F:\git\java\mar3\filemonitor\target\go-ethereum\go-ethereum-2.doc

0:F:\git\coin\ethereum\go-ethereum\core\state_transition.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package core

import (
"errors"
"fmt"
"math/big"

"github.com/ethereum/go-ethereum/common"
"github.com/ethereum/go-ethereum/common/math"
"github.com/ethereum/go-ethereum/core/vm"
"github.com/ethereum/go-ethereum/log"
"github.com/ethereum/go-ethereum/params"
)

var (
Big0   = big.NewInt(0)
errInsufficientBalanceForGas = errors.New("insufficient balance to pay for gas")
)

/*
The State Transitioning Model

A state transition is a change made when a transaction is applied to the current world state
The state transitioning model does all all the necessary work to work out a valid new state root.

1) Nonce handling
2) Pre pay gas
3) Create a new state object if the recipient is \0*32
4) Value transfer
== If contract creation ==
  4a) Attempt to run transaction data
  4b) If valid, use result as code for the new state object
== end ==
5) Run Script section
6) Derive new state root
*/
type StateTransition struct {

```
gp         *GasPool
msg         Message
gas        uint64
gasPrice   *big.Int
initialGas *big.Int
value      *big.Int
data       []byte
state      vm.StateDB

evm *vm.EVM
}

// Message represents a message sent to a contract.
type Message interface {
From() common.Address
//FromFrontier() (common.Address, error)
To() *common.Address

GasPrice() *big.Int
Gas() *big.Int
Value() *big.Int

Nonce() uint64
CheckNonce() bool
Data() []byte
}

// IntrinsicGas computes the 'intrinsic gas' for a message
// with the given data.
//
// TODO convert to uint64
func IntrinsicGas(data []byte, contractCreation, homestead bool) *big.Int {
igas := new(big.Int)
if contractCreation && homestead {
igas.SetUint64(params.TxGasContractCreation)
} else {
igas.SetUint64(params.TxGas)
}
if len(data) > 0 {
var nz int64
for _, byt := range data {
if byt != 0 {
```

```
nz++
}
}
m := big.NewInt(nz)
m.Mul(m, new(big.Int).SetUint64(params.TxDataNonZeroGas))
igas.Add(igas, m)
m.SetInt64(int64(len(data)) - nz)
m.Mul(m, new(big.Int).SetUint64(params.TxDataZeroGas))
igas.Add(igas, m)
}
return igas
}

// NewStateTransition initialises and returns a new state transition object.
func NewStateTransition(evm *vm.EVM, msg Message, gp *GasPool) *StateTransition {
return &StateTransition{
gp:         gp,
evm:        evm,
msg:        msg,
gasPrice:   msg.GasPrice(),
initialGas: new(big.Int),
value:      msg.Value(),
data:       msg.Data(),
state:      evm.StateDB,
}
}

// ApplyMessage computes the new state by applying the given message
// against the old state within the environment.
//
// ApplyMessage returns the bytes returned by any EVM execution (if it took place),
// the gas used (which includes gas refunds) and an error if it failed. An error always
// indicates a core error meaning that the message would always fail for that particular
// state and would never be accepted within a block.
func ApplyMessage(evm *vm.EVM, msg Message, gp *GasPool) ([]byte, *big.Int, error) {
st := NewStateTransition(evm, msg, gp)

ret, _, gasUsed, err := st.TransitionDb()
return ret, gasUsed, err
}

func (st *StateTransition) from() vm.AccountRef {
```

```
f := st.msg.From()
if !st.state.Exist(f) {
st.state.CreateAccount(f)
}
return vm.AccountRef(f)
}

func (st *StateTransition) to() vm.AccountRef {
if st.msg == nil {
return vm.AccountRef{}
}
to := st.msg.To()
if to == nil {
return vm.AccountRef{} // contract creation
}

reference := vm.AccountRef(*to)
if !st.state.Exist(*to) {
st.state.CreateAccount(*to)
}
return reference
}

func (st *StateTransition) useGas(amount uint64) error {
if st.gas < amount {
return vm.ErrOutOfGas
}
st.gas -= amount

return nil
}

func (st *StateTransition) buyGas() error {
mgas := st.msg.Gas()
if mgas.BitLen() > 64 {
return vm.ErrOutOfGas
}

mgval := new(big.Int).Mul(mgas, st.gasPrice)

var (
state  = st.state
```

```
sender = st.from()
)
if state.GetBalance(sender.Address()).Cmp(mgval) < 0 {
return errInsufficientBalanceForGas
}
if err := st.gp.SubGas(mgas); err != nil {
return err
}
st.gas += mgas.Uint64()

st.initialGas.Set(mgas)
state.SubBalance(sender.Address(), mgval)
return nil
}

func (st *StateTransition) preCheck() error {
msg := st.msg
sender := st.from()

// Make sure this transaction's nonce is correct
if msg.CheckNonce() {
if n := st.state.GetNonce(sender.Address()); n != msg.Nonce() {
return fmt.Errorf("invalid nonce: have %d, expected %d", msg.Nonce(), n)
}
}
return st.buyGas()
}

// TransitionDb will transition the state by applying the current message and returning the result
// including the required gas for the operation as well as the used gas. It returns an error if it
// failed. An error indicates a consensus issue.
func (st *StateTransition) TransitionDb() (ret []byte, requiredGas, usedGas *big.Int, err error) {
if err = st.preCheck(); err != nil {
return
}
msg := st.msg
sender := st.from() // err checked in preCheck

homestead := st.evm.ChainConfig().IsHomestead(st.evm.BlockNumber)
contractCreation := msg.To() == nil

// Pay intrinsic gas
```

```
// TODO convert to uint64
intrinsicGas := IntrinsicGas(st.data, contractCreation, homestead)
if intrinsicGas.BitLen() > 64 {
return nil, nil, nil, vm.ErrOutOfGas
}
if err = st.useGas(intrinsicGas.Uint64()); err != nil {
return nil, nil, nil, err
}

var (
evm = st.evm
// vm errors do not effect consensus and are therefor
// not assigned to err, except for insufficient balance
// error.
vmerr error
)
if contractCreation {
ret, _, st.gas, vmerr = evm.Create(sender, st.data, st.gas, st.value)
} else {
// Increment the nonce for the next transaction
st.state.SetNonce(sender.Address(), st.state.GetNonce(sender.Address())+1)
ret, st.gas, vmerr = evm.Call(sender, st.to().Address(), st.data, st.gas, st.value)
}
if vmerr != nil {
log.Debug("VM returned with error", "err", vmerr)
// The only possible consensus-error would be if there wasn't
// sufficient balance to make the transfer happen. The first
// balance transfer may never fail.
if vmerr == vm.ErrInsufficientBalance {
return nil, nil, nil, vmerr
}
}
requiredGas = new(big.Int).Set(st.gasUsed())

st.refundGas()
st.state.AddBalance(st.evm.Coinbase, new(big.Int).Mul(st.gasUsed(), st.gasPrice))

return ret, requiredGas, st.gasUsed(), err
}

func (st *StateTransition) refundGas() {
// Return eth for remaining gas to the sender account,
```

```
// exchanged at the original rate.
sender := st.from() // err already checked
remaining := new(big.Int).Mul(new(big.Int).SetUint64(st.gas), st.gasPrice)
st.state.AddBalance(sender.Address(), remaining)

// Apply refund counter, capped to half of the used gas.
uhalf := remaining.Div(st.gasUsed(), common.Big2)
refund := math.BigMin(uhalf, st.state.GetRefund())
st.gas += refund.Uint64()

st.state.AddBalance(sender.Address(), refund.Mul(refund, st.gasPrice))

// Also return remaining gas to the block gas counter so it is
// available for the next transaction.
st.gp.AddGas(new(big.Int).SetUint64(st.gas))
}

func (st *StateTransition) gasUsed() *big.Int {
return new(big.Int).Sub(st.initialGas, new(big.Int).SetUint64(st.gas))
}

1:F:\git\coin\ethereum\go-ethereum\core\tx_list.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package core

import (
"container/heap"
"math"
"math/big"
"sort"

"github.com/ethereum/go-ethereum/common"
"github.com/ethereum/go-ethereum/core/types"
"github.com/ethereum/go-ethereum/log"
)

// nonceHeap is a heap.Interface implementation over 64bit unsigned integers for
// retrieving sorted transactions from the possibly gapped future queue.
type nonceHeap []uint64

func (h nonceHeap) Len() int           { return len(h) }
```

```
func (h nonceHeap) Less(i, j int) bool { return h[i] < h[j] }
func (h nonceHeap) Swap(i, j int)      { h[i], h[j] = h[j], h[i] }

func (h *nonceHeap) Push(x interface{}) {
*h = append(*h, x.(uint64))
}

func (h *nonceHeap) Pop() interface{} {
old := *h
n := len(old)
x := old[n-1]
*h = old[0 : n-1]
return x
}

// txSortedMap is a nonce->transaction hash map with a heap based index to allow
// iterating over the contents in a nonce-incrementing way.
type txSortedMap struct {
items map[uint64]*types.Transaction // Hash map storing the transaction data
index *nonceHeap                    // Heap of nonces of all the stored transactions (non-strict mode)
cache types.Transactions            // Cache of the transactions already sorted
}

// newTxSortedMap creates a new nonce-sorted transaction map.
func newTxSortedMap() *txSortedMap {
return &txSortedMap{
items: make(map[uint64]*types.Transaction),
index: new(nonceHeap),
}
}

// Get retrieves the current transactions associated with the given nonce.
func (m *txSortedMap) Get(nonce uint64) *types.Transaction {
return m.items[nonce]
}

// Put inserts a new transaction into the map, also updating the map's nonce
// index. If a transaction already exists with the same nonce, it's overwritten.
func (m *txSortedMap) Put(tx *types.Transaction) {
nonce := tx.Nonce()
if m.items[nonce] == nil {
heap.Push(m.index, nonce)
```

```
}
m.items[nonce], m.cache = tx, nil
}

// Forward removes all transactions from the map with a nonce lower than the
// provided threshold. Every removed transaction is returned for any post-removal
// maintenance.
func (m *txSortedMap) Forward(threshold uint64) types.Transactions {
var removed types.Transactions

// Pop off heap items until the threshold is reached
for m.index.Len() > 0 && (*m.index)[0] < threshold {
nonce := heap.Pop(m.index).(uint64)
removed = append(removed, m.items[nonce])
delete(m.items, nonce)
}
// If we had a cached order, shift the front
if m.cache != nil {
m.cache = m.cache[len(removed):]
}
return removed
}

// Filter iterates over the list of transactions and removes all of them for which
// the specified function evaluates to true.
func (m *txSortedMap) Filter(filter func(*types.Transaction) bool) types.Transactions {
var removed types.Transactions

// Collect all the transactions to filter out
for nonce, tx := range m.items {
if filter(tx) {
removed = append(removed, tx)
delete(m.items, nonce)
}
}
// If transactions were removed, the heap and cache are ruined
if len(removed) > 0 {
*m.index = make([]uint64, 0, len(m.items))
for nonce := range m.items {
*m.index = append(*m.index, nonce)
}
heap.Init(m.index)
```

```
m.cache = nil
}
return removed
}

// Cap places a hard limit on the number of items, returning all transactions
// exceeding that limit.
func (m *txSortedMap) Cap(threshold int) types.Transactions {
// Short circuit if the number of items is under the limit
if len(m.items) <= threshold {
return nil
}
// Otherwise gather and drop the highest nonce'd transactions
var drops types.Transactions

sort.Sort(*m.index)
for size := len(m.items); size > threshold; size-- {
drops = append(drops, m.items[(*m.index)[size-1]])
delete(m.items, (*m.index)[size-1])
}
*m.index = (*m.index)[:threshold]
heap.Init(m.index)

// If we had a cache, shift the back
if m.cache != nil {
m.cache = m.cache[:len(m.cache)-len(drops)]
}
return drops
}

// Remove deletes a transaction from the maintained map, returning whether the
// transaction was found.
func (m *txSortedMap) Remove(nonce uint64) bool {
// Short circuit if no transaction is present
_, ok := m.items[nonce]
if !ok {
return false
}
// Otherwise delete the transaction and fix the heap index
for i := 0; i < m.index.Len(); i++ {
if (*m.index)[i] == nonce {
```

```
heap.Remove(m.index, i)
break
}
}
delete(m.items, nonce)
m.cache = nil

return true
}

// Ready retrieves a sequentially increasing list of transactions starting at the
// provided nonce that is ready for processing. The returned transactions will be
// removed from the list.
//
// Note, all transactions with nonces lower than start will also be returned to
// prevent getting into and invalid state. This is not something that should ever
// happen but better to be self correcting than failing!
func (m *txSortedMap) Ready(start uint64) types.Transactions {
// Short circuit if no transactions are available
if m.index.Len() == 0 || (*m.index)[0] > start {
return nil
}
// Otherwise start accumulating incremental transactions
var ready types.Transactions
for next := (*m.index)[0]; m.index.Len() > 0 && (*m.index)[0] == next; next++ {
ready = append(ready, m.items[next])
delete(m.items, next)
heap.Pop(m.index)
}
m.cache = nil

return ready
}

// Len returns the length of the transaction map.
func (m *txSortedMap) Len() int {
return len(m.items)
}

// Flatten creates a nonce-sorted slice of transactions based on the loosely
// sorted internal representation. The result of the sorting is cached in case
// it's requested again before any modifications are made to the contents.
```

```
func (m *txSortedMap) Flatten() types.Transactions {
// If the sorting was not cached yet, create and cache it
if m.cache == nil {
m.cache = make(types.Transactions, 0, len(m.items))
for _, tx := range m.items {
m.cache = append(m.cache, tx)
}
sort.Sort(types.TxByNonce(m.cache))
}
// Copy the cache to prevent accidental modifications
txs := make(types.Transactions, len(m.cache))
copy(txs, m.cache)
return txs
}

// txList is a "list" of transactions belonging to an account, sorted by account
// nonce. The same type can be used both for storing contiguous transactions for
// the executable/pending queue; and for storing gapped transactions for the non-
// executable/future queue, with minor behavioral changes.
type txList struct {
strict bool         // Whether nonces are strictly continuous or not
txs    *txSortedMap // Heap indexed sorted hash map of the transactions

costcap *big.Int // Price of the highest costing transaction (reset only if exceeds balance)
gascap  *big.Int // Gas limit of the highest spending transaction (reset only if exceeds block limit)
}

// newTxList create a new transaction list for maintaining nonce-indexable fast,
// gapped, sortable transaction lists.
func newTxList(strict bool) *txList {
return &txList{
strict:  strict,
txs:     newTxSortedMap(),
costcap: new(big.Int),
gascap:  new(big.Int),
}
}

// Overlaps returns whether the transaction specified has the same nonce as one
// already contained within the list.
func (l *txList) Overlaps(tx *types.Transaction) bool {
return l.txs.Get(tx.Nonce()) != nil
```

```
}

// Add tries to insert a new transaction into the list, returning whether the
// transaction was accepted, and if yes, any previous transaction it replaced.
//
// If the new transaction is accepted into the list, the lists' cost and gas
// thresholds are also potentially updated.
func (l *txList) Add(tx *types.Transaction, priceBump uint64) (bool, *types.Transaction) {
// If there's an older better transaction, abort
old := l.txs.Get(tx.Nonce())
if old != nil {
threshold := new(big.Int).Div(new(big.Int).Mul(old.GasPrice(), big.NewInt(100+int64(priceBump))),
big.NewInt(100))
if threshold.Cmp(tx.GasPrice()) >= 0 {
return false, nil
}
}
// Otherwise overwrite the old transaction with the current one
l.txs.Put(tx)
if cost := tx.Cost(); l.costcap.Cmp(cost) < 0 {
l.costcap = cost
}
if gas := tx.Gas(); l.gascap.Cmp(gas) < 0 {
l.gascap = gas
}
return true, old
}

// Forward removes all transactions from the list with a nonce lower than the
// provided threshold. Every removed transaction is returned for any post-removal
// maintenance.
func (l *txList) Forward(threshold uint64) types.Transactions {
return l.txs.Forward(threshold)
}

// Filter removes all transactions from the list with a cost or gas limit higher
// than the provided thresholds. Every removed transaction is returned for any
// post-removal maintenance. Strict-mode invalidated transactions are also
// returned.
//
// This method uses the cached costcap and gascap to quickly decide if there's even
// a point in calculating all the costs or if the balance covers all. If the threshold
```

```
// is lower than the costgas cap, the caps will be reset to a new high after removing
// the newly invalidated transactions.
func (l *txList) Filter(costLimit, gasLimit *big.Int) (types.Transactions, types.Transactions) {
// If all transactions are below the threshold, short circuit
if l.costcap.Cmp(costLimit) <= 0 && l.gascap.Cmp(gasLimit) <= 0 {
return nil, nil
}
l.costcap = new(big.Int).Set(costLimit) // Lower the caps to the thresholds
l.gascap = new(big.Int).Set(gasLimit)

// Filter out all the transactions above the account's funds
removed := l.txs.Filter(func(tx *types.Transaction) bool { return tx.Cost().Cmp(costLimit) > 0 ||
tx.Gas().Cmp(gasLimit) > 0 })

// If the list was strict, filter anything above the lowest nonce
var invalids types.Transactions
if l.strict && len(removed) > 0 {
lowest := uint64(math.MaxUint64)
for _, tx := range removed {
if nonce := tx.Nonce(); lowest > nonce {
lowest = nonce
}
}
invalids = l.txs.Filter(func(tx *types.Transaction) bool { return tx.Nonce() > lowest })
}
return removed, invalids
}

// Cap places a hard limit on the number of items, returning all transactions
// exceeding that limit.
func (l *txList) Cap(threshold int) types.Transactions {
return l.txs.Cap(threshold)
}

// Remove deletes a transaction from the maintained list, returning whether the
// transaction was found, and also returning any transaction invalidated due to
// the deletion (strict mode only).
func (l *txList) Remove(tx *types.Transaction) (bool, types.Transactions) {
// Remove the transaction from the set
nonce := tx.Nonce()
if removed := l.txs.Remove(nonce); !removed {
return false, nil
```

```
}
// In strict mode, filter out non-executable transactions
if l.strict {
return true, l.txs.Filter(func(tx *types.Transaction) bool { return tx.Nonce() > nonce })
}
return true, nil
}

// Ready retrieves a sequentially increasing list of transactions starting at the
// provided nonce that is ready for processing. The returned transactions will be
// removed from the list.
//
// Note, all transactions with nonces lower than start will also be returned to
// prevent getting into and invalid state. This is not something that should ever
// happen but better to be self correcting than failing!
func (l *txList) Ready(start uint64) types.Transactions {
return l.txs.Ready(start)
}

// Len returns the length of the transaction list.
func (l *txList) Len() int {
return l.txs.Len()
}

// Empty returns whether the list of transactions is empty or not.
func (l *txList) Empty() bool {
return l.Len() == 0
}

// Flatten creates a nonce-sorted slice of transactions based on the loosely
// sorted internal representation. The result of the sorting is cached in case
// it's requested again before any modifications are made to the contents.
func (l *txList) Flatten() types.Transactions {
return l.txs.Flatten()
}

// priceHeap is a heap.Interface implementation over transactions for retrieving
// price-sorted transactions to discard when the pool fills up.
type priceHeap []*types.Transaction

func (h priceHeap) Len() int           { return len(h) }
func (h priceHeap) Less(i, j int) bool { return h[i].GasPrice().Cmp(h[j].GasPrice()) < 0 }
```

```
func (h priceHeap) Swap(i, j int)      { h[i], h[j] = h[j], h[i] }

func (h *priceHeap) Push(x interface{}) {
*h = append(*h, x.(*types.Transaction))
}

func (h *priceHeap) Pop() interface{} {
old := *h
n := len(old)
x := old[n-1]
*h = old[0 : n-1]
return x
}

// txPricedList is a price-sorted heap to allow operating on transactions pool
// contents in a price-incrementing way.
type txPricedList struct {
all    *map[common.Hash]*types.Transaction // Pointer to the map of all transactions
items  *priceHeap                         // Heap of prices of all the stored transactions
stales int                                // Number of stale price points to (re-heap trigger)
}

// newTxPricedList creates a new price-sorted transaction heap.
func newTxPricedList(all *map[common.Hash]*types.Transaction) *txPricedList {
return &txPricedList{
all:   all,
items: new(priceHeap),
}
}

// Put inserts a new transaction into the heap.
func (l *txPricedList) Put(tx *types.Transaction) {
heap.Push(l.items, tx)
}

// Removed notifies the prices transaction list that an old transaction dropped
// from the pool. The list will just keep a counter of stale objects and update
// the heap if a large enough ratio of transactions go stale.
func (l *txPricedList) Removed() {
// Bump the stale counter, but exit if still too low (< 25%)
l.stales++
if l.stales <= len(*l.items)/4 {
```

```
return
}
// Seems we've reached a critical number of stale transactions, reheap
reheap := make(priceHeap, 0, len(*l.all))

l.stales, l.items = 0, &reheap
for _, tx := range *l.all {
*l.items = append(*l.items, tx)
}
heap.Init(l.items)
}

// Discard finds all the transactions below the given price threshold, drops them
// from the priced list and returs them for further removal from the entire pool.
func (l *txPricedList) Cap(threshold *big.Int, local *txSet) types.Transactions {
drop := make(types.Transactions, 0, 128) // Remote underpriced transactions to drop
save := make(types.Transactions, 0, 64)  // Local underpriced transactions to keep

for len(*l.items) > 0 {
// Discard stale transactions if found during cleanup
tx := heap.Pop(l.items).(*types.Transaction)

hash := tx.Hash()
if _, ok := (*l.all)[hash]; !ok {
l.stales--
continue
}
// Stop the discards if we've reached the threshold
if tx.GasPrice().Cmp(threshold) >= 0 {
break
}
// Non stale transaction found, discard unless local
if local.contains(hash) {
save = append(save, tx)
} else {
drop = append(drop, tx)
}
}
for _, tx := range save {
heap.Push(l.items, tx)
}
return drop
```

```
}

// Underpriced checks whether a transaction is cheaper than (or as cheap as) the
// lowest priced transaction currently being tracked.
func (l *txPricedList) Underpriced(tx *types.Transaction, local *txSet) bool {
// Local transactions cannot be underpriced
if local.contains(tx.Hash()) {
return false
}
// Discard stale price points if found at the heap start
for len(*l.items) > 0 {
head := []*types.Transaction(*l.items)[0]
if _, ok := (*l.all)[head.Hash()]; !ok {
l.stales--
heap.Pop(l.items)
continue
}
break
}
// Check if the transaction is underpriced or not
if len(*l.items) == 0 {
log.Error("Pricing query for empty pool") // This cannot happen, print to catch programming errors
return false
}
cheapest := []*types.Transaction(*l.items)[0]
return cheapest.GasPrice().Cmp(tx.GasPrice()) >= 0
}

// Discard finds a number of most underpriced transactions, removes them from the
// priced list and returs them for further removal from the entire pool.
func (l *txPricedList) Discard(count int, local *txSet) types.Transactions {
drop := make(types.Transactions, 0, count) // Remote underpriced transactions to drop
save := make(types.Transactions, 0, 64)    // Local underpriced transactions to keep

for len(*l.items) > 0 && count > 0 {
// Discard stale transactions if found during cleanup
tx := heap.Pop(l.items).(*types.Transaction)

hash := tx.Hash()
if _, ok := (*l.all)[hash]; !ok {
l.stales--
continue
```

```
}
// Non stale transaction found, discard unless local
if local.contains(hash) {
save = append(save, tx)
} else {
drop = append(drop, tx)
count--
}
}
for _, tx := range save {
heap.Push(l.items, tx)
}
return drop
}

2:F:\git\coin\ethereum\go-ethereum\core\tx_list_test.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package core

import (
"math/big"
"math/rand"
"testing"

"github.com/ethereum/go-ethereum/core/types"
"github.com/ethereum/go-ethereum/crypto"
)

// Tests that transactions can be added to strict lists and list contents and
// nonce boundaries are correctly maintained.
func TestStrictTxListAdd(t *testing.T) {
// Generate a list of transactions to insert
key, _ := crypto.GenerateKey()

txs := make(types.Transactions, 1024)
for i := 0; i < len(txs); i++ {
txs[i] = transaction(uint64(i), new(big.Int), key)
}
// Insert the transactions in a random order
list := newTxList(true)
for _, v := range rand.Perm(len(txs)) {
```

```
list.Add(txs[v], DefaultTxPoolConfig.PriceBump)
}
// Verify internal state
if len(list.txs.items) != len(txs) {
t.Errorf("transaction count mismatch: have %d, want %d", len(list.txs.items), len(txs))
}
for i, tx := range txs {
if list.txs.items[tx.Nonce()] != tx {
t.Errorf("item %d: transaction mismatch: have %v, want %v", i, list.txs.items[tx.Nonce()], tx)
}
}
}
```

