Mobly Documentation

Ang Li

Sep 21, 2023

Contents:

1 mobly package **1**
- 1.1 Subpackages . . . 1
  - 1.1.1 mobly.controllers package . . . 1
    - 1.1.1.1 Subpackages . . . 1
    - 1.1.1.2 Submodules . . . 23
    - 1.1.1.3 mobly.controllers.android_device module . . . 23
    - 1.1.1.4 mobly.controllers.attenuator module . . . 31
    - 1.1.1.5 mobly.controllers.iperf_server module . . . 32
    - 1.1.1.6 mobly.controllers.monsoon module . . . 33
    - 1.1.1.7 mobly.controllers.sniffer module . . . 33
    - 1.1.1.8 Module contents . . . 35
- 1.2 Submodules . . . 35
- 1.3 mobly.asserts module . . . 35
- 1.4 mobly.base_instrumentation_test module . . . 40
- 1.5 mobly.base_test module . . . 41
- 1.6 mobly.config_parser module . . . 46
- 1.7 mobly.controller_manager module . . . 47
- 1.8 mobly.expects module . . . 48
- 1.9 mobly.keys module . . . 48
- 1.10 mobly.logger module . . . 49
- 1.11 mobly.records module . . . 51
- 1.12 mobly.runtime_test_info module . . . 56
- 1.13 mobly.signals module . . . 57
- 1.14 mobly.suite_runner module . . . 58
- 1.15 mobly.test_runner module . . . 60
- 1.16 mobly.utils module . . . 61
- 1.17 Module contents . . . 65

2 Indices and tables **67**

Python Module Index **69**

Index **71**

CHAPTER 1

mobly package

1.1 Subpackages

1.1.1 mobly.controllers package

1.1.1.1 Subpackages

mobly.controllers.android_device_lib package

Subpackages

mobly.controllers.android_device_lib.services package

Submodules

mobly.controllers.android_device_lib.services.base_service module

Module for the BaseService.

class mobly.controllers.android_device_lib.services.base_service.**BaseService**(*device*, *configs=None*)

Bases: abc.ABC

Base class of a Mobly AndroidDevice service.

This class defines the interface for Mobly's AndroidDevice service.

alias

String, alias used to register this service with service manager.

This can be None if the service is never registered.

create_output_excerpts(*test_info*)
Creates excerpts of the service's output files.

[Optional] This method only applies to services with output files.

For services that generates output files, calling this method would create excerpts of the output files. An excerpt should contain info between two calls of *create_output_excerpts* or from the start of the service to the call to *create_output_excerpts*.

Use *AndroidDevice#generate_filename* to get the proper filenames for excerpts.

This is usually called at the end of: *setup_class*, *teardown_test*, or *teardown_class*.

Parameters **test_info** – RuntimeTestInfo, the test info associated with the scope of the excerpts.

Returns

List of strings, the absolute paths to the excerpt files created. Empty list if no excerpt files are created.

is_alive
True if the service is active; False otherwise.

pause()
Pauses a service temporarily.

For when the Python service object needs to temporarily lose connection to the device without shutting down the service running on the actual device.

This is relevant when a service needs to maintain a constant connection to the device and the connection is lost if USB connection to the device is disrupted.

E.g. a services that utilizes a socket connection over adb port forwarding would need to implement this for the situation where the USB connection to the device will be temporarily cut, but the device is not rebooted.

For more context, see: *mobly.controllers.android_device.AndroidDevice.handle_usb_disconnect*

If not implemented, we assume the service is not sensitive to device disconnect, and *stop* will be called by default.

resume()
Resumes a paused service.

Same context as the *pause* method. This should resume the service after the connection to the device has been re-established.

If not implemented, we assume the service is not sensitive to device disconnect, and *start* will be called by default.

start()
Starts the service.

stop()
Stops the service and cleans up all resources.

This method should handle any error and not throw.

mobly.controllers.android_device_lib.services.logcat module

class mobly.controllers.android_device_lib.services.logcat.**Config**(*logcat_params=None*, *clear_log=True*, *output_file_path=None*)

Bases: object

Config object for logcat service.

clear_log
bool, clears the logcat before collection if True.

logcat_params
string, extra params to be added to logcat command.

output_file_path
string, the path on the host to write the log file to, including the actual filename. The service will automatically generate one if not specified.

exception mobly.controllers.android_device_lib.services.logcat.**Error**(*device*, *msg*)

Bases: *mobly.controllers.android_device_lib.errors.ServiceError*

Root error type for logcat service.

SERVICE_TYPE = 'Logcat'

class mobly.controllers.android_device_lib.services.logcat.**Logcat**(*android_device*, *configs=None*)

Bases: *mobly.controllers.android_device_lib.services.base_service.BaseService*

Android logcat service for Mobly's AndroidDevice controller.

adb_logcat_file_path
string, path to the file that the service writes adb logcat to by default.

OUTPUT_FILE_TYPE = 'logcat'

clear_adb_log()
Clears cached adb content.

create_output_excerpts(*test_info*)
Convenient method for creating excerpts of adb logcat.

This copies logcat lines from self.adb_logcat_file_path to an excerpt file, starting from the location where the previous excerpt ended.

Call this method at the end of: *setup_class*, *teardown_test*, and *teardown_class*.

Parameters **test_info** – *self.current_test_info* in a Mobly test.

Returns List of strings, the absolute paths to excerpt files.

is_alive
True if the service is active; False otherwise.

pause()
Pauses logcat.

Note: the service is unable to collect the logs when paused, if more logs are generated on the device than the device's log buffer can hold, some logs would be lost.

resume()
Resumes a paused logcat service.

start()
Starts a standing adb logcat collection.

The collection runs in a separate subprocess and saves logs in a file.

stop()
Stops the adb logcat service.

update_config(*new_config*)
Updates the configuration for the service.

The service needs to be stopped before updating, and explicitly started after the update.

This will reset the service. Previous output files may be orphaned if output path is changed.

Parameters **new_config** – Config, the new config to use.

mobly.controllers.android_device_lib.services.sl4a_service module

Module for the Sl4aService.

class mobly.controllers.android_device_lib.services.sl4a_service.**Sl4aService**(*device*, *configs=None*)

Bases: *mobly.controllers.android_device_lib.services.base_service.BaseService*

Service for managing sl4a's client.

Direct calls on the service object will forwarded to the client object as syntactic sugar. So *Sl4aService.doFoo()* is equivalent to *Sl4aClient.doFoo()*.

is_alive
True if the service is active; False otherwise.

pause()
Pauses a service temporarily.

For when the Python service object needs to temporarily lose connection to the device without shutting down the service running on the actual device.

This is relevant when a service needs to maintain a constant connection to the device and the connection is lost if USB connection to the device is disrupted.

E.g. a services that utilizes a socket connection over adb port forwarding would need to implement this for the situation where the USB connection to the device will be temporarily cut, but the device is not rebooted.

For more context, see: *mobly.controllers.android_device.AndroidDevice.handle_usb_disconnect*

If not implemented, we assume the service is not sensitive to device disconnect, and *stop* will be called by default.

resume()
Resumes a paused service.

Same context as the *pause* method. This should resume the service after the connection to the device has been re-established.

If not implemented, we assume the service is not sensitive to device disconnect, and *start* will be called by default.

start()
Starts the service.

stop()
Stops the service and cleans up all resources.

This method should handle any error and not throw.

mobly.controllers.android_device_lib.services.snippet_management_service module

Module for the snippet management service.

exception mobly.controllers.android_device_lib.services.snippet_management_service.**Error**(*de*
m

Bases: *mobly.controllers.android_device_lib.errors.ServiceError*

Root error type for snippet management service.

SERVICE_TYPE = 'SnippetManagementService'

class mobly.controllers.android_device_lib.services.snippet_management_service.**SnippetManag**

Bases: *mobly.controllers.android_device_lib.services.base_service.BaseService*

Management service of snippet clients.

This service manages all the snippet clients associated with an Android device.

add_snippet_client(*name*, *package*, *config=None*)
Adds a snippet client to the management.

Parameters

- **name** – string, the attribute name to which to attach the snippet client. E.g. *name='maps'* attaches the snippet client to *ad.maps*.
- **package** – string, the package name of the snippet apk to connect to.
- **config** – snippet_client_v2.Config, the configuration object for controlling the snippet behaviors. See the docstring of the *Config* class for supported configurations.

Raises Error, if a duplicated name or package is passed in.

get_snippet_client(*name*)
Gets the snippet client managed under a given name.

Parameters **name** – string, the name of the snippet client under management.

Returns SnippetClient.

is_alive
True if any client is running, False otherwise.

pause()
Pauses all the snippet clients under management.

This clears the host port of a client because a new port will be allocated in *resume*.

remove_snippet_client(*name*)
Removes a snippet client from management.

Parameters **name** – string, the name of the snippet client to remove.

Raises *Error* – if no snippet client is managed under the specified name.

resume()
Resumes all paused snippet clients.

start()
Starts all the snippet clients under management.

stop()
Stops all the snippet clients under management.

Module contents

Submodules

mobly.controllers.android_device_lib.adb module

exception mobly.controllers.android_device_lib.adb.**AdbError**(*cmd*, *stdout*, *stderr*, *ret_code*, *serial=''*)
Bases: *mobly.controllers.android_device_lib.adb.Error*

Raised when an adb command encounters an error.

cmd
list of strings, the adb command executed.

stdout
byte string, the raw stdout of the command.

stderr
byte string, the raw stderr of the command.

ret_code
int, the return code of the command.

serial
string, the serial of the device the command is executed on. This is an empty string if the adb command is not specific to a device.

class mobly.controllers.android_device_lib.adb.**AdbProxy**(*serial=''*)
Bases: object

Proxy class for ADB.

For syntactic reasons, the '-' in adb commands need to be replaced with '_'. Can directly execute adb commands on an object: >> adb = AdbProxy(<serial>) >> adb.start_server() >> adb.devices() # will return the console output of "adb devices".

By default, command args are expected to be an iterable which is passed directly to subprocess.Popen(): >> adb.shell(['echo', 'a', 'b'])

This way of launching commands is recommended by the subprocess documentation to avoid shell injection vulnerabilities and avoid having to deal with multiple layers of shell quoting and different shell environments between different OSes.

If you really want to run the command through the system shell, this is possible by supplying shell=True, but try to avoid this if possible: >> adb.shell('cat /foo > /tmp/file', shell=True)

connect(*address*) → bytes
Executes the *adb connect* command with proper status checking.

Parameters **address** – string, the address of the Android instance to connect to.

Returns The stdout content.

Raises *AdbError* – if the connection failed.

current_user_id
The integer ID of the current Android user.

Some adb commands require specifying a user ID to work properly. Use this to get the current user ID.

Note a "user" is not the same as an "account" in Android. See AOSP's documentation for details. https://source.android.com/devices/tech/admin/multi-user

forward(*args=None*, *shell=False*) → bytes

getprop(*prop_name*, *timeout=10*)
Get a property of the device.

This is a convenience wrapper for *adb shell getprop xxx*.

Parameters

- **prop_name** – A string that is the name of the property to get.
- **timeout** – float, the number of seconds to wait before timing out. If not specified, the DEFAULT_GETPROP_TIMEOUT_SEC is used.

Returns A string that is the value of the property, or None if the property doesn't exist.

getprops(*prop_names*)
Get multiple properties of the device.

This is a convenience wrapper for *adb shell getprop*. Use this to reduce the number of adb calls when getting multiple properties.

Parameters **prop_names** – list of strings, the names of the properties to get.

Returns A dict containing name-value pairs of the properties requested, if they exist.

has_shell_command(*command*) → bool
Checks to see if a given check command exists on the device.

Parameters **command** – A string that is the name of the command to check.

Returns A boolean that is True if the command exists and False otherwise.

instrument(*package*, *options=None*, *runner=None*, *handler=None*) → bytes
Runs an instrumentation command on the device.

This is a convenience wrapper to avoid parameter formatting.

Example:

```
device.instrument(
  'com.my.package.test',
  options = {
    'class': 'com.my.package.test.TestSuite',
  },
)
```

Parameters

- **package** – string, the package of the instrumentation tests.
- **options** – dict, the instrumentation options including the test class.
- **runner** – string, the test runner name, which defaults to DEFAULT_INSTRUMENTATION_RUNNER.
- **handler** – optional func, when specified the function is used to parse the instrumentation stdout line by line as the output is generated; otherwise, the stdout is simply returned once the instrumentation is finished.

Returns

The stdout of instrumentation command or the stderr if the handler is set.

root() → bytes

Enables ADB root mode on the device.

This method will retry to execute the command *adb root* when an AdbError occurs, since sometimes the error *adb: unable to connect for root: closed* is raised when executing *adb root* immediately after the device is booted to OS.

Returns A string that is the stdout of root command.

Raises *AdbError* – If the command exit code is not 0.

exception mobly.controllers.android_device_lib.adb.**AdbTimeoutError**(*cmd*, *timeout*, *serial=''*)

Bases: *mobly.controllers.android_device_lib.adb.Error*

Raised when an command did not complete within expected time.

cmd

list of strings, the adb command that timed out

timeout

float, the number of seconds passed before timing out.

serial

string, the serial of the device the command is executed on. This is an empty string if the adb command is not specific to a device.

exception mobly.controllers.android_device_lib.adb.**Error**

Bases: Exception

Base error type for adb proxy module.

mobly.controllers.android_device_lib.adb.**is_adb_available**()

Checks if adb is available as a command line tool.

Returns True if adb binary is available in console, False otherwise.

mobly.controllers.android_device_lib.adb.**list_occupied_adb_ports**()

Lists all the host ports occupied by adb forward.

This is useful because adb will silently override the binding if an attempt to bind to a port already used by adb was made, instead of throwing binding error. So one should always check what ports adb is using before trying to bind to a port with adb.

Returns A list of integers representing occupied host ports.

mobly.controllers.android_device_lib.callback_handler module

class mobly.controllers.android_device_lib.callback_handler.**CallbackHandler**(*callback_id*, *event_client*, *ret_value*, *method_name*, *ad*)

Bases: object

The class used to handle a specific group of callback events.

DEPRECATED: Use mobly.controllers.android_device_lib.callback_handler_v2.CallbackHandlerV2 instead.

All the events handled by a CallbackHandler are originally triggered by one async Rpc call. All the events are tagged with a callback_id specific to a call to an AsyncRpc method defined on the server side.

The raw message representing an event looks like:

```
{
  'callbackId': <string, callbackId>,
  'name': <string, name of the event>,
  'time': <long, epoch time of when the event was created on the
    server side>,
  'data': <dict, extra data from the callback on the server side>
}
```

Each message is then used to create a SnippetEvent object on the client side.

ret_value
The direct return value of the async Rpc call.

callback_id

getAll(*event_name*)
Gets all the events of a certain name that have been received so far. This is a non-blocking call.

Parameters

- **callback_id** – The id of the callback.
- **event_name** – string, the name of the event to get.

Returns A list of SnippetEvent, each representing an event from the Java side.

waitAndGet(*event_name*, *timeout=120*)
Blocks until an event of the specified name has been received and return the event, or timeout.

Parameters

- **event_name** – string, name of the event to get.
- **timeout** – float, the number of seconds to wait before giving up.

Returns SnippetEvent, the oldest entry of the specified event.

Raises

- Error – If the specified timeout is longer than the max timeout supported.
- TimeoutError – The expected event does not occur within time limit.

waitForEvent(*event_name*, *predicate*, *timeout=120*)
Wait for an event of a specific name that satisfies the predicate.

This call will block until the expected event has been received or time out.

The predicate function defines the condition the event is expected to satisfy. It takes an event and returns True if the condition is satisfied, False otherwise.

Note all events of the same name that are received but don't satisfy the predicate will be discarded and not be available for further consumption.

Parameters

- **event_name** – string, the name of the event to wait for.
- **predicate** – function, a function that takes an event (dictionary) and returns a bool.
- **timeout** – float, default is 120s.

Returns dictionary, the event that satisfies the predicate if received.

Raises `TimeoutError` – raised if no event that satisfies the predicate is received after timeout seconds.

mobly.controllers.android_device_lib.errors module

exception mobly.controllers.android_device_lib.errors.**DeviceError**(*ad*, *msg*)
Bases: *mobly.controllers.android_device_lib.errors.Error*

Raised for errors specific to an AndroidDevice object.

exception mobly.controllers.android_device_lib.errors.**Error**
Bases: *mobly.signals.ControllerError*

exception mobly.controllers.android_device_lib.errors.**ServiceError**(*device*, *msg*)
Bases: *mobly.controllers.android_device_lib.errors.DeviceError*

Raised for errors specific to an AndroidDevice service.

A service is inherently associated with a device instance, so the service error type is a subtype of *DeviceError*.

SERVICE_TYPE = None

mobly.controllers.android_device_lib.event_dispatcher module

exception mobly.controllers.android_device_lib.event_dispatcher.**DuplicateError**
Bases: *mobly.controllers.android_device_lib.event_dispatcher.EventDispatcherError*

Raise when a duplicate is being created and it shouldn't.

class mobly.controllers.android_device_lib.event_dispatcher.**EventDispatcher**(*sl4a*)
Bases: `object`

Class managing events for an sl4a connection.

DEFAULT_TIMEOUT = 60

clean_up()
Clean up and release resources after the event dispatcher polling loop has been broken.

The following things happen: 1. Clear all events and flags. 2. Close the sl4a client the event_dispatcher object holds. 3. Shut down executor without waiting.

clear_all_events()
Clear all event queues and their cached events.

clear_events(*event_name*)
Clear all events of a particular name.

Parameters **event_name** – Name of the events to be popped.

get_event_q(*event_name*)
Obtain the queue storing events of the specified name.

If no event of this name has been polled, wait for one to.

Returns A queue storing all the events of the specified name. None if timed out.

Raises `queue.Empty` – Raised if the queue does not exist and timeout has passed.

handle_event(*event_handler*, *event_name*, *user_args*, *event_timeout=None*, *cond=None*, *cond_timeout=None*)
Handle events that don't have registered handlers

In a new thread, poll one event of specified type from its queue and execute its handler. If no such event exists, the thread waits until one appears.

Parameters

- **event_handler** – Handler for the event, which should take at least one argument - the event json object.
- **event_name** – Name of the event to be handled.
- **user_args** – User arguments for the handler; to be passed in after the event json.
- **event_timeout** – Number of seconds to wait for the event to come.
- **cond** – A condition to wait on before executing the handler. Should be a threading.Event object.
- **cond_timeout** – Number of seconds to wait before the condition times out. Never times out if None.

Returns A concurrent.Future object associated with the handler. If blocking call worker.result() is triggered, the handler needs to return something to unblock.

handle_subscribed_event(*event_obj*, *event_name*)
Execute the registered handler of an event.

Retrieve the handler and its arguments, and execute the handler in a new thread.

Parameters

- **event_obj** – Json object of the event.
- **event_name** – Name of the event to call handler for.

poll_events()
Continuously polls all types of events from sl4a.

Events are sorted by name and store in separate queues. If there are registered handlers, the handlers will be called with corresponding event immediately upon event discovery, and the event won't be stored. If exceptions occur, stop the dispatcher and return

pop_all(*event_name*)
Return and remove all stored events of a specified name.

Pops all events from their queue. May miss the latest ones. If no event is available, return immediately.

Parameters **event_name** – Name of the events to be popped.

Returns List of the desired events.

Raises *IllegalStateError* – Raised if pop is called before the dispatcher starts polling.

pop_event(*event_name*, *timeout=60*)
Pop an event from its queue.

Return and remove the oldest entry of an event. Block until an event of specified name is available or times out if timeout is set.

Parameters

- **event_name** – Name of the event to be popped.
- **timeout** – Number of seconds to wait when event is not present. Never times out if None.

Returns The oldest entry of the specified event. None if timed out.

Raises *IllegalStateError* – Raised if pop is called before the dispatcher starts polling.

pop_events(*regex_pattern*, *timeout*)
Pop events whose names match a regex pattern.

If such event(s) exist, pop one event from each event queue that satisfies the condition. Otherwise, wait for an event that satisfies the condition to occur, with timeout.

