Plugeth

Release Austin Roberts

Philip Morlier, Austin Roberts

Sep 12, 2023

OVERVIEW

The Geth fork to end all Forks.

PluGeth is a fork of the Go Ethereum Client, Geth, implementing the Golang plugin architecture allowing developers to adapt and extend Geth's capabilities using plugins rather than having to create additional new forks.

The PluGeth project aims to provide a secure and versitile tool for anyone who needs to run a Geth (or Geth-derived) node client that supports features beyond those offered by Geth's vanilla EVM.

All dependencies and updates are handled by the PluGeth project, and so, PluGeth enables developers to focus on their projects without having to maintian upstream code.

- *Project Design*
- *Install*
- *Build and Deploy*
- *Writing a Custom Plugin*

CHAPTER

ONE

PROJECT DESIGN

1.1 Design Goals

Upstream Geth exists primarily to serve as a client for the Ethereum mainnet, though it also supports a number of popular testnets. The Geth team generally avoids changes to support other networks, or to provide features only a small handful of users would be interested in.

The result is that many projects have forked Geth. Some implement their own consensus protocols or alter the behavior of the EVM to support other networks. Others are designed to extract information from the Ethereum mainnet in ways the standard Geth client does not support.

Creating numerous different forks to fill a variety of different needs comes with a number of drawbacks. Forks tend to drift apart from each other. Many networks that forked from Geth long ago have stopped merging updates; this makes some sense, given that those networks have moved in different directions than Geth and merging upstream changes while properly maintaining consensus rules of an existing network could prove quite challenging. But not merging changes from upstream can mean that security updates are easily missed, especially when the upstream team obscures security updates as optimizations as a matter of process.

PluGeth aims to provide a single Geth fork that developers can choose to extend rather than forking the Geth project. Out of the box, PluGeth behaves exactly like upstream Geth, but by installing plugins written in Golang, developers can extend its functionality in a wide variety of ways.

1.1.1 Three Repositories

PluGeth is an application built in three repositories:

1.1.2 PluGeth

The largest of the three, PluGeth is a fork of Geth which has been modified to enable a plugin architecture. The Plugin loader, wrappers, and hooks all reside in this repository.

1.1.3 PluGeth-Utils

Utils are small packages used to develop PluGeth plugins without Geth dependencies. For a more detailed analysis of the reasons see *PluGeth-utils Subpackages*. Imports from Utils happen automatically and so most users need not clone a local version.

1.1.4 PluGeth-Plugins

The packages from which plugins are buile are stored here. This repository contains premade plugins as well as providing a location for storing new custom plugins.

Version Control

Before using Plugeth users are enocuraged to familiarize themselves with the *version control* scheme of the project.

CHAPTER

TWO

BASIC TYPES OF PLUGINS

While PluGeth has been designed to be versatile and customizable, when learning the project it can be helpful to think of plugins as being of four different archetypes.

2.1 RPC Methods

These plugins provide new json rpc methods to access several objects containing real time and historic data.

2.2 Subcommand

A subcommand redefines the total behavior of Geth and could stand on its own. In contrast with the other plugin types which, in general, are meant to capture and manipulate information, a subcommand is meant to change the overall behavior of Geth. It may do this in order to capture information but the primary fuctionality is a modulation of geth behaviour.

2.3 Tracers

Tracers are used to collect information about a transaction or block during EVM execution. They can be written to refine the results of a `debug` trace or written to delver custom data using various other objects available from the PluGeth APIs.

2.4 Subscriptions

Subscriptions provide real time notification of data from the EVM as it processes transactions.

Note: Plugins are not limited to a singular functionality and can be customized to operate as hybrids of the above. See blockupdates as an example.

CHAPTER

THREE

INSTALL

Note: Prior to install make sure to be familiar with *system requirements*.

PluGeth can be installed in two ways. The repositories can be cloned and compiled from the source code. Alternatively PluGeth provides binaries of a PluGeth node as well as plugins.

In order to run PluGeth without the source code, download the latest release here.

The curated list of plugin builds can be found here

Note: Make sure versions of PluGeth and plugins are compatable see: *version control*.

After downloading plugins, move the `.so` files into the `~/.ethereum/plugins` directory.

Note: The above location may change when changing `--datadir`.

CHAPTER

FOUR

BUILD AND DEPLOY

4.1 Setting up the environment

Note: PluGeth is built on a fork of Geth and as such requires familiarity with Go and a funtional environment in which to build Go projects. Thankfully for everyone Go provides a compact and useful tutorial as well as a space for practice.

PluGeth is an application built in three seperate repositories.

