Python GLib Log Bridge Documentation

Neui

Mar 20, 2021

TABLE OF CONTENTS

This library allows you to redirect either from the python logger facility to the glib logging facility, or the other way around.

NOTE: THIS IS STILL IN DEVELOPMENT! IT MIGHT NOT WORK CORRECTLY, AND THE INTERFACE CAN CHANGE!

CHAPTER

ONE

QUICK USAGE

Look at the documentation for more information, more examples and the API reference.

1.1 GLib → Python

```
from gi.repository import GLib
import glib_log_bridge.glib2python as glib2python
g2plog = glib2python.Logger()
GLib.log_set_writer_func(g2plog.logWriterFunc, None)
```

1.2 Python → GLib

```
import logging
import glib_log_bridge.python2glib as python2glib
handler = python2glib.LoggerHandler()
logging.getLogger().addHandler(handler)
# Logger to apply, logger.getLogger() does it for all messages
```

1.2.1 Usage

This library allows you to forward logs from the python side to the GLib side and the other way around. You can even forward logs to Python but still use GLib writer functions, such as forwarding to `journald`. Depending what you do, look at the correct chapter of what you want to do.

Forward GLib → Python

To be written.

Quick usage:

```
from gi.repository import GLib
import glib_log_bridge.glib2python as glib2python
g2plog = glib2python.Logger()
GLib.log_set_writer_func(g2plog.logWriterFunc, None)
```

After importing, `glib_log_bridge.glib2python.Logger` is being instantiated, which accepts the log messages and forwards them to the Python Logger. The `glib_log_bridge.glib2python.Logger.logWriterFunc()` is the actual writer function that is then passed to `GLib.log_set_writer_func()` to

actually receive the messages and forward them. The userdata-parameter is ignored, so it can be anything, or simply `None`.

Customizing

To be written.

Respect `G_MESSAGES_DEBUG`

To be written.

The filter `glib_log_bridge.glib2python.FilterGLibMessagesDebug` respects the `G_MESSAGES_DEBUG` environment variable by only forwarding debug messages if the logger names appears in them (space-separated, and dash-separated "namespaces") or `all` has been specified.

See also:

Running and debugging GLib Applications - Environment variables

This can then be applied to an `logging.Logger` or `logging.Handler` via the `logging.Logger.addFilter()` or `logging.Handler.addFilter()` methods.

Note: `G_MESSAGES_DEBUG` is being converted by replacing the dashes to dots, since dashes are used in the GLib world to separate namespaces, but dots in the Python world.

Warning: Filters don't get propagated when applied to a logger (so filters for the root logger get ignored by the `"GLib"`-logger). Because of that, better apply it to the handler instead.

Warning: Since the filters need to get the debug messages, you should set the log level of the `logging.Logger` and `logging.Handler` to `logging.DEBUG` or lower, so they can be processed by the filter.

Alternatively, instead of applying a filter, you can use `glib_log_bridge.glib2python.FilterGLibMessagesDebug.register_loggers()` which registers the filter as well as set the log level of all loggers specified in `G_MESSAGES_DEBUG` (or just the root logger when `all` is specified). This as the feature that `logging.Logger.isEnabledFor()` will properly work for `logging.DEBUG`, which can be used to do some more costly operations when debugging.

Forward Python → GLib

To be written.

Quick usage:

```
import logging
import glib_log_bridge.python2glib as python2glib
handler = python2glib.LoggerHandler()
logging.getLogger().addHandler(handler)
# Logger to apply, logger.getLogger() does it for all messages
```

After importing, an normal *glib_log_bridge.python2glib.LoggerHandler* is being instantiated, which accepts the log messages from a logger and forwards them to GLib. To register the handler, you need to use `logging.Logger.addHandler()` method on a `logging.Logger`. You most likely want to use the root logger `logging.getLogger()`, so all logs are forwarded.

Alternatively, you can forward a specific logger. Note that the full logger name is being used and converted to GLib format (that uses dashes instead of dots), so usually you don't need to do anything special if you just want to forward one logger (such as only forward the logs done by the application itself).

Customizing

To be written.

You can create a custom `logging.Filter` and add them via `logging.Handler.addFilter` to the *glib_log_bridge.python2glib.LoggerHandler* if you want to determine which exact messages should be forwarded.

