time up to the *numberjobs* value that you specified when you created the work queue or up to the default value. Additionally, 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 method accepts the following arguments:

qspec

A string of compiler flags and qualifiers. See Compiler Flags and Qualifiers.

errorlog

A string of any error information, which is returned as output.

3.3 Properties of Work Queues

Each work queue (or instance of %SYSTEM.WorkMgr) has the following properties:

NumWorkers

The number of worker jobs assigned to the work queue.

NumActiveWorkers

The number of currently active workers.

3.4 Returning Information

The work units can return information (other than status), which is helpful especially in shard queue manager situations where communication with the parent process may not be simple. To do this, the work units can write to the public *%result* multidimensional array. The caller can access this array in either of two ways:

- This array is returned by reference in the **WaitOne()** method.

- The variable *%result* is available within the **QueueCallback()** method.

4

Managing Categories

A *category* is an independent pool of worker jobs for the Work Queue Manager. When you initialize a set of worker jobs, you can specify the category that supplies the workers. If any of the worker jobs in the set request additional worker jobs while executing work items, then the new worker jobs are from the same category.

Each work category always has at least one worker job available to immediately process work items. The category also has a maximum number of additional worker jobs that can be simultaneously working.

For example, suppose that you assign a maximum of eight workers to the SQL category supplied by the system. Then, suppose that you create a category for processes related to building Business Intelligence cubes, and assign a maximum of four workers to that category. Then whatever processing is occurring in the SQL category, the workers in the BusinessIntelligence category are available to process their work items immediately.

4.1 System Categories

The system includes two categories that you cannot delete: SQL and Default. The SQL category is for any SQL processing performed by the system, including parallel processing of queries. The Default category supplies worker jobs when you initialize a set of worker jobs without specifying a category.

4.2 Category Properties

Each category has properties that affect the behavior of each work queue in the category. These properties are:

DefaultWorkers

> When a work queue in this category is created and no worker job count is specified, this becomes the number of worker jobs in the work queue. The default value for this property is the number of cores.

MaxActiveWorkers

> Maximum number of active worker jobs kept in the pool of jobs servicing requests in this category (in addition to the one job that is always available). Idle jobs are detected and new jobs are started automatically to keep the maximum active job number around this limit. The default value is twice the number of cores.

MaxWorkers

Maximum number of worker jobs for a work queue in this category. If you specify a larger number of worker jobs when creating the work queue, this limit is used instead. The default is twice the number of cores.

4.3 Creating, Modifying, and Deleting Categories

You can create categories, adjust category properties, and delete custom categories in the Management Portal. See Configuring Work Queue Manager Categories.

If you prefer, you can also work with categories using the Config.WorkQueues APIs.

5
Using Callbacks

A callback is code that the Work Queue Manager must execute after completing a work item. You can use callbacks for two reasons:

- To perform work that is dependent on the completion of a work item

- To signal that all queued work is completed if you choose to complete work items asynchronously

5.1 Including Callbacks for Work Items

To add a callback, you call the **QueueCallback()** method instead of the **Queue()** method when adding work items to the work queue:

```
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. However, the *callback* argument specifies the callback code to execute using the following syntax:

- `##class(Classname).ClassMethod` for a class method

- `$$entry^rtn` for a subroutine

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 access 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 unit of work

- *%workqueue*, which is the OREF of the work queue instance

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

5.2 Including Callbacks to Determine Completion

Instead of using the **WaitForComplete()** method to wait for all the queued work in a work queue to be completed before returning to the master process, you can poll the Work Queue Manager to determine completion as follows:

- Use the **QueueCallback()** method instead of the **Queue()** method to add work items to the work queue as described in the previous section.

- When the work is completed for all work items, set the public variable *%exit* to 1 in the callback code.

- Use the **Wait()** method instead of the **WaitForComplete()** method:

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

The **Wait()** 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. **Wait()** returns *AtEnd* by reference. When *AtEnd* is 1, all the work is completed. Alternatively, if *AtEnd* is 0, one or more work items are not completed.

6

Controlling Output to the Current Device

This page describes how the Work Queue Manager handles output written 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 the %SYSTEM.Context.WorkMgr class, for example:

```
set sc = $system.Context.WorkMgr().Flush()
```

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

Additionally, 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.

7

Pausing and Resuming a Work Queue

For use with the Work Queue Manager, the %SYSTEM.WorkMgr class provides methods you can use to pause and resume work within a work queue.

For information on halting work completely, see Stopping a Work Queue and Removing Work Items.

7.1 Pausing Work

```
method Pause(timeout As %Integer, ByRef completed As %Boolean = 0) as %Status
```

Prevents the worker jobs associated with this work queue from accepting *additional* items from this work queue. The **Pause()** method also stops any work items that are in progress.

The *timeout* argument represents the amount of time in seconds that the method waits before stopping work items that are in progress. After the timeout period, the method returns the *completed* value, which indicates whether the work items that were in progress when you called the **Pause()** method were completed. Consequently, you can pass in a *timeout* value of 0 to know immediately whether the worker jobs completed all the work items in the work queue.

7.2 Resuming Work

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

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

8

Detaching and Attaching a Work Queue

Typically, when using the Work Queue Manager, you initialize a set of worker jobs, queue work items, and then wait for the worker jobs to complete the work items. However, you may encounter situations where worker jobs are taking longer than expected to complete work items or you cannot dedicate a single process to waiting. Consequently, the Work Queue Manager enables you to detach a work queue from a process and subsequently attach the work queue to the same process or to a different process.

For example, suppose that `queue` references a work queue that you initialized. Also suppose that you added several work items to the work queue. Before calling the **Wait()** or **WaitForComplete()** to determine the status of the work being processed, you could employ the following methods.

8.1 Detaching a Work Queue