Results are sorted by timestamp in ascending order.

Parameters

- **regex_pattern** – The regular expression pattern that an event name should match in order to be popped.
- **timeout** – Number of seconds to wait for events in case no event matching the condition exits when the function is called.

Returns Events whose names match a regex pattern. Empty if none exist and the wait timed out.

Raises

- *IllegalStateError* – Raised if pop is called before the dispatcher starts polling.
- `queue.Empty` – Raised if no event was found before time out.

register_handler(*handler*, *event_name*, *args*)
Registers an event handler.

One type of event can only have one event handler associated with it.

Parameters

- **handler** – The event handler function to be registered.
- **event_name** – Name of the event the handler is for.
- **args** – User arguments to be passed to the handler when it's called.

Raises

- *IllegalStateError* – Raised if attempts to register a handler after the dispatcher starts running.
- *DuplicateError* – Raised if attempts to register more than one handler for one type of event.

start()
Starts the event dispatcher.

Initiates executor and start polling events.

Raises *IllegalStateError* – Can't start a dispatcher again when it's already running.

wait_for_event(*event_name*, *predicate*, *timeout=60*, **args*, ***kwargs*)
Wait for an event that satisfies a predicate to appear.

Continuously pop events of a particular name and check against the predicate until an event that satisfies the predicate is popped or timed out. Note this will remove all the events of the same name that do not satisfy the predicate in the process.

Parameters

- **event_name** – Name of the event to be popped.
- **predicate** – A function that takes an event and returns True if the predicate is satisfied, False otherwise.
- **timeout** – Number of seconds to wait.
- ***args** – Optional positional args passed to predicate().
- ****kwargs** – Optional keyword args passed to predicate().

Returns The event that satisfies the predicate.

Raises `queue.Empty` – Raised if no event that satisfies the predicate was found before time out.

exception mobly.controllers.android_device_lib.event_dispatcher.**EventDispatcherError**
Bases: `Exception`

exception mobly.controllers.android_device_lib.event_dispatcher.**IllegalStateError**
Bases: *mobly.controllers.android_device_lib.event_dispatcher.EventDispatcherError*

Raise when user tries to put event_dispatcher into an illegal state.

mobly.controllers.android_device_lib.fastboot module

class mobly.controllers.android_device_lib.fastboot.**FastbootProxy**(*serial=''*)
Bases: `object`

Proxy class for fastboot.

For syntactic reasons, the '-' in fastboot commands need to be replaced with '_'. Can directly execute fastboot commands on an object: >> fb = FastbootProxy(<serial>) >> fb.devices() # will return the console output of "fastboot devices".

args(**args*)

mobly.controllers.android_device_lib.fastboot.**exe_cmd**(**cmds*)
Executes commands in a new shell. Directing stderr to PIPE.

This is fastboot's own exe_cmd because of its peculiar way of writing non-error info to stderr.

Parameters **cmds** – A sequence of commands and arguments.

Returns The output of the command run.

Raises `Exception` – An error occurred during the command execution.

mobly.controllers.android_device_lib.jsonrpc_client_base module

Base class for clients that communicate with apps over a JSON RPC interface.

The JSON protocol expected by this module is:

```
Request:
{
  "id": <monotonically increasing integer containing the ID of
       this request>
  "method": <string containing the name of the method to execute>
  "params": <JSON array containing the arguments to the method>
}

Response:
{
  "id": <int id of request that this response maps to>,
  "result": <Arbitrary JSON object containing the result of
         executing the method. If the method could not be
         executed or returned void, contains 'null'.>,
  "error": <String containing the error thrown by executing the
        method. If no error occurred, contains 'null'.>
  "callback": <String that represents a callback ID used to
         identify events associated with a particular
         CallbackHandler object.>
}
```

class mobly.controllers.android_device_lib.jsonrpc_client_base.**JsonRpcClientBase**(*app_name*, *ad*)

Bases: abc.ABC

Base class for jsonrpc clients that connect to remote servers.

Connects to a remote device running a jsonrpc-compatible app. Before opening a connection a port forward must be setup to go over usb. This be done using adb.forward([local, remote]). Once the port has been forwarded it can be used in this object as the port of communication.

host_port
(int) The host port of this RPC client.

device_port
(int) The device port of this RPC client.

app_name
(str) The user-visible name of the app being communicated with.

uid
(int) The uid of this session.

clear_host_port()
Stops the adb port forwarding of the host port used by this client.

close_socket_connection()
Closes the socket connection to the server.

connect(*uid=-1*, *cmd='initiate'*)
Opens a connection to a JSON RPC server.

Opens a connection to a remote client. The connection attempt will time out if it takes longer than _SOCKET_CONNECTION_TIMEOUT seconds. Each subsequent operation over this socket will time out after _SOCKET_READ_TIMEOUT seconds as well.

Parameters

- **uid** – int, The uid of the session to join, or UNKNOWN_UID to start a new session.
- **cmd** – JsonRpcCommand, The command to use for creating the connection.

Raises

- IOError – Raised when the socket times out from io error
- socket.timeout – Raised when the socket waits to long for connection.
- ProtocolError – Raised when there is an error in the protocol.

disable_hidden_api_blacklist()
If necessary and possible, disables hidden api blacklist.

disconnect()
Close the connection to the snippet server on the device.

This is a unilateral disconnect from the client side, without tearing down the snippet server running on the device.

The connection to the snippet server can be re-established by calling *SnippetClient.restore_app_connection*.

restore_app_connection(*port=None*)
Reconnects to the app after device USB was disconnected.

Instead of creating new instance of the client:

- Uses the given port (or finds a new available host_port if none is

given). - Tries to connect to remote server with selected port.

Must be implemented by subclasses.

Parameters **port** – If given, this is the host port from which to connect to remote device port. If not provided, find a new available port as host port.

Raises

- AppRestoreConnectionError – When the app was not able to be
- reconnected.

set_snippet_client_verbose_logging(*verbose*)
Switches verbose logging. True for logging full RPC response.

By default it will only write max_rpc_return_value_length for Rpc return strings. If you need to see full message returned from Rpc, please turn on verbose logging.

max_rpc_return_value_length will set to 1024 by default, the length contains full Rpc response in Json format, included 1st element "id".

Parameters **verbose** – bool. If True, turns on verbose logging, if False turns off

start_app_and_connect()
Starts the server app on the android device and connects to it.

After this, the self.host_port and self.device_port attributes must be set.

Must be implemented by subclasses.

Raises AppStartError – When the app was not able to be started.

stop_app()
Kills any running instance of the app.

Must be implemented by subclasses.

class mobly.controllers.android_device_lib.jsonrpc_client_base.**JsonRpcCommand**
Bases: object

Commands that can be invoked on all jsonrpc clients.

INIT: Initializes a new session. CONTINUE: Creates a connection.

CONTINUE = 'continue'

INIT = 'initiate'

mobly.controllers.android_device_lib.jsonrpc_shell_base module

Shared library for frontends to jsonrpc servers.

exception mobly.controllers.android_device_lib.jsonrpc_shell_base.**Error**
Bases: Exception

class mobly.controllers.android_device_lib.jsonrpc_shell_base.**JsonRpcShellBase**
Bases: object

load_device(*serial=None*)
Creates an AndroidDevice for the given serial number.

If no serial is given, it will read from the ANDROID_SERIAL environmental variable. If the environmental variable is not set, then it will read from 'adb devices' if there is only one.

main(*serial=None*)

start_console()

mobly.controllers.android_device_lib.service_manager module

Module for the manager of services.

exception mobly.controllers.android_device_lib.service_manager.**Error**(*ad*, *msg*)
Bases: *mobly.controllers.android_device_lib.errors.DeviceError*

Root error type for this module.

class mobly.controllers.android_device_lib.service_manager.**ServiceManager**(*device*)
Bases: object

Manager for services of AndroidDevice.

A service is a long running process that involves an Android device, like adb logcat or Snippet.

create_output_excerpts_all(*test_info*)
Creates output excerpts from all services.

This calls *create_output_excerpts* on all registered services.

Parameters **test_info** – RuntimeTestInfo, the test info associated with the scope of the excerpts.

Returns

Dict, keys are the names of the services, values are the paths to the excerpt files created by the corresponding services.

for_each(*func*)
Executes a function with all registered services.

Parameters **func** – function, the function to execute. This function should take a service object as args.

has_service_by_name(*name*)
Checks if the manager has a service registered with a specific name.

Parameters **name** – string, the name to look for.

Returns True if a service is registered with the specified name, False otherwise.

is_any_alive
True if any service is alive; False otherwise.

list_live_services()
Lists the aliases of all the services that are alive.

Order of this list is determined by the order the services are registered in.

Returns list of strings, the aliases of the services that are running.

pause_all()
Pauses all service instances.

Services will be paused in the reverse order they were registered.

register(*alias*, *service_class*, *configs=None*, *start_service=True*)
Registers a service.

This will create a service instance, starts the service, and adds the instance to the mananger.

Parameters

- **alias** – string, the alias for this instance.
- **service_class** – class, the service class to instantiate.
- **configs** – (optional) config object to pass to the service class's constructor.
- **start_service** – bool, whether to start the service instance or not. Default is True.

resume_all()
Resumes all service instances.

Services will be resumed in the order they were registered.

resume_services(*service_alises*)
Resumes the specified services.

Services will be resumed in the order specified by the input list.

Parameters **service_alises** – list of strings, the names of services to start.

start_all()
Starts all inactive service instances.

Services will be started in the order they were registered.

start_services(*service_alises*)
Starts the specified services.

Services will be started in the order specified by the input list. No-op for services that are already running.

Parameters **service_alises** – list of strings, the aliases of services to start.

stop_all()
Stops all active service instances.

Services will be stopped in the reverse order they were registered.

unregister(*alias*)
Unregisters a service instance.

Stops a service and removes it from the manager.

Parameters **alias** – string, the alias of the service instance to unregister.

unregister_all()
Safely unregisters all active instances.

Errors occurred here will be recorded but not raised.

mobly.controllers.android_device_lib.sl4a_client module

JSON RPC interface to android scripting engine.

class mobly.controllers.android_device_lib.sl4a_client.**Sl4aClient**(*ad*)
Bases: *mobly.controllers.android_device_lib.jsonrpc_client_base.JsonRpcClientBase*

A client for interacting with SL4A using Mobly Snippet Lib.

Extra public attributes: ed: Event dispatcher instance for this sl4a client.

restore_app_connection(*port=None*)
Restores the sl4a after device got disconnected.

Instead of creating new instance of the client:

- Uses the given port (or find a new available host_port if none is

given). - Tries to connect to remote server with selected port.

Parameters **port** – If given, this is the host port from which to connect to remote device port. If not provided, find a new available port as host port.

Raises AppRestoreConnectionError – When the app was not able to be started.

start_app_and_connect()
Overrides superclass.

stop_app()
Overrides superclass.

stop_event_dispatcher()

mobly.controllers.android_device_lib.snippet_client module

JSON RPC interface to Mobly Snippet Lib.

class mobly.controllers.android_device_lib.snippet_client.**SnippetClient**(*package*, *ad*)
Bases: *mobly.controllers.android_device_lib.jsonrpc_client_base.JsonRpcClientBase*

A client for interacting with snippet APKs using Mobly Snippet Lib.

DEPRECATED: Use mobly.controllers.android_device_lib.snippet_client_v2.SnippetClientV2 instead.

See superclass documentation for a list of public attributes.

For a description of the launch protocols, see the documentation in mobly-snippet-lib, SnippetRunner.java.

help(*print_output=True*)

Calls the help RPC, which returns the list of RPC calls available.

This RPC should normally be used in an interactive console environment where the output should be printed instead of returned. Otherwise, newlines will be escaped, which will make the output difficult to read.

Parameters **print_output** – A bool for whether the output should be printed.

Returns

A str containing the help output otherwise None if print_output wasn't set.

is_alive

Does the client have an active connection to the snippet server.

restore_app_connection(*port=None*)

Restores the app after device got reconnected.

Instead of creating new instance of the client:

- Uses the given port (or find a new available host_port if none is

given). - Tries to connect to remote server with selected port.

Parameters **port** – If given, this is the host port from which to connect to remote device port. If not provided, find a new available port as host port.

Raises AppRestoreConnectionError – When the app was not able to be started.

start_app_and_connect()

Starts snippet apk on the device and connects to it.

This wraps the main logic with safe handling

Raises AppStartPreCheckError, when pre-launch checks fail.

stop_app()

Kills any running instance of the app.

Must be implemented by subclasses.

user_id

The user id to use for this snippet client.

This value is cached and, once set, does not change through the lifecycles of this snippet client object. This caching also reduces the number of adb calls needed.

Because all the operations of the snippet client should be done for a partucular user.

mobly.controllers.android_device_lib.snippet_event module

class mobly.controllers.android_device_lib.snippet_event.**SnippetEvent**(*callback_id*, *name*, *creation_time*, *data*)

Bases: object

The class that represents callback events for mobly snippet library.

DEPRECATED: Use mobly.snippet.callback_event.CallbackEvent instead.

callback_id
string, the callback ID associated with the event.

name
string, the name of the event.

creation_time
int, the epoch time when the event is created on the Rpc server side.

data
dictionary, the data held by the event. Can be None.

mobly.controllers.android_device_lib.snippet_event.**from_dict**(*event_dict*)
Create a SnippetEvent object from a dictionary.

DEPRECATED: Use mobly.snippet.callback_event.from_dict instead.

Parameters **event_dict** – a dictionary representing an event.

Returns A SnippetEvent object.

Module contents

mobly.controllers.attenuator_lib package

Submodules

mobly.controllers.attenuator_lib.minicircuits module

This module has the class for controlling Mini-Circuits RCDAT series attenuators over Telnet.

See http://www.minicircuits.com/softwaredownload/Prog_Manual-6-Programmable_Attenuator.pdf

class mobly.controllers.attenuator_lib.minicircuits.**AttenuatorDevice**(*path_count=1*)

Bases: object

This provides a specific telnet-controlled implementation of AttenuatorDevice for Mini-Circuits RC-DAT attenuators.

path_count
The number of signal attenuation path this device has.

close()
Closes a telnet connection to the desired attenuator device.

This should be called as part of any teardown procedure prior to the attenuator instrument leaving scope.

get_atten(*idx=0*)
This function returns the current attenuation from an attenuator at a given index in the instrument.

Parameters **idx** – This zero-based index is the identifier for a particular attenuator in an instrument.

Raises Error – The underlying telnet connection to the instrument is not open.

Returns A float that is the current attenuation value.

is_open
This function returns the state of the telnet connection to the underlying AttenuatorDevice.

Returns True if there is a successfully open connection to the AttenuatorDevice.

open(*host*, *port=23*)
Opens a telnet connection to the desired AttenuatorDevice and queries basic information.

Parameters

- **host** – A valid hostname (IP address or DNS-resolvable name) to an MC-DAT attenuator instrument.
- **port** – An optional port number (defaults to telnet default 23)

set_atten(*idx*, *value*)
Sets the attenuation value for a particular signal path.

Parameters

- **idx** – Zero-based index int which is the identifier for a particular signal path in an instrument. For instruments that only has one channel, this is ignored by the device.
- **value** – A float that is the attenuation value to set.

Raises

- Error – The underlying telnet connection to the instrument is not open.
- IndexError – The index of the attenuator is greater than the maximum index of the underlying instrument.
- ValueError – The requested set value is greater than the maximum attenuation value.

mobly.controllers.attenuator_lib.telnet_scpi_client module

Helper module for common telnet capability to communicate with AttenuatorDevice(s).

User code shouldn't need to directly access this class.

class mobly.controllers.attenuator_lib.telnet_scpi_client.**TelnetScpiClient**(*tx_cmd_separator='n'*, *rx_cmd_separator='n'*, *prompt=''*)

Bases: object

This is an internal helper class for Telnet+SCPI command-based instruments. It should only be used by those implemention control libraries and not by any user code directly.

close()

cmd(*cmd_str*, *wait_ret=True*)

is_open

open(*host*, *port=23*)

Module contents

mobly.controllers.sniffer_lib package

Subpackages

mobly.controllers.sniffer_lib.local package

Submodules

mobly.controllers.sniffer_lib.local.local_base module

Class for Local sniffers - i.e. running on the local machine.

This class provides configuration for local interfaces but leaves the actual capture (sniff) to sub-classes.

class mobly.controllers.sniffer_lib.local.local_base.**SnifferLocalBase**(*interface*, *logger*, *base_configs=None*)

Bases: *mobly.controllers.sniffer.Sniffer*

This class defines the common behaviors of WLAN sniffers running on WLAN interfaces of the local machine.

Specific mechanisms to capture packets over the local WLAN interfaces are implemented by sub-classes of this class - i.e. it is not a final class.

get_capture_file()
The sniffer places a capture in the logger directory. This function enables the caller to obtain the path of that capture.

Returns The full path of the current or last capture.

get_interface()
See base class documentation

get_type()
See base class documentation

start_capture(*override_configs=None*, *additional_args=None*, *duration=None*, *packet_count=None*)
See base class documentation

stop_capture()
See base class documentation

wait_for_capture(*timeout=None*)
See base class documentation

mobly.controllers.sniffer_lib.local.tcpdump module

class mobly.controllers.sniffer_lib.local.tcpdump.**Sniffer**(*config_path*, *logger*, *base_configs=None*)

Bases: *mobly.controllers.sniffer_lib.local.local_base.SnifferLocalBase*

This class defines a sniffer which uses tcpdump as its back-end

get_descriptor()
See base class documentation

get_subtype()
See base class documentation

mobly.controllers.sniffer_lib.local.tshark module

class mobly.controllers.sniffer_lib.local.tshark.**Sniffer**(*config_path*, *logger*, *base_configs=None*)
Bases: *mobly.controllers.sniffer_lib.local.local_base.SnifferLocalBase*

This class defines a sniffer which uses tshark as its back-end

get_descriptor()
See base class documentation

get_subtype()
See base class documentation

Module contents

Module contents

1.1.1.2 Submodules

1.1.1.3 mobly.controllers.android_device module

class mobly.controllers.android_device.**AndroidDevice**(*serial=''*)
Bases: object

Class representing an android device.

Each object of this class represents one Android device in Mobly. This class provides various ways, like adb, fastboot, and Mobly snippets, to control an Android device, whether it's a real device or an emulator instance.

You can also register your own services to the device's service manager. See the docs of *service_manager* and *base_service* for details.

serial
A string that's the serial number of the Android device.

log_path
A string that is the path where all logs collected on this android device should be stored.

log
A logger adapted from root logger with an added prefix specific to an AndroidDevice instance. The default prefix is [AndroidDevice|<serial>]. Use self.debug_tag = 'tag' to use a different tag in the prefix.

adb_logcat_file_path
A string that's the full path to the adb logcat file collected, if any.

adb
An AdbProxy object used for interacting with the device via adb.

fastboot
A FastbootProxy object used for interacting with the device via fastboot.

services
ServiceManager, the manager of long-running services on the device.

adb_logcat_file_path

add_device_info(*name*, *info*)
Add information of the device to be pulled into controller info.

Adding the same info name the second time will override existing info.

Parameters

- **name** – string, name of this info.
- **info** – serializable, content of the info.

build_info
Gets the build info of this Android device, including build id and type.

This is not available if the device is in bootloader mode.

Returns A dict with the build info of this Android device, or None if the device is in bootloader mode.

debug_tag
A string that represents a device object in debug info. Default value is the device serial.

This will be used as part of the prefix of debugging messages emitted by this device object, like log lines and the message of DeviceError.

device_info
Information to be pulled into controller info.

The latest serial, model, and build_info are included. Additional info can be added via *add_device_info*.

ed
Attribute for direct access of sl4a's event dispatcher.

Not recommended. This is here for backward compatibility reasons.

Preferred: directly access *ad.services.sl4a.ed*.

generate_filename(*file_type*, *time_identifier=None*, *extension_name=None*)
Generates a name for an output file related to this device.

The name follows the pattern:

{file type},{debug_tag},{serial},{model},{time identifier}.{ext}

"debug_tag" is only added if it's different from the serial. "ext" is added if specified by user.

