Job Scripting Operators

Synopsis

Tractor and Alfred share a common scripting language. Input scripts for both are composed of expressions using the following operators:

```
Job [options]

Task {title} [options]

RemoteCmd {launch_expr} [options]

Cmd {launch_expr} [options]

Instance {task_title_to_reference}

Iterate varname -from n -to m -by i -template {script} [options]
```

See the Scripting document for the general concepts and examples.

Operator Details

The *options* listed below have the form "*-optionName optionValue*" and all options must be on the same logical input line as the operator. Backslash (\) can be used at the end of a line to explicitly escape the newline and continue input on the next line of the script file. Newlines within (curly) braces are treated as blanks and can improve clarity, especially for nested tree structures.

For example:

```
Task {frame.1} -subtasks {} -cmds {Cmd {/bin/date} -tags {simple}}
```

can also be written as:

```
Task {frame.1} -subtasks {
} -cmds {
   Cmd {/bin/date} -tags {simple}
}
```

Job [options]

-title {*job title text*}

this string is used as the label text in the job-queue window.

-after {*date*}

delays the start of job processing until the given time; the job is spooled and placed on the queue, but no processing occurs until after the date specified; dates are of the form *{month day hour:minute}* or for times later on the same day *{hour:minute}* (up until midnight). Hours should be given as for a 24-hour clock. For example to delay a job until June 23 at 1:45PM, use: Job -after {6 23 13:45} ...

-afterjids {*jobid jobid ...*}

delays the start of job processing until the specified jobs have completed; multiple job ids must be space-separated. e.g. -afterjids {123 124 130}

-subtasks {*Tasks or Instances*}

the nested **Tasks** that define the structure of the job are defined here.

-cleanup {*Cmd list*}

a list of Cmds to execute on job termination.

-atleast *min_slots*

sets the default value for RemoteCmds without an explicit *-atleast* option.

-atmost *max_slots*

sets the default value for RemoteCmds without an explicit *-atmost* option.

-maxactive *max_commands*

specifies that the job will be allowed to have at most *max_commands* concurrent active commands.

-tags {*tag tag ...*}

limit-tags to be added to each RemoteCmd's existing tag list.

-projects {*name name ...*}

Names of project affiliations for this job, to be matched with *Share Names* in the site-wide limit sharing definitions in limit.config. Typically this the name of the show or department (or its internal nickname) for which the job was spooled.

-tier {*name*}

The list of valid site-wide tiers are defined in tractor.config, with each tier representing a particular global job priority and scheduling discipline. By default, jobs are spooled into the tier named "default" and may be moved to other tiers later by administrators.

-service {*key expression*}

additional service key expressions to be ANDed with each RemoteCmd's existing service expression.

-envkey {*key*}
-envkey {{*key1*} {*key2*} {*key3*}}

Specifies an arbitrary key (a name or number) that will be passed to the remote tractor-blade. This Job option is applied as the default RemoteCmd -envkey option, when individual RemoteCmds do not specify their own per-command value. See additional details in the RemoteCmd -envkey option description, below.

-priority *float*

specifies a sorting criteria for jobs on the central job queue. Jobs with a larger priority, numerically, are considerd before jobs with a lower value. Ties are broken according to the current dispatching policy, but typically spool time is considered, so that jobs spooled earlier are dispatched than those added to the queue later, i.e. a FIFO order is the default. Note that the old Alfred "-pbias" parameter is translated directly to a simple priority in Tractor.

-crews {*crew name(s)*}

specifies the list of Crews to be used when determining remote server access; applies to users who are members of multiple crews but who want a particular job to execute with a *more restricted* set of server access permissions. Crews listed here but that don't apply to the user are ignored. If the list is empty, then the default list is derived from the crews.config and blade.config.

-avoid {*service names...*}

this is an alternate way to specify a list of hostnames or other service key types that should be excluded from consideration during task assignment. The list of keys is appended to each RemoteCmd's own -service expression as a logical "AND NOT (keys)" clause.