3:F:\git\coin\ethereum\go-ethereum\core\tx_pool.go

```
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package core

import (
"errors"
"fmt"
"math/big"
"sort"
"sync"
"time"

"github.com/ethereum/go-ethereum/common"
"github.com/ethereum/go-ethereum/core/state"
"github.com/ethereum/go-ethereum/core/types"
"github.com/ethereum/go-ethereum/event"
"github.com/ethereum/go-ethereum/log"
"github.com/ethereum/go-ethereum/metrics"
"github.com/ethereum/go-ethereum/params"
"gopkg.in/karalabe/cookiejar.v2/collections/prque"
)

var (
// ErrInvalidSender is returned if the transaction contains an invalid signature.
ErrInvalidSender = errors.New("invalid sender")

// ErrNonceTooLow is returned if the nonce of a transaction is lower than the
// one present in the local chain.
```

```
ErrNonceTooLow = errors.New("nonce too low")

// ErrUnderpriced is returned if a transaction's gas price is below the minimum
// configured for the transaction pool.
ErrUnderpriced = errors.New("transaction underpriced")

// ErrReplaceUnderpriced is returned if a transaction is attempted to be replaced
// with a different one without the required price bump.
ErrReplaceUnderpriced = errors.New("replacement transaction underpriced")

// ErrInsufficientFunds is returned if the total cost of executing a transaction
// is higher than the balance of the user's account.
ErrInsufficientFunds = errors.New("insufficient funds for gas * price + value")

// ErrIntrinsicGas is returned if the transaction is specified to use less gas
// than required to start the invocation.
ErrIntrinsicGas = errors.New("intrinsic gas too low")

// ErrGasLimit is returned if a transaction's requested gas limit exceeds the
// maximum allowance of the current block.
ErrGasLimit = errors.New("exceeds block gas limit")

// ErrNegativeValue is a sanity error to ensure noone is able to specify a
// transaction with a negative value.
ErrNegativeValue = errors.New("negative value")

// ErrOversizedData is returned if the input data of a transaction is greater
// than some meaningful limit a user might use. This is not a consensus error
// making the transaction invalid, rather a DOS protection.
ErrOversizedData = errors.New("oversized data")
)

var (
evictionInterval    = time.Minute     // Time interval to check for evictable transactions
statsReportInterval = 8 * time.Second // Time interval to report transaction pool stats
)

var (
// Metrics for the pending pool
pendingDiscardCounter   = metrics.NewCounter("txpool/pending/discard")
pendingReplaceCounter   = metrics.NewCounter("txpool/pending/replace")
pendingRateLimitCounter = metrics.NewCounter("txpool/pending/ratelimit") // Dropped due to rate
```

```
limiting
pendingNofundsCounter   = metrics.NewCounter("txpool/pending/nofunds")   // Dropped due to
out-of-funds

// Metrics for the queued pool
queuedDiscardCounter   = metrics.NewCounter("txpool/queued/discard")
queuedReplaceCounter   = metrics.NewCounter("txpool/queued/replace")
queuedRateLimitCounter = metrics.NewCounter("txpool/queued/ratelimit") // Dropped due to rate
limiting
queuedNofundsCounter   = metrics.NewCounter("txpool/queued/nofunds")   // Dropped due to out-
of-funds

// General tx metrics
invalidTxCounter     = metrics.NewCounter("txpool/invalid")
underpricedTxCounter = metrics.NewCounter("txpool/underpriced")
)

type stateFn func() (*state.StateDB, error)

// TxPoolConfig are the configuration parameters of the transaction pool.
type TxPoolConfig struct {
PriceLimit uint64 // Minimum gas price to enforce for acceptance into the pool
PriceBump  uint64 // Minimum price bump percentage to replace an already existing transaction
(nonce)

AccountSlots uint64 // Minimum number of executable transaction slots guaranteed per account
GlobalSlots  uint64 // Maximum number of executable transaction slots for all accounts
AccountQueue uint64 // Maximum number of non-executable transaction slots permitted per
account
GlobalQueue  uint64 // Maximum number of non-executable transaction slots for all accounts

Lifetime time.Duration // Maximum amount of time non-executable transaction are queued
}

// DefaultTxPoolConfig contains the default configurations for the transaction
// pool.
var DefaultTxPoolConfig = TxPoolConfig{
PriceLimit: 1,
PriceBump:  10,

AccountSlots: 16,
GlobalSlots:  4096,
```

```
AccountQueue: 64,
GlobalQueue:  1024,

Lifetime: 3 * time.Hour,
}

// sanitize checks the provided user configurations and changes anything that's
// unreasonable or unworkable.
func (config *TxPoolConfig) sanitize() TxPoolConfig {
conf := *config
if conf.PriceLimit < 1 {
log.Warn("Sanitizing invalid txpool price limit", "provided", conf.PriceLimit, "updated",
DefaultTxPoolConfig.PriceLimit)
conf.PriceLimit = DefaultTxPoolConfig.PriceLimit
}
if conf.PriceBump < 1 {
log.Warn("Sanitizing invalid txpool price bump", "provided", conf.PriceBump, "updated",
DefaultTxPoolConfig.PriceBump)
conf.PriceBump = DefaultTxPoolConfig.PriceBump
}
return conf
}

// TxPool contains all currently known transactions. Transactions
// enter the pool when they are received from the network or submitted
// locally. They exit the pool when they are included in the blockchain.
//
// The pool separates processable transactions (which can be applied to the
// current state) and future transactions. Transactions move between those
// two states over time as they are received and processed.
type TxPool struct {
config       TxPoolConfig
chainconfig  *params.ChainConfig
currentState stateFn // The state function which will allow us to do some pre checks
pendingState *state.ManagedState
gasLimit     func() *big.Int // The current gas limit function callback
gasPrice     *big.Int
eventMux     *event.TypeMux
events       *event.TypeMuxSubscription
locals       *txSet
signer       types.Signer
mu           sync.RWMutex
```

```
pending map[common.Address]*txList         // All currently processable transactions
queue   map[common.Address]*txList         // Queued but non-processable transactions
beats   map[common.Address]time.Time       // Last heartbeat from each known account
all     map[common.Hash]*types.Transaction // All transactions to allow lookups
priced  *txPricedList                      // All transactions sorted by price

wg   sync.WaitGroup // for shutdown sync
quit chan struct{}

homestead bool
}

// NewTxPool creates a new transaction pool to gather, sort and filter inbound
// trnsactions from the network.
func NewTxPool(config TxPoolConfig, chainconfig *params.ChainConfig, eventMux
*event.TypeMux, currentStateFn stateFn, gasLimitFn func() *big.Int) *TxPool {
// Sanitize the input to ensure no vulnerable gas prices are set
config = (&config).sanitize()

// Create the transaction pool with its initial settings
pool := &TxPool{
config:       config,
chainconfig:  chainconfig,
signer:       types.NewEIP155Signer(chainconfig.ChainId),
pending:      make(map[common.Address]*txList),
queue:        make(map[common.Address]*txList),
beats:        make(map[common.Address]time.Time),
all:          make(map[common.Hash]*types.Transaction),
eventMux:     eventMux,
currentState: currentStateFn,
gasLimit:     gasLimitFn,
gasPrice:     new(big.Int).SetUint64(config.PriceLimit),
pendingState: nil,
locals:       newTxSet(),
events:       eventMux.Subscribe(ChainHeadEvent{}, RemovedTransactionEvent{}),
quit:         make(chan struct{}),
}
pool.priced = newTxPricedList(&pool.all)
pool.resetState()

// Start the various events loops and return
```

```
pool.wg.Add(2)
go pool.eventLoop()
go pool.expirationLoop()

return pool
}

func (pool *TxPool) eventLoop() {
defer pool.wg.Done()

// Start a ticker and keep track of interesting pool stats to report
var prevPending, prevQueued, prevStales int

report := time.NewTicker(statsReportInterval)
defer report.Stop()

// Track chain events. When a chain events occurs (new chain canon block)
// we need to know the new state. The new state will help us determine
// the nonces in the managed state
for {
select {
// Handle any events fired by the system
case ev, ok := <-pool.events.Chan():
if !ok {
return
}
switch ev := ev.Data.(type) {
case ChainHeadEvent:
pool.mu.Lock()
if ev.Block != nil {
if pool.chainconfig.IsHomestead(ev.Block.Number()) {
pool.homestead = true
}
}
pool.resetState()
pool.mu.Unlock()

case RemovedTransactionEvent:
pool.AddBatch(ev.Txs)
}

// Handle stats reporting ticks
```

```
case <-report.C:
pool.mu.RLock()
pending, queued := pool.stats()
stales := pool.priced.stales
pool.mu.RUnlock()

if pending != prevPending || queued != prevQueued || stales != prevStales {
log.Debug("Transaction pool status report", "executable", pending, "queued", queued, "stales",
stales)
prevPending, prevQueued, prevStales = pending, queued, stales
}
}
}
}

func (pool *TxPool) resetState() {
currentState, err := pool.currentState()
if err != nil {
log.Error("Failed reset txpool state", "err", err)
return
}
pool.pendingState = state.ManageState(currentState)

// validate the pool of pending transactions, this will remove
// any transactions that have been included in the block or
// have been invalidated because of another transaction (e.g.
// higher gas price)
pool.demoteUnexecutables(currentState)

// Update all accounts to the latest known pending nonce
for addr, list := range pool.pending {
txs := list.Flatten() // Heavy but will be cached and is needed by the miner anyway
pool.pendingState.SetNonce(addr, txs[len(txs)-1].Nonce()+1)
}
// Check the queue and move transactions over to the pending if possible
// or remove those that have become invalid
pool.promoteExecutables(currentState, nil)
}

// Stop terminates the transaction pool.
func (pool *TxPool) Stop() {
pool.events.Unsubscribe()
```

```
close(pool.quit)
pool.wg.Wait()

log.Info("Transaction pool stopped")
}

// GasPrice returns the current gas price enforced by the transaction pool.
func (pool *TxPool) GasPrice() *big.Int {
pool.mu.RLock()
defer pool.mu.RUnlock()

return new(big.Int).Set(pool.gasPrice)
}

// SetGasPrice updates the minimum price required by the transaction pool for a
// new transaction, and drops all transactions below this threshold.
func (pool *TxPool) SetGasPrice(price *big.Int) {
pool.mu.Lock()
defer pool.mu.Unlock()

pool.gasPrice = price
for _, tx := range pool.priced.Cap(price, pool.locals) {
pool.removeTx(tx.Hash())
}
log.Info("Transaction pool price threshold updated", "price", price)
}

// State returns the virtual managed state of the transaction pool.
func (pool *TxPool) State() *state.ManagedState {
pool.mu.RLock()
defer pool.mu.RUnlock()

return pool.pendingState
}

// Stats retrieves the current pool stats, namely the number of pending and the
// number of queued (non-executable) transactions.
func (pool *TxPool) Stats() (int, int) {
pool.mu.RLock()
defer pool.mu.RUnlock()

return pool.stats()
```

```
}

// stats retrieves the current pool stats, namely the number of pending and the
// number of queued (non-executable) transactions.
func (pool *TxPool) stats() (int, int) {
pending := 0
for _, list := range pool.pending {
pending += list.Len()
}
queued := 0
for _, list := range pool.queue {
queued += list.Len()
}
return pending, queued
}

// Content retrieves the data content of the transaction pool, returning all the
// pending as well as queued transactions, grouped by account and sorted by nonce.
func (pool *TxPool) Content() (map[common.Address]types.Transactions,
map[common.Address]types.Transactions) {
pool.mu.RLock()
defer pool.mu.RUnlock()

pending := make(map[common.Address]types.Transactions)
for addr, list := range pool.pending {
pending[addr] = list.Flatten()
}
queued := make(map[common.Address]types.Transactions)
for addr, list := range pool.queue {
queued[addr] = list.Flatten()
}
return pending, queued
}

// Pending retrieves all currently processable transactions, groupped by origin
// account and sorted by nonce. The returned transaction set is a copy and can be
// freely modified by calling code.
func (pool *TxPool) Pending() (map[common.Address]types.Transactions, error) {
pool.mu.Lock()
defer pool.mu.Unlock()

pending := make(map[common.Address]types.Transactions)
```

```
for addr, list := range pool.pending {
pending[addr] = list.Flatten()
}
return pending, nil
}

// SetLocal marks a transaction as local, skipping gas price
//  check against local miner minimum in the future
func (pool *TxPool) SetLocal(tx *types.Transaction) {
pool.mu.Lock()
defer pool.mu.Unlock()
pool.locals.add(tx.Hash())
}

// validateTx checks whether a transaction is valid according
// to the consensus rules.
func (pool *TxPool) validateTx(tx *types.Transaction) error {
local := pool.locals.contains(tx.Hash())
// Drop transactions under our own minimal accepted gas price
if !local && pool.gasPrice.Cmp(tx.GasPrice()) > 0 {
return ErrUnderpriced
}

currentState, err := pool.currentState()
if err != nil {
return err
}

from, err := types.Sender(pool.signer, tx)
if err != nil {
return ErrInvalidSender
}
// Last but not least check for nonce errors
if currentState.GetNonce(from) > tx.Nonce() {
return ErrNonceTooLow
}

// Check the transaction doesn't exceed the current
// block limit gas.
if pool.gasLimit().Cmp(tx.Gas()) < 0 {
return ErrGasLimit
}
```

```
// Transactions can't be negative. This may never happen
// using RLP decoded transactions but may occur if you create
// a transaction using the RPC for example.
if tx.Value().Sign() < 0 {
return ErrNegativeValue
}

// Transactor should have enough funds to cover the costs
// cost == V + GP * GL
if currentState.GetBalance(from).Cmp(tx.Cost()) < 0 {
return ErrInsufficientFunds
}
intrGas := IntrinsicGas(tx.Data(), tx.To() == nil, pool.homestead)
if tx.Gas().Cmp(intrGas) < 0 {
return ErrIntrinsicGas
}

// Heuristic limit, reject transactions over 32KB to prevent DOS attacks
if tx.Size() > 32*1024 {
return ErrOversizedData
}
return nil
}

// add validates a transaction and inserts it into the non-executable queue for
// later pending promotion and execution. If the transaction is a replacement for
// an already pending or queued one, it overwrites the previous and returns this
// so outer code doesn't uselessly call promote.
func (pool *TxPool) add(tx *types.Transaction) (bool, error) {
// If the transaction is already known, discard it
hash := tx.Hash()
if pool.all[hash] != nil {
log.Trace("Discarding already known transaction", "hash", hash)
return false, fmt.Errorf("known transaction: %x", hash)
}
// If the transaction fails basic validation, discard it
if err := pool.validateTx(tx); err != nil {
log.Trace("Discarding invalid transaction", "hash", hash, "err", err)
invalidTxCounter.Inc(1)
return false, err
}
```

```
// If the transaction pool is full, discard underpriced transactions
if uint64(len(pool.all)) >= pool.config.GlobalSlots+pool.config.GlobalQueue {
// If the new transaction is underpriced, don't accept it
if pool.priced.Underpriced(tx, pool.locals) {
log.Trace("Discarding underpriced transaction", "hash", hash, "price", tx.GasPrice())
underpricedTxCounter.Inc(1)
return false, ErrUnderpriced
}
// New transaction is better than our worse ones, make room for it
drop := pool.priced.Discard(len(pool.all)-int(pool.config.GlobalSlots+pool.config.GlobalQueue-1),
pool.locals)
for _, tx := range drop {
log.Trace("Discarding freshly underpriced transaction", "hash", tx.Hash(), "price", tx.GasPrice())
underpricedTxCounter.Inc(1)
pool.removeTx(tx.Hash())
}
}
// If the transaction is replacing an already pending one, do directly
from, _ := types.Sender(pool.signer, tx) // already validated
if list := pool.pending[from]; list != nil && list.Overlaps(tx) {
// Nonce already pending, check if required price bump is met
inserted, old := list.Add(tx, pool.config.PriceBump)
if !inserted {
pendingDiscardCounter.Inc(1)
return false, ErrReplaceUnderpriced
}
// New transaction is better, replace old one
if old != nil {
delete(pool.all, old.Hash())
pool.priced.Removed()
pendingReplaceCounter.Inc(1)
}
pool.all[tx.Hash()] = tx
pool.priced.Put(tx)

log.Trace("Pooled new executable transaction", "hash", hash, "from", from, "to", tx.To())
return old != nil, nil
}
// New transaction isn't replacing a pending one, push into queue
replace, err := pool.enqueueTx(hash, tx)
if err != nil {
return false, err
```

```
}
log.Trace("Pooled new future transaction", "hash", hash, "from", from, "to", tx.To())
return replace, nil
}

// enqueueTx inserts a new transaction into the non-executable transaction queue.
//
// Note, this method assumes the pool lock is held!
func (pool *TxPool) enqueueTx(hash common.Hash, tx *types.Transaction) (bool, error) {
// Try to insert the transaction into the future queue
from, _ := types.Sender(pool.signer, tx) // already validated
if pool.queue[from] == nil {
pool.queue[from] = newTxList(false)
}
inserted, old := pool.queue[from].Add(tx, pool.config.PriceBump)
if !inserted {
// An older transaction was better, discard this
queuedDiscardCounter.Inc(1)
return false, ErrReplaceUnderpriced
}
// Discard any previous transaction and mark this
if old != nil {
delete(pool.all, old.Hash())
pool.priced.Removed()
queuedReplaceCounter.Inc(1)
}
pool.all[hash] = tx
pool.priced.Put(tx)
return old != nil, nil
}

// promoteTx adds a transaction to the pending (processable) list of transactions.
//
// Note, this method assumes the pool lock is held!
func (pool *TxPool) promoteTx(addr common.Address, hash common.Hash, tx
*types.Transaction) {
// Try to insert the transaction into the pending queue
if pool.pending[addr] == nil {
pool.pending[addr] = newTxList(true)
}
list := pool.pending[addr]
```

```
inserted, old := list.Add(tx, pool.config.PriceBump)
if !inserted {
// An older transaction was better, discard this
delete(pool.all, hash)
pool.priced.Removed()

pendingDiscardCounter.Inc(1)
return
}
// Otherwise discard any previous transaction and mark this
if old != nil {
delete(pool.all, old.Hash())
pool.priced.Removed()

pendingReplaceCounter.Inc(1)
}
// Failsafe to work around direct pending inserts (tests)
if pool.all[hash] == nil {
pool.all[hash] = tx
pool.priced.Put(tx)
}
// Set the potentially new pending nonce and notify any subsystems of the new tx
pool.beats[addr] = time.Now()
pool.pendingState.SetNonce(addr, tx.Nonce()+1)
go pool.eventMux.Post(TxPreEvent{tx})
}

// Add queues a single transaction in the pool if it is valid.
func (pool *TxPool) Add(tx *types.Transaction) error {
pool.mu.Lock()
defer pool.mu.Unlock()

// Try to inject the transaction and update any state
replace, err := pool.add(tx)
if err != nil {
return err
}
// If we added a new transaction, run promotion checks and return
if !replace {
state, err := pool.currentState()
if err != nil {
return err
```

```
}
from, _ := types.Sender(pool.signer, tx) // already validated
pool.promoteExecutables(state, []common.Address{from})
}
return nil
}

// AddBatch attempts to queue a batch of transactions.
func (pool *TxPool) AddBatch(txs []*types.Transaction) error {
pool.mu.Lock()
defer pool.mu.Unlock()

// Add the batch of transaction, tracking the accepted ones
dirty := make(map[common.Address]struct{})
for _, tx := range txs {
if replace, err := pool.add(tx); err == nil {
if !replace {
from, _ := types.Sender(pool.signer, tx) // already validated
dirty[from] = struct{}{}
}
}
}
// Only reprocess the internal state if something was actually added
if len(dirty) > 0 {
state, err := pool.currentState()
if err != nil {
return err
}
addrs := make([]common.Address, 0, len(dirty))
for addr, _ := range dirty {
addrs = append(addrs, addr)
}
pool.promoteExecutables(state, addrs)
}
return nil
}

// Get returns a transaction if it is contained in the pool
// and nil otherwise.
func (pool *TxPool) Get(hash common.Hash) *types.Transaction {
pool.mu.RLock()
defer pool.mu.RUnlock()
```

```
return pool.all[hash]
}

// Remove removes the transaction with the given hash from the pool.
func (pool *TxPool) Remove(hash common.Hash) {
pool.mu.Lock()
defer pool.mu.Unlock()

pool.removeTx(hash)
}

// RemoveBatch removes all given transactions from the pool.
func (pool *TxPool) RemoveBatch(txs types.Transactions) {
pool.mu.Lock()
defer pool.mu.Unlock()

for _, tx := range txs {
pool.removeTx(tx.Hash())
}
}

// removeTx removes a single transaction from the queue, moving all subsequent
// transactions back to the future queue.
func (pool *TxPool) removeTx(hash common.Hash) {
// Fetch the transaction we wish to delete
tx, ok := pool.all[hash]
if !ok {
return
}
addr, _ := types.Sender(pool.signer, tx) // already validated during insertion

// Remove it from the list of known transactions
delete(pool.all, hash)
pool.priced.Removed()

// Remove the transaction from the pending lists and reset the account nonce
if pending := pool.pending[addr]; pending != nil {
if removed, invalids := pending.Remove(tx); removed {
// If no more transactions are left, remove the list
if pending.Empty() {
delete(pool.pending, addr)
```

```
delete(pool.beats, addr)
} else {
// Otherwise postpone any invalidated transactions
for _, tx := range invalids {
pool.enqueueTx(tx.Hash(), tx)
}
}
// Update the account nonce if needed
if nonce := tx.Nonce(); pool.pendingState.GetNonce(addr) > nonce {
pool.pendingState.SetNonce(addr, nonce)
}
return
}
}
// Transaction is in the future queue
if future := pool.queue[addr]; future != nil {
future.Remove(tx)
if future.Empty() {
delete(pool.queue, addr)
}
}
}

// promoteExecutables moves transactions that have become processable from the
// future queue to the set of pending transactions. During this process, all
// invalidated transactions (low nonce, low balance) are deleted.
func (pool *TxPool) promoteExecutables(state *state.StateDB, accounts []common.Address) {
gaslimit := pool.gasLimit()

// Gather all the accounts potentially needing updates
if accounts == nil {
accounts = make([]common.Address, 0, len(pool.queue))
for addr, _ := range pool.queue {
accounts = append(accounts, addr)
}
}
// Iterate over all accounts and promote any executable transactions
queued := uint64(0)
for _, addr := range accounts {
list := pool.queue[addr]
if list == nil {
continue // Just in case someone calls with a non existing account
```

```
}
// Drop all transactions that are deemed too old (low nonce)
for _, tx := range list.Forward(state.GetNonce(addr)) {
hash := tx.Hash()
log.Trace("Removed old queued transaction", "hash", hash)
delete(pool.all, hash)
pool.priced.Removed()
}
// Drop all transactions that are too costly (low balance or out of gas)
drops, _ := list.Filter(state.GetBalance(addr), gaslimit)
for _, tx := range drops {
hash := tx.Hash()
log.Trace("Removed unpayable queued transaction", "hash", hash)
delete(pool.all, hash)
pool.priced.Removed()
queuedNofundsCounter.Inc(1)
}
// Gather all executable transactions and promote them
for _, tx := range list.Ready(pool.pendingState.GetNonce(addr)) {
hash := tx.Hash()
log.Trace("Promoting queued transaction", "hash", hash)
pool.promoteTx(addr, hash, tx)
}
// Drop all transactions over the allowed limit
for _, tx := range list.Cap(int(pool.config.AccountQueue)) {
hash := tx.Hash()
delete(pool.all, hash)
pool.priced.Removed()
queuedRateLimitCounter.Inc(1)
log.Trace("Removed cap-exceeding queued transaction", "hash", hash)
}
queued += uint64(list.Len())

// Delete the entire queue entry if it became empty.
if list.Empty() {
delete(pool.queue, addr)
}
}
// If the pending limit is overflown, start equalizing allowances
pending := uint64(0)
for _, list := range pool.pending {
pending += uint64(list.Len())
```

```
}
if pending > pool.config.GlobalSlots {
pendingBeforeCap := pending
// Assemble a spam order to penalize large transactors first
spammers := prque.New()
for addr, list := range pool.pending {
// Only evict transactions from high rollers
if uint64(list.Len()) > pool.config.AccountSlots {
// Skip local accounts as pools should maintain backlogs for themselves
for _, tx := range list.txs.items {
if !pool.locals.contains(tx.Hash()) {
spammers.Push(addr, float32(list.Len()))
}
break // Checking on transaction for locality is enough
}
}
}
// Gradually drop transactions from offenders
offenders := []common.Address{}
for pending > pool.config.GlobalSlots && !spammers.Empty() {
// Retrieve the next offender if not local address
offender, _ := spammers.Pop()
offenders = append(offenders, offender.(common.Address))

// Equalize balances until all the same or below threshold
if len(offenders) > 1 {
// Calculate the equalization threshold for all current offenders
threshold := pool.pending[offender.(common.Address)].Len()

// Iteratively reduce all offenders until below limit or threshold reached
for pending > pool.config.GlobalSlots && pool.pending[offenders[len(offenders)-2]].Len() >
threshold {
for i := 0; i < len(offenders)-1; i++ {
list := pool.pending[offenders[i]]
for _, tx := range list.Cap(list.Len() - 1) {
// Drop the transaction from the global pools too
hash := tx.Hash()
delete(pool.all, hash)
pool.priced.Removed()

// Update the account nonce to the dropped transaction
if nonce := tx.Nonce(); pool.pendingState.GetNonce(offenders[i]) > nonce {
```

```
pool.pendingState.SetNonce(offenders[i], nonce)
}
log.Trace("Removed fairness-exceeding pending transaction", "hash", hash)
}
pending--
}
}
}
}
// If still above threshold, reduce to limit or min allowance
if pending > pool.config.GlobalSlots && len(offenders) > 0 {
for pending > pool.config.GlobalSlots && uint64(pool.pending[offenders[len(offenders)-1]].Len()) >
pool.config.AccountSlots {
for _, addr := range offenders {
list := pool.pending[addr]
for _, tx := range list.Cap(list.Len() - 1) {
// Drop the transaction from the global pools too
hash := tx.Hash()
delete(pool.all, hash)
pool.priced.Removed()

// Update the account nonce to the dropped transaction
if nonce := tx.Nonce(); pool.pendingState.GetNonce(addr) > nonce {
pool.pendingState.SetNonce(addr, nonce)
}
log.Trace("Removed fairness-exceeding pending transaction", "hash", hash)
}
pending--
}
}
}
pendingRateLimitCounter.Inc(int64(pendingBeforeCap - pending))
}
// If we've queued more transactions than the hard limit, drop oldest ones
if queued > pool.config.GlobalQueue {
// Sort all accounts with queued transactions by heartbeat
addresses := make(addresssByHeartbeat, 0, len(pool.queue))
for addr := range pool.queue {
addresses = append(addresses, addressByHeartbeat{addr, pool.beats[addr]})
}
sort.Sort(addresses)
```

```
// Drop transactions until the total is below the limit
for drop := queued - pool.config.GlobalQueue; drop > 0; {
addr := addresses[len(addresses)-1]
list := pool.queue[addr.address]

addresses = addresses[:len(addresses)-1]

// Drop all transactions if they are less than the overflow
if size := uint64(list.Len()); size <= drop {
for _, tx := range list.Flatten() {
pool.removeTx(tx.Hash())
}
drop -= size
queuedRateLimitCounter.Inc(int64(size))
continue
}
// Otherwise drop only last few transactions
txs := list.Flatten()
for i := len(txs) - 1; i >= 0 && drop > 0; i-- {
pool.removeTx(txs[i].Hash())
drop--
queuedRateLimitCounter.Inc(1)
}
}
}
}

// demoteUnexecutables removes invalid and processed transactions from the pools
// executable/pending queue and any subsequent transactions that become unexecutable
// are moved back into the future queue.
func (pool *TxPool) demoteUnexecutables(state *state.StateDB) {
gaslimit := pool.gasLimit()

// Iterate over all accounts and demote any non-executable transactions
for addr, list := range pool.pending {
nonce := state.GetNonce(addr)

// Drop all transactions that are deemed too old (low nonce)
for _, tx := range list.Forward(nonce) {
hash := tx.Hash()
log.Trace("Removed old pending transaction", "hash", hash)
delete(pool.all, hash)
```

```
pool.priced.Removed()
}
// Drop all transactions that are too costly (low balance or out of gas), and queue any invalids back for later
drops, invalids := list.Filter(state.GetBalance(addr), gaslimit)
for _, tx := range drops {
hash := tx.Hash()
log.Trace("Removed unpayable pending transaction", "hash", hash)
delete(pool.all, hash)
pool.priced.Removed()
pendingNofundsCounter.Inc(1)
}
for _, tx := range invalids {
hash := tx.Hash()
log.Trace("Demoting pending transaction", "hash", hash)
pool.enqueueTx(hash, tx)
}
// Delete the entire queue entry if it became empty.
if list.Empty() {
delete(pool.pending, addr)
delete(pool.beats, addr)
}
}
}

// expirationLoop is a loop that periodically iterates over all accounts with
// queued transactions and drop all that have been inactive for a prolonged amount
// of time.
func (pool *TxPool) expirationLoop() {
defer pool.wg.Done()

evict := time.NewTicker(evictionInterval)
defer evict.Stop()

for {
select {
case <-evict.C:
pool.mu.Lock()
for addr := range pool.queue {
if time.Since(pool.beats[addr]) > pool.config.Lifetime {
for _, tx := range pool.queue[addr].Flatten() {
pool.removeTx(tx.Hash())
```

```
}
}
}
pool.mu.Unlock()

case <-pool.quit:
return
}
}
}

// addressByHeartbeat is an account address tagged with its last activity timestamp.
type addressByHeartbeat struct {
address   common.Address
heartbeat time.Time
}

type addresssByHeartbeat []addressByHeartbeat

func (a addresssByHeartbeat) Len() int           { return len(a) }
func (a addresssByHeartbeat) Less(i, j int) bool { return a[i].heartbeat.Before(a[j].heartbeat) }
func (a addresssByHeartbeat) Swap(i, j int)      { a[i], a[j] = a[j], a[i] }

// txSet represents a set of transaction hashes in which entries
//  are automatically dropped after txSetDuration time
type txSet struct {
txMap          map[common.Hash]struct{}
txOrd          map[uint64]txOrdType
addPtr, delPtr uint64
}

const txSetDuration = time.Hour * 2

// txOrdType represents an entry in the time-ordered list of transaction hashes
type txOrdType struct {
hash common.Hash
time time.Time
}

// newTxSet creates a new transaction set
func newTxSet() *txSet {
return &txSet{
```

```
txMap: make(map[common.Hash]struct{}),
txOrd: make(map[uint64]txOrdType),
}
}

// contains returns true if the set contains the given transaction hash
// (not thread safe, should be called from a locked environment)
func (ts *txSet) contains(hash common.Hash) bool {
_, ok := ts.txMap[hash]
return ok
}

// add adds a transaction hash to the set, then removes entries older than txSetDuration
// (not thread safe, should be called from a locked environment)
func (ts *txSet) add(hash common.Hash) {
ts.txMap[hash] = struct{}{}
now := time.Now()
ts.txOrd[ts.addPtr] = txOrdType{hash: hash, time: now}
ts.addPtr++
delBefore := now.Add(-txSetDuration)
for ts.delPtr < ts.addPtr && ts.txOrd[ts.delPtr].time.Before(delBefore) {
delete(ts.txMap, ts.txOrd[ts.delPtr].hash)
delete(ts.txOrd, ts.delPtr)
ts.delPtr++
}
}

4:F:\git\coin\ethereum\go-ethereum\core\tx_pool_test.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package core

import (
"crypto/ecdsa"
"fmt"
"math/big"
"math/rand"
"testing"
"time"

"github.com/ethereum/go-ethereum/common"
"github.com/ethereum/go-ethereum/core/state"
```

```
"github.com/ethereum/go-ethereum/core/types"
"github.com/ethereum/go-ethereum/crypto"
"github.com/ethereum/go-ethereum/ethdb"
"github.com/ethereum/go-ethereum/event"
"github.com/ethereum/go-ethereum/params"
)

func transaction(nonce uint64, gaslimit *big.Int, key *ecdsa.PrivateKey) *types.Transaction {
return pricedTransaction(nonce, gaslimit, big.NewInt(1), key)
}

func pricedTransaction(nonce uint64, gaslimit, gasprice *big.Int, key *ecdsa.PrivateKey)
*types.Transaction {
tx, _ := types.SignTx(types.NewTransaction(nonce, common.Address{}, big.NewInt(100), gaslimit,
gasprice, nil), types.HomesteadSigner{}, key)
return tx
}

func setupTxPool() (*TxPool, *ecdsa.PrivateKey) {
db, _ := ethdb.NewMemDatabase()
statedb, _ := state.New(common.Hash{}, state.NewDatabase(db))

key, _ := crypto.GenerateKey()
newPool := NewTxPool(DefaultTxPoolConfig, params.TestChainConfig, new(event.TypeMux),
func() (*state.StateDB, error) { return statedb, nil }, func() *big.Int { return big.NewInt(1000000) })
newPool.resetState()

return newPool, key
}

// validateTxPoolInternals checks various consistency invariants within the pool.
func validateTxPoolInternals(pool *TxPool) error {
pool.mu.RLock()
defer pool.mu.RUnlock()

// Ensure the total transaction set is consistent with pending + queued
pending, queued := pool.stats()
if total := len(pool.all); total != pending+queued {
return fmt.Errorf("total transaction count %d != %d pending + %d queued", total, pending, queued)
}
if priced := pool.priced.items.Len() - pool.priced.stales; priced != pending+queued {
return fmt.Errorf("total priced transaction count %d != %d pending + %d queued", priced, pending,
```

```
queued)
}
// Ensure the next nonce to assign is the correct one
for addr, txs := range pool.pending {
// Find the last transaction
var last uint64
for nonce, _ := range txs.txs.items {
if last < nonce {
last = nonce
}
}
if nonce := pool.pendingState.GetNonce(addr); nonce != last+1 {
return fmt.Errorf("pending nonce mismatch: have %v, want %v", nonce, last+1)
}
}
return nil
}

func deriveSender(tx *types.Transaction) (common.Address, error) {
return types.Sender(types.HomesteadSigner{}, tx)
}

// This test simulates a scenario where a new block is imported during a
// state reset and tests whether the pending state is in sync with the
// block head event that initiated the resetState().
func TestStateChangeDuringPoolReset(t *testing.T) {
var (
db, _      = ethdb.NewMemDatabase()
key, _     = crypto.GenerateKey()
address    = crypto.PubkeyToAddress(key.PublicKey)
mux        = new(event.TypeMux)
statedb, _ = state.New(common.Hash{}, state.NewDatabase(db))
trigger    = false
)

// setup pool with 2 transaction in it
statedb.SetBalance(address, new(big.Int).SetUint64(params.Ether))

tx0 := transaction(0, big.NewInt(100000), key)
tx1 := transaction(1, big.NewInt(100000), key)

// stateFunc is used multiple times to reset the pending state.
```

```
// when simulate is true it will create a state that indicates
// that tx0 and tx1 are included in the chain.
stateFunc := func() (*state.StateDB, error) {
// delay "state change" by one. The tx pool fetches the
// state multiple times and by delaying it a bit we simulate
// a state change between those fetches.
stdb := statedb
if trigger {
statedb, _ = state.New(common.Hash{}, state.NewDatabase(db))
// simulate that the new head block included tx0 and tx1
statedb.SetNonce(address, 2)
statedb.SetBalance(address, new(big.Int).SetUint64(params.Ether))
trigger = false
}
return stdb, nil
}

gasLimitFunc := func() *big.Int { return big.NewInt(1000000000) }

txpool := NewTxPool(DefaultTxPoolConfig, params.TestChainConfig, mux, stateFunc,
gasLimitFunc)
txpool.resetState()

nonce := txpool.State().GetNonce(address)
if nonce != 0 {
t.Fatalf("Invalid nonce, want 0, got %d", nonce)
}

txpool.AddBatch(types.Transactions{tx0, tx1})

nonce = txpool.State().GetNonce(address)
if nonce != 2 {
t.Fatalf("Invalid nonce, want 2, got %d", nonce)
}

// trigger state change in the background
trigger = true

txpool.resetState()

pendingTx, err := txpool.Pending()
if err != nil {
```

```
t.Fatalf("Could not fetch pending transactions: %v", err)
}

for addr, txs := range pendingTx {
t.Logf("%0x: %d\n", addr, len(txs))
}

nonce = txpool.State().GetNonce(address)
if nonce != 2 {
t.Fatalf("Invalid nonce, want 2, got %d", nonce)
}
}

func TestInvalidTransactions(t *testing.T) {
pool, key := setupTxPool()

tx := transaction(0, big.NewInt(100), key)
from, _ := deriveSender(tx)
currentState, _ := pool.currentState()
currentState.AddBalance(from, big.NewInt(1))
if err := pool.Add(tx); err != ErrInsufficientFunds {
t.Error("expected", ErrInsufficientFunds)
}

balance := new(big.Int).Add(tx.Value(), new(big.Int).Mul(tx.Gas(), tx.GasPrice()))
currentState.AddBalance(from, balance)
if err := pool.Add(tx); err != ErrIntrinsicGas {
t.Error("expected", ErrIntrinsicGas, "got", err)
}

currentState.SetNonce(from, 1)
currentState.AddBalance(from, big.NewInt(0xffffffffffffff))
tx = transaction(0, big.NewInt(100000), key)
if err := pool.Add(tx); err != ErrNonceTooLow {
t.Error("expected", ErrNonceTooLow)
}

tx = transaction(1, big.NewInt(100000), key)
pool.gasPrice = big.NewInt(1000)
if err := pool.Add(tx); err != ErrUnderpriced {
t.Error("expected", ErrUnderpriced, "got", err)
}
```

```
pool.SetLocal(tx)
if err := pool.Add(tx); err != nil {
t.Error("expected", nil, "got", err)
}
}

func TestTransactionQueue(t *testing.T) {
pool, key := setupTxPool()
tx := transaction(0, big.NewInt(100), key)
from, _ := deriveSender(tx)
currentState, _ := pool.currentState()
currentState.AddBalance(from, big.NewInt(1000))
pool.resetState()
pool.enqueueTx(tx.Hash(), tx)

pool.promoteExecutables(currentState, []common.Address{from})
if len(pool.pending) != 1 {
t.Error("expected valid txs to be 1 is", len(pool.pending))
}

tx = transaction(1, big.NewInt(100), key)
from, _ = deriveSender(tx)
currentState.SetNonce(from, 2)
pool.enqueueTx(tx.Hash(), tx)
pool.promoteExecutables(currentState, []common.Address{from})
if _, ok := pool.pending[from].txs.items[tx.Nonce()]; ok {
t.Error("expected transaction to be in tx pool")
}

if len(pool.queue) > 0 {
t.Error("expected transaction queue to be empty. is", len(pool.queue))
}

pool, key = setupTxPool()
tx1 := transaction(0, big.NewInt(100), key)
tx2 := transaction(10, big.NewInt(100), key)
tx3 := transaction(11, big.NewInt(100), key)
from, _ = deriveSender(tx1)
currentState, _ = pool.currentState()
currentState.AddBalance(from, big.NewInt(1000))
pool.resetState()
```

```
pool.enqueueTx(tx1.Hash(), tx1)
pool.enqueueTx(tx2.Hash(), tx2)
pool.enqueueTx(tx3.Hash(), tx3)

pool.promoteExecutables(currentState, []common.Address{from})

if len(pool.pending) != 1 {
t.Error("expected tx pool to be 1, got", len(pool.pending))
}
if pool.queue[from].Len() != 2 {
t.Error("expected len(queue) == 2, got", pool.queue[from].Len())
}
}

func TestRemoveTx(t *testing.T) {
pool, key := setupTxPool()
addr := crypto.PubkeyToAddress(key.PublicKey)
currentState, _ := pool.currentState()
currentState.AddBalance(addr, big.NewInt(1))

tx1 := transaction(0, big.NewInt(100), key)
tx2 := transaction(2, big.NewInt(100), key)

pool.promoteTx(addr, tx1.Hash(), tx1)
pool.enqueueTx(tx2.Hash(), tx2)

if len(pool.queue) != 1 {
t.Error("expected queue to be 1, got", len(pool.queue))
}
if len(pool.pending) != 1 {
t.Error("expected pending to be 1, got", len(pool.pending))
}
pool.Remove(tx1.Hash())
pool.Remove(tx2.Hash())

if len(pool.queue) > 0 {
t.Error("expected queue to be 0, got", len(pool.queue))
}
if len(pool.pending) > 0 {
t.Error("expected pending to be 0, got", len(pool.pending))
}
```

```
}

func TestNegativeValue(t *testing.T) {
pool, key := setupTxPool()

tx, _ := types.SignTx(types.NewTransaction(0, common.Address{}, big.NewInt(-1),
big.NewInt(100), big.NewInt(1), nil), types.HomesteadSigner{}, key)
from, _ := deriveSender(tx)
currentState, _ := pool.currentState()
currentState.AddBalance(from, big.NewInt(1))
if err := pool.Add(tx); err != ErrNegativeValue {
t.Error("expected", ErrNegativeValue, "got", err)
}
}

func TestTransactionChainFork(t *testing.T) {
pool, key := setupTxPool()
addr := crypto.PubkeyToAddress(key.PublicKey)
resetState := func() {
db, _ := ethdb.NewMemDatabase()
statedb, _ := state.New(common.Hash{}, state.NewDatabase(db))
pool.currentState = func() (*state.StateDB, error) { return statedb, nil }
currentState, _ := pool.currentState()
currentState.AddBalance(addr, big.NewInt(100000000000000))
pool.resetState()
}
resetState()

tx := transaction(0, big.NewInt(100000), key)
if _, err := pool.add(tx); err != nil {
t.Error("didn't expect error", err)
}
pool.RemoveBatch([]*types.Transaction{tx})

// reset the pool's internal state
resetState()
if _, err := pool.add(tx); err != nil {
t.Error("didn't expect error", err)
}
}

func TestTransactionDoubleNonce(t *testing.T) {
```

```
pool, key := setupTxPool()
addr := crypto.PubkeyToAddress(key.PublicKey)
resetState := func() {
db, _ := ethdb.NewMemDatabase()
statedb, _ := state.New(common.Hash{}, state.NewDatabase(db))
pool.currentState = func() (*state.StateDB, error) { return statedb, nil }
currentState, _ := pool.currentState()
currentState.AddBalance(addr, big.NewInt(100000000000000))
pool.resetState()
}
resetState()

signer := types.HomesteadSigner{}
tx1, _ := types.SignTx(types.NewTransaction(0, common.Address{}, big.NewInt(100),
big.NewInt(100000), big.NewInt(1), nil), signer, key)
tx2, _ := types.SignTx(types.NewTransaction(0, common.Address{}, big.NewInt(100),
big.NewInt(1000000), big.NewInt(2), nil), signer, key)
tx3, _ := types.SignTx(types.NewTransaction(0, common.Address{}, big.NewInt(100),
big.NewInt(1000000), big.NewInt(1), nil), signer, key)

// Add the first two transaction, ensure higher priced stays only
if replace, err := pool.add(tx1); err != nil || replace {
t.Errorf("first transaction insert failed (%v) or reported replacement (%v)", err, replace)
}
if replace, err := pool.add(tx2); err != nil || !replace {
t.Errorf("second transaction insert failed (%v) or not reported replacement (%v)", err, replace)
}
state, _ := pool.currentState()
pool.promoteExecutables(state, []common.Address{addr})
if pool.pending[addr].Len() != 1 {
t.Error("expected 1 pending transactions, got", pool.pending[addr].Len())
}
if tx := pool.pending[addr].txs.items[0]; tx.Hash() != tx2.Hash() {
t.Errorf("transaction mismatch: have %x, want %x", tx.Hash(), tx2.Hash())
}
// Add the third transaction and ensure it's not saved (smaller price)
pool.add(tx3)
pool.promoteExecutables(state, []common.Address{addr})
if pool.pending[addr].Len() != 1 {
t.Error("expected 1 pending transactions, got", pool.pending[addr].Len())
}
if tx := pool.pending[addr].txs.items[0]; tx.Hash() != tx2.Hash() {
```

```
t.Errorf("transaction mismatch: have %x, want %x", tx.Hash(), tx2.Hash())
}
// Ensure the total transaction count is correct
if len(pool.all) != 1 {
t.Error("expected 1 total transactions, got", len(pool.all))
}
}

func TestMissingNonce(t *testing.T) {
pool, key := setupTxPool()
addr := crypto.PubkeyToAddress(key.PublicKey)
currentState, _ := pool.currentState()
currentState.AddBalance(addr, big.NewInt(100000000000000))
tx := transaction(1, big.NewInt(100000), key)
if _, err := pool.add(tx); err != nil {
t.Error("didn't expect error", err)
}
if len(pool.pending) != 0 {
t.Error("expected 0 pending transactions, got", len(pool.pending))
}
if pool.queue[addr].Len() != 1 {
t.Error("expected 1 queued transaction, got", pool.queue[addr].Len())
}
if len(pool.all) != 1 {
t.Error("expected 1 total transactions, got", len(pool.all))
}
}

func TestNonceRecovery(t *testing.T) {
const n = 10
pool, key := setupTxPool()
addr := crypto.PubkeyToAddress(key.PublicKey)
currentState, _ := pool.currentState()
currentState.SetNonce(addr, n)
currentState.AddBalance(addr, big.NewInt(100000000000000))
pool.resetState()
tx := transaction(n, big.NewInt(100000), key)
if err := pool.Add(tx); err != nil {
t.Error(err)
}
// simulate some weird re-order of transactions and missing nonce(s)
currentState.SetNonce(addr, n-1)
```

```
pool.resetState()
if fn := pool.pendingState.GetNonce(addr); fn != n+1 {
t.Errorf("expected nonce to be %d, got %d", n+1, fn)
}
}

func TestRemovedTxEvent(t *testing.T) {
pool, key := setupTxPool()
tx := transaction(0, big.NewInt(1000000), key)
from, _ := deriveSender(tx)
currentState, _ := pool.currentState()
currentState.AddBalance(from, big.NewInt(1000000000000))
pool.resetState()
pool.eventMux.Post(RemovedTransactionEvent{types.Transactions{tx}})
pool.eventMux.Post(ChainHeadEvent{nil})
if pool.pending[from].Len() != 1 {
t.Error("expected 1 pending tx, got", pool.pending[from].Len())
}
if len(pool.all) != 1 {
t.Error("expected 1 total transactions, got", len(pool.all))
}
}

// Tests that if an account runs out of funds, any pending and queued transactions
// are dropped.
func TestTransactionDropping(t *testing.T) {
// Create a test account and fund it
pool, key := setupTxPool()
account, _ := deriveSender(transaction(0, big.NewInt(0), key))

state, _ := pool.currentState()
state.AddBalance(account, big.NewInt(1000))

// Add some pending and some queued transactions
var (
tx0  = transaction(0, big.NewInt(100), key)
tx1  = transaction(1, big.NewInt(200), key)
tx2  = transaction(2, big.NewInt(300), key)
tx10 = transaction(10, big.NewInt(100), key)
tx11 = transaction(11, big.NewInt(200), key)
tx12 = transaction(12, big.NewInt(300), key)
)
```

```
pool.promoteTx(account, tx0.Hash(), tx0)
pool.promoteTx(account, tx1.Hash(), tx1)
pool.promoteTx(account, tx2.Hash(), tx2)
pool.enqueueTx(tx10.Hash(), tx10)
pool.enqueueTx(tx11.Hash(), tx11)
pool.enqueueTx(tx12.Hash(), tx12)

// Check that pre and post validations leave the pool as is
if pool.pending[account].Len() != 3 {
t.Errorf("pending transaction mismatch: have %d, want %d", pool.pending[account].Len(), 3)
}
if pool.queue[account].Len() != 3 {
t.Errorf("queued transaction mismatch: have %d, want %d", pool.queue[account].Len(), 3)
}
if len(pool.all) != 6 {
t.Errorf("total transaction mismatch: have %d, want %d", len(pool.all), 6)
}
pool.resetState()
if pool.pending[account].Len() != 3 {
t.Errorf("pending transaction mismatch: have %d, want %d", pool.pending[account].Len(), 3)
}
if pool.queue[account].Len() != 3 {
t.Errorf("queued transaction mismatch: have %d, want %d", pool.queue[account].Len(), 3)
}
if len(pool.all) != 6 {
t.Errorf("total transaction mismatch: have %d, want %d", len(pool.all), 6)
}
// Reduce the balance of the account, and check that invalidated transactions are dropped
state.AddBalance(account, big.NewInt(-650))
pool.resetState()

if _, ok := pool.pending[account].txs.items[tx0.Nonce()]; !ok {
t.Errorf("funded pending transaction missing: %v", tx0)
}
if _, ok := pool.pending[account].txs.items[tx1.Nonce()]; !ok {
t.Errorf("funded pending transaction missing: %v", tx0)
}
if _, ok := pool.pending[account].txs.items[tx2.Nonce()]; ok {
t.Errorf("out-of-fund pending transaction present: %v", tx1)
}
if _, ok := pool.queue[account].txs.items[tx10.Nonce()]; !ok {
t.Errorf("funded queued transaction missing: %v", tx10)
```

```
}
if _, ok := pool.queue[account].txs.items[tx11.Nonce()]; !ok {
t.Errorf("funded queued transaction missing: %v", tx10)
}
if _, ok := pool.queue[account].txs.items[tx12.Nonce()]; ok {
t.Errorf("out-of-fund queued transaction present: %v", tx11)
}
if len(pool.all) != 4 {
t.Errorf("total transaction mismatch: have %d, want %d", len(pool.all), 4)
}
// Reduce the block gas limit, check that invalidated transactions are dropped
pool.gasLimit = func() *big.Int { return big.NewInt(100) }
pool.resetState()

if _, ok := pool.pending[account].txs.items[tx0.Nonce()]; !ok {
t.Errorf("funded pending transaction missing: %v", tx0)
}
if _, ok := pool.pending[account].txs.items[tx1.Nonce()]; ok {
t.Errorf("over-gased pending transaction present: %v", tx1)
}
if _, ok := pool.queue[account].txs.items[tx10.Nonce()]; !ok {
t.Errorf("funded queued transaction missing: %v", tx10)
}
if _, ok := pool.queue[account].txs.items[tx11.Nonce()]; ok {
t.Errorf("over-gased queued transaction present: %v", tx11)
}
if len(pool.all) != 2 {
t.Errorf("total transaction mismatch: have %d, want %d", len(pool.all), 2)
}
}

// Tests that if a transaction is dropped from the current pending pool (e.g. out
// of fund), all consecutive (still valid, but not executable) transactions are
// postponed back into the future queue to prevent broadcasting them.
func TestTransactionPostponing(t *testing.T) {
// Create a test account and fund it
pool, key := setupTxPool()
account, _ := deriveSender(transaction(0, big.NewInt(0), key))

state, _ := pool.currentState()
state.AddBalance(account, big.NewInt(1000))
```

```
// Add a batch consecutive pending transactions for validation
txns := []*types.Transaction{}
for i := 0; i < 100; i++ {
var tx *types.Transaction
if i%2 == 0 {
tx = transaction(uint64(i), big.NewInt(100), key)
} else {
tx = transaction(uint64(i), big.NewInt(500), key)
}
pool.promoteTx(account, tx.Hash(), tx)
txns = append(txns, tx)
}
// Check that pre and post validations leave the pool as is
if pool.pending[account].Len() != len(txns) {
t.Errorf("pending transaction mismatch: have %d, want %d", pool.pending[account].Len(),
len(txns))
}
if len(pool.queue) != 0 {
t.Errorf("queued transaction mismatch: have %d, want %d", pool.queue[account].Len(), 0)
}
if len(pool.all) != len(txns) {
t.Errorf("total transaction mismatch: have %d, want %d", len(pool.all), len(txns))
}
pool.resetState()
if pool.pending[account].Len() != len(txns) {
t.Errorf("pending transaction mismatch: have %d, want %d", pool.pending[account].Len(),
len(txns))
}
if len(pool.queue) != 0 {
t.Errorf("queued transaction mismatch: have %d, want %d", pool.queue[account].Len(), 0)
}
if len(pool.all) != len(txns) {
t.Errorf("total transaction mismatch: have %d, want %d", len(pool.all), len(txns))
}
// Reduce the balance of the account, and check that transactions are reorganised
state.AddBalance(account, big.NewInt(-750))
pool.resetState()

if _, ok := pool.pending[account].txs.items[txns[0].Nonce()]; !ok {
t.Errorf("tx %d: valid and funded transaction missing from pending pool: %v", 0, txns[0])
}
if _, ok := pool.queue[account].txs.items[txns[0].Nonce()]; ok {
```

```
t.Errorf("tx %d: valid and funded transaction present in future queue: %v", 0, txns[0])
}
for i, tx := range txns[1:] {
if i%2 == 1 {
if _, ok := pool.pending[account].txs.items[tx.Nonce()]; ok {
t.Errorf("tx %d: valid but future transaction present in pending pool: %v", i+1, tx)
}
if _, ok := pool.queue[account].txs.items[tx.Nonce()]; !ok {
t.Errorf("tx %d: valid but future transaction missing from future queue: %v", i+1, tx)
}
} else {
if _, ok := pool.pending[account].txs.items[tx.Nonce()]; ok {
t.Errorf("tx %d: out-of-fund transaction present in pending pool: %v", i+1, tx)
}
if _, ok := pool.queue[account].txs.items[tx.Nonce()]; ok {
t.Errorf("tx %d: out-of-fund transaction present in future queue: %v", i+1, tx)
}
}
}
if len(pool.all) != len(txns)/2 {
t.Errorf("total transaction mismatch: have %d, want %d", len(pool.all), len(txns)/2)
}
}

// Tests that if the transaction count belonging to a single account goes above
// some threshold, the higher transactions are dropped to prevent DOS attacks.
func TestTransactionQueueAccountLimiting(t *testing.T) {
// Create a test account and fund it
pool, key := setupTxPool()
account, _ := deriveSender(transaction(0, big.NewInt(0), key))

state, _ := pool.currentState()
state.AddBalance(account, big.NewInt(1000000))
pool.resetState()

// Keep queuing up transactions and make sure all above a limit are dropped
for i := uint64(1); i <= DefaultTxPoolConfig.AccountQueue+5; i++ {
if err := pool.Add(transaction(i, big.NewInt(100000), key)); err != nil {
t.Fatalf("tx %d: failed to add transaction: %v", i, err)
}
if len(pool.pending) != 0 {
t.Errorf("tx %d: pending pool size mismatch: have %d, want %d", i, len(pool.pending), 0)
```

```
}
if i <= DefaultTxPoolConfig.AccountQueue {
if pool.queue[account].Len() != int(i) {
t.Errorf("tx %d: queue size mismatch: have %d, want %d", i, pool.queue[account].Len(), i)
}
} else {
if pool.queue[account].Len() != int(DefaultTxPoolConfig.AccountQueue) {
t.Errorf("tx %d: queue limit mismatch: have %d, want %d", i, pool.queue[account].Len(),
DefaultTxPoolConfig.AccountQueue)
}
}
}
if len(pool.all) != int(DefaultTxPoolConfig.AccountQueue) {
t.Errorf("total transaction mismatch: have %d, want %d", len(pool.all),
DefaultTxPoolConfig.AccountQueue)
}
}

// Tests that if the transaction count belonging to multiple accounts go above
// some threshold, the higher transactions are dropped to prevent DOS attacks.
func TestTransactionQueueGlobalLimiting(t *testing.T) {
// Reduce the queue limits to shorten test time
defer func(old uint64) { DefaultTxPoolConfig.GlobalQueue = old
}(DefaultTxPoolConfig.GlobalQueue)
DefaultTxPoolConfig.GlobalQueue = DefaultTxPoolConfig.AccountQueue * 3

// Create the pool to test the limit enforcement with
db, _ := ethdb.NewMemDatabase()
statedb, _ := state.New(common.Hash{}, state.NewDatabase(db))

pool := NewTxPool(DefaultTxPoolConfig, params.TestChainConfig, new(event.TypeMux), func()
(*state.StateDB, error) { return statedb, nil }, func() *big.Int { return big.NewInt(1000000) })
pool.resetState()

// Create a number of test accounts and fund them
state, _ := pool.currentState()

keys := make([]*ecdsa.PrivateKey, 5)
for i := 0; i < len(keys); i++ {
keys[i], _ = crypto.GenerateKey()
state.AddBalance(crypto.PubkeyToAddress(keys[i].PublicKey), big.NewInt(1000000))
}
```

```
// Generate and queue a batch of transactions
nonces := make(map[common.Address]uint64)

txs := make(types.Transactions, 0, 3*DefaultTxPoolConfig.GlobalQueue)
for len(txs) < cap(txs) {
key := keys[rand.Intn(len(keys))]
addr := crypto.PubkeyToAddress(key.PublicKey)

txs = append(txs, transaction(nonces[addr]+1, big.NewInt(100000), key))
nonces[addr]++
}
// Import the batch and verify that limits have been enforced
pool.AddBatch(txs)

queued := 0
for addr, list := range pool.queue {
if list.Len() > int(DefaultTxPoolConfig.AccountQueue) {
t.Errorf("addr %x: queued accounts overflown allowance: %d > %d", addr, list.Len(),
DefaultTxPoolConfig.AccountQueue)
}
queued += list.Len()
}
if queued > int(DefaultTxPoolConfig.GlobalQueue) {
t.Fatalf("total transactions overflow allowance: %d > %d", queued,
DefaultTxPoolConfig.GlobalQueue)
}
}

// Tests that if an account remains idle for a prolonged amount of time, any
// non-executable transactions queued up are dropped to prevent wasting resources
// on shuffling them around.
func TestTransactionQueueTimeLimiting(t *testing.T) {
// Reduce the queue limits to shorten test time
defer func(old time.Duration) { DefaultTxPoolConfig.Lifetime = old }(DefaultTxPoolConfig.Lifetime)
defer func(old time.Duration) { evictionInterval = old }(evictionInterval)
DefaultTxPoolConfig.Lifetime = time.Second
evictionInterval = time.Second

// Create a test account and fund it
pool, key := setupTxPool()
account, _ := deriveSender(transaction(0, big.NewInt(0), key))
```

```
state, _ := pool.currentState()
state.AddBalance(account, big.NewInt(1000000))

// Queue up a batch of transactions
for i := uint64(1); i <= DefaultTxPoolConfig.AccountQueue; i++ {
if err := pool.Add(transaction(i, big.NewInt(100000), key)); err != nil {
t.Fatalf("tx %d: failed to add transaction: %v", i, err)
}
}
// Wait until at least two expiration cycles hit and make sure the transactions are gone
time.Sleep(2 * evictionInterval)
if len(pool.queue) > 0 {
t.Fatalf("old transactions remained after eviction")
}
}

// Tests that even if the transaction count belonging to a single account goes
// above some threshold, as long as the transactions are executable, they are
// accepted.
func TestTransactionPendingLimiting(t *testing.T) {
// Create a test account and fund it
pool, key := setupTxPool()
account, _ := deriveSender(transaction(0, big.NewInt(0), key))

state, _ := pool.currentState()
state.AddBalance(account, big.NewInt(1000000))
pool.resetState()

// Keep queuing up transactions and make sure all above a limit are dropped
for i := uint64(0); i < DefaultTxPoolConfig.AccountQueue+5; i++ {
if err := pool.Add(transaction(i, big.NewInt(100000), key)); err != nil {
t.Fatalf("tx %d: failed to add transaction: %v", i, err)
}
if pool.pending[account].Len() != int(i)+1 {
t.Errorf("tx %d: pending pool size mismatch: have %d, want %d", i, pool.pending[account].Len(),
i+1)
}
if len(pool.queue) != 0 {
t.Errorf("tx %d: queue size mismatch: have %d, want %d", i, pool.queue[account].Len(), 0)
}
}
if len(pool.all) != int(DefaultTxPoolConfig.AccountQueue+5) {
```

```
t.Errorf("total transaction mismatch: have %d, want %d", len(pool.all),
DefaultTxPoolConfig.AccountQueue+5)
}
}

// Tests that the transaction limits are enforced the same way irrelevant whether
// the transactions are added one by one or in batches.
func TestTransactionQueueLimitingEquivalency(t *testing.T)   {
testTransactionLimitingEquivalency(t, 1) }
func TestTransactionPendingLimitingEquivalency(t *testing.T) {
testTransactionLimitingEquivalency(t, 0) }

func testTransactionLimitingEquivalency(t *testing.T, origin uint64) {
// Add a batch of transactions to a pool one by one
pool1, key1 := setupTxPool()
account1, _ := deriveSender(transaction(0, big.NewInt(0), key1))
state1, _ := pool1.currentState()
state1.AddBalance(account1, big.NewInt(1000000))

for i := uint64(0); i < DefaultTxPoolConfig.AccountQueue+5; i++ {
if err := pool1.Add(transaction(origin+i, big.NewInt(100000), key1)); err != nil {
t.Fatalf("tx %d: failed to add transaction: %v", i, err)
}
}
// Add a batch of transactions to a pool in one big batch
pool2, key2 := setupTxPool()
account2, _ := deriveSender(transaction(0, big.NewInt(0), key2))
state2, _ := pool2.currentState()
state2.AddBalance(account2, big.NewInt(1000000))

txns := []*types.Transaction{}
for i := uint64(0); i < DefaultTxPoolConfig.AccountQueue+5; i++ {
txns = append(txns, transaction(origin+i, big.NewInt(100000), key2))
}
pool2.AddBatch(txns)

// Ensure the batch optimization honors the same pool mechanics
if len(pool1.pending) != len(pool2.pending) {
t.Errorf("pending transaction count mismatch: one-by-one algo: %d, batch algo: %d",
len(pool1.pending), len(pool2.pending))
}
if len(pool1.queue) != len(pool2.queue) {
```

```
t.Errorf("queued transaction count mismatch: one-by-one algo: %d, batch algo: %d",
len(pool1.queue), len(pool2.queue))
}
if len(pool1.all) != len(pool2.all) {
t.Errorf("total transaction count mismatch: one-by-one algo %d, batch algo %d", len(pool1.all),
len(pool2.all))
}
if err := validateTxPoolInternals(pool1); err != nil {
t.Errorf("pool 1 internal state corrupted: %v", err)
}
if err := validateTxPoolInternals(pool2); err != nil {
t.Errorf("pool 2 internal state corrupted: %v", err)
}
}

// Tests that if the transaction count belonging to multiple accounts go above
// some hard threshold, the higher transactions are dropped to prevent DOS
// attacks.
func TestTransactionPendingGlobalLimiting(t *testing.T) {
// Reduce the queue limits to shorten test time
defer func(old uint64) { DefaultTxPoolConfig.GlobalSlots = old }(DefaultTxPoolConfig.GlobalSlots)
DefaultTxPoolConfig.GlobalSlots = DefaultTxPoolConfig.AccountSlots * 10

// Create the pool to test the limit enforcement with
db, _ := ethdb.NewMemDatabase()
statedb, _ := state.New(common.Hash{}, state.NewDatabase(db))

pool := NewTxPool(DefaultTxPoolConfig, params.TestChainConfig, new(event.TypeMux), func()
(*state.StateDB, error) { return statedb, nil }, func() *big.Int { return big.NewInt(1000000) })
pool.resetState()

// Create a number of test accounts and fund them
state, _ := pool.currentState()

keys := make([]*ecdsa.PrivateKey, 5)
for i := 0; i < len(keys); i++ {
keys[i], _ = crypto.GenerateKey()
state.AddBalance(crypto.PubkeyToAddress(keys[i].PublicKey), big.NewInt(1000000))
}
// Generate and queue a batch of transactions
nonces := make(map[common.Address]uint64)
```

```
txs := types.Transactions{}
for _, key := range keys {
addr := crypto.PubkeyToAddress(key.PublicKey)
for j := 0; j < int(DefaultTxPoolConfig.GlobalSlots)/len(keys)*2; j++ {
txs = append(txs, transaction(nonces[addr], big.NewInt(100000), key))
nonces[addr]++
}
}
// Import the batch and verify that limits have been enforced
pool.AddBatch(txs)

pending := 0
for _, list := range pool.pending {
pending += list.Len()
}
if pending > int(DefaultTxPoolConfig.GlobalSlots) {
t.Fatalf("total pending transactions overflow allowance: %d > %d", pending,
DefaultTxPoolConfig.GlobalSlots)
}
if err := validateTxPoolInternals(pool); err != nil {
t.Fatalf("pool internal state corrupted: %v", err)
}
}

// Tests that if transactions start being capped, transasctions are also removed from 'all'
func TestTransactionCapClearsFromAll(t *testing.T) {
// Reduce the queue limits to shorten test time
defer func(old uint64) { DefaultTxPoolConfig.AccountSlots = old
}(DefaultTxPoolConfig.AccountSlots)
defer func(old uint64) { DefaultTxPoolConfig.AccountQueue = old
}(DefaultTxPoolConfig.AccountQueue)
defer func(old uint64) { DefaultTxPoolConfig.GlobalSlots = old }(DefaultTxPoolConfig.GlobalSlots)

DefaultTxPoolConfig.AccountSlots = 2
DefaultTxPoolConfig.AccountQueue = 2
DefaultTxPoolConfig.GlobalSlots = 8

// Create the pool to test the limit enforcement with
db, _ := ethdb.NewMemDatabase()
statedb, _ := state.New(common.Hash{}, state.NewDatabase(db))

pool := NewTxPool(DefaultTxPoolConfig, params.TestChainConfig, new(event.TypeMux), func()
```

```
(*state.StateDB, error) { return statedb, nil }, func() *big.Int { return big.NewInt(1000000) })
pool.resetState()

// Create a number of test accounts and fund them
state, _ := pool.currentState()

key, _ := crypto.GenerateKey()
addr := crypto.PubkeyToAddress(key.PublicKey)
state.AddBalance(addr, big.NewInt(1000000))

txs := types.Transactions{}
for j := 0; j < int(DefaultTxPoolConfig.GlobalSlots)*2; j++ {
txs = append(txs, transaction(uint64(j), big.NewInt(100000), key))
}
// Import the batch and verify that limits have been enforced
pool.AddBatch(txs)
if err := validateTxPoolInternals(pool); err != nil {
t.Fatalf("pool internal state corrupted: %v", err)
}
}

// Tests that if the transaction count belonging to multiple accounts go above
// some hard threshold, if they are under the minimum guaranteed slot count then
// the transactions are still kept.
func TestTransactionPendingMinimumAllowance(t *testing.T) {
// Reduce the queue limits to shorten test time
defer func(old uint64) { DefaultTxPoolConfig.GlobalSlots = old }(DefaultTxPoolConfig.GlobalSlots)
DefaultTxPoolConfig.GlobalSlots = 0

// Create the pool to test the limit enforcement with
db, _ := ethdb.NewMemDatabase()
statedb, _ := state.New(common.Hash{}, state.NewDatabase(db))

pool := NewTxPool(DefaultTxPoolConfig, params.TestChainConfig, new(event.TypeMux), func()
(*state.StateDB, error) { return statedb, nil }, func() *big.Int { return big.NewInt(1000000) })
pool.resetState()

// Create a number of test accounts and fund them
state, _ := pool.currentState()

keys := make([]*ecdsa.PrivateKey, 5)
for i := 0; i < len(keys); i++ {
```

```
keys[i], _ = crypto.GenerateKey()
state.AddBalance(crypto.PubkeyToAddress(keys[i].PublicKey), big.NewInt(1000000))
}
// Generate and queue a batch of transactions
nonces := make(map[common.Address]uint64)

txs := types.Transactions{}
for _, key := range keys {
addr := crypto.PubkeyToAddress(key.PublicKey)
for j := 0; j < int(DefaultTxPoolConfig.AccountSlots)*2; j++ {
txs = append(txs, transaction(nonces[addr], big.NewInt(100000), key))
nonces[addr]++
}
}
// Import the batch and verify that limits have been enforced
pool.AddBatch(txs)

for addr, list := range pool.pending {
if list.Len() != int(DefaultTxPoolConfig.AccountSlots) {
t.Errorf("addr %x: total pending transactions mismatch: have %d, want %d", addr, list.Len(),
DefaultTxPoolConfig.AccountSlots)
}
}
if err := validateTxPoolInternals(pool); err != nil {
t.Fatalf("pool internal state corrupted: %v", err)
}
}

// Tests that setting the transaction pool gas price to a higher value correctly
// discards everything cheaper than that and moves any gapped transactions back
// from the pending pool to the queue.
//
// Note, local transactions are never allowed to be dropped.
func TestTransactionPoolRepricing(t *testing.T) {
// Create the pool to test the pricing enforcement with
db, _ := ethdb.NewMemDatabase()
statedb, _ := state.New(common.Hash{}, state.NewDatabase(db))

pool := NewTxPool(DefaultTxPoolConfig, params.TestChainConfig, new(event.TypeMux), func()
(*state.StateDB, error) { return statedb, nil }, func() *big.Int { return big.NewInt(1000000) })
pool.resetState()
```

```
// Create a number of test accounts and fund them
state, _ := pool.currentState()

keys := make([]*ecdsa.PrivateKey, 3)
for i := 0; i < len(keys); i++ {
keys[i], _ = crypto.GenerateKey()
state.AddBalance(crypto.PubkeyToAddress(keys[i].PublicKey), big.NewInt(1000000))
}
// Generate and queue a batch of transactions, both pending and queued
txs := types.Transactions{}

txs = append(txs, pricedTransaction(0, big.NewInt(100000), big.NewInt(2), keys[0]))
txs = append(txs, pricedTransaction(1, big.NewInt(100000), big.NewInt(1), keys[0]))
txs = append(txs, pricedTransaction(2, big.NewInt(100000), big.NewInt(2), keys[0]))

txs = append(txs, pricedTransaction(1, big.NewInt(100000), big.NewInt(2), keys[1]))
txs = append(txs, pricedTransaction(2, big.NewInt(100000), big.NewInt(1), keys[1]))
txs = append(txs, pricedTransaction(3, big.NewInt(100000), big.NewInt(2), keys[1]))

txs = append(txs, pricedTransaction(0, big.NewInt(100000), big.NewInt(1), keys[2]))
pool.SetLocal(txs[len(txs)-1]) // prevent this one from ever being dropped

// Import the batch and that both pending and queued transactions match up
pool.AddBatch(txs)

pending, queued := pool.stats()
if pending != 4 {
t.Fatalf("pending transactions mismatched: have %d, want %d", pending, 4)
}
if queued != 3 {
t.Fatalf("queued transactions mismatched: have %d, want %d", queued, 3)
}
if err := validateTxPoolInternals(pool); err != nil {
t.Fatalf("pool internal state corrupted: %v", err)
}
// Reprice the pool and check that underpriced transactions get dropped
pool.SetGasPrice(big.NewInt(2))

pending, queued = pool.stats()
if pending != 2 {
t.Fatalf("pending transactions mismatched: have %d, want %d", pending, 2)
}
```

```
if queued != 3 {
t.Fatalf("queued transactions mismatched: have %d, want %d", queued, 3)
}
if err := validateTxPoolInternals(pool); err != nil {
t.Fatalf("pool internal state corrupted: %v", err)
}
// Check that we can't add the old transactions back
if err := pool.Add(pricedTransaction(1, big.NewInt(100000), big.NewInt(1), keys[0])); err !=
ErrUnderpriced {
t.Fatalf("adding underpriced pending transaction error mismatch: have %v, want %v", err,
ErrUnderpriced)
}
if err := pool.Add(pricedTransaction(2, big.NewInt(100000), big.NewInt(1), keys[1])); err !=
ErrUnderpriced {
t.Fatalf("adding underpriced queued transaction error mismatch: have %v, want %v", err,
ErrUnderpriced)
}
if err := validateTxPoolInternals(pool); err != nil {
t.Fatalf("pool internal state corrupted: %v", err)
}
// However we can add local underpriced transactions
tx := pricedTransaction(1, big.NewInt(100000), big.NewInt(1), keys[2])

pool.SetLocal(tx) // prevent this one from ever being dropped
if err := pool.Add(tx); err != nil {
t.Fatalf("failed to add underpriced local transaction: %v", err)
}
if pending, _ = pool.stats(); pending != 3 {
t.Fatalf("pending transactions mismatched: have %d, want %d", pending, 3)
}
if err := validateTxPoolInternals(pool); err != nil {
t.Fatalf("pool internal state corrupted: %v", err)
}
}

// Tests that when the pool reaches its global transaction limit, underpriced
// transactions are gradually shifted out for more expensive ones and any gapped
// pending transactions are moved into te queue.
//
// Note, local transactions are never allowed to be dropped.
func TestTransactionPoolUnderpricing(t *testing.T) {
// Reduce the queue limits to shorten test time
```

```
defer func(old uint64) { DefaultTxPoolConfig.GlobalSlots = old }(DefaultTxPoolConfig.GlobalSlots)
DefaultTxPoolConfig.GlobalSlots = 2

defer func(old uint64) { DefaultTxPoolConfig.GlobalQueue = old
}(DefaultTxPoolConfig.GlobalQueue)
DefaultTxPoolConfig.GlobalQueue = 2

// Create the pool to test the pricing enforcement with
db, _ := ethdb.NewMemDatabase()
statedb, _ := state.New(common.Hash{}, state.NewDatabase(db))

pool := NewTxPool(DefaultTxPoolConfig, params.TestChainConfig, new(event.TypeMux), func()
(*state.StateDB, error) { return statedb, nil }, func() *big.Int { return big.NewInt(1000000) })
pool.resetState()

// Create a number of test accounts and fund them
state, _ := pool.currentState()

keys := make([]*ecdsa.PrivateKey, 3)
for i := 0; i < len(keys); i++ {
keys[i], _ = crypto.GenerateKey()
state.AddBalance(crypto.PubkeyToAddress(keys[i].PublicKey), big.NewInt(1000000))
}
// Generate and queue a batch of transactions, both pending and queued
txs := types.Transactions{}

txs = append(txs, pricedTransaction(0, big.NewInt(100000), big.NewInt(1), keys[0]))
txs = append(txs, pricedTransaction(1, big.NewInt(100000), big.NewInt(2), keys[0]))

txs = append(txs, pricedTransaction(1, big.NewInt(100000), big.NewInt(1), keys[1]))

txs = append(txs, pricedTransaction(0, big.NewInt(100000), big.NewInt(1), keys[2]))
pool.SetLocal(txs[len(txs)-1]) // prevent this one from ever being dropped

// Import the batch and that both pending and queued transactions match up
pool.AddBatch(txs)

pending, queued := pool.stats()
if pending != 3 {
t.Fatalf("pending transactions mismatched: have %d, want %d", pending, 3)
}
if queued != 1 {
```

```
t.Fatalf("queued transactions mismatched: have %d, want %d", queued, 1)
}
if err := validateTxPoolInternals(pool); err != nil {
t.Fatalf("pool internal state corrupted: %v", err)
}
// Ensure that adding an underpriced transaction on block limit fails
if err := pool.Add(pricedTransaction(0, big.NewInt(100000), big.NewInt(1), keys[1])); err !=
ErrUnderpriced {
t.Fatalf("adding underpriced pending transaction error mismatch: have %v, want %v", err,
ErrUnderpriced)
}
// Ensure that adding high priced transactions drops cheap ones, but not own
if err := pool.Add(pricedTransaction(0, big.NewInt(100000), big.NewInt(3), keys[1])); err != nil {
t.Fatalf("failed to add well priced transaction: %v", err)
}
if err := pool.Add(pricedTransaction(2, big.NewInt(100000), big.NewInt(4), keys[1])); err != nil {
t.Fatalf("failed to add well priced transaction: %v", err)
}
if err := pool.Add(pricedTransaction(3, big.NewInt(100000), big.NewInt(5), keys[1])); err != nil {
t.Fatalf("failed to add well priced transaction: %v", err)
}
pending, queued = pool.stats()
if pending != 2 {
t.Fatalf("pending transactions mismatched: have %d, want %d", pending, 2)
}
if queued != 2 {
t.Fatalf("queued transactions mismatched: have %d, want %d", queued, 2)
}
if err := validateTxPoolInternals(pool); err != nil {
t.Fatalf("pool internal state corrupted: %v", err)
}
// Ensure that adding local transactions can push out even higher priced ones
tx := pricedTransaction(1, big.NewInt(100000), big.NewInt(0), keys[2])

pool.SetLocal(tx) // prevent this one from ever being dropped
if err := pool.Add(tx); err != nil {
t.Fatalf("failed to add underpriced local transaction: %v", err)
}
pending, queued = pool.stats()
if pending != 2 {
t.Fatalf("pending transactions mismatched: have %d, want %d", pending, 2)
}
```

```
if queued != 2 {
t.Fatalf("queued transactions mismatched: have %d, want %d", queued, 2)
}
if err := validateTxPoolInternals(pool); err != nil {
t.Fatalf("pool internal state corrupted: %v", err)
}
}

// Tests that the pool rejects replacement transactions that don't meet the minimum
// price bump required.
func TestTransactionReplacement(t *testing.T) {
// Create the pool to test the pricing enforcement with
db, _ := ethdb.NewMemDatabase()
statedb, _ := state.New(common.Hash{}, state.NewDatabase(db))

pool := NewTxPool(DefaultTxPoolConfig, params.TestChainConfig, new(event.TypeMux), func()
(*state.StateDB, error) { return statedb, nil }, func() *big.Int { return big.NewInt(1000000) })
pool.resetState()

// Create a a test account to add transactions with
key, _ := crypto.GenerateKey()

state, _ := pool.currentState()
state.AddBalance(crypto.PubkeyToAddress(key.PublicKey), big.NewInt(1000000000))

// Add pending transactions, ensuring the minimum price bump is enforced for replacement (for
ultra low prices too)
price := int64(100)
threshold := (price * (100 + int64(DefaultTxPoolConfig.PriceBump))) / 100

if err := pool.Add(pricedTransaction(0, big.NewInt(100000), big.NewInt(1), key)); err != nil {
t.Fatalf("failed to add original cheap pending transaction: %v", err)
}
if err := pool.Add(pricedTransaction(0, big.NewInt(100001), big.NewInt(1), key)); err !=
ErrReplaceUnderpriced {
t.Fatalf("original cheap pending transaction replacement error mismatch: have %v, want %v", err,
ErrReplaceUnderpriced)
}
if err := pool.Add(pricedTransaction(0, big.NewInt(100000), big.NewInt(2), key)); err != nil {
t.Fatalf("failed to replace original cheap pending transaction: %v", err)
}
```

```
if err := pool.Add(pricedTransaction(0, big.NewInt(100000), big.NewInt(price), key)); err != nil {
t.Fatalf("failed to add original proper pending transaction: %v", err)
}
if err := pool.Add(pricedTransaction(0, big.NewInt(100000), big.NewInt(threshold), key)); err !=
ErrReplaceUnderpriced {
t.Fatalf("original proper pending transaction replacement error mismatch: have %v, want %v", err,
ErrReplaceUnderpriced)
}
if err := pool.Add(pricedTransaction(0, big.NewInt(100000), big.NewInt(threshold+1), key)); err !=
nil {
t.Fatalf("failed to replace original proper pending transaction: %v", err)
}
// Add queued transactions, ensuring the minimum price bump is enforced for replacement (for
ultra low prices too)
if err := pool.Add(pricedTransaction(2, big.NewInt(100000), big.NewInt(1), key)); err != nil {
t.Fatalf("failed to add original queued transaction: %v", err)
}
if err := pool.Add(pricedTransaction(2, big.NewInt(100001), big.NewInt(1), key)); err !=
ErrReplaceUnderpriced {
t.Fatalf("original queued transaction replacement error mismatch: have %v, want %v", err,
ErrReplaceUnderpriced)
}
if err := pool.Add(pricedTransaction(2, big.NewInt(100000), big.NewInt(2), key)); err != nil {
t.Fatalf("failed to replace original queued transaction: %v", err)
}

if err := pool.Add(pricedTransaction(2, big.NewInt(100000), big.NewInt(price), key)); err != nil {
t.Fatalf("failed to add original queued transaction: %v", err)
}
if err := pool.Add(pricedTransaction(2, big.NewInt(100001), big.NewInt(threshold), key)); err !=
ErrReplaceUnderpriced {
t.Fatalf("original queued transaction replacement error mismatch: have %v, want %v", err,
ErrReplaceUnderpriced)
}
if err := pool.Add(pricedTransaction(2, big.NewInt(100000), big.NewInt(threshold+1), key)); err !=
nil {
t.Fatalf("failed to replace original queued transaction: %v", err)
}
if err := validateTxPoolInternals(pool); err != nil {
t.Fatalf("pool internal state corrupted: %v", err)
}
}
```

```
// Benchmarks the speed of validating the contents of the pending queue of the
// transaction pool.
func BenchmarkPendingDemotion100(b *testing.B)   { benchmarkPendingDemotion(b, 100) }
func BenchmarkPendingDemotion1000(b *testing.B)  { benchmarkPendingDemotion(b, 1000) }
func BenchmarkPendingDemotion10000(b *testing.B) { benchmarkPendingDemotion(b, 10000) }

func benchmarkPendingDemotion(b *testing.B, size int) {
// Add a batch of transactions to a pool one by one
pool, key := setupTxPool()
account, _ := deriveSender(transaction(0, big.NewInt(0), key))
state, _ := pool.currentState()
state.AddBalance(account, big.NewInt(1000000))

for i := 0; i < size; i++ {
tx := transaction(uint64(i), big.NewInt(100000), key)
pool.promoteTx(account, tx.Hash(), tx)
}
// Benchmark the speed of pool validation
b.ResetTimer()
for i := 0; i < b.N; i++ {
pool.demoteUnexecutables(state)
}
}

// Benchmarks the speed of scheduling the contents of the future queue of the
// transaction pool.
func BenchmarkFuturePromotion100(b *testing.B)   { benchmarkFuturePromotion(b, 100) }
func BenchmarkFuturePromotion1000(b *testing.B)  { benchmarkFuturePromotion(b, 1000) }
func BenchmarkFuturePromotion10000(b *testing.B) { benchmarkFuturePromotion(b, 10000) }

func benchmarkFuturePromotion(b *testing.B, size int) {
// Add a batch of transactions to a pool one by one
pool, key := setupTxPool()
account, _ := deriveSender(transaction(0, big.NewInt(0), key))
state, _ := pool.currentState()
state.AddBalance(account, big.NewInt(1000000))

for i := 0; i < size; i++ {
tx := transaction(uint64(1+i), big.NewInt(100000), key)
pool.enqueueTx(tx.Hash(), tx)
}
```

```
// Benchmark the speed of pool validation
b.ResetTimer()
for i := 0; i < b.N; i++ {
pool.promoteExecutables(state, nil)
}
}

// Benchmarks the speed of iterative transaction insertion.
func BenchmarkPoolInsert(b *testing.B) {
// Generate a batch of transactions to enqueue into the pool
pool, key := setupTxPool()
account, _ := deriveSender(transaction(0, big.NewInt(0), key))
state, _ := pool.currentState()
state.AddBalance(account, big.NewInt(1000000))

txs := make(types.Transactions, b.N)
for i := 0; i < b.N; i++ {
txs[i] = transaction(uint64(i), big.NewInt(100000), key)
}
// Benchmark importing the transactions into the queue
b.ResetTimer()
for _, tx := range txs {
pool.Add(tx)
}
}

// Benchmarks the speed of batched transaction insertion.
func BenchmarkPoolBatchInsert100(b *testing.B)   { benchmarkPoolBatchInsert(b, 100) }
func BenchmarkPoolBatchInsert1000(b *testing.B)  { benchmarkPoolBatchInsert(b, 1000) }
func BenchmarkPoolBatchInsert10000(b *testing.B) { benchmarkPoolBatchInsert(b, 10000) }

func benchmarkPoolBatchInsert(b *testing.B, size int) {
// Generate a batch of transactions to enqueue into the pool
pool, key := setupTxPool()
account, _ := deriveSender(transaction(0, big.NewInt(0), key))
state, _ := pool.currentState()
state.AddBalance(account, big.NewInt(1000000))

batches := make([]types.Transactions, b.N)
for i := 0; i < b.N; i++ {
batches[i] = make(types.Transactions, size)
for j := 0; j < size; j++ {
```

```
batches[i][j] = transaction(uint64(size*i+j), big.NewInt(100000), key)
}
}
// Benchmark importing the transactions into the queue
b.ResetTimer()
for _, batch := range batches {
pool.AddBatch(batch)
}
}
```