Parameters

- **file_type** – string, type of this file, like "logcat" etc.
- **time_identifier** – string or RuntimeTestInfo. If a *RuntimeTestInfo* is passed in, the *signature* of the test case will be used. If a string is passed in, the string itself will be used. Otherwise the current timestamp will be used.
- **extension_name** – string, the extension name of the file.

Returns String, the filename generated.

handle_reboot()
Properly manage the service life cycle when the device needs to temporarily disconnect.

The device can temporarily lose adb connection due to user-triggered reboot. Use this function to make sure the services started by Mobly are properly stopped and restored afterwards.

For sample usage, see self.reboot().

handle_usb_disconnect()
Properly manage the service life cycle when USB is disconnected.

The device can temporarily lose adb connection due to user-triggered USB disconnection, e.g. the following cases can be handled by this method:

- Power measurement: Using Monsoon device to measure battery consumption would potentially disconnect USB.
- Unplug USB so device loses connection.
- ADB connection over WiFi and WiFi got disconnected.
- Any other type of USB disconnection, as long as snippet session can be kept alive while USB disconnected (reboot caused USB disconnection is not one of these cases because snippet session cannot survive reboot. Use handle_reboot() instead).

Use this function to make sure the services started by Mobly are properly reconnected afterwards.

Just like the usage of self.handle_reboot(), this method does not automatically detect if the disconnection is because of a reboot or USB disconnect. Users of this function should make sure the right handle_* function is used to handle the correct type of disconnection.

This method also reconnects snippet event client. Therefore, the callback objects created (by calling Async RPC methods) before disconnection would still be valid and can be used to retrieve RPC execution result after device got reconnected.

Example Usage:

```
with ad.handle_usb_disconnect():
  try:
    # User action that triggers USB disconnect, could throw
    # exceptions.
    do_something()
  finally:
    # User action that triggers USB reconnect
    action_that_reconnects_usb()
    # Make sure device is reconnected before returning from this
    # context
    ad.adb.wait_for_device(timeout=SOME_TIMEOUT)
```

has_active_service
True if any service is running on the device.

A service can be a snippet or logcat collection.

is_adb_detectable()
Checks if USB is on and device is ready by verifying adb devices.

is_adb_root
True if adb is running as root for this device.

is_boot_completed()
Checks if device boot is completed by verifying system property.

is_bootloader
True if the device is in bootloader mode.

is_emulator
Whether this device is probably an emulator.

Returns True if this is probably an emulator.

is_rootable

load_config(*config*)
Add attributes to the AndroidDevice object based on config.

Parameters **config** – A dictionary representing the configs.

Raises Error – The config is trying to overwrite an existing attribute.

load_snippet(*name*, *package*, *config=None*)
Starts the snippet apk with the given package name and connects.

Examples:

```
ad.load_snippet(
    name='maps', package='com.google.maps.snippets')
ad.maps.activateZoom('3')
```

Parameters

- **name** – string, the attribute name to which to attach the snippet client. E.g. *name='maps'* attaches the snippet client to *ad.maps*.
- **package** – string, the package name of the snippet apk to connect to.
- **config** – snippet_client_v2.Config, the configuration object for controlling the snippet behaviors. See the docstring of the *Config* class for supported configurations.

Raises SnippetError – Illegal load operations are attempted.

log_path
A string that is the path for all logs collected from this device.

model
The Android code name for the device.

reboot()
Reboots the device.

Generally one should use this method to reboot the device instead of directly calling *adb.reboot*. Because this method gracefully handles the teardown and restoration of running services.

This method is blocking and only returns when the reboot has completed and the services restored.

Raises Error – Waiting for completion timed out.

root_adb()
Change adb to root mode for this device if allowed.

If executed on a production build, adb will not be switched to root mode per security restrictions.

run_iperf_client(*server_host*, *extra_args=''*)
Start iperf client on the device.

Return status as true if iperf client start successfully. And data flow information as results.

Parameters

- **server_host** – Address of the iperf server.

- **extra_args** – A string representing extra arguments for iperf client, e.g. '-i 1 -t 30'.

Returns true if iperf client start successfully. results: results have data flow information

Return type status

serial
The serial number used to identify a device.

This is essentially the value used for adb's *-s* arg, which means it can be a network address or USB bus number.

sl4a
Attribute for direct access of sl4a client.

Not recommended. This is here for backward compatibility reasons.

Preferred: directly access *ad.services.sl4a*.

take_bug_report(*test_name=None*, *begin_time=None*, *timeout=300*, *destination=None*)
Takes a bug report on the device and stores it in a file.

Parameters

- **test_name** – Name of the test method that triggered this bug report.
- **begin_time** – Timestamp of when the test started. If not set, then this will default to the current time.
- **timeout** – float, the number of seconds to wait for bugreport to complete, default is 5min.
- **destination** – string, path to the directory where the bugreport should be saved.

Returns A string that is the absolute path to the bug report on the host.

take_screenshot(*destination*, *prefix='screenshot'*)
Takes a screenshot of the device.

Parameters

- **destination** – string, full path to the directory to save in.
- **prefix** – string, prefix file name of the screenshot.

Returns string, full path to the screenshot file on the host.

unload_snippet(*name*)
Stops a snippet apk.

Parameters **name** – The attribute name the snippet server is attached with.

Raises `SnippetError` – The given snippet name is not registered.

update_serial(*new_serial*)
Updates the serial number of a device.

The "serial number" used with adb's *-s* arg is not necessarily the actual serial number. For remote devices, it could be a combination of host names and port numbers.

This is used for when such identifier of remote devices changes during a test. For example, when a remote device reboots, it may come back with a different serial number.

This is NOT meant for switching the object to represent another device.

We intentionally did not make it a regular setter of the serial property so people don't accidentally call this without understanding the consequences.

Parameters **new_serial** – string, the new serial number for the same device.

Raises DeviceError – tries to update serial when any service is running.

wait_for_boot_completion(*timeout=900*)

Waits for Android framework to broadcast ACTION_BOOT_COMPLETED.

This function times out after 15 minutes.

Parameters **timeout** – float, the number of seconds to wait before timing out. If not specified, no timeout takes effect.

class mobly.controllers.android_device.**AndroidDeviceLoggerAdapter**(*logger*, *extra*)

Bases: logging.LoggerAdapter

A wrapper class that adds a prefix to each log line.

Usage:

```
my_log = AndroidDeviceLoggerAdapter(logging.getLogger(), {
  'tag': <custom tag>
})
```

Then each log line added by my_log will have a prefix '[AndroidDevice|<tag>]'

process(*msg*, *kwargs*)

Process the logging message and keyword arguments passed in to a logging call to insert contextual information. You can either manipulate the message itself, the keyword args or both. Return the message and kwargs modified (or not) to suit your needs.

Normally, you'll only need to override this one method in a LoggerAdapter subclass for your specific needs.

class mobly.controllers.android_device.**BuildInfoConstants**(*build_info_key*, *system_prop_key*)

Bases: enum.Enum

Enums for build info constants used for AndroidDevice build info.

build_info_key

The key used for the build_info dictionary in AndroidDevice.

system_prop_key

The key used for getting the build info from system properties.

BUILD_CHARACTERISTICS = ('build_characteristics', 'ro.build.characteristics')

BUILD_FINGERPRINT = ('build_fingerprint', 'ro.build.fingerprint')

BUILD_ID = ('build_id', 'ro.build.id')

BUILD_PRODUCT = ('build_product', 'ro.build.product')

BUILD_TYPE = ('build_type', 'ro.build.type')

BUILD_VERSION_CODENAME = ('build_version_codename', 'ro.build.version.codename')

BUILD_VERSION_INCREMENTAL = ('build_version_incremental', 'ro.build.version.incrementa

BUILD_VERSION_SDK = ('build_version_sdk', 'ro.build.version.sdk')

DEBUGGABLE = ('debuggable', 'ro.debuggable')

HARDWARE = ('hardware', 'ro.hardware')

PRODUCT_NAME = ('product_name', 'ro.product.name')

mobly.controllers.android_device.**create**(*configs*)
Creates AndroidDevice controller objects.

Parameters **configs** – A list of dicts, each representing a configuration for an Android device.

Returns A list of AndroidDevice objects.

mobly.controllers.android_device.**destroy**(*ads*)
Cleans up AndroidDevice objects.

Parameters **ads** – A list of AndroidDevice objects.

mobly.controllers.android_device.**filter_devices**(*ads*, *func*)
Finds the AndroidDevice instances from a list that match certain conditions.

Parameters

- **ads** – A list of AndroidDevice instances.
- **func** – A function that takes an AndroidDevice object and returns True if the device satisfies the filter condition.

Returns A list of AndroidDevice instances that satisfy the filter condition.

mobly.controllers.android_device.**get_all_instances**(*include_fastboot=False*)
Create AndroidDevice instances for all attached android devices.

Parameters **include_fastboot** – Whether to include devices in bootloader mode or not.

Returns A list of AndroidDevice objects each representing an android device attached to the computer.

mobly.controllers.android_device.**get_device**(*ads*, ***kwargs*)
Finds a unique AndroidDevice instance from a list that has specific attributes of certain values.

Example

get_device(android_devices, label='foo', phone_number='1234567890') get_device(android_devices, model='angler')

Parameters

- **ads** – A list of AndroidDevice instances.
- **kwargs** – keyword arguments used to filter AndroidDevice instances.

Returns The target AndroidDevice instance.

Raises Error – None or more than one device is matched.

mobly.controllers.android_device.**get_devices**(*ads*, ***kwargs*)
Finds a list of AndroidDevice instance from a list that has specific attributes of certain values.

Example

get_devices(android_devices, label='foo', phone_number='1234567890') get_devices(android_devices, model='angler')

Parameters

- **ads** – A list of AndroidDevice instances.
- **kwargs** – keyword arguments used to filter AndroidDevice instances.

Returns A list of target AndroidDevice instances.

Raises `Error` – No devices are matched.

`mobly.controllers.android_device.`**`get_info`**(*ads*)
Get information on a list of AndroidDevice objects.

Parameters **`ads`** – A list of AndroidDevice objects.

Returns A list of dict, each representing info for an AndroidDevice objects.

`mobly.controllers.android_device.`**`get_instances`**(*serials*)
Create AndroidDevice instances from a list of serials.

Parameters **`serials`** – A list of android device serials.

Returns A list of AndroidDevice objects.

`mobly.controllers.android_device.`**`get_instances_with_configs`**(*configs*)
Create AndroidDevice instances from a list of dict configs.

Each config should have the required key-value pair 'serial'.

Parameters **`configs`** – A list of dicts each representing the configuration of one android device.

Returns A list of AndroidDevice objects.

`mobly.controllers.android_device.`**`list_adb_devices`**()
List all android devices connected to the computer that are detected by adb.

Returns A list of android device serials. Empty if there's none.

`mobly.controllers.android_device.`**`list_adb_devices_by_usb_id`**()
List the usb id of all android devices connected to the computer that are detected by adb.

Returns A list of strings that are android device usb ids. Empty if there's none.

`mobly.controllers.android_device.`**`list_fastboot_devices`**()
List all android devices connected to the computer that are in in fastboot mode. These are detected by fastboot.

This function doesn't raise any error if *fastboot* binary doesn't exist, because *FastbootProxy* itself doesn't raise any error.

Returns A list of android device serials. Empty if there's none.

`mobly.controllers.android_device.`**`parse_device_list`**(*device_list_str*, *key=None*)
Parses a byte string representing a list of devices.

The string is generated by calling either adb or fastboot. The tokens in each string is tab-separated.

Parameters

- **`device_list_str`** – Output of adb or fastboot.
- **`key`** – The token that signifies a device in device_list_str. Only devices with the specified key in device_list_str are parsed, such as 'device' or 'fastbootd'. If not specified, all devices listed are parsed.

Returns A list of android device serial numbers.

`mobly.controllers.android_device.`**`take_bug_reports`**(*ads*, *test_name=None*, *begin_time=None*, *destination=None*)
Takes bug reports on a list of android devices.

If you want to take a bug report, call this function with a list of android_device objects in on_fail. But reports will be taken on all the devices in the list concurrently. Bug report takes a relative long time to take, so use this cautiously.

Parameters

- **ads** – A list of AndroidDevice instances.
- **test_name** – Name of the test method that triggered this bug report. If None, the default name "bugreport" will be used.
- **begin_time** – timestamp taken when the test started, can be either string or int. If None, the current time will be used.
- **destination** – string, path to the directory where the bugreport should be saved.

1.1.1.4 mobly.controllers.attenuator module

Controller module for attenuators.

Sample Config:

```
"Attenuator": [
    {
        "address": "192.168.1.12",
        "port": 23,
        "model": "minicircuits",
        "paths": ["AP1-2G", "AP1-5G", "AP2-2G", "AP2-5G"]
    },
    {
        "address": "192.168.1.14",
        "port": 23,
        "model": "minicircuits",
        "paths": ["AP-DUT"]
    }
]
```

class mobly.controllers.attenuator.**AttenuatorPath**(*attenuation_device*, *idx=0*, *name=None*)

Bases: object

A convenience class that allows users to control each attenuator path separately as different objects, as opposed to passing in an index number to the functions of an attenuator device object.

This decouples the test code from the actual attenuator device used in the physical test bed.

For example, if a test needs to attenuate four signal paths, this allows the test to do:

```
self.attenuation_paths[0].set_atten(50)
self.attenuation_paths[1].set_atten(40)
```

instead of:

```
self.attenuators[0].set_atten(0, 50)
self.attenuators[0].set_atten(1, 40)
```

The benefit the former is that the physical test bed can use either four single-channel attenuators, or one four-channel attenuators. Whereas the latter forces the test bed to use a four-channel attenuator.

get_atten()

Gets the current attenuation setting of Attenuator.

Returns A float that is the current attenuation value. Unit is db.

get_max_atten()
Gets the max attenuation supported by the Attenuator.

Returns A float that is the max attenuation value.

set_atten(*value*)
This function sets the attenuation of Attenuator.

Parameters **value** – This is a floating point value for nominal attenuation to be set. Unit is db.

exception mobly.controllers.attenuator.**Error**
Bases: Exception

This is the Exception class defined for all errors generated by Attenuator-related modules.

mobly.controllers.attenuator.**create**(*configs*)

mobly.controllers.attenuator.**destroy**(*objs*)

1.1.1.5 mobly.controllers.iperf_server module

class mobly.controllers.iperf_server.**IPerfResult**(*result_path*)
Bases: object

avg_rate
Average receiving rate in MB/s over the entire run.

If the result is not from a success run, this property is None.

avg_receive_rate
Average receiving rate in MB/s over the entire run. This data may not exist if iperf was interrupted.

If the result is not from a success run, this property is None.

avg_send_rate
Average sending rate in MB/s over the entire run. This data may not exist if iperf was interrupted.

If the result is not from a success run, this property is None.

error

get_json()

Returns The raw json output from iPerf.

class mobly.controllers.iperf_server.**IPerfServer**(*port*, *log_path*)
Bases: object

Class that handles iperf3 operations.

start(*extra_args=''*, *tag=''*)
Starts iperf server on specified port.

Parameters

- **extra_args** – A string representing extra arguments to start iperf server with.
- **tag** – Appended to log file name to identify logs from different iperf runs.

stop()

mobly.controllers.iperf_server.**create**(*configs*)

mobly.controllers.iperf_server.**destroy**(*objs*)

1.1.1.6 mobly.controllers.monsoon module

1.1.1.7 mobly.controllers.sniffer module

class mobly.controllers.sniffer.**ActiveCaptureContext**(*sniffer*, *timeout=None*)
Bases: object

This class defines an object representing an active sniffer capture.

The object is returned by a Sniffer.start_capture() command and terminates the capture when the 'with' clause exits. It is syntactic sugar for try/finally.

exception mobly.controllers.sniffer.**ExecutionError**
Bases: *mobly.controllers.sniffer.SnifferError*

This exception is thrown when trying to configure the capture device or when trying to execute the capture operation.

When this exception is seen, it is possible that the sniffer module is run without sudo (for local sniffers) or keys are out-of-date (for remote sniffers).

exception mobly.controllers.sniffer.**InvalidDataError**
Bases: Exception

This exception is thrown when invalid configuration data is passed to a method.

exception mobly.controllers.sniffer.**InvalidOperationError**
Bases: *mobly.controllers.sniffer.SnifferError*

Certain methods may only be accessed when the instance upon which they are invoked is in a certain state. This indicates that the object is not in the correct state for a method to be called.

class mobly.controllers.sniffer.**Sniffer**(*interface*, *logger*, *base_configs=None*)
Bases: object

This class defines an object representing a sniffer.

The object defines the generic behavior of sniffers - irrespective of how they are implemented, or where they are located: on the local machine or on the remote machine.

CONFIG_KEY_CHANNEL = 'channel'

get_capture_file()
The sniffer places a capture in the logger directory. This function enables the caller to obtain the path of that capture.

Returns The full path of the current or last capture.

get_descriptor()
This function returns a string describing the sniffer. The specific string (and its format) is up to each derived sniffer type.

Returns A string describing the sniffer.

get_interface()
This function returns The interface used to configure the sniffer, e.g. 'wlan0'.

Returns The interface (string) used to configure the sniffer. Corresponds to the 'Interface' key of the sniffer configuration.

get_subtype()
This function returns the sub-type of the sniffer.

Returns The sub-type (string) of the sniffer. Corresponds to the 'SubType' key of the sniffer configuration.

get_type()
This function returns the type of the sniffer.

Returns The type (string) of the sniffer. Corresponds to the 'Type' key of the sniffer configuration.

start_capture(*override_configs=None*, *additional_args=None*, *duration=None*, *packet_count=None*)
This function starts a capture which is saved to the specified file path.

Depending on the type/subtype and configuration of the sniffer the capture may terminate on its own or may require an explicit call to the stop_capture() function.

This is a non-blocking function so a terminating function must be called either explicitly or implicitly:

- Explicitly: call either stop_capture() or wait_for_capture()
- **Implicitly: use with a with clause. The wait_for_capture()** function will be called if a duration is specified (i.e. is not None), otherwise a stop_capture() will be called.

The capture is saved to a file in the log path of the logger. Use the get_capture_file() to get the full path to the current or most recent capture.

Parameters

- **override_configs** – A dictionary which is combined with the base_configs ("BaseConfigs" in the sniffer configuration). The keys (specified by Sniffer.CONFIG_KEY_*) determine the configuration of the sniffer for this specific capture.
- **additional_args** – A string specifying additional raw command-line arguments to pass to the underlying sniffer. The interpretation of these flags is sniffer-dependent.
- **duration** – An integer specifying the number of seconds over which to capture packets. The sniffer will be terminated after this duration. Used in implicit mode when using a 'with' clause. In explicit control cases may have to be performed using a sleep+stop or as the timeout argument to the wait function.
- **packet_count** – An integer specifying the number of packets to capture before terminating. Should be used with duration to guarantee that capture terminates at some point (even if did not capture the specified number of packets).

Returns An ActiveCaptureContext process which can be used with a 'with' clause.

Raises

- *InvalidDataError* – for invalid configurations
- NoPermissionError – if an error occurs while configuring and running the sniffer.

stop_capture()
This function stops a capture and guarantees that the capture is saved to the capture file configured during the start_capture() method. Depending on the type of the sniffer the file may previously contain partial results (e.g. for a local sniffer) or may not exist until the stop_capture() method is executed (e.g. for a remote sniffer).

Depending on the type/subtype and configuration of the sniffer the capture may terminate on its own without requiring a call to this function. In such a case it is still necessary to call either this function or the wait_for_capture() function to make sure that the capture file is moved to the correct location.

Raises NoPermissionError – No permission when trying to stop a capture and save the capture file.

wait_for_capture(*timeout=None*)
This function waits for a capture to terminate and guarantees that the capture is saved to the capture file configured during the start_capture() method. Depending on the type of the sniffer the file may previously contain partial results (e.g. for a local sniffer) or may not exist until the stop_capture() method is executed (e.g. for a remote sniffer).

Depending on the type/subtype and configuration of the sniffer the capture may terminate on its own without requiring a call to this function. In such a case it is still necessary to call either this function or the stop_capture() function to make sure that the capture file is moved to the correct location.