-etalevel *n*

specifies that level *n* of the Task tree depth hierarchy should be used to compute the job's estimated time-to-completion value. The outermost level of Tasks in the job script are level zero, their subtasks are level one, etc. The default estimate is based on the number of remaining level 0 nodes multiplied by the average time required by each completed level 0 node (i.e. all its subtasks and its own Cmds).

-comment {*text*}

specifies an arbitrary string that is stored with the job, it is primarily used by job-generator programs to record their version information and/or the context under which the job was created (user name, project info).

-metadata {*text*}

specifies an arbitrary string that is stored with the job. The metadata contents are studio dependent and ignored by Tractor itself. The values may be used in tq searches or in Dashboard job list filters.

-editpolicy {*policy_name*}

specifies the name of one of the JobEditAccessPolicies defined in crews.config, the policy named "defaultPolicy" is used by default. Note that, in a concession to the realities of production wrangling, if a job specifies a policy name here that has not yet been defined in crews.config, then a fallback to "defaultPolicy" occurs for that job, until the policy has been defined and crews.config is reloaded by the engine. In all cases, users in the "Administrators" crew have full job edit access.

-dirmaps {*{{srcdir} {dstdir} zone} ...*}

specifies the mapping of one path to another path under a given zone. Details can be found here.

-postscript {*cmd_list*}

-whendone {*shell_command*}

-whenerror {*shell_command*}

The optional "-postscript" clause can be added to the main "Job" keyword in a job script. This clause adds Cmds or RemoteCmds that will be executed *after* those in the main job proper, in the following situations: (1) after the main job tasks have all completed successfully; and (2) after the main job stalled due to errors, when all other remaining tasks are blocked by errors in subtasks. These commands are similar in spirit to constructs like the "finally" block in a Python exception handler. This clause is structured just like the "-cmds" block in task:

```
Job -title {...} -subtasks {
    Task {...} -cmds {
    }
} -postscript {
   Cmd {touch /tmp/always_runs_on_spooling_host} -when always
   RemoteCmd {touch /tmp/there_were_errors} -when error
   RemoteCmd {touch /tmp/all_done_with_no_errors} -when done
}
```

The old alfred scripting constructs "-whendone' and "-whenerror" are implemented in Tractor as special cases of the -postscript clause. For compatibility reasons they are translated to *local* Cmds (requiring a tractor-blade on the user's spooling host), unless the tractor-spool option "--remotecleankeys=kkkk" is given. These commands launch the given "when" command via a "system shell" in order to accommodate shell expressions allowed by the old syntax. The newer postscript syntax, with "RemoteCmd {...} -when {...}" is recommended instead.

Task {title} [options]

`title`

this string is used both as label text in the user-interface and as the name by which Instance nodes refer to their targets.

`-subtasks` {*`Tasks or Instances`*}

tasks are arranged in a grouping hierarchy; child tasks are defined in the current task's subtask list, these are dependencies that must be completed before the current task can proceed.

`-cmds` {*`Cmd list`*}

An optional list of RemoteCmds associated with this Task node. The commands are launched in sequence (as resources become available) after all of this Task's subtasks are completed successfully. If a Task is simply a grouping node in the task hierarchy then it does not need to specify any commands.

`-cleanup` {*`Cmd list`*}

clean-up Cmd operators for this node; these commands are executed after those in the *-cmds* section (if any), or during job deletion if this node was on an active path (some child had dispatched a command); they also run during task-restart if this task or one of its parents had a Task-level (shared) server checked out.

`-chaser` {launch_expr}

an external application launched by users from the UI, typically to view a rendered frame; the UI activates the launching button only after the regular commands for this task have completed successfully. When a web browser is connected, there is special handling of chaser commands. Specifically, the named "imagefile" will be made available for downloading to the browser. See the site configuration file for details on enabling this feature (httpImgRoot).

`-preview` {launch_expr}

launches an external application in the same manner as **-chaser**, however this version activates the launching button before the commands have completed.

