Simplifying, Consolidating & Documenting LLDB’s Scripting Functionalities

Ismail Bennani

EuroLLVM 2024 | Apple Inc | April 11th 2024
“LLVM Project is a collection of modular and reusable compiler and toolchain technologies”

The LLVM Project website
Scripting API

LLDB
Scripting API

(lldb) script

>>> target = lldb.dbg.CreateTarget("a.out")
>>> bkpt = target.BreakpointCreateByLocation("main.c", 42)

>>> process = target.Launch(lldb.SBLaunchInfo(None), lldb.SBError())

>>> thread = process.GetSelectedThread()
>>> frame_0 = thread.GetFrameAtIndex(0)

>>> frame_0.FindVariable("foo")
(int) foo = 19
Scripting Extensions

- Data Formatter
- Custom Command
- Operating System Plugin
- Scripted Process
- Scripted Thread Plan
- Breakpoint Command
- Watchpoint Command
- Target Stop Hook
Scripting Extensions

Data Formatter
Custom Command
Scripted Thread Plan
Watchpoint Command
Target Stop Hook
Scripted Process
Breakpoint Command
Operating System Plugin
Scripting Extensions

Data Formatter Example:

class MySingleChildProvider:
    def __init__(self, valobj, dict):
        self.valobj = valobj

    def num_children(self):
        return 1

    def has_children(self):
        return True

    def get_child_index(self, name):
        return 0
def has_children(self):
    return True

def get_child_index(self, name):
    return 0

def get_child_at_index(self, index):
    if index != 0 or not self.valobj.IsValid():
        return None
    return self.valobj.GetChildAtIndex(0)

def update(self):
    pass

Scripting Extensions

Data Formatter Example:

```python
def get_child_at_index(self, index):
    if index != 0 or not self.valobj.IsValid():
        return None
    return self.valobj.GetChildAtIndex(0)

def update(self):
    pass
```
Scripting Extensions

Data Formatter
Custom Command
Scripted Thread Plan
Watchpoint Command
Target Stop Hook
Scripted Process
Breakpoint Command
Operating System Plugin
1. Improve discoverability
2. Keep documentation up-to-date
3. Reduce boilerplate code
4. Reduce high maintenance cost
1. Improve discoverability
2. Keep documentation up-to-date
3. Reduce boilerplate code
4. Reduce high maintenance cost
1. Improve discoverability
2. Keep documentation up-to-date
3. Reduce boilerplate code
4. Reduce high maintenance cost
1. Improve discoverability

2. Keep documentation up-to-date

3. Reduce boilerplate code

4. Reduce high maintenance cost
1. Improve discoverability
2. Keep documentation up-to-date
3. Reduce boilerplate code
4. Reduce high maintenance cost
Scripted Process 101

Interactive Crashlogs in LLDB
Scripted Process 101

Scripted Process

LLDB

lldb.py

scripted_process.py
Scripted Process 101

Scripted Process  <->  LLDB  <->  py

lldb.py  <->  scripted_process.py

html

lldb.llvm.org
The LLDB Debugger

Welcome to the LLDB documentation!

LLDB is a next generation, high-performance debugger. It is built as a set of reusable components which highly leverage existing libraries in the larger LLVM Project, such as the Clang expression parser and LLVM disassembler.

LLDB is the default debugger in Xcode on macOS and supports debugging C, Objective-C and C++ on the desktop and iOS devices and simulator.

All of the code in the LLDB project is available under the "Apache 2.0 License with LLVM exceptions".

Using LLDB

For an introduction into the LLDB command language, head over to the LLDB Tutorial. For users already familiar with GDB there is a cheat sheet listing common tasks and their LLDB equivalent in the GDB to LLDB command map.

There are also multiple resources on how to script LLDB using Python: the Python Reference is a great starting point for that.

Compiler Integration Benefits

LLDB converts debug information into Clang types so that it can leverage the Clang compiler infrastructure. This allows LLDB to support the latest C, C++, Objective-C and Objective-C++ language features and runtimes in expressions without having to reimplement any of this functionality. It also leverages the compiler to take care of all ABI details when making functions calls for expressions, when disassembling instructions and extracting instruction details, and much more.