- PluGeth
- PluGeth-Utils
- PluGeth-Plugins

For the purposes here you will only need to clone PluGeth and PluGeth-Plugins. Once you have them cloned you are ready to begin. First we need to build PluGeth though the PluGeth project. Navigate to `plugeth/cmd/geth` and run:

```
$ go get
```

This will download all dependencies needed for the project. This process will take a moment or two the first time through. Next run:

```
$ go build
```

At this point you are ready to start downloading local ethereum nodes. In order to do so, from `plugeth/cmd/geth` run:

```
$ ./geth
```

4.2 Build your first plugin

For the sake of this tutorial we will be building the Hello plugin. Navigate to `plugethPlugins/packages/hello`. Inside you will see a `main.go` file. From this location run:

```
$ go build -buildmode=plugin
```

This will compile the plugin and produce a `hello.so` file. Move `hello.so` into `~/.ethereum/plugins` . In order to use this plugin geth will need to be started with a `http.api=mynamespace` flag. Additionally you will need to include a `--http` flag in order to access the standard json rpc methods.

Note: The above location may change when changing `--datadir`.

Once geth has started you should see that the first `INFO` log reads: `initialized hello` . A new json rpc method, called hello, has been been appended to the list of available json rpc methods. In order to access this method you will need to `curl` into the network with this command:

```
$ curl 127.0.0.1:8545 -H "Content-Type: application/json" --data '{"jsonrpc":"2.0",
↪"method":"mynamespace_hello","params":[],"id":0}'
```

You should see that the network has responded with:

```
``{"jsonrpc":"2.0","id":0,"result":"Hello world"}``
```

You have just built and run your first Plugeth plugin. From here you can follow the steps above to build any of the plugins you choose.

Note: Each plugin will vary in terms of the requirements to deploy. Refer to the documentation of the plugin itself in order to assure that you know how to use it.

CHAPTER

FIVE

WRITING A CUSTOM PLUGIN

5.1 RPC Methods

5.1.1 GetAPIs

For **RPC Methods** a Get APIs method is required in the body of the plugin in order to make the plugin available. The bulk of the implementation will be in the MyService struct. MyService should be a struct which includes two public functions.

```
type MyService struct {
    backend core.Backend
    stack   core.Node
}

func GetAPIs(stack core.Node, backend core.Backend) []core.API {
  return []core.API{
   {
      Namespace: "plugeth",
      Version:   "1.0",
      Service:   &MyService{backend, stack},
      Public:    true,
   },
 }
}
```

5.1.2 RPC Method

(**accurate heading?**)

For RPC calls, a function should have a `context.Context` object as the first argument, followed by an arbitrary number of JSON marshallable arguments, and return either a single JSON marshal object, or a JSON marshallable object and an error. The RPC framework will take care of decoding inputs to this function and encoding outputs, and if the error is non-nil it will serve an error response.

A simple implimentation would look like so:

eventual link to documentation for hello or some other rpc plugin

```
func (h *MyService) HelloWorld(ctx context.Context) string {
  return "Hello World"
}
```

Note: For plugins such as RPC Methods whcih impliment a GetAPIs function, an **Initialize Node** function may not be necesary as the `core.Node` and `core.Backend` will be made available with GetAPIs.

5.1.3 Access

As with pre-built plugins, a``.so`` will need to be built from``main.go`` and moved into `~/.ethereum/plugins`. Geth will need to be started with with a `http.api=mynamespace` flag. Additionally you will need to include a `--http` flag in order to access the standard json rpc methods.

The plugin can now be accessed with an rpc call to `mynamespace_helloWorld`.

5.2 Subscription

In addition to the initial template containing an intialize function, plugins providing **Subscriptions** will require two additional elements.

5.2.1 GetAPIs

A GetAPIs method is required in the body of the plugin in order to make the plugin available. The bulk of the implementation will be in the MyService struct. MyService should be a struct which includes two public functions.

```
type MyService struct {
    backend core.Backend
    stack   core.Node
}

func GetAPIs(stack core.Node, backend core.Backend) []core.API {
  return []core.API{
   {
     Namespace: "plugeth",
     Version:   "1.0",
     Service:   &MyService{backend, stack},
     Public:    true,
   },
 }
}
```

5.2.2 Subscription Function

For subscriptions (supported on IPC and websockets), a function should take MyService as a reciever and a context.Context object as an argument and return a channel and an error. The following is a subscription function that implements a timer.

```
func (*myservice) Timer(ctx context.Context) (<-chan int64, error) {
        ticker := time.NewTicker(time.Second)
        ch := make(chan int64)
        go func() {
```

(continues on next page)

(continued from previous page)

```
				defer ticker.Stop()
				for {
						select {
						case <-ctx.Done():
								close(ch)
								return
						case t := <-ticker.C:
								ch <- t.UnixNano()
						}
				}
		}()
		return ch, nil
}
```

Warning: Notice in the example above, the `ctx.Done()` or Context.Done() method closes the channel. If this is not present the go routine will run for the life of the process.

5.2.3 Access

Note: Plugins providing subscriptions can be accessed via IPC and websockets. In the below example we will be using wscat to connect a websocket to a local Geth node.

As with pre-built plugins, a `.so` will need to be built from `main.go` and moved into `~/.ethereum/plugins`. Geth will need to be started with `--ws --ws.api=mynamespace` flags. Additionally you will need to include a `--http` flag in order to access the standard json rpc methods.

