Android Init Language

The Android Init Language consists of five broad classes of statements: Actions, Commands, Services, Options, and Imports.

All of these are line-oriented, consisting of tokens separated by whitespace. The c-style backslash escapes may be used to insert whitespace into a token. Double quotes may also be used to prevent whitespace from breaking text into multiple tokens. The backslash, when it is the last character on a line, may be used for line-folding.

Lines which start with a # (leading whitespace allowed) are comments.

Actions and Services implicitly declare a new section. All commands or options belong to the section most recently declared. Commands or options before the first section are ignored.

Services have unique names. If a second Service is defined with the same name as an existing one, it is ignored and an error message is logged.

Init .rc Files

The init language is used in plain text files that take the .rc file extension. There are typically multiple of these in multiple locations on the system, described below.

/init.rc is the primary .rc file and is loaded by the init executable at the beginning of its execution. It is responsible for the initial set up of the system.

Devices that mount /system, /vendor through the first stage mount mechanism load all of the files contained within the /{system,vendor,odm}/etc/init/ directories immediately after loading the primary /init.rc. This is explained in more details in the Imports section of this file.

Legacy devices without the first stage mount mechanism do the following:

1. /init.rc imports /init.${ro.hardware}.rc which is the primary vendor supplied .rc file.
2. During the mount_all command, the init executable loads all of the files contained within the /{system,vendor,odm}/etc/init/ directories. These directories are intended for all Actions and Services used after file system mounting.

One may specify paths in the mount_all command line to have it import .rc files at the specified paths instead of the default ones listed above. This is primarily for supporting factory mode and other non-standard boot modes. The three default paths should be used for the normal boot process.

The intention of these directories is:

1. /system/etc/init/ is for core system items such as SurfaceFlinger, MediaService, and logcatd.
2. /vendor/etc/init/ is for SoC vendor items such as actions or daemons needed for core SoC functionality.
3. /odm/etc/init/ is for device manufacturer items such as actions or daemons needed for motion sensor or other peripheral functionality.

All services whose binaries reside on the system, vendor, or odm partitions should have their service entries placed into a corresponding init .rc file, located in the /etc/init/ directory of the partition where

they reside. There is a build system macro, LOCAL_INIT_RC, that handles this for developers. Each init .rc file should additionally contain any actions associated with its service.

An example is the logcatd.rc and Android.mk files located in the system/core/logcat directory. The LOCAL_INIT_RC macro in the Android.mk file places logcatd.rc in /system/etc/init/ during the build process. Init loads logcatd.rc during the mount_all command and allows the service to be run and the action to be queued when appropriate.

This break up of init .rc files according to their daemon is preferred to the previously used monolithic init .rc files. This approach ensures that the only service entries that init reads and the only actions that init performs correspond to services whose binaries are in fact present on the file system, which was not the case with the monolithic init .rc files. This additionally will aid in merge conflict resolution when multiple services are added to the system, as each one will go into a separate file.

There are two options “early” and “late” in mount_all command which can be set after optional paths. With “--early” set, the init executable will skip mounting entries with “latemount” flag and triggering fs encryption state event. With “--late” set, init executable will only mount entries with “latemount” flag but skip importing rc files. By default, no option is set, and mount_all will process all entries in the given fstab.

Actions

Actions are named sequences of commands. Actions have a trigger which is used to determine when the action is executed. When an event occurs which matches an action's trigger, that action is added to the tail of a to-be-executed queue (unless it is already on the queue).

Each action in the queue is dequeued in sequence and each command in that action is executed in sequence. Init handles other activities (device creation/destruction, property setting, process restarting) “between” the execution of the commands in activities.

Actions take the form of:

```
on <trigger> [&& <trigger>]*
   <command>
   <command>
   <command>
```

Actions are added to the queue and executed based on the order that the file that contains them was parsed (see the Imports section), then sequentially within an individual file.

For example if a file contains:

```
on boot
   setprop a 1
   setprop b 2

on boot && property:true=true
   setprop c 1
   setprop d 2

on boot
   setprop e 1
   setprop f 2
```

Then when the `boot` trigger occurs and assuming the property `true` equals `true`, then the order of the commands executed will be:

```
setprop a 1
setprop b 2
setprop c 1
setprop d 2
setprop e 1
setprop f 2
```

Services

Services are programs which init launches and (optionally) restarts when they exit. Services take the form of:

```
service <name> <pathname> [ <argument> ]*
   <option>
   <option>
   ...
```

Options

Options are modifiers to services. They affect how and when init runs the service.