The major benefits include:

- Up to date language support for C, C++, Objective-C
- Multi-line expressions that can declare local variables and types
- Utilize the JIT for expressions when supported
- Evaluate expression Intermediate Representation (IR) when JIT can’t be used
LLDB Python API

Illdb Package

The Illdb module contains the public APIs for Python binding.

Some of the important classes are described here:

- **SBTarget**: Represents the target program running under the debugger.
- **SBProcess**: Represents the process associated with the target program.
- **SBThread**: Represents a thread of execution. **SBProcess** contains **SBThreads**.
- **SBFrame**: Represents one of the stack frames associated with a thread. **SBThread** contains **SBFrame**s.
- **SBSymbolContext**: A container that stores various debugger related info.
- **SVValue**: Represents the value of a variable, a register, or an expression.
- **SBModule**: Represents an executable image and its associated object and symbol files. **SBTarget** contains **SBModule**.
- **SBBreakpoint**: Represents a logical breakpoint and its associated settings. **SBTarget** contains **SBBreakpoints**.
- **SBSymbol**: Represents the symbol possibly associated with a stack frame.
- **SBCompileUnit**: Represents a compilation unit, or compiled source file.
- **SBFunction**: Represents a generic function, which can be inlined or not.
- **SBBlock**: Represents a lexical block. **SBFunction** contains **SBBlocks**.
- **SBLLineEntry**: Specifies an association with a contiguous range of instructions and a source file location. **SBCompileUnit** contains **SBLLineEntry**.

The different enums in the **Illdb** module are described in Python API enumerators and constants.

Classes

- **SBAddress**(*args*)
  A section + offset based address class.
- **SBAttachInfo**(*args*)
  Describes how to attach when calling **SBTarget.Attach**.
- **SBBlock**(*args*)
  Represents a lexical block.
SBLLineEntry

class lldb.SBLLineEntry(*args)

Specifies an association with a contiguous range of instructions and a source file for it. `SBLLineEntry` contains `SBLLineEntry` objects. For example,

```python
for lineEntry in compiledUnit:
    print('line entry: %s' % str(lineEntry.GetFileSpec()),
          lineEntry.GetLine()))
    print('start addr: %s %s %s' % (str(lineEntry.GetStartAddress())))
    print('end addr: %s' % str(lineEntry.GetEndAddress()))
```

produces:

```
line entry: /Volumes/data/llvm/trunk/python_api/symbol-context/main.c:2:
start addr: a.out[0x800000000022]
end addr: a.out[0x800000000022]
line entry: /Volumes/data/llvm/trunk/python_api/symbol-context/main.c:3:
start addr: a.out[0x800000000022]
end addr: a.out[0x800000000022]
line entry: /Volumes/data/llvm/trunk/python_api/symbol-context/main.c:22:
start addr: a.out[0x800000000022]
end addr: a.out[0x800000000022]
line entry: /Volumes/data/llvm/trunk/python_api/symbol-context/main.c:23:
start addr: a.out[0x800000000022]
end addr: a.out[0x800000000022]
...
```

See also `SBCOMPILEUNIT`.

**ATTRIBUTES SUMMARY**

- **addr**
  A read only property that returns an lldb object that represents the start address (lldb.LBAAddress) for this line entry.

- **column**
  A read only property that returns the 1 based column number for this line entry, a return value of zero indicates that no column information is available.

- **end**
  A read only property that returns an lldb object that represents the end address.
ScriptedProcess

```python
class lldb.plugins.scripted_process.ScriptedProcess(exe_ctx, args)
```

The base class for a scripted process.

Most of the base class methods are `abstractmethod` that need to be overwritten by the inheriting class.

**ATTRIBUTES SUMMARY**

- capabilities
- loaded_images
- memory_regions
- metadata
- threads

**METHODS SUMMARY**

- `attach (attach_info)`
  - Simulate the scripted process attach.

- `create_breakpoint (addr, error)`
  - Create a breakpoint in the scripted process from an address.

- `get_capabilities ()`
  - Get a dictionary containing the process capabilities.

- `get_loaded_images ()`
  - Get the list of loaded images for the scripted process.