After starting Geth, from a seperate terminal run:

```
wscat -c ws://127.0.0.1:8546
```

Note: Websockets are available via port 8546

Once the connection has been established from the websocket cursor enter the following argument:

```
{"jsonrpc":"2.0","method":"mynamespace_hello","params":[],"id":0}

You should see that the network has responded with:
```

```
``{"jsonrpc":"2.0","id":0,"result":"Hello world"}``
```

5.3 Tracer

In addition to the initial template containing an intialize function, plugins providing **Tracers** will require three additional elements.

Warning: Caution: Modifying of the values passed into tracer functions can alter the results of the EVM execution in unpredictable ways. Additionally, some objects may be reused across calls, so data you wish to capture should be copied rather than retained be reference.

5.3.1 MyService Struct

First an empty MyService Struct.

```
type MyService struct {
}
```

5.3.2 Map

Next, a map of tracers to functions returning a `core.TracerResult` which will be implemented like so:

```
var Tracers = map[string]func(core.StateDB) core.TracerResult{
    "myTracer": func(core.StateDB) core.TracerResult {
        return &MyBasicTracerService{}
    },
}
```

5.3.3 TracerResult Functions

Finally a series of functions which points to the MyService struct and coresponds to the interface which geth anticipates.

```
func (b *MyBasicTracerService) CaptureStart(from core.Address, to core.Address, create␣
↪bool, input []byte, gas uint64, value *big.Int) {
}
func (b *MyBasicTracerService) CaptureState(pc uint64, op core.OpCode, gas, cost uint64,␣
↪scope core.ScopeContext, rData []byte, depth int, err error) {
}
func (b *MyBasicTracerService) CaptureFault(pc uint64, op core.OpCode, gas, cost uint64,␣
↪scope core.ScopeContext, depth int, err error) {
}
func (b *MyBasicTracerService) CaptureEnd(output []byte, gasUsed uint64, t time.Duration,
↪ err error) {
}
func (b *MyBasicTracerService) CaptureEnter(typ core.OpCode, from core.Address, to core.
↪Address, input []byte, gas uint64, value *big.Int) {
```

(continues on next page)

(continued from previous page)

```
}
func (b *MyBasicTracerService) CaptureExit(output []byte, gasUsed uint64, err error) {
}
func (b *MyBasicTracerService) Result() (interface{}, error) { return "hello world", nil␣
↪}
```

5.3.4 Access

As with pre-built plugins, a `.so` will need to be built from `main.go` and moved into `~/.ethereum/plugins`. Geth will need to be started with with a `--http.api+debug` flag.

From a terminal pass the following argument to the api:

```
curl 127.0.0.1:8545 -H "Content-Type: application/json" --data '{"jsonrpc":"2.0","method
↪":"debug_traceCall","params":[{"to":"0x32Be343B94f860124dC4fEe278FDCBD38C102D88"},
↪"latest",{"tracer":"myTracer"}],"id":0}'
```

Note: The address used above is a test adress and will need to be replaced by whatever address you wish to access. Also `traceCall` is one of several methods available for use.

If using the template above, the call should return:

```
{"jsonrpc":"2.0","id":0,"result":"hello world"}
```

Before setting out to write a plugin it will be helpful to be familiar with the *Basic Types of Plugins*. Different plugins will require different implementation.

5.4 Basic Implementation

All plugins will share some common aspects.

5.4.1 Package

Any plugin will need its own package located in `plugeth-plugins/packages`. The package will need to include a `main.go` from which the `.so` file will be built. The package and main file should share the same name and the name should be a word that describes the basic functionality of the plugin.

5.4.2 Initialize

Most plugins will need to be initialized with a `func Initialize`. The initialize function will need to be passed at least three arguments: `cli.Context`, `core.PluginLoader`, `core.Logger`.

All plugins could have an intial template that looks something like this:

```
package main

import (
```

(continues on next page)

(continued from previous page)

```
        "github.com/openrelayxyz/plugeth-utils/core"
        "gopkg.in/urfave/cli.v1"
)

var log core.Logger

func Initialize(ctx *cli.Context, loader core.PluginLoader, logger core.Logger) {
        log = logger
        log.Info("loaded New Custom Plugin")
}
```

5.4.3 InitializeNode

Many plugins will make use of `func InitializeNode`.

```
func InitializeNode(stack core.Node, b core.Backend) {
        backend = b
        log.Info("Initialized node and backend")
}
```

`InitializeNode` is called as soon as the Geth node is initialized. The `core.Node` object represents the running node with P2P and RPC capabilities, while the `core.Backend` gives you access to blocks and other data you may need to access.

5.5 Specialization

From this point implimentation becomes more specialized to the particular plugin type. Continue from here for specific instructions for the following plugins:

- *RPC Methods*
- *Subscription*
- *Tracer*

CHAPTER

SIX

EXISTING PLUINGS

6.1 getRPCCalls

getRPCCalls is a subcommand written to print a log containing information about RPC methods upon execution. Namely the id, method name, and parameters into the method.

6.1.1 Usage

Once compiled the plugin will execute automatically as RPC methods are passed into the api.