5:F:\git\coin\ethereum\go-ethereum\core\types\block.go

```
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

// Package types contains data types related to Ethereum consensus.
package types

import (
"encoding/binary"
"fmt"
"io"
"math/big"
"sort"
"sync/atomic"
"time"

"github.com/ethereum/go-ethereum/common"
"github.com/ethereum/go-ethereum/common/hexutil"
"github.com/ethereum/go-ethereum/crypto/sha3"
"github.com/ethereum/go-ethereum/rlp"
)

var (
EmptyRootHash  = DeriveSha(Transactions{})
EmptyUncleHash = CalcUncleHash(nil)
)

// A BlockNonce is a 64-bit hash which proves (combined with the
// mix-hash) that a sufficient amount of computation has been carried
// out on a block.
type BlockNonce [8]byte

// EncodeNonce converts the given integer to a block nonce.
```

```
func EncodeNonce(i uint64) BlockNonce {
var n BlockNonce
binary.BigEndian.PutUint64(n[:], i)
return n
}

// Uint64 returns the integer value of a block nonce.
func (n BlockNonce) Uint64() uint64 {
return binary.BigEndian.Uint64(n[:])
}

// MarshalText encodes n as a hex string with 0x prefix.
func (n BlockNonce) MarshalText() ([]byte, error) {
return hexutil.Bytes(n[:]).MarshalText()
}

// UnmarshalText implements encoding.TextUnmarshaler.
func (n *BlockNonce) UnmarshalText(input []byte) error {
return hexutil.UnmarshalFixedText("BlockNonce", input, n[:])
}

//go:generate gencodec -type Header -field-override headerMarshaling -out gen_header_json.go

// Header represents a block header in the Ethereum blockchain.
type Header struct {
ParentHash  common.Hash    `json:"parentHash"       gencodec:"required"`
UncleHash   common.Hash    `json:"sha3Uncles"       gencodec:"required"`
Coinbase    common.Address `json:"miner"            gencodec:"required"`
Root        common.Hash    `json:"stateRoot"        gencodec:"required"`
TxHash      common.Hash    `json:"transactionsRoot" gencodec:"required"`
ReceiptHash common.Hash    `json:"receiptsRoot"     gencodec:"required"`
Bloom       Bloom          `json:"logsBloom"        gencodec:"required"`
Difficulty  *big.Int       `json:"difficulty"       gencodec:"required"`
Number      *big.Int       `json:"number"           gencodec:"required"`
GasLimit    *big.Int       `json:"gasLimit"         gencodec:"required"`
GasUsed     *big.Int       `json:"gasUsed"          gencodec:"required"`
Time        *big.Int       `json:"timestamp"        gencodec:"required"`
Extra       []byte         `json:"extraData"        gencodec:"required"`
MixDigest   common.Hash    `json:"mixHash"          gencodec:"required"`
Nonce       BlockNonce     `json:"nonce"            gencodec:"required"`
}
```

```
// field type overrides for gencodec
type headerMarshaling struct {
Difficulty *hexutil.Big
Number     *hexutil.Big
GasLimit   *hexutil.Big
GasUsed    *hexutil.Big
Time       *hexutil.Big
Extra      hexutil.Bytes
Hash       common.Hash `json:"hash"` // adds call to Hash() in MarshalJSON
}

// Hash returns the block hash of the header, which is simply the keccak256 hash of its
// RLP encoding.
func (h *Header) Hash() common.Hash {
return rlpHash(h)
}

// HashNoNonce returns the hash which is used as input for the proof-of-work search.
func (h *Header) HashNoNonce() common.Hash {
return rlpHash([]interface{}{
h.ParentHash,
h.UncleHash,
h.Coinbase,
h.Root,
h.TxHash,
h.ReceiptHash,
h.Bloom,
h.Difficulty,
h.Number,
h.GasLimit,
h.GasUsed,
h.Time,
h.Extra,
})
}

func rlpHash(x interface{}) (h common.Hash) {
hw := sha3.NewKeccak256()
rlp.Encode(hw, x)
hw.Sum(h[:0])
return h
}
```

```
// Body is a simple (mutable, non-safe) data container for storing and moving
// a block's data contents (transactions and uncles) together.
type Body struct {
Transactions []*Transaction
Uncles       []*Header
}

// Block represents an entire block in the Ethereum blockchain.
type Block struct {
header       *Header
uncles       []*Header
transactions Transactions

// caches
hash atomic.Value
size atomic.Value

// Td is used by package core to store the total difficulty
// of the chain up to and including the block.
td *big.Int

// These fields are used by package eth to track
// inter-peer block relay.
ReceivedAt   time.Time
ReceivedFrom interface{}
}

// DeprecatedTd is an old relic for extracting the TD of a block. It is in the
// code solely to facilitate upgrading the database from the old format to the
// new, after which it should be deleted. Do not use!
func (b *Block) DeprecatedTd() *big.Int {
return b.td
}

// [deprecated by eth/63]
// StorageBlock defines the RLP encoding of a Block stored in the
// state database. The StorageBlock encoding contains fields that
// would otherwise need to be recomputed.
type StorageBlock Block

// "external" block encoding. used for eth protocol, etc.
```

```
type extblock struct {
Header *Header
Txs    []*Transaction
Uncles []*Header
}

// [deprecated by eth/63]
// "storage" block encoding. used for database.
type storageblock struct {
Header *Header
Txs    []*Transaction
Uncles []*Header
TD     *big.Int
}

// NewBlock creates a new block. The input data is copied,
// changes to header and to the field values will not affect the
// block.
//
// The values of TxHash, UncleHash, ReceiptHash and Bloom in header
// are ignored and set to values derived from the given txs, uncles
// and receipts.
func NewBlock(header *Header, txs []*Transaction, uncles []*Header, receipts []*Receipt) *Block {
b := &Block{header: CopyHeader(header), td: new(big.Int)}

// TODO: panic if len(txs) != len(receipts)
if len(txs) == 0 {
b.header.TxHash = EmptyRootHash
} else {
b.header.TxHash = DeriveSha(Transactions(txs))
b.transactions = make(Transactions, len(txs))
copy(b.transactions, txs)
}

if len(receipts) == 0 {
b.header.ReceiptHash = EmptyRootHash
} else {
b.header.ReceiptHash = DeriveSha(Receipts(receipts))
b.header.Bloom = CreateBloom(receipts)
}

if len(uncles) == 0 {
```

```
b.header.UncleHash = EmptyUncleHash
} else {
b.header.UncleHash = CalcUncleHash(uncles)
b.uncles = make([]*Header, len(uncles))
for i := range uncles {
b.uncles[i] = CopyHeader(uncles[i])
}
}

return b
}

// NewBlockWithHeader creates a block with the given header data. The
// header data is copied, changes to header and to the field values
// will not affect the block.
func NewBlockWithHeader(header *Header) *Block {
return &Block{header: CopyHeader(header)}
}

// CopyHeader creates a deep copy of a block header to prevent side effects from
// modifying a header variable.
func CopyHeader(h *Header) *Header {
cpy := *h
if cpy.Time = new(big.Int); h.Time != nil {
cpy.Time.Set(h.Time)
}
if cpy.Difficulty = new(big.Int); h.Difficulty != nil {
cpy.Difficulty.Set(h.Difficulty)
}
if cpy.Number = new(big.Int); h.Number != nil {
cpy.Number.Set(h.Number)
}
if cpy.GasLimit = new(big.Int); h.GasLimit != nil {
cpy.GasLimit.Set(h.GasLimit)
}
if cpy.GasUsed = new(big.Int); h.GasUsed != nil {
cpy.GasUsed.Set(h.GasUsed)
}
if len(h.Extra) > 0 {
cpy.Extra = make([]byte, len(h.Extra))
copy(cpy.Extra, h.Extra)
}
```

```
return &cpy
}

// DecodeRLP decodes the Ethereum
func (b *Block) DecodeRLP(s *rlp.Stream) error {
var eb extblock
_, size, _ := s.Kind()
if err := s.Decode(&eb); err != nil {
return err
}
b.header, b.uncles, b.transactions = eb.Header, eb.Uncles, eb.Txs
b.size.Store(common.StorageSize(rlp.ListSize(size)))
return nil
}

// EncodeRLP serializes b into the Ethereum RLP block format.
func (b *Block) EncodeRLP(w io.Writer) error {
return rlp.Encode(w, extblock{
Header: b.header,
Txs:    b.transactions,
Uncles: b.uncles,
})
}

// [deprecated by eth/63]
func (b *StorageBlock) DecodeRLP(s *rlp.Stream) error {
var sb storageblock
if err := s.Decode(&sb); err != nil {
return err
}
b.header, b.uncles, b.transactions, b.td = sb.Header, sb.Uncles, sb.Txs, sb.TD
return nil
}

// TODO: copies

func (b *Block) Uncles() []*Header          { return b.uncles }
func (b *Block) Transactions() Transactions { return b.transactions }

func (b *Block) Transaction(hash common.Hash) *Transaction {
for _, transaction := range b.transactions {
if transaction.Hash() == hash {
```

```
return transaction
}
}
return nil
}

func (b *Block) Number() *big.Int     { return new(big.Int).Set(b.header.Number) }
func (b *Block) GasLimit() *big.Int   { return new(big.Int).Set(b.header.GasLimit) }
func (b *Block) GasUsed() *big.Int    { return new(big.Int).Set(b.header.GasUsed) }
func (b *Block) Difficulty() *big.Int { return new(big.Int).Set(b.header.Difficulty) }
func (b *Block) Time() *big.Int       { return new(big.Int).Set(b.header.Time) }

func (b *Block) NumberU64() uint64        { return b.header.Number.Uint64() }
func (b *Block) MixDigest() common.Hash   { return b.header.MixDigest }
func (b *Block) Nonce() uint64            { return binary.BigEndian.Uint64(b.header.Nonce[:]) }
func (b *Block) Bloom() Bloom             { return b.header.Bloom }
func (b *Block) Coinbase() common.Address { return b.header.Coinbase }
func (b *Block) Root() common.Hash        { return b.header.Root }
func (b *Block) ParentHash() common.Hash  { return b.header.ParentHash }
func (b *Block) TxHash() common.Hash      { return b.header.TxHash }
func (b *Block) ReceiptHash() common.Hash { return b.header.ReceiptHash }
func (b *Block) UncleHash() common.Hash   { return b.header.UncleHash }
func (b *Block) Extra() []byte            { return common.CopyBytes(b.header.Extra) }

func (b *Block) Header() *Header { return CopyHeader(b.header) }

// Body returns the non-header content of the block.
func (b *Block) Body() *Body { return &Body{b.transactions, b.uncles} }

func (b *Block) HashNoNonce() common.Hash {
return b.header.HashNoNonce()
}

func (b *Block) Size() common.StorageSize {
if size := b.size.Load(); size != nil {
return size.(common.StorageSize)
}
c := writeCounter(0)
rlp.Encode(&c, b)
b.size.Store(common.StorageSize(c))
return common.StorageSize(c)
}
```

```
type writeCounter common.StorageSize

func (c *writeCounter) Write(b []byte) (int, error) {
*c += writeCounter(len(b))
return len(b), nil
}

func CalcUncleHash(uncles []*Header) common.Hash {
return rlpHash(uncles)
}

// WithSeal returns a new block with the data from b but the header replaced with
// the sealed one.
func (b *Block) WithSeal(header *Header) *Block {
cpy := *header

return &Block{
header:       &cpy,
transactions: b.transactions,
uncles:       b.uncles,
}
}

// WithBody returns a new block with the given transaction and uncle contents.
func (b *Block) WithBody(transactions []*Transaction, uncles []*Header) *Block {
block := &Block{
header:       CopyHeader(b.header),
transactions: make([]*Transaction, len(transactions)),
uncles:       make([]*Header, len(uncles)),
}
copy(block.transactions, transactions)
for i := range uncles {
block.uncles[i] = CopyHeader(uncles[i])
}
return block
}

// Hash returns the keccak256 hash of b's header.
// The hash is computed on the first call and cached thereafter.
func (b *Block) Hash() common.Hash {
if hash := b.hash.Load(); hash != nil {
```

```
return hash.(common.Hash)
}
v := b.header.Hash()
b.hash.Store(v)
return v
}

func (b *Block) String() string {
str := fmt.Sprintf(`Block(#%v): Size: %v {
MinerHash: %x
%v
Transactions:
%v
Uncles:
%v
}
`, b.Number(), b.Size(), b.header.HashNoNonce(), b.header, b.transactions, b.uncles)
return str
}

func (h *Header) String() string {
return fmt.Sprintf(`Header(%x):
[
ParentHash:    %x
UncleHash:    %x
Coinbase:    %x
Root:    %x
TxSha    %x
ReceiptSha:    %x
Bloom:    %x
Difficulty:    %v
Number:    %v
GasLimit:    %v
GasUsed:    %v
Time:    %v
Extra:    %s
MixDigest:      %x
Nonce:    %x
]`, h.Hash(), h.ParentHash, h.UncleHash, h.Coinbase, h.Root, h.TxHash, h.ReceiptHash, h.Bloom,
h.Difficulty, h.Number, h.GasLimit, h.GasUsed, h.Time, h.Extra, h.MixDigest, h.Nonce)
}
```

```
type Blocks []*Block

type BlockBy func(b1, b2 *Block) bool

func (self BlockBy) Sort(blocks Blocks) {
bs := blockSorter{
blocks: blocks,
by:     self,
}
sort.Sort(bs)
}

type blockSorter struct {
blocks Blocks
by     func(b1, b2 *Block) bool
}

func (self blockSorter) Len() int { return len(self.blocks) }
func (self blockSorter) Swap(i, j int) {
self.blocks[i], self.blocks[j] = self.blocks[j], self.blocks[i]
}
func (self blockSorter) Less(i, j int) bool { return self.by(self.blocks[i], self.blocks[j]) }

func Number(b1, b2 *Block) bool { return b1.header.Number.Cmp(b2.header.Number) < 0 }

6:F:\git\coin\ethereum\go-ethereum\core\types\block_test.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package types

import (
"bytes"
"fmt"
"math/big"
"reflect"
"testing"

"github.com/ethereum/go-ethereum/common"
"github.com/ethereum/go-ethereum/rlp"
)

// from bcValidBlockTest.json, "SimpleTx"
```

```
func TestBlockEncoding(t *testing.T) {
blockEnc :=
common.FromHex("f90260f901f9a083cafc574e1f51ba9dc0568fc617a08ea2429fb384059c972f13
b19fa1c8dd55a01dcc4de8dec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40d4934794
8888f1f195afa192cfee860698584c030f4c9db1a0ef1552a40b7165c3cd773806b9e0c165b75356e
0314bf0706f279c729f51e017a05fe50b260da6308036625b850b5d6ced6d0a9f814c0688bc91ffb7b
7a3a54b67a0bc37d79753ad738a6dac4921e57392f145d8887476de3f783dfa7edae9283e52b9010
0000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000830200001832fefd8825208845506eb07
80a0bd4472abb6659ebe3ee06ee4d7b72a00a9f4d001caca51342001075469aff49888a13a5a8c8f
2bb1c4f861f85f800a82c35094095e7baea6a6c7c4c2dfeb977efac326af552d870a801ba09bea4c4d
aac7c7c52e093e6a4c35dbbcf8856f1af7b059ba20253e70848d094fa08a8fae537ce25ed8cb5af9ad
ac3f141af69bd515bd2ba031522df09b97dd72b1c0")
var block Block
if err := rlp.DecodeBytes(blockEnc, &block); err != nil {
t.Fatal("decode error: ", err)
}

check := func(f string, got, want interface{}) {
if !reflect.DeepEqual(got, want) {
t.Errorf("%s mismatch: got %v, want %v", f, got, want)
}
}
check("Difficulty", block.Difficulty(), big.NewInt(131072))
check("GasLimit", block.GasLimit(), big.NewInt(3141592))
check("GasUsed", block.GasUsed(), big.NewInt(21000))
check("Coinbase", block.Coinbase(),
common.HexToAddress("8888f1f195afa192cfee860698584c030f4c9db1"))
check("MixDigest", block.MixDigest(),
common.HexToHash("bd4472abb6659ebe3ee06ee4d7b72a00a9f4d001caca51342001075469aff
498"))
check("Root", block.Root(),
common.HexToHash("ef1552a40b7165c3cd773806b9e0c165b75356e0314bf0706f279c729f51e0
17"))
check("Hash", block.Hash(),
common.HexToHash("0a5843ac1cb04865017cb35a57b50b07084e5fcee39b5acadade33149f4fff9
e"))
```

```
check("Nonce", block.Nonce(), uint64(0xa13a5a8c8f2bb1c4))
check("Time", block.Time(), big.NewInt(1426516743))
check("Size", block.Size(), common.StorageSize(len(blockEnc)))

tx1 := NewTransaction(0,
common.HexToAddress("095e7baea6a6c7c4c2dfeb977efac326af552d87"), big.NewInt(10),
big.NewInt(50000), big.NewInt(10), nil)

tx1, _ = tx1.WithSignature(HomesteadSigner{},
common.Hex2Bytes("9bea4c4daac7c7c52e093e6a4c35dbbcf8856f1af7b059ba20253e70848d094
f8a8fae537ce25ed8cb5af9adac3f141af69bd515bd2ba031522df09b97dd72b100"))
fmt.Println(block.Transactions()[0].Hash())
fmt.Println(tx1.data)
fmt.Println(tx1.Hash())
check("len(Transactions)", len(block.Transactions()), 1)
check("Transactions[0].Hash", block.Transactions()[0].Hash(), tx1.Hash())

ourBlockEnc, err := rlp.EncodeToBytes(&block)
if err != nil {
t.Fatal("encode error: ", err)
}
if !bytes.Equal(ourBlockEnc, blockEnc) {
t.Errorf("encoded block mismatch:\ngot:  %x\nwant: %x", ourBlockEnc, blockEnc)
}
}
```

7:F:\git\coin\ethereum\go-ethereum\core\types\bloom9.go

```
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package types

import (
"fmt"
"math/big"

"github.com/ethereum/go-ethereum/common/hexutil"
"github.com/ethereum/go-ethereum/crypto"
)

type bytesBacked interface {
Bytes() []byte
}
```

```
const bloomLength = 256

// Bloom represents a 256 bit bloom filter.
type Bloom [bloomLength]byte

// BytesToBloom converts a byte slice to a bloom filter.
// It panics if b is not of suitable size.
func BytesToBloom(b []byte) Bloom {
var bloom Bloom
bloom.SetBytes(b)
return bloom
}

// SetBytes sets the content of b to the given bytes.
// It panics if d is not of suitable size.
func (b *Bloom) SetBytes(d []byte) {
if len(b) < len(d) {
panic(fmt.Sprintf("bloom bytes too big %d %d", len(b), len(d)))
}
copy(b[bloomLength-len(d):], d)
}

// Add adds d to the filter. Future calls of Test(d) will return true.
func (b *Bloom) Add(d *big.Int) {
bin := new(big.Int).SetBytes(b[:])
bin.Or(bin, bloom9(d.Bytes()))
b.SetBytes(bin.Bytes())
}

// Big converts b to a big integer.
func (b Bloom) Big() *big.Int {
return new(big.Int).SetBytes(b[:])
}

func (b Bloom) Bytes() []byte {
return b[:]
}

func (b Bloom) Test(test *big.Int) bool {
return BloomLookup(b, test)
}
```

```
func (b Bloom) TestBytes(test []byte) bool {
return b.Test(new(big.Int).SetBytes(test))

}

// MarshalText encodes b as a hex string with 0x prefix.
func (b Bloom) MarshalText() ([]byte, error) {
return hexutil.Bytes(b[:]).MarshalText()
}

// UnmarshalText b as a hex string with 0x prefix.
func (b *Bloom) UnmarshalText(input []byte) error {
return hexutil.UnmarshalFixedText("Bloom", input, b[:])
}

func CreateBloom(receipts Receipts) Bloom {
bin := new(big.Int)
for _, receipt := range receipts {
bin.Or(bin, LogsBloom(receipt.Logs))
}

return BytesToBloom(bin.Bytes())
}

func LogsBloom(logs []*Log) *big.Int {
bin := new(big.Int)
for _, log := range logs {
bin.Or(bin, bloom9(log.Address.Bytes()))
for _, b := range log.Topics {
bin.Or(bin, bloom9(b[:]))
}
}

return bin
}

func bloom9(b []byte) *big.Int {
b = crypto.Keccak256(b[:])

r := new(big.Int)
```

```
for i := 0; i < 6; i += 2 {
t := big.NewInt(1)
b := (uint(b[i+1]) + (uint(b[i]) << 8)) & 2047
r.Or(r, t.Lsh(t, b))
}

return r
}

var Bloom9 = bloom9

func BloomLookup(bin Bloom, topic bytesBacked) bool {
bloom := bin.Big()
cmp := bloom9(topic.Bytes()[:])

return bloom.And(bloom, cmp).Cmp(cmp) == 0
}

8:F:\git\coin\ethereum\go-ethereum\core\types\bloom9_test.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package types

import (
"math/big"
"testing"
)

func TestBloom(t *testing.T) {
positive := []string{
"testtest",
"test",
"hallo",
"other",
}
negative := []string{
"tes",
"lo",
}

var bloom Bloom
for _, data := range positive {
```

```
bloom.Add(new(big.Int).SetBytes([]byte(data)))
}

for _, data := range positive {
if !bloom.TestBytes([]byte(data)) {
t.Error("expected", data, "to test true")
}
}
for _, data := range negative {
if bloom.TestBytes([]byte(data)) {
t.Error("did not expect", data, "to test true")
}
}
}

/*
import (
"testing"

"github.com/ethereum/go-ethereum/core/state"
)

func TestBloom9(t *testing.T) {
testCase := []byte("testtest")
bin := LogsBloom([]state.Log{
{testCase, [][]byte{[]byte("hellohello")}, nil},
}).Bytes()
res := BloomLookup(bin, testCase)

if !res {
t.Errorf("Bloom lookup failed")
}
}



func TestAddress(t *testing.T) {
block := &Block{}
block.Coinbase = common.Hex2Bytes("22341ae42d6dd7384bc8584e50419ea3ac75b83f")
fmt.Printf("%x\n", crypto.Keccak256(block.Coinbase))

bin := CreateBloom(block)
fmt.Printf("bin = %x\n", common.LeftPadBytes(bin, 64))
```

```
}
*/

9:F:\git\coin\ethereum\go-ethereum\core\types\derive_sha.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package types

import (
"bytes"

"github.com/ethereum/go-ethereum/common"
"github.com/ethereum/go-ethereum/rlp"
"github.com/ethereum/go-ethereum/trie"
)

type DerivableList interface {
Len() int
GetRlp(i int) []byte
}

func DeriveSha(list DerivableList) common.Hash {
keybuf := new(bytes.Buffer)
trie := new(trie.Trie)
for i := 0; i < list.Len(); i++ {
keybuf.Reset()
rlp.Encode(keybuf, uint(i))
trie.Update(keybuf.Bytes(), list.GetRlp(i))
}
return trie.Hash()
}

10:F:\git\coin\ethereum\go-ethereum\core\types\gen_header_json.go
type Header struct {
ParentHash  common.Hash    `json:"parentHash"       gencodec:"required"`
UncleHash   common.Hash    `json:"sha3Uncles"       gencodec:"required"`
Coinbase    common.Address `json:"miner"            gencodec:"required"`
Root        common.Hash    `json:"stateRoot"        gencodec:"required"`
TxHash      common.Hash    `json:"transactionsRoot" gencodec:"required"`
ReceiptHash common.Hash    `json:"receiptsRoot"     gencodec:"required"`
Bloom       Bloom          `json:"logsBloom"        gencodec:"required"`
Difficulty  *hexutil.Big   `json:"difficulty"       gencodec:"required"`
```

```
Number      *hexutil.Big    `json:"number"          gencodec:"required"`
GasLimit    *hexutil.Big    `json:"gasLimit"        gencodec:"required"`
GasUsed     *hexutil.Big    `json:"gasUsed"         gencodec:"required"`
Time        *hexutil.Big    `json:"timestamp"       gencodec:"required"`
Extra       hexutil.Bytes   `json:"extraData"       gencodec:"required"`
MixDigest   common.Hash     `json:"mixHash"         gencodec:"required"`
Nonce       BlockNonce      `json:"nonce"           gencodec:"required"`
Hash        common.Hash     `json:"hash"`
}
var enc Header
enc.ParentHash = h.ParentHash
enc.UncleHash = h.UncleHash
enc.Coinbase = h.Coinbase
enc.Root = h.Root
enc.TxHash = h.TxHash
enc.ReceiptHash = h.ReceiptHash
enc.Bloom = h.Bloom
enc.Difficulty = (*hexutil.Big)(h.Difficulty)
enc.Number = (*hexutil.Big)(h.Number)
enc.GasLimit = (*hexutil.Big)(h.GasLimit)
enc.GasUsed = (*hexutil.Big)(h.GasUsed)
enc.Time = (*hexutil.Big)(h.Time)
enc.Extra = h.Extra
enc.MixDigest = h.MixDigest
enc.Nonce = h.Nonce
enc.Hash = h.Hash()
return json.Marshal(&enc)
}

func (h *Header) UnmarshalJSON(input []byte) error {
type Header struct {
ParentHash  *common.Hash    `json:"parentHash"      gencodec:"required"`
UncleHash   *common.Hash    `json:"sha3Uncles"      gencodec:"required"`
Coinbase    *common.Address `json:"miner"           gencodec:"required"`
Root        *common.Hash    `json:"stateRoot"       gencodec:"required"`
TxHash      *common.Hash    `json:"transactionsRoot" gencodec:"required"`
ReceiptHash *common.Hash    `json:"receiptsRoot"    gencodec:"required"`
Bloom       *Bloom          `json:"logsBloom"       gencodec:"required"`
Difficulty  *hexutil.Big    `json:"difficulty"      gencodec:"required"`
Number      *hexutil.Big    `json:"number"          gencodec:"required"`
GasLimit    *hexutil.Big    `json:"gasLimit"        gencodec:"required"`
GasUsed     *hexutil.Big    `json:"gasUsed"         gencodec:"required"`
```

```
Time        *hexutil.Big    `json:"timestamp"        gencodec:"required"`
Extra       hexutil.Bytes   `json:"extraData"        gencodec:"required"`
MixDigest   *common.Hash    `json:"mixHash"          gencodec:"required"`
Nonce       *BlockNonce     `json:"nonce"            gencodec:"required"`
}
var dec Header
if err := json.Unmarshal(input, &dec); err != nil {
return err
}
if dec.ParentHash == nil {
return errors.New("missing required field 'parentHash' for Header")
}
h.ParentHash = *dec.ParentHash
if dec.UncleHash == nil {
return errors.New("missing required field 'sha3Uncles' for Header")
}
h.UncleHash = *dec.UncleHash
if dec.Coinbase == nil {
return errors.New("missing required field 'miner' for Header")
}
h.Coinbase = *dec.Coinbase
if dec.Root == nil {
return errors.New("missing required field 'stateRoot' for Header")
}
h.Root = *dec.Root
if dec.TxHash == nil {
return errors.New("missing required field 'transactionsRoot' for Header")
}
h.TxHash = *dec.TxHash
if dec.ReceiptHash == nil {
return errors.New("missing required field 'receiptsRoot' for Header")
}
h.ReceiptHash = *dec.ReceiptHash
if dec.Bloom == nil {
return errors.New("missing required field 'logsBloom' for Header")
}
h.Bloom = *dec.Bloom
if dec.Difficulty == nil {
return errors.New("missing required field 'difficulty' for Header")
}
h.Difficulty = (*big.Int)(dec.Difficulty)
if dec.Number == nil {
```

```
return errors.New("missing required field 'number' for Header")
}
h.Number = (*big.Int)(dec.Number)
if dec.GasLimit == nil {
return errors.New("missing required field 'gasLimit' for Header")
}
h.GasLimit = (*big.Int)(dec.GasLimit)
if dec.GasUsed == nil {
return errors.New("missing required field 'gasUsed' for Header")
}
h.GasUsed = (*big.Int)(dec.GasUsed)
if dec.Time == nil {
return errors.New("missing required field 'timestamp' for Header")
}
h.Time = (*big.Int)(dec.Time)
if dec.Extra == nil {
return errors.New("missing required field 'extraData' for Header")
}
h.Extra = dec.Extra
if dec.MixDigest == nil {
return errors.New("missing required field 'mixHash' for Header")
}
h.MixDigest = *dec.MixDigest
if dec.Nonce == nil {
return errors.New("missing required field 'nonce' for Header")
}
h.Nonce = *dec.Nonce
return nil
}

11:F:\git\coin\ethereum\go-ethereum\core\types\gen_log_json.go
Address     common.Address `json:"address" gencodec:"required"`
Topics      []common.Hash  `json:"topics" gencodec:"required"`
Data        hexutil.Bytes  `json:"data" gencodec:"required"`
BlockNumber hexutil.Uint64 `json:"blockNumber"`
TxHash      common.Hash    `json:"transactionHash" gencodec:"required"`
TxIndex     hexutil.Uint   `json:"transactionIndex" gencodec:"required"`
BlockHash   common.Hash    `json:"blockHash"`
Index       hexutil.Uint   `json:"logIndex" gencodec:"required"`
Removed     bool           `json:"removed"`
}
var enc Log
```

```
enc.Address = l.Address
enc.Topics = l.Topics
enc.Data = l.Data
enc.BlockNumber = hexutil.Uint64(l.BlockNumber)
enc.TxHash = l.TxHash
enc.TxIndex = hexutil.Uint(l.TxIndex)
enc.BlockHash = l.BlockHash
enc.Index = hexutil.Uint(l.Index)
enc.Removed = l.Removed
return json.Marshal(&enc)
}

func (l *Log) UnmarshalJSON(input []byte) error {
type Log struct {
Address     *common.Address `json:"address" gencodec:"required"`
Topics      []common.Hash   `json:"topics" gencodec:"required"`
Data        hexutil.Bytes   `json:"data" gencodec:"required"`
BlockNumber *hexutil.Uint64 `json:"blockNumber"`
TxHash      *common.Hash    `json:"transactionHash" gencodec:"required"`
TxIndex     *hexutil.Uint   `json:"transactionIndex" gencodec:"required"`
BlockHash   *common.Hash    `json:"blockHash"`
Index       *hexutil.Uint   `json:"logIndex" gencodec:"required"`
Removed     *bool           `json:"removed"`
}
var dec Log
if err := json.Unmarshal(input, &dec); err != nil {
return err
}
if dec.Address == nil {
return errors.New("missing required field 'address' for Log")
}
l.Address = *dec.Address
if dec.Topics == nil {
return errors.New("missing required field 'topics' for Log")
}
l.Topics = dec.Topics
if dec.Data == nil {
return errors.New("missing required field 'data' for Log")
}
l.Data = dec.Data
if dec.BlockNumber != nil {
l.BlockNumber = uint64(*dec.BlockNumber)
```

```
}
if dec.TxHash == nil {
return errors.New("missing required field 'transactionHash' for Log")
}
l.TxHash = *dec.TxHash
if dec.TxIndex == nil {
return errors.New("missing required field 'transactionIndex' for Log")
}
l.TxIndex = uint(*dec.TxIndex)
if dec.BlockHash != nil {
l.BlockHash = *dec.BlockHash
}
if dec.Index == nil {
return errors.New("missing required field 'logIndex' for Log")
}
l.Index = uint(*dec.Index)
if dec.Removed != nil {
l.Removed = *dec.Removed
}
return nil
}
```

```
12:F:\git\coin\ethereum\go-ethereum\core\types\gen_receipt_json.go
type Receipt struct {
PostState         hexutil.Bytes  `json:"root"              gencodec:"required"`
CumulativeGasUsed *hexutil.Big   `json:"cumulativeGasUsed" gencodec:"required"`
Bloom             Bloom          `json:"logsBloom"         gencodec:"required"`
Logs              []*Log         `json:"logs"              gencodec:"required"`
TxHash            common.Hash    `json:"transactionHash" gencodec:"required"`
ContractAddress   common.Address `json:"contractAddress"`
GasUsed           *hexutil.Big   `json:"gasUsed" gencodec:"required"`
}
var enc Receipt
enc.PostState = r.PostState
enc.CumulativeGasUsed = (*hexutil.Big)(r.CumulativeGasUsed)
enc.Bloom = r.Bloom
enc.Logs = r.Logs
enc.TxHash = r.TxHash
enc.ContractAddress = r.ContractAddress
enc.GasUsed = (*hexutil.Big)(r.GasUsed)
return json.Marshal(&enc)
}
```

```
func (r *Receipt) UnmarshalJSON(input []byte) error {
type Receipt struct {
PostState         hexutil.Bytes   `json:"root"              gencodec:"required"`
CumulativeGasUsed *hexutil.Big    `json:"cumulativeGasUsed" gencodec:"required"`
Bloom             *Bloom          `json:"logsBloom"         gencodec:"required"`
Logs              []*Log          `json:"logs"              gencodec:"required"`
TxHash            *common.Hash    `json:"transactionHash" gencodec:"required"`
ContractAddress   *common.Address `json:"contractAddress"`
GasUsed           *hexutil.Big    `json:"gasUsed" gencodec:"required"`
}
var dec Receipt
if err := json.Unmarshal(input, &dec); err != nil {
return err
}
if dec.PostState == nil {
return errors.New("missing required field 'root' for Receipt")
}
r.PostState = dec.PostState
if dec.CumulativeGasUsed == nil {
return errors.New("missing required field 'cumulativeGasUsed' for Receipt")
}
r.CumulativeGasUsed = (*big.Int)(dec.CumulativeGasUsed)
if dec.Bloom == nil {
return errors.New("missing required field 'logsBloom' for Receipt")
}
r.Bloom = *dec.Bloom
if dec.Logs == nil {
return errors.New("missing required field 'logs' for Receipt")
}
r.Logs = dec.Logs
if dec.TxHash == nil {
return errors.New("missing required field 'transactionHash' for Receipt")
}
r.TxHash = *dec.TxHash
if dec.ContractAddress != nil {
r.ContractAddress = *dec.ContractAddress
}
if dec.GasUsed == nil {
return errors.New("missing required field 'gasUsed' for Receipt")
}
r.GasUsed = (*big.Int)(dec.GasUsed)
```

```
return nil
}

13:F:\git\coin\ethereum\go-ethereum\core\types\gen_tx_json.go
type txdata struct {
AccountNonce hexutil.Uint64  `json:"nonce"    gencodec:"required"`
Price        *hexutil.Big    `json:"gasPrice" gencodec:"required"`
GasLimit     *hexutil.Big    `json:"gas"      gencodec:"required"`
Recipient    *common.Address `json:"to"       rlp:"nil"`
Amount       *hexutil.Big    `json:"value"    gencodec:"required"`
Payload      hexutil.Bytes   `json:"input"    gencodec:"required"`
V            *hexutil.Big    `json:"v" gencodec:"required"`
R            *hexutil.Big    `json:"r" gencodec:"required"`
S            *hexutil.Big    `json:"s" gencodec:"required"`
Hash         *common.Hash    `json:"hash" rlp:"-"`
}
var enc txdata
enc.AccountNonce = hexutil.Uint64(t.AccountNonce)
enc.Price = (*hexutil.Big)(t.Price)
enc.GasLimit = (*hexutil.Big)(t.GasLimit)
enc.Recipient = t.Recipient
enc.Amount = (*hexutil.Big)(t.Amount)
enc.Payload = t.Payload
enc.V = (*hexutil.Big)(t.V)
enc.R = (*hexutil.Big)(t.R)
enc.S = (*hexutil.Big)(t.S)
enc.Hash = t.Hash
return json.Marshal(&enc)
}

func (t *txdata) UnmarshalJSON(input []byte) error {
type txdata struct {
AccountNonce *hexutil.Uint64 `json:"nonce"    gencodec:"required"`
Price        *hexutil.Big    `json:"gasPrice" gencodec:"required"`
GasLimit     *hexutil.Big    `json:"gas"      gencodec:"required"`
Recipient    *common.Address `json:"to"       rlp:"nil"`
Amount       *hexutil.Big    `json:"value"    gencodec:"required"`
Payload      hexutil.Bytes   `json:"input"    gencodec:"required"`
V            *hexutil.Big    `json:"v" gencodec:"required"`
R            *hexutil.Big    `json:"r" gencodec:"required"`
S            *hexutil.Big    `json:"s" gencodec:"required"`
Hash         *common.Hash    `json:"hash" rlp:"-"`
```

```
}
var dec txdata
if err := json.Unmarshal(input, &dec); err != nil {
return err
}
if dec.AccountNonce == nil {
return errors.New("missing required field 'nonce' for txdata")
}
t.AccountNonce = uint64(*dec.AccountNonce)
if dec.Price == nil {
return errors.New("missing required field 'gasPrice' for txdata")
}
t.Price = (*big.Int)(dec.Price)
if dec.GasLimit == nil {
return errors.New("missing required field 'gas' for txdata")
}
t.GasLimit = (*big.Int)(dec.GasLimit)
if dec.Recipient != nil {
t.Recipient = dec.Recipient
}
if dec.Amount == nil {
return errors.New("missing required field 'value' for txdata")
}
t.Amount = (*big.Int)(dec.Amount)
if dec.Payload == nil {
return errors.New("missing required field 'input' for txdata")
}
t.Payload = dec.Payload
if dec.V == nil {
return errors.New("missing required field 'v' for txdata")
}
t.V = (*big.Int)(dec.V)
if dec.R == nil {
return errors.New("missing required field 'r' for txdata")
}
t.R = (*big.Int)(dec.R)
if dec.S == nil {
return errors.New("missing required field 's' for txdata")
}
t.S = (*big.Int)(dec.S)
if dec.Hash != nil {
t.Hash = dec.Hash
```

```
}
return nil
}

14:F:\git\coin\ethereum\go-ethereum\core\types\log.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package types

import (
"fmt"
"io"

"github.com/ethereum/go-ethereum/common"
"github.com/ethereum/go-ethereum/common/hexutil"
"github.com/ethereum/go-ethereum/rlp"
)

//go:generate gencodec -type Log -field-override logMarshaling -out gen_log_json.go

// Log represents a contract log event. These events are generated by the LOG opcode and
// stored/indexed by the node.
type Log struct {
// Consensus fields:
// address of the contract that generated the event
Address common.Address `json:"address" gencodec:"required"`
// list of topics provided by the contract.
Topics []common.Hash `json:"topics" gencodec:"required"`
// supplied by the contract, usually ABI-encoded
Data []byte `json:"data" gencodec:"required"`

// Derived fields. These fields are filled in by the node
// but not secured by consensus.
// block in which the transaction was included
BlockNumber uint64 `json:"blockNumber"`
// hash of the transaction
TxHash common.Hash `json:"transactionHash" gencodec:"required"`
// index of the transaction in the block
TxIndex uint `json:"transactionIndex" gencodec:"required"`
// hash of the block in which the transaction was included
BlockHash common.Hash `json:"blockHash"`
// index of the log in the receipt
```

```
Index uint `json:"logIndex" gencodec:"required"`

// The Removed field is true if this log was reverted due to a chain reorganisation.
// You must pay attention to this field if you receive logs through a filter query.
Removed bool `json:"removed"`
}

type logMarshaling struct {
Data        hexutil.Bytes
BlockNumber hexutil.Uint64
TxIndex     hexutil.Uint
Index       hexutil.Uint
}

type rlpLog struct {
Address common.Address
Topics  []common.Hash
Data    []byte
}

type rlpStorageLog struct {
Address     common.Address
Topics      []common.Hash
Data        []byte
BlockNumber uint64
TxHash      common.Hash
TxIndex     uint
BlockHash   common.Hash
Index       uint
}

// EncodeRLP implements rlp.Encoder.
func (l *Log) EncodeRLP(w io.Writer) error {
return rlp.Encode(w, rlpLog{Address: l.Address, Topics: l.Topics, Data: l.Data})
}

// DecodeRLP implements rlp.Decoder.
func (l *Log) DecodeRLP(s *rlp.Stream) error {
var dec rlpLog
err := s.Decode(&dec)
if err == nil {
l.Address, l.Topics, l.Data = dec.Address, dec.Topics, dec.Data
```

```
}
return err
}

func (l *Log) String() string {
return fmt.Sprintf(`log: %x %x %x %x %d %x %d`, l.Address, l.Topics, l.Data, l.TxHash, l.TxIndex,
l.BlockHash, l.Index)
}

// LogForStorage is a wrapper around a Log that flattens and parses the entire content of
// a log including non-consensus fields.
type LogForStorage Log

// EncodeRLP implements rlp.Encoder.
func (l *LogForStorage) EncodeRLP(w io.Writer) error {
return rlp.Encode(w, rlpStorageLog{
Address:     l.Address,
Topics:      l.Topics,
Data:        l.Data,
BlockNumber: l.BlockNumber,
TxHash:      l.TxHash,
TxIndex:     l.TxIndex,
BlockHash:   l.BlockHash,
Index:       l.Index,
})
}

// DecodeRLP implements rlp.Decoder.
func (l *LogForStorage) DecodeRLP(s *rlp.Stream) error {
var dec rlpStorageLog
err := s.Decode(&dec)
if err == nil {
*l = LogForStorage{
Address:     dec.Address,
Topics:      dec.Topics,
Data:        dec.Data,
BlockNumber: dec.BlockNumber,
TxHash:      dec.TxHash,
TxIndex:     dec.TxIndex,
BlockHash:   dec.BlockHash,
Index:       dec.Index,
}
```

```
}
return err
}

15:F:\git\coin\ethereum\go-ethereum\core\types\log_test.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package types

import (
"encoding/json"
"fmt"
"reflect"
"testing"

"github.com/davecgh/go-spew/spew"
"github.com/ethereum/go-ethereum/common"
"github.com/ethereum/go-ethereum/common/hexutil"
)

var unmarshalLogTests = map[string]struct {
input     string
want      *Log
wantError error
}{
"ok": {
input:
`{"address":"0xecf8f87f810ecf450940c9f60066b4a7a501d6a7","blockHash":"0x656c34545f90a730
a19008c0e7a7cd4fb3895064b48d6d69761bd5abad681056","blockNumber":"0x1ecfa4","data":"0x
0000000000000000000000000000000000000000000000001a055690d9db80000","logIndex":"0x2"
,"topics":["0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef","0x0000
000000000000000000080b2c9d7cbbf30a1b0fc8983c647d754c6525615"],"transactionHash":"0x
3b198bfd5d2907285af009e9ae84a0ecd63677110d89d7e030251acb87f6487e","transactionIndex"
:"0x3"}`,
want: &Log{
Address:     common.HexToAddress("0xecf8f87f810ecf450940c9f60066b4a7a501d6a7"),
BlockHash:
common.HexToHash("0x656c34545f90a730a19008c0e7a7cd4fb3895064b48d6d69761bd5abad6
81056"),
BlockNumber: 2019236,
Data:
hexutil.MustDecode("0x0000000000000000000000000000000000000000000000001a055690d9db
```

```
80000"),
Index:       2,
TxIndex:     3,
TxHash:
common.HexToHash("0x3b198bfd5d2907285af009e9ae84a0ecd63677110d89d7e030251acb87f6
487e"),
Topics: []common.Hash{
common.HexToHash("0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b
3ef"),
common.HexToHash("0x000000000000000000000000080b2c9d7cbbf30a1b0fc8983c647d754c65
25615"),
},
},
},
"empty data": {
input:
`{"address":"0xecf8f87f810ecf450940c9f60066b4a7a501d6a7","blockHash":"0x656c34545f90a730
a19008c0e7a7cd4fb3895064b48d6d69761bd5abad681056","blockNumber":"0x1ecfa4","data":"0x"
,"logIndex":"0x2","topics":["0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df5
23b3ef","0x000000000000000000000000080b2c9d7cbbf30a1b0fc8983c647d754c6525615"],"trans
actionHash":"0x3b198bfd5d2907285af009e9ae84a0ecd63677110d89d7e030251acb87f6487e","tr
ansactionIndex":"0x3"}`,
want: &Log{
Address:     common.HexToAddress("0xecf8f87f810ecf450940c9f60066b4a7a501d6a7"),
BlockHash:
common.HexToHash("0x656c34545f90a730a19008c0e7a7cd4fb3895064b48d6d69761bd5abad6
81056"),
BlockNumber: 2019236,
Data:        []byte{},
Index:       2,
TxIndex:     3,
TxHash:
common.HexToHash("0x3b198bfd5d2907285af009e9ae84a0ecd63677110d89d7e030251acb87f6
487e"),
Topics: []common.Hash{
common.HexToHash("0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b
3ef"),
common.HexToHash("0x000000000000000000000000080b2c9d7cbbf30a1b0fc8983c647d754c65
25615"),
},
},
},
```