- `get_memory_region_containing_address (addr)`
  - Get the memory region for the scripted process, containing a

- `get_process_id ()`
  - Get the scripted process identifier.

- `get_process_metadata ()`
  - Get some metadata for the scripted process.
Scripted Process 101

Scripted Process

LLDB

lldb.py

scripted_process.py

html

lldb.llvm.org
import llmdb
from llmdb.plugins.scripted_process import ScriptedProcess
from llmdb.plugins.scripted_process import ScriptedThread

class MyScriptedProcess(ScriptedProcess):
    
    def __init__(self, target, args):
        super().__init__(target, args)
Scripting Interfaces Architecture

Scripted Process <-> LLDB <-> my_scripted_process.py
Scripting Interfaces Architecture

Scripted Process Interface

- Attach();
- Launch();
- Resume();
- GetThreadsInfo();
- CreateBreakpoint();
- ReadMemoryAtAddress();
- WriteMemoryAtAddress();
- GetProcessID();

my_script.py
Scripting Interfaces Architecture

Scripted Process
- Lua Interface
Scripted Thread
- Python Interface
- Lua Interface

my_script.py
Scripting Interfaces Architecture

Scripted Process
- Scripted Process Interface
- Scripted Process Lua Interface
- Scripted Process Python Interface

Scripted Thread
- Scripted Thread Interface
- Scripted Thread Lua Interface
- Scripted Thread Python Interface

my_script.py
Scripted Process Interface

Scripted Lua Interface

Scripted Python Interface

my_script.py

1. Import & register the python class

(lldb) command script import /tmp/my_scripted_process.py

(lldb) process launch --script-class my_scripted_process.MyScriptedProcess
Scripting Python Interface Usage

1. Import & register the python class
2. Create the interfaces
Scripting Python Interface Usage

1. Import & register the python class
2. Create the interfaces
3. Instantiate the script object
“Classes are callable. [...] The arguments of the call are passed [...] to \_\_init\_\_() to initialize the new instance.”

The Python Documentation website, Data Model, 3.2.8.8. Classes
Scripting Python Interface Usage

1. Import & register the python class
2. Create the interfaces
3. Call the script methods

```
my_script.py
```
Scripting Python Interface Usage

```
my_script.py

Attach()
Launch()
Resume()
GetThreadsInfo()
CreateBreakpoint()
ReadMemoryAtAddress()
WriteMemoryAtAddress()
GetProcessID()

Data ReadMemoryAtAddress(size_t addr,
size_t size,
bool &error) {
    Data data = Dispatch<Data>("read_memory_at_address",
    error, addr, size);
    if (error)
        return {};
    return data;
}
```
Import & register the python class

Create the interfaces

Instantiate the script object
Call the script methods

Scripted Process
Python Interface

my_script.py

Attach()
Launch()
Resume()
GetThreadsInfo()
CreateBreakpoint()
ReadMemoryAtAddress()
WriteMemoryAtAddress()
GetProcessID()

Scripted Python Interface::Dispatch

```
template<typename T, typename... Args>
T Dispatch(llvm::StringRef method_name, bool &error, Args &&...args);
```

```
Data ReadMemoryAtAddress(size_t addr, size_t size, bool &error) {
    Data data = Dispatch<Data>("read_memory_at_address", error, addr, size);
    if (error)
        return {};
    return data;
}
```
Scripted Python Interface::Dispatch

1. Import & register the python class
2. Create the interfaces
3. Call the script methods

A. Resolve method object
1. Import & register the python class
2. Create the interfaces
3. Call the script methods

A. Resolve method object
B. Transform arguments & make call
Object Calling API

Various functions are available for calling a Python object. Each converts its arguments to a convention supported by the called object – either tp_call or vectorcall. In order to do as little conversion as possible, pick one that best fits the format of data you have available.

The following table summarizes the available functions; please see individual documentation for details.

