Android Debug Bridge

Android Debug Bridge (adb) is a versatile command line tool that lets you communicate with an emulator instance or connected Android-powered device. It is a client-server program that includes three components:

- A client, which runs on your development machine. You can invoke a client from a shell by issuing an adb command. Other Android tools such as the ADT plugin and DDMS also create adb clients.
- A server, which runs as a background process on your development machine. The server manages communication between the client and the adb daemon running on an emulator or device.
- A daemon, which runs as a background process on each emulator or device instance.

You can find the `adb` tool in `<sdk>/platform-tools/`.

When you start an adb client, the client first checks whether there is an adb server process already running. If there isn't, it starts the server process. When the server starts, it binds to local TCP port 5037 and listens for commands sent from adb clients—all adb clients use port 5037 to communicate with the adb server.

ADB quickview

- Manage the state of an emulator or device
- Run shell commands on a device
- Manage port forwarding on an emulator or device
- Copy files to/from an emulator or device

In this document

See also

Emulator

The server then sets up connections to all running emulator/device instances. It locates emulator/device instances by scanning odd-numbered ports in the range 5555 to 5585, the range used by emulators/devices. Where the server finds an adb daemon, it sets up a connection to that port. Note that each emulator/device instance acquires a pair of sequential ports — an even-numbered port for console connections and an odd-numbered port for adb connections. For example:

Emulator 1, console: 5554
Emulator 1, adb: 5555
Emulator 2, console: 5556
Emulator 2, adb: 5557 ...

As shown, the emulator instance connected to adb on port 5555 is the same as the instance whose console listens on port 5554.

Once the server has set up connections to all emulator instances, you can use adb commands to control and access those instances. Because the server manages connections to emulator/device instances and handles commands from multiple adb clients, you can control any emulator/device instance from any client (or from a script).

The sections below describe the commands that you can use to access adb capabilities and manage the state of an emulator/device. Note that if you are developing Android applications in Eclipse and have installed the ADT plugin, you do not need to access adb from the command line. The ADT plugin provides a transparent integration of adb into the Eclipse IDE. However, you can still use adb directly as necessary, such as for debugging.

Issuing adb Commands

You can issue adb commands from a command line on your development machine or from a script. The usage is:

```
adb [-d|-e|-s <serialNumber>] <command>
```

When you issue a command, the program invokes an adb client. The client is not specifically associated with any emulator instance, so if multiple emulators/devices are running, you need to use the `-d` option to specify the target instance to which the command should be directed. For more information about using this option, see Directing Commands to a Specific

Emulator/Device Instance.

Querying for Emulator/Device Instances

Before issuing adb commands, it is helpful to know what emulator/device instances are connected to the adb server. You can generate a list of attached emulators/devices using the `devices` command:

```
adb devices
```

In response, adb prints this status information for each instance:

- Serial number — A string created by adb to uniquely identify an emulator/device instance by its console port number. The format of the serial number is `<type>-<consolePort>`. Here's an example serial number: `emulator-5554`
- State — The connection state of the instance. Three states are supported:
  - `offline` — the instance is not connected to adb or is not responding.
  - `device` — the instance is now connected to the adb server. Note that this state does not imply that the Android system is fully booted and operational, since the instance connects to adb while the system is still booting. However, after boot-up, this is the normal operational state of an emulator/device instance.

The output for each instance is formatted like this:

```
[serialNumber] [state]
```

Here's an example showing the `devices` command and its output:

```
$ adb devices
List of devices attached
emulator-5554  device
emulator-5556  device
emulator-5558  device
```

If there is no emulator/device running, adb returns `no device`.

Directing Commands to a Specific Emulator/Device Instance

If multiple emulator/device instances are running, you need to specify a target instance when issuing adb commands. To so so, use the `-s` option in the commands. The usage for the `-s` option is:

```
adb -s <serialNumber> <command>
```

As shown, you specify the target instance for a command using its adb-assigned serial number. You can use the `devices` command to obtain the serial numbers of running emulator/device instances.

Here is an example:

```
adb -s emulator-5556 install helloWorld.apk
```

Note that, if you issue a command without specifying a target emulator/device instance using `-s`, adb generates an error.

Installing an Application

You can use adb to copy an application from your development computer and install it on an emulator/device instance. To do so, use the `install` command. With the command, you must specify the path to the .apk file that you want to install:

```
adb install <path_to_apk>
```

For more information about how to create an .apk file that you can install on an emulator/device instance, see Building and Running

Note that, if you are using the Eclipse IDE and have the ADT plugin installed, you do not need to use adb (or aapt) directly to install your application on the emulator/device. Instead, the ADT plugin handles the packaging and installation of the application for you.