"missing block fields (pending logs)": {
input:
`{"address":"0xecf8f87f810ecf450940c9f60066b4a7a501d6a7","data":"0x","logIndex":"0x0","topics":["0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef"],"transactionHash":"0x3b198bfd5d2907285af009e9ae84a0ecd63677110d89d7e030251acb87f6487e","transactionIndex":"0x3"}`,
want: &Log{
Address: common.HexToAddress("0xecf8f87f810ecf450940c9f60066b4a7a501d6a7"),
BlockHash: common.Hash{},
BlockNumber: 0,
Data: []byte{},
Index: 0,
TxIndex: 3,
TxHash:
common.HexToHash("0x3b198bfd5d2907285af009e9ae84a0ecd63677110d89d7e030251acb87f6487e"),
Topics: []common.Hash{
common.HexToHash("0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef"),
},
},
},
"Removed: true": {
input:
`{"address":"0xecf8f87f810ecf450940c9f60066b4a7a501d6a7","blockHash":"0x656c34545f90a730a19008c0e7a7cd4fb3895064b48d6d69761bd5abad681056","blockNumber":"0x1ecfa4","data":"0x","logIndex":"0x2","topics":["0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef"],"transactionHash":"0x3b198bfd5d2907285af009e9ae84a0ecd63677110d89d7e030251acb87f6487e","transactionIndex":"0x3","removed":true}`,
want: &Log{
Address: common.HexToAddress("0xecf8f87f810ecf450940c9f60066b4a7a501d6a7"),
BlockHash:
common.HexToHash("0x656c34545f90a730a19008c0e7a7cd4fb3895064b48d6d69761bd5abad681056"),
BlockNumber: 2019236,
Data: []byte{},
Index: 2,
TxIndex: 3,
TxHash:
common.HexToHash("0x3b198bfd5d2907285af009e9ae84a0ecd63677110d89d7e030251acb87f6487e"),
Topics: []common.Hash{

```
common.HexToHash("0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b
3ef"),
},
Removed: true,
},
},
"missing data": {
input:
`{"address":"0xecf8f87f810ecf450940c9f60066b4a7a501d6a7","blockHash":"0x656c34545f90a730
a19008c0e7a7cd4fb3895064b48d6d69761bd5abad681056","blockNumber":"0x1ecfa4","logIndex"
:"0x2","topics":["0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef","0
x000000000000000000000000080b2c9d7cbbf30a1b0fc8983c647d754c6525615","0x00000000000
0000000000000000f9dff387dcb5cc4cca5b91adb07a95f54e9f1bb6"],"transactionHash":"0x3b198bfd5
d2907285af009e9ae84a0ecd63677110d89d7e030251acb87f6487e","transactionIndex":"0x3"}`,
wantError: fmt.Errorf("missing required field 'data' for Log"),
},
}

func TestUnmarshalLog(t *testing.T) {
dumper := spew.ConfigState{DisableMethods: true, Indent: "    "}
for name, test := range unmarshalLogTests {
var log *Log
err := json.Unmarshal([]byte(test.input), &log)
checkError(t, name, err, test.wantError)
if test.wantError == nil && err == nil {
if !reflect.DeepEqual(log, test.want) {
t.Errorf("test %q:\nGOT %sWANT %s", name, dumper.Sdump(log), dumper.Sdump(test.want))
}
}
}
}

func checkError(t *testing.T, testname string, got, want error) bool {
if got == nil {
if want != nil {
t.Errorf("test %q: got no error, want %q", testname, want)
return false
}
return true
}
if want == nil {
t.Errorf("test %q: unexpected error %q", testname, got)
```

```
} else if got.Error() != want.Error() {
t.Errorf("test %q: got error %q, want %q", testname, got, want)
}
return false
}

16:F:\git\coin\ethereum\go-ethereum\core\types\receipt.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package types

import (
"fmt"
"io"
"math/big"

"github.com/ethereum/go-ethereum/common"
"github.com/ethereum/go-ethereum/common/hexutil"
"github.com/ethereum/go-ethereum/rlp"
)

//go:generate gencodec -type Receipt -field-override receiptMarshaling -out gen_receipt_json.go

// Receipt represents the results of a transaction.
type Receipt struct {
// Consensus fields
PostState         []byte    `json:"root"              gencodec:"required"`
CumulativeGasUsed *big.Int `json:"cumulativeGasUsed" gencodec:"required"`
Bloom             Bloom     `json:"logsBloom"         gencodec:"required"`
Logs              []*Log    `json:"logs"              gencodec:"required"`

// Implementation fields (don't reorder!)
TxHash          common.Hash    `json:"transactionHash" gencodec:"required"`
ContractAddress common.Address `json:"contractAddress"`
GasUsed         *big.Int       `json:"gasUsed" gencodec:"required"`
}

type receiptMarshaling struct {
PostState         hexutil.Bytes
CumulativeGasUsed *hexutil.Big
GasUsed           *hexutil.Big
}
```

```
// NewReceipt creates a barebone transaction receipt, copying the init fields.
func NewReceipt(root []byte, cumulativeGasUsed *big.Int) *Receipt {
return &Receipt{PostState: common.CopyBytes(root), CumulativeGasUsed:
new(big.Int).Set(cumulativeGasUsed)}
}

// EncodeRLP implements rlp.Encoder, and flattens the consensus fields of a receipt
// into an RLP stream.
func (r *Receipt) EncodeRLP(w io.Writer) error {
return rlp.Encode(w, []interface{}{r.PostState, r.CumulativeGasUsed, r.Bloom, r.Logs})
}

// DecodeRLP implements rlp.Decoder, and loads the consensus fields of a receipt
// from an RLP stream.
func (r *Receipt) DecodeRLP(s *rlp.Stream) error {
var receipt struct {
PostState         []byte
CumulativeGasUsed *big.Int
Bloom             Bloom
Logs              []*Log
}
if err := s.Decode(&receipt); err != nil {
return err
}
r.PostState, r.CumulativeGasUsed, r.Bloom, r.Logs = receipt.PostState,
receipt.CumulativeGasUsed, receipt.Bloom, receipt.Logs
return nil
}

// String implements the Stringer interface.
func (r *Receipt) String() string {
return fmt.Sprintf("receipt{med=%x cgas=%v bloom=%x logs=%v}", r.PostState,
r.CumulativeGasUsed, r.Bloom, r.Logs)
}

// ReceiptForStorage is a wrapper around a Receipt that flattens and parses the
// entire content of a receipt, as opposed to only the consensus fields originally.
type ReceiptForStorage Receipt

// EncodeRLP implements rlp.Encoder, and flattens all content fields of a receipt
// into an RLP stream.
```

```
func (r *ReceiptForStorage) EncodeRLP(w io.Writer) error {
logs := make([]*LogForStorage, len(r.Logs))
for i, log := range r.Logs {
logs[i] = (*LogForStorage)(log)
}
return rlp.Encode(w, []interface{}{r.PostState, r.CumulativeGasUsed, r.Bloom, r.TxHash,
r.ContractAddress, logs, r.GasUsed})
}

// DecodeRLP implements rlp.Decoder, and loads both consensus and implementation
// fields of a receipt from an RLP stream.
func (r *ReceiptForStorage) DecodeRLP(s *rlp.Stream) error {
var receipt struct {
PostState         []byte
CumulativeGasUsed *big.Int
Bloom             Bloom
TxHash            common.Hash
ContractAddress   common.Address
Logs              []*LogForStorage
GasUsed           *big.Int
}
if err := s.Decode(&receipt); err != nil {
return err
}
// Assign the consensus fields
r.PostState, r.CumulativeGasUsed, r.Bloom = receipt.PostState, receipt.CumulativeGasUsed,
receipt.Bloom
r.Logs = make([]*Log, len(receipt.Logs))
for i, log := range receipt.Logs {
r.Logs[i] = (*Log)(log)
}
// Assign the implementation fields
r.TxHash, r.ContractAddress, r.GasUsed = receipt.TxHash, receipt.ContractAddress,
receipt.GasUsed

return nil
}

// Receipts is a wrapper around a Receipt array to implement DerivableList.
type Receipts []*Receipt

// Len returns the number of receipts in this list.
```

```
func (r Receipts) Len() int { return len(r) }

// GetRlp returns the RLP encoding of one receipt from the list.
func (r Receipts) GetRlp(i int) []byte {
bytes, err := rlp.EncodeToBytes(r[i])
if err != nil {
panic(err)
}
return bytes
}

17:F:\git\coin\ethereum\go-ethereum\core\types\transaction.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package types

import (
"container/heap"
"errors"
"fmt"
"io"
"math/big"
"sync/atomic"

"github.com/ethereum/go-ethereum/common"
"github.com/ethereum/go-ethereum/common/hexutil"
"github.com/ethereum/go-ethereum/crypto"
"github.com/ethereum/go-ethereum/rlp"
)

//go:generate gencodec -type txdata -field-override txdataMarshaling -out gen_tx_json.go

var (
ErrInvalidSig = errors.New("invalid transaction v, r, s values")
errNoSigner   = errors.New("missing signing methods")
)

// deriveSigner makes a *best* guess about which signer to use.
func deriveSigner(V *big.Int) Signer {
if V.Sign() != 0 && isProtectedV(V) {
return NewEIP155Signer(deriveChainId(V))
} else {
```

```
return HomesteadSigner{}
}
}

type Transaction struct {
data txdata
// caches
hash atomic.Value
size atomic.Value
from atomic.Value
}

type txdata struct {
AccountNonce uint64          `json:"nonce"    gencodec:"required"`
Price        *big.Int        `json:"gasPrice" gencodec:"required"`
GasLimit     *big.Int        `json:"gas"      gencodec:"required"`
Recipient    *common.Address `json:"to"       rlp:"nil"` // nil means contract creation
Amount       *big.Int        `json:"value"    gencodec:"required"`
Payload      []byte          `json:"input"    gencodec:"required"`

// Signature values
V *big.Int `json:"v" gencodec:"required"`
R *big.Int `json:"r" gencodec:"required"`
S *big.Int `json:"s" gencodec:"required"`

// This is only used when marshaling to JSON.
Hash *common.Hash `json:"hash" rlp:"-"`
}

type txdataMarshaling struct {
AccountNonce hexutil.Uint64
Price        *hexutil.Big
GasLimit     *hexutil.Big
Amount       *hexutil.Big
Payload      hexutil.Bytes
V            *hexutil.Big
R            *hexutil.Big
S            *hexutil.Big
}

func NewTransaction(nonce uint64, to common.Address, amount, gasLimit, gasPrice *big.Int, data
[]byte) *Transaction {
```

```
return newTransaction(nonce, &to, amount, gasLimit, gasPrice, data)
}

func NewContractCreation(nonce uint64, amount, gasLimit, gasPrice *big.Int, data []byte)
*Transaction {
return newTransaction(nonce, nil, amount, gasLimit, gasPrice, data)
}

func newTransaction(nonce uint64, to *common.Address, amount, gasLimit, gasPrice *big.Int,
data []byte) *Transaction {
if len(data) > 0 {
data = common.CopyBytes(data)
}
d := txdata{
AccountNonce: nonce,
Recipient:    to,
Payload:      data,
Amount:       new(big.Int),
GasLimit:     new(big.Int),
Price:        new(big.Int),
V:            new(big.Int),
R:            new(big.Int),
S:            new(big.Int),
}
if amount != nil {
d.Amount.Set(amount)
}
if gasLimit != nil {
d.GasLimit.Set(gasLimit)
}
if gasPrice != nil {
d.Price.Set(gasPrice)
}

return &Transaction{data: d}
}

// ChainId returns which chain id this transaction was signed for (if at all)
func (tx *Transaction) ChainId() *big.Int {
return deriveChainId(tx.data.V)
}
```

```
// Protected returns whether the transaction is protected from replay protection.
func (tx *Transaction) Protected() bool {
return isProtectedV(tx.data.V)
}

func isProtectedV(V *big.Int) bool {
if V.BitLen() <= 8 {
v := V.Uint64()
return v != 27 && v != 28
}
// anything not 27 or 28 are considered unprotected
return true
}

// DecodeRLP implements rlp.Encoder
func (tx *Transaction) EncodeRLP(w io.Writer) error {
return rlp.Encode(w, &tx.data)
}

// DecodeRLP implements rlp.Decoder
func (tx *Transaction) DecodeRLP(s *rlp.Stream) error {
_, size, _ := s.Kind()
err := s.Decode(&tx.data)
if err == nil {
tx.size.Store(common.StorageSize(rlp.ListSize(size)))
}

return err
}

func (tx *Transaction) MarshalJSON() ([]byte, error) {
hash := tx.Hash()
data := tx.data
data.Hash = &hash
return data.MarshalJSON()
}

// UnmarshalJSON decodes the web3 RPC transaction format.
func (tx *Transaction) UnmarshalJSON(input []byte) error {
var dec txdata
if err := dec.UnmarshalJSON(input); err != nil {
return err
```

```
}
var V byte
if isProtectedV(dec.V) {
chainId := deriveChainId(dec.V).Uint64()
V = byte(dec.V.Uint64() - 35 - 2*chainId)
} else {
V = byte(dec.V.Uint64() - 27)
}
if !crypto.ValidateSignatureValues(V, dec.R, dec.S, false) {
return ErrInvalidSig
}
*tx = Transaction{data: dec}
return nil
}

func (tx *Transaction) Data() []byte       { return common.CopyBytes(tx.data.Payload) }
func (tx *Transaction) Gas() *big.Int      { return new(big.Int).Set(tx.data.GasLimit) }
func (tx *Transaction) GasPrice() *big.Int { return new(big.Int).Set(tx.data.Price) }
func (tx *Transaction) Value() *big.Int    { return new(big.Int).Set(tx.data.Amount) }
func (tx *Transaction) Nonce() uint64      { return tx.data.AccountNonce }
func (tx *Transaction) CheckNonce() bool   { return true }

// To returns the recipient address of the transaction.
// It returns nil if the transaction is a contract creation.
func (tx *Transaction) To() *common.Address {
if tx.data.Recipient == nil {
return nil
} else {
to := *tx.data.Recipient
return &to
}
}

// Hash hashes the RLP encoding of tx.
// It uniquely identifies the transaction.
func (tx *Transaction) Hash() common.Hash {
if hash := tx.hash.Load(); hash != nil {
return hash.(common.Hash)
}
v := rlpHash(tx)
tx.hash.Store(v)
return v
```

```
}

// SigHash returns the hash to be signed by the sender.
// It does not uniquely identify the transaction.
func (tx *Transaction) SigHash(signer Signer) common.Hash {
return signer.Hash(tx)
}

func (tx *Transaction) Size() common.StorageSize {
if size := tx.size.Load(); size != nil {
return size.(common.StorageSize)
}
c := writeCounter(0)
rlp.Encode(&c, &tx.data)
tx.size.Store(common.StorageSize(c))
return common.StorageSize(c)
}

// AsMessage returns the transaction as a core.Message.
//
// AsMessage requires a signer to derive the sender.
//
// XXX Rename message to something less arbitrary?
func (tx *Transaction) AsMessage(s Signer) (Message, error) {
msg := Message{
nonce:      tx.data.AccountNonce,
price:      new(big.Int).Set(tx.data.Price),
gasLimit:   new(big.Int).Set(tx.data.GasLimit),
to:         tx.data.Recipient,
amount:     tx.data.Amount,
data:       tx.data.Payload,
checkNonce: true,
}

var err error
msg.from, err = Sender(s, tx)
return msg, err
}

// WithSignature returns a new transaction with the given signature.
// This signature needs to be formatted as described in the yellow paper (v+27).
func (tx *Transaction) WithSignature(signer Signer, sig []byte) (*Transaction, error) {
```

```
return signer.WithSignature(tx, sig)
}

// Cost returns amount + gasprice * gaslimit.
func (tx *Transaction) Cost() *big.Int {
total := new(big.Int).Mul(tx.data.Price, tx.data.GasLimit)
total.Add(total, tx.data.Amount)
return total
}

func (tx *Transaction) RawSignatureValues() (*big.Int, *big.Int, *big.Int) {
return tx.data.V, tx.data.R, tx.data.S
}

func (tx *Transaction) String() string {
var from, to string
if tx.data.V != nil {
// make a best guess about the signer and use that to derive
// the sender.
signer := deriveSigner(tx.data.V)
if f, err := Sender(signer, tx); err != nil { // derive but don't cache
from = "[invalid sender: invalid sig]"
} else {
from = fmt.Sprintf("%x", f[:])
}
} else {
from = "[invalid sender: nil V field]"
}

if tx.data.Recipient == nil {
to = "[contract creation]"
} else {
to = fmt.Sprintf("%x", tx.data.Recipient[:])
}
enc, _ := rlp.EncodeToBytes(&tx.data)
return fmt.Sprintf(`
TX(%x)
Contract: %v
From:     %s
To:       %s
Nonce:    %v
GasPrice: %#x
```

```
GasLimit  %#x
Value:    %#x
Data:     0x%x
V:        %#x
R:        %#x
S:        %#x
Hex:      %x
`,
tx.Hash(),
len(tx.data.Recipient) == 0,
from,
to,
tx.data.AccountNonce,
tx.data.Price,
tx.data.GasLimit,
tx.data.Amount,
tx.data.Payload,
tx.data.V,
tx.data.R,
tx.data.S,
enc,
)
}

// Transaction slice type for basic sorting.
type Transactions []*Transaction

// Len returns the length of s
func (s Transactions) Len() int { return len(s) }

// Swap swaps the i'th and the j'th element in s
func (s Transactions) Swap(i, j int) { s[i], s[j] = s[j], s[i] }

// GetRlp implements Rlpable and returns the i'th element of s in rlp
func (s Transactions) GetRlp(i int) []byte {
enc, _ := rlp.EncodeToBytes(s[i])
return enc
}

// Returns a new set t which is the difference between a to b
func TxDifference(a, b Transactions) (keep Transactions) {
keep = make(Transactions, 0, len(a))
```

```
remove := make(map[common.Hash]struct{})
for _, tx := range b {
remove[tx.Hash()] = struct{}{}
}

for _, tx := range a {
if _, ok := remove[tx.Hash()]; !ok {
keep = append(keep, tx)
}
}

return keep
}

// TxByNonce implements the sort interface to allow sorting a list of transactions
// by their nonces. This is usually only useful for sorting transactions from a
// single account, otherwise a nonce comparison doesn't make much sense.
type TxByNonce Transactions

func (s TxByNonce) Len() int           { return len(s) }
func (s TxByNonce) Less(i, j int) bool { return s[i].data.AccountNonce < s[j].data.AccountNonce }
func (s TxByNonce) Swap(i, j int)      { s[i], s[j] = s[j], s[i] }

// TxByPrice implements both the sort and the heap interface, making it useful
// for all at once sorting as well as individually adding and removing elements.
type TxByPrice Transactions

func (s TxByPrice) Len() int           { return len(s) }
func (s TxByPrice) Less(i, j int) bool { return s[i].data.Price.Cmp(s[j].data.Price) > 0 }
func (s TxByPrice) Swap(i, j int)      { s[i], s[j] = s[j], s[i] }

func (s *TxByPrice) Push(x interface{}) {
*s = append(*s, x.(*Transaction))
}

func (s *TxByPrice) Pop() interface{} {
old := *s
n := len(old)
x := old[n-1]
*s = old[0 : n-1]
return x
```

```
}

// TransactionsByPriceAndNonce represents a set of transactions that can return
// transactions in a profit-maximising sorted order, while supporting removing
// entire batches of transactions for non-executable accounts.
type TransactionsByPriceAndNonce struct {
txs   map[common.Address]Transactions // Per account nonce-sorted list of transactions
heads TxByPrice                       // Next transaction for each unique account (price heap)
}

// NewTransactionsByPriceAndNonce creates a transaction set that can retrieve
// price sorted transactions in a nonce-honouring way.
//
// Note, the input map is reowned so the caller should not interact any more with
// if after providng it to the constructor.
func NewTransactionsByPriceAndNonce(txs map[common.Address]Transactions)
*TransactionsByPriceAndNonce {
// Initialize a price based heap with the head transactions
heads := make(TxByPrice, 0, len(txs))
for acc, accTxs := range txs {
heads = append(heads, accTxs[0])
txs[acc] = accTxs[1:]
}
heap.Init(&heads)

// Assemble and return the transaction set
return &TransactionsByPriceAndNonce{
txs:   txs,
heads: heads,
}
}

// Peek returns the next transaction by price.
func (t *TransactionsByPriceAndNonce) Peek() *Transaction {
if len(t.heads) == 0 {
return nil
}
return t.heads[0]
}

// Shift replaces the current best head with the next one from the same account.
func (t *TransactionsByPriceAndNonce) Shift() {
```

```
signer := deriveSigner(t.heads[0].data.V)
// derive signer but don't cache.
acc, _ := Sender(signer, t.heads[0]) // we only sort valid txs so this cannot fail
if txs, ok := t.txs[acc]; ok && len(txs) > 0 {
t.heads[0], t.txs[acc] = txs[0], txs[1:]
heap.Fix(&t.heads, 0)
} else {
heap.Pop(&t.heads)
}
}

// Pop removes the best transaction, *not* replacing it with the next one from
// the same account. This should be used when a transaction cannot be executed
// and hence all subsequent ones should be discarded from the same account.
func (t *TransactionsByPriceAndNonce) Pop() {
heap.Pop(&t.heads)
}

// Message is a fully derived transaction and implements core.Message
//
// NOTE: In a future PR this will be removed.
type Message struct {
to                  *common.Address
from                common.Address
nonce               uint64
amount, price, gasLimit *big.Int
data                []byte
checkNonce          bool
}

func NewMessage(from common.Address, to *common.Address, nonce uint64, amount, gasLimit,
price *big.Int, data []byte, checkNonce bool) Message {
return Message{
from:       from,
to:         to,
nonce:      nonce,
amount:     amount,
price:      price,
gasLimit:   gasLimit,
data:       data,
checkNonce: checkNonce,
}
```

```
}

func (m Message) From() common.Address { return m.from }
func (m Message) To() *common.Address  { return m.to }
func (m Message) GasPrice() *big.Int   { return m.price }
func (m Message) Value() *big.Int      { return m.amount }
func (m Message) Gas() *big.Int        { return m.gasLimit }
func (m Message) Nonce() uint64        { return m.nonce }
func (m Message) Data() []byte         { return m.data }
func (m Message) CheckNonce() bool     { return m.checkNonce }

18:F:\git\coin\ethereum\go-ethereum\core\types\transaction_signing.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package types

import (
"crypto/ecdsa"
"errors"
"fmt"
"math/big"

"github.com/ethereum/go-ethereum/common"
"github.com/ethereum/go-ethereum/crypto"
"github.com/ethereum/go-ethereum/params"
)

var (
ErrInvalidChainId = errors.New("invalid chaid id for signer")

errAbstractSigner     = errors.New("abstract signer")
abstractSignerAddress = common.HexToAddress("ffffffffffffffffffffffffffffffffffffffff")
)

// sigCache is used to cache the derived sender and contains
// the signer used to derive it.
type sigCache struct {
signer Signer
from   common.Address
}

// MakeSigner returns a Signer based on the given chain config and block number.
```

```
func MakeSigner(config *params.ChainConfig, blockNumber *big.Int) Signer {
var signer Signer
switch {
case config.IsEIP155(blockNumber):
signer = NewEIP155Signer(config.ChainId)
case config.IsHomestead(blockNumber):
signer = HomesteadSigner{}
default:
signer = FrontierSigner{}
}
return signer
}

// SignTx signs the transaction using the given signer and private key
func SignTx(tx *Transaction, s Signer, prv *ecdsa.PrivateKey) (*Transaction, error) {
h := s.Hash(tx)
sig, err := crypto.Sign(h[:], prv)
if err != nil {
return nil, err
}
return s.WithSignature(tx, sig)
}

// Sender derives the sender from the tx using the signer derivation
// functions.

// Sender returns the address derived from the signature (V, R, S) using secp256k1
// elliptic curve and an error if it failed deriving or upon an incorrect
// signature.
//
// Sender may cache the address, allowing it to be used regardless of
// signing method. The cache is invalidated if the cached signer does
// not match the signer used in the current call.
func Sender(signer Signer, tx *Transaction) (common.Address, error) {
if sc := tx.from.Load(); sc != nil {
sigCache := sc.(sigCache)
// If the signer used to derive from in a previous
// call is not the same as used current, invalidate
// the cache.
if sigCache.signer.Equal(signer) {
return sigCache.from, nil
}
```

```
}

pubkey, err := signer.PublicKey(tx)
if err != nil {
return common.Address{}, err
}
var addr common.Address
copy(addr[:], crypto.Keccak256(pubkey[1:])[12:])
tx.from.Store(sigCache{signer: signer, from: addr})
return addr, nil
}

type Signer interface {
// Hash returns the rlp encoded hash for signatures
Hash(tx *Transaction) common.Hash
// PubilcKey returns the public key derived from the signature
PublicKey(tx *Transaction) ([]byte, error)
// WithSignature returns a copy of the transaction with the given signature.
// The signature must be encoded in [R || S || V] format where V is 0 or 1.
WithSignature(tx *Transaction, sig []byte) (*Transaction, error)
// Checks for equality on the signers
Equal(Signer) bool
}

// EIP155Transaction implements TransactionInterface using the
// EIP155 rules
type EIP155Signer struct {
HomesteadSigner

chainId, chainIdMul *big.Int
}

func NewEIP155Signer(chainId *big.Int) EIP155Signer {
if chainId == nil {
chainId = new(big.Int)
}
return EIP155Signer{
chainId:    chainId,
chainIdMul: new(big.Int).Mul(chainId, big.NewInt(2)),
}
}
```

```
func (s EIP155Signer) Equal(s2 Signer) bool {
eip155, ok := s2.(EIP155Signer)
return ok && eip155.chainId.Cmp(s.chainId) == 0
}

func (s EIP155Signer) PublicKey(tx *Transaction) ([]byte, error) {
// if the transaction is not protected fall back to homestead signer
if !tx.Protected() {
return (HomesteadSigner{}).PublicKey(tx)
}

if tx.ChainId().Cmp(s.chainId) != 0 {
return nil, ErrInvalidChainId
}

V := byte(new(big.Int).Sub(tx.data.V, s.chainIdMul).Uint64() - 35)
if !crypto.ValidateSignatureValues(V, tx.data.R, tx.data.S, true) {
return nil, ErrInvalidSig
}
// encode the signature in uncompressed format
R, S := tx.data.R.Bytes(), tx.data.S.Bytes()
sig := make([]byte, 65)
copy(sig[32-len(R):32], R)
copy(sig[64-len(S):64], S)
sig[64] = V

// recover the public key from the signature
hash := s.Hash(tx)
pub, err := crypto.Ecrecover(hash[:], sig)
if err != nil {
return nil, err
}
if len(pub) == 0 || pub[0] != 4 {
return nil, errors.New("invalid public key")
}
return pub, nil
}

// WithSignature returns a new transaction with the given signature. This signature
// needs to be in the [R || S || V] format where V is 0 or 1.
func (s EIP155Signer) WithSignature(tx *Transaction, sig []byte) (*Transaction, error) {
if len(sig) != 65 {
```

```
panic(fmt.Sprintf("wrong size for signature: got %d, want 65", len(sig)))
}

cpy := &Transaction{data: tx.data}
cpy.data.R = new(big.Int).SetBytes(sig[:32])
cpy.data.S = new(big.Int).SetBytes(sig[32:64])
cpy.data.V = new(big.Int).SetBytes([]byte{sig[64]})
if s.chainId.Sign() != 0 {
cpy.data.V = big.NewInt(int64(sig[64] + 35))
cpy.data.V.Add(cpy.data.V, s.chainIdMul)
}
return cpy, nil
}

// Hash returns the hash to be signed by the sender.
// It does not uniquely identify the transaction.
func (s EIP155Signer) Hash(tx *Transaction) common.Hash {
return rlpHash([]interface{}{
tx.data.AccountNonce,
tx.data.Price,
tx.data.GasLimit,
tx.data.Recipient,
tx.data.Amount,
tx.data.Payload,
s.chainId, uint(0), uint(0),
})
}

// HomesteadTransaction implements TransactionInterface using the
// homestead rules.
type HomesteadSigner struct{ FrontierSigner }

func (s HomesteadSigner) Equal(s2 Signer) bool {
_, ok := s2.(HomesteadSigner)
return ok
}

// WithSignature returns a new transaction with the given signature. This signature
// needs to be in the [R || S || V] format where V is 0 or 1.
func (hs HomesteadSigner) WithSignature(tx *Transaction, sig []byte) (*Transaction, error) {
if len(sig) != 65 {
panic(fmt.Sprintf("wrong size for snature: got %d, want 65", len(sig)))
```

```
}
cpy := &Transaction{data: tx.data}
cpy.data.R = new(big.Int).SetBytes(sig[:32])
cpy.data.S = new(big.Int).SetBytes(sig[32:64])
cpy.data.V = new(big.Int).SetBytes([]byte{sig[64] + 27})
return cpy, nil
}

func (hs HomesteadSigner) PublicKey(tx *Transaction) ([]byte, error) {
if tx.data.V.BitLen() > 8 {
return nil, ErrInvalidSig
}
V := byte(tx.data.V.Uint64() - 27)
if !crypto.ValidateSignatureValues(V, tx.data.R, tx.data.S, true) {
return nil, ErrInvalidSig
}
// encode the snature in uncompressed format
r, s := tx.data.R.Bytes(), tx.data.S.Bytes()
sig := make([]byte, 65)
copy(sig[32-len(r):32], r)
copy(sig[64-len(s):64], s)
sig[64] = V

// recover the public key from the snature
hash := hs.Hash(tx)
pub, err := crypto.Ecrecover(hash[:], sig)
if err != nil {
return nil, err
}
if len(pub) == 0 || pub[0] != 4 {
return nil, errors.New("invalid public key")
}
return pub, nil
}

type FrontierSigner struct{}

func (s FrontierSigner) Equal(s2 Signer) bool {
_, ok := s2.(FrontierSigner)
return ok
}
```

```
// WithSignature returns a new transaction with the given signature. This signature
// needs to be in the [R || S || V] format where V is 0 or 1.
func (fs FrontierSigner) WithSignature(tx *Transaction, sig []byte) (*Transaction, error) {
if len(sig) != 65 {
panic(fmt.Sprintf("wrong size for snature: got %d, want 65", len(sig)))
}
cpy := &Transaction{data: tx.data}
cpy.data.R = new(big.Int).SetBytes(sig[:32])
cpy.data.S = new(big.Int).SetBytes(sig[32:64])
cpy.data.V = new(big.Int).SetBytes([]byte{sig[64] + 27})
return cpy, nil
}

// Hash returns the hash to be sned by the sender.
// It does not uniquely identify the transaction.
func (fs FrontierSigner) Hash(tx *Transaction) common.Hash {
return rlpHash([]interface{}{
tx.data.AccountNonce,
tx.data.Price,
tx.data.GasLimit,
tx.data.Recipient,
tx.data.Amount,
tx.data.Payload,
})
}

func (fs FrontierSigner) PublicKey(tx *Transaction) ([]byte, error) {
if tx.data.V.BitLen() > 8 {
return nil, ErrInvalidSig
}

V := byte(tx.data.V.Uint64() - 27)
if !crypto.ValidateSignatureValues(V, tx.data.R, tx.data.S, false) {
return nil, ErrInvalidSig
}
// encode the snature in uncompressed format
r, s := tx.data.R.Bytes(), tx.data.S.Bytes()
sig := make([]byte, 65)
copy(sig[32-len(r):32], r)
copy(sig[64-len(s):64], s)
sig[64] = V
```

```
// recover the public key from the snature
hash := fs.Hash(tx)
pub, err := crypto.Ecrecover(hash[:], sig)
if err != nil {
return nil, err
}
if len(pub) == 0 || pub[0] != 4 {
return nil, errors.New("invalid public key")
}
return pub, nil
}

// deriveChainId derives the chain id from the given v parameter
func deriveChainId(v *big.Int) *big.Int {
if v.BitLen() <= 64 {
v := v.Uint64()
if v == 27 || v == 28 {
return new(big.Int)
}
return new(big.Int).SetUint64((v - 35) / 2)
}
v = new(big.Int).Sub(v, big.NewInt(35))
return v.Div(v, big.NewInt(2))
}

19:F:\git\coin\ethereum\go-ethereum\core\types\transaction_signing_test.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package types

import (
"math/big"
"testing"

"github.com/ethereum/go-ethereum/common"
"github.com/ethereum/go-ethereum/crypto"
"github.com/ethereum/go-ethereum/rlp"
)

func TestEIP155Signing(t *testing.T) {
key, _ := crypto.GenerateKey()
addr := crypto.PubkeyToAddress(key.PublicKey)
```

```
signer := NewEIP155Signer(big.NewInt(18))
tx, err := SignTx(NewTransaction(0, addr, new(big.Int), new(big.Int), new(big.Int), nil), signer, key)
if err != nil {
t.Fatal(err)
}

from, err := Sender(signer, tx)
if err != nil {
t.Fatal(err)
}
if from != addr {
t.Errorf("exected from and address to be equal. Got %x want %x", from, addr)
}
}

func TestEIP155ChainId(t *testing.T) {
key, _ := crypto.GenerateKey()
addr := crypto.PubkeyToAddress(key.PublicKey)

signer := NewEIP155Signer(big.NewInt(18))
tx, err := SignTx(NewTransaction(0, addr, new(big.Int), new(big.Int), new(big.Int), nil), signer, key)
if err != nil {
t.Fatal(err)
}
if !tx.Protected() {
t.Fatal("expected tx to be protected")
}

if tx.ChainId().Cmp(signer.chainId) != 0 {
t.Error("expected chainId to be", signer.chainId, "got", tx.ChainId())
}

tx = NewTransaction(0, addr, new(big.Int), new(big.Int), new(big.Int), nil)
tx, err = SignTx(tx, HomesteadSigner{}, key)
if err != nil {
t.Fatal(err)
}

if tx.Protected() {
t.Error("didn't expect tx to be protected")
}
```

```
if tx.ChainId().Sign() != 0 {
t.Error("expected chain id to be 0 got", tx.ChainId())
}
}

func TestEIP155SigningVitalik(t *testing.T) {
// Test vectors come from http://vitalik.ca/files/eip155_testvec.txt
for i, test := range []struct {
txRlp, addr string
}{
{"f864808504a817c800825208943535353535353535353535353535353535353535808025a0044
852b2a670ade5407e78fb2863c51de9fcb96542a07186fe3aeda6bb8a116da0044852b2a670ade54
07e78fb2863c51de9fcb96542a07186fe3aeda6bb8a116d",
"0xf0f6f18bca1b28cd68e4357452947e021241e9ce"},
{"f864018504a817c80182a410943535353535353535353535353535353535353535018025a0489
efdaa54c0f20c7adf612882df0950f5a951637e0307cdcb4c672f298b8bcaa0489efdaa54c0f20c7adf
612882df0950f5a951637e0307cdcb4c672f298b8bc6",
"0x23ef145a395ea3fa3deb533b8a9e1b4c6c25d112"},
{"f864028504a817c80282f618943535353535353535353535353535353535353535088025a02d7c
5bef027816a800da1736444fb58a807ef4c9603b7848673f7e3a68eb14a5a02d7c5bef027816a800d
a1736444fb58a807ef4c9603b7848673f7e3a68eb14a5",
"0x2e485e0c23b4c3c542628a5f672eeab0ad4888be"},
{"f865038504a817c803830148209435353535353535353535353535353535353535351b8025a02
a80e1ef1d7842f27f2e6be0972bb708b9a135c38860dbe73c27c3486c34f4e0a02a80e1ef1d7842f2
7f2e6be0972bb708b9a135c38860dbe73c27c3486c34f4de",
"0x82a88539669a3fd524d669e858935de5e5410cf0"},
{"f865048504a817c80483019a28943535353535353535353535353535353535353535408025a01
3600b294191fc92924bb3ce4b969c1e7e2bab8f4c93c3fc6d0a51733df3c063a013600b294191fc92
924bb3ce4b969c1e7e2bab8f4c93c3fc6d0a51733df3c060",
"0xf9358f2538fd5ccfeb848b64a96b743fcc930554"},
{"f865058504a817c8058301ec309435353535353535353535353535353535353535357d8025a04
eebf77a833b30520287ddd9478ff51abbdffa30aa90a8d655dba0e8a79ce0c1a04eebf77a833b3052
0287ddd9478ff51abbdffa30aa90a8d655dba0e8a79ce0c1",
"0xa8f7aba377317440bc5b26198a363ad22af1f3a4"},
{"f866068504a817c80683023e3894353535353535353535353535353535353535353581d88025a
06455bf8ea6e7463a1046a0b52804526e119b4bf5136279614e0b1e8e296a4e2fa06455bf8ea6e74
63a1046a0b52804526e119b4bf5136279614e0b1e8e296a4e2d",
"0xf1f571dc362a0e5b2696b8e775f8491d3e50de35"},
{"f867078504a817c807830290409435353535353535353535353535353535353535820157802
5a052f1a9b320cab38e5da8a8f97989383aab0a49165fc91c737310e4f7e9821021a052f1a9b320ca
b38e5da8a8f97989383aab0a49165fc91c737310e4f7e9821021",
```

```
"0xd37922162ab7cea97c97a87551ed02c9a38b7332"},
{"f867088504a817c8088302e2489435353535353535353535353535353535353535353535820200802
5a064b1702d9298fee62dfeccc57d322a463ad55ca201256d01f62b45b2e1c21c12a064b1702d929
8fee62dfeccc57d322a463ad55ca201256d01f62b45b2e1c21c10",
"0x9bddad43f934d313c2b79ca28a432dd2b7281029"},
{"f867098504a817c8098303345094353535353535353535353535353535353535358202d9802
5a052f8f61201b2b11a78d6e866abc9c3db2ae8631fa656bfe5cb53668255367afba052f8f61201b2b
11a78d6e866abc9c3db2ae8631fa656bfe5cb53668255367afb",
"0x3c24d7329e92f84f08556ceb6df1cdb0104ca49f"},
} {
signer := NewEIP155Signer(big.NewInt(1))

var tx *Transaction
err := rlp.DecodeBytes(common.Hex2Bytes(test.txRlp), &tx)
if err != nil {
t.Errorf("%d: %v", i, err)
continue
}

from, err := Sender(signer, tx)
if err != nil {
t.Errorf("%d: %v", i, err)
continue
}

addr := common.HexToAddress(test.addr)
if from != addr {
t.Errorf("%d: expected %x got %x", i, addr, from)
}

}
}

func TestChainId(t *testing.T) {
key, _ := defaultTestKey()

tx := NewTransaction(0, common.Address{}, new(big.Int), new(big.Int), new(big.Int), nil)

var err error
tx, err = SignTx(tx, NewEIP155Signer(big.NewInt(1)), key)
if err != nil {
t.Fatal(err)
```

```
}

_, err = Sender(NewEIP155Signer(big.NewInt(2)), tx)
if err != ErrInvalidChainId {
t.Error("expected error:", ErrInvalidChainId)
}

_, err = Sender(NewEIP155Signer(big.NewInt(1)), tx)
if err != nil {
t.Error("expected no error")
}
}
```

20:F:\git\coin\ethereum\go-ethereum\core\types\transaction_test.go

```
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package types

import (
"bytes"
"crypto/ecdsa"
"encoding/json"
"math/big"
"testing"

"github.com/ethereum/go-ethereum/common"
"github.com/ethereum/go-ethereum/crypto"
"github.com/ethereum/go-ethereum/rlp"
)

// The values in those tests are from the Transaction Tests
// at github.com/ethereum/tests.
var (
emptyTx = NewTransaction(
0,
common.HexToAddress("095e7baea6a6c7c4c2dfeb977efac326af552d87"),
big.NewInt(0), big.NewInt(0), big.NewInt(0),
nil,
)

rightvrsTx, _ = NewTransaction(
3,
```

```
common.HexToAddress("b94f5374fce5edbc8e2a8697c15331677e6ebf0b"),
big.NewInt(10),
big.NewInt(2000),
big.NewInt(1),
common.FromHex("5544"),
).WithSignature(
HomesteadSigner{},
common.Hex2Bytes("98ff921201554726367d2be8c804a7ff89ccf285ebc57dff8ae4c44b9c19ac4a8
887321be575c8095f789dd4c743dfe42c1820f9231f98a962b210e3ac2452a301"),
)
)

func TestTransactionSigHash(t *testing.T) {
if emptyTx.SigHash(HomesteadSigner{}) !=
common.HexToHash("c775b99e7ad12f50d819fcd602390467e28141316969f4b57f0626f74fe3b38
6") {
t.Errorf("empty transaction hash mismatch, got %x", emptyTx.Hash())
}
if rightvrsTx.SigHash(HomesteadSigner{}) !=
common.HexToHash("fe7a79529ed5f7c3375d06b26b186a8644e0e16c373d7a12be41c62d6042b
77a") {
t.Errorf("RightVRS transaction hash mismatch, got %x", rightvrsTx.Hash())
}
}

func TestTransactionEncode(t *testing.T) {
txb, err := rlp.EncodeToBytes(rightvrsTx)
if err != nil {
t.Fatalf("encode error: %v", err)
}
should :=
common.FromHex("f86103018207d094b94f5374fce5edbc8e2a8697c15331677e6ebf0b0a825544
1ca098ff921201554726367d2be8c804a7ff89ccf285ebc57dff8ae4c44b9c19ac4aa08887321be575
c8095f789dd4c743dfe42c1820f9231f98a962b210e3ac2452a3")
if !bytes.Equal(txb, should) {
t.Errorf("encoded RLP mismatch, got %x", txb)
}
}

func decodeTx(data []byte) (*Transaction, error) {
var tx Transaction
t, err := &tx, rlp.Decode(bytes.NewReader(data), &tx)
```

```
return t, err
}

func defaultTestKey() (*ecdsa.PrivateKey, common.Address) {
key, _ :=
crypto.HexToECDSA("45a915e4d060149eb4365960e6a7a45f334393093061116b197e3240065ff
2d8")
addr := crypto.PubkeyToAddress(key.PublicKey)
return key, addr
}

func TestRecipientEmpty(t *testing.T) {
_, addr := defaultTestKey()
tx, err :=
decodeTx(common.Hex2Bytes("f8498080808080011ca09b16de9d5bdee2cf56c28d16275a4da68
cd30273e2525f3959f5d62557489921a0372ebd8fb3345f7db7b5a86d42e24d36e983e259b0664ce
b8c227ec9af572f3d"))
if err != nil {
t.Error(err)
t.FailNow()
}

from, err := Sender(HomesteadSigner{}, tx)
if err != nil {
t.Error(err)
t.FailNow()
}
if addr != from {
t.Error("derived address doesn't match")
}
}

func TestRecipientNormal(t *testing.T) {
_, addr := defaultTestKey()

tx, err :=
decodeTx(common.Hex2Bytes("f85d8080809400000000000000000000000000000000000000080
0011ca0527c0d8f5c63f7b9f41324a7c8a563ee1190bcbf0dac8ab446291bdbf32f5c79a0552c4ef0a
09a04395074dab9ed34d3fbfb843c2f2546cc30fe89ec143ca94ca6"))
if err != nil {
t.Error(err)
```

```
t.FailNow()
}

from, err := Sender(HomesteadSigner{}, tx)
if err != nil {
t.Error(err)
t.FailNow()
}

if addr != from {
t.Error("derived address doesn't match")
}
}

// Tests that transactions can be correctly sorted according to their price in
// decreasing order, but at the same time with increasing nonces when issued by
// the same account.
func TestTransactionPriceNonceSort(t *testing.T) {
// Generate a batch of accounts to start with
keys := make([]*ecdsa.PrivateKey, 25)
for i := 0; i < len(keys); i++ {
keys[i], _ = crypto.GenerateKey()
}

signer := HomesteadSigner{}
// Generate a batch of transactions with overlapping values, but shifted nonces
groups := map[common.Address]Transactions{}
for start, key := range keys {
addr := crypto.PubkeyToAddress(key.PublicKey)
for i := 0; i < 25; i++ {
tx, _ := SignTx(NewTransaction(uint64(start+i), common.Address{}, big.NewInt(100),
big.NewInt(100), big.NewInt(int64(start+i)), nil), signer, key)
groups[addr] = append(groups[addr], tx)
}
}
// Sort the transactions and cross check the nonce ordering
txset := NewTransactionsByPriceAndNonce(groups)

txs := Transactions{}
for {
if tx := txset.Peek(); tx != nil {
txs = append(txs, tx)
```

```
txset.Shift()
}
break
}
for i, txi := range txs {
fromi, _ := Sender(signer, txi)

// Make sure the nonce order is valid
for j, txj := range txs[i+1:] {
fromj, _ := Sender(signer, txj)

if fromi == fromj && txi.Nonce() > txj.Nonce() {
t.Errorf("invalid nonce ordering: tx #%d (A=%x N=%v) < tx #%d (A=%x N=%v)", i, fromi[:4],
txi.Nonce(), i+j, fromj[:4], txj.Nonce())
}
}
// Find the previous and next nonce of this account
prev, next := i-1, i+1
for j := i - 1; j >= 0; j-- {
if fromj, _ := Sender(signer, txs[j]); fromi == fromj {
prev = j
break
}
}
for j := i + 1; j < len(txs); j++ {
if fromj, _ := Sender(signer, txs[j]); fromi == fromj {
next = j
break
}
}
// Make sure that in between the neighbor nonces, the transaction is correctly positioned price
wise
for j := prev + 1; j < next; j++ {
fromj, _ := Sender(signer, txs[j])
if j < i && txs[j].GasPrice().Cmp(txi.GasPrice()) < 0 {
t.Errorf("invalid gasprice ordering: tx #%d (A=%x P=%v) < tx #%d (A=%x P=%v)", j, fromj[:4],
txs[j].GasPrice(), i, fromi[:4], txi.GasPrice())
}
if j > i && txs[j].GasPrice().Cmp(txi.GasPrice()) > 0 {
t.Errorf("invalid gasprice ordering: tx #%d (A=%x P=%v) > tx #%d (A=%x P=%v)", j, fromj[:4],
txs[j].GasPrice(), i, fromi[:4], txi.GasPrice())
}
```

```
}
}
}

// TestTransactionJSON tests serializing/de-serializing to/from JSON.
func TestTransactionJSON(t *testing.T) {
key, err := crypto.GenerateKey()
if err != nil {
t.Fatalf("could not generate key: %v", err)
}
signer := NewEIP155Signer(common.Big1)

for i := uint64(0); i < 25; i++ {
var tx *Transaction
switch i % 2 {
case 0:
tx = NewTransaction(i, common.Address{1}, common.Big0, common.Big1, common.Big2,
[]byte("abcdef"))
case 1:
tx = NewContractCreation(i, common.Big0, common.Big1, common.Big2, []byte("abcdef"))
}

tx, err := SignTx(tx, signer, key)
if err != nil {
t.Fatalf("could not sign transaction: %v", err)
}

data, err := json.Marshal(tx)
if err != nil {
t.Errorf("json.Marshal failed: %v", err)
}

var parsedTx *Transaction
if err := json.Unmarshal(data, &parsedTx); err != nil {
t.Errorf("json.Unmarshal failed: %v", err)
}

// compare nonce, price, gaslimit, recipient, amount, payload, V, R, S
if tx.Hash() != parsedTx.Hash() {
t.Errorf("parsed tx differs from original tx, want %v, got %v", tx, parsedTx)
}
if tx.ChainId().Cmp(parsedTx.ChainId()) != 0 {
```

t.Errorf("invalid chain id, want %d, got %d", tx.ChainId(), parsedTx.ChainId())
}
}
}

21:F:\git\coin\ethereum\go-ethereum\core\types.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package core

import (
"math/big"

"github.com/ethereum/go-ethereum/core/state"
"github.com/ethereum/go-ethereum/core/types"
"github.com/ethereum/go-ethereum/core/vm"
)

// Validator is an interface which defines the standard for block validation. It
// is only responsible for validating block contents, as the header validation is
// done by the specific consensus engines.
//
type Validator interface {
// ValidateBody validates the given block's content.
ValidateBody(block *types.Block) error

// ValidateState validates the given statedb and optionally the receipts and
// gas used.
ValidateState(block, parent *types.Block, state *state.StateDB, receipts types.Receipts, usedGas
*big.Int) error
}

// Processor is an interface for processing blocks using a given initial state.
//
// Process takes the block to be processed and the statedb upon which the
// initial state is based. It should return the receipts generated, amount
// of gas used in the process and return an error if any of the internal rules
// failed.
type Processor interface {
Process(block *types.Block, statedb *state.StateDB, cfg vm.Config) (types.Receipts, []*types.Log,
*big.Int, error)
}

22:F:\git\coin\ethereum\go-ethereum\core\vm\analysis.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package vm

import (
"math/big"

"github.com/ethereum/go-ethereum/common"
)

// destinations stores one map per contract (keyed by hash of code).
// The maps contain an entry for each location of a JUMPDEST
// instruction.
type destinations map[common.Hash][]byte

// has checks whether code has a JUMPDEST at dest.
func (d destinations) has(codehash common.Hash, code []byte, dest *big.Int) bool {
// PC cannot go beyond len(code) and certainly can't be bigger than 63bits.
// Don't bother checking for JUMPDEST in that case.
udest := dest.Uint64()
if dest.BitLen() >= 63 || udest >= uint64(len(code)) {
return false
}

m, analysed := d[codehash]
if !analysed {
m = jumpdests(code)
d[codehash] = m
}
return (m[udest/8] & (1 << (udest % 8))) != 0
}

// jumpdests creates a map that contains an entry for each
// PC location that is a JUMPDEST instruction.
func jumpdests(code []byte) []byte {
m := make([]byte, len(code)/8+1)
for pc := uint64(0); pc < uint64(len(code)); pc++ {
op := OpCode(code[pc])
if op == JUMPDEST {
m[pc/8] |= 1 << (pc % 8)

```
} else if op >= PUSH1 && op <= PUSH32 {
a := uint64(op) - uint64(PUSH1) + 1
pc += a
}
}
return m
}

23:F:\git\coin\ethereum\go-ethereum\core\vm\common.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package vm

import (
"math/big"

"github.com/ethereum/go-ethereum/common"
"github.com/ethereum/go-ethereum/common/math"
)

// calculates the memory size required for a step
func calcMemSize(off, l *big.Int) *big.Int {
if l.Sign() == 0 {
return common.Big0
}

return new(big.Int).Add(off, l)
}

// getData returns a slice from the data based on the start and size and pads
// up to size with zero's. This function is overflow safe.
func getData(data []byte, start, size *big.Int) []byte {
dlen := big.NewInt(int64(len(data)))

s := math.BigMin(start, dlen)
e := math.BigMin(new(big.Int).Add(s, size), dlen)
return common.RightPadBytes(data[s.Uint64():e.Uint64()], int(size.Uint64()))
}

// bigUint64 returns the integer casted to a uint64 and returns whether it
// overflowed in the process.
func bigUint64(v *big.Int) (uint64, bool) {
```

```
return v.Uint64(), v.BitLen() > 64
}

// toWordSize returns the ceiled word size required for memory expansion.
func toWordSize(size uint64) uint64 {
if size > math.MaxUint64-31 {
return math.MaxUint64/32 + 1
}

return (size + 31) / 32
}

func allZero(b []byte) bool {
for _, byte := range b {
if byte != 0 {
return false
}
}
return true
}

24:F:\git\coin\ethereum\go-ethereum\core\vm\contract.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package vm

import (
"math/big"

"github.com/ethereum/go-ethereum/common"
)

// ContractRef is a reference to the contract's backing object
type ContractRef interface {
Address() common.Address
}

// AccountRef implements ContractRef.
//
// Account references are used during EVM initialisation and
// it's primary use is to fetch addresses. Removing this object
// proves difficult because of the cached jump destinations which
```

```
// are fetched from the parent contract (i.e. the caller), which
// is a ContractRef.
type AccountRef common.Address

// Address casts AccountRef to a Address
func (ar AccountRef) Address() common.Address { return (common.Address)(ar) }

// Contract represents an ethereum contract in the state database. It contains
// the the contract code, calling arguments. Contract implements ContractRef
type Contract struct {
// CallerAddress is the result of the caller which initialised this
// contract. However when the "call method" is delegated this value
// needs to be initialised to that of the caller's caller.
CallerAddress common.Address
caller        ContractRef
self          ContractRef

jumpdests destinations // result of JUMPDEST analysis.

Code     []byte
CodeHash common.Hash
CodeAddr *common.Address
Input    []byte

Gas   uint64
value *big.Int

Args []byte

DelegateCall bool
}

// NewContract returns a new contract environment for the execution of EVM.
func NewContract(caller ContractRef, object ContractRef, value *big.Int, gas uint64) *Contract {
c := &Contract{CallerAddress: caller.Address(), caller: caller, self: object, Args: nil}

if parent, ok := caller.(*Contract); ok {
// Reuse JUMPDEST analysis from parent context if available.
c.jumpdests = parent.jumpdests
} else {
c.jumpdests = make(destinations)
}
```

```
// Gas should be a pointer so it can safely be reduced through the run
// This pointer will be off the state transition
c.Gas = gas
// ensures a value is set
c.value = value

return c
}

// AsDelegate sets the contract to be a delegate call and returns the current
// contract (for chaining calls)
func (c *Contract) AsDelegate() *Contract {
c.DelegateCall = true
// NOTE: caller must, at all times be a contract. It should never happen
// that caller is something other than a Contract.
parent := c.caller.(*Contract)
c.CallerAddress = parent.CallerAddress
c.value = parent.value

return c
}

// GetOp returns the n'th element in the contract's byte array
func (c *Contract) GetOp(n uint64) OpCode {
return OpCode(c.GetByte(n))
}

// GetByte returns the n'th byte in the contract's byte array
func (c *Contract) GetByte(n uint64) byte {
if n < uint64(len(c.Code)) {
return c.Code[n]
}

return 0
}

// Caller returns the caller of the contract.
//
// Caller will recursively call caller when the contract is a delegate
// call, including that of caller's caller.
func (c *Contract) Caller() common.Address {
```

```
return c.CallerAddress
}

// UseGas attempts the use gas and subtracts it and returns true on success
func (c *Contract) UseGas(gas uint64) (ok bool) {
if c.Gas < gas {
return false
}
c.Gas -= gas
return true
}

// Address returns the contracts address
func (c *Contract) Address() common.Address {
return c.self.Address()
}

// Value returns the contracts value (sent to it from it's caller)
func (c *Contract) Value() *big.Int {
return c.value
}

// SetCode sets the code to the contract
func (self *Contract) SetCode(hash common.Hash, code []byte) {
self.Code = code
self.CodeHash = hash
}

// SetCallCode sets the code of the contract and address of the backing data
// object
func (self *Contract) SetCallCode(addr *common.Address, hash common.Hash, code []byte) {
self.Code = code
self.CodeHash = hash
self.CodeAddr = addr
}

25:F:\git\coin\ethereum\go-ethereum\core\vm\contracts.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package vm

import (
```

```
"crypto/sha256"
"errors"
"math/big"

"github.com/ethereum/go-ethereum/common"
"github.com/ethereum/go-ethereum/crypto"
"github.com/ethereum/go-ethereum/log"
"github.com/ethereum/go-ethereum/params"
"golang.org/x/crypto/ripemd160"
)

var errBadPrecompileInput = errors.New("bad pre compile input")

// Precompiled contract is the basic interface for native Go contracts. The implementation
// requires a deterministic gas count based on the input size of the Run method of the
// contract.
type PrecompiledContract interface {
RequiredGas(input []byte) uint64  // RequiredPrice calculates the contract gas use
Run(input []byte) ([]byte, error) // Run runs the precompiled contract
}

// PrecompiledContracts contains the default set of ethereum contracts
var PrecompiledContracts = map[common.Address]PrecompiledContract{
common.BytesToAddress([]byte{1}): &ecrecover{},
common.BytesToAddress([]byte{2}): &sha256hash{},
common.BytesToAddress([]byte{3}): &ripemd160hash{},
common.BytesToAddress([]byte{4}): &dataCopy{},
}

// RunPrecompile runs and evaluate the output of a precompiled contract defined in contracts.go
func RunPrecompiledContract(p PrecompiledContract, input []byte, contract *Contract) (ret []byte,
err error) {
gas := p.RequiredGas(input)
if contract.UseGas(gas) {
return p.Run(input)
} else {
return nil, ErrOutOfGas
}
}

// ECRECOVER implemented as a native contract
type ecrecover struct{}
```

```
func (c *ecrecover) RequiredGas(input []byte) uint64 {
return params.EcrecoverGas
}

func (c *ecrecover) Run(in []byte) ([]byte, error) {
const ecRecoverInputLength = 128

in = common.RightPadBytes(in, ecRecoverInputLength)
// "in" is (hash, v, r, s), each 32 bytes
// but for ecrecover we want (r, s, v)

r := new(big.Int).SetBytes(in[64:96])
s := new(big.Int).SetBytes(in[96:128])
v := in[63] - 27

// tighter sig s values in homestead only apply to tx sigs
if !allZero(in[32:63]) || !crypto.ValidateSignatureValues(v, r, s, false) {
log.Trace("ECRECOVER error: v, r or s value invalid")
return nil, nil
}
// v needs to be at the end for libsecp256k1
pubKey, err := crypto.Ecrecover(in[:32], append(in[64:128], v))
// make sure the public key is a valid one
if err != nil {
log.Trace("ECRECOVER failed", "err", err)
return nil, nil
}

// the first byte of pubkey is bitcoin heritage
return common.LeftPadBytes(crypto.Keccak256(pubKey[1:])[12:], 32), nil
}

// SHA256 implemented as a native contract
type sha256hash struct{}

// RequiredGas returns the gas required to execute the pre-compiled contract.
//
// This method does not require any overflow checking as the input size gas costs
// required for anything significant is so high it's impossible to pay for.
func (c *sha256hash) RequiredGas(input []byte) uint64 {
return uint64(len(input)+31)/32*params.Sha256WordGas + params.Sha256Gas
```

```
}
func (c *sha256hash) Run(in []byte) ([]byte, error) {
h := sha256.Sum256(in)
return h[:], nil
}

// RIPMED160 implemented as a native contract
type ripemd160hash struct{}

// RequiredGas returns the gas required to execute the pre-compiled contract.
//
// This method does not require any overflow checking as the input size gas costs
// required for anything significant is so high it's impossible to pay for.
func (c *ripemd160hash) RequiredGas(input []byte) uint64 {
return uint64(len(input)+31)/32*params.Ripemd160WordGas + params.Ripemd160Gas
}
func (c *ripemd160hash) Run(in []byte) ([]byte, error) {
ripemd := ripemd160.New()
ripemd.Write(in)
return common.LeftPadBytes(ripemd.Sum(nil), 32), nil
}

// data copy implemented as a native contract
type dataCopy struct{}

// RequiredGas returns the gas required to execute the pre-compiled contract.
//
// This method does not require any overflow checking as the input size gas costs
// required for anything significant is so high it's impossible to pay for.
func (c *dataCopy) RequiredGas(input []byte) uint64 {
return uint64(len(input)+31)/32*params.IdentityWordGas + params.IdentityGas
}
func (c *dataCopy) Run(in []byte) ([]byte, error) {
return in, nil
}

26:F:\git\coin\ethereum\go-ethereum\core\vm\contracts_test.go

27:F:\git\coin\ethereum\go-ethereum\core\vm\doc.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

/*
```

Package vm implements the Ethereum Virtual Machine.

The vm package implements two EVMs, a byte code VM and a JIT VM. The BC
(Byte Code) VM loops over a set of bytes and executes them according to the set
of rules defined in the Ethereum yellow paper. When the BC VM is invoked it
invokes the JIT VM in a separate goroutine and compiles the byte code in JIT
instructions.

The JIT VM, when invoked, loops around a set of pre-defined instructions until
it either runs of gas, causes an internal error, returns or stops.

The JIT optimiser attempts to pre-compile instructions in to chunks or segments
such as multiple PUSH operations and static JUMPs. It does this by analysing the
opcodes and attempts to match certain regions to known sets. Whenever the
optimiser finds said segments it creates a new instruction and replaces the
first occurrence in the sequence.
*/
package vm

28:F:\git\coin\ethereum\go-ethereum\core\vm\errors.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package vm

import "errors"

var (
ErrOutOfGas   = errors.New("out of gas")
ErrCodeStoreOutOfGas   = errors.New("contract creation code storage out of gas")
ErrDepth   = errors.New("max call depth exceeded")
ErrTraceLimitReached   = errors.New("the number of logs reached the specified limit")
ErrInsufficientBalance = errors.New("insufficient balance for transfer")
)

29:F:\git\coin\ethereum\go-ethereum\core\vm\evm.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package vm