<table>
<thead>
<tr>
<th>Function</th>
<th>callable</th>
<th>args</th>
<th>kwargs</th>
</tr>
</thead>
<tbody>
<tr>
<td>PyObject_Call()</td>
<td>PyObject</td>
<td>tuple</td>
<td>dict</td>
</tr>
<tr>
<td>PyObject_CallNoArgs()</td>
<td>PyObject</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>PyObject_CallOneArg()</td>
<td>PyObject</td>
<td>1 object</td>
<td>-</td>
</tr>
<tr>
<td>PyObject_CallObject()</td>
<td>PyObject</td>
<td>tuple/NULL</td>
<td>-</td>
</tr>
<tr>
<td>PyObject_CallFunction()</td>
<td>PyObject</td>
<td>format</td>
<td>-</td>
</tr>
<tr>
<td>PyObject_CallMethod()</td>
<td>obj + char*</td>
<td>format</td>
<td>-</td>
</tr>
<tr>
<td>PyObject_CallFunctionObjArgs()</td>
<td>PyObject</td>
<td>variadic</td>
<td>-</td>
</tr>
<tr>
<td>PyObject_CallMethodObjArgs()</td>
<td>obj + name</td>
<td>variadic</td>
<td>-</td>
</tr>
<tr>
<td>PyObject_CallMethodNoArgs()</td>
<td>obj + name</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>PyObject_CallMethodOneArg()</td>
<td>obj + name</td>
<td>1 object</td>
<td>-</td>
</tr>
<tr>
<td>PyObject_Vectorcall()</td>
<td>PyObject</td>
<td>vectorcall</td>
<td>vectorcall</td>
</tr>
<tr>
<td>PyObject_VectorcallDict()</td>
<td>PyObject</td>
<td>vectorcall</td>
<td>dict</td>
</tr>
<tr>
<td>PyObject_VectorcallMethod()</td>
<td>arg + name</td>
<td>vectorcall</td>
<td>vectorcall</td>
</tr>
</tbody>
</table>

\[
\text{PyObject} \ast \text{PyObject_Call}(\text{PyObject} \ast \text{callable}, \text{PyObject} \ast \text{args}, \text{PyObject} \ast \text{kwargs})
\]

Return value: New reference. Part of the Stable ABI.

Call a callable Python object callable, with arguments given by the tuple args, and named arguments given by the dictionary kwargs.

args must not be NULL; use an empty tuple if no arguments are needed. If no named arguments are needed, kwargs can be NULL.

Return the result of the call on success, or raise an exception and return NULL on failure.

This is the equivalent of the Python expression: \text{callable*(args, **kwargs)}. 


ScriptedPythonInterface::Dispatch

1. Import & register the python class
2. Create the interfaces
3. Call the script methods

A. Resolve method object
B. Transform arguments & make call
C. Reverse transform arguments & return type

diagram showing Scripted Process, Scripted Process Interface, Scripted Process Python Interface, and my_script.py
Conclusion
Conclusion

Reduced boilerplate code with scripting extensions base class
Conclusion

Reduced boilerplate code with scripting extensions base class

Used the base class to keep documentation up-to-date
Conclusion

Reduced boilerplate code with scripting extensions base class

Used the base class to keep documentation up-to-date

Built a unified, robust and generic infrastructure to interface with scripting extensions
Conclusion

Reduced boilerplate code with scripting extensions base class

Used the base class to keep documentation up-to-date

Built a unified, robust and generic infrastructure to interface with scripting extensions

Still more work to come ...
(lldb) scripting template list
Available scripted extensions:
  Name: ScriptedProcessPythonInterface
  Language: Python
  Description: Mock process state
  Command Interpreter Usages:
    process attach -C <script-name> [-k key -v value ...]
    process launch -C <script-name> [-k key -v value ...]
  API Usages:
    SBAttachInfo.SetScriptedProcessClassName
    SBAttachInfo.SetScriptedProcessDictionary
    SBTarget.Attach
    SBLaunchInfo.SetScriptedProcessClassName
    SBLaunchInfo.SetScriptedProcessDictionary
    SBTarget.Launch
Call to action

Data Formatters
Custom Commands
Scripted Thread Plans
Watchpoint Commands
Target Stop Hooks
Scripted Processes
Breakpoint Commands
Operating System Plugins
Simplifying, Consolidating & Documenting LLDB’s Scripting Functionalities

Ismail Bennani

EuroLLVM 2024 | Apple Inc | April 11th 2024