Forwarding Ports

You can use the `forward` command to set up arbitrary port forwarding — forwarding of requests on a specific host port to a different port on an emulator/device instance. Here's how you would set up forwarding of host port 6100 to emulator/device port 7100:

```
adb forward tcp:6100 tcp:7100
```

You can also use adb to set up forwarding to named abstract UNIX domain sockets, as illustrated here:

```
adb forward tcp:6100 local:logd
```

Copying Files to or from an Emulator/Device Instance

You can use the adb commands `pull` and `push` to copy files to and from an emulator/device instance's data file. Unlike the `install` command, which only copies an .apk file to a specific location, the `pull` and `push` commands let you copy arbitrary directories and files to any location in an emulator/device instance.

To copy a file or directory (recursively) *from* the emulator or device, use

```
adb pull <remote> <local>
```

To copy a file or directory (recursively) *to* the emulator or device, use

```
adb push <local> <remote>
```

In the commands, `<local>` and `<remote>` refer to the paths to the target files/directory on your development machine (local) and on the emulator/device instance (remote).

Here's an example:

```
adb push foo.txt /sdcard/foo.txt
```

Listing of adb Commands

The table below lists all of the supported adb commands and explains their meaning and usage.

Category	Command	Description	Comments
Options	`-d`	Direct an adb command to the only attached USB device.	Returns an error if more than one USB device is attached.
	`-e`	Direct an adb command to the only	Returns an error if more

		running emulator instance.	than one emulator instance is running.
	`-s <serialNumber>`	Direct an adb command a specific emulator/device instance, referred to by its adb-assigned serial number (such as "emulator-5556").	If not specified, adb generates an error.
General	`devices`	Prints a list of all attached emulator/device instances.	See Querying for Emulator/Device Instances for more information.
	`help`	Prints a list of supported adb commands.	
	`version`	Prints the adb version number.	
Debug	`logcat [<option>] [<filter-specs>]`	Prints log data to the screen.	
	`bugreport`	Prints `dumpsys`, `dumpstate`, and `logcat` data to the screen, for the purposes of bug reporting.	
	`jdwp`	Prints a list of available JDWP processes on a given device.	You can use the `forward jdwp:<pid>` port-forwarding specification to connect to a specific JDWP process. For example: `adb forward tcp:8000 jdwp:472` `jdb -attach localhost:8000`
Data	`install <path-to-apk>`	Pushes an Android application (specified as a full path to an .apk file) to the data file of an emulator/device.	
	`pull <remote> <local>`	Copies a specified file from an emulator/device instance to your development computer.	
	`push <local> <remote>`	Copies a specified file from your development computer to an emulator/device instance.	
Ports and Networking	`forward <local> <remote>`	Forwards socket connections from a specified local port to a specified remote port on the emulator/device instance.	Port specifications can use these schemes: • `tcp:<portnum>` • `local:<UNIX domain socket name>` • `dev:<character device name>` • `jdwp:<pid>`
	`ppp <tty> [parm]...`	Run PPP over USB. • `<tty>` — the tty for PPP stream. For example `dev:/dev/omap_csmi_ttyl`. • `[parm]...` &mdash zero or more PPP/PPPD options, such as `defaultroute`, `local`, `notty`, etc. Note that you should not	

		automatically start a PPP connection.	
Scripting	`get-serialno`	Prints the adb instance serial number string.	See Querying for Emulator/Device Instances for more information.
	`get-state`	Prints the adb state of an emulator/device instance.	

	`wait-for-device`	Blocks execution until the device is online — that is, until the instance state is `device`.	You can prepend this command to other adb commands, in which case adb will wait until the emulator/device instance is connected before issuing the other commands. Here's an example:
			`adb wait-for-device shell getprop`
			Note that this command does *not* cause adb to wait until the entire system is fully booted. For that reason, you should not prepend it to other commands that require a fully booted system. As an example, the `install` requires the Android package manager, which is available only after the system is fully booted. A command such as
			`adb wait-for-device install <app>.apk`
			would issue the `install` command as soon as the emulator or device instance connected to the adb server, but before the Android system was fully booted, so it would result in an error.

Server	`start-server`	Checks whether the adb server process is running and starts it, if not.	
	`kill-server`	Terminates the adb server process.	
Shell	`shell`	Starts a remote shell in the target emulator/device instance.	See Issuing Shell Commands for more information.
	`shell [<shellCommand>]`	Issues a shell command in the target emulator/device instance and then exits the remote shell.	

Issuing Shell Commands

Adb provides an ash shell that you can use to run a variety of commands on an emulator or device. The command binaries are stored in the file system of the emulator or device, in this location:

```
/system/bin/...
```

You can use the `shell` command to issue commands, with or without entering the adb remote shell on the emulator/device.