`-service` {*`server keys`*}

(Advanced) Under some unusual circumstances it can be useful to check-out a remote server for the duration of all of the subtasks and commands within a Task's hierarchical scope. A service specification *at the Task level* accomplishes this; the blade assignment is checked back in when the Task completes or when there are no more references to the Task's associated *id.* NOTE: it is far more typical to specify the -service as a RemoteCmd option.

`-serialsubtasks` *`1`*

This option alters the parallel execution strategy applied to this task's dependent child tasks, i.e. those specified within this task's "-subtasks {...}" block. Under normal circumstances it is desirable to allow tasks that are siblings to execute in parallel; that is: a task and all of its subtasks are presumed to be independent of its siblings, and thus the sub-trees rooted in the siblings can be dispatched in parallel. The "-serialsubtasks" option allows this scheme to be changed such that the top-level sibling tasks in a "-subtasks {...}" block are treated in a chain-like sense: each task waits for its prior sibling to complete before evaluating its own deeper subtasks and commands. This scheduling variant can simplify the scripting of jobs that have distinct sequential "chapters" or phases of actions; such as "prepare," "render," and "clean-up" phases, each of which consists of several subtasks. This sort of thing can also be accomplished with elaborate *Instance* node references, or sometimes with deep linear nesting of single "phase" tasks with lots of *Cmds*. The serialsubtasks serialization is only applied to the immediate child tasks in this task's "-subtasks {...}" block; nested child tasks below them are scheduled in parallel as usual, unless another "-serialsubtasks" is specified farther down the hierarchy.

`-id` {*`name`*}

an identifier for a task; used by Cmds for runtime substitution of task-level (shared) remote server names; the id is valid in the scope of this Task (i.e. its subtasks and commands).

`-resumeblock` *`1`*

This option works together with the RemoteCmd option -resumewhile (below) to implement a simple form of incremental checkpoint-resume cycle within a job. A Task with this option marks the "top" of a looping scope, namely the task-tree hierachy represented by this task's -subtask dependencies (and their dependencies).

If any RemoteCmd below this Task encounters a True "resumewhile" condition when its command exits, then all Tasks along the path from the RemoteCmd to this "resumeblock" Task will be automatically relaunched repeatedly until all "while" conditions are False.

The goal of this scheme is to allow a series of related commands to execute to a sensible checkpoint condition, likely leaving a reviewable intermediate result. Then the entire chain of tasks is executed again, applying additional incremental work to the intermediate result.

RemoteCmd {appname [app_args]} -service {expr} [options]
RemoteCmd specifies the name of a command-line program that Tractor will launch on some tractor-blade host on the compute farm. The "-service" expression specifies the host types that are appropriate to run the given command. Other options include environment configuration keys, and tracking controls for scarce resources. For example:

```
RemoteCmd {prman /shared/rib/some.rib} -service PixarRender
```

The prman executable will be launched on a Tractor blade server that *provides* the PixarRender service. In simple terms that means that a command will be dispatched when its service keyword matches an available blade.

These keys can be chosen arbitrarily by site administrators to represent, abstractly, capabilities of different types of hosts; they might represent installed software, or operating systems, or CPU architecture or graphics hardware. Each category of tractor blade *profile* lists different combinations of these keywords. Each job, or individual command in a job, then lists the keys that must be present on a host for it to run correctly. By convention, the "PixarRender" keyword is used to describe hosts where the Pixar RenderMan software is installed.