Parameters **timeout** – An integer specifying the number of seconds to wait for the capture to terminate on its own. On expiration of the timeout the sniffer is stopped explicitly using the stop_capture() function.

Raises NoPermissionError – No permission when trying to stop a capture and save the capture file.

exception mobly.controllers.sniffer.**SnifferError**
Bases: Exception

This is the Exception class defined for all errors generated by Sniffer-related modules.

mobly.controllers.sniffer.**create**(*configs*)
Initializes the sniffer structures based on the JSON configuration. The expected keys are:

- **Type: A first-level type of sniffer. Planned to be 'local' for** sniffers running on the local machine, or 'remote' for sniffers running remotely.
- SubType: The specific sniffer type to be used.
- Interface: The WLAN interface used to configure the sniffer.
- **BaseConfigs: A dictionary specifying baseline configurations of** the sniffer. Configurations can be overridden when starting a capture. The keys must be one of the Sniffer.CONFIG_KEY_* values.

mobly.controllers.sniffer.**destroy**(*objs*)
Destroys the sniffers and terminates any ongoing capture sessions.

1.1.1.8 Module contents

1.2 Submodules

1.3 mobly.asserts module

mobly.asserts.**abort_all**(*reason*, *extras=None*)
Abort all subsequent tests, including the ones not in this test class or iteration.

Parameters

- **reason** – The reason to abort.
- **extras** – An optional field for extra information to be included in test result.

Raises signals.TestAbortAll – Abort all subsequent tests.

mobly.asserts.**abort_all_if**(*expr*, *reason*, *extras=None*)
Abort all subsequent tests, if the expression evaluates to True.

Parameters

- **expr** – The expression that is evaluated.

- **reason** – The reason to abort.
- **extras** – An optional field for extra information to be included in test result.

Raises signals.TestAbortAll – Abort all subsequent tests.

mobly.asserts.**abort_class**(*reason*, *extras=None*)
Abort all subsequent tests within the same test class in one iteration.

If one test class is requested multiple times in a test run, this can only abort one of the requested executions, NOT all.

Parameters

- **reason** – The reason to abort.
- **extras** – An optional field for extra information to be included in test result.

Raises signals.TestAbortClass – Abort all subsequent tests in a test class.

mobly.asserts.**abort_class_if**(*expr*, *reason*, *extras=None*)
Abort all subsequent tests within the same test class in one iteration, if expression evaluates to True.

If one test class is requested multiple times in a test run, this can only abort one of the requested executions, NOT all.

Parameters

- **expr** – The expression that is evaluated.
- **reason** – The reason to abort.
- **extras** – An optional field for extra information to be included in test result.

Raises signals.TestAbortClass – Abort all subsequent tests in a test class.

mobly.asserts.**assert_almost_equal**(*first*, *second*, *places=None*, *msg=None*, *delta=None*, *extras=None*)
Asserts that first is almost equal to second.

Fails if the two objects are unequal as determined by their difference rounded to the given number of decimal places (default 7) and comparing to zero, or by comparing that the difference between the two objects is more than the given delta. If the two objects compare equal then they automatically compare almost equal.

Parameters

- **first** – The first value to compare.
- **second** – The second value to compare.
- **places** – How many decimal places to take into account for comparison. Note that decimal places (from zero) are usually not the same as significant digits (measured from the most significant digit).
- **msg** – A string that adds additional info about the failure.
- **delta** – Delta to use for comparison instead of decimal places.
- **extras** – An optional field for extra information to be included in test result.

mobly.asserts.**assert_count_equal**(*first*, *second*, *msg=None*, *extras=None*)
Asserts that two iterables have the same elements, the same number of times, without regard to order.

Similar to assert_equal(Counter(list(first)), Counter(list(second))).

Parameters

- **first** – The first iterable to compare.

- **second** – The second iterable to compare.
- **msg** – A string that adds additional info about the failure.
- **extras** – An optional field for extra information to be included in test result.

Example

assert_count_equal([0, 1, 1], [1, 0, 1]) passes the assertion. assert_count_equal([0, 0, 1], [0, 1]) raises an assertion error.

mobly.asserts.**assert_equal**(*first*, *second*, *msg=None*, *extras=None*)
Asserts the equality of objects, otherwise fail the test.

Error message is "first != second" by default. Additional explanation can be supplied in the message.

Parameters

- **first** – The first object to compare.
- **second** – The second object to compare.
- **msg** – A string that adds additional info about the failure.
- **extras** – An optional field for extra information to be included in test result.

mobly.asserts.**assert_false**(*expr*, *msg*, *extras=None*)
Assert an expression evaluates to false, otherwise fail the test.

Parameters

- **expr** – The expression that is evaluated.
- **msg** – A string explaining the details in case of failure.
- **extras** – An optional field for extra information to be included in test result.

mobly.asserts.**assert_greater**(*a*, *b*, *msg=None*, *extras=None*)
Asserts that a > b.

mobly.asserts.**assert_greater_equal**(*a*, *b*, *msg=None*, *extras=None*)
Asserts that a >= b.

mobly.asserts.**assert_in**(*member*, *container*, *msg=None*, *extras=None*)
Asserts that member is in container.

mobly.asserts.**assert_is**(*expr1*, *expr2*, *msg=None*, *extras=None*)
Asserts that expr1 is expr2.

mobly.asserts.**assert_is_instance**(*obj*, *cls*, *msg=None*, *extras=None*)
Asserts that obj is an instance of cls.

mobly.asserts.**assert_is_none**(*obj*, *msg=None*, *extras=None*)
Asserts that obj is None.

mobly.asserts.**assert_is_not**(*expr1*, *expr2*, *msg=None*, *extras=None*)
Asserts that expr1 is not expr2.

mobly.asserts.**assert_is_not_none**(*obj*, *msg=None*, *extras=None*)
Asserts that obj is not None.

mobly.asserts.**assert_less**(*a*, *b*, *msg=None*, *extras=None*)
Asserts that a < b.

mobly.asserts.**assert_less_equal**(*a*, *b*, *msg=None*, *extras=None*)
Asserts that a <= b.

mobly.asserts.**assert_not_almost_equal**(*first*, *second*, *places=None*, *msg=None*, *delta=None*, *extras=None*)
Asserts that first is not almost equal to second.

Parameters

- **first** – The first value to compare.
- **second** – The second value to compare.
- **places** – How many decimal places to take into account for comparison. Note that decimal places (from zero) are usually not the same as significant digits (measured from the most significant digit).
- **msg** – A string that adds additional info about the failure.
- **delta** – Delta to use for comparison instead of decimal places.
- **extras** – An optional field for extra information to be included in test result.

mobly.asserts.**assert_not_equal**(*first*, *second*, *msg=None*, *extras=None*)
Asserts that first is not equal (!=) to second.

mobly.asserts.**assert_not_in**(*member*, *container*, *msg=None*, *extras=None*)
Asserts that member is not in container.

mobly.asserts.**assert_not_is_instance**(*obj*, *cls*, *msg=None*, *extras=None*)
Asserts that obj is not an instance of cls.

mobly.asserts.**assert_not_regex**(*text*, *unexpected_regex*, *msg=None*, *extras=None*)
Fails the test if the text matches the regular expression.

mobly.asserts.**assert_raises**(*expected_exception*, *extras=None*, **args*, ***kwargs*)
Assert that an exception is raised when a function is called.

If no exception is raised, test fail. If an exception is raised but not of the expected type, the exception is let through.

This should only be used as a context manager:

with assert_raises(Exception): func()

Parameters

- **expected_exception** – An exception class that is expected to be raised.
- **extras** – An optional field for extra information to be included in test result.

mobly.asserts.**assert_raises_regex**(*expected_exception*, *expected_regex*, *extras=None*, **args*, ***kwargs*)
Assert that an exception is raised when a function is called.

If no exception is raised, test fail. If an exception is raised but not of the expected type, the exception is let through. If an exception of the expected type is raised but the error message does not match the expected_regex, test fail.

This should only be used as a context manager:

with assert_raises(Exception): func()

Parameters

- **expected_exception** – An exception class that is expected to be raised.
- **extras** – An optional field for extra information to be included in test result.

mobly.asserts.**assert_regex**(*text*, *expected_regex*, *msg=None*, *extras=None*)
Fails the test unless the text matches the regular expression.

mobly.asserts.**assert_true**(*expr*, *msg*, *extras=None*)
Assert an expression evaluates to true, otherwise fail the test.

Parameters

- **expr** – The expression that is evaluated.
- **msg** – A string explaining the details in case of failure.
- **extras** – An optional field for extra information to be included in test result.

mobly.asserts.**explicit_pass**(*msg*, *extras=None*)
Explicitly pass a test.

This will pass the test explicitly regardless of any other error happened in the test body. E.g. even if errors have been recorded with *expects*, the test will still be marked pass if this is called.

A test without uncaught exception will pass implicitly so this should be used scarcely.

Parameters

- **msg** – A string explaining the details of the passed test.
- **extras** – An optional field for extra information to be included in test result.

Raises `signals.TestPass` – Mark a test as passed.

mobly.asserts.**fail**(*msg*, *extras=None*)
Explicitly fail a test.

Parameters

- **msg** – A string explaining the details of the failure.
- **extras** – An optional field for extra information to be included in test result.

Raises `signals.TestFailure` – Mark a test as failed.

mobly.asserts.**skip**(*reason*, *extras=None*)
Skip a test.

Parameters

- **reason** – The reason this test is skipped.
- **extras** – An optional field for extra information to be included in test result.

Raises `signals.TestSkip` – Mark a test as skipped.

mobly.asserts.**skip_if**(*expr*, *reason*, *extras=None*)
Skip a test if expression evaluates to True.

Parameters

- **expr** – The expression that is evaluated.
- **reason** – The reason this test is skipped.
- **extras** – An optional field for extra information to be included in test result.

1.4 mobly.base_instrumentation_test module

class mobly.base_instrumentation_test.**BaseInstrumentationTestClass**(*configs*)
Bases: *mobly.base_instrumentation_test.InstrumentationTestMixin*, *mobly.base_test.BaseTestClass*

Base class for all instrumentation test classes to inherit from.

This class extends the BaseTestClass to add functionality to run and parse the output of instrumentation runs.

DEFAULT_INSTRUMENTATION_OPTION_PREFIX
string, the default prefix for instrumentation params contained within user params.

DEFAULT_INSTRUMENTATION_ERROR_MESSAGE
string, the default error message to set if something has prevented something in the instrumentation test run from completing properly.

class mobly.base_instrumentation_test.**InstrumentationTestMixin**
Bases: object

A mixin for Mobly test classes to inherit from for instrumentation tests.

This class should be used in a subclass of both BaseTestClass and this class in order to provide instrumentation test capabilities. This mixin is explicitly for the case where the underlying BaseTestClass cannot be replaced with BaseInstrumentationTestClass. In general, prefer using BaseInstrumentationTestClass instead.

DEFAULT_INSTRUMENTATION_OPTION_PREFIX
string, the default prefix for instrumentation params contained within user params.

DEFAULT_INSTRUMENTATION_ERROR_MESSAGE
string, the default error message to set if something has prevented something in the instrumentation test run from completing properly.

DEFAULT_INSTRUMENTATION_ERROR_MESSAGE = 'instrumentation run exited unexpectedly'

DEFAULT_INSTRUMENTATION_OPTION_PREFIX = 'instrumentation_option_'

parse_instrumentation_options(*parameters=None*)
Returns the options for the instrumentation test from user_params.

By default, this method assume that the correct instrumentation options all start with DEFAULT_INSTRUMENTATION_OPTION_PREFIX.

Parameters **parameters** – dict, the key value pairs representing an assortment of parameters including instrumentation options. Usually, this argument will be from self.user_params.

Returns A dictionary of options/parameters for the instrumentation tst.

run_instrumentation_test(*device*, *package*, *options=None*, *prefix=None*, *runner=None*)
Runs instrumentation tests on a device and creates test records.

Parameters

- **device** – AndroidDevice, the device to run instrumentation tests on.
- **package** – string, the package name of the instrumentation tests.
- **options** – dict, Instrumentation options for the instrumentation tests.
- **prefix** – string, an optional prefix for parser output for distinguishing between instrumentation test runs.
- **runner** – string, the runner to use for the instrumentation package, default to DEFAULT_INSTRUMENTATION_RUNNER.

Returns

A boolean indicating whether or not all the instrumentation test methods passed.

Raises *TestError if the instrumentation run crashed or if parsing the* – output failed.

1.5 mobly.base_test module

class mobly.base_test.**BaseTestClass**(*configs*)

Bases: object

Base class for all test classes to inherit from.

This class gets all the controller objects from test_runner and executes the tests requested within itself.

Most attributes of this class are set at runtime based on the configuration provided.

The default logger in logging module is set up for each test run. If you want to log info to the test run output file, use *logging* directly, like *logging.info*.

tests
A list of strings, each representing a test method name.

TAG
A string used to refer to a test class. Default is the test class name.

results
A records.TestResult object for aggregating test results from the execution of tests.

controller_configs
dict, controller configs provided by the user via test bed config.

current_test_info
RuntimeTestInfo, runtime information on the test currently being executed.

root_output_path
string, storage path for output files associated with the entire test run. A test run can have multiple test class executions. This includes the test summary and Mobly log files.

log_path
string, storage path for files specific to a single test class execution.

test_bed_name
[Deprecated, use 'testbed_name' instead] string, the name of the test bed used by a test run.

testbed_name
string, the name of the test bed used by a test run.

user_params
dict, custom parameters from user, to be consumed by the test logic.

TAG = None

exec_one_test(*test_name*, *test_method*, *record=None*)
Executes one test and update test results.

Executes setup_test, the test method, and teardown_test; then creates a records.TestResultRecord object with the execution information and adds the record to the test class's test results.

Parameters

- **test_name** – string, Name of the test.

- **test_method** – function, The test method to execute.
- **record** – records.TestResultRecord, optional arg for injecting a record object to use for this test execution. If not set, a new one is created created. This is meant for passing information between consecutive test case execution for retry purposes. Do NOT abuse this for "magical" features.

Returns TestResultRecord, the test result record object of the test execution. This object is strictly for read-only purposes. Modifying this record will not change what is reported in the test run's summary yaml file.

generate_tests(*test_logic*, *name_func*, *arg_sets*, *uid_func=None*)

Generates tests in the test class.

This function has to be called inside a test class's *self.pre_run* or *self.setup_generated_tests*.

Generated tests are not written down as methods, but as a list of parameter sets. This way we reduce code repetition and improve test scalability.

Users can provide an optional function to specify the UID of each test. Not all generated tests are required to have UID.

Parameters

- **test_logic** – function, the common logic shared by all the generated tests.
- **name_func** – function, generate a test name according to a set of test arguments. This function should take the same arguments as the test logic function.
- **arg_sets** – a list of tuples, each tuple is a set of arguments to be passed to the test logic function and name function.
- **uid_func** – function, an optional function that takes the same arguments as the test logic function and returns a string that is the corresponding UID.

get_existing_test_names()

Gets the names of existing tests in the class.

A method in the class is considered a test if its name starts with 'test_*'.

Note this only gets the names of tests that already exist. If *generate_tests* has not happened when this was called, the generated tests won't be listed.

Returns A list of strings, each is a test method name.

on_fail(*record*)

A function that is executed upon a test failure.

User implementation is optional.

Parameters **record** – records.TestResultRecord, a copy of the test record for this test, containing all information of the test execution including exception objects.

on_pass(*record*)

A function that is executed upon a test passing.

Implementation is optional.

Parameters **record** – records.TestResultRecord, a copy of the test record for this test, containing all information of the test execution including exception objects.

on_skip(*record*)

A function that is executed upon a test being skipped.

Implementation is optional.

Parameters **record** – records.TestResultRecord, a copy of the test record for this test, containing all information of the test execution including exception objects.

pre_run()
Preprocesses that need to be done before setup_class.

This phase is used to do pre-test processes like generating tests. This is the only place *self.generate_tests* should be called.

If this function throws an error, the test class will be marked failure and the "Requested" field will be 0 because the number of tests requested is unknown at this point.

record_data(*content*)
Record an entry in test summary file.

Sometimes additional data need to be recorded in summary file for debugging or post-test analysis.

Each call adds a new entry to the summary file, with no guarantee of its position among the summary file entries.

The content should be a dict. If absent, timestamp field is added for ease of parsing later.

Parameters **content** – dict, the data to add to summary file.

register_controller(*module*, *required=True*, *min_number=1*)
Loads a controller module and returns its loaded devices.

A Mobly controller module is a Python lib that can be used to control a device, service, or equipment. To be Mobly compatible, a controller module needs to have the following members:

```
def create(configs):
  [Required] Creates controller objects from configurations.

  Args:
    configs: A list of serialized data like string/dict. Each
      element of the list is a configuration for a controller
      object.

  Returns:
    A list of objects.

def destroy(objects):
  [Required] Destroys controller objects created by the create
  function. Each controller object shall be properly cleaned up
  and all the resources held should be released, e.g. memory
  allocation, sockets, file handlers etc.

  Args:
    A list of controller objects created by the create function.

def get_info(objects):
  [Optional] Gets info from the controller objects used in a test
  run. The info will be included in test_summary.yaml under
  the key 'ControllerInfo'. Such information could include unique
  ID, version, or anything that could be useful for describing the
  test bed and debugging.

  Args:
    objects: A list of controller objects created by the create
      function.
```

(continues on next page)

(continued from previous page)

```
  Returns:
    A list of json serializable objects: each represents the
      info of a controller object. The order of the info
      object should follow that of the input objects.
```

Registering a controller module declares a test class's dependency the controller. If the module config exists and the module matches the controller interface, controller objects will be instantiated with corresponding configs. The module should be imported first.

Parameters

- **module** – A module that follows the controller module interface.
- **required** – A bool. If True, failing to register the specified controller module raises exceptions. If False, the objects failed to instantiate will be skipped.
- **min_number** – An integer that is the minimum number of controller objects to be created. Default is one, since you should not register a controller module without expecting at least one object.

Returns A list of controller objects instantiated from controller_module, or None if no config existed for this controller and it was not a required controller.

Raises `ControllerError` – * The controller module has already been registered. * The actual number of objects instantiated is less than the * *min_number*. * *required* is True and no corresponding config can be found. * Any other error occurred in the registration process.

run(*test_names=None*)

Runs tests within a test class.

One of these test method lists will be executed, shown here in priority order:

1. The test_names list, which is passed from cmd line. Invalid names are guarded by cmd line arg parsing.
2. The self.tests list defined in test class. Invalid names are ignored.
3. All function that matches test method naming convention in the test class.

Parameters **test_names** – A list of string that are test method names requested in cmd line.

Returns The test results object of this class.

setup_class()

Setup function that will be called before executing any test in the class.

To signal setup failure, use asserts or raise your own exception.

Errors raised from *setup_class* will trigger *on_fail*.

Implementation is optional.

setup_generated_tests()

[DEPRECATED] Use *pre_run* instead.

Preprocesses that need to be done before setup_class.

This phase is used to do pre-test processes like generating tests. This is the only place *self.generate_tests* should be called.

If this function throws an error, the test class will be marked failure and the "Requested" field will be 0 because the number of tests requested is unknown at this point.

setup_test()
Setup function that will be called every time before executing each test method in the test class.

To signal setup failure, use asserts or raise your own exception.

Implementation is optional.

teardown_class()
Teardown function that will be called after all the selected tests in the test class have been executed.

Errors raised from *teardown_class* do not trigger *on_fail*.

Implementation is optional.

teardown_test()
Teardown function that will be called every time a test method has been executed.

Implementation is optional.

unpack_userparams(*req_param_names=None*, *opt_param_names=None*, ***kwargs*)
An optional function that unpacks user defined parameters into individual variables.

After unpacking, the params can be directly accessed with self.xxx.

If a required param is not provided, an exception is raised. If an optional param is not provided, a warning line will be logged.

To provide a param, add it in the config file or pass it in as a kwarg. If a param appears in both the config file and kwarg, the value in the config file is used.

User params from the config file can also be directly accessed in self.user_params.

