nChain

Script Engine SDK v1.0.0

Document Version: 1.0.0

1. Introduction

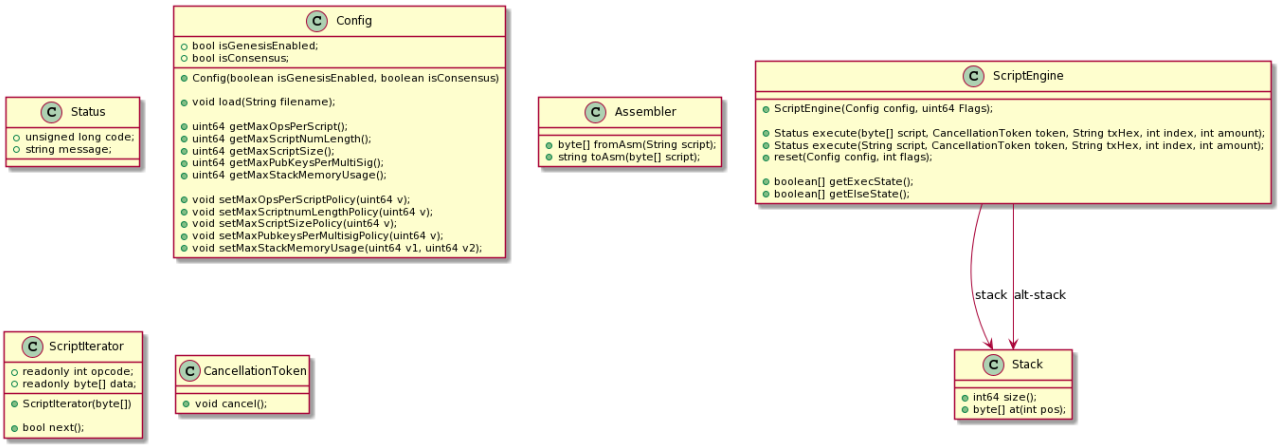
The Script Engine SDK v1.0.0 provides Java language bindings for the SV node script engine.

The Script Engine SDK is intended to assist in the development of complex script applications, including smart contracts. The SDK provides support for those wanting to create development or debugger environments or integrate the script engine into existing environments (VS Code, Eclipse), as well as providing necessary support for high level languages interpreters and compilers.

Future versions of the SDK will provide multiple language bindings for the script engine. The v1.0.0 release contains Python bindings from a pre-release version (v0.0.1) of the SDK; use at your own risk.

2. SDK Object Module

The Script Engine object model is shown in the UML diagram below. The associated Plant UML file can be found in the appendix.



All language bindings implement the same object model, or as close as the language allows (Only Java is provided in version 1.0.0)

3. Objects

3.1. ExecutionStatus

The *ExecutionStatus* object is returned by the script engine after execution of a script. The returned code and messages are the same as the codes and messages returned by the C++ script engine and can be found in the Appendix.

3.1.1. unsigned long code

Returns the status code stored in the *ExecutionStatus* after execution of a script.

3.1.2. string message

Returns the message associated with the status code stored in the *ExecutionStatus*.

3.2. Config

The *Config* object is passed into the script engine constructor; it provides the script engine with policy and consensus limits for

- Maximum script size
- Maximum script number length
- Maximum number of public keys in a multi-sig.
- Maximum number of operations per script
- Maximum stack memory usage (both stacks combined)

Setting a configuration value is subject to constraints and may fail, in which case an exception will be thrown.

The *Stack* object implements the *Autoclosable* interface.

The Config object, more than most, reflects the structure of the underlying C++ code. It is possible that the *Config* interface will be changed in future versions of the SDK.

3.2.1. Config(boolean IsGenesis, boolean useConsensusLimits)

Constructs *Config* object with specified values for (a) IsGenesisEnabled and (b) IsConsensus.

3.2.2. Config()

Constructs a *Config* object that uses Genesis opcodes and consensus limits.

3.2.3. void load(string filename)

This method loads the configuration from file. This method should be called before the *Config* object is passed into the *ScriptEngine* constructor.

The method will throw an exception if it is passed a null filename. It does not return an error if the file does not exist.