```
method Detach(ByRef token As %String, timeout As %Integer=86400) as Status
```

Detaches the work queue object from the object reference that you created when you initialized the work queue. The **Detach()** method enables any work in progress to continue and preserves the current state of the work queue.

The *token* argument represents a secure token that you can use to subsequently attach the work queue to another process. The *timeout* argument is optional and indicates the amount of time in seconds that the system retains the detached work queue object. After the timeout period has elapsed, the system removes any worker jobs and information associated with the work queue. The default value of *timeout* is one day.

After you call the **Detach()** method, most calls on the detached object reference return errors. However, the **NumActiveWorkers()** and **NumWorkers()** methods return `-1`.

8.2 Attaching a Work Queue

```
method Attach(token, ByRef sc As %Status) as WorkMgr
```

Attaches a new object reference to a previously detached work queue object if the work queue object is still in memory. The **Attach()** method returns a new instance of the Work Queue Manager associated with the work queue. You can subsequently call methods on the work queue. For example, you can call the **Wait()** method with a *timeout* value of `0` to determine whether the queue had completed any work items before being detached.

The *token* argument represents the secure token returned by the **Detach()** method that you previously called on the work queue.

8.3 Example

For example, you could detach and then attach a work queue as follows:

```
Set sc=queue.Detach(.token,60)
If $$$ISERR(sc) {
     Return sc
   }
Set queue=$system.WorkMgr.Attach(token,.sc)
If $$$ISERR(sc) {
     Return sc
   }
```

9

Stopping a Work Queue and Removing Work Items

When using the Work Queue Manager, you can stop a work queue, interrupting any work items in progress and removing any queued work items. To do so, call the **Clear()** method of the work queue.

```
method Clear(timeout As %Integer = 5) as %Status
```

Given the timeout period *timeout* (in seconds), this method waits for the worker jobs to finish their current tasks, and then kills the jobs. The system removes and then recreates the work queue, without any work items attached. Afterword, the system returns immediately from **Wait()** or **WaitForComplete()**.

10

Specifying Setup and Teardown Processing

For the Work Queue Manager, 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.

10.1 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, you must call it before calling the **Queue()** or **QueueCallback** method. **Setup()** accepts the following arguments:

work

> The setup code to execute. The supported syntax for this argument is the same as the supported syntax for the *work* argument of the **Queue()** method, which is described in a previous section.

args

> A comma-separated list of arguments for this code. To pass a multidimensional array as an argument, you can precede that argument with a period so that it is passed by reference.

> You should keep the size of the data passed in these arguments relatively small. To provide a large amount of information, you can use a global instead of passing arguments.

10.2 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, you must call it before calling the **Queue()** or **QueueCallback** method.

TearDown() accepts the same arguments as the **Setup()** method. However, the *work* argument specifies the teardown code to execute.

11
About Worker Jobs

This page provides more information on *worker jobs*, the processes that complete units of work for the Work Queue Manager. You can view, manage, and monitor worker jobs like other processes by using the %SYSTEM.Process class. If you need to need to know whether a given process is a worker job, you can call **$system.WorkMgr.IsWorkerJob()** from within the process; that is, you can call the **IsWorkerJob()** method of the %SYSTEM.WorkMgr class.

The Work Queue Manager directs worker jobs using the controller process, which is a dedicated process that performs the several operations:

- Starts up worker jobs

- Manages the number of worker jobs

- Detects and report on halted worker jobs

- Records workload metrics

- Detects inactive work queues

- Deletes work queues

A worker job can be in any of the following states:

- Waiting to attach to a work queue

- Waiting for units of work. A worker job can be in this state for only a short period of time before it is released.

- Active. A worker job is active only when it is making forward process while executing a unit of work.

- Blocked by a lock or event while processing a unit of work. A worker job that is blocked is not active. If a worker becomes blocked and there is additional work in the work queue, the Work Queue Manager may activate a retired worker or start up a new worker. When a worker job is no longer blocked, the number of active workers may exceed the maximum number of active workers specified for the work queue. If this occurs, the controller process retires the next worker that completes a unit of work. Consequently, there may be short periods of time when the active number of worker jobs exceeds the maximum number of worker jobs specified for a given work queue.

- Retired and available to be activated rapidly.

Unused worker jobs remain available for use by other Work Queue Manager queues for a short period of time. The timeout period is subject to change and is deliberately not documented. After the timeout period expires, the worker is removed.

If a worker job is actively processing a work item for a queue that has been deleted or cleared, the system waits a very short period of time before issuing an EXTERNAL INTERRUPT error. If the worker job continues processing after the error, the system waits for the number of seconds specified in the DeleteTimeout property before forcibly terminating the worker and starting up a new worker to process the unit of work.

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.