6.2 isSynced

The isSynced plugin was designed as an extention of the `eth_syncing` method available on standard Geth. `plugeth_isSynced` was desinged to return a status object such that a status report could be given as to the current state of the node as opposed to `eth_syncing` which returns the status object only if the node is actively syncing and a simple false if frozen or fully synced.

6.2.1 Usage

As with all `rpc` methods, isSynced is available by `curl` or the javascript console.

From the command line using the `curl` command:

`{"method":  "plugeth_isSynced", "params":  []}`

Which will return:

```
"activePeers": true,
"currentBlock": "0x60e880",
"healedBytecodeBytes": "0x0",
"healedBytecodes": "0x0",
"healedTrienodeBytes": "0x0",
"healedTrienodes": "0x0",
"healingBytecode": "0x0",
"healingTrienodes": "0x0",
"highestBlock": "0x60e880",
"nodeIsSynced": true,
"startingBlock": "0x0",
"syncedAccountBytes": "0x0",
```

(continues on next page)

(continued from previous page)

```
"syncedAccounts": "0x0",
"syncedBytecodeBytes": "0x0",
"syncedBytecodes": "0x0",
"syncedStorage": "0x0",
"syncedStorageBytes": "0x0"
```

6.3 blockTracer

Blocktracer is an subscription plugin written such that for each block mined, blockTracer will return a json payload reporting the type, from and to addresses, gas, gas used, input, output, and calls made for each transaction. The data will stream in real time as the block is mined.

6.4 Usage

As with any websocket an initial connection will need to be established.

Here we are using wscat to connect to local host port 8556.

`wscat -c "http://127.0.0.1:8556"`

Once the connection has been made the method as well as blockTracer parameter will be passed in.

`{"method":"plugeth_subscribe","params":["traceBlock"],"id":0}`

Which will return a streaming result similar to the one below.

```
"type":"CALL","from":"0x75d5e88adf8f3597c7c3e4a930544fb48089c779","to":
↪"0x9ac40b4e6a0c60ca54a7fa2753d65448e6a71ecb","gas":"0x58cc2","gasUsed":"0x6007","input
↪":
↪"0x000000000000000000000000000000000000000000000000000000000000000000000000000000000000000028d2f41e4c
↪","output":"0x0000000000000000000000000000000000000000000000000000000000000000","calls
↪":[{"type":"DELEGATECALL","from":"0x9ac40b4e6a0c60ca54a7fa2753d65448e6a71ecb","to":
↪"0xae9a8ae28d55325dff2af4ed5fe2335c1a39139b","gas":"0x56308","gasUsed":"0x4c07","input
↪":
↪"0x0000000000000000000000000000000000000000000000000000000000000000abbfb9d900000000000000000000000000
↪","output":"0x0000000000000000000000000000000000000000000000000000000000000000"}]}]}]}}
```

CHAPTER SEVEN

SYSTEM REQUIREMENTS

If compiling from source code Plugeth requires that the user have Go version 1.17 or later.

Our provided release binaries require Glibc 2.29.

Note: System requirements vary depending on the network being run. For detailed requirements Refer to Geth documentation.

Warning: PluGeth relies on Golang's Plugin implementation, which is only supported on Linux, FreeBSD, and macOS. Windows support is unlikely to be added in the foreseeable future.

CHAPTER

EIGHT

VERSION CONTROL

PluGeth is separated into three packages in order to minimize dependency conflicts. Golang plugins cannot include different versions of the same packages as the program loading the plugin. If plugins had to import packages from PluGeth itself, a plugin build could only be loaded by that same version of PluGeth. By separating out the PluGeth-utils package, both PluGeth and the plugins must rely on the same version of PluGeth-utils, but plugins can be compatible with any version of PluGeth compiled with the same version of PluGeth-utils.

PluGeth builds will follow the naming convention:

```
geth-$PLUGETH_UTILS_VERSION-$GETH_VERSION-$RELEASE
```

For example:

```
geth-0.1.0-1.10.8-0
```

Tells us that:

- PluGeth-utils version is 0.1.0
- Geth version is 1.10.8
- This is the first release with that combination of dependencies.

Plugin builds will follow the naming convention:

```
$PLUGIN_NAME-$PLUGETH_UTILS_VERSION-$PLUGIN_VERSION
```

For example:

```
blockupdates-0.1.0-1.0.2
```

Tells us that:

- The plugin is “blockupdates”
- The PluGeth-utils version is 0.1.0
- The plugin version is 1.0.2

When a Geth update comes out, you can expect a release of *geth-0.1.0-1.10.9-0*, which will be compatible with the same set of plugins.

When PluGeth upgrades are necessary, plugins will need to be recompiled. Whenever possible, we will try to avoid forcing plugins to be recompiled for an immediate Geth upgrade. For example, if we have geth-0.1.0-1.10.8, and upgrade PluGeth-utils, we will have a geth-0.1.1-1.10.8, followed by a geth-0.1.1-1.10.9. This will give users time to upgrade plugins from PluGeth-utils 0.1.0 to 0.1.1 while staying on Geth 1.10.8, and when it is time to upgrade to Geth 1.10.9 they can continue using the plugins they were using with geth 1.10.8. Depending on upgrades to Geth, it may not always be possible to maintain compatibility with existing PluGeth versions, which will be noted in release notes.

