Inca Tool Documentation

Release 0.4.0

Lee Keitel

September 22, 2016

Contents

1 Task Files **3**
- 1.1 What are they? 3
- 1.2 File Extension 3
- 1.3 Syntax 3
- 1.4 Task File Structure 4
- 1.5 Task File Example 7

2 Inventory File **9**
- 2.1 Multiple Inventory Files 10

Contents:

CHAPTER 1

Task Files

1.1 What are they?

Task files are make up the core of Inca Tool. A task file is used to manage a particular job. A task file is meant to perform a single task. However, multiple tasks files may be ran at once if needed. Task files use a custom syntax that provides metadata for the task, various parameters, and the commands to execute for the task.

1.2 File Extension

Although the file extension doesn't matter, it's recommended to use `.itf` which stands for Inca Task File.

1.3 Syntax

The syntax of task files can be broken up into three types: key: value, simple list, and extended list.

1.3.1 Comments

Comments must be on their own line and begin with a pound sign `#`. Comments may not be on the same line as data.

1.3.2 Key Value

Most settings are simple key value pairs which take the form:

```
key: value
```

The space before the value is not required but is there for readability.

1.3.3 Simple List

A list is defined as:

```
key:
    value1
    value2
```

Note that the values are indented and have the same indention. For both types of lists, indention matters. If two values have a different indention, or if one uses tabs and the other spaces, the task file will return an error when it's parsed. Be sure to use consistent indention.

1.3.4 Extended List

And extended list is a simple list but can take a main value as well as key value pairs as settings for the list:

```
key: value setting=val setting2=val2
    list value1
    list value2
```

Currently, setting values cannot contain a space. Although it may seem strange at first, this is can be very powerful for creating complex task files.

1.3.5 User Defined Variables

Custom variables can be declared in a task file and then used with commands. This can be very helpful to avoid duplicate data and can provide a clean, easy way to update data later on. The format is simply a modified key value pair:

```
$key: value
```

To use the value in a command block, simply use the syntax `{{key}}`. Note, there's no dollar sign ($) when using the key, only when setting.

1.3.6 Included Files

Other files may be included into a task file. The included file is parsed as if it were part of the parent task file at the exact place it's included. This can be useful for creating command blocks to share amoung multiple task files. Here's the syntax:

```
@path/to/included/file.itf
```

Included files are relative to their respective parent task file.

1.4 Task File Structure

A task file is made up of several parts. Settings are optional unless marked otherwise.

1.4.1 Metadata

Metadata does not affect how the task is ran, but can be used to ensure the correct task was run and can serve as documentation. All metadata are simple key value pairs. Here's a list of all available metadata settings:

- name
- description
- author
- date
- version

1.4.2 Job Settings

There are a couple of settings that affect how templates are generated and ran.

- **concurrent**
  - Type: key-value integer
  - Default: 300
  - Valid values: Any integer
  - **Description:**
    - The number of devices that can be configured concurrently. The higher this number, the more file descriptors are needed to run the job. Setting this to 0 means no limit.
- **template**
  - Type: key-value string
  - Default: expect
  - Valid values: expect, bash
  - **Description:**
    - The template to use when generating a job file.
- **prompt**
  - Type key-value string
  - Default: #
  - Valid values: Any string
  - **Description:**
    - The unique part of a prompt to wait for when using Expect.
- **default command block**
  - Type key-value string
  - Default: Empty string
  - Valid values: Any command block name
  - **Description:**
    - This controls which command block acts as the entry point into the task. By default a nameless block will be used. Generally this setting should be used but is made available for customization.

1.4.3 Inventory

For a task to run, Inca Tool needs to know which devices should be configured. This is where the inventory file and device filter come in.

- **inventory**
  - Type: key-value string
  - Default: devices.conf
  - Valid values: File system path

  - **Description:**
    - The path to the inventory file. It may be absolute or relative. The path will be relative to the current working directory. NOTE: It is recommended to provide the inventory file via the -i cli flag. If this setting is used in a task file, it will override the file given on the command line.

- **devices**
  - Required
  - Type: simple list
  - Default: Empty
  - Valid values: group or device names
  - **Description:**
    - This list contains the group or devices names that will configured with the task. If a group or name doesn't exist in the provided inventory file, an error will be given.

1.4.4 Command Blocks

Command blocks are where the set of commands are defined that will be ran on the client device. Multiple command blocks may be created so long as they have different names. One command block must be named whatever `default command block` is set to. By default this is a nameless block. Names cannot contain an equal sign or space. This is the block that will be used as the entry point into the task. Other blocks can be included using the `_c` syntax described below.

Block Syntax:

```
commands: name setting=value
    command 1
    command 2
    _c other-command-block
    _b builtin-command-block
    _s /path/to/script
```

For a nameless block simply omit the name, settings can still be used as normal.