Parameters

- **req_param_names** – A list of names of the required user params.
- **opt_param_names** – A list of names of the optional user params.
- ****kwargs** – Arguments that provide default values. e.g. unpack_userparams(required_list, opt_list, arg_a='hello') self.arg_a will be 'hello' unless it is specified again in required_list or opt_list.

Raises *Error* – A required user params is not provided.

exception mobly.base_test.**Error**
Bases: Exception

Raised for exceptions that occurred in BaseTestClass.

mobly.base_test.**repeat**(*count*, *max_consecutive_error=None*)
Decorator for repeating a test case multiple times.

The BaseTestClass will execute the test cases annotated with this decorator the specified number of time.

This decorator only stores the information needed for the repeat. It does not execute the repeat.

Parameters

- **count** – int, the total number of times to execute the decorated test case.
- **max_consecutive_error** – int, the maximum number of consecutively failed iterations allowed. If reached, the remaining iterations is abandoned. By default this is not enabled.

Returns The wrapped test function.

Raises ValueError, if the user input is invalid.

mobly.base_test.**retry**(*max_count*)
Decorator for retrying a test case until it passes.

The BaseTestClass will keep executing the test cases annotated with this decorator until the test passes, or the maxinum number of iterations have been met.

This decorator only stores the information needed for the retry. It does not execute the retry.

Parameters **max_count** – int, the maximum number of times to execute the decorated test case.

Returns The wrapped test function.

Raises ValueError, if the user input is invalid.

1.6 mobly.config_parser module

exception mobly.config_parser.**MoblyConfigError**
Bases: Exception

Raised when there is a problem in test configuration file.

class mobly.config_parser.**TestRunConfig**
Bases: object

The data class that holds all the information needed for a test run.

log_path
string, specifies the root directory for all logs written by a test run.

test_bed_name
[Deprecated, use 'testbed_name' instead] string, the name of the test bed used by a test run.

testbed_name
string, the name of the test bed used by a test run.

controller_configs
dict, configs used for instantiating controller objects.

user_params
dict, all the parameters to be consumed by the test logic.

summary_writer
records.TestSummaryWriter, used to write elements to the test result summary file.

test_class_name_suffix
string, suffix to append to the class name for reporting. This is used for differentiating the same class executed with different parameters in a suite.

copy()
Returns a deep copy of the current config.

mobly.config_parser.**load_test_config_file**(*test_config_path*, *tb_filters=None*)
Processes the test configuration file provied by user.

Loads the configuration file into a dict, unpacks each testbed config into its own dict, and validate the configuration in the process.

Parameters

- **test_config_path** – Path to the test configuration file.
- **tb_filters** – A subset of test bed names to be pulled from the config file. If None, then all test beds will be selected.

Returns A list of test configuration dicts to be passed to test_runner.TestRunner.

1.7 mobly.controller_manager module

Module for Mobly controller management.

class mobly.controller_manager.**ControllerManager**(*class_name*, *controller_configs*)
Bases: object

Manages the controller objects for Mobly tests.

This manages the life cycles and info retrieval of all controller objects used in a test.

controller_configs
dict, controller configs provided by the user via test bed config.

get_controller_info_records()
Get the info records for all the controller objects in the manager.

New info records for each controller object are created for every call so the latest info is included.

Returns List of records.ControllerInfoRecord objects. Each opject conatins the info of a type of controller

register_controller(*module*, *required=True*, *min_number=1*)
Loads a controller module and returns its loaded devices.

This is to be used in a mobly test class.

Parameters

- **module** – A module that follows the controller module interface.
- **required** – A bool. If True, failing to register the specified controller module raises exceptions. If False, the objects failed to instantiate will be skipped.
- **min_number** – An integer that is the minimum number of controller objects to be created. Default is one, since you should not register a controller module without expecting at least one object.

Returns A list of controller objects instantiated from controller_module, or None if no config existed for this controller and it was not a required controller.

Raises ControllerError – * The controller module has already been registered. * The actual number of objects instantiated is less than the * *min_number*. * *required* is True and no corresponding config can be found. * Any other error occurred in the registration process.

unregister_controllers()
Destroy controller objects and clear internal registry.

This will be called after each test class.

mobly.controller_manager.**verify_controller_module**(*module*)
Verifies a module object follows the required interface for controllers.

The interface is explained in the docstring of *base_test.BaseTestClass.register_controller*.

Parameters **module** – An object that is a controller module. This is usually imported with import statements or loaded by importlib.

Raises ControllerError – if the module does not match the Mobly controller interface, or one of the required members is null.

1.8 mobly.expects module

mobly.expects.**expect_equal**(*first*, *second*, *msg=None*, *extras=None*)
Expects the equality of objects, otherwise fail the test.

If the expectation is not met, the test is marked as fail after its execution finishes.

Error message is "first != second" by default. Additional explanation can be supplied in the message.

Parameters

- **first** – The first object to compare.
- **second** – The second object to compare.
- **msg** – A string that adds additional info about the failure.
- **extras** – An optional field for extra information to be included in test result.

mobly.expects.**expect_false**(*condition*, *msg*, *extras=None*)
Expects an expression evaluates to False.

If the expectation is not met, the test is marked as fail after its execution finishes.

Parameters

- **expr** – The expression that is evaluated.
- **msg** – A string explaining the details in case of failure.
- **extras** – An optional field for extra information to be included in test result.

mobly.expects.**expect_no_raises**(*message=None*, *extras=None*)
Expects no exception is raised in a context.

If the expectation is not met, the test is marked as fail after its execution finishes.

A default message is added to the exception *details*.

Parameters

- **message** – string, custom message to add to exception's *details*.
- **extras** – An optional field for extra information to be included in test result.

mobly.expects.**expect_true**(*condition*, *msg*, *extras=None*)
Expects an expression evaluates to True.

If the expectation is not met, the test is marked as fail after its execution finishes.

Parameters

- **expr** – The expression that is evaluated.
- **msg** – A string explaining the details in case of failure.
- **extras** – An optional field for extra information to be included in test result.

1.9 mobly.keys module

class mobly.keys.**Config**
Bases: enum.Enum

The reserved keywordss used in configurations.

key_log_path = 'LogPath'

key_mobly_params = 'MoblyParams'

key_testbed = 'TestBeds'

key_testbed_controllers = 'Controllers'

key_testbed_name = 'Name'

key_testbed_test_params = 'TestParams'

1.10 mobly.logger module

class mobly.logger.**PrefixLoggerAdapter**(*logger*, *extra*)

Bases: logging.LoggerAdapter

A wrapper that adds a prefix to each log line.

This logger adapter class is like a decorator to Logger. It takes one Logger-like object and returns a new Logger-like object. The new Logger-like object will print logs with a custom prefix added. Creating new Logger-like objects doesn't modify the behavior of the old Logger-like object.

Chaining multiple logger adapters is also supported. The multiple adapters will take effect in the order in which they are chained, i.e. the log will be '<prefix1> <prefix2> <prefix3> <message>' if we chain 3 PrefixLoggerAdapters.

Example Usage:

```
logger = PrefixLoggerAdapter(logging.getLogger(), {
  'log_prefix': <custom prefix>
})
```

Then each log line added by the logger will have a prefix: '<custom prefix> <message>'.

EXTRA_KEY_LOG_PREFIX = 'log_prefix'

process(*msg: str, kwargs: MutableMapping[str, Any]*) → Tuple[str, MutableMapping[str, Any]]

Processes the logging call to insert contextual information.

Parameters

- **msg** – The logging message.
- **kwargs** – Keyword arguments passed in to a logging call.

Returns The message and kwargs modified.

set_log_prefix(*prefix: str*) → None

Sets the log prefix to the given string.

Parameters **prefix** – The new log prefix.

mobly.logger.**create_latest_log_alias**(*actual_path*, *alias*)

Creates a symlink to the latest test run logs.

Parameters

- **actual_path** – string, the source directory where the latest test run's logs are.
- **alias** – string, the name of the directory to contain the latest log files.

mobly.logger.**epoch_to_log_line_timestamp**(*epoch_time*, *time_zone=None*)

Converts an epoch timestamp in ms to log line timestamp format, which is readible for humans.

Parameters

- **epoch_time** – integer, an epoch timestamp in ms.
- **time_zone** – instance of tzinfo, time zone information. Using pytz rather than python 3.2 time_zone implementation for python 2 compatibility reasons.

Returns A string that is the corresponding timestamp in log line timestamp format.

mobly.logger.**get_log_file_timestamp**(*delta=None*)
Returns a timestamp in the format used for log file names.

Default is current time. If a delta is set, the return value will be the current time offset by delta seconds.

Parameters **delta** – Number of seconds to offset from current time; can be negative.

Returns A timestamp in log filen name format with an offset.

mobly.logger.**get_log_line_timestamp**(*delta=None*)
Returns a timestamp in the format used by log lines.

Default is current time. If a delta is set, the return value will be the current time offset by delta seconds.

Parameters **delta** – Number of seconds to offset from current time; can be negative.

Returns A timestamp in log line format with an offset.

mobly.logger.**is_valid_logline_timestamp**(*timestamp*)

mobly.logger.**kill_test_logger**(*logger*)
Cleans up a test logger object by removing all of its handlers.

Parameters **logger** – The logging object to clean up.

mobly.logger.**logline_timestamp_comparator**(*t1*, *t2*)
Comparator for timestamps in logline format.

Parameters

- **t1** – Timestamp in logline format.
- **t2** – Timestamp in logline format.

Returns -1 if t1 < t2; 1 if t1 > t2; 0 if t1 == t2.

mobly.logger.**normalize_log_line_timestamp**(*log_line_timestamp*)
Replace special characters in log line timestamp with normal characters.

Deprecated since version 1.10: This method is obsolete with the more general *sanitize_filename* method and is only kept for backwards compatibility. In a future update, this method may be removed.

Parameters **log_line_timestamp** – A string in the log line timestamp format. Obtained with get_log_line_timestamp.

Returns A string representing the same time as input timestamp, but without special characters.

mobly.logger.**sanitize_filename**(*filename*)
Sanitizes a filename for various operating systems.

Parameters **filename** – string, the filename to sanitize.

Returns A string that is safe to use as a filename on various operating systems.

mobly.logger.**setup_test_logger**(*log_path*, *prefix=None*, *alias='latest'*, *console_level=20*)
Customizes the root logger for a test run.

In addition to configuring the Mobly logging handlers, this also sets two attributes on the *logging* module for the output directories:

root_output_path: path to the directory for the entire test run. log_path: same as *root_output_path* outside of a test class run. In the

context of a test class run, this is the output directory for files specific to a test class.

Parameters

- **log_path** – string, the location of the report file.
- **prefix** – optional string, a prefix for each log line in terminal.
- **alias** – optional string, The name of the alias to use for the latest log directory. If a falsy value is provided, then the alias directory will not be created, which is useful to save storage space when the storage system (e.g. ZIP files) does not properly support shortcut/symlinks.
- **console_level** – optional logging level, log level threshold used for log messages printed to the console. Logs with a level less severe than console_level will not be printed to the console.

1.11 mobly.records module

This module has classes for test result collection, and test result output.

class mobly.records.**ControllerInfoRecord**(*test_class*, *controller_name*, *info*)
Bases: object

A record representing the controller info in test results.

KEY_CONTROLLER_INFO = 'Controller Info'

KEY_CONTROLLER_NAME = 'Controller Name'

KEY_TEST_CLASS = 'Test Class'

KEY_TIMESTAMP = 'Timestamp'

to_dict()

exception mobly.records.**Error**
Bases: Exception

Raised for errors in record module members.

class mobly.records.**ExceptionRecord**(*e*, *position=None*)
Bases: object

A record representing exception objects in TestResultRecord.

exception
Exception object, the original Exception.

type
string, type name of the exception object.

stacktrace
string, stacktrace of the Exception.

extras
optional serializable, this corresponds to the *TestSignal.extras* field.

position
string, an optional label specifying the position where the Exception ocurred.

to_dict()

class mobly.records.**TestResult**
Bases: object

A class that contains metrics of a test run.

This class is essentially a container of TestResultRecord objects.

requested
A list of strings, each is the name of a test requested by user.

failed
A list of records for tests failed.

executed
A list of records for tests that were actually executed.

passed
A list of records for tests passed.

skipped
A list of records for tests skipped.

error
A list of records for tests with error result token.

controller_info
list of ControllerInfoRecord.

add_class_error(*test_record*)
Add a record to indicate a test class has failed before any test could execute.

This is only called before any test is actually executed. So it only adds an error entry that describes why the class failed to the tally and does not affect the total number of tests requrested or exedcuted.

Parameters **test_record** – A TestResultRecord object for the test class.

add_controller_info_record(*controller_info_record*)
Adds a controller info record to results.

This can be called multiple times for each test class.

Parameters **controller_info_record** – ControllerInfoRecord object to be added to the result.

add_record(*record*)
Adds a test record to test result.

A record is considered executed once it's added to the test result.

Adding the record finalizes the content of a record, so no change should be made to the record afterwards.

Parameters **record** – A test record object to add.

is_all_pass
True if no tests failed or threw errors, False otherwise.

is_test_executed(*test_name*)
Checks if a specific test has been executed.

Parameters **test_name** – string, the name of the test to check.

Returns True if the test has been executed according to the test result, False otherwise.

requested_test_names_dict()
Gets the requested test names of a test run in a dict format.

Note a test can be requested multiple times, so there can be duplicated values

Returns A dict with a key and the list of strings.

summary_dict()
Gets a dictionary that summarizes the stats of this test result.

The summary provides the counts of how many tests fall into each category, like 'Passed', 'Failed' etc.

Returns A dictionary with the stats of this test result.

summary_str()
Gets a string that summarizes the stats of this test result.

The summary provides the counts of how many tests fall into each category, like 'Passed', 'Failed' etc.

Format of the string is: Requested <int>, Executed <int>, ...

Returns A summary string of this test result.

class mobly.records.**TestResultEnums**
Bases: object

Enums used for TestResultRecord class.

Includes the tokens to mark test result with, and the string names for each field in TestResultRecord.

RECORD_BEGIN_TIME = 'Begin Time'

RECORD_CLASS = 'Test Class'

RECORD_DETAILS = 'Details'

RECORD_END_TIME = 'End Time'

RECORD_EXTRAS = 'Extras'

RECORD_EXTRA_ERRORS = 'Extra Errors'

RECORD_NAME = 'Test Name'

RECORD_POSITION = 'Position'

RECORD_RESULT = 'Result'

RECORD_RETRY_PARENT = 'Retry Parent'

RECORD_SIGNATURE = 'Signature'

RECORD_STACKTRACE = 'Stacktrace'

RECORD_TERMINATION_SIGNAL_TYPE = 'Termination Signal Type'

RECORD_UID = 'UID'

TEST_RESULT_ERROR = 'ERROR'

TEST_RESULT_FAIL = 'FAIL'

TEST_RESULT_PASS = 'PASS'

TEST_RESULT_SKIP = 'SKIP'

class mobly.records.**TestResultRecord**(*t_name*, *t_class=None*)

Bases: object

A record that holds the information of a single test.

The record object holds all information of a test, including all the exceptions occurred during the test.

A test can terminate for two reasons:

1. the test function executes to the end and completes naturally.
2. the test is terminated by an exception, which we call

   "termination signal".

The termination signal is treated differently. Its content are extracted into first-tier attributes of the record object, like *details* and *stacktrace*, for easy consumption.

Note the termination signal is not always an error, it can also be explicit pass signal or abort/skip signals.

test_name
string, the name of the test.

begin_time
Epoch timestamp of when the test started.

end_time
Epoch timestamp of when the test ended.

uid
User-defined unique identifier of the test.

signature
string, unique identifier of a test record, the value is generated by Mobly.

retry_parent
TestResultRecord, only set for retry iterations. This is the test result record of the previous retry iteration. Parsers can use this field to construct the chain of execution for each retried test.

termination_signal
ExceptionRecord, the main exception of the test.

extra_errors
OrderedDict, all exceptions occurred during the entire test lifecycle. The order of occurrence is preserved.

result
TestResultEnum.TEST_RESULT_*, PASS/FAIL/SKIP.

add_error(*position*, *e*)
Add extra error happened during a test.

If the test has passed or skipped, this will mark the test result as ERROR.

If an error is added the test record, the record's result is equivalent to the case where an uncaught exception happened.

If the test record has not recorded any error, the newly added error would be the main error of the test record. Otherwise the newly added error is added to the record's extra errors.

Parameters

- **position** – string, where this error occurred, e.g. 'teardown_test'.
- **e** – An exception or a *signals.ExceptionRecord* object.

details
String description of the cause of the test's termination.

Note a passed test can have this as well due to the explicit pass signal. If the test passed implicitly, this field would be None.

extras
User defined extra information of the test result.

Must be serializable.

stacktrace
The stacktrace string for the exception that terminated the test.

termination_signal_type
Type name of the signal that caused the test's termination.

Note a passed test can have this as well due to the explicit pass signal. If the test passed implicitly, this field would be None.

test_begin()
Call this when the test begins execution.

Sets the begin_time of this record.

test_error(*e=None*)
To mark the test as error in this record.

Parameters **e** – An exception object.

test_fail(*e=None*)
To mark the test as failed in this record.

Only test_fail does instance check because we want 'assert xxx' to also fail the test same way assert_true does.

Parameters **e** – An exception object. It can be an instance of AssertionError or mobly.base_test.TestFailure.

test_pass(*e=None*)
To mark the test as passed in this record.

Parameters **e** – An instance of mobly.signals.TestPass.

test_skip(*e=None*)
To mark the test as skipped in this record.

Parameters **e** – An instance of mobly.signals.TestSkip.

to_dict()
Gets a dictionary representating the content of this class.

Returns A dictionary representating the content of this class.

update_record()
Updates the content of a record.

Several display fields like "details" and "stacktrace" need to be updated based on the content of the record object.

As the content of the record change, call this method to update all the appropirate fields.

class mobly.records.**TestSummaryEntryType**
Bases: enum.Enum

Constants used to identify the type of entries in test summary file.

Test summary file contains multiple yaml documents. In order to parse this file efficiently, the write adds the type of each entry when it writes the entry to the file.

The idea is similar to how *TestResult.json_str* categorizes different sections of a *TestResult* object in the serialized format.

CONTROLLER_INFO = 'ControllerInfo'

RECORD = 'Record'

SUMMARY = 'Summary'

TEST_NAME_LIST = 'TestNameList'

USER_DATA = 'UserData'

class mobly.records.**TestSummaryWriter**(*path*)
Bases: object

Writer for the test result summary file of a test run.

For each test run, a writer is created to stream test results to the summary file on disk.

The serialization and writing of the *TestResult* object is intentionally kept out of *TestResult* class and put in this class. Because *TestResult* can be operated on by suites, like + operation, and it is difficult to guarantee the consistency between *TestResult* in memory and the files on disk. Also, this separation makes it easier to provide a more generic way for users to consume the test summary, like via a database instead of a file.

dump(*content*, *entry_type*)
Dumps a dictionary as a yaml document to the summary file.

Each call to this method dumps a separate yaml document to the same summary file associated with a test run.

The content of the dumped dictionary has an extra field *TYPE* that specifies the type of each yaml document, which is the flag for parsers to identify each document.

Parameters

- **content** – dictionary, the content to serialize and write.
- **entry_type** – a member of enum TestSummaryEntryType.

Raises recoreds.Error – An invalid entry type is passed in.

mobly.records.**uid**(*uid*)
Decorator specifying the unique identifier (UID) of a test case.

The UID will be recorded in the test's record when executed by Mobly.

If you use any other decorator for the test method, you may want to use this as the outer-most one.

Note a common UID system is the Universal Unitque Identifier (UUID), but we are not limiting people to use UUID, hence the more generic name *UID*.

Parameters **uid** – string, the uid for the decorated test function.

1.12 mobly.runtime_test_info module

class mobly.runtime_test_info.**RuntimeTestInfo**(*test_name*, *log_path*, *record*)
Bases: object

Container class for runtime information of a test or test stage.

One object corresponds to one test. This is meant to be a read-only class.