3.2.4. void setMaxOpsPerScriptPolicy(long v)
3.2.5. void setMaxScriptnumLengthPolicy(long v)
3.2.6. void setMaxScriptSizePolicy(long v)
3.2.7. void setMaxPubkeysPerMultisigPolicy(long v)

These methods set policy limits.

3.2.8. void setMaxStackMemoryUsage(long maxStackMemoryConsensus, long maxStackMemoryPolicy)

This method sets both the consensus and policy maximum stack memory usage.

3.2.9. long getMaxOpsPerScript()
3.2.10. long getMaxScriptNumLength()
3.2.11. long getMaxScriptSize()
3.2.12. long getMaxPubKeysPerMultiSig()
3.2.13. long getMaxStackMemoryUsage()

These methods are used by the script engine to determine the limits it should use internally.

The methods retrieve limits dependent on the *Config* object settings. For example, if the *Config* object is constructed to use consensus limits, then these methods will return consensus limits. Similarly, if the *Config* object is setup to use policy limits, then these methods will return policy limits.

3.3. Assembler

The Assembler object is used to convert assembly to and from binary script.

3.3.1. byte[] fromAsm(string asm)

Converts from assembly to binary script.

3.3.2. string toAsm(byte[] script)

Converts from binary script to assembly.

3.3.3. Notes

The conversion between assembly and binary script, and vice versa, may lose information.

Example: The script 0x54 0x55 0x93 0x59 0x87 corresponds to the human readable assembly "4 5 ADD 9 EQUAL". However, so does the script 0x04 0x05 0x93 0x09 0x87. I.e. In this case, the conversion script → assembly → script does not restore the original script.

3.4. ScriptIterator

The script iterator allows a user to step through a script and opcode or data element at a time.

If an invalid script iterator is dereferenced, the iterator will throw an exception.

3.4.1. ScriptIterator(byte[] script)

Constructs a *ScriptIterator* on script.

3.4.2. boolean next()

Moves the iterator to the next element in the script. If the iterator points to the last script element, then the iterator becomes invalid and *next*() returns false.

If the iterator is not invalid, *next*() moves the iterator to the next element and returns true.

3.4.3. bool reset()

Resets the iterator back to its initial position.

3.4.4. int getOpcode()

Returns the opcode pointed to by the. If the iterator points to an opcode, then *getData*() should return ***null***.

3.4.5. byte[] getData()

Returns the data element pointed to by the iterator. If the iterator points to an opcode, then *getData*() should return ***null***.

3.4.6. Example

The script iterator allows users to step through a script and opcode at a time and examine the contents of the stacks. This type of behaviour is required for debugger.

The code below shows how to step through a script. A debugger could feed each opcode/data returned by the iterator into a script engine.

```
final ScriptIterator it = createScriptIterator("4 5 OP_ADD 9 OP_EQUAL");

Assert.assertTrue(it.next());
Assert.assertTrue(it.getOpcode() == OpCode._4);

Assert.assertTrue(it.next());
Assert.assertTrue(it.getOpcode() == OpCode._5);

Assert.assertTrue(it.next());
Assert.assertTrue(it.getOpcode() == OpCode.ADD);

Assert.assertTrue(it.next());
Assert.assertTrue(it.getOpcode() == OpCode._9);

Assert.assertTrue(it.next());
Assert.assertTrue(it.getOpcode() == OpCode.EQUAL);

// The iterator is at the end of the script.
```

```
Assert.assertFalse(it.next());
```

3.5. CancellationToken

The cancellation token allows a user to cancel execution of a long running script from another thread.

The cancellation token is a parameter in the *ScriptEngine*.*execute* method

The *CancellationToken* object implements the *Autoclosable* interface.

3.5.1. void cancel()

No parameters. Call to cancel execution of a long running script from another thread.

3.6. ScriptEngine

The *ScriptEngine* interprets scripts stored in BSV transactions.

The *ScriptEngine* object implements the *AutoClosable* interface; closing the *ScriptEngine* also closes the contained *Stack* objects.

3.6.1. ScriptEngine(Config config, int Flags)

The passed *Config* object reference and *Flags* are stored internally as in the *ScriptEngine* and use to determine limits as well as

3.6.2. unsigned long Flags

Stored flags passed into the script engine. A list of flags is found in the Appendix.

3.6.3. ExecutionStatus evaluate(byte[] script, byte[] transaction, bool consensus, CancellationToken token)

Executes the script passed into the script engine.