import (
"math/big"
"sync/atomic"

```
"github.com/ethereum/go-ethereum/common"
"github.com/ethereum/go-ethereum/crypto"
"github.com/ethereum/go-ethereum/params"
)

type (
CanTransferFunc func(StateDB, common.Address, *big.Int) bool
TransferFunc    func(StateDB, common.Address, common.Address, *big.Int)
// GetHashFunc returns the nth block hash in the blockchain
// and is used by the BLOCKHASH EVM op code.
GetHashFunc func(uint64) common.Hash
)

// run runs the given contract and takes care of running precompiles with a fallback to the byte code interpreter.
func run(evm *EVM, snapshot int, contract *Contract, input []byte) ([]byte, error) {
if contract.CodeAddr != nil {
precompiledContracts := PrecompiledContracts
if p := precompiledContracts[*contract.CodeAddr]; p != nil {
return RunPrecompiledContract(p, input, contract)
}
}

return evm.interpreter.Run(snapshot, contract, input)
}

// Context provides the EVM with auxiliary information. Once provided
// it shouldn't be modified.
type Context struct {
// CanTransfer returns whether the account contains
// sufficient ether to transfer the value
CanTransfer CanTransferFunc
// Transfer transfers ether from one account to the other
Transfer TransferFunc
// GetHash returns the hash corresponding to n
GetHash GetHashFunc

// Message information
Origin   common.Address // Provides information for ORIGIN
GasPrice *big.Int       // Provides information for GASPRICE
```

```
// Block information
Coinbase    common.Address // Provides information for COINBASE
GasLimit    *big.Int       // Provides information for GASLIMIT
BlockNumber *big.Int       // Provides information for NUMBER
Time        *big.Int       // Provides information for TIME
Difficulty  *big.Int       // Provides information for DIFFICULTY
}

// EVM is the Ethereum Virtual Machine base object and provides
// the necessary tools to run a contract on the given state with
// the provided context. It should be noted that any error
// generated through any of the calls should be considered a
// revert-state-and-consume-all-gas operation, no checks on
// specific errors should ever be performed. The interpreter makes
// sure that any errors generated are to be considered faulty code.
//
// The EVM should never be reused and is not thread safe.
type EVM struct {
// Context provides auxiliary blockchain related information
Context
// StateDB gives access to the underlying state
StateDB StateDB
// Depth is the current call stack
depth int

// chainConfig contains information about the current chain
chainConfig *params.ChainConfig
// chain rules contains the chain rules for the current epoch
chainRules params.Rules
// virtual machine configuration options used to initialise the
// evm.
vmConfig Config
// global (to this context) ethereum virtual machine
// used throughout the execution of the tx.
interpreter *Interpreter
// abort is used to abort the EVM calling operations
// NOTE: must be set atomically
abort int32
}

// NewEVM retutrns a new EVM evmironment. The returned EVM is not thread safe
// and should only ever be used *once*.
```

```
func NewEVM(ctx Context, statedb StateDB, chainConfig *params.ChainConfig, vmConfig Config)
*EVM {
evm := &EVM{
Context:     ctx,
StateDB:     statedb,
vmConfig:    vmConfig,
chainConfig: chainConfig,
chainRules:  chainConfig.Rules(ctx.BlockNumber),
}

evm.interpreter = NewInterpreter(evm, vmConfig)
return evm
}

// Cancel cancels any running EVM operation. This may be called concurrently and
// it's safe to be called multiple times.
func (evm *EVM) Cancel() {
atomic.StoreInt32(&evm.abort, 1)
}

// Call executes the contract associated with the addr with the given input as parameters. It also
handles any
// necessary value transfer required and takes the necessary steps to create accounts and
reverses the state in
// case of an execution error or failed value transfer.
func (evm *EVM) Call(caller ContractRef, addr common.Address, input []byte, gas uint64, value
*big.Int) (ret []byte, leftOverGas uint64, err error) {
if evm.vmConfig.NoRecursion && evm.depth > 0 {
return nil, gas, nil
}

// Depth check execution. Fail if we're trying to execute above the
// limit.
if evm.depth > int(params.CallCreateDepth) {
return nil, gas, ErrDepth
}
if !evm.Context.CanTransfer(evm.StateDB, caller.Address(), value) {
return nil, gas, ErrInsufficientBalance
}

var (
to       = AccountRef(addr)
```

```
snapshot = evm.StateDB.Snapshot()
)
if !evm.StateDB.Exist(addr) {
if PrecompiledContracts[addr] == nil && evm.ChainConfig().IsEIP158(evm.BlockNumber) &&
value.Sign() == 0 {
return nil, gas, nil
}

evm.StateDB.CreateAccount(addr)
}
evm.Transfer(evm.StateDB, caller.Address(), to.Address(), value)

// initialise a new contract and set the code that is to be used by the
// E The contract is a scoped evmironment for this execution context
// only.
contract := NewContract(caller, to, value, gas)
contract.SetCallCode(&addr, evm.StateDB.GetCodeHash(addr), evm.StateDB.GetCode(addr))

ret, err = run(evm, snapshot, contract, input)
// When an error was returned by the EVM or when setting the creation code
// above we revert to the snapshot and consume any gas remaining. Additionally
// when we're in homestead this also counts for code storage gas errors.
if err != nil {
contract.UseGas(contract.Gas)
evm.StateDB.RevertToSnapshot(snapshot)
}
return ret, contract.Gas, err
}

// CallCode executes the contract associated with the addr with the given input as parameters. It
also handles any
// necessary value transfer required and takes the necessary steps to create accounts and
reverses the state in
// case of an execution error or failed value transfer.
//
// CallCode differs from Call in the sense that it executes the given address' code with the caller as
context.
func (evm *EVM) CallCode(caller ContractRef, addr common.Address, input []byte, gas uint64,
value *big.Int) (ret []byte, leftOverGas uint64, err error) {
if evm.vmConfig.NoRecursion && evm.depth > 0 {
return nil, gas, nil
}
```

```
// Depth check execution. Fail if we're trying to execute above the
// limit.
if evm.depth > int(params.CallCreateDepth) {
return nil, gas, ErrDepth
}
if !evm.CanTransfer(evm.StateDB, caller.Address(), value) {
return nil, gas, ErrInsufficientBalance
}

var (
snapshot = evm.StateDB.Snapshot()
to       = AccountRef(caller.Address())
)
// initialise a new contract and set the code that is to be used by the
// E The contract is a scoped evmironment for this execution context
// only.
contract := NewContract(caller, to, value, gas)
contract.SetCallCode(&addr, evm.StateDB.GetCodeHash(addr), evm.StateDB.GetCode(addr))

ret, err = run(evm, snapshot, contract, input)
if err != nil {
contract.UseGas(contract.Gas)
evm.StateDB.RevertToSnapshot(snapshot)
}

return ret, contract.Gas, err
}

// DelegateCall executes the contract associated with the addr with the given input as parameters.
// It reverses the state in case of an execution error.
//
// DelegateCall differs from CallCode in the sense that it executes the given address' code with the
caller as context
// and the caller is set to the caller of the caller.
func (evm *EVM) DelegateCall(caller ContractRef, addr common.Address, input []byte, gas uint64)
(ret []byte, leftOverGas uint64, err error) {
if evm.vmConfig.NoRecursion && evm.depth > 0 {
return nil, gas, nil
}

// Depth check execution. Fail if we're trying to execute above the
```

```
// limit.
if evm.depth > int(params.CallCreateDepth) {
return nil, gas, ErrDepth
}

var (
snapshot = evm.StateDB.Snapshot()
to       = AccountRef(caller.Address())
)

// Iinitialise a new contract and make initialise the delegate values
contract := NewContract(caller, to, nil, gas).AsDelegate()
contract.SetCallCode(&addr, evm.StateDB.GetCodeHash(addr), evm.StateDB.GetCode(addr))

ret, err = run(evm, snapshot, contract, input)
if err != nil {
contract.UseGas(contract.Gas)
evm.StateDB.RevertToSnapshot(snapshot)
}

return ret, contract.Gas, err
}

// Create creates a new contract using code as deployment code.
func (evm *EVM) Create(caller ContractRef, code []byte, gas uint64, value *big.Int) (ret []byte,
contractAddr common.Address, leftOverGas uint64, err error) {
if evm.vmConfig.NoRecursion && evm.depth > 0 {
return nil, common.Address{}, gas, nil
}

// Depth check execution. Fail if we're trying to execute above the
// limit.
if evm.depth > int(params.CallCreateDepth) {
return nil, common.Address{}, gas, ErrDepth
}
if !evm.CanTransfer(evm.StateDB, caller.Address(), value) {
return nil, common.Address{}, gas, ErrInsufficientBalance
}

// Create a new account on the state
nonce := evm.StateDB.GetNonce(caller.Address())
evm.StateDB.SetNonce(caller.Address(), nonce+1)
```

```
snapshot := evm.StateDB.Snapshot()
contractAddr = crypto.CreateAddress(caller.Address(), nonce)
evm.StateDB.CreateAccount(contractAddr)
if evm.ChainConfig().IsEIP158(evm.BlockNumber) {
evm.StateDB.SetNonce(contractAddr, 1)
}
evm.Transfer(evm.StateDB, caller.Address(), contractAddr, value)

// initialise a new contract and set the code that is to be used by the
// E The contract is a scoped evmironment for this execution context
// only.
contract := NewContract(caller, AccountRef(contractAddr), value, gas)
contract.SetCallCode(&contractAddr, crypto.Keccak256Hash(code), code)

ret, err = run(evm, snapshot, contract, nil)
// check whether the max code size has been exceeded
maxCodeSizeExceeded := len(ret) > params.MaxCodeSize
// if the contract creation ran successfully and no errors were returned
// calculate the gas required to store the code. If the code could not
// be stored due to not enough gas set an error and let it be handled
// by the error checking condition below.
if err == nil && !maxCodeSizeExceeded {
createDataGas := uint64(len(ret)) * params.CreateDataGas
if contract.UseGas(createDataGas) {
evm.StateDB.SetCode(contractAddr, ret)
} else {
err = ErrCodeStoreOutOfGas
}
}

// When an error was returned by the EVM or when setting the creation code
// above we revert to the snapshot and consume any gas remaining. Additionally
// when we're in homestead this also counts for code storage gas errors.
if maxCodeSizeExceeded ||
(err != nil && (evm.ChainConfig().IsHomestead(evm.BlockNumber) || err !=
ErrCodeStoreOutOfGas)) {
contract.UseGas(contract.Gas)
evm.StateDB.RevertToSnapshot(snapshot)
}
// If the vm returned with an error the return value should be set to nil.
// This isn't consensus critical but merely to for behaviour reasons such as
```

```
// tests, RPC calls, etc.
if err != nil {
ret = nil
}

return ret, contractAddr, contract.Gas, err
}

// ChainConfig returns the evmironment's chain configuration
func (evm *EVM) ChainConfig() *params.ChainConfig { return evm.chainConfig }

// Interpreter returns the EVM interpreter
func (evm *EVM) Interpreter() *Interpreter { return evm.interpreter }

30:F:\git\coin\ethereum\go-ethereum\core\vm\gas.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package vm

import (
"math/big"

"github.com/ethereum/go-ethereum/params"
)

const (
GasQuickStep   uint64 = 2
GasFastestStep uint64 = 3
GasFastStep    uint64 = 5
GasMidStep     uint64 = 8
GasSlowStep    uint64 = 10
GasExtStep     uint64 = 20

GasReturn       uint64 = 0
GasStop         uint64 = 0
GasContractByte uint64 = 200
)

// calcGas returns the actual gas cost of the call.
//
// The cost of gas was changed during the homestead price change HF. To allow for EIP150
// to be implemented. The returned gas is gas - base * 63 / 64.
```

```
func callGas(gasTable params.GasTable, availableGas, base uint64, callCost *big.Int) (uint64,
error) {
if gasTable.CreateBySuicide > 0 {
availableGas = availableGas - base
gas := availableGas - availableGas/64
// If the bit length exceeds 64 bit we know that the newly calculated "gas" for EIP150
// is smaller than the requested amount. Therefor we return the new gas instead
// of returning an error.
if callCost.BitLen() > 64 || gas < callCost.Uint64() {
return gas, nil
}
}
if callCost.BitLen() > 64 {
return 0, errGasUintOverflow
}

return callCost.Uint64(), nil
}

31:F:\git\coin\ethereum\go-ethereum\core\vm\gas_table.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package vm

import (
"math/big"

"github.com/ethereum/go-ethereum/common"
"github.com/ethereum/go-ethereum/common/math"
"github.com/ethereum/go-ethereum/params"
)

// memoryGasCosts calculates the quadratic gas for memory expansion. It does so
// only for the memory region that is expanded, not the total memory.
func memoryGasCost(mem *Memory, newMemSize uint64) (uint64, error) {

if newMemSize == 0 {
return 0, nil
}
// The maximum that will fit in a uint64 is max_word_count - 1
// anything above that will result in an overflow.
// Additionally, a newMemSize which results in a
```

```
// newMemSizeWords larger than 0x7fffffff will cause the square operation
// to overflow.
// The constant 0xfffffffe0 is the highest number that can be used without
// overflowing the gas calculation
if newMemSize > 0xfffffffe0 {
return 0, errGasUintOverflow
}

newMemSizeWords := toWordSize(newMemSize)
newMemSize = newMemSizeWords * 32

if newMemSize > uint64(mem.Len()) {
square := newMemSizeWords * newMemSizeWords
linCoef := newMemSizeWords * params.MemoryGas
quadCoef := square / params.QuadCoeffDiv
newTotalFee := linCoef + quadCoef

fee := newTotalFee - mem.lastGasCost
mem.lastGasCost = newTotalFee

return fee, nil
}
return 0, nil
}

func constGasFunc(gas uint64) gasFunc {
return func(gt params.GasTable, evm *EVM, contract *Contract, stack *Stack, mem *Memory,
memorySize uint64) (uint64, error) {
return gas, nil
}
}

func gasCalldataCopy(gt params.GasTable, evm *EVM, contract *Contract, stack *Stack, mem
*Memory, memorySize uint64) (uint64, error) {
gas, err := memoryGasCost(mem, memorySize)
if err != nil {
return 0, err
}

var overflow bool
if gas, overflow = math.SafeAdd(gas, GasFastestStep); overflow {
return 0, errGasUintOverflow
```

```
}

words, overflow := bigUint64(stack.Back(2))
if overflow {
return 0, errGasUintOverflow
}

if words, overflow = math.SafeMul(toWordSize(words), params.CopyGas); overflow {
return 0, errGasUintOverflow
}

if gas, overflow = math.SafeAdd(gas, words); overflow {
return 0, errGasUintOverflow
}
return gas, nil
}

func gasSStore(gt params.GasTable, evm *EVM, contract *Contract, stack *Stack, mem *Memory,
memorySize uint64) (uint64, error) {
var (
y, x = stack.Back(1), stack.Back(0)
val  = evm.StateDB.GetState(contract.Address(), common.BigToHash(x))
)
// This checks for 3 scenario's and calculates gas accordingly
// 1. From a zero-value address to a non-zero value         (NEW VALUE)
// 2. From a non-zero value address to a zero-value address (DELETE)
// 3. From a non-zero to a non-zero                         (CHANGE)
if common.EmptyHash(val) && !common.EmptyHash(common.BigToHash(y)) {
// 0 => non 0
return params.SstoreSetGas, nil
} else if !common.EmptyHash(val) && common.EmptyHash(common.BigToHash(y)) {
evm.StateDB.AddRefund(new(big.Int).SetUint64(params.SstoreRefundGas))

return params.SstoreClearGas, nil
} else {
// non 0 => non 0 (or 0 => 0)
return params.SstoreResetGas, nil
}
}

func makeGasLog(n uint64) gasFunc {
return func(gt params.GasTable, evm *EVM, contract *Contract, stack *Stack, mem *Memory,
```

```
memorySize uint64) (uint64, error) {
requestedSize, overflow := bigUint64(stack.Back(1))
if overflow {
return 0, errGasUintOverflow
}

gas, err := memoryGasCost(mem, memorySize)
if err != nil {
return 0, err
}

if gas, overflow = math.SafeAdd(gas, params.LogGas); overflow {
return 0, errGasUintOverflow
}
if gas, overflow = math.SafeAdd(gas, n*params.LogTopicGas); overflow {
return 0, errGasUintOverflow
}

var memorySizeGas uint64
if memorySizeGas, overflow = math.SafeMul(requestedSize, params.LogDataGas); overflow {
return 0, errGasUintOverflow
}
if gas, overflow = math.SafeAdd(gas, memorySizeGas); overflow {
return 0, errGasUintOverflow
}
return gas, nil
}
}

func gasSha3(gt params.GasTable, evm *EVM, contract *Contract, stack *Stack, mem *Memory,
memorySize uint64) (uint64, error) {
var overflow bool
gas, err := memoryGasCost(mem, memorySize)
if err != nil {
return 0, err
}

if gas, overflow = math.SafeAdd(gas, params.Sha3Gas); overflow {
return 0, errGasUintOverflow
}

wordGas, overflow := bigUint64(stack.Back(1))
```

```
if overflow {
return 0, errGasUintOverflow
}
if wordGas, overflow = math.SafeMul(toWordSize(wordGas), params.Sha3WordGas); overflow {
return 0, errGasUintOverflow
}
if gas, overflow = math.SafeAdd(gas, wordGas); overflow {
return 0, errGasUintOverflow
}
return gas, nil
}

func gasCodeCopy(gt params.GasTable, evm *EVM, contract *Contract, stack *Stack, mem
*Memory, memorySize uint64) (uint64, error) {
gas, err := memoryGasCost(mem, memorySize)
if err != nil {
return 0, err
}

var overflow bool
if gas, overflow = math.SafeAdd(gas, GasFastestStep); overflow {
return 0, errGasUintOverflow
}

wordGas, overflow := bigUint64(stack.Back(2))
if overflow {
return 0, errGasUintOverflow
}
if wordGas, overflow = math.SafeMul(toWordSize(wordGas), params.CopyGas); overflow {
return 0, errGasUintOverflow
}
if gas, overflow = math.SafeAdd(gas, wordGas); overflow {
return 0, errGasUintOverflow
}
return gas, nil
}

func gasExtCodeCopy(gt params.GasTable, evm *EVM, contract *Contract, stack *Stack, mem
*Memory, memorySize uint64) (uint64, error) {
gas, err := memoryGasCost(mem, memorySize)
if err != nil {
return 0, err
```

```
}

var overflow bool
if gas, overflow = math.SafeAdd(gas, gt.ExtcodeCopy); overflow {
return 0, errGasUintOverflow
}

wordGas, overflow := bigUint64(stack.Back(3))
if overflow {
return 0, errGasUintOverflow
}

if wordGas, overflow = math.SafeMul(toWordSize(wordGas), params.CopyGas); overflow {
return 0, errGasUintOverflow
}

if gas, overflow = math.SafeAdd(gas, wordGas); overflow {
return 0, errGasUintOverflow
}
return gas, nil
}

func gasMLoad(gt params.GasTable, evm *EVM, contract *Contract, stack *Stack, mem *Memory,
memorySize uint64) (uint64, error) {
var overflow bool
gas, err := memoryGasCost(mem, memorySize)
if err != nil {
return 0, errGasUintOverflow
}
if gas, overflow = math.SafeAdd(gas, GasFastestStep); overflow {
return 0, errGasUintOverflow
}
return gas, nil
}

func gasMStore8(gt params.GasTable, evm *EVM, contract *Contract, stack *Stack, mem
*Memory, memorySize uint64) (uint64, error) {
var overflow bool
gas, err := memoryGasCost(mem, memorySize)
if err != nil {
return 0, errGasUintOverflow
}
```

```
if gas, overflow = math.SafeAdd(gas, GasFastestStep); overflow {
return 0, errGasUintOverflow
}
return gas, nil
}

func gasMStore(gt params.GasTable, evm *EVM, contract *Contract, stack *Stack, mem
*Memory, memorySize uint64) (uint64, error) {
var overflow bool
gas, err := memoryGasCost(mem, memorySize)
if err != nil {
return 0, errGasUintOverflow
}
if gas, overflow = math.SafeAdd(gas, GasFastestStep); overflow {
return 0, errGasUintOverflow
}
return gas, nil
}

func gasCreate(gt params.GasTable, evm *EVM, contract *Contract, stack *Stack, mem *Memory,
memorySize uint64) (uint64, error) {
var overflow bool
gas, err := memoryGasCost(mem, memorySize)
if err != nil {
return 0, err
}
if gas, overflow = math.SafeAdd(gas, params.CreateGas); overflow {
return 0, errGasUintOverflow
}
return gas, nil
}

func gasBalance(gt params.GasTable, evm *EVM, contract *Contract, stack *Stack, mem
*Memory, memorySize uint64) (uint64, error) {
return gt.Balance, nil
}

func gasExtCodeSize(gt params.GasTable, evm *EVM, contract *Contract, stack *Stack, mem
*Memory, memorySize uint64) (uint64, error) {
return gt.ExtcodeSize, nil
}
```

```
func gasSLoad(gt params.GasTable, evm *EVM, contract *Contract, stack *Stack, mem *Memory,
memorySize uint64) (uint64, error) {
return gt.SLoad, nil
}

func gasExp(gt params.GasTable, evm *EVM, contract *Contract, stack *Stack, mem *Memory,
memorySize uint64) (uint64, error) {
expByteLen := uint64((stack.data[stack.len()-2].BitLen() + 7) / 8)

var (
gas      = expByteLen * gt.ExpByte // no overflow check required. Max is 256 * ExpByte gas
overflow bool
)
if gas, overflow = math.SafeAdd(gas, GasSlowStep); overflow {
return 0, errGasUintOverflow
}
return gas, nil
}

func gasCall(gt params.GasTable, evm *EVM, contract *Contract, stack *Stack, mem *Memory,
memorySize uint64) (uint64, error) {
var (
gas            = gt.Calls
transfersValue = stack.Back(2).Sign() != 0
address        = common.BigToAddress(stack.Back(1))
eip158         = evm.ChainConfig().IsEIP158(evm.BlockNumber)
)
if eip158 {
if evm.StateDB.Empty(address) && transfersValue {
gas += params.CallNewAccountGas
}
} else if !evm.StateDB.Exist(address) {
gas += params.CallNewAccountGas
}
if transfersValue {
gas += params.CallValueTransferGas
}
memoryGas, err := memoryGasCost(mem, memorySize)
if err != nil {
return 0, err
}
var overflow bool
```

```
if gas, overflow = math.SafeAdd(gas, memoryGas); overflow {
return 0, errGasUintOverflow
}

cg, err := callGas(gt, contract.Gas, gas, stack.Back(0))
if err != nil {
return 0, err
}
// Replace the stack item with the new gas calculation. This means that
// either the original item is left on the stack or the item is replaced by:
// (availableGas - gas) * 63 / 64
// We replace the stack item so that it's available when the opCall instruction is
// called. This information is otherwise lost due to the dependency on *current*
// available gas.
stack.data[stack.len()-1] = new(big.Int).SetUint64(cg)

if gas, overflow = math.SafeAdd(gas, cg); overflow {
return 0, errGasUintOverflow
}
return gas, nil
}

func gasCallCode(gt params.GasTable, evm *EVM, contract *Contract, stack *Stack, mem
*Memory, memorySize uint64) (uint64, error) {
gas := gt.Calls
if stack.Back(2).Sign() != 0 {
gas += params.CallValueTransferGas
}
memoryGas, err := memoryGasCost(mem, memorySize)
if err != nil {
return 0, err
}
var overflow bool
if gas, overflow = math.SafeAdd(gas, memoryGas); overflow {
return 0, errGasUintOverflow
}

cg, err := callGas(gt, contract.Gas, gas, stack.Back(0))
if err != nil {
return 0, err
}
// Replace the stack item with the new gas calculation. This means that
```

```
// either the original item is left on the stack or the item is replaced by:
// (availableGas - gas) * 63 / 64
// We replace the stack item so that it's available when the opCall instruction is
// called. This information is otherwise lost due to the dependency on *current*
// available gas.
stack.data[stack.len()-1] = new(big.Int).SetUint64(cg)

if gas, overflow = math.SafeAdd(gas, cg); overflow {
return 0, errGasUintOverflow
}
return gas, nil
}

func gasReturn(gt params.GasTable, evm *EVM, contract *Contract, stack *Stack, mem *Memory,
memorySize uint64) (uint64, error) {
return memoryGasCost(mem, memorySize)
}

func gasSuicide(gt params.GasTable, evm *EVM, contract *Contract, stack *Stack, mem
*Memory, memorySize uint64) (uint64, error) {
var gas uint64
// EIP150 homestead gas reprice fork:
if evm.ChainConfig().IsEIP150(evm.BlockNumber) {
gas = gt.Suicide
var (
address = common.BigToAddress(stack.Back(0))
eip158  = evm.ChainConfig().IsEIP158(evm.BlockNumber)
)

if eip158 {
// if empty and transfers value
if evm.StateDB.Empty(address) && evm.StateDB.GetBalance(contract.Address()).Sign() != 0 {
gas += gt.CreateBySuicide
}
} else if !evm.StateDB.Exist(address) {
gas += gt.CreateBySuicide
}
}

if !evm.StateDB.HasSuicided(contract.Address()) {
evm.StateDB.AddRefund(new(big.Int).SetUint64(params.SuicideRefundGas))
}
```

```
return gas, nil
}

func gasDelegateCall(gt params.GasTable, evm *EVM, contract *Contract, stack *Stack, mem
*Memory, memorySize uint64) (uint64, error) {
gas, err := memoryGasCost(mem, memorySize)
if err != nil {
return 0, err
}
var overflow bool
if gas, overflow = math.SafeAdd(gas, gt.Calls); overflow {
return 0, errGasUintOverflow
}

cg, err := callGas(gt, contract.Gas, gas, stack.Back(0))
if err != nil {
return 0, err
}
// Replace the stack item with the new gas calculation. This means that
// either the original item is left on the stack or the item is replaced by:
// (availableGas - gas) * 63 / 64
// We replace the stack item so that it's available when the opCall instruction is
// called.
stack.data[stack.len()-1] = new(big.Int).SetUint64(cg)

if gas, overflow = math.SafeAdd(gas, cg); overflow {
return 0, errGasUintOverflow
}
return gas, nil
}

func gasPush(gt params.GasTable, evm *EVM, contract *Contract, stack *Stack, mem *Memory,
memorySize uint64) (uint64, error) {
return GasFastestStep, nil
}

func gasSwap(gt params.GasTable, evm *EVM, contract *Contract, stack *Stack, mem *Memory,
memorySize uint64) (uint64, error) {
return GasFastestStep, nil
}

func gasDup(gt params.GasTable, evm *EVM, contract *Contract, stack *Stack, mem *Memory,
```

```
memorySize uint64) (uint64, error) {
return GasFastestStep, nil
}

32:F:\git\coin\ethereum\go-ethereum\core\vm\gas_table_test.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package vm

import "testing"

func TestMemoryGasCost(t *testing.T) {
//size := uint64(math.MaxUint64 - 64)
size := uint64(0xfffffffe0)
v, err := memoryGasCost(&Memory{}, size)
if err != nil {
t.Error("didn't expect error:", err)
}
if v != 36028899963961341 {
t.Errorf("Expected: 36028899963961341, got %d", v)
}

_, err = memoryGasCost(&Memory{}, size+1)
if err == nil {
t.Error("expected error")
}
}

33:F:\git\coin\ethereum\go-ethereum\core\vm\gen_structlog.go
type StructLog struct {
Pc         uint64                      `json:"pc"`
Op         OpCode                      `json:"op"`
Gas        math.HexOrDecimal64         `json:"gas"`
GasCost    math.HexOrDecimal64         `json:"gasCost"`
Memory     hexutil.Bytes               `json:"memory"`
MemorySize int                         `json:"memSize"`
Stack      []*math.HexOrDecimal256     `json:"stack"`
Storage    map[common.Hash]common.Hash `json:"-"`
Depth      int                         `json:"depth"`
Err        error                       `json:"error"`
OpName     string                      `json:"opName"`
}
```

```
var enc StructLog
enc.Pc = s.Pc
enc.Op = s.Op
enc.Gas = math.HexOrDecimal64(s.Gas)
enc.GasCost = math.HexOrDecimal64(s.GasCost)
enc.Memory = s.Memory
enc.MemorySize = s.MemorySize
if s.Stack != nil {
enc.Stack = make([]*math.HexOrDecimal256, len(s.Stack))
for k, v := range s.Stack {
enc.Stack[k] = (*math.HexOrDecimal256)(v)
}
}
enc.Storage = s.Storage
enc.Depth = s.Depth
enc.Err = s.Err
enc.OpName = s.OpName()
return json.Marshal(&enc)
}

func (s *StructLog) UnmarshalJSON(input []byte) error {
type StructLog struct {
Pc         *uint64                `json:"pc"`
Op         *OpCode                `json:"op"`
Gas        *math.HexOrDecimal64   `json:"gas"`
GasCost    *math.HexOrDecimal64   `json:"gasCost"`
Memory     hexutil.Bytes          `json:"memory"`
MemorySize *int                   `json:"memSize"`
Stack      []*math.HexOrDecimal256 `json:"stack"`
Storage    map[common.Hash]common.Hash `json:"-"`
Depth      *int                   `json:"depth"`
Err        *error                 `json:"error"`
}
var dec StructLog
if err := json.Unmarshal(input, &dec); err != nil {
return err
}
if dec.Pc != nil {
s.Pc = *dec.Pc
}
if dec.Op != nil {
s.Op = *dec.Op
```

```
}
if dec.Gas != nil {
s.Gas = uint64(*dec.Gas)
}
if dec.GasCost != nil {
s.GasCost = uint64(*dec.GasCost)
}
if dec.Memory != nil {
s.Memory = dec.Memory
}
if dec.MemorySize != nil {
s.MemorySize = *dec.MemorySize
}
if dec.Stack != nil {
s.Stack = make([]*big.Int, len(dec.Stack))
for k, v := range dec.Stack {
s.Stack[k] = (*big.Int)(v)
}
}
if dec.Storage != nil {
s.Storage = dec.Storage
}
if dec.Depth != nil {
s.Depth = *dec.Depth
}
if dec.Err != nil {
s.Err = *dec.Err
}
return nil
}

34:F:\git\coin\ethereum\go-ethereum\core\vm\instructions.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package vm

import (
"fmt"
"math/big"

"github.com/ethereum/go-ethereum/common"
"github.com/ethereum/go-ethereum/common/math"
```

```
"github.com/ethereum/go-ethereum/core/types"
"github.com/ethereum/go-ethereum/crypto"
"github.com/ethereum/go-ethereum/params"
)

var (
bigZero = new(big.Int)
)

func opAdd(pc *uint64, evm *EVM, contract *Contract, memory *Memory, stack *Stack) ([]byte,
error) {
x, y := stack.pop(), stack.pop()
stack.push(math.U256(x.Add(x, y)))

evm.interpreter.intPool.put(y)

return nil, nil
}

func opSub(pc *uint64, evm *EVM, contract *Contract, memory *Memory, stack *Stack) ([]byte,
error) {
x, y := stack.pop(), stack.pop()
stack.push(math.U256(x.Sub(x, y)))

evm.interpreter.intPool.put(y)

return nil, nil
}

func opMul(pc *uint64, evm *EVM, contract *Contract, memory *Memory, stack *Stack) ([]byte,
error) {
x, y := stack.pop(), stack.pop()
stack.push(math.U256(x.Mul(x, y)))

evm.interpreter.intPool.put(y)

return nil, nil
}

func opDiv(pc *uint64, evm *EVM, contract *Contract, memory *Memory, stack *Stack) ([]byte,
error) {
x, y := stack.pop(), stack.pop()
```

```
if y.Sign() != 0 {
stack.push(math.U256(x.Div(x, y)))
} else {
stack.push(new(big.Int))
}

evm.interpreter.intPool.put(y)

return nil, nil
}

func opSdiv(pc *uint64, evm *EVM, contract *Contract, memory *Memory, stack *Stack) ([]byte,
error) {
x, y := math.S256(stack.pop()), math.S256(stack.pop())
if y.Sign() == 0 {
stack.push(new(big.Int))
return nil, nil
} else {
n := new(big.Int)
if evm.interpreter.intPool.get().Mul(x, y).Sign() < 0 {
n.SetInt64(-1)
} else {
n.SetInt64(1)
}

res := x.Div(x.Abs(x), y.Abs(y))
res.Mul(res, n)

stack.push(math.U256(res))
}
evm.interpreter.intPool.put(y)
return nil, nil
}

func opMod(pc *uint64, evm *EVM, contract *Contract, memory *Memory, stack *Stack) ([]byte,
error) {
x, y := stack.pop(), stack.pop()
if y.Sign() == 0 {
stack.push(new(big.Int))
} else {
stack.push(math.U256(x.Mod(x, y)))
}
```

```
evm.interpreter.intPool.put(y)
return nil, nil
}

func opSmod(pc *uint64, evm *EVM, contract *Contract, memory *Memory, stack *Stack) ([]byte,
error) {
x, y := math.S256(stack.pop()), math.S256(stack.pop())

if y.Sign() == 0 {
stack.push(new(big.Int))
} else {
n := new(big.Int)
if x.Sign() < 0 {
n.SetInt64(-1)
} else {
n.SetInt64(1)
}

res := x.Mod(x.Abs(x), y.Abs(y))
res.Mul(res, n)

stack.push(math.U256(res))
}
evm.interpreter.intPool.put(y)
return nil, nil
}

func opExp(pc *uint64, evm *EVM, contract *Contract, memory *Memory, stack *Stack) ([]byte,
error) {
base, exponent := stack.pop(), stack.pop()
stack.push(math.Exp(base, exponent))

evm.interpreter.intPool.put(base, exponent)

return nil, nil
}

func opSignExtend(pc *uint64, evm *EVM, contract *Contract, memory *Memory, stack *Stack)
([]byte, error) {
back := stack.pop()
if back.Cmp(big.NewInt(31)) < 0 {
bit := uint(back.Uint64()*8 + 7)
```

```
num := stack.pop()
mask := back.Lsh(common.Big1, bit)
mask.Sub(mask, common.Big1)
if num.Bit(int(bit)) > 0 {
num.Or(num, mask.Not(mask))
} else {
num.And(num, mask)
}

stack.push(math.U256(num))
}

evm.interpreter.intPool.put(back)
return nil, nil
}

func opNot(pc *uint64, evm *EVM, contract *Contract, memory *Memory, stack *Stack) ([]byte,
error) {
x := stack.pop()
stack.push(math.U256(x.Not(x)))
return nil, nil
}

func opLt(pc *uint64, evm *EVM, contract *Contract, memory *Memory, stack *Stack) ([]byte,
error) {
x, y := stack.pop(), stack.pop()
if x.Cmp(y) < 0 {
stack.push(evm.interpreter.intPool.get().SetUint64(1))
} else {
stack.push(new(big.Int))
}

evm.interpreter.intPool.put(x, y)
return nil, nil
}

func opGt(pc *uint64, evm *EVM, contract *Contract, memory *Memory, stack *Stack) ([]byte,
error) {
x, y := stack.pop(), stack.pop()
if x.Cmp(y) > 0 {
stack.push(evm.interpreter.intPool.get().SetUint64(1))
} else {
```

```
stack.push(new(big.Int))
}

evm.interpreter.intPool.put(x, y)
return nil, nil
}

func opSlt(pc *uint64, evm *EVM, contract *Contract, memory *Memory, stack *Stack) ([]byte,
error) {
x, y := math.S256(stack.pop()), math.S256(stack.pop())
if x.Cmp(math.S256(y)) < 0 {
stack.push(evm.interpreter.intPool.get().SetUint64(1))
} else {
stack.push(new(big.Int))
}

evm.interpreter.intPool.put(x, y)
return nil, nil
}

func opSgt(pc *uint64, evm *EVM, contract *Contract, memory *Memory, stack *Stack) ([]byte,
error) {
x, y := math.S256(stack.pop()), math.S256(stack.pop())
if x.Cmp(y) > 0 {
stack.push(evm.interpreter.intPool.get().SetUint64(1))
} else {
stack.push(new(big.Int))
}

evm.interpreter.intPool.put(x, y)
return nil, nil
}

func opEq(pc *uint64, evm *EVM, contract *Contract, memory *Memory, stack *Stack) ([]byte,
error) {
x, y := stack.pop(), stack.pop()
if x.Cmp(y) == 0 {
stack.push(evm.interpreter.intPool.get().SetUint64(1))
} else {
stack.push(new(big.Int))
}
```

```
evm.interpreter.intPool.put(x, y)
return nil, nil
}

func opIszero(pc *uint64, evm *EVM, contract *Contract, memory *Memory, stack *Stack) ([]byte,
error) {
x := stack.pop()
if x.Sign() > 0 {
stack.push(new(big.Int))
} else {
stack.push(evm.interpreter.intPool.get().SetUint64(1))
}

evm.interpreter.intPool.put(x)
return nil, nil
}

func opAnd(pc *uint64, evm *EVM, contract *Contract, memory *Memory, stack *Stack) ([]byte,
error) {
x, y := stack.pop(), stack.pop()
stack.push(x.And(x, y))

evm.interpreter.intPool.put(y)
return nil, nil
}
func opOr(pc *uint64, evm *EVM, contract *Contract, memory *Memory, stack *Stack) ([]byte,
error) {
x, y := stack.pop(), stack.pop()
stack.push(x.Or(x, y))

evm.interpreter.intPool.put(y)
return nil, nil
}
func opXor(pc *uint64, evm *EVM, contract *Contract, memory *Memory, stack *Stack) ([]byte,
error) {
x, y := stack.pop(), stack.pop()
stack.push(x.Xor(x, y))

evm.interpreter.intPool.put(y)
return nil, nil
}
```

```
func opByte(pc *uint64, evm *EVM, contract *Contract, memory *Memory, stack *Stack) ([]byte,
error) {
th, val := stack.pop(), stack.peek()
if th.Cmp(common.Big32) < 0 {
b := math.Byte(val, 32, int(th.Int64()))
val.SetUint64(uint64(b))
} else {
val.SetUint64(0)
}
evm.interpreter.intPool.put(th)
return nil, nil
}
func opAddmod(pc *uint64, evm *EVM, contract *Contract, memory *Memory, stack *Stack)
([]byte, error) {
x, y, z := stack.pop(), stack.pop(), stack.pop()
if z.Cmp(bigZero) > 0 {
add := x.Add(x, y)
add.Mod(add, z)
stack.push(math.U256(add))
} else {
stack.push(new(big.Int))
}

evm.interpreter.intPool.put(y, z)
return nil, nil
}
func opMulmod(pc *uint64, evm *EVM, contract *Contract, memory *Memory, stack *Stack)
([]byte, error) {
x, y, z := stack.pop(), stack.pop(), stack.pop()
if z.Cmp(bigZero) > 0 {
mul := x.Mul(x, y)
mul.Mod(mul, z)
stack.push(math.U256(mul))
} else {
stack.push(new(big.Int))
}

evm.interpreter.intPool.put(y, z)
return nil, nil
}

func opSha3(pc *uint64, evm *EVM, contract *Contract, memory *Memory, stack *Stack) ([]byte,
```

```
error) {
offset, size := stack.pop(), stack.pop()
data := memory.Get(offset.Int64(), size.Int64())
hash := crypto.Keccak256(data)

if evm.vmConfig.EnablePreimageRecording {
evm.StateDB.AddPreimage(common.BytesToHash(hash), data)
}

stack.push(new(big.Int).SetBytes(hash))

evm.interpreter.intPool.put(offset, size)
return nil, nil
}

func opAddress(pc *uint64, evm *EVM, contract *Contract, memory *Memory, stack *Stack)
([]byte, error) {
stack.push(contract.Address().Big())
return nil, nil
}

func opBalance(pc *uint64, evm *EVM, contract *Contract, memory *Memory, stack *Stack)
([]byte, error) {
addr := common.BigToAddress(stack.pop())
balance := evm.StateDB.GetBalance(addr)

stack.push(new(big.Int).Set(balance))
return nil, nil
}

func opOrigin(pc *uint64, evm *EVM, contract *Contract, memory *Memory, stack *Stack) ([]byte,
error) {
stack.push(evm.Origin.Big())
return nil, nil
}

func opCaller(pc *uint64, evm *EVM, contract *Contract, memory *Memory, stack *Stack) ([]byte,
error) {
stack.push(contract.Caller().Big())
return nil, nil
}
```

```
func opCallValue(pc *uint64, evm *EVM, contract *Contract, memory *Memory, stack *Stack)
([]byte, error) {
stack.push(evm.interpreter.intPool.get().Set(contract.value))
return nil, nil
}

func opCalldataLoad(pc *uint64, evm *EVM, contract *Contract, memory *Memory, stack *Stack)
([]byte, error) {
stack.push(new(big.Int).SetBytes(getData(contract.Input, stack.pop(), common.Big32)))
return nil, nil
}

func opCalldataSize(pc *uint64, evm *EVM, contract *Contract, memory *Memory, stack *Stack)
([]byte, error) {
stack.push(evm.interpreter.intPool.get().SetInt64(int64(len(contract.Input))))
return nil, nil
}

func opCalldataCopy(pc *uint64, evm *EVM, contract *Contract, memory *Memory, stack *Stack)
([]byte, error) {
var (
mOff = stack.pop()
cOff = stack.pop()
l    = stack.pop()
)
memory.Set(mOff.Uint64(), l.Uint64(), getData(contract.Input, cOff, l))

evm.interpreter.intPool.put(mOff, cOff, l)
return nil, nil
}

func opExtCodeSize(pc *uint64, evm *EVM, contract *Contract, memory *Memory, stack *Stack)
([]byte, error) {
a := stack.pop()

addr := common.BigToAddress(a)
a.SetInt64(int64(evm.StateDB.GetCodeSize(addr)))
stack.push(a)

return nil, nil
}
```

```
func opCodeSize(pc *uint64, evm *EVM, contract *Contract, memory *Memory, stack *Stack)
([]byte, error) {
l := evm.interpreter.intPool.get().SetInt64(int64(len(contract.Code)))
stack.push(l)
return nil, nil
}

func opCodeCopy(pc *uint64, evm *EVM, contract *Contract, memory *Memory, stack *Stack)
([]byte, error) {
var (
mOff = stack.pop()
cOff = stack.pop()
l    = stack.pop()
)
codeCopy := getData(contract.Code, cOff, l)

memory.Set(mOff.Uint64(), l.Uint64(), codeCopy)

evm.interpreter.intPool.put(mOff, cOff, l)
return nil, nil
}

func opExtCodeCopy(pc *uint64, evm *EVM, contract *Contract, memory *Memory, stack *Stack)
([]byte, error) {
var (
addr = common.BigToAddress(stack.pop())
mOff = stack.pop()
cOff = stack.pop()
l    = stack.pop()
)
codeCopy := getData(evm.StateDB.GetCode(addr), cOff, l)

memory.Set(mOff.Uint64(), l.Uint64(), codeCopy)

evm.interpreter.intPool.put(mOff, cOff, l)

return nil, nil
}

func opGasprice(pc *uint64, evm *EVM, contract *Contract, memory *Memory, stack *Stack)
([]byte, error) {
stack.push(evm.interpreter.intPool.get().Set(evm.GasPrice))
```

```
return nil, nil
}

func opBlockhash(pc *uint64, evm *EVM, contract *Contract, memory *Memory, stack *Stack)
([]byte, error) {
num := stack.pop()

n := evm.interpreter.intPool.get().Sub(evm.BlockNumber, common.Big257)
if num.Cmp(n) > 0 && num.Cmp(evm.BlockNumber) < 0 {
stack.push(evm.GetHash(num.Uint64()).Big())
} else {
stack.push(new(big.Int))
}

evm.interpreter.intPool.put(num, n)
return nil, nil
}

func opCoinbase(pc *uint64, evm *EVM, contract *Contract, memory *Memory, stack *Stack)
([]byte, error) {
stack.push(evm.Coinbase.Big())
return nil, nil
}

func opTimestamp(pc *uint64, evm *EVM, contract *Contract, memory *Memory, stack *Stack)
([]byte, error) {
stack.push(math.U256(new(big.Int).Set(evm.Time)))
return nil, nil
}

func opNumber(pc *uint64, evm *EVM, contract *Contract, memory *Memory, stack *Stack)
([]byte, error) {
stack.push(math.U256(new(big.Int).Set(evm.BlockNumber)))
return nil, nil
}

func opDifficulty(pc *uint64, evm *EVM, contract *Contract, memory *Memory, stack *Stack)
([]byte, error) {
stack.push(math.U256(new(big.Int).Set(evm.Difficulty)))
return nil, nil
}
```

```
func opGasLimit(pc *uint64, evm *EVM, contract *Contract, memory *Memory, stack *Stack)
([]byte, error) {
stack.push(math.U256(new(big.Int).Set(evm.GasLimit)))
return nil, nil
}

func opPop(pc *uint64, evm *EVM, contract *Contract, memory *Memory, stack *Stack) ([]byte,
error) {
evm.interpreter.intPool.put(stack.pop())
return nil, nil
}

func opMload(pc *uint64, evm *EVM, contract *Contract, memory *Memory, stack *Stack) ([]byte,
error) {
offset := stack.pop()
val := new(big.Int).SetBytes(memory.Get(offset.Int64(), 32))
stack.push(val)

evm.interpreter.intPool.put(offset)
return nil, nil
}

func opMstore(pc *uint64, evm *EVM, contract *Contract, memory *Memory, stack *Stack) ([]byte,
error) {
// pop value of the stack
mStart, val := stack.pop(), stack.pop()
memory.Set(mStart.Uint64(), 32, math.PaddedBigBytes(val, 32))

evm.interpreter.intPool.put(mStart, val)
return nil, nil
}

func opMstore8(pc *uint64, evm *EVM, contract *Contract, memory *Memory, stack *Stack)
([]byte, error) {
off, val := stack.pop().Int64(), stack.pop().Int64()
memory.store[off] = byte(val & 0xff)

return nil, nil
}

func opSload(pc *uint64, evm *EVM, contract *Contract, memory *Memory, stack *Stack) ([]byte,
error) {
```

```
loc := common.BigToHash(stack.pop())
val := evm.StateDB.GetState(contract.Address(), loc).Big()
stack.push(val)
return nil, nil
}

func opSstore(pc *uint64, evm *EVM, contract *Contract, memory *Memory, stack *Stack) ([]byte,
error) {
loc := common.BigToHash(stack.pop())
val := stack.pop()
evm.StateDB.SetState(contract.Address(), loc, common.BigToHash(val))

evm.interpreter.intPool.put(val)
return nil, nil
}

func opJump(pc *uint64, evm *EVM, contract *Contract, memory *Memory, stack *Stack) ([]byte,
error) {
pos := stack.pop()
if !contract.jumpdests.has(contract.CodeHash, contract.Code, pos) {
nop := contract.GetOp(pos.Uint64())
return nil, fmt.Errorf("invalid jump destination (%v) %v", nop, pos)
}
*pc = pos.Uint64()

evm.interpreter.intPool.put(pos)
return nil, nil
}
func opJumpi(pc *uint64, evm *EVM, contract *Contract, memory *Memory, stack *Stack) ([]byte,
error) {
pos, cond := stack.pop(), stack.pop()
if cond.Sign() != 0 {
if !contract.jumpdests.has(contract.CodeHash, contract.Code, pos) {
nop := contract.GetOp(pos.Uint64())
return nil, fmt.Errorf("invalid jump destination (%v) %v", nop, pos)
}
*pc = pos.Uint64()
} else {
*pc++
}

evm.interpreter.intPool.put(pos, cond)
```

```
return nil, nil
}
func opJumpdest(pc *uint64, evm *EVM, contract *Contract, memory *Memory, stack *Stack)
([]byte, error) {
return nil, nil
}

func opPc(pc *uint64, evm *EVM, contract *Contract, memory *Memory, stack *Stack) ([]byte,
error) {
stack.push(evm.interpreter.intPool.get().SetUint64(*pc))
return nil, nil
}

func opMsize(pc *uint64, evm *EVM, contract *Contract, memory *Memory, stack *Stack) ([]byte,
error) {
stack.push(evm.interpreter.intPool.get().SetInt64(int64(memory.Len())))
return nil, nil
}

func opGas(pc *uint64, evm *EVM, contract *Contract, memory *Memory, stack *Stack) ([]byte,
error) {
stack.push(evm.interpreter.intPool.get().SetUint64(contract.Gas))
return nil, nil
}

func opCreate(pc *uint64, evm *EVM, contract *Contract, memory *Memory, stack *Stack) ([]byte,
error) {
var (
value        = stack.pop()
offset, size = stack.pop(), stack.pop()
input        = memory.Get(offset.Int64(), size.Int64())
gas          = contract.Gas
)
if evm.ChainConfig().IsEIP150(evm.BlockNumber) {
gas -= gas / 64
}

contract.UseGas(gas)
_, addr, returnGas, suberr := evm.Create(contract, input, gas, value)
// Push item on the stack based on the returned error. If the ruleset is
// homestead we must check for CodeStoreOutOfGasError (homestead only
// rule) and treat as an error, if the ruleset is frontier we must
```

```
// ignore this error and pretend the operation was successful.
if evm.ChainConfig().IsHomestead(evm.BlockNumber) && suberr == ErrCodeStoreOutOfGas {
stack.push(new(big.Int))
} else if suberr != nil && suberr != ErrCodeStoreOutOfGas {
stack.push(new(big.Int))
} else {
stack.push(addr.Big())
}
contract.Gas += returnGas

evm.interpreter.intPool.put(value, offset, size)

return nil, nil
}

func opCall(pc *uint64, evm *EVM, contract *Contract, memory *Memory, stack *Stack) ([]byte,
error) {
gas := stack.pop().Uint64()
// pop gas and value of the stack.
addr, value := stack.pop(), stack.pop()
value = math.U256(value)
// pop input size and offset
inOffset, inSize := stack.pop(), stack.pop()
// pop return size and offset
retOffset, retSize := stack.pop(), stack.pop()

address := common.BigToAddress(addr)

// Get the arguments from the memory
args := memory.Get(inOffset.Int64(), inSize.Int64())

if value.Sign() != 0 {
gas += params.CallStipend
}

ret, returnGas, err := evm.Call(contract, address, args, gas, value)
if err != nil {
stack.push(new(big.Int))
} else {
stack.push(big.NewInt(1))

memory.Set(retOffset.Uint64(), retSize.Uint64(), ret)
```

```
}
contract.Gas += returnGas

evm.interpreter.intPool.put(addr, value, inOffset, inSize, retOffset, retSize)
return ret, nil
}

func opCallCode(pc *uint64, evm *EVM, contract *Contract, memory *Memory, stack *Stack)
([]byte, error) {
gas := stack.pop().Uint64()
// pop gas and value of the stack.
addr, value := stack.pop(), stack.pop()
value = math.U256(value)
// pop input size and offset
inOffset, inSize := stack.pop(), stack.pop()
// pop return size and offset
retOffset, retSize := stack.pop(), stack.pop()

address := common.BigToAddress(addr)

// Get the arguments from the memory
args := memory.Get(inOffset.Int64(), inSize.Int64())

if value.Sign() != 0 {
gas += params.CallStipend
}

ret, returnGas, err := evm.CallCode(contract, address, args, gas, value)
if err != nil {
stack.push(new(big.Int))

} else {
stack.push(big.NewInt(1))

memory.Set(retOffset.Uint64(), retSize.Uint64(), ret)
}
contract.Gas += returnGas

evm.interpreter.intPool.put(addr, value, inOffset, inSize, retOffset, retSize)
return ret, nil
}
```

```
func opDelegateCall(pc *uint64, evm *EVM, contract *Contract, memory *Memory, stack *Stack)
([]byte, error) {
gas, to, inOffset, inSize, outOffset, outSize := stack.pop().Uint64(), stack.pop(), stack.pop(),
stack.pop(), stack.pop(), stack.pop()

toAddr := common.BigToAddress(to)
args := memory.Get(inOffset.Int64(), inSize.Int64())

ret, returnGas, err := evm.DelegateCall(contract, toAddr, args, gas)
if err != nil {
stack.push(new(big.Int))
} else {
stack.push(big.NewInt(1))
memory.Set(outOffset.Uint64(), outSize.Uint64(), ret)
}
contract.Gas += returnGas

evm.interpreter.intPool.put(to, inOffset, inSize, outOffset, outSize)
return ret, nil
}

func opReturn(pc *uint64, evm *EVM, contract *Contract, memory *Memory, stack *Stack) ([]byte,
error) {
offset, size := stack.pop(), stack.pop()
ret := memory.GetPtr(offset.Int64(), size.Int64())

evm.interpreter.intPool.put(offset, size)

return ret, nil
}

func opStop(pc *uint64, evm *EVM, contract *Contract, memory *Memory, stack *Stack) ([]byte,
error) {
return nil, nil
}

func opSuicide(pc *uint64, evm *EVM, contract *Contract, memory *Memory, stack *Stack) ([]byte,
error) {
balance := evm.StateDB.GetBalance(contract.Address())
evm.StateDB.AddBalance(common.BigToAddress(stack.pop()), balance)

evm.StateDB.Suicide(contract.Address())
```

```
return nil, nil
}

// following functions are used by the instruction jump  table

// make log instruction function
func makeLog(size int) executionFunc {
return func(pc *uint64, evm *EVM, contract *Contract, memory *Memory, stack *Stack) ([]byte,
error) {
topics := make([]common.Hash, size)
mStart, mSize := stack.pop(), stack.pop()
for i := 0; i < size; i++ {
topics[i] = common.BigToHash(stack.pop())
}

d := memory.Get(mStart.Int64(), mSize.Int64())
evm.StateDB.AddLog(&types.Log{
Address: contract.Address(),
Topics:  topics,
Data:    d,
// This is a non-consensus field, but assigned here because
// core/state doesn't know the current block number.
BlockNumber: evm.BlockNumber.Uint64(),
})

evm.interpreter.intPool.put(mStart, mSize)
return nil, nil
}
}

// make push instruction function
func makePush(size uint64, pushByteSize int) executionFunc {
return func(pc *uint64, evm *EVM, contract *Contract, memory *Memory, stack *Stack) ([]byte,
error) {
codeLen := len(contract.Code)

startMin := codeLen
if int(*pc+1) < startMin {
startMin = int(*pc + 1)
}
```

```
endMin := codeLen
if startMin+pushByteSize < endMin {
endMin = startMin + pushByteSize
}

integer := evm.interpreter.intPool.get()
stack.push(integer.SetBytes(common.RightPadBytes(contract.Code[startMin:endMin],
pushByteSize)))

*pc += size
return nil, nil
}
}

// make push instruction function
func makeDup(size int64) executionFunc {
return func(pc *uint64, evm *EVM, contract *Contract, memory *Memory, stack *Stack) ([]byte,
error) {
stack.dup(evm.interpreter.intPool, int(size))
return nil, nil
}
}

// make swap instruction function
func makeSwap(size int64) executionFunc {
// switch n + 1 otherwise n would be swapped with n
size += 1
return func(pc *uint64, evm *EVM, contract *Contract, memory *Memory, stack *Stack) ([]byte,
error) {
stack.swap(int(size))
return nil, nil
}
}

35:F:\git\coin\ethereum\go-ethereum\core\vm\instructions_test.go
stack = newstack()
)
tests := []struct {
v        string
th       uint64
expected *big.Int
}{
```

```
{"ABCDEF0908070605040302010000000000000000000000000000000000000000", 0,
big.NewInt(0xAB)},
{"ABCDEF0908070605040302010000000000000000000000000000000000000000", 1,
big.NewInt(0xCD)},
{"00CDEF09080706050403020 1ffffffffffffffffffffffffffffffffffffff", 0, big.NewInt(0x00)},
{"00CDEF09080706050403020 1ffffffffffffffffffffffffffffffffffffff", 1, big.NewInt(0xCD)},
{"0000000000000000000000000000000000000000000000000000000000102030", 31,
big.NewInt(0x30)},
{"0000000000000000000000000000000000000000000000000000000000102030", 30,
big.NewInt(0x20)},
{"ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff", 32, big.NewInt(0x0)},
{"ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff", 0xFFFFFFFFFFFFFFFF, big.NewInt(0x0)},
}
pc := uint64(0)
for _, test := range tests {
val := new(big.Int).SetBytes(common.Hex2Bytes(test.v))
th := new(big.Int).SetUint64(test.th)
stack.push(val)
stack.push(th)
opByte(&pc, env, nil, nil, stack)
actual := stack.pop()
if actual.Cmp(test.expected) != 0 {
t.Fatalf("Expected  [%v] %v:th byte to be %v, was %v.", test.v, test.th, test.expected, actual)
}
}
}

func opBenchmark(bench *testing.B, op func(pc *uint64, evm *EVM, contract *Contract, memory
*Memory, stack *Stack) ([]byte, error), args ...string) {
var (
env   = NewEVM(Context{}, nil, params.TestChainConfig, Config{EnableJit: false, ForceJit: false})
stack = newstack()
)
// convert args
byteArgs := make([][]byte, len(args))
for i, arg := range args {
byteArgs[i] = common.Hex2Bytes(arg)
}
pc := uint64(0)
bench.ResetTimer()
for i := 0; i < bench.N; i++ {
for _, arg := range byteArgs {
```

```
a := new(big.Int).SetBytes(arg)
stack.push(a)
}
op(&pc, env, nil, nil, stack)
stack.pop()
}
}

func precompiledBenchmark(addr, input, expected string, gas uint64, bench *testing.B) {

contract := NewContract(AccountRef(common.HexToAddress("1337")),
nil, new(big.Int), gas)

p := PrecompiledContracts[common.HexToAddress(addr)]
in := common.Hex2Bytes(input)
var (
res []byte
err error
)
data := make([]byte, len(in))
bench.ResetTimer()
for i := 0; i < bench.N; i++ {
contract.Gas = gas
copy(data, in)
res, err = RunPrecompiledContract(p, data, contract)
}
bench.StopTimer()
//Check if it is correct
if err != nil {
bench.Error(err)
return
}
if common.Bytes2Hex(res) != expected {
bench.Error(fmt.Sprintf("Expected %v, got %v", expected, common.Bytes2Hex(res)))
return
}
}

func BenchmarkPrecompiledEcdsa(bench *testing.B) {
var (
addr = "01"
inp  =
```

```
"38d18acb67d25c8bb9942764b62f18e17054f66a817bd4295423adf9ed98873e000000000000000
0000000000000000000000000000000000000000001b38d18acb67d25c8bb9942764b62f18
e17054f66a817bd4295423adf9ed98873e789d1dd423d25f0772d2748d60f7e4b81bb14d086eba8e
8e8efb6dcff8a4ae02"
exp  = "0000000000000000000000ceaccac640adf55b2028469bd36ba501f28b699d"
gas  = uint64(4000000)
)
precompiledBenchmark(addr, inp, exp, gas, bench)
}
func BenchmarkPrecompiledSha256(bench *testing.B) {
var (
addr = "02"
inp  =
"38d18acb67d25c8bb9942764b62f18e17054f66a817bd4295423adf9ed98873e000000000000000
0000000000000000000000000000000000000000001b38d18acb67d25c8bb9942764b62f18
e17054f66a817bd4295423adf9ed98873e789d1dd423d25f0772d2748d60f7e4b81bb14d086eba8e
8e8efb6dcff8a4ae02"
exp  = "811c7003375852fabd0d362e40e68607a12bdabae61a7d068fe5fdd1dbbf2a5d"
gas  = uint64(4000000)
)
precompiledBenchmark(addr, inp, exp, gas, bench)
}
func BenchmarkPrecompiledRipeMD(bench *testing.B) {
var (
addr = "03"
inp  =
"38d18acb67d25c8bb9942764b62f18e17054f66a817bd4295423adf9ed98873e000000000000000
0000000000000000000000000000000000000000001b38d18acb67d25c8bb9942764b62f18
e17054f66a817bd4295423adf9ed98873e789d1dd423d25f0772d2748d60f7e4b81bb14d086eba8e
8e8efb6dcff8a4ae02"
exp  = "0000000000000000000000009215b8d9882ff46f0dfde6684d78e831467f65e6"
gas  = uint64(4000000)
)
precompiledBenchmark(addr, inp, exp, gas, bench)
}
func BenchmarkPrecompiledIdentity(bench *testing.B) {
var (
addr = "04"
inp  =
"38d18acb67d25c8bb9942764b62f18e17054f66a817bd4295423adf9ed98873e000000000000000
0000000000000000000000000000000000000000001b38d18acb67d25c8bb9942764b62f18
e17054f66a817bd4295423adf9ed98873e789d1dd423d25f0772d2748d60f7e4b81bb14d086eba8e
```

```
8e8efb6dcff8a4ae02"
exp  =
"38d18acb67d25c8bb9942764b62f18e17054f66a817bd4295423adf9ed98873e000000000000000
0000000000000000000000000000000000000000000001b38d18acb67d25c8bb9942764b62f18
e17054f66a817bd4295423adf9ed98873e789d1dd423d25f0772d2748d60f7e4b81bb14d086eba8e
8e8efb6dcff8a4ae02"
gas  = uint64(4000000)
)
precompiledBenchmark(addr, inp, exp, gas, bench)
}
func BenchmarkOpAdd(b *testing.B) {
x := "ABCDEF090807060504030201ffffffffffffffffffffffffffffffffffffffffffffffff"
y := "ABCDEF090807060504030201ffffffffffffffffffffffffffffffffffffffffffffffff"

opBenchmark(b, opAdd, x, y)

}
func BenchmarkOpSub(b *testing.B) {
x := "ABCDEF090807060504030201ffffffffffffffffffffffffffffffffffffffffffffffff"
y := "ABCDEF090807060504030201ffffffffffffffffffffffffffffffffffffffffffffffff"

opBenchmark(b, opSub, x, y)

}
func BenchmarkOpMul(b *testing.B) {
x := "ABCDEF090807060504030201ffffffffffffffffffffffffffffffffffffffffffffffff"
y := "ABCDEF090807060504030201ffffffffffffffffffffffffffffffffffffffffffffffff"

opBenchmark(b, opMul, x, y)

}
func BenchmarkOpDiv(b *testing.B) {
x := "ABCDEF090807060504030201ffffffffffffffffffffffffffffffffffffffffffffffff"
y := "ABCDEF090807060504030201ffffffffffffffffffffffffffffffffffffffffffffffff"

opBenchmark(b, opDiv, x, y)

}
func BenchmarkOpSdiv(b *testing.B) {
x := "ABCDEF090807060504030201ffffffffffffffffffffffffffffffffffffffffffffffff"
y := "ABCDEF090807060504030201ffffffffffffffffffffffffffffffffffffffffffffffff"
```

```
opBenchmark(b, opSdiv, x, y)

}
func BenchmarkOpMod(b *testing.B) {
x := "ABCDEF090807060504030201ffffffffffffffffffffffffffffffffffffffff"
y := "ABCDEF090807060504030201ffffffffffffffffffffffffffffffffffffffff"

opBenchmark(b, opMod, x, y)

}
func BenchmarkOpSmod(b *testing.B) {
x := "ABCDEF090807060504030201ffffffffffffffffffffffffffffffffffffffff"
y := "ABCDEF090807060504030201ffffffffffffffffffffffffffffffffffffffff"

opBenchmark(b, opSmod, x, y)

}
func BenchmarkOpExp(b *testing.B) {
x := "ABCDEF090807060504030201ffffffffffffffffffffffffffffffffffffffff"
y := "ABCDEF090807060504030201ffffffffffffffffffffffffffffffffffffffff"

opBenchmark(b, opExp, x, y)

}
func BenchmarkOpSignExtend(b *testing.B) {
x := "ABCDEF090807060504030201ffffffffffffffffffffffffffffffffffffffff"
y := "ABCDEF090807060504030201ffffffffffffffffffffffffffffffffffffffff"

opBenchmark(b, opSignExtend, x, y)

}
func BenchmarkOpLt(b *testing.B) {
x := "ABCDEF090807060504030201ffffffffffffffffffffffffffffffffffffffff"
y := "ABCDEF090807060504030201ffffffffffffffffffffffffffffffffffffffff"

opBenchmark(b, opLt, x, y)

}
func BenchmarkOpGt(b *testing.B) {
x := "ABCDEF090807060504030201ffffffffffffffffffffffffffffffffffffffff"
y := "ABCDEF090807060504030201ffffffffffffffffffffffffffffffffffffffff"
```

```
opBenchmark(b, opGt, x, y)

}
func BenchmarkOpSlt(b *testing.B) {
x := "ABCDEF090807060504030201ffffffffffffffffffffffffffffffffffffffffffffffff"
y := "ABCDEF090807060504030201ffffffffffffffffffffffffffffffffffffffffffffffff"

opBenchmark(b, opSlt, x, y)

}
func BenchmarkOpSgt(b *testing.B) {
x := "ABCDEF090807060504030201ffffffffffffffffffffffffffffffffffffffffffffffff"
y := "ABCDEF090807060504030201ffffffffffffffffffffffffffffffffffffffffffffffff"

opBenchmark(b, opSgt, x, y)

}
func BenchmarkOpEq(b *testing.B) {
x := "ABCDEF090807060504030201ffffffffffffffffffffffffffffffffffffffffffffffff"
y := "ABCDEF090807060504030201ffffffffffffffffffffffffffffffffffffffffffffffff"

opBenchmark(b, opEq, x, y)

}
func BenchmarkOpAnd(b *testing.B) {
x := "ABCDEF090807060504030201ffffffffffffffffffffffffffffffffffffffffffffffff"
y := "ABCDEF090807060504030201ffffffffffffffffffffffffffffffffffffffffffffffff"

opBenchmark(b, opAnd, x, y)

}
func BenchmarkOpOr(b *testing.B) {
x := "ABCDEF090807060504030201ffffffffffffffffffffffffffffffffffffffffffffffff"
y := "ABCDEF090807060504030201ffffffffffffffffffffffffffffffffffffffffffffffff"

opBenchmark(b, opOr, x, y)

}
func BenchmarkOpXor(b *testing.B) {
x := "ABCDEF090807060504030201ffffffffffffffffffffffffffffffffffffffffffffffff"
y := "ABCDEF090807060504030201ffffffffffffffffffffffffffffffffffffffffffffffff"
```

```
opBenchmark(b, opXor, x, y)

}
func BenchmarkOpByte(b *testing.B) {
x := "ABCDEF090807060504030201ffffffffffffffffffffffffffffffffffffffffffffffff"
y := "ABCDEF090807060504030201ffffffffffffffffffffffffffffffffffffffffffffffff"

opBenchmark(b, opByte, x, y)

}

func BenchmarkOpAddmod(b *testing.B) {
x := "ABCDEF090807060504030201ffffffffffffffffffffffffffffffffffffffffffffffff"
y := "ABCDEF090807060504030201ffffffffffffffffffffffffffffffffffffffffffffffff"
z := "ABCDEF090807060504030201ffffffffffffffffffffffffffffffffffffffffffffffff"

opBenchmark(b, opAddmod, x, y, z)

}
func BenchmarkOpMulmod(b *testing.B) {
x := "ABCDEF090807060504030201ffffffffffffffffffffffffffffffffffffffffffffffff"
y := "ABCDEF090807060504030201ffffffffffffffffffffffffffffffffffffffffffffffff"
z := "ABCDEF090807060504030201ffffffffffffffffffffffffffffffffffffffffffffffff"

opBenchmark(b, opMulmod, x, y, z)

}

//func BenchmarkOpSha3(b *testing.B) {
//x := "0"
//y := "32"
//
//opBenchmark(b,opSha3, x, y)
//
//
//}

36:F:\git\coin\ethereum\go-ethereum\core\vm\interface.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package vm
```

```
import (
"math/big"

"github.com/ethereum/go-ethereum/common"
"github.com/ethereum/go-ethereum/core/types"
)

// StateDB is an EVM database for full state querying.
type StateDB interface {
CreateAccount(common.Address)

SubBalance(common.Address, *big.Int)
AddBalance(common.Address, *big.Int)
GetBalance(common.Address) *big.Int

GetNonce(common.Address) uint64
SetNonce(common.Address, uint64)

GetCodeHash(common.Address) common.Hash
GetCode(common.Address) []byte
SetCode(common.Address, []byte)
GetCodeSize(common.Address) int

AddRefund(*big.Int)
GetRefund() *big.Int

GetState(common.Address, common.Hash) common.Hash
SetState(common.Address, common.Hash, common.Hash)

Suicide(common.Address) bool
HasSuicided(common.Address) bool

// Exist reports whether the given account exists in state.
// Notably this should also return true for suicided accounts.
Exist(common.Address) bool
// Empty returns whether the given account is empty. Empty
// is defined according to EIP161 (balance = nonce = code = 0).
Empty(common.Address) bool

RevertToSnapshot(int)
Snapshot() int
```

```
AddLog(*types.Log)
AddPreimage(common.Hash, []byte)

ForEachStorage(common.Address, func(common.Hash, common.Hash) bool)
}

// CallContext provides a basic interface for the EVM calling conventions. The EVM EVM
// depends on this context being implemented for doing subcalls and initialising new EVM
contracts.
type CallContext interface {
// Call another contract
Call(env *EVM, me ContractRef, addr common.Address, data []byte, gas, value *big.Int) ([]byte,
error)
// Take another's contract code and execute within our own context
CallCode(env *EVM, me ContractRef, addr common.Address, data []byte, gas, value *big.Int)
([]byte, error)
// Same as CallCode except sender and value is propagated from parent to child scope
DelegateCall(env *EVM, me ContractRef, addr common.Address, data []byte, gas *big.Int) ([]byte,
error)
// Create a new contract
Create(env *EVM, me ContractRef, data []byte, gas, value *big.Int) ([]byte, common.Address,
error)
}

37:F:\git\coin\ethereum\go-ethereum\core\vm\interpreter.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package vm

import (
"fmt"
"sync/atomic"
"time"

"github.com/ethereum/go-ethereum/common"
"github.com/ethereum/go-ethereum/common/math"
"github.com/ethereum/go-ethereum/crypto"
"github.com/ethereum/go-ethereum/log"
"github.com/ethereum/go-ethereum/params"
)

// Config are the configuration options for the Interpreter
```

```
type Config struct {
// Debug enabled debugging Interpreter options
Debug bool
// EnableJit enabled the JIT VM
EnableJit bool
// ForceJit forces the JIT VM
ForceJit bool
// Tracer is the op code logger
Tracer Tracer
// NoRecursion disabled Interpreter call, callcode,
// delegate call and create.
NoRecursion bool
// Disable gas metering
DisableGasMetering bool
// Enable recording of SHA3/keccak preimages
EnablePreimageRecording bool
// JumpTable contains the EVM instruction table. This
// may me left uninitialised and will be set the default
// table.
JumpTable [256]operation
}

// Interpreter is used to run Ethereum based contracts and will utilise the
// passed evmironment to query external sources for state information.
// The Interpreter will run the byte code VM or JIT VM based on the passed
// configuration.
type Interpreter struct {
evm      *EVM
cfg      Config
gasTable params.GasTable
intPool  *intPool

readonly bool
}

// NewInterpreter returns a new instance of the Interpreter.
func NewInterpreter(evm *EVM, cfg Config) *Interpreter {
// We use the STOP instruction whether to see
// the jump table was initialised. If it was not
// we'll set the default jump table.
if !cfg.JumpTable[STOP].valid {
switch {
```

```
case evm.ChainConfig().IsHomestead(evm.BlockNumber):
cfg.JumpTable = homesteadInstructionSet
default:
cfg.JumpTable = frontierInstructionSet
}
}

return &Interpreter{
evm:      evm,
cfg:      cfg,
gasTable: evm.ChainConfig().GasTable(evm.BlockNumber),
intPool:  newIntPool(),
}
}

func (in *Interpreter) enforceRestrictions(op OpCode, operation operation, stack *Stack) error {
return nil
}

// Run loops and evaluates the contract's code with the given input data and returns
// the return byte-slice and an error if one occurred.
//
// It's important to note that any errors returned by the interpreter should be
// considered a revert-and-consume-all-gas operation. No error specific checks
// should be handled to reduce complexity and errors further down the in.
func (in *Interpreter) Run(snapshot int, contract *Contract, input []byte) (ret []byte, err error) {
in.evm.depth++
defer func() { in.evm.depth-- }()

// Don't bother with the execution if there's no code.
if len(contract.Code) == 0 {
return nil, nil
}

codehash := contract.CodeHash // codehash is used when doing jump dest caching
if codehash == (common.Hash{}) {
codehash = crypto.Keccak256Hash(contract.Code)
}

var (
op    OpCode        // current opcode
mem   = NewMemory() // bound memory
```

```
stack = newstack()  // local stack
// For optimisation reason we're using uint64 as the program counter.
// It's theoretically possible to go above 2^64. The YP defines the PC
// to be uint256. Practically much less so feasible.
pc   = uint64(0) // program counter
cost uint64
)
contract.Input = input

// User defer pattern to check for an error and, based on the error being nil or not, use all gas and return.
defer func() {
if err != nil && in.cfg.Debug {
// XXX For debugging
//fmt.Printf("%04d: %8v    cost = %-8d stack = %-8d ERR = %v\n", pc, op, cost, stack.len(), err)
in.cfg.Tracer.CaptureState(in.evm, pc, op, contract.Gas, cost, mem, stack, contract, in.evm.depth, err)
}
}()

log.Debug("interpreter running contract", "hash", codehash[:])
tstart := time.Now()
defer log.Debug("interpreter finished running contract", "hash", codehash[:], "elapsed", time.Since(tstart))

// The Interpreter main run loop (contextual). This loop runs until either an
// explicit STOP, RETURN or SELFDESTRUCT is executed, an error occurred during
// the execution of one of the operations or until the done flag is set by the
// parent context.
for atomic.LoadInt32(&in.evm.abort) == 0 {
// Get the memory location of pc
op = contract.GetOp(pc)

// get the operation from the jump table matching the opcode
operation := in.cfg.JumpTable[op]
if err := in.enforceRestrictions(op, operation, stack); err != nil {
return nil, err
}

// if the op is invalid abort the process and return an error
if !operation.valid {
return nil, fmt.Errorf("invalid opcode 0x%x", int(op))
```

```
}

// validate the stack and make sure there enough stack items available
// to perform the operation
if err := operation.validateStack(stack); err != nil {
return nil, err
}

var memorySize uint64
// calculate the new memory size and expand the memory to fit
// the operation
if operation.memorySize != nil {
memSize, overflow := bigUint64(operation.memorySize(stack))
if overflow {
return nil, errGasUintOverflow
}
// memory is expanded in words of 32 bytes. Gas
// is also calculated in words.
if memorySize, overflow = math.SafeMul(toWordSize(memSize), 32); overflow {
return nil, errGasUintOverflow
}
}

if !in.cfg.DisableGasMetering {
// consume the gas and return an error if not enough gas is available.
// cost is explicitly set so that the capture state defer method cas get the proper cost
cost, err = operation.gasCost(in.gasTable, in.evm, contract, stack, mem, memorySize)
if err != nil || !contract.UseGas(cost) {
return nil, ErrOutOfGas
}
}
if memorySize > 0 {
mem.Resize(memorySize)
}

if in.cfg.Debug {
in.cfg.Tracer.CaptureState(in.evm, pc, op, contract.Gas, cost, mem, stack, contract, in.evm.depth,
err)
}
// XXX For debugging
//fmt.Printf("%04d: %8v    cost = %-8d stack = %-8d\n", pc, op, cost, stack.len())
```

```
// execute the operation
res, err := operation.execute(&pc, in.evm, contract, mem, stack)
// verifyPool is a build flag. Pool verification makes sure the integrity
// of the integer pool by comparing values to a default value.
if verifyPool {
verifyIntegerPool(in.intPool)
}

switch {
case err != nil:
return nil, err
case operation.halts:
return res, nil
case !operation.jumps:
pc++
}
// if the operation returned a value make sure that is also set
// the last return data.
if res != nil {
mem.lastReturn = ret
}
}
return nil, nil
}

38:F:\git\coin\ethereum\go-ethereum\core\vm\intpool.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package vm

import "math/big"

var checkVal = big.NewInt(-42)

const poolLimit = 256

// intPool is a pool of big integers that
// can be reused for all big.Int operations.
type intPool struct {
pool *Stack
}
```

```
func newIntPool() *intPool {
return &intPool{pool: newstack()}
}

func (p *intPool) get() *big.Int {
if p.pool.len() > 0 {
return p.pool.pop()
}
return new(big.Int)
}
func (p *intPool) put(is ...*big.Int) {
if len(p.pool.data) > poolLimit {
return
}

for _, i := range is {
// verifyPool is a build flag. Pool verification makes sure the integrity
// of the integer pool by comparing values to a default value.
if verifyPool {
i.Set(checkVal)
}

p.pool.push(i)
}
}

39:F:\git\coin\ethereum\go-ethereum\core\vm\int_pool_verifier.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

// +build VERIFY_EVM_INTEGER_POOL

package vm

import "fmt"

const verifyPool = true

func verifyIntegerPool(ip *intPool) {
for i, item := range ip.pool.data {
if item.Cmp(checkVal) != 0 {
panic(fmt.Sprintf("%d'th item failed aggressive pool check. Value was modified", i))
}
```

```
}
}
```