To issue a single command without entering a remote shell, use the `shell` command like this:

```
adb [-d|-e|-s {<serialNumber>}] shell <shellCommand>
```

To drop into a remote shell on a emulator/device instance, use the `shell` command like this:

```
adb [-d|-e|-s {<serialNumber>}] shell
```

When you are ready to exit the remote shell, use `CTRL+D` or `exit` to end the shell session.

The sections below provide more information about shell commands that you can use.

Examining sqlite3 Databases from a Remote Shell

From an adb remote shell, you can use the sqlite3 command-line program to manage SQLite databases created by Android applications. The `sqlite3` tool includes many useful commands, such as `.dump` to print out the contents of a table and `.schema` to print the SQL CREATE statement for an existing table. The tool also gives you the ability to execute SQLite commands on the fly.

To use `sqlite3`, enter a remote shell on the emulator instance, as described above, then invoke the tool using the `sqlite3` command. Optionally, when invoking `sqlite3` you can specify the full path to the database you want to explore. Emulator/device instances store SQLite3 databases in the folder `/data/data/<package_name>/databases/`.

Here's an example:

```
$ adb -s emulator-5554 shell
# sqlite3 /data/data/com.example.google.rss.rssexample/databases/rssitems.db
SQLite version 3.3.12
Enter ".help" for instructions
.... enter commands, then quit...
sqlite> .exit
```

Once you've invoked `sqlite3`, you can issue `sqlite3` commands in the shell. To exit and return to the adb remote shell, use `exit` or `CTRL+D`.

UI/Application Exerciser Monkey

The Monkey is a program that runs on your emulator or device and generates pseudo-random streams of user events such as clicks, touches, or gestures, as well as a number of system-level events. You can use the Monkey to stress-test applications that you are developing, in a random yet repeatable manner.

The simplest way to use the monkey is with the following command, which will launch your application and send 500 pseudo-random events to it.

```
$ adb shell monkey -v -p your.package.name 500
```

For more information about command options for Monkey, see the complete UI/Application Exerciser Monkey documentation page.

Other Shell Commands

The table below lists several of the adb shell commands available. For a complete list of commands and programs, start an emulator instance and use the `adb -help` command.

```
adb shell ls /system/bin
```

Help is available for most of the commands.

Shell Command	Description	Comments
`dumpsys`	Dumps system data to the screen.	The Dalvik Debug Monitor Server (DDMS) tool offers integrated debug environment that you may find easier to use.
`dumpstate`	Dumps state to a file.	
`logcat [<option>]... [<filter-spec>]...`	Enables radio logging and prints output to the screen.	
`dmesg`	Prints kernel debugging messages to the screen.	
`start`	Starts (restarts) an emulator/device instance.	
`stop`	Stops execution of an emulator/device instance.	

Enabling logcat Logging

The Android logging system provides a mechanism for collecting and viewing system debug output. Logs from various applications and portions of the system are collected in a series of circular buffers, which then can be viewed and filtered by the `logcat` command.

Using logcat Commands

You can use the `logcat` command to view and follow the contents of the system's log buffers. The general usage is:

```
[adb] logcat [<option>] ... [<filter-spec>] ...
```

The sections below explain filter specifications and the command options. See Listing of logcat Command Options for a summary of options.

You can use the `logcat` command from your development computer or from a remote adb shell in an emulator/device instance. To view log output in your development computer, you use

```
$ adb logcat
```

and from a remote adb shell you use

```
# logcat
```

Filtering Log Output

Every Android log message has a *tag* and a *priority* associated with it.

- The tag of a log message is a short string indicating the system component from which the message originates (for example, "View" for the view system).

- The priority is one of the following character values, ordered from lowest to highest priority:
  - `V` — Verbose (lowest priority)
  - `D` — Debug
  - `I` — Info
  - `W` — Warning
  - `E` — Error
  - `F` — Fatal
  - `S` — Silent (highest priority, on which nothing is ever printed)

You can obtain a list of tags used in the system, together with priorities, by running `logcat` and observing the first two columns of each message, given as `<priority>/<tag>`.

Here's an example of logcat output that shows that the message relates to priority level "I" and tag "ActivityManager":

```
I/ActivityManager(  585): Starting activity: Intent { action=android.intent.action...}
```

To reduce the log output to a manageable level, you can restrict log output using *filter expressions*. Filter expressions let you indicate to the system the tags-priority combinations that you are interested in — the system suppresses other messages for the specified tags.