This also applies to test stages like *setup_class*, which has its own runtime info but is not part of any single test.

name
string, name of the test.

signature
string, an identifier of the test, a combination of test name and begin time.

record
TestResultRecord, the current test result record. This changes as the test's execution progresses.

output_path
string, path to the test's output directory. It's created upon accessing.

name

output_path

record

signature

1.13 mobly.signals module

This module is where all the test signal classes and related utilities live.

exception mobly.signals.**ControllerError**
Bases: Exception

Raised when an error occurred in controller classes.

exception mobly.signals.**TestAbortAll**(*details*, *extras=None*)
Bases: *mobly.signals.TestAbortSignal*

Raised when all subsequent tests should be aborted.

exception mobly.signals.**TestAbortClass**(*details*, *extras=None*)
Bases: *mobly.signals.TestAbortSignal*

Raised when all subsequent tests within the same test class should be aborted.

exception mobly.signals.**TestAbortSignal**(*details*, *extras=None*)
Bases: *mobly.signals.TestSignal*

Base class for abort signals.

exception mobly.signals.**TestError**(*details*, *extras=None*)
Bases: *mobly.signals.TestSignal*

Raised when a test has an unexpected error.

exception mobly.signals.**TestFailure**(*details*, *extras=None*)
Bases: *mobly.signals.TestSignal*

Raised when a test has failed.

exception mobly.signals.**TestPass**(*details*, *extras=None*)
Bases: *mobly.signals.TestSignal*

Raised when a test has passed.

exception mobly.signals.**TestSignal**(*details*, *extras=None*)
Bases: Exception

Base class for all test result control signals. This is used to signal the result of a test.

details
A string that describes the reason for raising this signal.

extras
A json-serializable data type to convey extra information about a test result.

exception mobly.signals.**TestSignalError**
Bases: Exception

Raised when an error occurs inside a test signal.

exception mobly.signals.**TestSkip**(*details*, *extras=None*)
Bases: *mobly.signals.TestSignal*

Raised when a test has been skipped.

1.14 mobly.suite_runner module

Runner for Mobly test suites.

These is just example code to help users run a collection of Mobly test classes. Users can use it as is or customize it based on their requirements.

There are two ways to use this runner.

1. Call suite_runner.run_suite() with one or more individual test classes. This is for users who just need to execute a collection of test classes without any additional steps.

```
from mobly import suite_runner

from my.test.lib import foo_test
from my.test.lib import bar_test
...
if __name__ == '__main__':
  suite_runner.run_suite(foo_test.FooTest, bar_test.BarTest)
```

2. Create a subclass of base_suite.BaseSuite and add the individual test classes. Using the BaseSuite class allows users to define their own setup and teardown steps on the suite level as well as custom config for each test class.

```
from mobly import base_suite
from mobly import suite_runner

from my.path import MyFooTest
from my.path import MyBarTest


class MySuite(base_suite.BaseSuite):

  def setup_suite(self, config):
    # Add a class with default config.
    self.add_test_class(MyFooTest)
    # Add a class with test selection.
    self.add_test_class(MyBarTest,
                        tests=['test_a', 'test_b'])
```

(continues on next page)

(continued from previous page)

```
    # Add the same class again with a custom config and suffix.
    my_config = some_config_logic(config)
    self.add_test_class(MyBarTest,
                        config=my_config,
                        name_suffix='WithCustomConfig')


if __name__ == '__main__':
  suite_runner.run_suite_class()
```

exception mobly.suite_runner.**Error**
Bases: Exception

mobly.suite_runner.**compute_selected_tests**(*test_classes*, *selected_tests*)
Computes tests to run for each class from selector strings.

This function transforms a list of selector strings (such as FooTest or FooTest.test_method_a) to a dict where keys are test_name classes, and values are lists of selected tests in those classes. None means all tests in that class are selected.

Parameters

- **test_classes** – list of strings, names of all the classes that are part of a suite.
- **selected_tests** – list of strings, list of tests to execute. If empty, all classes *test_classes* are selected. E.g.

```
[
  'FooTest',
  'BarTest',
  'BazTest.test_method_a',
  'BazTest.test_method_b'
]
```

Returns

Identifiers for TestRunner. Keys are test class names; values are lists of test names within class. E.g. the example in *selected_tests* would translate to:

```
{
  FooTest: None,
  BarTest: None,
  BazTest: ['test_method_a', 'test_method_b']
}
```

This dict is easy to consume for *TestRunner*.

Return type dict

mobly.suite_runner.**run_suite**(*test_classes*, *argv=None*)
Executes multiple test classes as a suite.

This is the default entry point for running a test suite script file directly.

Parameters

- **test_classes** – List of python classes containing Mobly tests.
- **argv** – A list that is then parsed as cli args. If None, defaults to cli input.

mobly.suite_runner.**run_suite_class**(*argv=None*)
Executes tests in the test suite.

Parameters **argv** – A list that is then parsed as CLI args. If None, defaults to sys.argv.

1.15 mobly.test_runner module

exception mobly.test_runner.**Error**
Bases: Exception

class mobly.test_runner.**TestRunner**(*log_dir*, *testbed_name*)
Bases: object

The class that instantiates test classes, executes tests, and report results.

One TestRunner instance is associated with one specific output folder and testbed. TestRunner.run() will generate a single set of output files and results for all tests that have been added to this runner.

results
records.TestResult, object used to record the results of a test run.

add_test_class(*config*, *test_class*, *tests=None*, *name_suffix=None*)
Adds tests to the execution plan of this TestRunner.

Parameters

- **config** – config_parser.TestRunConfig, configuration to execute this test class with.
- **test_class** – class, test class to execute.
- **tests** – list of strings, optional list of test names within the class to execute.
- **name_suffix** – string, suffix to append to the class name for reporting. This is used for differentiating the same class executed with different parameters in a suite.

Raises *Error* – if the provided config has a log_path or testbed_name which differs from the arguments provided to this TestRunner's constructor.

mobly_logger(*alias='latest'*, *console_level=20*)
Starts and stops a logging context for a Mobly test run.

Parameters

- **alias** – optional string, the name of the latest log alias directory to create. If a falsy value is specified, then the directory will not be created.
- **console_level** – optional logging level, log level threshold used for log messages printed to the console. Logs with a level less severe than console_level will not be printed to the console.

Yields The host file path where the logs for the test run are stored.

run()
Executes tests.

This will instantiate controller and test classes, execute tests, and print a summary.

This meethod should usually be called within the runner's *mobly_logger* context. If you must use this method outside of the context, you should make sure *self._test_run_metadata.generate_test_run_log_path* is called before each invocation of *run*.

Raises *Error* – if no tests have previously been added to this runner using add_test_class(...).

mobly.test_runner.**main**(*argv=None*)
Execute the test class in a test module.

This is the default entry point for running a test script file directly. In this case, only one test class in a test script is allowed.

To make your test script executable, add the following to your file:

```
from mobly import test_runner
...
if __name__ == '__main__':
  test_runner.main()
```

If you want to implement your own cli entry point, you could use function execute_one_test_class(test_class, test_config, test_identifier)

Parameters **argv** – A list that is then parsed as cli args. If None, defaults to cli input.

mobly.test_runner.**parse_mobly_cli_args**(*argv*)
Parses cli args that are consumed by Mobly.

This is the arg parsing logic for the default test_runner.main entry point.

Multiple arg parsers can be applied to the same set of cli input. So you can use this logic in addition to any other args you want to parse. This function ignores the args that don't apply to default *test_runner.main*.

Parameters **argv** – A list that is then parsed as cli args. If None, defaults to cli input.

Returns Namespace containing the parsed args.

1.16 mobly.utils module

exception mobly.utils.**Error**
Bases: Exception

Raised when an error occurs in a util

mobly.utils.**abs_path**(*path*)
Resolve the '.' and '~' in a path to get the absolute path.

Parameters **path** – The path to expand.

Returns The absolute path of the input path.

mobly.utils.**cli_cmd_to_string**(*args*)
Converts a cmd arg list to string.

Parameters **args** – list of strings, the arguments of a command.

Returns String representation of the command.

mobly.utils.**concurrent_exec**(*func*, *param_list*, *max_workers=30*, *raise_on_exception=False*)
Executes a function with different parameters pseudo-concurrently.

This is basically a map function. Each element (should be an iterable) in the param_list is unpacked and passed into the function. Due to Python's GIL, there's no true concurrency. This is suited for IO-bound tasks.

Parameters

- **func** – The function that performs a task.
- **param_list** – A list of iterables, each being a set of params to be passed into the function.

- **max_workers** – int, the number of workers to use for parallelizing the tasks. By default, this is 30 workers.
- **raise_on_exception** – bool, raises all of the task failures if any of the tasks failed if *True*. By default, this is *False*.

Returns A list of return values from each function execution. If an execution caused an exception, the exception object will be the corresponding result.

Raises `RuntimeError` – If executing any of the tasks failed and *raise_on_exception* is True.

`mobly.utils.`**create_alias**(*target_path*, *alias_path*)
Creates an alias at 'alias_path' pointing to the file 'target_path'.

On Unix, this is implemented via symlink. On Windows, this is done by creating a Windows shortcut file.

Parameters

- **target_path** – Destination path that the alias should point to.
- **alias_path** – Path at which to create the new alias.

`mobly.utils.`**create_dir**(*path*)
Creates a directory if it does not exist already.

Parameters **path** – The path of the directory to create.

`mobly.utils.`**epoch_to_human_time**(*epoch_time*)
Converts an epoch timestamp to human readable time.

This essentially converts an output of get_current_epoch_time to an output of get_current_human_time

Parameters **epoch_time** – An integer representing an epoch timestamp in milliseconds.

Returns A time string representing the input time. None if input param is invalid.

`mobly.utils.`**find_field**(*item_list*, *cond*, *comparator*, *target_field*)
Finds the value of a field in a dict object that satisfies certain conditions.

Parameters

- **item_list** – A list of dict objects.
- **cond** – A param that defines the condition.
- **comparator** – A function that checks if an dict satisfies the condition.
- **target_field** – Name of the field whose value to be returned if an item satisfies the condition.

Returns Target value or None if no item satisfies the condition.

`mobly.utils.`**find_files**(*paths*, *file_predicate*)
Locate files whose names and extensions match the given predicate in the specified directories.

Parameters

- **paths** – A list of directory paths where to find the files.
- **file_predicate** – A function that returns True if the file name and extension are desired.

Returns A list of files that match the predicate.

`mobly.utils.`**find_subclass_in_module**(*base_class*, *module*)
Finds the single subclass of the given base class in the given module.

Parameters

- **base_class** – class, the base class to look for a subclass of in the module.
- **module** – module, the module to look for the single subclass in.

Returns The single subclass of the given base class.

Raises `ValueError` – If the number of subclasses found was not exactly one.

`mobly.utils.`**find_subclasses_in_module**(*base_classes*, *module*)
Finds the subclasses of the given classes in the given module.

Parameters

- **base_classes** – list of classes, the base classes to look for the subclasses of in the module.
- **module** – module, the module to look for the subclasses in.

Returns A list of all of the subclasses found in the module.

`mobly.utils.`**get_available_host_port**()
Gets a host port number available for adb forward.

Returns An integer representing a port number on the host available for adb forward.

Raises *Error* – when no port is found after MAX_PORT_ALLOCATION_RETRY times.

`mobly.utils.`**get_current_epoch_time**()
Current epoch time in milliseconds.

Returns An integer representing the current epoch time in milliseconds.

`mobly.utils.`**get_current_human_time**()
Returns the current time in human readable format.

Returns Min:Sec format.

Return type The current time stamp in Month-Day-Year Hour

`mobly.utils.`**get_settable_properties**(*cls*)
Gets the settable properties of a class.

Only returns the explicitly defined properties with setters.

Parameters **cls** – A class in Python.

`mobly.utils.`**get_timezone_olson_id**()
Return the Olson ID of the local (non-DST) timezone.

Returns A string representing one of the Olson IDs of the local (non-DST) timezone.

`mobly.utils.`**grep**(*regex*, *output*)
Similar to linux's *grep*, this returns the line in an output stream that matches a given regex pattern.

It does not rely on the *grep* binary and is not sensitive to line endings, so it can be used cross-platform.

Parameters

- **regex** – string, a regex that matches the expected pattern.
- **output** – byte string, the raw output of the adb cmd.

Returns A list of strings, all of which are output lines that matches the regex pattern.

`mobly.utils.`**load_file_to_base64_str**(*f_path*)
Loads the content of a file into a base64 string.

Parameters **f_path** – full path to the file including the file name.

Returns A base64 string representing the content of the file in utf-8 encoding.

mobly.utils.**rand_ascii_str**(*length*)

Generates a random string of specified length, composed of ascii letters and digits.

Parameters **length** – The number of characters in the string.

Returns The random string generated.

mobly.utils.**run_command**(*cmd*, *stdout=None*, *stderr=None*, *shell=False*, *timeout=None*, *cwd=None*, *env=None*, *universal_newlines=False*)

Runs a command in a subprocess.

This function is very similar to subprocess.check_output. The main difference is that it returns the return code and std error output as well as supporting a timeout parameter.

Parameters

- **cmd** – string or list of strings, the command to run. See subprocess.Popen() documentation.
- **stdout** – file handle, the file handle to write std out to. If None is given, then subprocess.PIPE is used. See subprocess.Popen() documentation.
- **stderr** – file handle, the file handle to write std err to. If None is given, then subprocess.PIPE is used. See subprocess.Popen() documentation.
- **shell** – bool, True to run this command through the system shell, False to invoke it directly. See subprocess.Popen() docs.
- **timeout** – float, the number of seconds to wait before timing out. If not specified, no timeout takes effect.
- **cwd** – string, the path to change the child's current directory to before it is executed. Note that this directory is not considered when searching the executable, so you can't specify the program's path relative to cwd.
- **env** – dict, a mapping that defines the environment variables for the new process. Default behavior is inheriting the current process' environment.
- **universal_newlines** – bool, True to open file objects in text mode, False in binary mode.

Returns

A 3-tuple of the consisting of the return code, the std output, and the std error.

Raises `subprocess.TimeoutExpired` – The command timed out.

mobly.utils.**start_standing_subprocess**(*cmd*, *shell=False*, *env=None*)

Starts a long-running subprocess.

This is not a blocking call and the subprocess started by it should be explicitly terminated with stop_standing_subprocess.

For short-running commands, you should use subprocess.check_call, which blocks.

Parameters

- **cmd** – string, the command to start the subprocess with.
- **shell** – bool, True to run this command through the system shell, False to invoke it directly. See subprocess.Proc() docs.
- **env** – dict, a custom environment to run the standing subprocess. If not specified, inherits the current environment. See subprocess.Popen() docs.

Returns The subprocess that was started.

mobly.utils.**stop_standing_subprocess**(*proc*)
Stops a subprocess started by start_standing_subprocess.

Before killing the process, we check if the process is running, if it has terminated, Error is raised.

Catches and ignores the PermissionError which only happens on Macs.

Parameters **proc** – Subprocess to terminate.

Raises *Error* – if the subprocess could not be stopped.

mobly.utils.**wait_for_standing_subprocess**(*proc*, *timeout=None*)
Waits for a subprocess started by start_standing_subprocess to finish or times out.

Propagates the exception raised by the subprocess.wait(.) function. The subprocess.TimeoutExpired exception is raised if the process timed-out rather than terminating.

If no exception is raised: the subprocess terminated on its own. No need to call stop_standing_subprocess() to kill it.

If an exception is raised: the subprocess is still alive - it did not terminate. Either call stop_standing_subprocess() to kill it, or call wait_for_standing_subprocess() to keep waiting for it to terminate on its own.

If the corresponding subprocess command generates a large amount of output and this method is called with a timeout value, then the command can hang indefinitely. See http://go/pylib/subprocess.html#subprocess.Popen.wait

This function does not support Python 2.

Parameters

- **p** – Subprocess to wait for.
- **timeout** – An integer number of seconds to wait before timing out.

1.17 Module contents

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

m

mobly, 65
mobly.asserts, 35
mobly.base_instrumentation_test, 40
mobly.base_test, 41
mobly.config_parser, 46
mobly.controller_manager, 47
mobly.controllers, 35
mobly.controllers.android_device, 23
mobly.controllers.android_device_lib, 20
mobly.controllers.android_device_lib.adb, 6
mobly.controllers.android_device_lib.callback_handler, 9
mobly.controllers.android_device_lib.errors, 10
mobly.controllers.android_device_lib.event_dispatcher, 10
mobly.controllers.android_device_lib.fastboot, 13
mobly.controllers.android_device_lib.jsonrpc_client_base, 14
mobly.controllers.android_device_lib.jsonrpc_shell_base, 16
mobly.controllers.android_device_lib.service_manager, 16
mobly.controllers.android_device_lib.services, 6
mobly.controllers.android_device_lib.services.base_service, 1
mobly.controllers.android_device_lib.services.logcat, 3
mobly.controllers.android_device_lib.services.sl4a_service, 4
mobly.controllers.android_device_lib.services.snippet_management_service, 5
mobly.controllers.android_device_lib.sl4a_client, 18
mobly.controllers.android_device_lib.snippet_client, 18
mobly.controllers.android_device_lib.snippet_event, 20
mobly.controllers.attenuator, 31
mobly.controllers.attenuator_lib, 22
mobly.controllers.attenuator_lib.minicircuits, 20
mobly.controllers.attenuator_lib.telnet_scpi_client, 21
mobly.controllers.iperf_server, 32
mobly.controllers.sniffer, 33
mobly.controllers.sniffer_lib, 23
mobly.controllers.sniffer_lib.local, 23
mobly.controllers.sniffer_lib.local.local_base, 22
mobly.controllers.sniffer_lib.local.tcpdump, 22
mobly.controllers.sniffer_lib.local.tshark, 23
mobly.expects, 48
mobly.keys, 48
mobly.logger, 49
mobly.records, 51
mobly.runtime_test_info, 56
mobly.signals, 57
mobly.suite_runner, 58
mobly.test_runner, 60
mobly.utils, 61

Index

A

abort_all() (*in module mobly.asserts*), 35
abort_all_if() (*in module mobly.asserts*), 35
abort_class() (*in module mobly.asserts*), 36
abort_class_if() (*in module mobly.asserts*), 36
abs_path() (*in module mobly.utils*), 61
ActiveCaptureContext (*class in mobly.controllers.sniffer*), 33
adb (*mobly.controllers.android_device.AndroidDevice attribute*), 23
adb_logcat_file_path (*mobly.controllers.android_device.AndroidDevice attribute*), 23, 24
adb_logcat_file_path (*mobly.controllers.android_device_lib.services.logcat.Logcat attribute*), 3
AdbError, 6
AdbProxy (*class in mobly.controllers.android_device_lib.adb*), 6
AdbTimeoutError, 8
add_class_error() (*mobly.records.TestResult method*), 52
add_controller_info_record() (*mobly.records.TestResult method*), 52
add_device_info() (*mobly.controllers.android_device.AndroidDevice method*), 24
add_error() (*mobly.records.TestResultRecord method*), 54
add_record() (*mobly.records.TestResult method*), 52
add_snippet_client() (*mobly.controllers.android_device_lib.services.snippet_management_service.SnippetManagementService method*), 5
add_test_class() (*mobly.test_runner.TestRunner method*), 60
alias (*mobly.controllers.android_device_lib.services.base_service.BaseService attribute*), 1
AndroidDevice (*class in mobly.controllers.android_device*), 23
AndroidDeviceLoggerAdapter (*class in mobly.controllers.android_device*), 28
app_name (*mobly.controllers.android_device_lib.jsonrpc_client_base.JsonRpcClientBase attribute*), 14
args() (*mobly.controllers.android_device_lib.fastboot.FastbootProxy method*), 13
assert_almost_equal() (*in module mobly.asserts*), 36
assert_count_equal() (*in module mobly.asserts*), 36
assert_equal() (*in module mobly.asserts*), 37
assert_false() (*in module mobly.asserts*), 37
assert_greater() (*in module mobly.asserts*), 37
assert_greater_equal() (*in module mobly.asserts*), 37
assert_in() (*in module mobly.asserts*), 37
assert_is() (*in module mobly.asserts*), 37
assert_is_instance() (*in module mobly.asserts*), 37
assert_is_none() (*in module mobly.asserts*), 37
assert_is_not() (*in module mobly.asserts*), 37
assert_is_not_none() (*in module mobly.asserts*), 37
assert_less() (*in module mobly.asserts*), 37
assert_less_equal() (*in module mobly.asserts*), 37
assert_not_almost_equal() (*in module mobly.asserts*), 38
assert_not_equal() (*in module mobly.asserts*), 38
assert_not_in() (*in module mobly.asserts*), 38
assert_not_is_instance() (*in module mobly.asserts*), 38
assert_not_regex() (*in module mobly.asserts*), 38
assert_raises() (*in module mobly.asserts*), 38
assert_raises_regex() (*in module mobly.asserts*), 38
assert_regex() (*in module mobly.asserts*), 39
assert_true() (*in module mobly.asserts*), 39
AttenuatorDevice (*class in mobly.controllers.attenuator_lib.minicircuits*),