CHAPTER

NINE

API

9.1 Anatomy of a Plugin

Plugins for Plugeth use Golang's Native Plugin System. Plugin modules must export variables using specific names and types. These will be processed by the plugin loader, and invoked at certain points during Geth's operations.

9.1.1 Flags

- **Name:** Flags
- **Type:** flag.FlagSet
- **Behavior:** This FlagSet will be parsed and your plugin will be able to access the resulting flags. Flags will be passed to Geth from the command line and are intended to configure the behavior of the plugin. Passed flags must follow `--` to be parsed by this FlagSet, which is necessary to avoid Geth failing due to unexpected flags.

9.1.2 Subcommands

- **Name:** Subcommands
- **Type:** map[string]func(ctx *cli.Context, args []string) error
- **Behavior:** If Geth is invoked with `./geth YOUR_COMMAND`, the plugin loader will look for `YOUR_COMMAND` within this map, and invoke the corresponding function. This can be useful for certain behaviors like manipulating Geth's database without having to build a separate binary.

9.1.3 Initialize

- **Name:** Initialize
- **Type:** func(*cli.Context, core.PluginLoader, core.logs)
- **Behavior:** Called as soon as the plugin is loaded, with the cli context and a reference to the plugin loader. This is your plugin's opportunity to initialize required variables as needed. Note that using the context object you can check arguments, and optionally can manipulate arguments if needed for your plugin.

9.1.4 InitializeNode

- **Name:** InitializeNode
- **Type:** func(core.Node, core.Backend)
- **Behavior:** This is called as soon as the Geth node is initialized. The core.Node object represents the running node with p2p and RPC capabilities, while the Backend gives you access to a wide array of data you may need to access.

Note: If a particular plugin requires access to the node.Node object it can be obtained using the restricted package located in PluGeth-Utils.

9.1.5 GetAPIs

- **Name:** GetAPIs
- **Type:** func(core.Node, core.Backend) []rpc.API
- **Behavior:** This allows you to register new RPC methods to run within Geth.

The GetAPIs function itself will generally be fairly brief, and will looks something like this:

```
  ``func GetAPIs(stack *node.Node, backend core.Backend) []core.API {
  return []rpc.API{
   {
     Namespace: "mynamespace",
     Version:      "1.0",
     Service:      &MyService{backend},
     Public:              true,
   },
  }
}``
```

The bulk of the implementation will be in the `MyService` struct. MyService should be a struct with public functions. These functions can have two different types of signatures:

- RPC Calls: For straight RPC calls, a function should have a `context.Context` object as the first argument, followed by an arbitrary number of JSON marshallable arguments, and return either a single JSON marshal object, or a JSON marshallable object and an error. The RPC framework will take care of decoding inputs to this function and encoding outputs, and if the error is non-nil it will serve an error response.
- Subscriptions: For subscriptions (supported on IPC and websockets), a function should have a `context.Context` object as the first argument followed by an arbitrary number of JSON marshallable arguments, and should return an `*rpc.Subscription` object. The subscription object can be created with `rpcSub := notifier.CreateSubscription()`, and JSON marshallable data can be sent to the subscriber with `notifier.Notify(rpcSub.ID, b)`.

A very simple MyService might look like:

```
``type MyService struct{}

  func (h MyService) HelloWorld(ctx context.Context) string {
    return "Hello World"
  }``
```

And the client could access this with an rpc call to `mynamespace_helloworld`

9.2 Injected APIs

In addition to hooks that get invoked by Geth, several objects are injected that give you access to additional information.

9.2.1 Backend Object

The `core.Backend` object is injected by the `InitializeNode()` and `GetAPI()` functions. It offers the following functions:

Downloader

`Downloader() Downloader`

Returns a Downloader objects, which can provide Syncing status

SuggestGasTipCap

`SuggestGasTipCap(ctx context.Context) (*big.Int, error)`

Suggests a Gas tip for the current block.

ExtRPCEnabled

`ExtRPCEnabled() bool`

Returns whether RPC external RPC calls are enabled.

RPCGasCap

`RPCGasCap() uint64`

Returns the maximum Gas available to RPC Calls.

RPCTxFeeCap

`RPCTxFeeCap() float64`

Returns the maximum transaction fee for a transaction submitted via RPC.

UnprotectedAllowed

`UnprotectedAllowed() bool`

Returns whether or not unprotected transactions can be transmitted through this node via RPC.

SetHead

`SetHead(number uint64)`

Resets the head to the specified block number.

HeaderByNumber

`HeaderByNumber(ctx context.Context, number int64) ([]byte, error)`

Returns an RLP encoded block header for the specified block number.

The RLP encoded response can be decoded into a *plugeth-utils/restricted/types.Header* object.