Command Block Settings

- **type**
  - Default: expect
  - Valid Values: expect, raw
  - **Description:**
    - Determines any extra processing needed for the block. Expect will encapsulate the commands in a `send` and add a corresponding `expect` command.

Special Command Syntax

There are a few special command prefixes that change how the command block is parsed and even how the job is ran.

- `_c foobar` - Inline a command block named foobar

- `_s foobar.sh -a arg1 arg2` - Immediately execute the file named foobar.sh. This stops all parsing and immediately executes the file. When the file is done executing, the job is complete. All other command lines are ignored.
- `_b foo` - This functions the same as `_c` but can only be used with builtin command block. Inca Tool has a few builtin command blocks for common functions on Juniper and Cisco devices. A list of block names can be found below.

Builtin Command Blocks

- `juniper-configure` - Enter Juniper's configure mode.
- `juniper-exit-nocommit` - Exits from the Juniper configure mode and if requested will exit without committing changes.
- `juniper-commit-rollback-failed` - Attempt to commit changes on a Juniper device and rollback if the commit fails. The script as a whole will fail for that device and an error will be show to the console.
- `cisco-enable-mode` - Enter Cisco's Enable exec mode.
- `cisco-end-wrmem` - Exit a Cisco's configure terminal mode and save the running configuration.

1.5 Task File Example

This is a minimal example that uses all the default settings and adds remote logging to a Cisco device:

```
# Metadata - Doesn't really matter, for information purposes
name: Cisco Logging
description: Add switch logging to host 10.0.0.1
author: John Doe
date: 10/27/2015
version: 1.0.0

devices:
    group1
    device2

commands: main
    _b cisco-enable-mode
    set logging 10.0.0.1
    _b cisco-end-wrmem
```

CHAPTER 2

Inventory File

The inventory file is the list of all devices that a task file could possibly run against. They are separated into groups which can then be specified in a task file to run against. An inventory file can be set in the task file itself, or on the command line using the *-i* flag. Using the cli flag is recommended.

- All devices must be in a group.
- The global group may contain settings that apply to all devices unless overridden by the device.
- The global group cannot contain devices.
- Device names cannot contain a space.
- Group names may contain numbers, letters, underscores, hyphens and spaces.
- **If multiple groups are declared with the same name, the devices will be appended to a single group.**
  - Example: The following will result in a single group named "group1" with devices device1, device2, device3, and device4:

```
[group1]
device1
device2

[group1]
device3
device4
```

- Devices may be in multiple groups. Any device settings must be declared on the first declaration.
- Settings are "key=value" pairs separated by a space on the same line as the device name. If a setting value contains a space, it must be enclosed in double quotes.
- Both devices and groups may have settings
- Order of setting precedence is Global -> Task -> Group -> Device
- **Available settings:**
  - remote_user - Defaults to "root"
  - remote_password - Defaults to ""
  - cisco_enable - Defaults to remote_password
  - protocol - Defaults to "ssh"
  - address - Defaults to device name

Example:

```
[global]
remote_user = user
remote_password = pass

[server room]
Server_Switch_1 address=10.0.0.1
Switch2.example.com

[building 1] remote_user="jarvis"
Building1_1 address=10.0.0.2 protocol=telnet
Builsing1_2 address=10.0.0.3 remote_password="chicken feet"

[all switches]
Server_Switch_1
Switch2.example.com
Building1_1
Building1_2
```

2.1 Multiple Inventory Files

Inventories can be separated into multiple files and then brought together at run time. There two ways to do this. The first is by including each file individually. The second is to use the output of an executable file and add it to the inventory. Both methods simply replace the include line in the parent file with the text from the included file itself or from the standard output of the executable. The purpose of includes is to provide a way to separate the different parts of a network/system and break them into manageable chunks.

2.1.1 Example using normal file includes

Example root inventory file:

```
[global]
remote_user = user
remote_password = pass

@devices/server_room_1.conf
@devices/server_room_2.conf
```

devices/server_room_1.conf:

```
[server room 1]
webserver1.example.com
webserver2.example.com
```

devices/server_room_2.conf:

```
[server room 2]
db1.example.com
db2.example.com
```

When compiled, it will turn into this:

```
[global]
remote_user = user
remote_password = pass
```

```
[server room 1]
webserver1.example.com
webserver2.example.com

[server room 2]
db1.example.com
db2.example.com
```

2.1.2 Example using script include

Say we have a directory that contains ".conf" files containing our inventory. We can create a script that will dynamically concatenate the files together. This way we don't need to specify each file by hand.

Example root inventory file:

```
@!devices/compile.sh
```

devices/compile.sh:

```
#!/bin/sh
for f in *.conf; do
    cat $f
done
```

When ran, it could generate something like the example file above. Again, this would make it so when a new ".conf" file is created in the directory, it would be picked up automatically on the next run. This can be very helpful for dynamic environments.