20
AttenuatorPath (*class in mobly.controllers.attenuator*), 31
avg_rate (*mobly.controllers.iperf_server.IPerfResult attribute*), 32
avg_receive_rate (*mobly.controllers.iperf_server.IPerfResult attribute*), 32
avg_send_rate (*mobly.controllers.iperf_server.IPerfResult attribute*), 32

B

BaseInstrumentationTestClass (*class in mobly.base_instrumentation_test*), 40
BaseService (*class in mobly.controllers.android_device_lib.services.base_service*), 1
BaseTestClass (*class in mobly.base_test*), 41
begin_time (*mobly.records.TestResultRecord attribute*), 54
BUILD_CHARACTERISTICS (*mobly.controllers.android_device.BuildInfoConstants attribute*), 28
BUILD_FINGERPRINT (*mobly.controllers.android_device.BuildInfoConstants attribute*), 28
BUILD_ID (*mobly.controllers.android_device.BuildInfoConstants attribute*), 28
build_info (*mobly.controllers.android_device.AndroidDevice attribute*), 24
build_info_key (*mobly.controllers.android_device.BuildInfoConstants attribute*), 28
BUILD_PRODUCT (*mobly.controllers.android_device.BuildInfoConstants attribute*), 28
BUILD_TYPE (*mobly.controllers.android_device.BuildInfoConstants attribute*), 28
BUILD_VERSION_CODENAME (*mobly.controllers.android_device.BuildInfoConstants attribute*), 28
BUILD_VERSION_INCREMENTAL (*mobly.controllers.android_device.BuildInfoConstants attribute*), 28
BUILD_VERSION_SDK (*mobly.controllers.android_device.BuildInfoConstants attribute*), 28
BuildInfoConstants (*class in mobly.controllers.android_device*), 28

C

callback_id (*mobly.controllers.android_device_lib.callback_handler.CallbackHandler attribute*), 9
callback_id (*mobly.controllers.android_device_lib.snippet_event.SnippetEvent attribute*), 20
CallbackHandler (*class in mobly.controllers.android_device_lib.callback_handler*), 9
clean_up() (*mobly.controllers.android_device_lib.event_dispatcher.EventDispatcher method*), 10
clear_adb_log() (*mobly.controllers.android_device_lib.services.logcat.Logcat method*), 3
clear_all_events() (*mobly.controllers.android_device_lib.event_dispatcher.EventDispatcher method*), 10
clear_events() (*mobly.controllers.android_device_lib.event_dispatcher.EventDispatcher method*), 10
clear_host_port() (*mobly.controllers.android_device_lib.jsonrpc_client_base.JsonRpcClientBase method*), 14
clear_log (*mobly.controllers.android_device_lib.services.logcat.Config attribute*), 3
cli_cmd_to_string() (*in module mobly.utils*), 61
close() (*mobly.controllers.attenuator_lib.minicircuits.AttenuatorDevice method*), 20
close() (*mobly.controllers.attenuator_lib.telnet_scpi_client.TelnetScpiClient method*), 21
close_socket_connection() (*mobly.controllers.android_device_lib.jsonrpc_client_base.JsonRpcClientBase method*), 14
cmd (*mobly.controllers.android_device_lib.adb.AdbError attribute*), 6
cmd (*mobly.controllers.android_device_lib.adb.AdbTimeoutError attribute*), 8
cmd() (*mobly.controllers.attenuator_lib.telnet_scpi_client.TelnetScpiClient method*), 21
compute_selected_tests() (*in module mobly.suite_runner*), 59
concurrent_exec() (*in module mobly.utils*), 61
Config (*class in mobly.controllers.android_device_lib.services.logcat*), 3
Config (*class in mobly.keys*), 48
CONFIG_KEY_CHANNEL (*mobly.controllers.sniffer.Sniffer attribute*), 33
connect() (*mobly.controllers.android_device_lib.adb.AdbProxy method*), 7
connect() (*mobly.controllers.android_device_lib.jsonrpc_client_base.JsonRpcClientBase method*), 14
CONTINUE (*mobly.controllers.android_device_lib.jsonrpc_client_base.JsonRpcClientBase attribute*), 16
controller_configs (*mobly.base_test.BaseTestClass attribute*), 41
controller_configs (*mobly.config_parser.TestRunConfig attribute*), 46
controller_configs (*mobly.controller_manager.ControllerManager attribute*), 47
controller_info (*mobly.records.TestResult at-*

tribute), 52
CONTROLLER_INFO (*mobly.records.TestSummaryEntryType attribute*), 56
ControllerError, 57
ControllerInfoRecord (*class in mobly.records*), 51
ControllerManager (*class in mobly.controller_manager*), 47
copy() (*mobly.config_parser.TestRunConfig method*), 46
create() (*in module mobly.controllers.android_device*), 28
create() (*in module mobly.controllers.attenuator*), 32
create() (*in module mobly.controllers.iperf_server*), 32
create() (*in module mobly.controllers.sniffer*), 35
create_alias() (*in module mobly.utils*), 62
create_dir() (*in module mobly.utils*), 62
create_latest_log_alias() (*in module mobly.logger*), 49
create_output_excerpts() (*mobly.controllers.android_device_lib.services.base_service.BaseService method*), 1
create_output_excerpts() (*mobly.controllers.android_device_lib.services.logcat_service.Logcat method*), 3
create_output_excerpts_all() (*mobly.controllers.android_device_lib.service_manager.ServiceManager method*), 16
creation_time (*mobly.controllers.android_device_lib.snippet_event.SnippetEvent attribute*), 20
current_test_info (*mobly.base_test.BaseTestClass attribute*), 41
current_user_id (*mobly.controllers.android_device_lib.adb.AdbProxy attribute*), 7

D

data (*mobly.controllers.android_device_lib.snippet_event.SnippetEvent attribute*), 20
debug_tag (*mobly.controllers.android_device.AndroidDevice attribute*), 24
DEBUGGABLE (*mobly.controllers.android_device.BuildInfoConstants attribute*), 28
DEFAULT_INSTRUMENTATION_ERROR_MESSAGE (*mobly.base_instrumentation_test.BaseInstrumentationTestClass attribute*), 40
DEFAULT_INSTRUMENTATION_ERROR_MESSAGE (*mobly.base_instrumentation_test.InstrumentationTestMixin attribute*), 40
DEFAULT_INSTRUMENTATION_OPTION_PREFIX (*mobly.base_instrumentation_test.BaseInstrumentationTestClass attribute*), 40
DEFAULT_INSTRUMENTATION_OPTION_PREFIX (*mobly.base_instrumentation_test.InstrumentationTestMixin attribute*), 40
DEFAULT_TIMEOUT (*mobly.controllers.android_device_lib.event_dispatcher attribute*), 10
destroy() (*in module mobly.controllers.android_device*), 29
destroy() (*in module mobly.controllers.attenuator*), 32
destroy() (*in module mobly.controllers.iperf_server*), 32
destroy() (*in module mobly.controllers.sniffer*), 35
details (*mobly.records.TestResultRecord attribute*), 54
details (*mobly.signals.TestSignal attribute*), 58
device_info (*mobly.controllers.android_device.AndroidDevice attribute*), 24
device_port (*mobly.controllers.android_device_lib.jsonrpc_client_base attribute*), 14
DeviceError, 10
disable_hidden_api_blacklist() (*mobly.controllers.android_device_lib.jsonrpc_client_base.JsonRpcClientBase method*), 15
disable_hidden_api_blacklist() (*mobly.controllers.android_device_lib.jsonrpc_client_base.JsonRpcClientBase method*), 15
dump() (*mobly.records.TestSummaryWriter method*), 56
DuplicateError, 10

E

emulator_port (*mobly.controllers.android_device.AndroidDevice attribute*), 24
end_time (*mobly.records.TestResultRecord attribute*), 54
epoch_to_human_time() (*in module mobly.utils*), 62
epoch_to_log_line_timestamp() (*in module mobly.logger*), 49
Error, 3, 5, 8, 10, 16, 32, 45, 51, 59–61
error (*mobly.controllers.iperf_server.IPerfResult attribute*), 32
error (*mobly.records.TestResult attribute*), 52
EventDispatcher (*class in mobly.controllers.android_device_lib.event_dispatcher*), 10
EventDispatcherError, 13
exception (*mobly.records.ExceptionRecord attribute*), 51
ExceptionRecord (*class in mobly.records*), 51
exe_cmd() (*in module mobly.controllers.android_device_lib.fastboot*), 13
exec_one_test() (*mobly.base_test.BaseTestClass method*), 41
executed (*mobly.records.TestResult attribute*), 52
ExecutionError, 33
expect_equal() (*in module mobly.expects*), 48

expect_false() (*in module mobly.expects*), 48
expect_no_raises() (*in module mobly.expects*), 48
expect_true() (*in module mobly.expects*), 48
explicit_pass() (*in module mobly.asserts*), 39
extra_errors (*mobly.records.TestResultRecord attribute*), 54
EXTRA_KEY_LOG_PREFIX (*mobly.logger.PrefixLoggerAdapter attribute*), 49
extras (*mobly.records.ExceptionRecord attribute*), 51
extras (*mobly.records.TestResultRecord attribute*), 55
extras (*mobly.signals.TestSignal attribute*), 58

F

fail() (*in module mobly.asserts*), 39
failed (*mobly.records.TestResult attribute*), 52
fastboot (*mobly.controllers.android_device.AndroidDevice attribute*), 23
FastbootProxy (*class in mobly.controllers.android_device_lib.fastboot*), 13
filter_devices() (*in module mobly.controllers.android_device*), 29
find_field() (*in module mobly.utils*), 62
find_files() (*in module mobly.utils*), 62
find_subclass_in_module() (*in module mobly.utils*), 62
find_subclasses_in_module() (*in module mobly.utils*), 63
for_each() (*mobly.controllers.android_device_lib.service_manager.ServiceManager method*), 17
forward() (*mobly.controllers.android_device_lib.adb.AdbProxy method*), 7
from_dict() (*in module mobly.controllers.android_device_lib.snippet_event*), 20

G

generate_filename() (*mobly.controllers.android_device.AndroidDevice method*), 24
generate_tests() (*mobly.base_test.BaseTestClass method*), 42
get_all_instances() (*in module mobly.controllers.android_device*), 29
get_atten() (*mobly.controllers.attenuator.AttenuatorPath method*), 31
get_atten() (*mobly.controllers.attenuator_lib.minicircuits.AttenuatorDevice method*), 20
get_available_host_port() (*in module mobly.utils*), 63
get_capture_file() (*mobly.controllers.sniffer.Sniffer method*), 33
get_capture_file() (*mobly.controllers.sniffer_lib.local.local_base.SnifferLocalBase method*), 22
get_controller_info_records() (*mobly.controller_manager.ControllerManager method*), 47
get_current_epoch_time() (*in module mobly.utils*), 63
get_current_human_time() (*in module mobly.utils*), 63
get_descriptor() (*mobly.controllers.sniffer.Sniffer method*), 33
get_descriptor() (*mobly.controllers.sniffer_lib.local.tcpdump.Sniffer method*), 22
get_descriptor() (*mobly.controllers.sniffer_lib.local.tshark.Sniffer method*), 23
get_device() (*in module mobly.controllers.android_device*), 29
get_devices() (*in module mobly.controllers.android_device*), 29
get_event_q() (*mobly.controllers.android_device_lib.event_dispatcher method*), 11
get_existing_test_names() (*mobly.base_test.BaseTestClass method*), 42
get_info() (*in module mobly.controllers.android_device*), 30
get_instances() (*in module mobly.controllers.android_device*), 30
get_instances_with_configs() (*in module mobly.controllers.android_device*), 30
get_interface() (*mobly.controllers.sniffer.Sniffer method*), 33
get_interface() (*mobly.controllers.sniffer_lib.local.local_base.Sniffer method*), 22
get_json() (*mobly.controllers.iperf_server.IPerfResult method*), 32
get_log_file_timestamp() (*in module mobly.logger*), 50
get_log_line_timestamp() (*in module mobly.logger*), 50
get_max_atten() (*mobly.controllers.attenuator.AttenuatorPath method*), 32
get_settable_properties() (*in module mobly.utils*), 63
get_snippet_client() (*mobly.controllers.android_device_lib.services.snippet_management_service.SnippetManagementService method*), 5
get_subtype() (*mobly.controllers.sniffer.Sniffer method*), 33
get_subtype() (*mobly.controllers.sniffer_lib.local.tcpdump.Sniffer method*), 23
get_subtype() (*mobly.controllers.sniffer_lib.local.tshark.Sniffer method*), 23

get_timezone_olson_id() (*in module mobly.utils*), 63
get_type() (*mobly.controllers.sniffer.Sniffer method*), 34
get_type() (*mobly.controllers.sniffer_lib.local.local_base.SnifferLocalBase method*), 22
getAll() (*mobly.controllers.android_device_lib.callback_handler.CallbackHandler method*), 9
getprop() (*mobly.controllers.android_device_lib.adb.AdbProxy method*), 7
getprops() (*mobly.controllers.android_device_lib.adb.AdbProxy method*), 7
grep() (*in module mobly.utils*), 63

H

handle_event() (*mobly.controllers.android_device_lib.event_dispatcher.EventDispatcher method*), 11
handle_reboot() (*mobly.controllers.android_device.AndroidDevice method*), 24
handle_subscribed_event() (*mobly.controllers.android_device_lib.event_dispatcher.EventDispatcher method*), 11
handle_usb_disconnect() (*mobly.controllers.android_device.AndroidDevice method*), 25
HARDWARE (*mobly.controllers.android_device.BuildInfoConstants attribute*), 28
has_active_service (*mobly.controllers.android_device.AndroidDevice attribute*), 25
has_service_by_name() (*mobly.controllers.android_device_lib.service_manager.ServiceManager method*), 17
has_shell_command() (*mobly.controllers.android_device_lib.adb.AdbProxy method*), 7
help() (*mobly.controllers.android_device_lib.snippet_client.SnippetClient method*), 19
host_port (*mobly.controllers.android_device_lib.jsonrpc_client_base.JsonRpcClientBase attribute*), 14

I

IllegalStateError, 13
INIT (*mobly.controllers.android_device_lib.jsonrpc_client_base.JsonRpcCommand attribute*), 16
instrument() (*mobly.controllers.android_device_lib.adb.AdbProxy method*), 7
InstrumentationTestMixin (*class in mobly.base_instrumentation_test*), 40
InvalidDataError, 33
InvalidOperationError, 33
IPerfResult (*class in mobly.controllers.iperf_server*), 32
IPerfServer (*class in mobly.controllers.iperf_server*), 32
is_adb_available() (*in module mobly.controllers.android_device_lib.adb*), 8
is_adb_detectable() (*mobly.controllers.android_device.AndroidDevice method*), 25
is_adb_root (*mobly.controllers.android_device.AndroidDevice attribute*), 25
is_alive (*mobly.controllers.android_device_lib.services.base_service.BaseService attribute*), 2
is_alive (*mobly.controllers.android_device_lib.services.logcat.Logcat attribute*), 3
is_alive (*mobly.controllers.android_device_lib.services.sl4a_service.Sl4aService attribute*), 4
is_alive (*mobly.controllers.android_device_lib.services.snippet_management_service.SnippetManagementService attribute*), 5
is_alive (*mobly.controllers.android_device_lib.snippet_client.SnippetClient attribute*), 19
is_all_pass (*mobly.records.TestResult attribute*), 52
is_any_alive (*mobly.controllers.android_device_lib.service_manager.ServiceManager attribute*), 17
is_boot_completed() (*mobly.controllers.android_device.AndroidDevice method*), 25
is_bootloader (*mobly.controllers.android_device.AndroidDevice attribute*), 25
is_emulator (*mobly.controllers.android_device.AndroidDevice attribute*), 25
is_open (*mobly.controllers.attenuator_lib.minicircuits.AttenuatorDevice attribute*), 21
is_open (*mobly.controllers.attenuator_lib.telnet_scpi_client.TelnetScpiClient attribute*), 21
is_rootable (*mobly.controllers.android_device.AndroidDevice attribute*), 26
is_test_executed() (*mobly.records.TestResult method*), 52
is_valid_logline_timestamp() (*in module mobly.logger*), 50

J

JsonRpcClientBase (*class in mobly.controllers.android_device_lib.jsonrpc_client_base*), 14
JsonRpcCommand (*class in mobly.controllers.android_device_lib.jsonrpc_client_base*), 16
JsonRpcShellBase (*class in mobly.controllers.android_device_lib.jsonrpc_shell_base*), 16

K

KEY_CONTROLLER_INFO

(*mobly.records.ControllerInfoRecord attribute*), 51
KEY_CONTROLLER_NAME (*mobly.records.ControllerInfoRecord attribute*), 51
key_log_path (*mobly.keys.Config attribute*), 48
key_mobly_params (*mobly.keys.Config attribute*), 49
KEY_TEST_CLASS (*mobly.records.ControllerInfoRecord attribute*), 51
key_testbed (*mobly.keys.Config attribute*), 49
key_testbed_controllers (*mobly.keys.Config attribute*), 49
key_testbed_name (*mobly.keys.Config attribute*), 49
key_testbed_test_params (*mobly.keys.Config attribute*), 49
KEY_TIMESTAMP (*mobly.records.ControllerInfoRecord attribute*), 51
kill_test_logger() (*in module mobly.logger*), 50

L

list_adb_devices() (*in module mobly.controllers.android_device*), 30
list_adb_devices_by_usb_id() (*in module mobly.controllers.android_device*), 30
list_fastboot_devices() (*in module mobly.controllers.android_device*), 30
list_live_services() (*mobly.controllers.android_device_lib.service_manager.ServiceManager method*), 17
list_occupied_adb_ports() (*in module mobly.controllers.android_device_lib.adb*), 8
load_config() (*mobly.controllers.android_device.AndroidDevice method*), 26
load_device() (*mobly.controllers.android_device_lib.jsonrpc_shell_base.JsonRpcShellBase method*), 16
load_file_to_base64_str() (*in module mobly.utils*), 63
load_snippet() (*mobly.controllers.android_device.AndroidDevice method*), 26
load_test_config_file() (*in module mobly.config_parser*), 46
log (*mobly.controllers.android_device.AndroidDevice attribute*), 23
log_path (*mobly.base_test.BaseTestClass attribute*), 41
log_path (*mobly.config_parser.TestRunConfig attribute*), 46
log_path (*mobly.controllers.android_device.AndroidDevice attribute*), 23, 26
Logcat (*class in mobly.controllers.android_device_lib.services.logcat*), 3
logcat_params (*mobly.controllers.android_device_lib.services.logcat.Config attribute*), 3
logline_timestamp_comparator() (*in module mobly.logger*), 50