HeaderByHash

`HeaderByHash(ctx context.Context, hash Hash) ([]byte, error)`

Returns an RLP encoded block header for the specified block hash.

The RLP encoded response can be decoded into a *plugeth-utils/restricted/types.Header* object.

CurrentHeader

`CurrentHeader() []byte`

Returns an RLP encoded block header for the current block.

The RLP encoded response can be decoded into a *plugeth-utils/restricted/types.Header* object.

CurrentBlock

`CurrentBlock() []byte`

Returns an RLP encoded full block for the current block.

The RLP encoded response can be decoded into a *plugeth-utils/restricted/types.Block* object.

BlockByNumber

`BlockByNumber(ctx context.Context, number int64) ([]byte, error)`

Returns an RLP encoded full block for the specified block number.

The RLP encoded response can be decoded into a *plugeth-utils/restricted/types.Block* object.

BlockByHash

`BlockByHash(ctx context.Context, hash Hash) ([]byte, error)`

Returns an RLP encoded full block for the specified block hash.

The RLP encoded response can be decoded into a *plugeth-utils/restricted/types.Block* object.

GetReceipts

`GetReceipts(ctx context.Context, hash Hash) ([]byte, error)`

Returns an JSON encoded list of receipts for the specified block hash.

The JSON encoded response can be decoded into a *plugeth-utils/restricted/types.Receipts* object.

GetTd

`GetTd(ctx context.Context, hash Hash) *big.Int`

Returns the total difficulty for the specified block hash.

SubscribeChainEvent

`SubscribeChainEvent(ch chan<- ChainEvent) Subscription`

Subscribes the provided channel to new chain events.

SubscribeChainHeadEvent

`SubscribeChainHeadEvent(ch chan<- ChainHeadEvent) Subscription`

Subscribes the provided channel to new chain head events.

SubscribeChainSideEvent

`SubscribeChainSideEvent(ch chan<- ChainSideEvent) Subscription`

Subscribes the provided channel to new chain side events.

SendTx

`SendTx(ctx context.Context, signedTx []byte) error`

Sends an RLP encoded, signed transaction to the network.

GetTransaction

```
GetTransaction(ctx context.Context, txHash Hash) ([]byte, Hash, uint64, uint64, error)
```

Returns an RLP encoded transaction at the specified hash, along with the hash and number of the included block, and the transaction's position within that block.

GetPoolTransactions

```
GetPoolTransactions() ([][]byte, error)
```

Returns a list of RLP encoded transactions found in the mempool

GetPoolTransaction

```
GetPoolTransaction(txHash Hash) []byte
```

Returns the RLP encoded transaction from the mempool at the specified hash.

GetPoolNonce

```
GetPoolNonce(ctx context.Context, addr Address) (uint64, error)
```

Returns the nonce of the last transaction for a given address, including transactions found in the mempool.

Stats

```
Stats() (pending int, queued int)
```

Returns the number of pending and queued transactions in the mempool.

TxPoolContent

```
TxPoolContent() (map[Address][][]byte, map[Address][][]byte)
```

Returns a map of addresses to the list of RLP encoded transactions pending in the mempool, and queued in the mempool.

SubscribeNewTxsEvent

```
SubscribeNewTxsEvent(chan<- NewTxsEvent) Subscription
```

Subscribe to a feed of new transactions added to the mempool.

GetLogs

`GetLogs(ctx context.Context, blockHash Hash) ([][]byte, error)`

Returns a list of RLP encoded logs found in the specified block.

SubscribeLogsEvent

`SubscribeLogsEvent(ch chan<- [][]byte) Subscription`

Subscribe to logs included in a confirmed block.

SubscribePendingLogsEvent

`SubscribePendingLogsEvent(ch chan<- [][]byte) Subscription`

Subscribe to logs from pending transactions.

SubscribeRemovedLogsEvent

`SubscribeRemovedLogsEvent(ch chan<- []byte) Subscription`

Subscribe to logs removed from the canonical chain in reorged blocks.

9.2.2 Node Object

The `core.Node` object is injected by the `InitializeNode()` and `GetAPI()` functions. It offers the following functions:

Server

`Server() Server`

The Server object provides access to `server.PeerCount()`, the number of peers connected to the node.

DataDir

`DataDir() string`

Returns the Ethereuem datadir.

InstanceDir

`InstanceDir() string`

Returns the instancedir used by the protocol stack.

IPCEndpoint

```
IPCEndpoint() string
```

The path of the IPC Endpoint for this node.

HTTPEndpoint

```
HTTPEndpoint() string
```

The url of the HTTP Endpoint for this node.

WSEndpoint

```
WSEndpoint() string
```

The url of the websockets Endpoint for this node.

ResolvePath

```
ResolvePath(x string) string
```

Resolves a path within the DataDir.

9.2.3 Logger

The Logger object is injected by the `Initialize()` function. It implements logging based on the interfaces of Log15.