Parameter details:

appname

the name of an application to launch, and its arguments. See the launch expression discussion for details on runtime substitution variables and restrictions (especially with regard to shell-style command pipelines). This must be a properly tokenized and escaped Tcl list; a blank-delimeted list is sufficient for simple commands, but in general use nested curly-braces to clearly distinguish tokens that may contain spaces, etc. For example:

```
RemoteCmd {/bin/ps -eaf} -service Linux
RemoteCmd {{C:\\app path\\appname} {arg1} {arg two}} -service Windows
```

See the parameter substitution list for run-time values that tractor-blade will add before launching the command. For commands that are participating in a *re sume block* cycle, the substitution pattern "%r" is particularly useful since it will be "1" when a computation is resumed from a checkpoint. Similarly the pattern "%R" will be an incrementing integer giving the current continuation pass count, starting with 0. The pattern %q will have the value 1.0 when a task is being executed due to final quality runs from the subtasks it depends upon, it will be less than 1.0 if some subtask only reached a checkpoint during this resumewhile loop.

-service {*server keys*}

specifies a keyword search expression used to associate a remote server with this command; see the service keys discussion for syntax. The server lookup is done by the tractor-engine using the service and access definitions defined in the blade.config file. Command launching blocks until an appropriate server becomes available and is checked out. The server is checked back in when the launched application program exits, such as when a render is complete.

-tags {*limit tag list*}

specifies a list of words (blank-delimited limit names) that will be tallied by the tractor-engine. Limit tag tallies are compared against the limit constraints defined in limits.config to determine whether the command can be launched at this time, or whether it must wait for the tallies to fall below the constraint limits after other commands using the same tags complete. Limit tags are frequently used to ensure that available license counts are not exceeded by concurrent launches of a given app.

-maxrunsecs {*float_maximum_allowed_runtime*}
-minrunsecs {*float_minimum_acceptable_runtime*}

Task Elapsed Time Bounds -- specifies an acceptable elapsed time range, in seconds, for a given launched command. Commands with elapsed times outside the acceptable range will be marked as an error. Commands that run past the maximum time boundary will be killed. Both parameters are optional. The default value for minrunsecs is 0.0, and the default for maxrunsecs is unlimited. Example job script syntax:

```
RemoteCmd {sleep 15} -service PixarRender -minrunsecs 5 -maxrunsecs 20
```

-atleast *min_slots*

blade matching criterion, specifying the minimum number of free "slots" that must be available on a tractor-blade in order to launch this command. The blades are divided into abstract slots by administrators (in blade.config), and every command launch consumes one or more slots. Slots may represent CPUs or memory or a combination of resources, as defined by each studio. Job authoring tools should follow matching conventions when specifying the command utilization "footprint" with this option and "-atmost" below.

-atmost *max_slots*

blade matching criterion, used with "-atleast" above, specifying the maximum number of free "slots" that this command will consume when launched. The blades are divided into abstract slots by administrators (in blade.config), and every command launch consumes one or more slots. Slots may represent CPUs or memory or a combination of resources, as defined by each studio. Job authoring tools should follow matching conventions when specifying the command utilization "footprint" with this option and "-atleast" above. The two parameters can be set to the same value.

-envkey {*key*}
-envkey {{*key1*} {*key2*} {*key3*}}

specifies an arbitrary key or keys (a name or list of names or other strings) that will be passed to tractor-blade. If this option is not specified, then the command inherits the settings given at the Job level. Based on each envkey string, and other properties such as user name, tractor-blade will invoke a series of *environment handlers* that prepare environment variables such as paths and other settings prior to launching each command. There are several built-in key handlers, and custom handlers can also be added. The handlers available for each type of blade are specified with the "EnvHandler" settings in blade.config. One usage scenario addresses the common requirement for a given production to lock down a variety of paths and other pipeline env settings. These settings can be neatly encapsulated in a single handler definition, and then each job simply specifies -envkey=NAME to have the given collection of settings applied to its commands. Since a single tractor-blade may launch commands from many different jobs, sometimes concurrently, and each job may require unique env settings for its show, the blade applies the env handler set-up as part of each command's launch procedure.

A special key of setenv can be used to set specific environment variables. For example, {setenv SRC=/src/path DST=/dst/path} could be used in concert with a command like cp $SRC $DST.

A command-level envkey setting causes the job-level envkey setting to be ignored.