M

main() (*in module mobly.test_runner*), 60
main() (*mobly.controllers.android_device_lib.jsonrpc_shell_base.JsonRpcShellBase method*), 16
mobly (*module*), 65
mobly.asserts (*module*), 35
mobly.base_instrumentation_test (*module*), 40
mobly.base_test (*module*), 41
mobly.config_parser (*module*), 46
mobly.controller_manager (*module*), 47
mobly.controllers (*module*), 35
mobly.controllers.android_device (*module*), 23
mobly.controllers.android_device_lib (*module*), 20
mobly.controllers.android_device_lib.adb (*module*), 6
mobly.controllers.android_device_lib.callback_handl (*module*), 9
mobly.controllers.android_device_lib.errors (*module*), 10
mobly.controllers.android_device_lib.event_dispatch (*module*), 10
mobly.controllers.android_device_lib.fastboot (*module*), 13
mobly.controllers.android_device_lib.jsonrpc_client (*module*), 14
mobly.controllers.android_device_lib.jsonrpc_shell_ (*module*), 16
mobly.controllers.android_device_lib.service_manage (*module*), 16
mobly.controllers.android_device_lib.services (*module*), 6
mobly.controllers.android_device_lib.services.base_ (*module*), 1
mobly.controllers.android_device_lib.services.logca (*module*), 3
mobly.controllers.android_device_lib.services.sl4a_ (*module*), 4
mobly.controllers.android_device_lib.services.snipp (*module*), 5
mobly.controllers.android_device_lib.sl4a_client (*module*), 18
mobly.controllers.android_device_lib.snippet_client (*module*), 18
mobly.controllers.android_device_lib.snippet_event (*module*), 20
mobly.controllers.attenuator (*module*), 31
mobly.controllers.attenuator_lib (*module*), 22

mobly.controllers.attenuator_lib.minicircuits (*module*), 20
mobly.controllers.attenuator_lib.telnet_scpi_client (*module*), 21
mobly.controllers.iperf_server (*module*), 32
mobly.controllers.sniffer (*module*), 33
mobly.controllers.sniffer_lib (*module*), 23
mobly.controllers.sniffer_lib.local (*module*), 23
mobly.controllers.sniffer_lib.local.local_base (*module*), 22
mobly.controllers.sniffer_lib.local.tcpdump (*module*), 22
mobly.controllers.sniffer_lib.local.tshark (*module*), 23
mobly.expects (*module*), 48
mobly.keys (*module*), 48
mobly.logger (*module*), 49
mobly.records (*module*), 51
mobly.runtime_test_info (*module*), 56
mobly.signals (*module*), 57
mobly.suite_runner (*module*), 58
mobly.test_runner (*module*), 60
mobly.utils (*module*), 61
mobly_logger() (*mobly.test_runner.TestRunner method*), 60
MoblyConfigError, 46
model (*mobly.controllers.android_device.AndroidDevice attribute*), 26

N

name (*mobly.controllers.android_device_lib.snippet_event.SnippetEvent attribute*), 20
name (*mobly.runtime_test_info.RuntimeTestInfo attribute*), 57
normalize_log_line_timestamp() (*in module mobly.logger*), 50

O

on_fail() (*mobly.base_test.BaseTestClass method*), 42
on_pass() (*mobly.base_test.BaseTestClass method*), 42
on_skip() (*mobly.base_test.BaseTestClass method*), 42
open() (*mobly.controllers.attenuator_lib.minicircuits.AttenuatorDevice method*), 21
open() (*mobly.controllers.attenuator_lib.telnet_scpi_client.TelnetScpiClient method*), 21
output_file_path (*mobly.controllers.android_device_lib.services.logcat.Config attribute*), 3
OUTPUT_FILE_TYPE (*mobly.controllers.android_device_lib.services.logcat.Logcat attribute*), 3
output_path (*mobly.runtime_test_info.RuntimeTestInfo attribute*), 57

P

parse_device_list() (*in module mobly.controllers.android_device*), 30
parse_instrumentation_options() (*mobly.base_instrumentation_test.InstrumentationTestMixin method*), 40
parse_mobly_cli_args() (*in module mobly.test_runner*), 61
passed (*mobly.records.TestResult attribute*), 52
path_count (*mobly.controllers.attenuator_lib.minicircuits.AttenuatorDev attribute*), 20
pause() (*mobly.controllers.android_device_lib.services.base_service.Bas method*), 2
pause() (*mobly.controllers.android_device_lib.services.logcat.Logcat method*), 3
pause() (*mobly.controllers.android_device_lib.services.sl4a_service.Sl4a method*), 4
pause() (*mobly.controllers.android_device_lib.services.snippet_manager method*), 5
pause_all() (*mobly.controllers.android_device_lib.service_manager.Se method*), 17
poll_events() (*mobly.controllers.android_device_lib.event_dispatcher method*), 11
pop_all() (*mobly.controllers.android_device_lib.event_dispatcher.Event method*), 11
pop_event() (*mobly.controllers.android_device_lib.event_dispatcher.Ev method*), 12
pop_events() (*mobly.controllers.android_device_lib.event_dispatcher.E method*), 12
position (*mobly.records.ExceptionRecord attribute*), 51
pre_run() (*mobly.base_test.BaseTestClass method*), 43
PrefixLoggerAdapter (*class in mobly.logger*), 49
process() (*mobly.controllers.android_device.AndroidDeviceLoggerAdap method*), 28
process() (*mobly.logger.PrefixLoggerAdapter method*), 49
PRODUCT_NAME (*mobly.controllers.android_device.BuildInfoConstants attribute*), 28

R

rand_ascii_str() (*in module mobly.utils*), 64
reboot() (*mobly.controllers.android_device.AndroidDevice method*), 26
RECORD (*mobly.records.TestSummaryEntryType attribute*), 56
record (*mobly.runtime_test_info.RuntimeTestInfo attribute*), 57
RECORD_BEGIN_TIME (*mobly.records.TestResultEnums attribute*),

53
RECORD_CLASS (*mobly.records.TestResultEnums attribute*), 53
record_data() (*mobly.base_test.BaseTestClass method*), 43
RECORD_DETAILS (*mobly.records.TestResultEnums attribute*), 53
RECORD_END_TIME (*mobly.records.TestResultEnums attribute*), 53
RECORD_EXTRA_ERRORS (*mobly.records.TestResultEnums attribute*), 53
RECORD_EXTRAS (*mobly.records.TestResultEnums attribute*), 53
RECORD_NAME (*mobly.records.TestResultEnums attribute*), 53
RECORD_POSITION (*mobly.records.TestResultEnums attribute*), 53
RECORD_RESULT (*mobly.records.TestResultEnums attribute*), 53
RECORD_RETRY_PARENT (*mobly.records.TestResultEnums attribute*), 53
RECORD_SIGNATURE (*mobly.records.TestResultEnums attribute*), 53
RECORD_STACKTRACE (*mobly.records.TestResultEnums attribute*), 53
RECORD_TERMINATION_SIGNAL_TYPE (*mobly.records.TestResultEnums attribute*), 53
RECORD_UID (*mobly.records.TestResultEnums attribute*), 53
register() (*mobly.controllers.android_device_lib.service_manager.ServiceManager method*), 17
register_controller() (*mobly.base_test.BaseTestClass method*), 43
register_controller() (*mobly.controller_manager.ControllerManager method*), 47
register_handler() (*mobly.controllers.android_device_lib.event_dispatcher.EventDispatcher method*), 12
remove_snippet_client() (*mobly.controllers.android_device_lib.services.snippet_management_service.SnippetManagementService method*), 5
repeat() (*in module mobly.base_test*), 45
requested (*mobly.records.TestResult attribute*), 52
requested_test_names_dict() (*mobly.records.TestResult method*), 52
restore_app_connection() (*mobly.controllers.android_device_lib.jsonrpc_client_base.JsonRpcClientBase method*), 15
restore_app_connection() (*mobly.controllers.android_device_lib.sl4a_client.Sl4aClient method*), 18
restore_app_connection() (*mobly.controllers.android_device_lib.snippet_client.SnippetClient method*), 19
result (*mobly.records.TestResultRecord attribute*), 54
results (*mobly.base_test.BaseTestClass attribute*), 41
results (*mobly.test_runner.TestRunner attribute*), 60
resume() (*mobly.controllers.android_device_lib.services.base_service.BaseService method*), 2
resume() (*mobly.controllers.android_device_lib.services.logcat.Logcat method*), 3
resume() (*mobly.controllers.android_device_lib.services.sl4a_service.Sl4aService method*), 4
resume() (*mobly.controllers.android_device_lib.services.snippet_management_service.SnippetManagementService method*), 6
resume_all() (*mobly.controllers.android_device_lib.service_manager.ServiceManager method*), 17
resume_services() (*mobly.controllers.android_device_lib.service_manager.ServiceManager method*), 17
ret_code (*mobly.controllers.android_device_lib.adb.AdbError attribute*), 6
ret_value (*mobly.controllers.android_device_lib.callback_handler.CallbackHandler attribute*), 9
retry() (*in module mobly.base_test*), 45
retry_parent (*mobly.records.TestResultRecord attribute*), 54
root() (*mobly.controllers.android_device_lib.adb.AdbProxy method*), 8
root_adb() (*mobly.controllers.android_device.AndroidDevice method*), 26
run() (*mobly.base_test.BaseTestClass attribute*), 41
run() (*mobly.base_test.BaseTestClass method*), 44
run() (*mobly.test_runner.TestRunner method*), 60
run_command() (*in module mobly.utils*), 64
run_instrumentation_test() (*mobly.base_instrumentation_test.InstrumentationTestMixin method*), 40
run_iperf_client() (*mobly.controllers.android_device.AndroidDevice method*), 26
run_suite() (*in module mobly.suite_runner*), 59
RuntimeTestInfo (*class in mobly.runtime_test_info*), 56

S

sanitize_filename() (*in module mobly.logger*), 50

serial (*mobly.controllers.android_device.AndroidDevice attribute*), 23, 27
serial (*mobly.controllers.android_device_lib.adb.AdbError attribute*), 6
serial (*mobly.controllers.android_device_lib.adb.AdbTimeoutError attribute*), 8
SERVICE_TYPE (*mobly.controllers.android_device_lib.errors.ServiceError attribute*), 10
SERVICE_TYPE (*mobly.controllers.android_device_lib.services.logcat attribute*), 3
SERVICE_TYPE (*mobly.controllers.android_device_lib.services.snippet_management_service attribute*), 5
ServiceError, 10
ServiceManager (*class in mobly.controllers.android_device_lib.service_manager*), 16
services (*mobly.controllers.android_device.AndroidDevice attribute*), 23
set_atten() (*mobly.controllers.attenuator.AttenuatorPath method*), 32
set_atten() (*mobly.controllers.attenuator_lib.minicircuits.AttenuatorDevice method*), 21
set_log_prefix() (*mobly.logger.PrefixLoggerAdapter method*), 49
set_snippet_client_verbose_logging() (*mobly.controllers.android_device_lib.jsonrpc_client_base.JsonRpcClientBase method*), 15
setup_class() (*mobly.base_test.BaseTestClass method*), 44
setup_generated_tests() (*mobly.base_test.BaseTestClass method*), 44
setup_test() (*mobly.base_test.BaseTestClass method*), 44
setup_test_logger() (*in module mobly.logger*), 50
signature (*mobly.records.TestResultRecord attribute*), 54
signature (*mobly.runtime_test_info.RuntimeTestInfo attribute*), 57
skip() (*in module mobly.asserts*), 39
skip_if() (*in module mobly.asserts*), 39
skipped (*mobly.records.TestResult attribute*), 52
sl4a (*mobly.controllers.android_device.AndroidDevice attribute*), 27
Sl4aClient (*class in mobly.controllers.android_device_lib.sl4a_client*), 18
Sl4aService (*class in mobly.controllers.android_device_lib.services.sl4a_service*), 4
Sniffer (*class in mobly.controllers.sniffer*), 33
Sniffer (*class in mobly.controllers.sniffer_lib.local.tcpdump*), 22
Sniffer (*class in mobly.controllers.sniffer_lib.local.tshark*), 23
SnifferError, 35
SnifferLocalBase (*class in mobly.controllers.sniffer_lib.local.local_base*), 22
SnippetClient (*class in mobly.controllers.android_device_lib.snippet_client*), 18
SnippetError (*mobly.controllers.android_device_lib.snippet_client.SnippetClient attribute*)
SnippetEvent (*class in mobly.controllers.android_device_lib.snippet_event*), 20
SnippetManagementService (*class in mobly.controllers.android_device_lib.services.snippet_manageme*
stacktrace (*mobly.records.ExceptionRecord attribute*), 51
stacktrace (*mobly.records.TestResultRecord attribute*), 55
start() (*mobly.controllers.android_device_lib.event_dispatcher.EventDi method*), 12
start() (*mobly.controllers.android_device_lib.services.base_service.Bas method*), 2
start() (*mobly.controllers.android_device_lib.services.logcat.Logcat method*), 4
start() (*mobly.controllers.android_device_lib.services.sl4a_service.Sl4a method*), 5
start() (*mobly.controllers.android_device_lib.services.snippet_manager method*), 6
start() (*mobly.controllers.iperf_server.IPerfServer method*), 32
start_all() (*mobly.controllers.android_device_lib.service_manager.Se method*), 17
start_app_and_connect() (*mobly.controllers.android_device_lib.jsonrpc_client_base.JsonR method*), 15
start_app_and_connect() (*mobly.controllers.android_device_lib.sl4a_client.Sl4aClient method*), 18
start_app_and_connect() (*mobly.controllers.android_device_lib.snippet_client.SnippetClien method*), 19
start_capture() (*mobly.controllers.sniffer.Sniffer method*), 34
start_capture() (*mobly.controllers.sniffer_lib.local.local_base.Sniffe method*), 22
start_console() (*mobly.controllers.android_device_lib.jsonrpc_shell_ method*), 16
start_services() (*mobly.controllers.android_device_lib.service_man method*), 17
start_standing_subprocess() (*in module mobly.utils*), 64
stderr (*mobly.controllers.android_device_lib.adb.AdbError attribute*), 6

stdout (*mobly.controllers.android_device_lib.adb.AdbError attribute*), 6
stop() (*mobly.controllers.android_device_lib.services.base_service.BaseService method*), 2
stop() (*mobly.controllers.android_device_lib.services.logcat.Logcat method*), 4
stop() (*mobly.controllers.android_device_lib.services.sl4a_service.Sl4aService method*), 5
stop() (*mobly.controllers.android_device_lib.services.snippet_management_service.SnippetManagementService method*), 6
stop() (*mobly.controllers.iperf_server.IPerfServer method*), 32
stop_all() (*mobly.controllers.android_device_lib.service_manager.ServiceManager method*), 18
stop_app() (*mobly.controllers.android_device_lib.jsonrpc_client_base.JsonRpcClientBase method*), 15
stop_app() (*mobly.controllers.android_device_lib.sl4a_client.Sl4aClient method*), 18
stop_app() (*mobly.controllers.android_device_lib.snippet_client.SnippetClient method*), 19
stop_capture() (*mobly.controllers.sniffer.Sniffer method*), 34
stop_capture() (*mobly.controllers.sniffer_lib.local.local_base_sniffer.LocalBase method*), 22
stop_event_dispatcher() (*mobly.controllers.android_device_lib.sl4a_client.Sl4aClient method*), 18
stop_standing_subprocess() (*in module mobly.utils*), 65
SUMMARY (*mobly.records.TestSummaryEntryType attribute*), 56
summary_dict() (*mobly.records.TestResult method*), 53
summary_str() (*mobly.records.TestResult method*), 53
summary_writer (*mobly.config_parser.TestRunConfig attribute*), 46
system_prop_key (*mobly.controllers.android_device.BuildInfoConstants attribute*), 28

T

TAG (*mobly.base_test.BaseTestClass attribute*), 41
take_bug_report() (*mobly.controllers.android_device.AndroidDevice method*), 27
take_bug_reports() (*in module mobly.controllers.android_device*), 30
take_screenshot() (*mobly.controllers.android_device.AndroidDevice method*), 27
teardown_class() (*mobly.base_test.BaseTestClass method*), 45
teardown_test() (*mobly.base_test.BaseTestClass method*), 45
TelnetScpiClient (*class in mobly.controllers.attenuator_lib.telnet_scpi_client*), 33
termination_signal (*mobly.records.TestResultRecord attribute*), 54
termination_signal_type (*mobly.records.TestResultRecord attribute*), 55
test_bed_name (*mobly.base_test.BaseTestClass attribute*), 41
test_bed_name (*mobly.config_parser.TestRunConfig attribute*), 46
test_begin() (*mobly.records.TestResultRecord method*), 55
test_class_name_suffix (*mobly.config_parser.TestRunConfig attribute*), 46
test_end() (*mobly.records.TestResultRecord method*), 55
test_fail() (*mobly.records.TestResultRecord method*), 55
test_name (*mobly.records.TestResultRecord attribute*), 54
TEST_NAME_LIST (*mobly.records.TestSummaryEntryType attribute*), 56
test_pass() (*mobly.records.TestResultRecord method*), 55
TEST_RESULT_ERROR (*mobly.records.TestResultEnums attribute*), 53
TEST_RESULT_FAIL (*mobly.records.TestResultEnums attribute*), 53
TEST_RESULT_PASS (*mobly.records.TestResultEnums attribute*), 53
TEST_RESULT_SKIP (*mobly.records.TestResultEnums attribute*), 53
test_skip() (*mobly.records.TestResultRecord method*), 55
TestAbortAll, 57
TestAbortClass, 57
TestAbortSignal, 57
testbed_name (*mobly.base_test.BaseTestClass attribute*), 41
testbed_name (*mobly.config_parser.TestRunConfig attribute*), 46
TestError, 57
TestFailure, 57
TestPass, 57
TestResult (*class in mobly.records*), 52
TestResultEnums (*class in mobly.records*), 53
TestResultRecord (*class in mobly.records*), 53
TestRunConfig (*class in mobly.config_parser*), 46
TestRunner (*class in mobly.test_runner*), 60

tests (*mobly.base_test.BaseTestClass attribute*), 41
TestSignal, 57
TestSignalError, 58
TestSkip, 58
TestSummaryEntryType (*class in mobly.records*), 55
TestSummaryWriter (*class in mobly.records*), 56
timeout (*mobly.controllers.android_device_lib.adb.AdbTimeoutError attribute*), 8
to_dict() (*mobly.records.ControllerInfoRecord method*), 51
to_dict() (*mobly.records.ExceptionRecord method*), 51
to_dict() (*mobly.records.TestResultRecord method*), 55
type (*mobly.records.ExceptionRecord attribute*), 51

U

uid (*mobly.controllers.android_device_lib.jsonrpc_client_base.JsonRpcClientBase attribute*), 14
uid (*mobly.records.TestResultRecord attribute*), 54
uid() (*in module mobly.records*), 56
unload_snippet() (*mobly.controllers.android_device.AndroidDevice method*), 27
unpack_userparams() (*mobly.base_test.BaseTestClass method*), 45
unregister() (*mobly.controllers.android_device_lib.service_manager.ServiceManager method*), 18
unregister_all() (*mobly.controllers.android_device_lib.service_manager.ServiceManager method*), 18
unregister_controllers() (*mobly.controller_manager.ControllerManager method*), 47
update_config() (*mobly.controllers.android_device_lib.services.logcat.Logcat method*), 4
update_record() (*mobly.records.TestResultRecord method*), 55
update_serial() (*mobly.controllers.android_device.AndroidDevice method*), 27
USER_DATA (*mobly.records.TestSummaryEntryType attribute*), 56
user_id (*mobly.controllers.android_device_lib.snippet_client.SnippetClient attribute*), 19
user_params (*mobly.base_test.BaseTestClass attribute*), 41
user_params (*mobly.config_parser.TestRunConfig attribute*), 46

V

verify_controller_module() (*in module mobly.controller_manager*), 47

W

wait_for_boot_completion() (*mobly.controllers.android_device.AndroidDevice method*), 28
wait_for_capture() (*mobly.controllers.sniffer.Sniffer method*), 34
wait_for_capture() (*mobly.controllers.sniffer_lib.local.local_base.SnifferLocalBase method*), 22
wait_for_event() (*mobly.controllers.android_device_lib.event_dispat method*), 13
wait_for_standing_subprocess() (*in module mobly.utils*), 65
waitAndGet() (*mobly.controllers.android_device_lib.callback_handler.C method*), 9
waitForEvent() (*mobly.controllers.android_device_lib.callback_handl method*), 9