40:F:\git\coin\ethereum\go-ethereum\core\vm\int_pool_verifier_empty.go

```
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

// +build !VERIFY_EVM_INTEGER_POOL

package vm

const verifyPool = false

func verifyIntegerPool(ip *intPool) {}
```

41:F:\git\coin\ethereum\go-ethereum\core\vm\jump_table.go

```
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package vm

import (
"errors"
"math/big"

"github.com/ethereum/go-ethereum/params"
)

type (
executionFunc       func(pc *uint64, env *EVM, contract *Contract, memory *Memory, stack *Stack)
([]byte, error)
gasFunc             func(params.GasTable, *EVM, *Contract, *Stack, *Memory, uint64) (uint64, error)
// last parameter is the requested memory size as a uint64
stackValidationFunc func(*Stack) error
memorySizeFunc      func(*Stack) *big.Int
)

var errGasUintOverflow = errors.New("gas uint64 overflow")

type operation struct {
// op is the operation function
execute executionFunc
// gasCost is the gas function and returns the gas required for execution
gasCost gasFunc
```

```
// validateStack validates the stack (size) for the operation
validateStack stackValidationFunc
// memorySize returns the memory size required for the operation
memorySize memorySizeFunc
// halts indicates whether the operation shoult halt further execution
// and return
halts bool
// jumps indicates whether operation made a jump. This prevents the program
// counter from further incrementing.
jumps bool
// writes determines whether this a state modifying operation
writes bool
// valid is used to check whether the retrieved operation is valid and known
valid bool
// reverts determined whether the operation reverts state
reverts bool
}

var (
frontierInstructionSet  = NewFrontierInstructionSet()
homesteadInstructionSet = NewHomesteadInstructionSet()
)

// NewHomesteadInstructionSet returns the frontier and homestead
// instructions that can be executed during the homestead phase.
func NewHomesteadInstructionSet() [256]operation {
instructionSet := NewFrontierInstructionSet()
instructionSet[DELEGATECALL] = operation{
execute:       opDelegateCall,
gasCost:       gasDelegateCall,
validateStack: makeStackFunc(6, 1),
memorySize:    memoryDelegateCall,
valid:         true,
}
return instructionSet
}

// NewFrontierInstructionSet returns the frontier instructions
// that can be executed during the frontier phase.
func NewFrontierInstructionSet() [256]operation {
return [256]operation{
STOP: {
```

```
execute:       opStop,
gasCost:       constGasFunc(0),
validateStack: makeStackFunc(0, 0),
halts:         true,
valid:         true,
},
ADD: {
execute:       opAdd,
gasCost:       constGasFunc(GasFastestStep),
validateStack: makeStackFunc(2, 1),
valid:         true,
},
MUL: {
execute:       opMul,
gasCost:       constGasFunc(GasFastStep),
validateStack: makeStackFunc(2, 1),
valid:         true,
},
SUB: {
execute:       opSub,
gasCost:       constGasFunc(GasFastestStep),
validateStack: makeStackFunc(2, 1),
valid:         true,
},
DIV: {
execute:       opDiv,
gasCost:       constGasFunc(GasFastStep),
validateStack: makeStackFunc(2, 1),
valid:         true,
},
SDIV: {
execute:       opSdiv,
gasCost:       constGasFunc(GasFastStep),
validateStack: makeStackFunc(2, 1),
valid:         true,
},
MOD: {
execute:       opMod,
gasCost:       constGasFunc(GasFastStep),
validateStack: makeStackFunc(2, 1),
valid:         true,
},
```

```
SMOD: {
execute:       opSmod,
gasCost:       constGasFunc(GasFastStep),
validateStack: makeStackFunc(2, 1),
valid:         true,
},
ADDMOD: {
execute:       opAddmod,
gasCost:       constGasFunc(GasMidStep),
validateStack: makeStackFunc(3, 1),
valid:         true,
},
MULMOD: {
execute:       opMulmod,
gasCost:       constGasFunc(GasMidStep),
validateStack: makeStackFunc(3, 1),
valid:         true,
},
EXP: {
execute:       opExp,
gasCost:       gasExp,
validateStack: makeStackFunc(2, 1),
valid:         true,
},
SIGNEXTEND: {
execute:       opSignExtend,
gasCost:       constGasFunc(GasFastStep),
validateStack: makeStackFunc(2, 1),
valid:         true,
},
LT: {
execute:       opLt,
gasCost:       constGasFunc(GasFastestStep),
validateStack: makeStackFunc(2, 1),
valid:         true,
},
GT: {
execute:       opGt,
gasCost:       constGasFunc(GasFastestStep),
validateStack: makeStackFunc(2, 1),
valid:         true,
},
```

```
SLT: {
execute:       opSlt,
gasCost:       constGasFunc(GasFastestStep),
validateStack: makeStackFunc(2, 1),
valid:         true,
},
SGT: {
execute:       opSgt,
gasCost:       constGasFunc(GasFastestStep),
validateStack: makeStackFunc(2, 1),
valid:         true,
},
EQ: {
execute:       opEq,
gasCost:       constGasFunc(GasFastestStep),
validateStack: makeStackFunc(2, 1),
valid:         true,
},
ISZERO: {
execute:       opIszero,
gasCost:       constGasFunc(GasFastestStep),
validateStack: makeStackFunc(1, 1),
valid:         true,
},
AND: {
execute:       opAnd,
gasCost:       constGasFunc(GasFastestStep),
validateStack: makeStackFunc(2, 1),
valid:         true,
},
XOR: {
execute:       opXor,
gasCost:       constGasFunc(GasFastestStep),
validateStack: makeStackFunc(2, 1),
valid:         true,
},
OR: {
execute:       opOr,
gasCost:       constGasFunc(GasFastestStep),
validateStack: makeStackFunc(2, 1),
valid:         true,
},
```

```
NOT: {
execute:       opNot,
gasCost:       constGasFunc(GasFastestStep),
validateStack: makeStackFunc(1, 1),
valid:         true,
},
BYTE: {
execute:       opByte,
gasCost:       constGasFunc(GasFastestStep),
validateStack: makeStackFunc(2, 1),
valid:         true,
},
SHA3: {
execute:       opSha3,
gasCost:       gasSha3,
validateStack: makeStackFunc(2, 1),
memorySize:    memorySha3,
valid:         true,
},
ADDRESS: {
execute:       opAddress,
gasCost:       constGasFunc(GasQuickStep),
validateStack: makeStackFunc(0, 1),
valid:         true,
},
BALANCE: {
execute:       opBalance,
gasCost:       gasBalance,
validateStack: makeStackFunc(1, 1),
valid:         true,
},
ORIGIN: {
execute:       opOrigin,
gasCost:       constGasFunc(GasQuickStep),
validateStack: makeStackFunc(0, 1),
valid:         true,
},
CALLER: {
execute:       opCaller,
gasCost:       constGasFunc(GasQuickStep),
validateStack: makeStackFunc(0, 1),
valid:         true,
```

```
},
CALLVALUE: {
execute:       opCallValue,
gasCost:       constGasFunc(GasQuickStep),
validateStack: makeStackFunc(0, 1),
valid:         true,
},
CALLDATALOAD: {
execute:       opCalldataLoad,
gasCost:       constGasFunc(GasFastestStep),
validateStack: makeStackFunc(1, 1),
valid:         true,
},
CALLDATASIZE: {
execute:       opCalldataSize,
gasCost:       constGasFunc(GasQuickStep),
validateStack: makeStackFunc(0, 1),
valid:         true,
},
CALLDATACOPY: {
execute:       opCalldataCopy,
gasCost:       gasCalldataCopy,
validateStack: makeStackFunc(3, 0),
memorySize:    memoryCalldataCopy,
valid:         true,
},
CODESIZE: {
execute:       opCodeSize,
gasCost:       constGasFunc(GasQuickStep),
validateStack: makeStackFunc(0, 1),
valid:         true,
},
CODECOPY: {
execute:       opCodeCopy,
gasCost:       gasCodeCopy,
validateStack: makeStackFunc(3, 0),
memorySize:    memoryCodeCopy,
valid:         true,
},
GASPRICE: {
execute:       opGasprice,
gasCost:       constGasFunc(GasQuickStep),
```

```
validateStack: makeStackFunc(0, 1),
valid:         true,
},
EXTCODESIZE: {
execute:       opExtCodeSize,
gasCost:       gasExtCodeSize,
validateStack: makeStackFunc(1, 1),
valid:         true,
},
EXTCODECOPY: {
execute:       opExtCodeCopy,
gasCost:       gasExtCodeCopy,
validateStack: makeStackFunc(4, 0),
memorySize:    memoryExtCodeCopy,
valid:         true,
},
BLOCKHASH: {
execute:       opBlockhash,
gasCost:       constGasFunc(GasExtStep),
validateStack: makeStackFunc(1, 1),
valid:         true,
},
COINBASE: {
execute:       opCoinbase,
gasCost:       constGasFunc(GasQuickStep),
validateStack: makeStackFunc(0, 1),
valid:         true,
},
TIMESTAMP: {
execute:       opTimestamp,
gasCost:       constGasFunc(GasQuickStep),
validateStack: makeStackFunc(0, 1),
valid:         true,
},
NUMBER: {
execute:       opNumber,
gasCost:       constGasFunc(GasQuickStep),
validateStack: makeStackFunc(0, 1),
valid:         true,
},
DIFFICULTY: {
execute:       opDifficulty,
```

```
gasCost:       constGasFunc(GasQuickStep),
validateStack: makeStackFunc(0, 1),
valid:         true,
},
GASLIMIT: {
execute:       opGasLimit,
gasCost:       constGasFunc(GasQuickStep),
validateStack: makeStackFunc(0, 1),
valid:         true,
},
POP: {
execute:       opPop,
gasCost:       constGasFunc(GasQuickStep),
validateStack: makeStackFunc(1, 0),
valid:         true,
},
MLOAD: {
execute:       opMload,
gasCost:       gasMLoad,
validateStack: makeStackFunc(1, 1),
memorySize:    memoryMLoad,
valid:         true,
},
MSTORE: {
execute:       opMstore,
gasCost:       gasMStore,
validateStack: makeStackFunc(2, 0),
memorySize:    memoryMStore,
valid:         true,
},
MSTORE8: {
execute:       opMstore8,
gasCost:       gasMStore8,
memorySize:    memoryMStore8,
validateStack: makeStackFunc(2, 0),

valid: true,
},
SLOAD: {
execute:       opSload,
gasCost:       gasSLoad,
validateStack: makeStackFunc(1, 1),
```

```
valid:        true,
},
SSTORE: {
execute:      opSstore,
gasCost:      gasSStore,
validateStack: makeStackFunc(2, 0),
valid:        true,
writes:       true,
},
JUMP: {
execute:      opJump,
gasCost:      constGasFunc(GasMidStep),
validateStack: makeStackFunc(1, 0),
jumps:        true,
valid:        true,
},
JUMPI: {
execute:      opJumpi,
gasCost:      constGasFunc(GasSlowStep),
validateStack: makeStackFunc(2, 0),
jumps:        true,
valid:        true,
},
PC: {
execute:      opPc,
gasCost:      constGasFunc(GasQuickStep),
validateStack: makeStackFunc(0, 1),
valid:        true,
},
MSIZE: {
execute:      opMsize,
gasCost:      constGasFunc(GasQuickStep),
validateStack: makeStackFunc(0, 1),
valid:        true,
},
GAS: {
execute:      opGas,
gasCost:      constGasFunc(GasQuickStep),
validateStack: makeStackFunc(0, 1),
valid:        true,
},
JUMPDEST: {
```

```
execute:       opJumpdest,
gasCost:       constGasFunc(params.JumpdestGas),
validateStack: makeStackFunc(0, 0),
valid:         true,
},
PUSH1: {
execute:       makePush(1, 1),
gasCost:       gasPush,
validateStack: makeStackFunc(0, 1),
valid:         true,
},
PUSH2: {
execute:       makePush(2, 2),
gasCost:       gasPush,
validateStack: makeStackFunc(0, 1),
valid:         true,
},
PUSH3: {
execute:       makePush(3, 3),
gasCost:       gasPush,
validateStack: makeStackFunc(0, 1),
valid:         true,
},
PUSH4: {
execute:       makePush(4, 4),
gasCost:       gasPush,
validateStack: makeStackFunc(0, 1),
valid:         true,
},
PUSH5: {
execute:       makePush(5, 5),
gasCost:       gasPush,
validateStack: makeStackFunc(0, 1),
valid:         true,
},
PUSH6: {
execute:       makePush(6, 6),
gasCost:       gasPush,
validateStack: makeStackFunc(0, 1),
valid:         true,
},
PUSH7: {
```

```
execute:       makePush(7, 7),
gasCost:       gasPush,
validateStack: makeStackFunc(0, 1),
valid:         true,
},
PUSH8: {
execute:       makePush(8, 8),
gasCost:       gasPush,
validateStack: makeStackFunc(0, 1),
valid:         true,
},
PUSH9: {
execute:       makePush(9, 9),
gasCost:       gasPush,
validateStack: makeStackFunc(0, 1),
valid:         true,
},
PUSH10: {
execute:       makePush(10, 10),
gasCost:       gasPush,
validateStack: makeStackFunc(0, 1),
valid:         true,
},
PUSH11: {
execute:       makePush(11, 11),
gasCost:       gasPush,
validateStack: makeStackFunc(0, 1),
valid:         true,
},
PUSH12: {
execute:       makePush(12, 12),
gasCost:       gasPush,
validateStack: makeStackFunc(0, 1),
valid:         true,
},
PUSH13: {
execute:       makePush(13, 13),
gasCost:       gasPush,
validateStack: makeStackFunc(0, 1),
valid:         true,
},
PUSH14: {
```

```
execute:       makePush(14, 14),
gasCost:       gasPush,
validateStack: makeStackFunc(0, 1),
valid:         true,
},
PUSH15: {
execute:       makePush(15, 15),
gasCost:       gasPush,
validateStack: makeStackFunc(0, 1),
valid:         true,
},
PUSH16: {
execute:       makePush(16, 16),
gasCost:       gasPush,
validateStack: makeStackFunc(0, 1),
valid:         true,
},
PUSH17: {
execute:       makePush(17, 17),
gasCost:       gasPush,
validateStack: makeStackFunc(0, 1),
valid:         true,
},
PUSH18: {
execute:       makePush(18, 18),
gasCost:       gasPush,
validateStack: makeStackFunc(0, 1),
valid:         true,
},
PUSH19: {
execute:       makePush(19, 19),
gasCost:       gasPush,
validateStack: makeStackFunc(0, 1),
valid:         true,
},
PUSH20: {
execute:       makePush(20, 20),
gasCost:       gasPush,
validateStack: makeStackFunc(0, 1),
valid:         true,
},
PUSH21: {
```

```
execute:       makePush(21, 21),
gasCost:       gasPush,
validateStack: makeStackFunc(0, 1),
valid:         true,
},
PUSH22: {
execute:       makePush(22, 22),
gasCost:       gasPush,
validateStack: makeStackFunc(0, 1),
valid:         true,
},
PUSH23: {
execute:       makePush(23, 23),
gasCost:       gasPush,
validateStack: makeStackFunc(0, 1),
valid:         true,
},
PUSH24: {
execute:       makePush(24, 24),
gasCost:       gasPush,
validateStack: makeStackFunc(0, 1),
valid:         true,
},
PUSH25: {
execute:       makePush(25, 25),
gasCost:       gasPush,
validateStack: makeStackFunc(0, 1),
valid:         true,
},
PUSH26: {
execute:       makePush(26, 26),
gasCost:       gasPush,
validateStack: makeStackFunc(0, 1),
valid:         true,
},
PUSH27: {
execute:       makePush(27, 27),
gasCost:       gasPush,
validateStack: makeStackFunc(0, 1),
valid:         true,
},
PUSH28: {
```

```
execute:       makePush(28, 28),
gasCost:       gasPush,
validateStack: makeStackFunc(0, 1),
valid:         true,
},
PUSH29: {
execute:       makePush(29, 29),
gasCost:       gasPush,
validateStack: makeStackFunc(0, 1),
valid:         true,
},
PUSH30: {
execute:       makePush(30, 30),
gasCost:       gasPush,
validateStack: makeStackFunc(0, 1),
valid:         true,
},
PUSH31: {
execute:       makePush(31, 31),
gasCost:       gasPush,
validateStack: makeStackFunc(0, 1),
valid:         true,
},
PUSH32: {
execute:       makePush(32, 32),
gasCost:       gasPush,
validateStack: makeStackFunc(0, 1),
valid:         true,
},
DUP1: {
execute:       makeDup(1),
gasCost:       gasDup,
validateStack: makeDupStackFunc(1),
valid:         true,
},
DUP2: {
execute:       makeDup(2),
gasCost:       gasDup,
validateStack: makeDupStackFunc(2),
valid:         true,
},
DUP3: {
```

```
execute:       makeDup(3),
gasCost:       gasDup,
validateStack: makeDupStackFunc(3),
valid:         true,
},
DUP4: {
execute:       makeDup(4),
gasCost:       gasDup,
validateStack: makeDupStackFunc(4),
valid:         true,
},
DUP5: {
execute:       makeDup(5),
gasCost:       gasDup,
validateStack: makeDupStackFunc(5),
valid:         true,
},
DUP6: {
execute:       makeDup(6),
gasCost:       gasDup,
validateStack: makeDupStackFunc(6),
valid:         true,
},
DUP7: {
execute:       makeDup(7),
gasCost:       gasDup,
validateStack: makeDupStackFunc(7),
valid:         true,
},
DUP8: {
execute:       makeDup(8),
gasCost:       gasDup,
validateStack: makeDupStackFunc(8),
valid:         true,
},
DUP9: {
execute:       makeDup(9),
gasCost:       gasDup,
validateStack: makeDupStackFunc(9),
valid:         true,
},
DUP10: {
```

```
execute:       makeDup(10),
gasCost:       gasDup,
validateStack: makeDupStackFunc(10),
valid:         true,
},
DUP11: {
execute:       makeDup(11),
gasCost:       gasDup,
validateStack: makeDupStackFunc(11),
valid:         true,
},
DUP12: {
execute:       makeDup(12),
gasCost:       gasDup,
validateStack: makeDupStackFunc(12),
valid:         true,
},
DUP13: {
execute:       makeDup(13),
gasCost:       gasDup,
validateStack: makeDupStackFunc(13),
valid:         true,
},
DUP14: {
execute:       makeDup(14),
gasCost:       gasDup,
validateStack: makeDupStackFunc(14),
valid:         true,
},
DUP15: {
execute:       makeDup(15),
gasCost:       gasDup,
validateStack: makeDupStackFunc(15),
valid:         true,
},
DUP16: {
execute:       makeDup(16),
gasCost:       gasDup,
validateStack: makeDupStackFunc(16),
valid:         true,
},
SWAP1: {
```

```
execute:       makeSwap(1),
gasCost:       gasSwap,
validateStack: makeSwapStackFunc(2),
valid:         true,
},
SWAP2: {
execute:       makeSwap(2),
gasCost:       gasSwap,
validateStack: makeSwapStackFunc(3),
valid:         true,
},
SWAP3: {
execute:       makeSwap(3),
gasCost:       gasSwap,
validateStack: makeSwapStackFunc(4),
valid:         true,
},
SWAP4: {
execute:       makeSwap(4),
gasCost:       gasSwap,
validateStack: makeSwapStackFunc(5),
valid:         true,
},
SWAP5: {
execute:       makeSwap(5),
gasCost:       gasSwap,
validateStack: makeSwapStackFunc(6),
valid:         true,
},
SWAP6: {
execute:       makeSwap(6),
gasCost:       gasSwap,
validateStack: makeSwapStackFunc(7),
valid:         true,
},
SWAP7: {
execute:       makeSwap(7),
gasCost:       gasSwap,
validateStack: makeSwapStackFunc(8),
valid:         true,
},
SWAP8: {
```

```
execute:       makeSwap(8),
gasCost:       gasSwap,
validateStack: makeSwapStackFunc(9),
valid:         true,
},
SWAP9: {
execute:       makeSwap(9),
gasCost:       gasSwap,
validateStack: makeSwapStackFunc(10),
valid:         true,
},
SWAP10: {
execute:       makeSwap(10),
gasCost:       gasSwap,
validateStack: makeSwapStackFunc(11),
valid:         true,
},
SWAP11: {
execute:       makeSwap(11),
gasCost:       gasSwap,
validateStack: makeSwapStackFunc(12),
valid:         true,
},
SWAP12: {
execute:       makeSwap(12),
gasCost:       gasSwap,
validateStack: makeSwapStackFunc(13),
valid:         true,
},
SWAP13: {
execute:       makeSwap(13),
gasCost:       gasSwap,
validateStack: makeSwapStackFunc(14),
valid:         true,
},
SWAP14: {
execute:       makeSwap(14),
gasCost:       gasSwap,
validateStack: makeSwapStackFunc(15),
valid:         true,
},
SWAP15: {
```

```
execute:       makeSwap(15),
gasCost:       gasSwap,
validateStack: makeSwapStackFunc(16),
valid:         true,
},
SWAP16: {
execute:       makeSwap(16),
gasCost:       gasSwap,
validateStack: makeSwapStackFunc(17),
valid:         true,
},
LOG0: {
execute:       makeLog(0),
gasCost:       makeGasLog(0),
validateStack: makeStackFunc(2, 0),
memorySize:    memoryLog,
valid:         true,
},
LOG1: {
execute:       makeLog(1),
gasCost:       makeGasLog(1),
validateStack: makeStackFunc(3, 0),
memorySize:    memoryLog,
valid:         true,
},
LOG2: {
execute:       makeLog(2),
gasCost:       makeGasLog(2),
validateStack: makeStackFunc(4, 0),
memorySize:    memoryLog,
valid:         true,
},
LOG3: {
execute:       makeLog(3),
gasCost:       makeGasLog(3),
validateStack: makeStackFunc(5, 0),
memorySize:    memoryLog,
valid:         true,
},
LOG4: {
execute:       makeLog(4),
gasCost:       makeGasLog(4),
```

```
validateStack: makeStackFunc(6, 0),
memorySize:    memoryLog,
valid:         true,
},
CREATE: {
execute:       opCreate,
gasCost:       gasCreate,
validateStack: makeStackFunc(3, 1),
memorySize:    memoryCreate,
valid:         true,
writes:        true,
},
CALL: {
execute:       opCall,
gasCost:       gasCall,
validateStack: makeStackFunc(7, 1),
memorySize:    memoryCall,
valid:         true,
},
CALLCODE: {
execute:       opCallCode,
gasCost:       gasCallCode,
validateStack: makeStackFunc(7, 1),
memorySize:    memoryCall,
valid:         true,
},
RETURN: {
execute:       opReturn,
gasCost:       gasReturn,
validateStack: makeStackFunc(2, 0),
memorySize:    memoryReturn,
halts:         true,
valid:         true,
},
SELFDESTRUCT: {
execute:       opSuicide,
gasCost:       gasSuicide,
validateStack: makeStackFunc(1, 0),
halts:         true,
valid:         true,
writes:        true,
},
```

```
}
}

42:F:\git\coin\ethereum\go-ethereum\core\vm\logger.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package vm

import (
"encoding/hex"
"fmt"
"io"
"math/big"
"time"

"github.com/ethereum/go-ethereum/common"
"github.com/ethereum/go-ethereum/common/hexutil"
"github.com/ethereum/go-ethereum/common/math"
"github.com/ethereum/go-ethereum/core/types"
)

type Storage map[common.Hash]common.Hash

func (self Storage) Copy() Storage {
cpy := make(Storage)
for key, value := range self {
cpy[key] = value
}

return cpy
}

// LogConfig are the configuration options for structured logger the EVM
type LogConfig struct {
DisableMemory  bool // disable memory capture
DisableStack   bool // disable stack capture
DisableStorage bool // disable storage capture
FullStorage    bool // show full storage (slow)
Limit          int  // maximum length of output, but zero means unlimited
}

//go:generate gencodec -type StructLog -field-override structLogMarshaling -out gen_structlog.go
```

```
// StructLog is emitted to the EVM each cycle and lists information about the current internal state
// prior to the execution of the statement.
type StructLog struct {
Pc         uint64                    `json:"pc"`
Op         OpCode                    `json:"op"`
Gas        uint64                    `json:"gas"`
GasCost    uint64                    `json:"gasCost"`
Memory     []byte                    `json:"memory"`
MemorySize int                       `json:"memSize"`
Stack      []*big.Int                `json:"stack"`
Storage    map[common.Hash]common.Hash `json:"-"`
Depth      int                       `json:"depth"`
Err        error                     `json:"error"`
}

// overrides for gencodec
type structLogMarshaling struct {
Stack   []*math.HexOrDecimal256
Gas     math.HexOrDecimal64
GasCost math.HexOrDecimal64
Memory  hexutil.Bytes
OpName  string `json:"opName"`
}

func (s *StructLog) OpName() string {
return s.Op.String()
}

// Tracer is used to collect execution traces from an EVM transaction
// execution. CaptureState is called for each step of the VM with the
// current VM state.
// Note that reference types are actual VM data structures; make copies
// if you need to retain them beyond the current call.
type Tracer interface {
CaptureState(env *EVM, pc uint64, op OpCode, gas, cost uint64, memory *Memory, stack *Stack,
contract *Contract, depth int, err error) error
CaptureEnd(output []byte, gasUsed uint64, t time.Duration) error
}

// StructLogger is an EVM state logger and implements Tracer.
//
```

```
// StructLogger can capture state based on the given Log configuration and also keeps
// a track record of modified storage which is used in reporting snapshots of the
// contract their storage.
type StructLogger struct {
cfg LogConfig

logs          []StructLog
changedValues map[common.Address]Storage
}

// NewStructLogger returns a new logger
func NewStructLogger(cfg *LogConfig) *StructLogger {
logger := &StructLogger{
changedValues: make(map[common.Address]Storage),
}
if cfg != nil {
logger.cfg = *cfg
}
return logger
}

// CaptureState logs a new structured log message and pushes it out to the environment
//
// CaptureState also tracks SSTORE ops to track dirty values.
func (l *StructLogger) CaptureState(env *EVM, pc uint64, op OpCode, gas, cost uint64, memory
*Memory, stack *Stack, contract *Contract, depth int, err error) error {
// check if already accumulated the specified number of logs
if l.cfg.Limit != 0 && l.cfg.Limit <= len(l.logs) {
return ErrTraceLimitReached
}

// initialise new changed values storage container for this contract
// if not present.
if l.changedValues[contract.Address()] == nil {
l.changedValues[contract.Address()] = make(Storage)
}

// capture SSTORE opcodes and determine the changed value and store
// it in the local storage container. NOTE: we do not need to do any
// range checks here because that's already handler prior to calling
// this function.
switch op {
```

```
case SSTORE:
var (
value   = common.BigToHash(stack.data[stack.len()-2])
address = common.BigToHash(stack.data[stack.len()-1])
)
l.changedValues[contract.Address()][address] = value
}

// copy a snapstot of the current memory state to a new buffer
var mem []byte
if !l.cfg.DisableMemory {
mem = make([]byte, len(memory.Data()))
copy(mem, memory.Data())
}

// copy a snapshot of the current stack state to a new buffer
var stck []*big.Int
if !l.cfg.DisableStack {
stck = make([]*big.Int, len(stack.Data()))
for i, item := range stack.Data() {
stck[i] = new(big.Int).Set(item)
}
}

// Copy the storage based on the settings specified in the log config. If full storage
// is disabled (default) we can use the simple Storage.Copy method, otherwise we use
// the state object to query for all values (slow process).
var storage Storage
if !l.cfg.DisableStorage {
if l.cfg.FullStorage {
storage = make(Storage)
// Get the contract account and loop over each storage entry. This may involve looping over
// the trie and is a very expensive process.

env.StateDB.ForEachStorage(contract.Address(), func(key, value common.Hash) bool {
storage[key] = value
// Return true, indicating we'd like to continue.
return true
})
} else {
// copy a snapshot of the current storage to a new container.
storage = l.changedValues[contract.Address()].Copy()
```

```
}
}
// create a new snaptshot of the EVM.
log := StructLog{pc, op, gas, cost, mem, memory.Len(), stck, storage, depth, err}

l.logs = append(l.logs, log)
return nil
}

func (l *StructLogger) CaptureEnd(output []byte, gasUsed uint64, t time.Duration) error {
fmt.Printf("0x%x", output)
return nil
}

// StructLogs returns a list of captured log entries
func (l *StructLogger) StructLogs() []StructLog {
return l.logs
}

// WriteTrace writes a formatted trace to the given writer
func WriteTrace(writer io.Writer, logs []StructLog) {
for _, log := range logs {
fmt.Fprintf(writer, "%-10spc=%08d gas=%v cost=%v", log.Op, log.Pc, log.Gas, log.GasCost)
if log.Err != nil {
fmt.Fprintf(writer, " ERROR: %v", log.Err)
}
fmt.Fprintf(writer, "\n")

for i := len(log.Stack) - 1; i >= 0; i-- {
fmt.Fprintf(writer, "%08d  %x\n", len(log.Stack)-i-1, math.PaddedBigBytes(log.Stack[i], 32))
}

fmt.Fprint(writer, hex.Dump(log.Memory))

for h, item := range log.Storage {
fmt.Fprintf(writer, "%x: %x\n", h, item)
}
fmt.Fprintln(writer)
}
}

// WriteLogs writes vm logs in a readable format to the given writer
```

```
func WriteLogs(writer io.Writer, logs []*types.Log) {
for _, log := range logs {
fmt.Fprintf(writer, "LOG%d: %x bn=%d txi=%x\n", len(log.Topics), log.Address, log.BlockNumber,
log.TxIndex)

for i, topic := range log.Topics {
fmt.Fprintf(writer, "%08d  %x\n", i, topic)
}

fmt.Fprint(writer, hex.Dump(log.Data))
fmt.Fprintln(writer)
}
}
```

43:F:\git\coin\ethereum\go-ethereum\core\vm\logger_test.go

```
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package vm

import (
"math/big"
"testing"

"github.com/ethereum/go-ethereum/common"
"github.com/ethereum/go-ethereum/params"
)

type dummyContractRef struct {
calledForEach bool
}

func (dummyContractRef) ReturnGas(*big.Int)          {}
func (dummyContractRef) Address() common.Address     { return common.Address{} }
func (dummyContractRef) Value() *big.Int             { return new(big.Int) }
func (dummyContractRef) SetCode(common.Hash, []byte) {}
func (d *dummyContractRef) ForEachStorage(callback func(key, value common.Hash) bool) {
d.calledForEach = true
}
func (d *dummyContractRef) SubBalance(amount *big.Int) {}
func (d *dummyContractRef) AddBalance(amount *big.Int) {}
func (d *dummyContractRef) SetBalance(*big.Int)        {}
func (d *dummyContractRef) SetNonce(uint64)            {}
```

```
func (d *dummyContractRef) Balance() *big.Int          { return new(big.Int) }

type dummyStateDB struct {
NoopStateDB
ref *dummyContractRef
}

func TestStoreCapture(t *testing.T) {
var (
env      = NewEVM(Context{}, nil, params.TestChainConfig, Config{EnableJit: false, ForceJit:
false})
logger   = NewStructLogger(nil)
mem      = NewMemory()
stack    = newstack()
contract = NewContract(&dummyContractRef{}, &dummyContractRef{}, new(big.Int), 0)
)
stack.push(big.NewInt(1))
stack.push(big.NewInt(0))

var index common.Hash

logger.CaptureState(env, 0, SSTORE, 0, 0, mem, stack, contract, 0, nil)
if len(logger.changedValues[contract.Address()]) == 0 {
t.Fatalf("expected exactly 1 changed value on address %x, got %d", contract.Address(),
len(logger.changedValues[contract.Address()]))
}

exp := common.BigToHash(big.NewInt(1))
if logger.changedValues[contract.Address()][index] != exp {
t.Errorf("expected %x, got %x", exp, logger.changedValues[contract.Address()][index])
}
}

func TestStorageCapture(t *testing.T) {
t.Skip("implementing this function is difficult. it requires all sort of interfaces to be implemented
which isn't trivial. The value (the actual test) isn't worth it")
var (
ref      = &dummyContractRef{}
contract = NewContract(ref, ref, new(big.Int), 0)
env      = NewEVM(Context{}, dummyStateDB{ref: ref}, params.TestChainConfig,
Config{EnableJit: false, ForceJit: false})
logger   = NewStructLogger(nil)
```

```
mem      = NewMemory()
stack    = newstack()
)

logger.CaptureState(env, 0, STOP, 0, 0, mem, stack, contract, 0, nil)
if ref.calledForEach {
t.Error("didn't expect for each to be called")
}

logger = NewStructLogger(&LogConfig{FullStorage: true})
logger.CaptureState(env, 0, STOP, 0, 0, mem, stack, contract, 0, nil)
if !ref.calledForEach {
t.Error("expected for each to be called")
}
}

44:F:\git\coin\ethereum\go-ethereum\core\vm\memory.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package vm

import "fmt"

// Memory implements a simple memory model for the ethereum virtual machine.
type Memory struct {
store       []byte
lastGasCost uint64
lastReturn  []byte
}

func NewMemory() *Memory {
return &Memory{}
}

// Set sets offset + size to value
func (m *Memory) Set(offset, size uint64, value []byte) {
// length of store may never be less than offset + size.
// The store should be resized PRIOR to setting the memory
if size > uint64(len(m.store)) {
panic("INVALID memory: store empty")
}
```

```
// It's possible the offset is greater than 0 and size equals 0. This is because
// the calcMemSize (common.go) could potentially return 0 when size is zero (NO-OP)
if size > 0 {
copy(m.store[offset:offset+size], value)
}
}

// Resize resizes the memory to size
func (m *Memory) Resize(size uint64) {
if uint64(m.Len()) < size {
m.store = append(m.store, make([]byte, size-uint64(m.Len()))...)
}
}

// Get returns offset + size as a new slice
func (self *Memory) Get(offset, size int64) (cpy []byte) {
if size == 0 {
return nil
}

if len(self.store) > int(offset) {
cpy = make([]byte, size)
copy(cpy, self.store[offset:offset+size])

return
}

return
}

// GetPtr returns the offset + size
func (self *Memory) GetPtr(offset, size int64) []byte {
if size == 0 {
return nil
}

if len(self.store) > int(offset) {
return self.store[offset : offset+size]
}

return nil
}
```

```
// Len returns the length of the backing slice
func (m *Memory) Len() int {
return len(m.store)
}

// Data returns the backing slice
func (m *Memory) Data() []byte {
return m.store
}

func (m *Memory) Print() {
fmt.Printf("### mem %d bytes ###\n", len(m.store))
if len(m.store) > 0 {
addr := 0
for i := 0; i+32 <= len(m.store); i += 32 {
fmt.Printf("%03d: % x\n", addr, m.store[i:i+32])
addr++
}
} else {
fmt.Println("-- empty --")
}
fmt.Println("######################")
}

45:F:\git\coin\ethereum\go-ethereum\core\vm\memory_table.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package vm

import (
"math/big"

"github.com/ethereum/go-ethereum/common/math"
)

func memorySha3(stack *Stack) *big.Int {
return calcMemSize(stack.Back(0), stack.Back(1))
}

func memoryCalldataCopy(stack *Stack) *big.Int {
return calcMemSize(stack.Back(0), stack.Back(2))
```

```
}

func memoryCodeCopy(stack *Stack) *big.Int {
return calcMemSize(stack.Back(0), stack.Back(2))
}

func memoryExtCodeCopy(stack *Stack) *big.Int {
return calcMemSize(stack.Back(1), stack.Back(3))
}

func memoryMLoad(stack *Stack) *big.Int {
return calcMemSize(stack.Back(0), big.NewInt(32))
}

func memoryMStore8(stack *Stack) *big.Int {
return calcMemSize(stack.Back(0), big.NewInt(1))
}

func memoryMStore(stack *Stack) *big.Int {
return calcMemSize(stack.Back(0), big.NewInt(32))
}

func memoryCreate(stack *Stack) *big.Int {
return calcMemSize(stack.Back(1), stack.Back(2))
}

func memoryCall(stack *Stack) *big.Int {
x := calcMemSize(stack.Back(5), stack.Back(6))
y := calcMemSize(stack.Back(3), stack.Back(4))

return math.BigMax(x, y)
}

func memoryCallCode(stack *Stack) *big.Int {
x := calcMemSize(stack.Back(5), stack.Back(6))
y := calcMemSize(stack.Back(3), stack.Back(4))

return math.BigMax(x, y)
}
func memoryDelegateCall(stack *Stack) *big.Int {
x := calcMemSize(stack.Back(4), stack.Back(5))
y := calcMemSize(stack.Back(2), stack.Back(3))
```

```
return math.BigMax(x, y)
}

func memoryReturn(stack *Stack) *big.Int {
return calcMemSize(stack.Back(0), stack.Back(1))
}

func memoryLog(stack *Stack) *big.Int {
mSize, mStart := stack.Back(1), stack.Back(0)
return calcMemSize(mStart, mSize)
}

46:F:\git\coin\ethereum\go-ethereum\core\vm\noop.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package vm

import (
"math/big"

"github.com/ethereum/go-ethereum/common"
"github.com/ethereum/go-ethereum/core/types"
)

func NoopCanTransfer(db StateDB, from common.Address, balance *big.Int) bool {
return true
}
func NoopTransfer(db StateDB, from, to common.Address, amount *big.Int) {}

type NoopEVMCallContext struct{}

func (NoopEVMCallContext) Call(caller ContractRef, addr common.Address, data []byte, gas,
value *big.Int) ([]byte, error) {
return nil, nil
}
func (NoopEVMCallContext) CallCode(caller ContractRef, addr common.Address, data []byte, gas,
value *big.Int) ([]byte, error) {
return nil, nil
}
func (NoopEVMCallContext) Create(caller ContractRef, data []byte, gas, value *big.Int) ([]byte,
common.Address, error) {
```

```
return nil, common.Address{}, nil
}
func (NoopEVMCallContext) DelegateCall(me ContractRef, addr common.Address, data []byte,
gas *big.Int) ([]byte, error) {
return nil, nil
}

type NoopStateDB struct{}

func (NoopStateDB) CreateAccount(common.Address)                                    {}
func (NoopStateDB) SubBalance(common.Address, *big.Int)                             {}
func (NoopStateDB) AddBalance(common.Address, *big.Int)                             {}
func (NoopStateDB) GetBalance(common.Address) *big.Int                              { return nil }
func (NoopStateDB) GetNonce(common.Address) uint64                                  { return 0 }
func (NoopStateDB) SetNonce(common.Address, uint64)                                 {}
func (NoopStateDB) GetCodeHash(common.Address) common.Hash                                 { return
common.Hash{} }
func (NoopStateDB) GetCode(common.Address) []byte                                   { return nil }
func (NoopStateDB) SetCode(common.Address, []byte)                                  {}
func (NoopStateDB) GetCodeSize(common.Address) int                                  { return 0 }
func (NoopStateDB) AddRefund(*big.Int)                                              {}
func (NoopStateDB) GetRefund() *big.Int                                             { return nil }
func (NoopStateDB) GetState(common.Address, common.Hash) common.Hash                       {
return common.Hash{} }
func (NoopStateDB) SetState(common.Address, common.Hash, common.Hash)                      {}
func (NoopStateDB) Suicide(common.Address) bool                                     { return false }
func (NoopStateDB) HasSuicided(common.Address) bool                                 { return false }
func (NoopStateDB) Exist(common.Address) bool                                       { return false }
func (NoopStateDB) Empty(common.Address) bool                                       { return false }
func (NoopStateDB) RevertToSnapshot(int)                                            {}
func (NoopStateDB) Snapshot() int                                                   { return 0 }
func (NoopStateDB) AddLog(*types.Log)                                               {}
func (NoopStateDB) AddPreimage(common.Hash, []byte)                                 {}
func (NoopStateDB) ForEachStorage(common.Address, func(common.Hash, common.Hash)
bool) {}

47:F:\git\coin\ethereum\go-ethereum\core\vm\opcodes.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package vm

import (
```

```
"fmt"
)

// OpCode is an EVM opcode
type OpCode byte

func (op OpCode) IsPush() bool {
switch op {
case PUSH1, PUSH2, PUSH3, PUSH4, PUSH5, PUSH6, PUSH7, PUSH8, PUSH9, PUSH10,
PUSH11, PUSH12, PUSH13, PUSH14, PUSH15, PUSH16, PUSH17, PUSH18, PUSH19,
PUSH20, PUSH21, PUSH22, PUSH23, PUSH24, PUSH25, PUSH26, PUSH27, PUSH28,
PUSH29, PUSH30, PUSH31, PUSH32:
return true
}
return false
}

func (op OpCode) IsStaticJump() bool {
return op == JUMP
}

const (
// 0x0 range - arithmetic ops
STOP OpCode = iota
ADD
MUL
SUB
DIV
SDIV
MOD
SMOD
ADDMOD
MULMOD
EXP
SIGNEXTEND
)

const (
LT OpCode = iota + 0x10
GT
SLT
SGT
```

```
EQ
ISZERO
AND
OR
XOR
NOT
BYTE

SHA3 = 0x20
)

const (
// 0x30 range - closure state
ADDRESS OpCode = 0x30 + iota
BALANCE
ORIGIN
CALLER
CALLVALUE
CALLDATALOAD
CALLDATASIZE
CALLDATACOPY
CODESIZE
CODECOPY
GASPRICE
EXTCODESIZE
EXTCODECOPY
)

const (

// 0x40 range - block operations
BLOCKHASH OpCode = 0x40 + iota
COINBASE
TIMESTAMP
NUMBER
DIFFICULTY
GASLIMIT
)

const (
// 0x50 range - 'storage' and execution
POP OpCode = 0x50 + iota
```

```
MLOAD
MSTORE
MSTORE8
SLOAD
SSTORE
JUMP
JUMPI
PC
MSIZE
GAS
JUMPDEST
)

const (
// 0x60 range
PUSH1 OpCode = 0x60 + iota
PUSH2
PUSH3
PUSH4
PUSH5
PUSH6
PUSH7
PUSH8
PUSH9
PUSH10
PUSH11
PUSH12
PUSH13
PUSH14
PUSH15
PUSH16
PUSH17
PUSH18
PUSH19
PUSH20
PUSH21
PUSH22
PUSH23
PUSH24
PUSH25
PUSH26
PUSH27
```

```
PUSH28
PUSH29
PUSH30
PUSH31
PUSH32
DUP1
DUP2
DUP3
DUP4
DUP5
DUP6
DUP7
DUP8
DUP9
DUP10
DUP11
DUP12
DUP13
DUP14
DUP15
DUP16
SWAP1
SWAP2
SWAP3
SWAP4
SWAP5
SWAP6
SWAP7
SWAP8
SWAP9
SWAP10
SWAP11
SWAP12
SWAP13
SWAP14
SWAP15
SWAP16
)

const (
LOG0 OpCode = 0xa0 + iota
LOG1
```

```
LOG2
LOG3
LOG4
)

// unofficial opcodes used for parsing
const (
PUSH OpCode = 0xb0 + iota
DUP
SWAP
)

const (
// 0xf0 range - closures
CREATE OpCode = 0xf0 + iota
CALL
CALLCODE
RETURN
DELEGATECALL

SELFDESTRUCT = 0xff
)

// Since the opcodes aren't all in order we can't use a regular slice
var opCodeToString = map[OpCode]string{
// 0x0 range - arithmetic ops
STOP:       "STOP",
ADD:        "ADD",
MUL:        "MUL",
SUB:        "SUB",
DIV:        "DIV",
SDIV:       "SDIV",
MOD:        "MOD",
SMOD:       "SMOD",
EXP:        "EXP",
NOT:        "NOT",
LT:         "LT",
GT:         "GT",
SLT:        "SLT",
SGT:        "SGT",
EQ:         "EQ",
ISZERO:     "ISZERO",
```

```
SIGNEXTEND: "SIGNEXTEND",

// 0x10 range - bit ops
AND:    "AND",
OR:     "OR",
XOR:    "XOR",
BYTE:   "BYTE",
ADDMOD: "ADDMOD",
MULMOD: "MULMOD",

// 0x20 range - crypto
SHA3: "SHA3",

// 0x30 range - closure state
ADDRESS:      "ADDRESS",
BALANCE:      "BALANCE",
ORIGIN:       "ORIGIN",
CALLER:       "CALLER",
CALLVALUE:    "CALLVALUE",
CALLDATALOAD: "CALLDATALOAD",
CALLDATASIZE: "CALLDATASIZE",
CALLDATACOPY: "CALLDATACOPY",
CODESIZE:     "CODESIZE",
CODECOPY:     "CODECOPY",
GASPRICE:     "GASPRICE",

// 0x40 range - block operations
BLOCKHASH:   "BLOCKHASH",
COINBASE:    "COINBASE",
TIMESTAMP:   "TIMESTAMP",
NUMBER:      "NUMBER",
DIFFICULTY:  "DIFFICULTY",
GASLIMIT:    "GASLIMIT",
EXTCODESIZE: "EXTCODESIZE",
EXTCODECOPY: "EXTCODECOPY",

// 0x50 range - 'storage' and execution
POP: "POP",
//DUP:     "DUP",
//SWAP:    "SWAP",
MLOAD:    "MLOAD",
MSTORE:   "MSTORE",
```

```
MSTORE8:  "MSTORE8",
SLOAD:    "SLOAD",
SSTORE:   "SSTORE",
JUMP:     "JUMP",
JUMPI:    "JUMPI",
PC:       "PC",
MSIZE:    "MSIZE",
GAS:      "GAS",
JUMPDEST: "JUMPDEST",

// 0x60 range - push
PUSH1:  "PUSH1",
PUSH2:  "PUSH2",
PUSH3:  "PUSH3",
PUSH4:  "PUSH4",
PUSH5:  "PUSH5",
PUSH6:  "PUSH6",
PUSH7:  "PUSH7",
PUSH8:  "PUSH8",
PUSH9:  "PUSH9",
PUSH10: "PUSH10",
PUSH11: "PUSH11",
PUSH12: "PUSH12",
PUSH13: "PUSH13",
PUSH14: "PUSH14",
PUSH15: "PUSH15",
PUSH16: "PUSH16",
PUSH17: "PUSH17",
PUSH18: "PUSH18",
PUSH19: "PUSH19",
PUSH20: "PUSH20",
PUSH21: "PUSH21",
PUSH22: "PUSH22",
PUSH23: "PUSH23",
PUSH24: "PUSH24",
PUSH25: "PUSH25",
PUSH26: "PUSH26",
PUSH27: "PUSH27",
PUSH28: "PUSH28",
PUSH29: "PUSH29",
PUSH30: "PUSH30",
PUSH31: "PUSH31",
```

```
PUSH32: "PUSH32",

DUP1:  "DUP1",
DUP2:  "DUP2",
DUP3:  "DUP3",
DUP4:  "DUP4",
DUP5:  "DUP5",
DUP6:  "DUP6",
DUP7:  "DUP7",
DUP8:  "DUP8",
DUP9:  "DUP9",
DUP10: "DUP10",
DUP11: "DUP11",
DUP12: "DUP12",
DUP13: "DUP13",
DUP14: "DUP14",
DUP15: "DUP15",
DUP16: "DUP16",

SWAP1:  "SWAP1",
SWAP2:  "SWAP2",
SWAP3:  "SWAP3",
SWAP4:  "SWAP4",
SWAP5:  "SWAP5",
SWAP6:  "SWAP6",
SWAP7:  "SWAP7",
SWAP8:  "SWAP8",
SWAP9:  "SWAP9",
SWAP10: "SWAP10",
SWAP11: "SWAP11",
SWAP12: "SWAP12",
SWAP13: "SWAP13",
SWAP14: "SWAP14",
SWAP15: "SWAP15",
SWAP16: "SWAP16",
LOG0:   "LOG0",
LOG1:   "LOG1",
LOG2:   "LOG2",
LOG3:   "LOG3",
LOG4:   "LOG4",

// 0xf0 range
```

```
CREATE:       "CREATE",
CALL:         "CALL",
RETURN:       "RETURN",
CALLCODE:     "CALLCODE",
DELEGATECALL: "DELEGATECALL",
SELFDESTRUCT: "SELFDESTRUCT",

PUSH: "PUSH",
DUP:  "DUP",
SWAP: "SWAP",
}

func (o OpCode) String() string {
str := opCodeToString[o]
if len(str) == 0 {
return fmt.Sprintf("Missing opcode 0x%x", int(o))
}

return str
}

var stringToOp = map[string]OpCode{
"STOP":         STOP,
"ADD":          ADD,
"MUL":          MUL,
"SUB":          SUB,
"DIV":          DIV,
"SDIV":         SDIV,
"MOD":          MOD,
"SMOD":         SMOD,
"EXP":          EXP,
"NOT":          NOT,
"LT":           LT,
"GT":           GT,
"SLT":          SLT,
"SGT":          SGT,
"EQ":           EQ,
"ISZERO":       ISZERO,
"SIGNEXTEND":   SIGNEXTEND,
"AND":          AND,
"OR":           OR,
"XOR":          XOR,
```

```
"BYTE":        BYTE,
"ADDMOD":      ADDMOD,
"MULMOD":      MULMOD,
"SHA3":        SHA3,
"ADDRESS":     ADDRESS,
"BALANCE":     BALANCE,
"ORIGIN":      ORIGIN,
"CALLER":      CALLER,
"CALLVALUE":   CALLVALUE,
"CALLDATALOAD": CALLDATALOAD,
"CALLDATASIZE": CALLDATASIZE,
"CALLDATACOPY": CALLDATACOPY,
"DELEGATECALL": DELEGATECALL,
"CODESIZE":    CODESIZE,
"CODECOPY":    CODECOPY,
"GASPRICE":    GASPRICE,
"BLOCKHASH":   BLOCKHASH,
"COINBASE":    COINBASE,
"TIMESTAMP":   TIMESTAMP,
"NUMBER":      NUMBER,
"DIFFICULTY":  DIFFICULTY,
"GASLIMIT":    GASLIMIT,
"EXTCODESIZE": EXTCODESIZE,
"EXTCODECOPY": EXTCODECOPY,
"POP":         POP,
"MLOAD":       MLOAD,
"MSTORE":      MSTORE,
"MSTORE8":     MSTORE8,
"SLOAD":       SLOAD,
"SSTORE":      SSTORE,
"JUMP":        JUMP,
"JUMPI":       JUMPI,
"PC":          PC,
"MSIZE":       MSIZE,
"GAS":         GAS,
"JUMPDEST":    JUMPDEST,
"PUSH1":       PUSH1,
"PUSH2":       PUSH2,
"PUSH3":       PUSH3,
"PUSH4":       PUSH4,
"PUSH5":       PUSH5,
"PUSH6":       PUSH6,
```