CHAPTER

TEN

PLUGIN LOADER

The Plugin Loader is provided to each Plugin through the Initialize()`` function. It provides plugins with:

10.1 Lookup

`Lookup(name string, validate func(interface{}) bool) []interface{}`

Returns a list of values from plugins identified by `name`, which match the provided `validate` predicate. For example:

```
pl.Lookup("Version", func(item interface{}) bool {
  _, ok := item.(int)
  return ok
})
```

Would return a list of `int` objects named `Version` in any loaded plugins. This can enable Plugins to interact with each other, accessing values and functions implemented in other plugins.

10.2 GetFeed

`GetFeed() Feed`

Returns a new feed that the plugin can use for publish/subscribe models.

For example:

```
feed := pl.GetFeed()
go func() {
  ch := make(chan string)
  sub := feed.Subscribe(ch)
  for {
    select {
    case item := <-ch:
      // Do something with item
    case err := <sub.Err():
      log.Error("An error has occurred", "err", err)
      sub.Unsubscribe()
      close(ch)
      return
    }
  }
```

(continues on next page)

(continued from previous page)

```
}()

feed.Send("hello")
feed.Send("world")
```

Note that you can send any type through a feed, but the subscribed channel and sent objects must be of matching types.

CHAPTER

ELEVEN

SELECTED PLUGIN HOOKS

11.1 Plugin Hooks

Plugeth provides several hooks from which the plugin can capture data from Geth. Additionally in the case of **subcommands** the provided hooks are designed to change the behavior of Geth.

Hooks are called from functions within the plugin. For example, if we wanted to bring in data from the StateUpdate hook. We would impliment it like so: (from blockupdates)

```
func StateUpdate(blockRoot core.Hash, parentRoot core.Hash, destructs map[core.
↪Hash]struct{}, accounts map[core.Hash][]byte, storage map[core.Hash]map[core.
↪Hash][]byte, codeUpdates map[core.Hash][]byte) {
	su := &stateUpdate{
		Destructs: destructs,
		Accounts: accounts,
		Storage: storage,
		Code: codeUpdates,
	}
	cache.Add(blockRoot, su)
	data, _ := rlp.EncodeToBytes(su)
	backend.ChainDb().Put(append([]byte("su"), blockRoot.Bytes()...), data)
}
```

Many hooks can be deployed in one plugin as is the case with the **BlockUpdater** plugin.

11.2 StateUpdate

Function Signature:`func(root common.Hash, parentRoot common.Hash, destructs map[common.Hash]struct{}, accounts map[common.Hash][]byte, storage map[common.Hash]map[common.Hash][]byte)`

The state update plugin provides a snapshot of the state subsystem in the form of a a stateUpdate object. The stateUpdate object contains all information transformed by a transaction but not the transaction itself.

Invoked for each new block, StateUpdate provides the changes to the blockchain state. root corresponds to the state root of the new block. parentRoot corresponds to the state root of the parent block. destructs serves as a set of accounts that self-destructed in this block. accounts maps the hash of each account address to the SlimRLP encoding of the account data. storage maps the hash of each account to a map of that account's stored data.

Warning: StateUpdate is only called if Geth is running with `-snapshots=true`. This is the default behavior for Geth, but if you are explicitly running with `--snapshot=false` this function will not be invoked.

11.3 AppendAncient

Function Signature:`func(number uint64, hash, header, body, receipts, td []byte)`

Invoked when the freezer moves a block from LevelDB to the ancients database. `number` is the number of the block. `hash` is the 32 byte hash of the block as a raw `[]byte`. `header`, `body`, and `receipts` are the RLP encoded versions of their respective block elements. `td` is the byte encoded total difficulty of the block.

11.4 GetRPCCalls

Function Signature:`func(string, string, string)`

Invoked when the RPC handler registers a method call. Returns the call `id`, method `name`, and any `params` that may have been passed in.

Todo: missing a couple of hooks

11.5 PreProcessBlock

Function Signature:`func(*types.Block)`

Invoked before the transactions of a block are processed. Returns a block object.

11.6 PreProcessTransaction

Function Signature:`func(*types.Transaction, *types.Block, int)`

Invoked before each individual transaction of a block is processed. Returns a transaction, block, and index number.

11.7 BlockProcessingError

Function Signature:`func(*types.Transaction, *types.Block, error)`

Invoked if an error occurs while processing a transaction. This only applies to errors that would unvalidate the block were this transaction is included not errors such as reverts or opcode errors. Returns a transaction, block, and error.

11.8 NewHead

Function Signature:`func(*types.Block, common.Hash, []*types.Log)`

Invoked when a new block becomes the canonical latest block. Returns a block, hash, and logs.

Note: If several blocks are processed in a group (such as during a reorg) this may not be called for each block. You should track the prior latest head if you need to process intermediate blocks.

11.9 NewSideBlock

Function Signature:`func(*types.Block, common.Hash, []*types.Log)`

Invoked when a block is side-chained. Returns a block, hash, and logs.

Note: Blocks passed to this method are non-canonical blocks.