-id {*name*}

an identifier name for the current command, used by other subsequent RemoteCmds in the job to request execution on the same blade host as the current command, rather than on some other blade that may become available sooner. See the discussion of remote server names substitution for examples of using "-id" and the matching -refersto option.

-refersto *idref*

a command can request reuse of the same blade as another command by referring to that other commands "id" name, rather than specifying its own "-service" selection criteria. In some unusual cases the given id refers to a Task that will carry the blade name bound by the first resolved command that references it (see the shared servers discussion).

-expand 1/0

setting the option value to 1 indicates that this command produces *new Tractor task descriptions* as its output. This is a mechanism for growing the job task hierarchy dynamically during the execution of the job. Everything printed to stdout by the launched app will be collected when the command completes, and will be assumed to be syntactically valid Task descriptions. Those new tasks will become new subtasks of the current Task. This feature has primarily used by processing pipelines that prefer a just-in-time style of processing where commands procedurally generate assets based on prior results, rather than "unrolling" all processing steps at job authoring time.

Tractor 1.7 also supports setting the value to a filename (string). In this case the running application is expected to coded or parameterized to write the task descriptions into the named file, rather than writing them to stdout. Then tractor-blade will send the file contents to the engine for insertion into the parent job. After delivering the contents, tractor-blade will remove the file. Note that since several such commands may be running simultaneously on a given blade, it is important to chose unique names for the temporary expand file.

Tractor 2.1 also introduced an expand variant intended for use by long-running applications that generate a series of results, such as simulations. In this situation, the application might want to inject new tasks that can run *in parallel* with the ongoing application itself -- unlike the other expand use cases where the expand text is sent after the application has fully exited. To use this concurrent approach, the application should write the new task descriptions to a temporary file; then it should emit a line of the form:

```
TR_EXPAND_CHUNK "filename"
```

to stdout. Note the important required double-quotes around the filename. Tractor-blade will detect that line and then deliver the contents of that file to the engine. Once the delivery occurs, then tractor-blade will remove the temporary file.

-msg {*text_for_app_stdin*}

a text string to be piped to stdin of the launched app, usually representing a message requesting some action. See the launch expression discussion for an example using a system shell.

-retryrc {*integer_return_codes*}

When specified, the value(s) specified will be compared to the the numeric exit status from the executed command. If a match occurs, then the command will be reset to the "Ready" state, and will be dispatched to the next available matching blade. The idea is to provide a way to work around failures that are known to be transient. See the "CmdAutoRetryAttempts" setting in tractor.config for a more global mechanism that affects all commands that exit with a non-zero status.

-resumewhile {*condition*}

This option is part of mechanism for implementing a simple checkpoint-resume form of incremental processing. It is only meaningful when combined with the Task option "-resumeblock" and with an application that is compliant with this continuation model.

When this option is given, then the command being launched is expected to *exit* after performing an *incremental* amount of work. This option requests that Tractor should relaunch the same command repeatedly until the command determines that it has finally completed its overall work. Stated another way: the command will consume a fixed amount of CPU time and then *yield* the blade to another command with the expectation that Tractor will eventually relaunch it so that it can resume processing. This simple continuation mechanism requires several cooperating components:

The *condition* expression is used by tractor-blade to determine whether the command process exit represents an intermediate checkpoint, or the final completion of the particular task. There are two supported expressions:

- {exitcode NNN} -- the keyword "exitcode" causes tractor-blade to examine the process exit status code. If the numeric code *matches* the given value, then a checkpoint has been reached, and the command will be relaunched later. This condition can be read as "resume this command while the exitcode is NNN", otherwise the command should be considered finally complete if the exit code is zero, or to have encountered an error for any other value.
- {testcheckpoint TYPE VALUE} -- the keyword *testcheckpoint* causes tractor-blade to perform one of several built-in tests for command completion. Today, the only supported TYPE is the keyword "exr" referring to the OpenEXR image format, and a VALUE which is an output image filename. Tractor-blade will examine the named image to see if it contains auxilliary data indicating that it is an incomplete checkpoint image. If the image is a checkpoint, then Tractor will schedule another incremental execution of the command. If the image is not a checkpoint image then the rendering is considered to be complete, or an error depending on the process exit status.