"PUSH7": PUSH7,
"PUSH8": PUSH8,
"PUSH9": PUSH9,
"PUSH10": PUSH10,
"PUSH11": PUSH11,
"PUSH12": PUSH12,
"PUSH13": PUSH13,
"PUSH14": PUSH14,
"PUSH15": PUSH15,
"PUSH16": PUSH16,
"PUSH17": PUSH17,
"PUSH18": PUSH18,
"PUSH19": PUSH19,
"PUSH20": PUSH20,
"PUSH21": PUSH21,
"PUSH22": PUSH22,
"PUSH23": PUSH23,
"PUSH24": PUSH24,
"PUSH25": PUSH25,
"PUSH26": PUSH26,
"PUSH27": PUSH27,
"PUSH28": PUSH28,
"PUSH29": PUSH29,
"PUSH30": PUSH30,
"PUSH31": PUSH31,
"PUSH32": PUSH32,
"DUP1": DUP1,
"DUP2": DUP2,
"DUP3": DUP3,
"DUP4": DUP4,
"DUP5": DUP5,
"DUP6": DUP6,
"DUP7": DUP7,
"DUP8": DUP8,
"DUP9": DUP9,
"DUP10": DUP10,
"DUP11": DUP11,
"DUP12": DUP12,
"DUP13": DUP13,
"DUP14": DUP14,
"DUP15": DUP15,
"DUP16": DUP16,

```
"SWAP1":       SWAP1,
"SWAP2":       SWAP2,
"SWAP3":       SWAP3,
"SWAP4":       SWAP4,
"SWAP5":       SWAP5,
"SWAP6":       SWAP6,
"SWAP7":       SWAP7,
"SWAP8":       SWAP8,
"SWAP9":       SWAP9,
"SWAP10":      SWAP10,
"SWAP11":      SWAP11,
"SWAP12":      SWAP12,
"SWAP13":      SWAP13,
"SWAP14":      SWAP14,
"SWAP15":      SWAP15,
"SWAP16":      SWAP16,
"LOG0":        LOG0,
"LOG1":        LOG1,
"LOG2":        LOG2,
"LOG3":        LOG3,
"LOG4":        LOG4,
"CREATE":      CREATE,
"CALL":        CALL,
"RETURN":      RETURN,
"CALLCODE":    CALLCODE,
"SELFDESTRUCT": SELFDESTRUCT,
}

func StringToOp(str string) OpCode {
return stringToOp[str]
}

48:F:\git\coin\ethereum\go-ethereum\core\vm\runtime\doc.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

// Package runtime provides a basic execution model for executing EVM code.
package runtime

49:F:\git\coin\ethereum\go-ethereum\core\vm\runtime\env.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package runtime
```

```
import (
"math/big"

"github.com/ethereum/go-ethereum/common"
"github.com/ethereum/go-ethereum/core"
"github.com/ethereum/go-ethereum/core/state"
"github.com/ethereum/go-ethereum/core/vm"
)

func NewEnv(cfg *Config, state *state.StateDB) *vm.EVM {
context := vm.Context{
CanTransfer: core.CanTransfer,
Transfer:    core.Transfer,
GetHash:     func(uint64) common.Hash { return common.Hash{} },

Origin:      cfg.Origin,
Coinbase:    cfg.Coinbase,
BlockNumber: cfg.BlockNumber,
Time:        cfg.Time,
Difficulty:  cfg.Difficulty,
GasLimit:    new(big.Int).SetUint64(cfg.GasLimit),
GasPrice:    new(big.Int),
}

return vm.NewEVM(context, cfg.State, cfg.ChainConfig, cfg.EVMConfig)
}

50:F:\git\coin\ethereum\go-ethereum\core\vm\runtime\fuzz.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

// +build gofuzz

package runtime

// Fuzz is the basic entry point for the go-fuzz tool
//
// This returns 1 for valid parsable/runable code, 0
// for invalid opcode.
func Fuzz(input []byte) int {
_, _, err := Execute(input, input, &Config{
GasLimit: 3000000,
```

```
})

// invalid opcode
if err != nil && len(err.Error()) > 6 && string(err.Error()[:7]) == "invalid" {
return 0
}

return 1
}

51:F:\git\coin\ethereum\go-ethereum\core\vm\runtime\runtime.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package runtime

import (
"math"
"math/big"
"time"

"github.com/ethereum/go-ethereum/common"
"github.com/ethereum/go-ethereum/core/state"
"github.com/ethereum/go-ethereum/core/vm"
"github.com/ethereum/go-ethereum/crypto"
"github.com/ethereum/go-ethereum/ethdb"
"github.com/ethereum/go-ethereum/params"
)

// Config is a basic type specifying certain configuration flags for running
// the EVM.
type Config struct {
ChainConfig *params.ChainConfig
Difficulty  *big.Int
Origin      common.Address
Coinbase    common.Address
BlockNumber *big.Int
Time        *big.Int
GasLimit    uint64
GasPrice    *big.Int
Value       *big.Int
DisableJit  bool // "disable" so it's enabled by default
Debug       bool
```

```
EVMConfig   vm.Config

State     *state.StateDB
GetHashFn func(n uint64) common.Hash
}

// sets defaults on the config
func setDefaults(cfg *Config) {
if cfg.ChainConfig == nil {
cfg.ChainConfig = &params.ChainConfig{
ChainId:        big.NewInt(1),
HomesteadBlock: new(big.Int),
DAOForkBlock:   new(big.Int),
DAOForkSupport: false,
EIP150Block:    new(big.Int),
EIP155Block:    new(big.Int),
EIP158Block:    new(big.Int),
}
}

if cfg.Difficulty == nil {
cfg.Difficulty = new(big.Int)
}
if cfg.Time == nil {
cfg.Time = big.NewInt(time.Now().Unix())
}
if cfg.GasLimit == 0 {
cfg.GasLimit = math.MaxUint64
}
if cfg.GasPrice == nil {
cfg.GasPrice = new(big.Int)
}
if cfg.Value == nil {
cfg.Value = new(big.Int)
}
if cfg.BlockNumber == nil {
cfg.BlockNumber = new(big.Int)
}
if cfg.GetHashFn == nil {
cfg.GetHashFn = func(n uint64) common.Hash {
return common.BytesToHash(crypto.Keccak256([]byte(new(big.Int).SetUint64(n).String())))
}
```

```
}
}

// Execute executes the code using the input as call data during the execution.
// It returns the EVM's return value, the new state and an error if it failed.
//
// Executes sets up a in memory, temporarily, environment for the execution of
// the given code. It enabled the JIT by default and make sure that it's restored
// to it's original state afterwards.
func Execute(code, input []byte, cfg *Config) ([]byte, *state.StateDB, error) {
if cfg == nil {
cfg = new(Config)
}
setDefaults(cfg)

if cfg.State == nil {
db, _ := ethdb.NewMemDatabase()
cfg.State, _ = state.New(common.Hash{}, state.NewDatabase(db))
}
var (
address = common.StringToAddress("contract")
vmenv   = NewEnv(cfg, cfg.State)
sender  = vm.AccountRef(cfg.Origin)
)
cfg.State.CreateAccount(address)
// set the receiver's (the executing contract) code for execution.
cfg.State.SetCode(address, code)
// Call the code with the given configuration.
ret, _, err := vmenv.Call(
sender,
common.StringToAddress("contract"),
input,
cfg.GasLimit,
cfg.Value,
)

return ret, cfg.State, err
}

// Create executes the code using the EVM create method
func Create(input []byte, cfg *Config) ([]byte, common.Address, uint64, error) {
if cfg == nil {
```

```
cfg = new(Config)
}
setDefaults(cfg)

if cfg.State == nil {
db, _ := ethdb.NewMemDatabase()
cfg.State, _ = state.New(common.Hash{}, state.NewDatabase(db))
}
var (
vmenv  = NewEnv(cfg, cfg.State)
sender = vm.AccountRef(cfg.Origin)
)

// Call the code with the given configuration.
code, address, leftOverGas, err := vmenv.Create(
sender,
input,
cfg.GasLimit,
cfg.Value,
)
return code, address, leftOverGas, err
}

// Call executes the code given by the contract's address. It will return the
// EVM's return value or an error if it failed.
//
// Call, unlike Execute, requires a config and also requires the State field to
// be set.
func Call(address common.Address, input []byte, cfg *Config) ([]byte, uint64, error) {
setDefaults(cfg)

vmenv := NewEnv(cfg, cfg.State)

sender := cfg.State.GetOrNewStateObject(cfg.Origin)
// Call the code with the given configuration.
ret, leftOverGas, err := vmenv.Call(
sender,
address,
input,
cfg.GasLimit,
cfg.Value,
)
```

```
return ret, leftOverGas, err
}

52:F:\git\coin\ethereum\go-ethereum\core\vm\runtime\runtime_example_test.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package runtime_test

import (
"fmt"

"github.com/ethereum/go-ethereum/common"
"github.com/ethereum/go-ethereum/core/vm/runtime"
)

func ExampleExecute() {
ret, _, err :=
runtime.Execute(common.Hex2Bytes("6060604052600a8060106000396000f3606060405260085
65b00"), nil, nil)
if err != nil {
fmt.Println(err)
}
fmt.Println(ret)
// Output:
// [96 96 96 64 82 96 8 86 91 0]
}

53:F:\git\coin\ethereum\go-ethereum\core\vm\runtime\runtime_test.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package runtime

import (
"math/big"
"strings"
"testing"

"github.com/ethereum/go-ethereum/accounts/abi"
"github.com/ethereum/go-ethereum/common"
"github.com/ethereum/go-ethereum/core/state"
"github.com/ethereum/go-ethereum/core/vm"
```

```
"github.com/ethereum/go-ethereum/ethdb"
)

func TestDefaults(t *testing.T) {
cfg := new(Config)
setDefaults(cfg)

if cfg.Difficulty == nil {
t.Error("expected difficulty to be non nil")
}

if cfg.Time == nil {
t.Error("expected time to be non nil")
}
if cfg.GasLimit == 0 {
t.Error("didn't expect gaslimit to be zero")
}
if cfg.GasPrice == nil {
t.Error("expected time to be non nil")
}
if cfg.Value == nil {
t.Error("expected time to be non nil")
}
if cfg.GetHashFn == nil {
t.Error("expected time to be non nil")
}
if cfg.BlockNumber == nil {
t.Error("expected block number to be non nil")
}
}

func TestEVM(t *testing.T) {
defer func() {
if r := recover(); r != nil {
t.Fatalf("crashed with: %v", r)
}
}()

Execute([]byte{
byte(vm.DIFFICULTY),
byte(vm.TIMESTAMP),
byte(vm.GASLIMIT),
```

```
byte(vm.PUSH1),
byte(vm.ORIGIN),
byte(vm.BLOCKHASH),
byte(vm.COINBASE),
}, nil, nil)
}

func TestExecute(t *testing.T) {
ret, _, err := Execute([]byte{
byte(vm.PUSH1), 10,
byte(vm.PUSH1), 0,
byte(vm.MSTORE),
byte(vm.PUSH1), 32,
byte(vm.PUSH1), 0,
byte(vm.RETURN),
}, nil, nil)
if err != nil {
t.Fatal("didn't expect error", err)
}

num := new(big.Int).SetBytes(ret)
if num.Cmp(big.NewInt(10)) != 0 {
t.Error("Expected 10, got", num)
}
}

func TestCall(t *testing.T) {
db, _ := ethdb.NewMemDatabase()
state, _ := state.New(common.Hash{}, state.NewDatabase(db))
address := common.HexToAddress("0x0a")
state.SetCode(address, []byte{
byte(vm.PUSH1), 10,
byte(vm.PUSH1), 0,
byte(vm.MSTORE),
byte(vm.PUSH1), 32,
byte(vm.PUSH1), 0,
byte(vm.RETURN),
})

ret, _, err := Call(address, nil, &Config{State: state})
if err != nil {
t.Fatal("didn't expect error", err)
```

}

num := new(big.Int).SetBytes(ret)
if num.Cmp(big.NewInt(10)) != 0 {
t.Error("Expected 10, got", num)
}
}

func BenchmarkCall(b *testing.B) {
var definition =
`[{"constant":true,"inputs":[],"name":"seller","outputs":[{"name":"","type":"address"}],"type":"function"
},{"constant":false,"inputs":[],"name":"abort","outputs":[],"type":"function"},{"constant":true,"inputs":[]
,"name":"value","outputs":[{"name":"","type":"uint256"}],"type":"function"},{"constant":false,"inputs":[
],"name":"refund","outputs":[],"type":"function"},{"constant":true,"inputs":[],"name":"buyer","outputs":
[{"name":"","type":"address"}],"type":"function"},{"constant":false,"inputs":[],"name":"confirmReceive
d","outputs":[],"type":"function"},{"constant":true,"inputs":[],"name":"state","outputs":[{"name":"","typ
e":"uint8"}],"type":"function"},{"constant":false,"inputs":[],"name":"confirmPurchase","outputs":[],"typ
e":"function"},{"inputs":[],"type":"constructor"},{"anonymous":false,"inputs":[],"name":"Aborted","type
":"event"},{"anonymous":false,"inputs":[],"name":"PurchaseConfirmed","type":"event"},{"anonymou
s":false,"inputs":[],"name":"ItemReceived","type":"event"},{"anonymous":false,"inputs":[],"name":"R
efunded","type":"event"}]`

var code =
common.Hex2Bytes("6060604052361561006c5760e060020a600035046308551a5381146100745
7806335a063b4146100865780633fa4f245146100a6578063590e1ae3146100af5780637150d8ae
146100cf57806373fac6f0146100e1578063c19d93fb146100fe578063d6960971461011257 5b610
131610002565b6101336004600160a060020a031681565b6101316004600160a060020a039
0811633919091161461015057610002565b6101466005481565b6101316004600160a060020
a0390811633919091161461020d557610002565b6101336002460016a060020a031681565b610
131600254600160a060020a033381169116146102375761000 2565b6101466002546 0ff60a06002
0a9091041681565b6101316002546 0009060ff60a060020a9091041681146101cc57610002565b0
05b600160a060020a0316606090815260209 0f35b606090815260209 0f35b60025460009060a060
020a900460ff16811461016b57610002565b600154600160a060020a0390811690829030163160 6
082818181858883f150506002805460a060020a60ff02191660a160020a179055506040517f72c87
4aeff0b183a56e2b79c71b46e1aed4dee5e09862134b8821ba2fddbf8bf9250a150565b805460020
23414806101dd57610002565b6002805460a060020a60ff021973ffffffffffffffffffffffffffffffffffffffff199091
1633171660a060020a1790557fd5d55c8a68912e9a110618df8d5e2e83b8d83211c57a8ddd1203df
92885dc881826060a15050565b60025460019060a060020a900460ff16811461025257610002565
b6002546000805460016 0a060020a0390921691606082818181858883f1505083546040516001 6
0a060020a0391821694503090911631915082818181858883f150506002805460a060020a60ff02
191660a160020a179055506040517fe89152acd703c9d8c7d28829d443260b411454d45394e7995
815140c8cbcbcf79250a150565b60025460019060a060020a900460ff1681146102f057610002565

```
b60028054600080546001600a060020a03909216929091026060828181858883f150508354604
051600160a060020a03918216945030909116319150828181858883f150506002805460a0600
20a60ff02191660a160020a179055506040517f8616bbbbad963e4e65b1366f1d75dfb63f9e9704bb
bf91fb01bec70849906cf79250a15056")

abi, err := abi.JSON(strings.NewReader(definition))
if err != nil {
b.Fatal(err)
}

cpurchase, err := abi.Pack("confirmPurchase")
if err != nil {
b.Fatal(err)
}
creceived, err := abi.Pack("confirmReceived")
if err != nil {
b.Fatal(err)
}
refund, err := abi.Pack("refund")
if err != nil {
b.Fatal(err)
}

b.ResetTimer()
for i := 0; i < b.N; i++ {
for j := 0; j < 400; j++ {
Execute(code, cpurchase, nil)
Execute(code, creceived, nil)
Execute(code, refund, nil)
}
}
}

54:F:\git\coin\ethereum\go-ethereum\core\vm\stack.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package vm

import (
"fmt"
"math/big"
)
```

```
// stack is an object for basic stack operations. Items popped to the stack are
// expected to be changed and modified. stack does not take care of adding newly
// initialised objects.
type Stack struct {
data []*big.Int
}

func newstack() *Stack {
return &Stack{data: make([]*big.Int, 0, 1024)}
}

func (st *Stack) Data() []*big.Int {
return st.data
}

func (st *Stack) push(d *big.Int) {
// NOTE push limit (1024) is checked in baseCheck
//stackItem := new(big.Int).Set(d)
//st.data = append(st.data, stackItem)
st.data = append(st.data, d)
}
func (st *Stack) pushN(ds ...*big.Int) {
st.data = append(st.data, ds...)
}

func (st *Stack) pop() (ret *big.Int) {
ret = st.data[len(st.data)-1]
st.data = st.data[:len(st.data)-1]
return
}

func (st *Stack) len() int {
return len(st.data)
}

func (st *Stack) swap(n int) {
st.data[st.len()-n], st.data[st.len()-1] = st.data[st.len()-1], st.data[st.len()-n]
}

func (st *Stack) dup(pool *intPool, n int) {
st.push(pool.get().Set(st.data[st.len()-n]))
```

```
}

func (st *Stack) peek() *big.Int {
return st.data[st.len()-1]
}

// Back returns the n'th item in stack
func (st *Stack) Back(n int) *big.Int {
return st.data[st.len()-n-1]
}

func (st *Stack) require(n int) error {
if st.len() < n {
return fmt.Errorf("stack underflow (%d <=> %d)", len(st.data), n)
}
return nil
}

func (st *Stack) Print() {
fmt.Println("### stack ###")
if len(st.data) > 0 {
for i, val := range st.data {
fmt.Printf("%-3d  %v\n", i, val)
}
} else {
fmt.Println("-- empty --")
}
fmt.Println("###############")
}

55:F:\git\coin\ethereum\go-ethereum\core\vm\stack_table.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package vm

import (
"fmt"

"github.com/ethereum/go-ethereum/params"
)

func makeStackFunc(pop, push int) stackValidationFunc {
```

```
return func(stack *Stack) error {
if err := stack.require(pop); err != nil {
return err
}

if stack.len()+push-pop > int(params.StackLimit) {
return fmt.Errorf("stack limit reached %d (%d)", stack.len(), params.StackLimit)
}
return nil
}
}

func makeDupStackFunc(n int) stackValidationFunc {
return makeStackFunc(n, n+1)
}

func makeSwapStackFunc(n int) stackValidationFunc {
return makeStackFunc(n, n)
}

56:F:\git\coin\ethereum\go-ethereum\core\vm\vm_jit.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

// +build evmjit

package vm

/*

void* evmjit_create();
int   evmjit_run(void* _jit, void* _data, void* _env);
void  evmjit_destroy(void* _jit);

// Shared library evmjit (e.g. libevmjit.so) is expected to be installed in /usr/local/lib
// More: https://github.com/ethereum/evmjit
#cgo LDFLAGS: -levmjit
*/
import "C"

/*
import (
"bytes"
```

```
"errors"
"fmt"
"math/big"
"unsafe"

"github.com/ethereum/go-ethereum/core/state"
"github.com/ethereum/go-ethereum/crypto"
"github.com/ethereum/go-ethereum/params"
)

type JitVm struct {
env        EVM
me         ContextRef
callerAddr []byte
price      *big.Int
data       RuntimeData
}

type i256 [32]byte

type RuntimeData struct {
gas          int64
gasPrice     int64
callData     *byte
callDataSize uint64
address      i256
caller       i256
origin       i256
callValue    i256
coinBase     i256
difficulty   i256
gasLimit     i256
number       uint64
timestamp    int64
code         *byte
codeSize     uint64
codeHash     i256
}

func hash2llvm(h []byte) i256 {
var m i256
copy(m[len(m)-len(h):], h) // right aligned copy
```

```
return m
}

func llvm2hash(m *i256) []byte {
return C.GoBytes(unsafe.Pointer(m), C.int(len(m)))
}

func llvm2hashRef(m *i256) []byte {
return (*[1 << 30]byte)(unsafe.Pointer(m))[:len(m):len(m)]
}

func address2llvm(addr []byte) i256 {
n := hash2llvm(addr)
bswap(&n)
return n
}

// bswap swap bytes of the 256-bit integer on LLVM side
// TODO: Do not change memory on LLVM side, that can conflict with memory access
optimizations
func bswap(m *i256) *i256 {
for i, l := 0, len(m); i < l/2; i++ {
m[i], m[l-i-1] = m[l-i-1], m[i]
}
return m
}

func trim(m []byte) []byte {
skip := 0
for i := 0; i < len(m); i++ {
if m[i] == 0 {
skip++
} else {
break
}
}
return m[skip:]
}

func getDataPtr(m []byte) *byte {
var p *byte
if len(m) > 0 {
```

```
p = &m[0]
}
return p
}

func big2llvm(n *big.Int) i256 {
m := hash2llvm(n.Bytes())
bswap(&m)
return m
}

func llvm2big(m *i256) *big.Int {
n := big.NewInt(0)
for i := 0; i < len(m); i++ {
b := big.NewInt(int64(m[i]))
b.Lsh(b, uint(i)*8)
n.Add(n, b)
}
return n
}

// llvm2bytesRef creates a []byte slice that references byte buffer on LLVM side (as of that not controller by GC)
// User must ensure that referenced memory is available to Go until the data is copied or not needed any more
func llvm2bytesRef(data *byte, length uint64) []byte {
if length == 0 {
return nil
}
if data == nil {
panic("Unexpected nil data pointer")
}
return (*[1 << 30]byte)(unsafe.Pointer(data))[:length:length]
}

func untested(condition bool, message string) {
if condition {
panic("Condition `" + message + "` tested. Remove assert.")
}
}

func assert(condition bool, message string) {
```

```
if !condition {
panic("Assert `" + message + "` failed!")
}
}

func NewJitVm(env EVM) *JitVm {
return &JitVm{env: env}
}

func (self *JitVm) Run(me, caller ContextRef, code []byte, value, gas, price *big.Int, callData []byte)
(ret []byte, err error) {
// TODO: depth is increased but never checked by VM. VM should not know about it at all.
self.env.SetDepth(self.env.Depth() + 1)

// TODO: Move it to Env.Call() or sth
if Precompiled[string(me.Address())] != nil {
// if it's address of precompiled contract
// fallback to standard VM
stdVm := New(self.env)
return stdVm.Run(me, caller, code, value, gas, price, callData)
}

if self.me != nil {
panic("JitVm.Run() can be called only once per JitVm instance")
}

self.me = me
self.callerAddr = caller.Address()
self.price = price

self.data.gas = gas.Int64()
self.data.gasPrice = price.Int64()
self.data.callData = getDataPtr(callData)
self.data.callDataSize = uint64(len(callData))
self.data.address = address2llvm(self.me.Address())
self.data.caller = address2llvm(caller.Address())
self.data.origin = address2llvm(self.env.Origin())
self.data.callValue = big2llvm(value)
self.data.coinBase = address2llvm(self.env.Coinbase())
self.data.difficulty = big2llvm(self.env.Difficulty())
self.data.gasLimit = big2llvm(self.env.GasLimit())
self.data.number = self.env.BlockNumber().Uint64()
```

```
self.data.timestamp = self.env.Time()
self.data.code = getDataPtr(code)
self.data.codeSize = uint64(len(code))
self.data.codeHash = hash2llvm(crypto.Keccak256(code)) // TODO: Get already computed hash?

jit := C.evmjit_create()
retCode := C.evmjit_run(jit, unsafe.Pointer(&self.data), unsafe.Pointer(self))

if retCode < 0 {
err = errors.New("OOG from JIT")
gas.SetInt64(0) // Set gas to 0, JIT does not bother
} else {
gas.SetInt64(self.data.gas)
if retCode == 1 { // RETURN
ret = C.GoBytes(unsafe.Pointer(self.data.callData), C.int(self.data.callDataSize))
} else if retCode == 2 { // SUICIDE
// TODO: Suicide support logic should be moved to Env to be shared by VM implementations
state := self.Env().State()
receiverAddr := llvm2hashRef(bswap(&self.data.address))
receiver := state.GetOrNewStateObject(receiverAddr)
balance := state.GetBalance(me.Address())
receiver.AddBalance(balance)
state.Delete(me.Address())
}
}

C.evmjit_destroy(jit)
return
}

func (self *JitVm) Printf(format string, v ...interface{}) VirtualMachine {
return self
}

func (self *JitVm) Endl() VirtualMachine {
return self
}

func (self *JitVm) Env() EVM {
return self.env
}
```

```
//export env_sha3
func env_sha3(dataPtr *byte, length uint64, resultPtr unsafe.Pointer) {
data := llvm2bytesRef(dataPtr, length)
hash := crypto.Keccak256(data)
result := (*i256)(resultPtr)
*result = hash2llvm(hash)
}

//export env_sstore
func env_sstore(vmPtr unsafe.Pointer, indexPtr unsafe.Pointer, valuePtr unsafe.Pointer) {
vm := (*JitVm)(vmPtr)
index := llvm2hash(bswap((*i256)(indexPtr)))
value := llvm2hash(bswap((*i256)(valuePtr)))
value = trim(value)
if len(value) == 0 {
prevValue := vm.env.State().GetState(vm.me.Address(), index)
if len(prevValue) != 0 {
vm.Env().State().Refund(vm.callerAddr, GasSStoreRefund)
}
}

vm.env.State().SetState(vm.me.Address(), index, value)
}

//export env_sload
func env_sload(vmPtr unsafe.Pointer, indexPtr unsafe.Pointer, resultPtr unsafe.Pointer) {
vm := (*JitVm)(vmPtr)
index := llvm2hash(bswap((*i256)(indexPtr)))
value := vm.env.State().GetState(vm.me.Address(), index)
result := (*i256)(resultPtr)
*result = hash2llvm(value)
bswap(result)
}

//export env_balance
func env_balance(_vm unsafe.Pointer, _addr unsafe.Pointer, _result unsafe.Pointer) {
vm := (*JitVm)(_vm)
addr := llvm2hash((*i256)(_addr))
balance := vm.Env().State().GetBalance(addr)
result := (*i256)(_result)
*result = big2llvm(balance)
}
```

```
//export env_blockhash
func env_blockhash(_vm unsafe.Pointer, _number unsafe.Pointer, _result unsafe.Pointer) {
vm := (*JitVm)(_vm)
number := llvm2big((*i256)(_number))
result := (*i256)(_result)

currNumber := vm.Env().BlockNumber()
limit := big.NewInt(0).Sub(currNumber, big.NewInt(256))
if number.Cmp(limit) >= 0 && number.Cmp(currNumber) < 0 {
hash := vm.Env().GetHash(uint64(number.Int64()))
*result = hash2llvm(hash)
} else {
*result = i256{}
}
}

//export env_call
func env_call(_vm unsafe.Pointer, _gas *int64, _receiveAddr unsafe.Pointer, _value
unsafe.Pointer, inDataPtr unsafe.Pointer, inDataLen uint64, outDataPtr *byte, outDataLen uint64,
_codeAddr unsafe.Pointer) bool {
vm := (*JitVm)(_vm)

//fmt.Printf("env_call (depth %d)\n", vm.Env().Depth())

defer func() {
if r := recover(); r != nil {
fmt.Printf("Recovered in env_call (depth %d, out %p %d): %s\n", vm.Env().Depth(), outDataPtr,
outDataLen, r)
}
}()

balance := vm.Env().State().GetBalance(vm.me.Address())
value := llvm2big((*i256)(_value))

if balance.Cmp(value) >= 0 {
receiveAddr := llvm2hash((*i256)(_receiveAddr))
inData := C.GoBytes(inDataPtr, C.int(inDataLen))
outData := llvm2bytesRef(outDataPtr, outDataLen)
codeAddr := llvm2hash((*i256)(_codeAddr))
gas := big.NewInt(*_gas)
var out []byte
```

```
var err error
if bytes.Equal(codeAddr, receiveAddr) {
out, err = vm.env.Call(vm.me, codeAddr, inData, gas, vm.price, value)
} else {
out, err = vm.env.CallCode(vm.me, codeAddr, inData, gas, vm.price, value)
}
*_gas = gas.Int64()
if err == nil {
copy(outData, out)
return true
}
}

return false
}

//export env_create
func env_create(_vm unsafe.Pointer, _gas *int64, _value unsafe.Pointer, initDataPtr
unsafe.Pointer, initDataLen uint64, _result unsafe.Pointer) {
vm := (*JitVm)(_vm)

value := llvm2big((*i256)(_value))
initData := C.GoBytes(initDataPtr, C.int(initDataLen)) // TODO: Unnecessary if low balance
result := (*i256)(_result)
*result = i256{}

gas := big.NewInt(*_gas)
ret, suberr, ref := vm.env.Create(vm.me, nil, initData, gas, vm.price, value)
if suberr == nil {
dataGas := big.NewInt(int64(len(ret))) // TODO: Not the best design. env.Create can do it, it has
the reference to gas counter
dataGas.Mul(dataGas, params.CreateDataGas)
gas.Sub(gas, dataGas)
*result = hash2llvm(ref.Address())
}
*_gas = gas.Int64()
}

//export env_log
func env_log(_vm unsafe.Pointer, dataPtr unsafe.Pointer, dataLen uint64, _topic1 unsafe.Pointer,
_topic2 unsafe.Pointer, _topic3 unsafe.Pointer, _topic4 unsafe.Pointer) {
vm := (*JitVm)(_vm)
```

```
data := C.GoBytes(dataPtr, C.int(dataLen))

topics := make([][]byte, 0, 4)
if _topic1 != nil {
topics = append(topics, llvm2hash((*i256)(_topic1)))
}
if _topic2 != nil {
topics = append(topics, llvm2hash((*i256)(_topic2)))
}
if _topic3 != nil {
topics = append(topics, llvm2hash((*i256)(_topic3)))
}
if _topic4 != nil {
topics = append(topics, llvm2hash((*i256)(_topic4)))
}

vm.Env().AddLog(state.NewLog(vm.me.Address(), topics, data, vm.env.BlockNumber().Uint64()))
}

//export env_extcode
func env_extcode(_vm unsafe.Pointer, _addr unsafe.Pointer, o_size *uint64) *byte {
vm := (*JitVm)(_vm)
addr := llvm2hash((*i256)(_addr))
code := vm.Env().State().GetCode(addr)
*o_size = uint64(len(code))
return getDataPtr(code)
}*/

57:F:\git\coin\ethereum\go-ethereum\core\vm\vm_jit_fake.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

// +build !evmjit

package vm

58:F:\git\coin\ethereum\go-ethereum\crypto\bn256\bn256.go
// http://cryptojedi.org/papers/dclxvi-20100714.pdf. Its output is compatible
// with the implementation described in that paper.
package bn256

import (
```

```
"crypto/rand"
"io"
"math/big"
)

// BUG(agl): this implementation is not constant time.
// TODO(agl): keep GF(p²) elements in Mongomery form.

// G1 is an abstract cyclic group. The zero value is suitable for use as the
// output of an operation, but cannot be used as an input.
type G1 struct {
p *curvePoint
}

// RandomG1 returns x and g where x is a random, non-zero number read from r.
func RandomG1(r io.Reader) (*big.Int, *G1, error) {
var k *big.Int
var err error

for {
k, err = rand.Int(r, Order)
if err != nil {
return nil, nil, err
}
if k.Sign() > 0 {
break
}
}

return k, new(G1).ScalarBaseMult(k), nil
}

func (g *G1) String() string {
return "bn256.G1" + g.p.String()
}

// CurvePoints returns p's curve points in big integer
func (e *G1) CurvePoints() (*big.Int, *big.Int, *big.Int, *big.Int) {
return e.p.x, e.p.y, e.p.z, e.p.t
}

// ScalarBaseMult sets e to g*k where g is the generator of the group and
```

```
// then returns e.
func (e *G1) ScalarBaseMult(k *big.Int) *G1 {
if e.p == nil {
e.p = newCurvePoint(nil)
}
e.p.Mul(curveGen, k, new(bnPool))
return e
}

// ScalarMult sets e to a*k and then returns e.
func (e *G1) ScalarMult(a *G1, k *big.Int) *G1 {
if e.p == nil {
e.p = newCurvePoint(nil)
}
e.p.Mul(a.p, k, new(bnPool))
return e
}

// Add sets e to a+b and then returns e.
// BUG(agl): this function is not complete: a==b fails.
func (e *G1) Add(a, b *G1) *G1 {
if e.p == nil {
e.p = newCurvePoint(nil)
}
e.p.Add(a.p, b.p, new(bnPool))
return e
}

// Neg sets e to -a and then returns e.
func (e *G1) Neg(a *G1) *G1 {
if e.p == nil {
e.p = newCurvePoint(nil)
}
e.p.Negative(a.p)
return e
}

// Marshal converts n to a byte slice.
func (n *G1) Marshal() []byte {
n.p.MakeAffine(nil)

xBytes := new(big.Int).Mod(n.p.x, P).Bytes()
```

```
yBytes := new(big.Int).Mod(n.p.y, P).Bytes()

// Each value is a 256-bit number.
const numBytes = 256 / 8

ret := make([]byte, numBytes*2)
copy(ret[1*numBytes-len(xBytes):], xBytes)
copy(ret[2*numBytes-len(yBytes):], yBytes)

return ret
}

// Unmarshal sets e to the result of converting the output of Marshal back into
// a group element and then returns e.
func (e *G1) Unmarshal(m []byte) (*G1, bool) {
// Each value is a 256-bit number.
const numBytes = 256 / 8

if len(m) != 2*numBytes {
return nil, false
}

if e.p == nil {
e.p = newCurvePoint(nil)
}

e.p.x.SetBytes(m[0*numBytes : 1*numBytes])
e.p.y.SetBytes(m[1*numBytes : 2*numBytes])

if e.p.x.Sign() == 0 && e.p.y.Sign() == 0 {
// This is the point at infinity.
e.p.y.SetInt64(1)
e.p.z.SetInt64(0)
e.p.t.SetInt64(0)
} else {
e.p.z.SetInt64(1)
e.p.t.SetInt64(1)

if !e.p.IsOnCurve() {
return nil, false
}
}
```

```
return e, true
}

// G2 is an abstract cyclic group. The zero value is suitable for use as the
// output of an operation, but cannot be used as an input.
type G2 struct {
p *twistPoint
}

// RandomG1 returns x and g where x is a random, non-zero number read from r.
func RandomG2(r io.Reader) (*big.Int, *G2, error) {
var k *big.Int
var err error

for {
k, err = rand.Int(r, Order)
if err != nil {
return nil, nil, err
}
if k.Sign() > 0 {
break
}
}

return k, new(G2).ScalarBaseMult(k), nil
}

func (g *G2) String() string {
return "bn256.G2" + g.p.String()
}

// CurvePoints returns the curve points of p which includes the real
// and imaginary parts of the curve point.
func (e *G2) CurvePoints() (*gfP2, *gfP2, *gfP2, *gfP2) {
return e.p.x, e.p.y, e.p.z, e.p.t
}

// ScalarBaseMult sets e to g*k where g is the generator of the group and
// then returns out.
func (e *G2) ScalarBaseMult(k *big.Int) *G2 {
if e.p == nil {
```

```
e.p = newTwistPoint(nil)
}
e.p.Mul(twistGen, k, new(bnPool))
return e
}

// ScalarMult sets e to a*k and then returns e.
func (e *G2) ScalarMult(a *G2, k *big.Int) *G2 {
if e.p == nil {
e.p = newTwistPoint(nil)
}
e.p.Mul(a.p, k, new(bnPool))
return e
}

// Add sets e to a+b and then returns e.
// BUG(agl): this function is not complete: a==b fails.
func (e *G2) Add(a, b *G2) *G2 {
if e.p == nil {
e.p = newTwistPoint(nil)
}
e.p.Add(a.p, b.p, new(bnPool))
return e
}

// Marshal converts n into a byte slice.
func (n *G2) Marshal() []byte {
n.p.MakeAffine(nil)

xxBytes := new(big.Int).Mod(n.p.x.x, P).Bytes()
xyBytes := new(big.Int).Mod(n.p.x.y, P).Bytes()
yxBytes := new(big.Int).Mod(n.p.y.x, P).Bytes()
yyBytes := new(big.Int).Mod(n.p.y.y, P).Bytes()

// Each value is a 256-bit number.
const numBytes = 256 / 8

ret := make([]byte, numBytes*4)
copy(ret[1*numBytes-len(xxBytes):], xxBytes)
copy(ret[2*numBytes-len(xyBytes):], xyBytes)
copy(ret[3*numBytes-len(yxBytes):], yxBytes)
copy(ret[4*numBytes-len(yyBytes):], yyBytes)
```

```
return ret
}

// Unmarshal sets e to the result of converting the output of Marshal back into
// a group element and then returns e.
func (e *G2) Unmarshal(m []byte) (*G2, bool) {
// Each value is a 256-bit number.
const numBytes = 256 / 8

if len(m) != 4*numBytes {
return nil, false
}

if e.p == nil {
e.p = newTwistPoint(nil)
}

e.p.x.x.SetBytes(m[0*numBytes : 1*numBytes])
e.p.x.y.SetBytes(m[1*numBytes : 2*numBytes])
e.p.y.x.SetBytes(m[2*numBytes : 3*numBytes])
e.p.y.y.SetBytes(m[3*numBytes : 4*numBytes])

if e.p.x.x.Sign() == 0 &&
e.p.x.y.Sign() == 0 &&
e.p.y.x.Sign() == 0 &&
e.p.y.y.Sign() == 0 {
// This is the point at infinity.
e.p.y.SetOne()
e.p.z.SetZero()
e.p.t.SetZero()
} else {
e.p.z.SetOne()
e.p.t.SetOne()

if !e.p.IsOnCurve() {
return nil, false
}
}

return e, true
}
```

```
// GT is an abstract cyclic group. The zero value is suitable for use as the
// output of an operation, but cannot be used as an input.
type GT struct {
p *gfP12
}

func (g *GT) String() string {
return "bn256.GT" + g.p.String()
}

// ScalarMult sets e to a*k and then returns e.
func (e *GT) ScalarMult(a *GT, k *big.Int) *GT {
if e.p == nil {
e.p = newGFp12(nil)
}
e.p.Exp(a.p, k, new(bnPool))
return e
}

// Add sets e to a+b and then returns e.
func (e *GT) Add(a, b *GT) *GT {
if e.p == nil {
e.p = newGFp12(nil)
}
e.p.Mul(a.p, b.p, new(bnPool))
return e
}

// Neg sets e to -a and then returns e.
func (e *GT) Neg(a *GT) *GT {
if e.p == nil {
e.p = newGFp12(nil)
}
e.p.Invert(a.p, new(bnPool))
return e
}

// Marshal converts n into a byte slice.
func (n *GT) Marshal() []byte {
n.p.Minimal()
```

```
xxxBytes := n.p.x.x.x.Bytes()
xxyBytes := n.p.x.x.y.Bytes()
xyxBytes := n.p.x.y.x.Bytes()
xyyBytes := n.p.x.y.y.Bytes()
xzxBytes := n.p.x.z.x.Bytes()
xzyBytes := n.p.x.z.y.Bytes()
yxxBytes := n.p.y.x.x.Bytes()
yxyBytes := n.p.y.x.y.Bytes()
yyxBytes := n.p.y.y.x.Bytes()
yyyBytes := n.p.y.y.y.Bytes()
yzxBytes := n.p.y.z.x.Bytes()
yzyBytes := n.p.y.z.y.Bytes()

// Each value is a 256-bit number.
const numBytes = 256 / 8

ret := make([]byte, numBytes*12)
copy(ret[1*numBytes-len(xxxBytes):], xxxBytes)
copy(ret[2*numBytes-len(xxyBytes):], xxyBytes)
copy(ret[3*numBytes-len(xyxBytes):], xyxBytes)
copy(ret[4*numBytes-len(xyyBytes):], xyyBytes)
copy(ret[5*numBytes-len(xzxBytes):], xzxBytes)
copy(ret[6*numBytes-len(xzyBytes):], xzyBytes)
copy(ret[7*numBytes-len(yxxBytes):], yxxBytes)
copy(ret[8*numBytes-len(yxyBytes):], yxyBytes)
copy(ret[9*numBytes-len(yyxBytes):], yyxBytes)
copy(ret[10*numBytes-len(yyyBytes):], yyyBytes)
copy(ret[11*numBytes-len(yzxBytes):], yzxBytes)
copy(ret[12*numBytes-len(yzyBytes):], yzyBytes)

return ret
}

// Unmarshal sets e to the result of converting the output of Marshal back into
// a group element and then returns e.
func (e *GT) Unmarshal(m []byte) (*GT, bool) {
// Each value is a 256-bit number.
const numBytes = 256 / 8

if len(m) != 12*numBytes {
return nil, false
}
```

```
if e.p == nil {
e.p = newGFp12(nil)
}

e.p.x.x.x.SetBytes(m[0*numBytes : 1*numBytes])
e.p.x.x.y.SetBytes(m[1*numBytes : 2*numBytes])
e.p.x.y.x.SetBytes(m[2*numBytes : 3*numBytes])
e.p.x.y.y.SetBytes(m[3*numBytes : 4*numBytes])
e.p.x.z.x.SetBytes(m[4*numBytes : 5*numBytes])
e.p.x.z.y.SetBytes(m[5*numBytes : 6*numBytes])
e.p.y.x.x.SetBytes(m[6*numBytes : 7*numBytes])
e.p.y.x.y.SetBytes(m[7*numBytes : 8*numBytes])
e.p.y.y.x.SetBytes(m[8*numBytes : 9*numBytes])
e.p.y.y.y.SetBytes(m[9*numBytes : 10*numBytes])
e.p.y.z.x.SetBytes(m[10*numBytes : 11*numBytes])
e.p.y.z.y.SetBytes(m[11*numBytes : 12*numBytes])

return e, true
}

// Pair calculates an Optimal Ate pairing.
func Pair(g1 *G1, g2 *G2) *GT {
return &GT{optimalAte(g2.p, g1.p, new(bnPool))}
}

func PairingCheck(a []*G1, b []*G2) bool {
pool := new(bnPool)
e := newGFp12(pool)
e.SetOne()
for i := 0; i < len(a); i++ {
new_e := miller(b[i].p, a[i].p, pool)
e.Mul(e, new_e, pool)
}
ret := finalExponentiation(e, pool)
e.Put(pool)
return ret.IsOne()
}

// bnPool implements a tiny cache of *big.Int objects that's used to reduce the
// number of allocations made during processing.
type bnPool struct {
```

```
bns   []*big.Int
count int
}

func (pool *bnPool) Get() *big.Int {
if pool == nil {
return new(big.Int)
}

pool.count++
l := len(pool.bns)
if l == 0 {
return new(big.Int)
}

bn := pool.bns[l-1]
pool.bns = pool.bns[:l-1]
return bn
}

func (pool *bnPool) Put(bn *big.Int) {
if pool == nil {
return
}
pool.bns = append(pool.bns, bn)
pool.count--
}

func (pool *bnPool) Count() int {
return pool.count
}

59:F:\git\coin\ethereum\go-ethereum\crypto\bn256\bn256_test.go
pool := new(bnPool)

a := newGFp2(pool)
a.x.SetString("23423492374", 10)
a.y.SetString("12934872398472394827398470", 10)

inv := newGFp2(pool)
inv.Invert(a, pool)
```

```
b := newGFp2(pool).Mul(inv, a, pool)
if b.x.Int64() != 0 || b.y.Int64() != 1 {
t.Fatalf("bad result for a^-1*a: %s %s", b.x, b.y)
}

a.Put(pool)
b.Put(pool)
inv.Put(pool)

if c := pool.Count(); c > 0 {
t.Errorf("Pool count non-zero: %d\n", c)
}
}

func isZero(n *big.Int) bool {
return new(big.Int).Mod(n, P).Int64() == 0
}

func isOne(n *big.Int) bool {
return new(big.Int).Mod(n, P).Int64() == 1
}

func TestGFp6Invert(t *testing.T) {
pool := new(bnPool)

a := newGFp6(pool)
a.x.x.SetString("239487238491", 10)
a.x.y.SetString("2356249827341", 10)
a.y.x.SetString("082659782", 10)
a.y.y.SetString("182703523765", 10)
a.z.x.SetString("978236549263", 10)
a.z.y.SetString("64893242", 10)

inv := newGFp6(pool)
inv.Invert(a, pool)

b := newGFp6(pool).Mul(inv, a, pool)
if !isZero(b.x.x) ||
!isZero(b.x.y) ||
!isZero(b.y.x) ||
!isZero(b.y.y) ||
!isZero(b.z.x) ||
```

```
!isOne(b.z.y) {
t.Fatalf("bad result for a^-1*a: %s", b)
}

a.Put(pool)
b.Put(pool)
inv.Put(pool)

if c := pool.Count(); c > 0 {
t.Errorf("Pool count non-zero: %d\n", c)
}
}

func TestGFp12Invert(t *testing.T) {
pool := new(bnPool)

a := newGFp12(pool)
a.x.x.x.SetString("239846234862342323958623", 10)
a.x.x.y.SetString("2359862352529835623", 10)
a.x.y.x.SetString("928836523", 10)
a.x.y.y.SetString("9856234", 10)
a.x.z.x.SetString("235635286", 10)
a.x.z.y.SetString("5628392833", 10)
a.y.x.x.SetString("252936598265329856238956532167968", 10)
a.y.x.y.SetString("23596239865236954178968", 10)
a.y.y.x.SetString("95421692834", 10)
a.y.y.y.SetString("236548", 10)
a.y.z.x.SetString("924523", 10)
a.y.z.y.SetString("12954623", 10)

inv := newGFp12(pool)
inv.Invert(a, pool)

b := newGFp12(pool).Mul(inv, a, pool)
if !isZero(b.x.x.x) ||
!isZero(b.x.x.y) ||
!isZero(b.x.y.x) ||
!isZero(b.x.y.y) ||
!isZero(b.x.z.x) ||
!isZero(b.x.z.y) ||
!isZero(b.y.x.x) ||
!isZero(b.y.x.y) ||
```

```
!isZero(b.y.y.x) ||
!isZero(b.y.y.y) ||
!isZero(b.y.z.x) ||
!isOne(b.y.z.y) {
t.Fatalf("bad result for a^-1*a: %s", b)
}

a.Put(pool)
b.Put(pool)
inv.Put(pool)

if c := pool.Count(); c > 0 {
t.Errorf("Pool count non-zero: %d\n", c)
}
}

func TestCurveImpl(t *testing.T) {
pool := new(bnPool)

g := &curvePoint{
pool.Get().SetInt64(1),
pool.Get().SetInt64(-2),
pool.Get().SetInt64(1),
pool.Get().SetInt64(0),
}

x := pool.Get().SetInt64(32498273234)
X := newCurvePoint(pool).Mul(g, x, pool)

y := pool.Get().SetInt64(98732423523)
Y := newCurvePoint(pool).Mul(g, y, pool)

s1 := newCurvePoint(pool).Mul(X, y, pool).MakeAffine(pool)
s2 := newCurvePoint(pool).Mul(Y, x, pool).MakeAffine(pool)

if s1.x.Cmp(s2.x) != 0 ||
s2.x.Cmp(s1.x) != 0 {
t.Errorf("DH points don't match: (%s, %s) (%s, %s)", s1.x, s1.y, s2.x, s2.y)
}

pool.Put(x)
X.Put(pool)
```

```
pool.Put(y)
Y.Put(pool)
s1.Put(pool)
s2.Put(pool)
g.Put(pool)

if c := pool.Count(); c > 0 {
t.Errorf("Pool count non-zero: %d\n", c)
}
}

func TestOrderG1(t *testing.T) {
g := new(G1).ScalarBaseMult(Order)
if !g.p.IsInfinity() {
t.Error("G1 has incorrect order")
}

one := new(G1).ScalarBaseMult(new(big.Int).SetInt64(1))
g.Add(g, one)
g.p.MakeAffine(nil)
if g.p.x.Cmp(one.p.x) != 0 || g.p.y.Cmp(one.p.y) != 0 {
t.Errorf("1+0 != 1 in G1")
}
}

func TestOrderG2(t *testing.T) {
g := new(G2).ScalarBaseMult(Order)
if !g.p.IsInfinity() {
t.Error("G2 has incorrect order")
}

one := new(G2).ScalarBaseMult(new(big.Int).SetInt64(1))
g.Add(g, one)
g.p.MakeAffine(nil)
if g.p.x.x.Cmp(one.p.x.x) != 0 ||
g.p.x.y.Cmp(one.p.x.y) != 0 ||
g.p.y.x.Cmp(one.p.y.x) != 0 ||
g.p.y.y.Cmp(one.p.y.y) != 0 {
t.Errorf("1+0 != 1 in G2")
}
}
```

```
func TestOrderGT(t *testing.T) {
gt := Pair(&G1{curveGen}, &G2{twistGen})
g := new(GT).ScalarMult(gt, Order)
if !g.p.IsOne() {
t.Error("GT has incorrect order")
}
}

func TestBilinearity(t *testing.T) {
for i := 0; i < 2; i++ {
a, p1, _ := RandomG1(rand.Reader)
b, p2, _ := RandomG2(rand.Reader)
e1 := Pair(p1, p2)

e2 := Pair(&G1{curveGen}, &G2{twistGen})
e2.ScalarMult(e2, a)
e2.ScalarMult(e2, b)

minusE2 := new(GT).Neg(e2)
e1.Add(e1, minusE2)

if !e1.p.IsOne() {
t.Fatalf("bad pairing result: %s", e1)
}
}
}

func TestG1Marshal(t *testing.T) {
g := new(G1).ScalarBaseMult(new(big.Int).SetInt64(1))
form := g.Marshal()
_, ok := new(G1).Unmarshal(form)
if !ok {
t.Fatalf("failed to unmarshal")
}

g.ScalarBaseMult(Order)
form = g.Marshal()
g2, ok := new(G1).Unmarshal(form)
if !ok {
t.Fatalf("failed to unmarshal ")
}
if !g2.p.IsInfinity() {
```

```
t.Fatalf(" unmarshaled incorrectly")
}
}

func TestG2Marshal(t *testing.T) {
g := new(G2).ScalarBaseMult(new(big.Int).SetInt64(1))
form := g.Marshal()
_, ok := new(G2).Unmarshal(form)
if !ok {
t.Fatalf("failed to unmarshal")
}

g.ScalarBaseMult(Order)
form = g.Marshal()
g2, ok := new(G2).Unmarshal(form)
if !ok {
t.Fatalf("failed to unmarshal ")
}
if !g2.p.IsInfinity() {
t.Fatalf(" unmarshaled incorrectly")
}
}

func TestG1Identity(t *testing.T) {
g := new(G1).ScalarBaseMult(new(big.Int).SetInt64(0))
if !g.p.IsInfinity() {
t.Error("failure")
}
}

func TestG2Identity(t *testing.T) {
g := new(G2).ScalarBaseMult(new(big.Int).SetInt64(0))
if !g.p.IsInfinity() {
t.Error("failure")
}
}

func TestTripartiteDiffieHellman(t *testing.T) {
a, _ := rand.Int(rand.Reader, Order)
b, _ := rand.Int(rand.Reader, Order)
c, _ := rand.Int(rand.Reader, Order)
```

```
pa, _ := new(G1).Unmarshal(new(G1).ScalarBaseMult(a).Marshal())
qa, _ := new(G2).Unmarshal(new(G2).ScalarBaseMult(a).Marshal())
pb, _ := new(G1).Unmarshal(new(G1).ScalarBaseMult(b).Marshal())
qb, _ := new(G2).Unmarshal(new(G2).ScalarBaseMult(b).Marshal())
pc, _ := new(G1).Unmarshal(new(G1).ScalarBaseMult(c).Marshal())
qc, _ := new(G2).Unmarshal(new(G2).ScalarBaseMult(c).Marshal())

k1 := Pair(pb, qc)
k1.ScalarMult(k1, a)
k1Bytes := k1.Marshal()

k2 := Pair(pc, qa)
k2.ScalarMult(k2, b)
k2Bytes := k2.Marshal()

k3 := Pair(pa, qb)
k3.ScalarMult(k3, c)
k3Bytes := k3.Marshal()

if !bytes.Equal(k1Bytes, k2Bytes) || !bytes.Equal(k2Bytes, k3Bytes) {
t.Errorf("keys didn't agree")
}
}

func BenchmarkPairing(b *testing.B) {
for i := 0; i < b.N; i++ {
Pair(&G1{curveGen}, &G2{twistGen})
}
}
```