Directly use a GLib writer/handler

The classes *glib_log_bridge.python2glib.GLibWriterHandler* and *glib_log_bridge.python2glib.GLibLogHandler* are like *glib_log_bridge.python2glib.LoggerHandler*, but instead forward to an GLib-compatible `GLib.LogWriterFunc` or `Glib.LogFunc`. They accept the corresponding function (and userdata) as the parameters when instantiating:

```
glibWriterHandlerDefault = GLibWriterHandler(GLib.log_writer_default)
```

Warning: *glib_log_bridge.python2glib.GLibLogHandler* uses the old non-structured GLib logging API, which only accepts the log domain, log level and the message itself. Other fields/information are dropped silently.

Instead please use *glib_log_bridge.python2glib.GLibWriterHandler*, which uses the newer structured GLib logging API and thus does not drop the additional fields.

There are pre-existing instances using the default writers GLib provides:

- *glib_log_bridge.python2glib.glibWriterHandlerDefault* (uses `GLib.log_writer_default()`)
- *glib_log_bridge.python2glib.glibWriterHandlerStandardStreams* (uses `GLib.log_writer_standard_streams()`)
- *glib_log_bridge.python2glib.glibWriterHandlerJournald* (uses `GLib.log_writer_journald()`)
- *glib_log_bridge.python2glib.glibLogHandlerDefault* (uses `GLib.log_default_handler()`, not recommended as per the warning above)

For example, if you want to forward to `jorunald`, you can do:

```
logging.getLogger().addHandler(python2glib.glibWriterHandlerJournald)
```

1.2.2 API Reference

GLib → Python

Python → GLib

class glib_log_bridge.python2glib.**GLibLogHandler**(*writer*, *user_data=None*, *level=0*, ***kwargs*)

Python logger handler that directly forwards to an GLib old-style log handler. Example:

```
>>> obj = GLibLogHandler(GLib.log_default_handler)
```

Warning: Uses the old-style GLib log API, so only the message, log domain and level are used, other fields are silently dropped. Use *GLibWriterHandler* instead.

Note that there is an pre-existing instance at:

- ***glibLogHandlerDefault*** (uses `GLib.log_default_handler()`)

Note that since this subclasses `logging.Handler`, view their documentation for more information, such as filters and so on.

Parameters

- **writer** (`Callable`[[`str`, `LogLevelFlags`, `str`, `Any`], `Any`]) –
- **user_data** (`Optional`[`Any`]) –

__init__(*writer*, *user_data=None*, *level=0*, ***kwargs*)

Initializes the instance, basically setting the formatter to `None` and the filter list to empty.

Parameters

- **handler** – The log handler function to forward to.
- **user_data** (`Optional`[`Any`]) – Additional data to forward to the handler function.
- **writer** (`Callable`[[`str`, `LogLevelFlags`, `str`, `Any`], `Any`]) –

_get_message(*record*)

Returns the message to pass to GLib for the specified record.

Parameters **record** (`LogRecord`) – The record to retrieve the message from.

Return type `str`

Returns The message to pass to GLib.

emit(*record*)

Log the specified record, by converting and forwarding it to the specified GLib Log Writer Function.

Normally, you wouldn't use this directly but rather implicitly via Pythons logging system.

Parameters **record** – The record to forward to GLib.

class glib_log_bridge.python2glib.**GLibWriterHandler**(*writer*, *user_data=None*, *level=0*, ***kwargs*)

Python logger handler that directly forwards to an GLib logger writer function. Example:

```
>>> obj = GLibWriterHandler(GLib.log_writer_default)
```

Note that there are pre-existing instances at:

- ***glibWriterHandlerDefault*** (uses `GLib.log_writer_default()`)
- ***glibWriterHandlerStandardStreams*** (uses `GLib.log_writer_standard_streams()`)
- ***glibWriterHandlerJournald*** (uses `GLib.log_writer_journald()`)

Note that since this subclasses `logging.Handler`, view their documentation for more information, such as filters and so on.

Parameters

- **writer** (`Callable`[[`LogLevelFlags`, `LogField`, `Any`], `LogWriterOutput`]) –
- **user_data** (`Optional`[`Any`]) –

__init__(*writer*, *user_data=None*, *level=0*, ***kwargs*)

Initializes the instance, basically setting the formatter to `None` and the filter list to empty.

Parameters

- **writer** (`Callable`[[`LogLevelFlags`, `LogField`, `Any`], `LogWriterOutput`]) – The writer function to forward to.
- **user_data** (`Optional`[`Any`]) – Additional data to forward to the writer function.

_convert_fields(*fields*)

Convert a record fields to an list of `GLib.LogField`.