3.6.4. Stack getStack()

Returns access to the script engine main stack.

3.6.5. Stack getAltStack()

Returns access to the script engine alt-stack.

3.6.6. reset(Config config, int flags)

Resets the script engine back to the state it would be in if it had been constructed using the passed config and flags.

3.6.7. Example

The code below shows the creation of a configuration object and script engine. The script engine is used to execute a script.

```
Config c = new Config();
ScriptEngine se = new ScriptEngine(c, ScriptFlag.VERIFY_NONE);
final String scriptStr = new String("4 5 OP_ADD");

{
	Status result = se.execute(scriptStr, token, "", 0, 0);
	Assert.assertEquals(Status.OK, result.getCode());
	Stack stack = se.getStack();
	Assert.assertEquals(stack.size(), 1);
	byte[] value = stack.at(0);
	Assert.assertEquals(value[0], 9);
}
```

The script engine needs to be reset before it is used again, otherwise the script engine will assume that it should continue using it current state.

```
se.reset(c, 0);
{
	Stack stack = se.getStack();
	Assert.assertEquals(stack.size(), 0);   // stack cleared.
}
```

The script is executed again, and the same result achieved.

```
{
	Status result2 = se.execute(scriptStr, token, "", 0, 0);
	Assert.assertEquals(Status.OK, result2.getCode());
	Stack stack2 = se.getStack();
	Assert.assertEquals(stack2.size(), 1);
	byte[] value2 = stack2.at(0);
	Assert.assertEquals(value2[0], 9);
}
```

Finally the script engine and configuration object are closed.

```
se.close();
final long v1 = c.getMaxOpsPerScript();   // config is still valid.
c.close();
```

3.7. Stack

The S*tack* objects are the primary internal data structures manipulated by the script engine.

The stacks are normally only updated by executing scripts containing opcodes and data in the script engine.

The *Stack* object implements the *Autoclosable* interface.

3.7.1. long size()

Returns the current stack size (the number of elements on the stack)

3.7.2. byte[] at(int pos)

Returns the data element at position *pos* on the stack, throws an exception if out of bounds.

3.8. Memory Management

The following Java object implement *AutoDisposable* interface.

- CancellationToken
- Config
- ScriptEngine
- Stack

The *ScriptEngine.close* method calls the close methods on the associated config and both stacks. Developers should use patterns such as *try-with-resources* to ensure that memory is reclaimed appropriately.

4. Appendix

The script used to generate the Object Model UML diagram is shown below.

4.9. Plant UML file

```
@startuml
class Status
{
        +unsigned long code;
        +string message;
}
class Config
{
        +Config(boolean isGenesisEnabled, boolean isConsensus)

  +void load(String filename);

  +uint64 getMaxOpsPerScript();
  +uint64 getMaxScriptNumLength();
  +uint64 getMaxScriptSize();
  +uint64 getMaxPubKeysPerMultiSig();
  +uint64 getMaxStackMemoryUsage();

  +void setMaxOpsPerScriptPolicy(uint64 v);
  +void setMaxScriptnumLengthPolicy(uint64 v);
  +void setMaxScriptSizePolicy(uint64 v);
  +void setMaxPubkeysPerMultisigPolicy(uint64 v);
  +void setMaxStackMemoryUsage(uint64 v1, uint64 v2);

  +bool isGenesisEnabled;
  +bool isConsensus;
}
class Assembler
{
  +byte[] fromAsm(String script);
  +string toAsm(byte[] script);
}
class ScriptIterator
{
        +ScriptIterator(byte[])

        +bool next();
        +readonly int opcode;
        +readonly byte[] data;
}
class CancellationToken
{
        +void cancel();
}
class Stack
{
  +int64 size();
```

```
   +byte[] at(int pos);
}
class ScriptEngine
{
        +ScriptEngine(Config config, uint64 Flags);

        +Status execute(byte[] script, CancellationToken token, String txHex, int index, int amount);
        +Status execute(String script, CancellationToken token, String txHex, int index, int amount);

        +reset(Config config, int flags);

        +boolean[] getExecState();
        +boolean[] getElseState();
}
ScriptEngine --> Stack : stack
ScriptEngine --> Stack : alt-stack
@enduml
```