A filter expression follows this format `tag:priority ...`, where `tag` indicates the tag of interest and `priority` indicates the *minimum* level of priority to report for that tag. Messages for that tag at or above the specified priority are written to the log. You can supply any number of `tag:priority` specifications in a single filter expression. The series of specifications is whitespace-delimited.

Here's an example of a filter expression that suppresses all log messages except those with the tag "ActivityManager", at priority "Info" or above, and all log messages with tag "MyApp", with priority "Debug" or above:

```
adb logcat ActivityManager:I MyApp:D *:S
```

The final element in the above expression, `*:S`, sets the priority level for all tags to "silent", thus ensuring only log messages with "View" and "MyApp" are displayed. Using `*:S` is an excellent way to ensure that log output is restricted to the filters that you have explicitly specified — it lets your filters serve as a "whitelist" for log output.

The following filter expression displays all log messages with priority level "warning" and higher, on all tags:

```
adb logcat *:W
```

If you're running `logcat` from your development computer (versus running it on a remote adb shell), you can also set a default filter expression by exporting a value for the environment variable `ANDROID_LOG_TAGS`:

```
export ANDROID_LOG_TAGS="ActivityManager:I MyApp:D *:S"
```

Note that `ANDROID_LOG_TAGS` filter is not exported to the emulator/device instance, if you are running `logcat` from a remote shell or using `adb shell logcat`.

Controlling Log Output Format

Log messages contain a number of metadata fields, in addition to the tag and priority. You can modify the output format for messages so that they display a specific metadata field. To do so, you use the `-v` option and specify one of the supported output formats listed below.

- `brief` — Display priority/tag and PID of originating process (the default format).
- `process` — Display PID only.
- `tag` — Display the priority/tag only.
- `thread` — Display process:thread and priority/tag only.
- `raw` — Display the raw log message, with no other metadata fields.
- `time` — Display the date, invocation time, priority/tag, and PID of the originating process.

- `long` — Display all metadata fields and separate messages with a blank lines.

When starting `logcat`, you can specify the output format you want by using the `-v` option:

```
[adb] logcat [-v <format>]
```

Here's an example that shows how to generate messages in `thread` output format:

```
adb logcat -v thread
```

Note that you can only specify one output format with the `-v` option.

Viewing Alternative Log Buffers

The Android logging system keeps multiple circular buffers for log messages, and not all of the log messages are sent to the default circular buffer. To see additional log messages, you can start `logcat` with the `-b` option, to request viewing of an alternate circular buffer. You can view any of these alternate buffers:

- `radio` — View the buffer that contains radio/telephony related messages.
- `events` — View the buffer containing events-related messages.
- `main` — View the main log buffer (default)

The usage of the `-b` option is:

```
[adb] logcat [-b <buffer>]
```

Here's an example of how to view a log buffer containing radio and telephony messages:

```
adb logcat -b radio
```

Viewing stdout and stderr

By default, the Android system sends `stdout` and `stderr` (`System.out` and `System.err`) output to `/dev/null`. In processes that run the Dalvik VM, you can have the system write a copy of the output to the log file. In this case, the system writes the messages to the log using the log tags `stdout` and `stderr`, both with priority `I`.

To route the output in this way, you stop a running emulator/device instance and then use the shell command `setprop` to enable the redirection of output. Here's how you do it:

```
$ adb shell stop
$ adb shell setprop log.redirect-stdio true
$ adb shell start
```

The system retains this setting until you terminate the emulator/device instance. To use the setting as a default on the emulator/device instance, you can add an entry to `/data/local.prop` on the device.

Listing of logcat Command Options

Option	Description
`-b <buffer>`	Loads an alternate log buffer for viewing, such as `event` or `radio`. The `main` buffer is used by default. See Viewing Alternative Log Buffers.
`-c`	Clears (flushes) the entire log and exits.
`-d`	Dumps the log to the screen and exits.
`-f <filename>`	Writes log message output to `<filename>`. The default is `stdout`.
	Prints the size of the specified log buffer and exits.

`-g`	
`-n <count>`	Sets the maximum number of rotated logs to `<count>`. The default value is 4. Requires the `-r` option.
`-r <kbytes>`	Rotates the log file every `<kbytes>` of output. The default value is 16. Requires the `-f` option.
`-s`	Sets the default filter spec to silent.
`-v <format>`	Sets the output format for log messages. The default is `brief` format. For a list of supported formats, see Controlling Log Output Format.

Stopping the adb Server

In some cases, you might need to terminate the adb server process and then restart it. For example, if adb does not respond to a command, you can terminate the server and restart it and that may resolve the problem.

To stop the adb server, use the `kill-server`. You can then restart the server by issuing any adb command.

← Back to Tools ↑ Go to top