Parameters **fields** (`Dict`[`str`, `Any`]) – The fields to convert.

Return type `List`[`LogField`]

Returns The converted fields.

_get_fields(*record*, ***kwargs*)

Return fields to use based on the given log record.

See *`LoggerHandler._get_fields()`* for more information.

This implementation will also set `GLIB_DOMAIN` when not set.

Parameters

- **record** (`LogRecord`) – The record to convert into a suitable `dict`.
- **update_from_record** (*`bool`*) – Extend it with `record.glib_fields`, when it exists and is a `dict`. Defaults to `True`.

Return type `Dict`[`str`, `Any`]

Returns Converted `dict` from the specified record.

_get_logfields(*record*)

Returns the `GLib.LogField` to pass to GLib for the specified record.

Parameters **record** (`LogRecord`) – The record to convert from.

Return type `List`[`LogField`]

Returns The fields to pass to GLib.

emit(*record*)

Log the specified record, by converting and forwarding it to the specified GLib Log Writer Function.

Normally, you wouldn't use this directly but rather implicitly via Pythons logging system.

Parameters **record** – The record to forward to GLib.

class glib_log_bridge.python2glib.**LoggerHandler**(*level=0*, *replace_module_char='-'*, *log_domain_prefix=''*, *log_domain_suffix=''*)

Python logger handle that just forwards message records to the GLib logger.

Note that since this subclasses `logging.Handler`, view their documentation for more information, such as filters and so on.

Parameters

- **replace_module_char** (`str`) – What to replace the dots (logger namespace separator) with when converting. Also see *LoggerHandler.replace_module_char*.
- **log_domain_prefix** (`str`) – What it should put before the converted logger name. Also see *LoggerHandler.log_domain_prefix*.
- **log_domain_suffix** (`str`) – What it should put after the converted logger name. Also see *LoggerHandler.log_domain_suffix*.

__init__(*level=0*, *replace_module_char='-'*, *log_domain_prefix=''*, *log_domain_suffix=''*)

Initializes the instance, basically setting the formatter to `None` and the filter list to empty.

Parameters

- **replace_module_char** (`str`) – What to replace the dots (logger namespace separator) with when converting. Also see *LoggerHandler.replace_module_char*.
- **log_domain_prefix** (`str`) – What it should put before the converted logger name. Also see *LoggerHandler.log_domain_prefix*.
- **log_domain_suffix** (`str`) – What it should put after the converted logger name. Also see *LoggerHandler.log_domain_suffix*.

_convert_fields_dict(*fields*)

Modifies a dictionary of the fields to convert their values into GLib Variants, ready to be passed into `GLib.log_variant()`.

By default, existing `GLib.Variant` objects are untouched, strings are converted to `Glib.Variant` strings, and `bytes` objects to `Glib.Variant` bytes, as per the official documentation.

For other objects `str()` is called and the resulting string is inserted.

Note that strings containing an null-byte will be cut off for that point. An warning will be emitted in that case.

Parameters **fields** (`Dict[str, Any]`) – The fields to convert.

Return type `Dict[str, Variant]`

Returns `fields`, which has been modified to be converted to `GLib.Variant`.

_get_fields(*record*, ***kwargs*)

Return fields to use based on the given log record.

The default implementation will insert the following keys:

- `MESSAGE`: The formatted message
- `CODE_FUNC`, `CODE_FILE`, `CODE_LINE`: Where it logged
- `PYTHON_MESSAGE`: The unformatted message
- `PYTHON_MODULE`: What module the log was emitted from
- `PYTHON_LOGGER`: To what logger name it was supposed to log to
- `PYTHON_TNAME`: Thread Name

- PYTHON_TID: Thread ID

The default implementaion will also insert exception information:

- PYTHON_EXC: Exception type with complete name
- PYTHON_EXC_MESSAGE: Stringify exception message

Additionally, the default implementation will also insert (and override) values from the glib_fields attribute of the record, if it exists and is a dict, when update_from_record is True (the default).

Subclasses can override this function to insert their own values and such. They can use the lower-scoped methods for more control:

- *LoggerHandler._get_fields_basic()*
- *LoggerHandler._get_fields_metadata()*
- *LoggerHandler._get_fields_exception()*
- *LoggerHandler._get_fields_record()*

Parameters

- **record** (LogRecord) – The record to convert into a suitable dict.
- **update_from_record** (*bool*) – Extend it with record.glib_fields, when it exists and is a dict. Defaults to True.

Return type Dict[str, Any]

Returns Converted dict from the specified record.