11.10 Reorg

Function Signature:`func(common *types.Block, oldChain, newChain types.Blocks)`

Invoked when a chain reorg occurs, that is; at least one block is removed and one block is added. (`oldChain` is a list of removed blocks, `newChain` is a list of newliy added blocks, and `common` is the latest block that is an ancestor to both oldChain and newChain.) Returns a block, a list of old blocks, and a list of new blocks.

CHAPTER

TWELVE

HOOK WRITING GUIDE

If you're trying to interact with Geth in a way not already supported by PluGeth, we're happy to accept pull requests adding new hooks so long as they comply with certain standards. We strongly encourage you to *contact us* first. We may have suggestions on how to do what you're trying to do without adding new hooks, or easier ways to implement hooks to get the information you need.

Warning: Plugin hooks *must not* require plugins to import any packages from `github.com/ethereum/go-ethereum`. Doing so means that plugins must be recompiled for each version of Geth. Many types have been re-implemented in `github.com/openrelayxyz/plugeth-utils`. If you need a type for your hook not already provided by plugeth-utils, you may make a pull request to that project as well.

When extending the plugin API, a primary concern is leaving a minimal footprint in the core Geth codebase to avoid future merge conflicts. To achieve this, when we want to add a hook within some existing Geth code, we create a plugin_hooks.go in the same package. For example, in the core/rawdb package we have:

```go
// This file is part of the package we are adding hooks to
package rawdb

// Import whatever is necessary
import (
  "github.com/ethereum/go-ethereum/plugins"
  "github.com/ethereum/go-ethereum/log"
)


// PluginAppendAncient is the public plugin hook function, available for testing
func PluginAppendAncient(pl *plugins.PluginLoader, number uint64, hash, header, body,␣
↪receipts, td []byte) {
  fnList := pl.Lookup("AppendAncient", func(item interface{}) bool {
    _, ok := item.(func(number uint64, hash, header, body, receipts, td []byte))
    return ok
  })
  for _, fni := range fnList {
    if fn, ok := fni.(func(number uint64, hash, header, body, receipts, td []byte)); ok {
      fn(number, hash, header, body, receipts, td)
    }
  }
}

// pluginAppendAncient is the private plugin hook function
```

(continues on next page)

(continued from previous page)

```
func pluginAppendAncient(number uint64, hash, header, body, receipts, td []byte) {
  if plugins.DefaultPluginLoader == nil {
          log.Warn("Attempting AppendAncient, but default PluginLoader has not been␣
↪initialized")
    return
  }
  PluginAppendAncient(plugins.DefaultPluginLoader, number, hash, header, body, receipts,␣
↪td)
}
```

12.1 The Public Plugin Hook Function

The public plugin hook function should follow the naming convention Plugin$HookName. The first argument should be a `*plugins.PluginLoader`, followed by any arguments required by the functions to be provided by nay plugins implementing this hook.

The plugin hook function should use `PluginLoader.Lookup("$HookName", func(item interface{}) bool` to get a list of the plugin-provided functions to be invoked. The provided function should verify that the provided function implements the expected interface. After the first time a given hook is looked up through the plugin loader, the PluginLoader will cache references to those hooks.

Given the function list provided by the plugin loader, the public plugin hook function should iterate over the list, cast the elements to the appropriate type, and call the function with the provided arguments.

Unless there is a clear justification to the contrary, the function should be called in the current goroutine. Plugins may choose to spawn off a separate goroutine as appropriate, but for the sake of thread safety we should generally not assume that plugins will be implemented in a threadsafe manner. If a plugin degrades the performance of Geth significantly, that will generally be obvious, and plugin authors can take appropriate measures to improve performance. If a plugin introduces thread safety issues, those can go unnoticed during testing.

12.2 The Private Plugin Hook Function

The private plugin hook function should bear the same name as the public plugin hook function, but with a lower case first letter. The signature should match the public plugin hook function, except that the first argument referencing the PluginLoader should be removed. It should invoke the public plugin hook function on `plugins.DefaultPluginLoader`. It should always verify that the DefaultPluginLoader is non-nil, log warning and return if the DefaultPluginLoader has not been initialized.

12.3 In-Line Invocation

Within the Geth codebase, the private plugin hook function should be invoked with the appropriate arguments in a single line, to minimize unexpected conflicts merging the upstream geth codebase into plugeth.

CHAPTER

THIRTEEN

PLUGETH-UTILS SUBPACKAGES

PluGeth-utils is separated into two main packages: core, and restricted.

The *core* package has been implemented by the Rivet team, and is licensed under the MIT license, allowing it to be used in open source and closed source plugins alike. Nearly all plugins will need to import plugeth-utils/core in order to

The *restricted* package copies code from the go-ethereum project, which means it must be licensed under the LGPL license. If you import plugeth-utils/restricted, you must be sure that your plugin complies with requirements of linking to LGPL code, which will usually require making your source code available to anyone you distribute the plugin to.

CHAPTER FOURTEEN

GET IN TOUCH WITH US

We want to hear from you! The best way to reach the PluGeth team is through our Discord server. Drop in, say hello, we are anxious to hear your ideas and help work through any problems you may be having.

Join our Discord