`console [<console>]`

> This service needs a console. The optional second parameter chooses a specific console instead of the default. The default "/dev/console" can be changed by setting the "androidboot.console" kernel parameter. In all cases the leading "/dev/" should be omitted, so "/dev/tty0" would be specified as just "console tty0".

`critical`

> This is a device-critical service. If it exits more than four times in four minutes, the device will reboot into recovery mode.

`disabled`

> This service will not automatically start with its class. It must be explicitly started by name.

`setenv <name> <value>`

> Set the environment variable *name* to *value* in the launched process.

`socket <name> <type> <perm> [ <user> [ <group> [ <seclabel> ] ] ]`

> Create a unix domain socket named /dev/socket/*name* and pass its fd to the launched process. *type* must be “dgram”, “stream” or “seqpacket”. User and group default to 0. ‘seclabel’ is the SELinux security context for the socket. It defaults to the service security context, as specified by seclabel or computed based on the service executable file security context. For native executables see libcutils android_get_control_socket().

`file <path> <type>`

> Open a file path and pass its fd to the launched process. *type* must be “r”, “w” or “rw”. For native executables see libcutils android_get_control_file().

`user <username>`

> Change to ‘username’ before exec'ing this service. Currently defaults to root. (??? probably should default to nobody) As of Android M, processes should use this option even if they require Linux capabilities. Previously, to acquire Linux capabilities, a process would need to run as root, request the capabilities, then drop to its desired uid. There is a new mechanism through fs_config that allows device manufacturers to add Linux capabilities to specific binaries on a file system that should be used instead. This mechanism is described on http://source.android.com/devices/tech/config/filesystem.html. When using this new mechanism, processes can use the user option to select their desired uid without ever running as root. As of Android O, processes can also request capabilities directly in their .rc files. See the “capabilities” option below.

`group <groupname> [ <groupname>\* ]`

> Change to ‘groupname’ before exec'ing this service. Additional groupnames beyond the (required) first one are used to set the supplemental groups of the process (via setgroups()). Currently defaults to root. (??? probably should default to nobody)

`capabilities <capability> [ <capability>\* ]`

> Set capabilities when exec'ing this service. ‘capability’ should be a Linux capability without the “CAP_” prefix, like “NET_ADMIN” or “SETPCAP”. See http://man7.org/linux/man-pages/man7/capabilities.7.html for a list of Linux capabilities.

`setrlimit <resource> <cur> <max>`

> This applies the given rlimit to the service. rlimits are inherited by child processes, so this effectively applies the given rlimit to the process tree started by this service. It is parsed similarly to the setrlimit command specified below.

`seclabel <seclabel>`

> Change to ‘seclabel’ before exec'ing this service. Primarily for use by services run from the rootfs, e.g. ueventd, adbd. Services on the system partition can instead use policy-defined transitions based on their file security context. If not specified and no transition is defined in policy, defaults to the init context.

`oneshot`

> Do not restart the service when it exits.

`class <name> [ <name>\* ]`

> Specify class names for the service. All services in a named class may be started or stopped together. A service is in the class “default” if one is not specified via the class option. Additional classnames beyond the (required) first one are used to group services. `animation class` ‘animation’ class should include all services necessary for both boot animation and shutdown animation. As these services can be launched very early during bootup and can run until the last stage of shutdown, access to /data partition is not guaranteed. These services can check files under /data but it should not keep files opened and should work when /data is not available.

`onrestart`

> Execute a Command (see below) when service restarts.

`writepid <file> [ <file>\* ]`

> Write the child's pid to the given files when it forks. Meant for cgroup/cpuset usage. If no files under /dev/cpuset/ are specified, but the system property ‘ro.cpuset.default’ is set to a non-empty cpuset name (e.g. ‘/foreground’), then the pid is written to file /dev/cpuset/*cpuset_name*/tasks.

`priority <priority>`

> Scheduling priority of the service process. This value has to be in range -20 to 19. Default priority is 0. Priority is set via setpriority().

`namespace <pid|mnt>`

> Enter a new PID or mount namespace when forking the service.

`oom_score_adjust <value>`

> Sets the child's /proc/self/oom_score_adj to the specified value, which must range from -1000 to 1000.

`memcg.swappiness <value>`

> Sets the child's memory.swappiness to the specified value (only if memcg is mounted), which must be equal or greater than 0.

`memcg.soft_limit_in_bytes <value>`

> Sets the child's memory.soft_limit_in_bytes to the specified value (only if memcg is mounted), which must be equal or greater than 0.

`memcg.limit_in_bytes <value>`

> Sets the child's memory.limit_in_bytes to the specified value (only if memcg is mounted), which must be equal or greater than 0.