_get_fields_basic(*record*, *fields*)

Insert and return basic essential fields to use based on the given log record.

The default implementation will insert the following keys:

- MESSAGE: The formatted message
- CODE_FUNC, CODE_FILE, CODE_LINE: Where it logged

Parameters

- **record** (LogRecord) – The record to convert into a suitable dict.
- **fields** (Optional[Dict[str, Any]]) – The existing fields to update. If None, a new one is created and returned.

Return type Dict[str, Any]

Returns Converted dict from the specified record.

_get_fields_exception(*record*, *fields*)

Insert and return fields related to the current exception based on the given log record, if they contain them.

The default implementation will insert the following keys:

- PYTHON_EXC: Exception type with complete name
- PYTHON_EXC_MESSAGE: Stringify exception message

They won't be inserted when no exception is available.

Parameters

- **record** (LogRecord) – The record to convert into a suitable dict.

- **fields** (Optional[Dict[str, Any]]) – The existing fields to update. If None, a new one is created and returned.

Return type Dict[str, Any]

Returns Converted dict from the specified record.

_get_fields_metadata(*record*, *fields*)
Return basic essential fields to use based on the given log record.

The default implementation will insert the following keys:

- PYTHON_MESSAGE: The unformatted message
- PYTHON_MODULE: What module the log was emitted from
- PYTHON_LOGGER: To what logger name it was supposed to log to
- PYTHON_TNAME: Thread Name
- PYTHON_TID: Thread ID

Parameters

- **record** (LogRecord) – The record to convert into a suitable dict.
- **fields** (Optional[Dict[str, Any]]) – The existing fields to update. If None, a new one is created and returned.

Return type Dict[str, Any]

Returns Converted dict from the specified record.

_get_fields_record(*record*, *fields*)
Insert and return additional fields specified in the given log record glib_fields attribute of the record, if it exists.

Parameters

- **record** (LogRecord) – The record to convert into a suitable dict.
- **fields** (Optional[Dict[str, Any]]) – The existing fields to update. If None, a new one is created and returned.

Return type Dict[str, Any]

Returns Converted dict from the specified record.

_get_log_domain(*record*)
Returns the log domain for the specified record. The default implementation takes *LoggerHandler.log_domain_prefix* and *LoggerHandler.log_domain_prefix* into consideration.

Parameters **record** (LogRecord) – The record to retrieve (and convert) the log domain from.

Return type str

Returns The log domain name to use to log to GLib.

_level_to_glib(*level*, *default=gi.repository.GLib.LogLevelFlags.LEVEL_DEBUG*)
Converts a Python loglevel to a GLib log level. If no mapping exists, use the specified default value.

The default implementation will use the *LoggerHandler._level_to_glib_map* map.

Parameters

- **level** (int) –

- **default** (LogLevelFlags) –

Return type LogLevelFlags

_level_to_glib_map: Dict[int, gi.repository.GLib.LogLevelFlags] = {10: gi.repository
Map used to convert from a Python logger level to the GLib Log Level.

emit(*record*)
Log the specified record, by converting and forwarding it to the GLib logging system.

Normally, you wouldn't use this directly but rather implicitly via Pythons logging system.

Parameters **record** (LogRecord) – The record to log to GLib.

log_domain_prefix: str = ''
What it should put before the converted logger name.

log_domain_suffix: str = ''
What it should put after the converted logger name.

replace_module_char: str = '-'
What to replace the dots (logger namespace separator) with when converting.

glib_log_bridge.python2glib.**glibLogHandlerDefault = <GLibLogHandler (NOTSET)>**
Python Logger Handler to forward to GLib.log_default_handler().

> **Warning:** Uses the old-style GLib log API, so only the message, log domain and level are used, other fields are silently dropped. Use *GLibWriterHandler* or one of their pre-existing instances instead.

glib_log_bridge.python2glib.**glibWriterHandlerDefault = <GLibWriterHandler (NOTSET)>**
Python Logger Handler to forward to GLib.log_writer_default().

glib_log_bridge.python2glib.**glibWriterHandlerJournald = <GLibWriterHandler (NOTSET)>**
Python Logger Handler to forward to GLib.log_writer_journald().

glib_log_bridge.python2glib.**glibWriterHandlerStandardStreams = <GLibWriterHandler (NOTSET)>**
Python Logger Handler to forward to GLib.log_writer_standard_streams().

- genindex
- modindex
- search

PYTHON MODULE INDEX

g

INDEX

Symbols

E

G

L

M

R