-resumepin 1

controls whether a "resumewhile" continuation must occur on the blade as the initial pass; that is: is this series of continuations "pinned" to the same blade? Default is "0" meaning that a resume can occur on any available blade (with matching service criteria).

Cmd {appname [app args]} [options]

This operator, also referred to as "local Cmd" is only used in a few restricted cases, and it accepts the same options as RemoteCmd, above. There are two important differences:

- The specified application is constrained to *execute on the same user desktop host that spooled the job.* This may be valuable in the case where the command presents a UI or displays an image, or if it must access files that are only on that machine. Therefore, the user's desktop machine must *also be running a tractor-blade* so that tractor-engine can dispatch the command back to it.
- In the case of *netrender* (or rare applications like it), Tractor can collect several available remote blades to be used in concert with the user's desktop local blade. See the "-atleast" description below.

-atleast {*min_servers*}
-atmost {*max_servers*}

In the unusual case of launching the Pixar *netrender* client program on a user's local desktop, the netrender command must be given additional tractor-blade names on its own command line. The atleast/atmost variants on *local* Cmds ask Tractor to assign those additional blades. The netrender client launch will occur when at least *min_servers* remote servers have been assigned, and no more than *max_servers*. These options on a local Cmd are only meaningful if -service is also given to specify the type of blade that should be bound. See the "%h" and "%H" substitution patterns for a way to substitute the as-bound hostnames into the commandline argument list.

`-samehost` *`1`*

constrains a multislot checkout (using atleast/atmost) to slots that are all on the same physical host. This is a requirement for multi-threaded applications, since threads are part of a single process. See the "%n" substitution pattern for a useful mechanism for inserting the actual number of found slots into the application's argument list.

Instance {taskref}
`taskref`

the name by which an Instance node refers to its target. Instances are essentially pointers to Tasks elsewhere in the tree, and they function as Tasks in the dependency scheme, i.e. if a Task contains an Instance in its subtask list, it will block until the real Task pointed to by the Instance is complete. Instances are most often used to indicate a shared prerequisite, for example several independent/parallel frame renderings may depend on the prior creation of an environment map. The referenced Task is identified by its title string, which must exactly match the string given here.

Assign varname {value_string}

NOTE: the Assign operator is deprecated and will not function in Tractor as it did in Alfred. Specifically the stored values will not be available in future "expand" tasks, and the job parsing and substitution modes can effect whether substitions occur at all. Variable substituion of this type is usually better suited to the scripting environment that is creating the job script, such as the Tractor job authoring API for Python.

`varname`

name of a job global variable that will be referenced in the job Tasks

Iterate varname -from n -to m -by i -template {script} [options]

`varname`

the name of a variable that will be iterated during the course of the job.

`-from` *`nnn`*

initial value of the iterated variable. May be an integer or real number.

`-to` *`nnn`*

final value of the iterated variable (note: this is inclusive; the iterate will continue to run while the variable is less than, or equal to, this value. If the *to* is less than the *from* value then a decrementing iteration is expected and the termination test will "greater than or equal to"). May be an integer or real number.

`-by` *`nnn`*

increment to be applied to the iterated variable, may be an integer or real number. The special notation -by binary causes the variable to cycle through its values using a non-linear, so-called "binary" sequence, which is useful for producing animation frames in progressive detail.

`-subtasks` {*`Tasks or Instances`*}

As with Tasks, the subtasks are dependencies that must be completed before the Iterate can proceed with its first cycle.

`-template` {*`Tasks or Instances`*}

Defines the a block of script that will be replicated every time the Iterate operator cycles. On every cycle the current value of *varname* will be substituted wherever $varname occurs in the template script, then the resulting subscript is inserted into the job tree just before the Iterate node's current position.