`shutdown <shutdown_behavior>`

> Set shutdown behavior of the service process. When this is not specified, the service is killed during shutdown process by using SIGTERM and SIGKILL. The service with shutdown_behavior of “critical” is not killed during shutdown until shutdown times out. When shutdown times out, even services tagged with “shutdown critical” will be killed. When the service tagged with “shutdown critical” is not running when shut down starts, it will be started.

Triggers

Triggers are strings which can be used to match certain kinds of events and used to cause an action to occur.

Triggers are subdivided into event triggers and property triggers.

Event triggers are strings triggered by the ‘trigger’ command or by the QueueEventTrigger() function within the init executable. These take the form of a simple string such as ‘boot’ or ‘late-init’.

Property triggers are strings triggered when a named property changes value to a given new value or when a named property changes value to any new value. These take the form of ‘property:=’ and ‘property:=*’ respectively. Property triggers are additionally evaluated and triggered accordingly during the initial boot phase of init.

An Action can have multiple property triggers but may only have one event trigger.

For example: `on boot && property:a=b` defines an action that is only executed when the ‘boot’ event trigger happens and the property a equals b.

`on property:a=b && property:c=d` defines an action that is executed at three times:

1. During initial boot if property a=b and property c=d.
2. Any time that property a transitions to value b, while property c already equals d.
3. Any time that property c transitions to value d, while property a already equals b.

Commands

`bootchart [start|stop]`

> Start/stop bootcharting. These are present in the default init.rc files, but bootcharting is only active if the file /data/bootchart/enabled exists; otherwise bootchart start/stop are no-ops.

`chmod <octal-mode> <path>`

> Change file access permissions.

`chown <owner> <group> <path>`

> Change file owner and group.

`class_start <serviceclass>`

> Start all services of the specified class if they are not already running. See the start entry for more information on starting services.

`class_stop <serviceclass>`

> Stop and disable all services of the specified class if they are currently running.

`class_reset <serviceclass>`

> Stop all services of the specified class if they are currently running, without disabling them. They can be restarted later using `class_start` .

`class_restart <serviceclass>`

> Restarts all services of the specified class.

`copy <src> <dst>`

> Copies a file. Similar to write, but useful for binary/large amounts of data. Regarding to the src file, copying from symbolic link file and world-writable or group-writable files are not allowed. Regarding to the dst file, the default mode created is 0600 if it does not exist. And it will be truncated if dst file is a normal regular file and already exists.

`domainname <name>`

> Set the domain name.

`enable <servicename>`

> Turns a disabled service into an enabled one as if the service did not specify disabled. If the service is supposed to be running, it will be started now. Typically used when the bootloader sets a variable that indicates a specific service should be started when needed. E.g.

```
on property:ro.boot.myfancyhardware=1
    enable my_fancy_service_for_my_fancy_hardware
```

`exec [ <seclabel> [ <user> [ <group>\* ] ] ] -- <command> [ <argument>\* ]`

> Fork and execute command with the given arguments. The command starts after "--" so that an optional security context, user, and supplementary groups can be provided. No other commands will be run until this one finishes. *seclabel* can be a - to denote default. Properties are expanded within *argument*. Init halts executing commands until the forked process exits.

`exec_background [ <seclabel> [ <user> [ <group>\* ] ] ] -- <command> [ <argument>\* ]`

> Fork and execute command with the given arguments. This is handled similarly to the `exec` command. The difference is that init does not halt executing commands until the process exits for `exec_background` .

`exec_start <service>`

> Start a given service and halt the processing of additional init commands until it returns. The command functions similarly to the `exec` command, but uses an existing service definition in place of the exec argument vector.

`export <name> <value>`

> Set the environment variable *name* equal to *value* in the global environment (which will be inherited by all processes started after this command is executed)

`hostname <name>`

> Set the host name.

`ifup <interface>`

> Bring the network interface *interface* online.

`insmod [-f] <path> [<options>]`

> Install the module at *path* with the specified options. -f: force installation of the module even if the version of the running kernel and the version of the kernel for which the module was compiled do not match.

`load_all_props`

> Loads properties from /system, /vendor, et cetera. This is included in the default init.rc.

`load_persist_props`

> Loads persistent properties when /data has been decrypted. This is included in the default init.rc.

`loglevel <level>`

> Sets the kernel log level to level. Properties are expanded within *level*.

`mkdir <path> [mode] [owner] [group]`

> Create a directory at *path*, optionally with the given mode, owner, and group. If not provided, the directory is created with permissions 755 and owned by the root user and root group. If provided,

> the mode, owner and group will be updated if the directory exists already.

`mount_all <fstab> [ <path> ]\* [--<option>]`

> Calls fs_mgr_mount_all on the given fs_mgr-format fstab and imports .rc files at the specified paths (e.g., on the partitions just mounted) with optional options “early” and “late”. Refer to the section of “Init .rc Files” for detail.

`mount <type> <device> <dir> [ <flag>\* ] [<options>]`

> Attempt to mount the named device at the directory *dir* _flag_s include “ro”, “rw”, “remount”, “noatime”, … *options* include “barrier=1”, “noauto_da_alloc”, “discard”, … as a comma separated string, eg: barrier=1,noauto_da_alloc

`restart <service>`

> Stops and restarts a running service, does nothing if the service is currently restarting, otherwise, it just starts the service.

`restorecon <path> [ <path>\* ]`

> Restore the file named by *path* to the security context specified in the file_contexts configuration. Not required for directories created by the init.rc as these are automatically labeled correctly by init.

`restorecon_recursive <path> [ <path>\* ]`

> Recursively restore the directory tree named by *path* to the security contexts specified in the file_contexts configuration.

`rm <path>`

> Calls unlink(2) on the given path. You might want to use “exec -- rm …” instead (provided the system partition is already mounted).

`rmdir <path>`

> Calls rmdir(2) on the given path.

`readahead <file|dir> [--fully]`

> Calls readahead(2) on the file or files within given directory. Use option --fully to read the full file content.

`setprop <name> <value>`

> Set system property *name* to *value*. Properties are expanded within *value*.

`setrlimit <resource> <cur> <max>`

> Set the rlimit for a resource. This applies to all processes launched after the limit is set. It is intended to be set early in init and applied globally. *resource* is best specified using its text representation ('cpu', 'rtio', etc or 'RLIM_CPU', 'RLIM_RTIO', etc). It also may be specified as the int value that the resource enum corresponds to.

`start <service>`

> Start a service running if it is not already running. Note that this is *not* synchronous, and even if it were, there is no guarantee that the operating system's scheduler will execute the service sufficiently to guarantee anything about the service's status.
>
> This creates an important consequence that if the service offers functionality to other services, such as providing a communication channel, simply starting this service before those services is *not* sufficient to guarantee that the channel has been set up before those services ask for it. There must be a separate mechanism to make any such guarantees.

`stop <service>`

> Stop a service from running if it is currently running.

`swapon_all <fstab>`

> Calls fs_mgr_swapon_all on the given fstab file.

`symlink <target> <path>`

> Create a symbolic link at *path* with the value *target*

`sysclktz <mins_west_of_gmt>`

> Set the system clock base (0 if system clock ticks in GMT)

`trigger <event>`

> Trigger an event. Used to queue an action from another action.

`umount <path>`

> Unmount the filesystem mounted at that path.

`verity_load_state`

> Internal implementation detail used to load dm-verity state.

`verity_update_state <mount-point>`

> Internal implementation detail used to update dm-verity state and set the partition.*mount-point*.verified properties used by adb remount because fs_mgr can't set them directly itself.

`wait <path> [ <timeout> ]`

> Poll for the existence of the given file and return when found, or the timeout has been reached. If timeout is not specified it currently defaults to five seconds.

`wait_for_prop <name> <value>`

> Wait for system property *name* to be *value*. Properties are expanded within *value*. If property *name* is already set to *value*, continue immediately.

`write <path> <content>`

> Open the file at *path* and write a string to it with write(2). If the file does not exist, it will be created. If it does exist, it will be truncated. Properties are expanded within *content*.

Imports

`import <path>`

> Parse an init config file, extending the current configuration. If *path* is a directory, each file in the directory is parsed as a config file. It is not recursive, nested directories will not be parsed.

The import keyword is not a command, but rather its own section, meaning that it does not happen as part of an Action, but rather, imports are handled as a file is being parsed and follow the below logic.

There are only three times where the init executable imports .rc files:

1. When it imports /init.rc or the script indicated by the property `ro.boot.init_rc` during initial boot.
2. When it imports /{system,vendor,odm}/etc/init/ for first stage mount devices immediately after importing /init.rc.
3. When it imports /{system,vendor,odm}/etc/init/ or .rc files at specified paths during mount_all.

The order that files are imported is a bit complex for legacy reasons and to keep backwards compatibility. It is not strictly guaranteed.

The only correct way to guarantee that a command has been run before a different command is to either 1) place it in an Action with an earlier executed trigger, or 2) place it in an Action with the same trigger within the same file at an earlier line.

Nonetheless, the defacto order for first stage mount devices is:

1. /init.rc is parsed then recursively each of its imports are parsed.
2. The contents of /system/etc/init/ are alphabetized and parsed sequentially, with imports happening recursively after each file is parsed.
3. Step 2 is repeated for /vendor/etc/init then /odm/etc/init

The below pseudocode may explain this more clearly:

```
fn Import(file)
  Parse(file)
  for (import : file.imports)
    Import(import)

Import(/init.rc)
Directories = [/system/etc/init, /vendor/etc/init, /odm/etc/init]
for (directory : Directories)
  files = <Alphabetical order of directory's contents>
  for (file : files)
    Import(file)
```

Properties

Init provides information about the services that it is responsible for via the below properties.

`init.svc.<name>`

> State of a named service (“stopped”, “stopping”, “running”, “restarting”)

Boot timing

Init records some boot timing information in system properties.

`ro.boottime.init`

> Time after boot in ns (via the CLOCK_BOOTTIME clock) at which the first stage of init started.

`ro.boottime.init.selinux`

> How long it took the first stage to initialize SELinux.

`ro.boottime.init.cold_boot_wait`

> How long init waited for ueventd's coldboot phase to end.

`ro.boottime.<service-name>`

> Time after boot in ns (via the CLOCK_BOOTTIME clock) that the service was first started.

Bootcharting

This version of init contains code to perform “bootcharting”: generating log files that can be later processed by the tools provided by http://www.bootchart.org/.

On the emulator, use the -bootchart *timeout* option to boot with bootcharting activated for *timeout* seconds.

On a device:

```
adb shell 'touch /data/bootchart/enabled'
```

Don't forget to delete this file when you're done collecting data!

The log files are written to /data/bootchart/. A script is provided to retrieve them and create a bootchart.tgz file that can be used with the bootchart command-line utility:

```
sudo apt-get install pybootchartgui
# grab-bootchart.sh uses $ANDROID_SERIAL.
$ANDROID_BUILD_TOP/system/core/init/grab-bootchart.sh
```

One thing to watch for is that the bootchart will show init as if it started running at 0s. You'll have to look at dmesg to work out when the kernel actually started init.

Comparing two bootcharts

A handy script named compare-bootcharts.py can be used to compare the start/end time of selected processes. The aforementioned grab-bootchart.sh will leave a bootchart tarball named bootchart.tgz at /tmp/android-bootchart. If two such barballs are preserved on the host machine under different directories, the script can list the timestamps differences. For example:

Usage: system/core/init/compare-bootcharts.py *base-bootchart-dir exp-bootchart-dir*

```
process: baseline experiment (delta) - Unit is ms (a jiffy is 10 ms on the system)
-----------------------------------
/init: 50 40 (-10)
/system/bin/surfaceflinger: 4320 4470 (+150)
/system/bin/bootanimation: 6980 6990 (+10)
zygote64: 10410 10640 (+230)
zygote: 10410 10640 (+230)
system_server: 15350 15150 (-200)
bootanimation ends at: 33790 31230 (-2560)
```

Systrace

Systrace (http://developer.android.com/tools/help/systrace.html) can be used for obtaining performance analysis reports during boot time on userdebug or eng builds.

Here is an example of trace events of "wm" and "am" categories:

```
$ANDROID_BUILD_TOP/external/chromium-trace/systrace.py \
      wm am --boot
```

This command will cause the device to reboot. After the device is rebooted and the boot sequence has finished, the trace report is obtained from the device and written as trace.html on the host by hitting Ctrl+C.

Limitation: recording trace events is started after persistent properties are loaded, so the trace events that are emitted before that are not recorded. Several services such as vold, surfaceflinger, and servicemanager are affected by this limitation since they are started before persistent properties are loaded. Zygote initialization and the processes that are forked from the zygote are not affected.

Debugging init

By default, programs executed by init will drop stdout and stderr into /dev/null. To help with debugging, you can execute your program via the Android program logwrapper. This will redirect stdout/stderr into the Android logging system (accessed via logcat).

For example service akmd /system/bin/logwrapper /sbin/akmd

For quicker turnaround when working on init itself, use:

```
mm -j &&
m ramdisk-nodeps &&
m bootimage-nodeps &&
adb reboot bootloader &&
fastboot boot $ANDROID_PRODUCT_OUT/boot.img
```

Alternatively, use the emulator:

```
emulator -partition-size 1024 \
    -verbose -show-kernel -no-window
```

Powered by Gitiles source log blame