60:F:\git\coin\ethereum\go-ethereum\crypto\bn256\constants.go

```
// u is the BN parameter that determines the prime: 1868033³.
var u = bigFromBase10("4965661367192848881")

// p is a prime over which we form a basic field: 36u+36u³+24u²+6u+1.
var P =
bigFromBase10("21888242871839275222246405745257275088696311157297823662689037894645226208583")

// Order is the number of elements in both G and G: 36u+36u³+18u²+6u+1.
var Order =
```

```
bigFromBase10("2188824287183927522224640574525727508854836440041603434369820418
6575808495617")

// xiToPMinus1Over6 is ^((p-1)/6) where  = i+9.
var xiToPMinus1Over6 =
&gfP2{bigFromBase10("164698233230778082238891372411765367990092866461081699356593016139617121983 16"),
bigFromBase10("837611886576382149658397386762636409258990606586829877690961791601876834008 0")}

// xiToPMinus1Over3 is ^((p-1)/3) where  = i+9.
var xiToPMinus1Over3 =
&gfP2{bigFromBase10("103076015958737097001522842738161122640692301306164367556251948548158757139 54"),
bigFromBase10("215754636382808430103983242694308260992690442743472168272126138678364350272 61")}

// xiToPMinus1Over2 is ^((p-1)/2) where  = i+9.
var xiToPMinus1Over2 =
&gfP2{bigFromBase10("350584376791155637868703030998424884554024350989925964101367809303313093040 3"),
bigFromBase10("282156518219453684454815956169350265935961718524412036707807955418648412655 4")}

// xiToPSquaredMinus1Over3 is ^((p²-1)/3) where  = i+9.
var xiToPSquaredMinus1Over3 =
bigFromBase10("2188824287183927522004244526010915316727770741447206164171475863576502055661 6")

// xiTo2PSquaredMinus2Over3 is ^((2p²-2)/3) where  = i+9 (a cubic root of unity, mod p).
var xiTo2PSquaredMinus2Over3 =
bigFromBase10("220396048514812192141860374282576202097427925888020565196 6")

// xiToPSquaredMinus1Over6 is ^((1p²-1)/6) where  = i+9 (a cubic root of -1, mod p).
var xiToPSquaredMinus1Over6 =
bigFromBase10("2188824287183927522004244526010915316727770741447206164171475863576502055661 7")

// xiTo2PMinus2Over3 is ^((2p-2)/3) where  = i+9.
var xiTo2PMinus2Over3 =
&gfP2{bigFromBase10("199377569717756479879959321699293419943146406529649494483133744724007166610 30"),
```

```
bigFromBase10("25819113444670093352673111154688030995516656050761967408678052585682343463 38")}

61:F:\git\coin\ethereum\go-ethereum\crypto\bn256\curve.go
x, y, z, t *big.Int
}

var curveB = new(big.Int).SetInt64(3)

// curveGen is the generator of G.
var curveGen = &curvePoint{
new(big.Int).SetInt64(1),
new(big.Int).SetInt64(-2),
new(big.Int).SetInt64(1),
new(big.Int).SetInt64(1),
}

func newCurvePoint(pool *bnPool) *curvePoint {
return &curvePoint{
pool.Get(),
pool.Get(),
pool.Get(),
pool.Get(),
}
}

func (c *curvePoint) String() string {
c.MakeAffine(new(bnPool))
return "(" + c.x.String() + ", " + c.y.String() + ")"
}

func (c *curvePoint) Put(pool *bnPool) {
pool.Put(c.x)
pool.Put(c.y)
pool.Put(c.z)
pool.Put(c.t)
}

func (c *curvePoint) Set(a *curvePoint) {
c.x.Set(a.x)
c.y.Set(a.y)
c.z.Set(a.z)
```

```
c.t.Set(a.t)
}

// IsOnCurve returns true iff c is on the curve where c must be in affine form.
func (c *curvePoint) IsOnCurve() bool {
yy := new(big.Int).Mul(c.y, c.y)
xxx := new(big.Int).Mul(c.x, c.x)
xxx.Mul(xxx, c.x)
yy.Sub(yy, xxx)
yy.Sub(yy, curveB)
if yy.Sign() < 0 || yy.Cmp(P) >= 0 {
yy.Mod(yy, P)
}
return yy.Sign() == 0
}

func (c *curvePoint) SetInfinity() {
c.z.SetInt64(0)
}

func (c *curvePoint) IsInfinity() bool {
return c.z.Sign() == 0
}

func (c *curvePoint) Add(a, b *curvePoint, pool *bnPool) {
if a.IsInfinity() {
c.Set(b)
return
}
if b.IsInfinity() {
c.Set(a)
return
}

// See http://hyperelliptic.org/EFD/g1p/auto-code/shortw/jacobian-0/addition/add-2007-bl.op3

// Normalize the points by replacing a = [x1:y1:z1] and b = [x2:y2:z2]
// by [u1:s1:z1·z2] and [u2:s2:z1·z2]
// where u1 = x1·z2², s1 = y1·z2³ and u1 = x2·z1², s2 = y2·z1³
z1z1 := pool.Get().Mul(a.z, a.z)
z1z1.Mod(z1z1, P)
z2z2 := pool.Get().Mul(b.z, b.z)
```

```
z2z2.Mod(z2z2, P)
u1 := pool.Get().Mul(a.x, z2z2)
u1.Mod(u1, P)
u2 := pool.Get().Mul(b.x, z1z1)
u2.Mod(u2, P)

t := pool.Get().Mul(b.z, z2z2)
t.Mod(t, P)
s1 := pool.Get().Mul(a.y, t)
s1.Mod(s1, P)

t.Mul(a.z, z1z1)
t.Mod(t, P)
s2 := pool.Get().Mul(b.y, t)
s2.Mod(s2, P)

// Compute x = (2h)²(s²-u1-u2)
// where s = (s2-s1)/(u2-u1) is the slope of the line through
// (u1,s1) and (u2,s2). The extra factor 2h = 2(u2-u1) comes from the value of z below.
// This is also:
// 4(s2-s1)² - 4h²(u1+u2) = 4(s2-s1)² - 4h³ - 4h²(2u1)
//                        = r² - j - 2v
// with the notations below.
h := pool.Get().Sub(u2, u1)
xEqual := h.Sign() == 0

t.Add(h, h)
// i = 4h²
i := pool.Get().Mul(t, t)
i.Mod(i, P)
// j = 4h³
j := pool.Get().Mul(h, i)
j.Mod(j, P)

t.Sub(s2, s1)
yEqual := t.Sign() == 0
if xEqual && yEqual {
c.Double(a, pool)
return
}
r := pool.Get().Add(t, t)
```

```
v := pool.Get().Mul(u1, i)
v.Mod(v, P)

// t4 = 4(s2-s1)²
t4 := pool.Get().Mul(r, r)
t4.Mod(t4, P)
t.Add(v, v)
t6 := pool.Get().Sub(t4, j)
c.x.Sub(t6, t)

// Set y = -(2h)³(s1 + s*(x/4h²-u1))
// This is also
// y = - 2·s1·j - (s2-s1)(2x - 2i·u1) = r(v-x) - 2·s1·j
t.Sub(v, c.x) // t7
t4.Mul(s1, j) // t8
t4.Mod(t4, P)
t6.Add(t4, t4) // t9
t4.Mul(r, t)   // t10
t4.Mod(t4, P)
c.y.Sub(t4, t6)

// Set z = 2(u2-u1)·z1·z2 = 2h·z1·z2
t.Add(a.z, b.z) // t11
t4.Mul(t, t)    // t12
t4.Mod(t4, P)
t.Sub(t4, z1z1) // t13
t4.Sub(t, z2z2) // t14
c.z.Mul(t4, h)
c.z.Mod(c.z, P)

pool.Put(z1z1)
pool.Put(z2z2)
pool.Put(u1)
pool.Put(u2)
pool.Put(t)
pool.Put(s1)
pool.Put(s2)
pool.Put(h)
pool.Put(i)
pool.Put(j)
pool.Put(r)
pool.Put(v)
```

```
pool.Put(t4)
pool.Put(t6)
}

func (c *curvePoint) Double(a *curvePoint, pool *bnPool) {
// See http://hyperelliptic.org/EFD/g1p/auto-code/shortw/jacobian-0/doubling/dbl-2009-l.op3
A := pool.Get().Mul(a.x, a.x)
A.Mod(A, P)
B := pool.Get().Mul(a.y, a.y)
B.Mod(B, P)
C_ := pool.Get().Mul(B, B)
C_.Mod(C_, P)

t := pool.Get().Add(a.x, B)
t2 := pool.Get().Mul(t, t)
t2.Mod(t2, P)
t.Sub(t2, A)
t2.Sub(t, C_)
d := pool.Get().Add(t2, t2)
t.Add(A, A)
e := pool.Get().Add(t, A)
f := pool.Get().Mul(e, e)
f.Mod(f, P)

t.Add(d, d)
c.x.Sub(f, t)

t.Add(C_, C_)
t2.Add(t, t)
t.Add(t2, t2)
c.y.Sub(d, c.x)
t2.Mul(e, c.y)
t2.Mod(t2, P)
c.y.Sub(t2, t)

t.Mul(a.y, a.z)
t.Mod(t, P)
c.z.Add(t, t)

pool.Put(A)
pool.Put(B)
pool.Put(C_)
```

```
pool.Put(t)
pool.Put(t2)
pool.Put(d)
pool.Put(e)
pool.Put(f)
}

func (c *curvePoint) Mul(a *curvePoint, scalar *big.Int, pool *bnPool) *curvePoint {
sum := newCurvePoint(pool)
sum.SetInfinity()
t := newCurvePoint(pool)

for i := scalar.BitLen(); i >= 0; i-- {
t.Double(sum, pool)
if scalar.Bit(i) != 0 {
sum.Add(t, a, pool)
} else {
sum.Set(t)
}
}

c.Set(sum)
sum.Put(pool)
t.Put(pool)
return c
}

func (c *curvePoint) MakeAffine(pool *bnPool) *curvePoint {
if words := c.z.Bits(); len(words) == 1 && words[0] == 1 {
return c
}

zInv := pool.Get().ModInverse(c.z, P)
t := pool.Get().Mul(c.y, zInv)
t.Mod(t, P)
zInv2 := pool.Get().Mul(zInv, zInv)
zInv2.Mod(zInv2, P)
c.y.Mul(t, zInv2)
c.y.Mod(c.y, P)
t.Mul(c.x, zInv2)
t.Mod(t, P)
c.x.Set(t)
```

```
c.z.SetInt64(1)
c.t.SetInt64(1)

pool.Put(zInv)
pool.Put(t)
pool.Put(zInv2)

return c
}

func (c *curvePoint) Negative(a *curvePoint) {
c.x.Set(a.x)
c.y.Neg(a.y)
c.z.Set(a.z)
c.t.SetInt64(0)
}
```

62:F:\git\coin\ethereum\go-ethereum\crypto\bn256\example_test.go

```
// Each of three parties, a, b and c, generate a private value.
a, _ := rand.Int(rand.Reader, Order)
b, _ := rand.Int(rand.Reader, Order)
c, _ := rand.Int(rand.Reader, Order)

// Then each party calculates g and g times their private value.
pa := new(G1).ScalarBaseMult(a)
qa := new(G2).ScalarBaseMult(a)

pb := new(G1).ScalarBaseMult(b)
qb := new(G2).ScalarBaseMult(b)

pc := new(G1).ScalarBaseMult(c)
qc := new(G2).ScalarBaseMult(c)

// Now each party exchanges its public values with the other two and
// all parties can calculate the shared key.
k1 := Pair(pb, qc)
k1.ScalarMult(k1, a)

k2 := Pair(pc, qa)
k2.ScalarMult(k2, b)
```

```
k3 := Pair(pa, qb)
k3.ScalarMult(k3, c)

// k1, k2 and k3 will all be equal.
}

63:F:\git\coin\ethereum\go-ethereum\crypto\bn256\gfp12.go
// gfP12 implements the field of size p¹² as a quadratic extension of gfP6
// where ²=.
type gfP12 struct {
x, y *gfP6 // value is x + y
}

func newGFp12(pool *bnPool) *gfP12 {
return &gfP12{newGFp6(pool), newGFp6(pool)}
}

func (e *gfP12) String() string {
return "(" + e.x.String() + "," + e.y.String() + ")"
}

func (e *gfP12) Put(pool *bnPool) {
e.x.Put(pool)
e.y.Put(pool)
}

func (e *gfP12) Set(a *gfP12) *gfP12 {
e.x.Set(a.x)
e.y.Set(a.y)
return e
}

func (e *gfP12) SetZero() *gfP12 {
e.x.SetZero()
e.y.SetZero()
return e
}

func (e *gfP12) SetOne() *gfP12 {
e.x.SetZero()
e.y.SetOne()
return e
```

```
}

func (e *gfP12) Minimal() {
e.x.Minimal()
e.y.Minimal()
}

func (e *gfP12) IsZero() bool {
e.Minimal()
return e.x.IsZero() && e.y.IsZero()
}

func (e *gfP12) IsOne() bool {
e.Minimal()
return e.x.IsZero() && e.y.IsOne()
}

func (e *gfP12) Conjugate(a *gfP12) *gfP12 {
e.x.Negative(a.x)
e.y.Set(a.y)
return a
}

func (e *gfP12) Negative(a *gfP12) *gfP12 {
e.x.Negative(a.x)
e.y.Negative(a.y)
return e
}

// Frobenius computes (x+y)^p = x^p ·^((p-1)/6) + y^p
func (e *gfP12) Frobenius(a *gfP12, pool *bnPool) *gfP12 {
e.x.Frobenius(a.x, pool)
e.y.Frobenius(a.y, pool)
e.x.MulScalar(e.x, xiToPMinus1Over6, pool)
return e
}

// FrobeniusP2 computes (x+y)^p² = x^p² ·^((p²-1)/6) + y^p²
func (e *gfP12) FrobeniusP2(a *gfP12, pool *bnPool) *gfP12 {
e.x.FrobeniusP2(a.x)
e.x.MulGFP(e.x, xiToPSquaredMinus1Over6)
e.y.FrobeniusP2(a.y)
```

```
return e
}

func (e *gfP12) Add(a, b *gfP12) *gfP12 {
e.x.Add(a.x, b.x)
e.y.Add(a.y, b.y)
return e
}

func (e *gfP12) Sub(a, b *gfP12) *gfP12 {
e.x.Sub(a.x, b.x)
e.y.Sub(a.y, b.y)
return e
}

func (e *gfP12) Mul(a, b *gfP12, pool *bnPool) *gfP12 {
tx := newGFp6(pool)
tx.Mul(a.x, b.y, pool)
t := newGFp6(pool)
t.Mul(b.x, a.y, pool)
tx.Add(tx, t)

ty := newGFp6(pool)
ty.Mul(a.y, b.y, pool)
t.Mul(a.x, b.x, pool)
t.MulTau(t, pool)
e.y.Add(ty, t)
e.x.Set(tx)

tx.Put(pool)
ty.Put(pool)
t.Put(pool)
return e
}

func (e *gfP12) MulScalar(a *gfP12, b *gfP6, pool *bnPool) *gfP12 {
e.x.Mul(e.x, b, pool)
e.y.Mul(e.y, b, pool)
return e
}

func (c *gfP12) Exp(a *gfP12, power *big.Int, pool *bnPool) *gfP12 {
```

```
sum := newGFp12(pool)
sum.SetOne()
t := newGFp12(pool)

for i := power.BitLen() - 1; i >= 0; i-- {
t.Square(sum, pool)
if power.Bit(i) != 0 {
sum.Mul(t, a, pool)
} else {
sum.Set(t)
}
}

c.Set(sum)

sum.Put(pool)
t.Put(pool)

return c
}

func (e *gfP12) Square(a *gfP12, pool *bnPool) *gfP12 {
// Complex squaring algorithm
v0 := newGFp6(pool)
v0.Mul(a.x, a.y, pool)

t := newGFp6(pool)
t.MulTau(a.x, pool)
t.Add(a.y, t)
ty := newGFp6(pool)
ty.Add(a.x, a.y)
ty.Mul(ty, t, pool)
ty.Sub(ty, v0)
t.MulTau(v0, pool)
ty.Sub(ty, t)

e.y.Set(ty)
e.x.Double(v0)

v0.Put(pool)
t.Put(pool)
ty.Put(pool)
```

```
return e
}

func (e *gfP12) Invert(a *gfP12, pool *bnPool) *gfP12 {
// See "Implementing cryptographic pairings", M. Scott, section 3.2.
// ftp://136.206.11.249/pub/crypto/pairings.pdf
t1 := newGFp6(pool)
t2 := newGFp6(pool)

t1.Square(a.x, pool)
t2.Square(a.y, pool)
t1.MulTau(t1, pool)
t1.Sub(t2, t1)
t2.Invert(t1, pool)

e.x.Negative(a.x)
e.y.Set(a.y)
e.MulScalar(e, t2, pool)

t1.Put(pool)
t2.Put(pool)

return e
}

64:F:\git\coin\ethereum\go-ethereum\crypto\bn256\gfp2.go
// gfP2 implements a field of size p² as a quadratic extension of the base
// field where i²=-1.
type gfP2 struct {
x, y *big.Int // value is xi+y.
}

func newGFp2(pool *bnPool) *gfP2 {
return &gfP2{pool.Get(), pool.Get()}
}

func (e *gfP2) String() string {
x := new(big.Int).Mod(e.x, P)
y := new(big.Int).Mod(e.y, P)
return "(" + x.String() + "," + y.String() + ")"
}
```

```
func (e *gfP2) Put(pool *bnPool) {
pool.Put(e.x)
pool.Put(e.y)
}

func (e *gfP2) Set(a *gfP2) *gfP2 {
e.x.Set(a.x)
e.y.Set(a.y)
return e
}

func (e *gfP2) SetZero() *gfP2 {
e.x.SetInt64(0)
e.y.SetInt64(0)
return e
}

func (e *gfP2) SetOne() *gfP2 {
e.x.SetInt64(0)
e.y.SetInt64(1)
return e
}

func (e *gfP2) Minimal() {
if e.x.Sign() < 0 || e.x.Cmp(P) >= 0 {
e.x.Mod(e.x, P)
}
if e.y.Sign() < 0 || e.y.Cmp(P) >= 0 {
e.y.Mod(e.y, P)
}
}

func (e *gfP2) IsZero() bool {
return e.x.Sign() == 0 && e.y.Sign() == 0
}

func (e *gfP2) IsOne() bool {
if e.x.Sign() != 0 {
return false
}
words := e.y.Bits()
```

```
return len(words) == 1 && words[0] == 1
}

func (e *gfP2) Conjugate(a *gfP2) *gfP2 {
e.y.Set(a.y)
e.x.Neg(a.x)
return e
}

func (e *gfP2) Negative(a *gfP2) *gfP2 {
e.x.Neg(a.x)
e.y.Neg(a.y)
return e
}

func (e *gfP2) Add(a, b *gfP2) *gfP2 {
e.x.Add(a.x, b.x)
e.y.Add(a.y, b.y)
return e
}

func (e *gfP2) Sub(a, b *gfP2) *gfP2 {
e.x.Sub(a.x, b.x)
e.y.Sub(a.y, b.y)
return e
}

func (e *gfP2) Double(a *gfP2) *gfP2 {
e.x.Lsh(a.x, 1)
e.y.Lsh(a.y, 1)
return e
}

func (c *gfP2) Exp(a *gfP2, power *big.Int, pool *bnPool) *gfP2 {
sum := newGFp2(pool)
sum.SetOne()
t := newGFp2(pool)

for i := power.BitLen() - 1; i >= 0; i-- {
t.Square(sum, pool)
if power.Bit(i) != 0 {
sum.Mul(t, a, pool)
```

```
} else {
sum.Set(t)
}
}

c.Set(sum)

sum.Put(pool)
t.Put(pool)

return c
}

// See "Multiplication and Squaring in Pairing-Friendly Fields",
// http://eprint.iacr.org/2006/471.pdf
func (e *gfP2) Mul(a, b *gfP2, pool *bnPool) *gfP2 {
tx := pool.Get().Mul(a.x, b.y)
t := pool.Get().Mul(b.x, a.y)
tx.Add(tx, t)
tx.Mod(tx, P)

ty := pool.Get().Mul(a.y, b.y)
t.Mul(a.x, b.x)
ty.Sub(ty, t)
e.y.Mod(ty, P)
e.x.Set(tx)

pool.Put(tx)
pool.Put(ty)
pool.Put(t)

return e
}

func (e *gfP2) MulScalar(a *gfP2, b *big.Int) *gfP2 {
e.x.Mul(a.x, b)
e.y.Mul(a.y, b)
return e
}

// MulXi sets e=a where =i+9 and then returns e.
func (e *gfP2) MulXi(a *gfP2, pool *bnPool) *gfP2 {
```

```
// (xi+y)(i+3) = (9x+y)i+(9y-x)
tx := pool.Get().Lsh(a.x, 3)
tx.Add(tx, a.x)
tx.Add(tx, a.y)

ty := pool.Get().Lsh(a.y, 3)
ty.Add(ty, a.y)
ty.Sub(ty, a.x)

e.x.Set(tx)
e.y.Set(ty)

pool.Put(tx)
pool.Put(ty)

return e
}

func (e *gfP2) Square(a *gfP2, pool *bnPool) *gfP2 {
// Complex squaring algorithm:
// (xi+b)² = (x+y)(y-x) + 2*i*x*y
t1 := pool.Get().Sub(a.y, a.x)
t2 := pool.Get().Add(a.x, a.y)
ty := pool.Get().Mul(t1, t2)
ty.Mod(ty, P)

t1.Mul(a.x, a.y)
t1.Lsh(t1, 1)

e.x.Mod(t1, P)
e.y.Set(ty)

pool.Put(t1)
pool.Put(t2)
pool.Put(ty)

return e
}

func (e *gfP2) Invert(a *gfP2, pool *bnPool) *gfP2 {
// See "Implementing cryptographic pairings", M. Scott, section 3.2.
// ftp://136.206.11.249/pub/crypto/pairings.pdf
```

```
t := pool.Get()
t.Mul(a.y, a.y)
t2 := pool.Get()
t2.Mul(a.x, a.x)
t.Add(t, t2)

inv := pool.Get()
inv.ModInverse(t, P)

e.x.Neg(a.x)
e.x.Mul(e.x, inv)
e.x.Mod(e.x, P)

e.y.Mul(a.y, inv)
e.y.Mod(e.y, P)

pool.Put(t)
pool.Put(t2)
pool.Put(inv)

return e
}

func (e *gfP2) Real() *big.Int {
return e.x
}

func (e *gfP2) Imag() *big.Int {
return e.y
}

65:F:\git\coin\ethereum\go-ethereum\crypto\bn256\gfp6.go
// gfP6 implements the field of size p as a cubic extension of gfP2 where ³=
// and =i+9.
type gfP6 struct {
x, y, z *gfP2 // value is x² + y + z
}

func newGFp6(pool *bnPool) *gfP6 {
return &gfP6{newGFp2(pool), newGFp2(pool), newGFp2(pool)}
}
```

```
func (e *gfP6) String() string {
return "(" + e.x.String() + "," + e.y.String() + "," + e.z.String() + ")"
}

func (e *gfP6) Put(pool *bnPool) {
e.x.Put(pool)
e.y.Put(pool)
e.z.Put(pool)
}

func (e *gfP6) Set(a *gfP6) *gfP6 {
e.x.Set(a.x)
e.y.Set(a.y)
e.z.Set(a.z)
return e
}

func (e *gfP6) SetZero() *gfP6 {
e.x.SetZero()
e.y.SetZero()
e.z.SetZero()
return e
}

func (e *gfP6) SetOne() *gfP6 {
e.x.SetZero()
e.y.SetZero()
e.z.SetOne()
return e
}

func (e *gfP6) Minimal() {
e.x.Minimal()
e.y.Minimal()
e.z.Minimal()
}

func (e *gfP6) IsZero() bool {
return e.x.IsZero() && e.y.IsZero() && e.z.IsZero()
}

func (e *gfP6) IsOne() bool {
```

```
return e.x.IsZero() && e.y.IsZero() && e.z.IsOne()
}

func (e *gfP6) Negative(a *gfP6) *gfP6 {
e.x.Negative(a.x)
e.y.Negative(a.y)
e.z.Negative(a.z)
return e
}

func (e *gfP6) Frobenius(a *gfP6, pool *bnPool) *gfP6 {
e.x.Conjugate(a.x)
e.y.Conjugate(a.y)
e.z.Conjugate(a.z)

e.x.Mul(e.x, xiTo2PMinus2Over3, pool)
e.y.Mul(e.y, xiToPMinus1Over3, pool)
return e
}

// FrobeniusP2 computes (x²+y+z)^(p²) = x^(2p²) + y^(p²) + z
func (e *gfP6) FrobeniusP2(a *gfP6) *gfP6 {
// ^(2p²) = ²^(2p²-2) = ²^((2p²-2)/3)
e.x.MulScalar(a.x, xiTo2PSquaredMinus2Over3)
// ^(p²) = ^(p²-1) = ^((p²-1)/3)
e.y.MulScalar(a.y, xiToPSquaredMinus1Over3)
e.z.Set(a.z)
return e
}

func (e *gfP6) Add(a, b *gfP6) *gfP6 {
e.x.Add(a.x, b.x)
e.y.Add(a.y, b.y)
e.z.Add(a.z, b.z)
return e
}

func (e *gfP6) Sub(a, b *gfP6) *gfP6 {
e.x.Sub(a.x, b.x)
e.y.Sub(a.y, b.y)
e.z.Sub(a.z, b.z)
return e
```

```
}

func (e *gfP6) Double(a *gfP6) *gfP6 {
e.x.Double(a.x)
e.y.Double(a.y)
e.z.Double(a.z)
return e
}

func (e *gfP6) Mul(a, b *gfP6, pool *bnPool) *gfP6 {
// "Multiplication and Squaring on Pairing-Friendly Fields"
// Section 4, Karatsuba method.
// http://eprint.iacr.org/2006/471.pdf

v0 := newGFp2(pool)
v0.Mul(a.z, b.z, pool)
v1 := newGFp2(pool)
v1.Mul(a.y, b.y, pool)
v2 := newGFp2(pool)
v2.Mul(a.x, b.x, pool)

t0 := newGFp2(pool)
t0.Add(a.x, a.y)
t1 := newGFp2(pool)
t1.Add(b.x, b.y)
tz := newGFp2(pool)
tz.Mul(t0, t1, pool)

tz.Sub(tz, v1)
tz.Sub(tz, v2)
tz.MulXi(tz, pool)
tz.Add(tz, v0)

t0.Add(a.y, a.z)
t1.Add(b.y, b.z)
ty := newGFp2(pool)
ty.Mul(t0, t1, pool)
ty.Sub(ty, v0)
ty.Sub(ty, v1)
t0.MulXi(v2, pool)
ty.Add(ty, t0)
```

```
t0.Add(a.x, a.z)
t1.Add(b.x, b.z)
tx := newGFp2(pool)
tx.Mul(t0, t1, pool)
tx.Sub(tx, v0)
tx.Add(tx, v1)
tx.Sub(tx, v2)

e.x.Set(tx)
e.y.Set(ty)
e.z.Set(tz)

t0.Put(pool)
t1.Put(pool)
tx.Put(pool)
ty.Put(pool)
tz.Put(pool)
v0.Put(pool)
v1.Put(pool)
v2.Put(pool)
return e
}

func (e *gfP6) MulScalar(a *gfP6, b *gfP2, pool *bnPool) *gfP6 {
e.x.Mul(a.x, b, pool)
e.y.Mul(a.y, b, pool)
e.z.Mul(a.z, b, pool)
return e
}

func (e *gfP6) MulGFP(a *gfP6, b *big.Int) *gfP6 {
e.x.MulScalar(a.x, b)
e.y.MulScalar(a.y, b)
e.z.MulScalar(a.z, b)
return e
}

// MulTau computes ·(a²+b+c) = b²+c+a
func (e *gfP6) MulTau(a *gfP6, pool *bnPool) {
tz := newGFp2(pool)
tz.MulXi(a.x, pool)
ty := newGFp2(pool)
```

```
ty.Set(a.y)
e.y.Set(a.z)
e.x.Set(ty)
e.z.Set(tz)
tz.Put(pool)
ty.Put(pool)
}

func (e *gfP6) Square(a *gfP6, pool *bnPool) *gfP6 {
v0 := newGFp2(pool).Square(a.z, pool)
v1 := newGFp2(pool).Square(a.y, pool)
v2 := newGFp2(pool).Square(a.x, pool)

c0 := newGFp2(pool).Add(a.x, a.y)
c0.Square(c0, pool)
c0.Sub(c0, v1)
c0.Sub(c0, v2)
c0.MulXi(c0, pool)
c0.Add(c0, v0)

c1 := newGFp2(pool).Add(a.y, a.z)
c1.Square(c1, pool)
c1.Sub(c1, v0)
c1.Sub(c1, v1)
xiV2 := newGFp2(pool).MulXi(v2, pool)
c1.Add(c1, xiV2)

c2 := newGFp2(pool).Add(a.x, a.z)
c2.Square(c2, pool)
c2.Sub(c2, v0)
c2.Add(c2, v1)
c2.Sub(c2, v2)

e.x.Set(c2)
e.y.Set(c1)
e.z.Set(c0)

v0.Put(pool)
v1.Put(pool)
v2.Put(pool)
c0.Put(pool)
c1.Put(pool)
```

```
c2.Put(pool)
xiV2.Put(pool)

return e
}

func (e *gfP6) Invert(a *gfP6, pool *bnPool) *gfP6 {
// See "Implementing cryptographic pairings", M. Scott, section 3.2.
// ftp://136.206.11.249/pub/crypto/pairings.pdf

// Here we can give a short explanation of how it works: let j be a cubic root of
// unity in GF(p²) so that 1+j+j²=0.
// Then (x² + y + z)(xj²² + yj + z)(xj² + yj² + z)
// = (x² + y + z)(C²+B+A)
// = (x³²+y³+z³-3xyz) = F is an element of the base field (the norm).
//
// On the other hand (xj²² + yj + z)(xj² + yj² + z)
// = ²(y²-xz) + (x²-yz) + (z²-xy)
//
// So that's why A = (z²-xy), B = (x²-yz), C = (y²-xz)
t1 := newGFp2(pool)

A := newGFp2(pool)
A.Square(a.z, pool)
t1.Mul(a.x, a.y, pool)
t1.MulXi(t1, pool)
A.Sub(A, t1)

B := newGFp2(pool)
B.Square(a.x, pool)
B.MulXi(B, pool)
t1.Mul(a.y, a.z, pool)
B.Sub(B, t1)

C_ := newGFp2(pool)
C_.Square(a.y, pool)
t1.Mul(a.x, a.z, pool)
C_.Sub(C_, t1)

F := newGFp2(pool)
F.Mul(C_, a.y, pool)
F.MulXi(F, pool)
```

```
t1.Mul(A, a.z, pool)
F.Add(F, t1)
t1.Mul(B, a.x, pool)
t1.MulXi(t1, pool)
F.Add(F, t1)

F.Invert(F, pool)

e.x.Mul(C_, F, pool)
e.y.Mul(B, F, pool)
e.z.Mul(A, F, pool)

t1.Put(pool)
A.Put(pool)
B.Put(pool)
C_.Put(pool)
F.Put(pool)

return e
}

66:F:\git\coin\ethereum\go-ethereum\crypto\bn256\main_test.go
}
t.Logf("%d: %x\n", n, g2.Marshal())
}
}

func TestPairings(t *testing.T) {
a1 := new(G1).ScalarBaseMult(bigFromBase10("1"))
a2 := new(G1).ScalarBaseMult(bigFromBase10("2"))
a37 := new(G1).ScalarBaseMult(bigFromBase10("37"))
an1 :=
new(G1).ScalarBaseMult(bigFromBase10("21888242871839275222246405745257275088548364
400416034343698204186575808495616"))

b0 := new(G2).ScalarBaseMult(bigFromBase10("0"))
b1 := new(G2).ScalarBaseMult(bigFromBase10("1"))
b2 := new(G2).ScalarBaseMult(bigFromBase10("2"))
b27 := new(G2).ScalarBaseMult(bigFromBase10("27"))
b999 := new(G2).ScalarBaseMult(bigFromBase10("999"))
bn1 :=
new(G2).ScalarBaseMult(bigFromBase10("21888242871839275222246405745257275088548364
```

```
4400416034343698204186575808495616"))

p1 := Pair(a1, b1)
pn1 := Pair(a1, bn1)
np1 := Pair(an1, b1)
if pn1.String() != np1.String() {
t.Error("Pairing mismatch: e(a, -b) != e(-a, b)")
}
if !PairingCheck([]*G1{a1, an1}, []*G2{b1, b1}) {
t.Error("MultiAte check gave false negative!")
}
p0 := new(GT).Add(p1, pn1)
p0_2 := Pair(a1, b0)
if p0.String() != p0_2.String() {
t.Error("Pairing mismatch: e(a, b) * e(a, -b) != 1")
}
p0_3 := new(GT).ScalarMult(p1,
bigFromBase10("218882428718392752222464057452572750885483644004160343436982041
6575808495617"))
if p0.String() != p0_3.String() {
t.Error("Pairing mismatch: e(a, b) has wrong order")
}
p2 := Pair(a2, b1)
p2_2 := Pair(a1, b2)
p2_3 := new(GT).ScalarMult(p1, bigFromBase10("2"))
if p2.String() != p2_2.String() {
t.Error("Pairing mismatch: e(a, b * 2) != e(a * 2, b)")
}
if p2.String() != p2_3.String() {
t.Error("Pairing mismatch: e(a, b * 2) != e(a, b) ** 2")
}
if p2.String() == p1.String() {
t.Error("Pairing is degenerate!")
}
if PairingCheck([]*G1{a1, a1}, []*G2{b1, b1}) {
t.Error("MultiAte check gave false positive!")
}
p999 := Pair(a37, b27)
p999_2 := Pair(a1, b999)
if p999.String() != p999_2.String() {
t.Error("Pairing mismatch: e(a * 37, b * 27) != e(a, b * 999)")
}
```

```
}

67:F:\git\coin\ethereum\go-ethereum\crypto\bn256\optate.go
D.Sub(D, r2)
D.Sub(D, r.t)
D.Mul(D, r.t, pool)

H := newGFp2(pool).Sub(B, r.x)
I := newGFp2(pool).Square(H, pool)

E := newGFp2(pool).Add(I, I)
E.Add(E, E)

J := newGFp2(pool).Mul(H, E, pool)

L1 := newGFp2(pool).Sub(D, r.y)
L1.Sub(L1, r.y)

V := newGFp2(pool).Mul(r.x, E, pool)

rOut = newTwistPoint(pool)
rOut.x.Square(L1, pool)
rOut.x.Sub(rOut.x, J)
rOut.x.Sub(rOut.x, V)
rOut.x.Sub(rOut.x, V)

rOut.z.Add(r.z, H)
rOut.z.Square(rOut.z, pool)
rOut.z.Sub(rOut.z, r.t)
rOut.z.Sub(rOut.z, I)

t := newGFp2(pool).Sub(V, rOut.x)
t.Mul(t, L1, pool)
t2 := newGFp2(pool).Mul(r.y, J, pool)
t2.Add(t2, t2)
rOut.y.Sub(t, t2)

rOut.t.Square(rOut.z, pool)

t.Add(p.y, rOut.z)
t.Square(t, pool)
t.Sub(t, r2)
```

```
t.Sub(t, rOut.t)

t2.Mul(L1, p.x, pool)
t2.Add(t2, t2)
a = newGFp2(pool)
a.Sub(t2, t)

c = newGFp2(pool)
c.MulScalar(rOut.z, q.y)
c.Add(c, c)

b = newGFp2(pool)
b.SetZero()
b.Sub(b, L1)
b.MulScalar(b, q.x)
b.Add(b, b)

B.Put(pool)
D.Put(pool)
H.Put(pool)
I.Put(pool)
E.Put(pool)
J.Put(pool)
L1.Put(pool)
V.Put(pool)
t.Put(pool)
t2.Put(pool)

return
}

func lineFunctionDouble(r *twistPoint, q *curvePoint, pool *bnPool) (a, b, c *gfP2, rOut *twistPoint)
{
// See the doubling algorithm for a=0 from "Faster Computation of the
// Tate Pairing", http://arxiv.org/pdf/0904.0854v3.pdf

A := newGFp2(pool).Square(r.x, pool)
B := newGFp2(pool).Square(r.y, pool)
C_ := newGFp2(pool).Square(B, pool)

D := newGFp2(pool).Add(r.x, B)
D.Square(D, pool)
```

```
D.Sub(D, A)
D.Sub(D, C_)
D.Add(D, D)

E := newGFp2(pool).Add(A, A)
E.Add(E, A)

G := newGFp2(pool).Square(E, pool)

rOut = newTwistPoint(pool)
rOut.x.Sub(G, D)
rOut.x.Sub(rOut.x, D)

rOut.z.Add(r.y, r.z)
rOut.z.Square(rOut.z, pool)
rOut.z.Sub(rOut.z, B)
rOut.z.Sub(rOut.z, r.t)

rOut.y.Sub(D, rOut.x)
rOut.y.Mul(rOut.y, E, pool)
t := newGFp2(pool).Add(C_, C_)
t.Add(t, t)
t.Add(t, t)
rOut.y.Sub(rOut.y, t)

rOut.t.Square(rOut.z, pool)

t.Mul(E, r.t, pool)
t.Add(t, t)
b = newGFp2(pool)
b.SetZero()
b.Sub(b, t)
b.MulScalar(b, q.x)

a = newGFp2(pool)
a.Add(r.x, E)
a.Square(a, pool)
a.Sub(a, A)
a.Sub(a, G)
t.Add(B, B)
t.Add(t, t)
a.Sub(a, t)
```

```
c = newGFp2(pool)
c.Mul(rOut.z, r.t, pool)
c.Add(c, c)
c.MulScalar(c, q.y)

A.Put(pool)
B.Put(pool)
C_.Put(pool)
D.Put(pool)
E.Put(pool)
G.Put(pool)
t.Put(pool)

return
}

func mulLine(ret *gfP12, a, b, c *gfP2, pool *bnPool) {
a2 := newGFp6(pool)
a2.x.SetZero()
a2.y.Set(a)
a2.z.Set(b)
a2.Mul(a2, ret.x, pool)
t3 := newGFp6(pool).MulScalar(ret.y, c, pool)

t := newGFp2(pool)
t.Add(b, c)
t2 := newGFp6(pool)
t2.x.SetZero()
t2.y.Set(a)
t2.z.Set(t)
ret.x.Add(ret.x, ret.y)

ret.y.Set(t3)

ret.x.Mul(ret.x, t2, pool)
ret.x.Sub(ret.x, a2)
ret.x.Sub(ret.x, ret.y)
a2.MulTau(a2, pool)
ret.y.Add(ret.y, a2)

a2.Put(pool)
```

```
t3.Put(pool)
t2.Put(pool)
t.Put(pool)
}

// sixuPlus2NAF is 6u+2 in non-adjacent form.
var sixuPlus2NAF = []int8{0, 0, 0, 1, 0, 1, 0, -1, 0, 0, 1, -1, 0, 0, 1, 0,
0, 1, 1, 0, -1, 0, 0, 1, 0, -1, 0, 0, 0, 0, 1, 1,
1, 0, 0, -1, 0, 0, 1, 0, 0, 0, 0, 0, -1, 0, 0, 1,
1, 0, 0, -1, 0, 0, 0, 1, 1, 0, -1, 0, 0, 1, 0, 1, 1}

// miller implements the Miller loop for calculating the Optimal Ate pairing.
// See algorithm 1 from http://cryptojedi.org/papers/dclxvi-20100714.pdf
func miller(q *twistPoint, p *curvePoint, pool *bnPool) *gfP12 {
ret := newGFp12(pool)
ret.SetOne()

aAffine := newTwistPoint(pool)
aAffine.Set(q)
aAffine.MakeAffine(pool)

bAffine := newCurvePoint(pool)
bAffine.Set(p)
bAffine.MakeAffine(pool)

minusA := newTwistPoint(pool)
minusA.Negative(aAffine, pool)

r := newTwistPoint(pool)
r.Set(aAffine)

r2 := newGFp2(pool)
r2.Square(aAffine.y, pool)

for i := len(sixuPlus2NAF) - 1; i > 0; i-- {
a, b, c, newR := lineFunctionDouble(r, bAffine, pool)
if i != len(sixuPlus2NAF)-1 {
ret.Square(ret, pool)
}

mulLine(ret, a, b, c, pool)
a.Put(pool)
```

```
b.Put(pool)
c.Put(pool)
r.Put(pool)
r = newR

switch sixuPlus2NAF[i-1] {
case 1:
a, b, c, newR = lineFunctionAdd(r, aAffine, bAffine, r2, pool)
case -1:
a, b, c, newR = lineFunctionAdd(r, minusA, bAffine, r2, pool)
default:
continue
}

mulLine(ret, a, b, c, pool)
a.Put(pool)
b.Put(pool)
c.Put(pool)
r.Put(pool)
r = newR
}

// In order to calculate Q1 we have to convert q from the sextic twist
// to the full GF(p^12) group, apply the Frobenius there, and convert
// back.
//
// The twist isomorphism is (x', y') -> (x², y³). If we consider just
// x for a moment, then after applying the Frobenius, we have x^(2p)
// where x is the conjugate of x. If we are going to apply the inverse
// isomorphism we need a value with a single coefficient of ² so we
// rewrite this as x^(2p-2)².  =  and, due to the construction of
// p, 2p-2 is a multiple of six. Therefore we can rewrite as
// x^((p-1)/3)² and applying the inverse isomorphism eliminates the
// ².
//
// A similar argument can be made for the y value.

q1 := newTwistPoint(pool)
q1.x.Conjugate(aAffine.x)
q1.x.Mul(q1.x, xiToPMinus1Over3, pool)
q1.y.Conjugate(aAffine.y)
q1.y.Mul(q1.y, xiToPMinus1Over2, pool)
```

```
q1.z.SetOne()
q1.t.SetOne()

// For Q2 we are applying the p² Frobenius. The two conjugations cancel
// out and we are left only with the factors from the isomorphism. In
// the case of x, we end up with a pure number which is why
// xiToPSquaredMinus1Over3 is  GF(p). With y we get a factor of -1. We
// ignore this to end up with -Q2.

minusQ2 := newTwistPoint(pool)
minusQ2.x.MulScalar(aAffine.x, xiToPSquaredMinus1Over3)
minusQ2.y.Set(aAffine.y)
minusQ2.z.SetOne()
minusQ2.t.SetOne()

r2.Square(q1.y, pool)
a, b, c, newR := lineFunctionAdd(r, q1, bAffine, r2, pool)
mulLine(ret, a, b, c, pool)
a.Put(pool)
b.Put(pool)
c.Put(pool)
r.Put(pool)
r = newR

r2.Square(minusQ2.y, pool)
a, b, c, newR = lineFunctionAdd(r, minusQ2, bAffine, r2, pool)
mulLine(ret, a, b, c, pool)
a.Put(pool)
b.Put(pool)
c.Put(pool)
r.Put(pool)
r = newR

aAffine.Put(pool)
bAffine.Put(pool)
minusA.Put(pool)
r.Put(pool)
r2.Put(pool)

return ret
}
```

```
// finalExponentiation computes the (p¹²-1)/Order-th power of an element of
// GF(p¹²) to obtain an element of GT (steps 13-15 of algorithm 1 from
// http://cryptojedi.org/papers/dclxvi-20100714.pdf)
func finalExponentiation(in *gfP12, pool *bnPool) *gfP12 {
t1 := newGFp12(pool)

// This is the p^6-Frobenius
t1.x.Negative(in.x)
t1.y.Set(in.y)

inv := newGFp12(pool)
inv.Invert(in, pool)
t1.Mul(t1, inv, pool)

t2 := newGFp12(pool).FrobeniusP2(t1, pool)
t1.Mul(t1, t2, pool)

fp := newGFp12(pool).Frobenius(t1, pool)
fp2 := newGFp12(pool).FrobeniusP2(t1, pool)
fp3 := newGFp12(pool).Frobenius(fp2, pool)

fu, fu2, fu3 := newGFp12(pool), newGFp12(pool), newGFp12(pool)
fu.Exp(t1, u, pool)
fu2.Exp(fu, u, pool)
fu3.Exp(fu2, u, pool)

y3 := newGFp12(pool).Frobenius(fu, pool)
fu2p := newGFp12(pool).Frobenius(fu2, pool)
fu3p := newGFp12(pool).Frobenius(fu3, pool)
y2 := newGFp12(pool).FrobeniusP2(fu2, pool)

y0 := newGFp12(pool)
y0.Mul(fp, fp2, pool)
y0.Mul(y0, fp3, pool)

y1, y4, y5 := newGFp12(pool), newGFp12(pool), newGFp12(pool)
y1.Conjugate(t1)
y5.Conjugate(fu2)
y3.Conjugate(y3)
y4.Mul(fu, fu2p, pool)
y4.Conjugate(y4)
```

```
y6 := newGFp12(pool)
y6.Mul(fu3, fu3p, pool)
y6.Conjugate(y6)

t0 := newGFp12(pool)
t0.Square(y6, pool)
t0.Mul(t0, y4, pool)
t0.Mul(t0, y5, pool)
t1.Mul(y3, y5, pool)
t1.Mul(t1, t0, pool)
t0.Mul(t0, y2, pool)
t1.Square(t1, pool)
t1.Mul(t1, t0, pool)
t1.Square(t1, pool)
t0.Mul(t1, y1, pool)
t1.Mul(t1, y0, pool)
t0.Square(t0, pool)
t0.Mul(t0, t1, pool)

inv.Put(pool)
t1.Put(pool)
t2.Put(pool)
fp.Put(pool)
fp2.Put(pool)
fp3.Put(pool)
fu.Put(pool)
fu2.Put(pool)
fu3.Put(pool)
fu2p.Put(pool)
fu3p.Put(pool)
y0.Put(pool)
y1.Put(pool)
y2.Put(pool)
y3.Put(pool)
y4.Put(pool)
y5.Put(pool)
y6.Put(pool)

return t0
}

func optimalAte(a *twistPoint, b *curvePoint, pool *bnPool) *gfP12 {
```

```
e := miller(a, b, pool)
ret := finalExponentiation(e, pool)
e.Put(pool)

if a.IsInfinity() || b.IsInfinity() {
ret.SetOne()
}

return ret
}

68:F:\git\coin\ethereum\go-ethereum\crypto\bn256\twist.go
x, y, z, t *gfP2
}

var twistB = &gfP2{
bigFromBase10("266929791119991161246907387137283842545076965332900288569378510910307636690"),
bigFromBase10("19485874751759354771024239261021720505790618469301721065564631296452457478373"),
}

// twistGen is the generator of group G.
var twistGen = &twistPoint{
&gfP2{
bigFromBase10("11559732032986387107991004021392285783925812861821192530917403151452391805634"),
bigFromBase10("10857046999023057135944570762232829481370756359578518086990519993285655852781"),
},
&gfP2{
bigFromBase10("40823678758634336813322034031454355683168513275934012081057410762141200935 31"),
bigFromBase10("84956539231234314176049732474892724384181905872636001487702806493069581019 30"),
},
&gfP2{
bigFromBase10("0"),
bigFromBase10("1"),
},
&gfP2{
bigFromBase10("0"),
```

```
bigFromBase10("1"),
},
}

func newTwistPoint(pool *bnPool) *twistPoint {
return &twistPoint{
newGFp2(pool),
newGFp2(pool),
newGFp2(pool),
newGFp2(pool),
}
}

func (c *twistPoint) String() string {
return "(" + c.x.String() + ", " + c.y.String() + ", " + c.z.String() + ")"
}

func (c *twistPoint) Put(pool *bnPool) {
c.x.Put(pool)
c.y.Put(pool)
c.z.Put(pool)
c.t.Put(pool)
}

func (c *twistPoint) Set(a *twistPoint) {
c.x.Set(a.x)
c.y.Set(a.y)
c.z.Set(a.z)
c.t.Set(a.t)
}

// IsOnCurve returns true iff c is on the curve where c must be in affine form.
func (c *twistPoint) IsOnCurve() bool {
pool := new(bnPool)
yy := newGFp2(pool).Square(c.y, pool)
xxx := newGFp2(pool).Square(c.x, pool)
xxx.Mul(xxx, c.x, pool)
yy.Sub(yy, xxx)
yy.Sub(yy, twistB)
yy.Minimal()
return yy.x.Sign() == 0 && yy.y.Sign() == 0
}
```

```
func (c *twistPoint) SetInfinity() {
c.z.SetZero()
}

func (c *twistPoint) IsInfinity() bool {
return c.z.IsZero()
}

func (c *twistPoint) Add(a, b *twistPoint, pool *bnPool) {
// For additional comments, see the same function in curve.go.

if a.IsInfinity() {
c.Set(b)
return
}
if b.IsInfinity() {
c.Set(a)
return
}

// See http://hyperelliptic.org/EFD/g1p/auto-code/shortw/jacobian-0/addition/add-2007-bl.op3
z1z1 := newGFp2(pool).Square(a.z, pool)
z2z2 := newGFp2(pool).Square(b.z, pool)
u1 := newGFp2(pool).Mul(a.x, z2z2, pool)
u2 := newGFp2(pool).Mul(b.x, z1z1, pool)

t := newGFp2(pool).Mul(b.z, z2z2, pool)
s1 := newGFp2(pool).Mul(a.y, t, pool)

t.Mul(a.z, z1z1, pool)
s2 := newGFp2(pool).Mul(b.y, t, pool)

h := newGFp2(pool).Sub(u2, u1)
xEqual := h.IsZero()

t.Add(h, h)
i := newGFp2(pool).Square(t, pool)
j := newGFp2(pool).Mul(h, i, pool)

t.Sub(s2, s1)
yEqual := t.IsZero()
```

```
if xEqual && yEqual {
c.Double(a, pool)
return
}
r := newGFp2(pool).Add(t, t)

v := newGFp2(pool).Mul(u1, i, pool)

t4 := newGFp2(pool).Square(r, pool)
t.Add(v, v)
t6 := newGFp2(pool).Sub(t4, j)
c.x.Sub(t6, t)

t.Sub(v, c.x)       // t7
t4.Mul(s1, j, pool) // t8
t6.Add(t4, t4)      // t9
t4.Mul(r, t, pool)  // t10
c.y.Sub(t4, t6)

t.Add(a.z, b.z)    // t11
t4.Square(t, pool) // t12
t.Sub(t4, z1z1)    // t13
t4.Sub(t, z2z2)    // t14
c.z.Mul(t4, h, pool)

z1z1.Put(pool)
z2z2.Put(pool)
u1.Put(pool)
u2.Put(pool)
t.Put(pool)
s1.Put(pool)
s2.Put(pool)
h.Put(pool)
i.Put(pool)
j.Put(pool)
r.Put(pool)
v.Put(pool)
t4.Put(pool)
t6.Put(pool)
}

func (c *twistPoint) Double(a *twistPoint, pool *bnPool) {
```

```
// See http://hyperelliptic.org/EFD/g1p/auto-code/shortw/jacobian-0/doubling/dbl-2009-l.op3
A := newGFp2(pool).Square(a.x, pool)
B := newGFp2(pool).Square(a.y, pool)
C_ := newGFp2(pool).Square(B, pool)

t := newGFp2(pool).Add(a.x, B)
t2 := newGFp2(pool).Square(t, pool)
t.Sub(t2, A)
t2.Sub(t, C_)
d := newGFp2(pool).Add(t2, t2)
t.Add(A, A)
e := newGFp2(pool).Add(t, A)
f := newGFp2(pool).Square(e, pool)

t.Add(d, d)
c.x.Sub(f, t)

t.Add(C_, C_)
t2.Add(t, t)
t.Add(t2, t2)
c.y.Sub(d, c.x)
t2.Mul(e, c.y, pool)
c.y.Sub(t2, t)

t.Mul(a.y, a.z, pool)
c.z.Add(t, t)

A.Put(pool)
B.Put(pool)
C_.Put(pool)
t.Put(pool)
t2.Put(pool)
d.Put(pool)
e.Put(pool)
f.Put(pool)
}

func (c *twistPoint) Mul(a *twistPoint, scalar *big.Int, pool *bnPool) *twistPoint {
sum := newTwistPoint(pool)
sum.SetInfinity()
t := newTwistPoint(pool)
```

```
for i := scalar.BitLen(); i >= 0; i-- {
t.Double(sum, pool)
if scalar.Bit(i) != 0 {
sum.Add(t, a, pool)
} else {
sum.Set(t)
}
}

c.Set(sum)
sum.Put(pool)
t.Put(pool)
return c
}

func (c *twistPoint) MakeAffine(pool *bnPool) *twistPoint {
if c.z.IsOne() {
return c
}

zInv := newGFp2(pool).Invert(c.z, pool)
t := newGFp2(pool).Mul(c.y, zInv, pool)
zInv2 := newGFp2(pool).Square(zInv, pool)
c.y.Mul(t, zInv2, pool)
t.Mul(c.x, zInv2, pool)
c.x.Set(t)
c.z.SetOne()
c.t.SetOne()

zInv.Put(pool)
t.Put(pool)
zInv2.Put(pool)

return c
}

func (c *twistPoint) Negative(a *twistPoint, pool *bnPool) {
c.x.Set(a.x)
c.y.SetZero()
c.y.Sub(c.y, a.y)
c.z.Set(a.z)
c.t.SetZero()
```

```
}

69:F:\git\coin\ethereum\go-ethereum\crypto\crypto.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package crypto

import (
"crypto/ecdsa"
"crypto/elliptic"
"crypto/rand"
"encoding/hex"
"errors"
"fmt"
"io"
"io/ioutil"
"math/big"
"os"

"github.com/ethereum/go-ethereum/common"
"github.com/ethereum/go-ethereum/common/math"
"github.com/ethereum/go-ethereum/crypto/sha3"
"github.com/ethereum/go-ethereum/rlp"
)

var (
secp256k1_N, _  =
new(big.Int).SetString("fffffffffffffffffffffffffffffffebaaedce6af48a03bbfd25e8cd0364141", 16)
secp256k1_halfN = new(big.Int).Div(secp256k1_N, big.NewInt(2))
)

// Keccak256 calculates and returns the Keccak256 hash of the input data.
func Keccak256(data ...[]byte) []byte {
d := sha3.NewKeccak256()
for _, b := range data {
d.Write(b)
}
return d.Sum(nil)
}

// Keccak256Hash calculates and returns the Keccak256 hash of the input data,
// converting it to an internal Hash data structure.
```

```
func Keccak256Hash(data ...[]byte) (h common.Hash) {
d := sha3.NewKeccak256()
for _, b := range data {
d.Write(b)
}
d.Sum(h[:0])
return h
}

// Keccak512 calculates and returns the Keccak512 hash of the input data.
func Keccak512(data ...[]byte) []byte {
d := sha3.NewKeccak512()
for _, b := range data {
d.Write(b)
}
return d.Sum(nil)
}

// Creates an ethereum address given the bytes and the nonce
func CreateAddress(b common.Address, nonce uint64) common.Address {
data, _ := rlp.EncodeToBytes([]interface{}{b, nonce})
return common.BytesToAddress(Keccak256(data)[12:])
}

// ToECDSA creates a private key with the given D value.
func ToECDSA(d []byte) (*ecdsa.PrivateKey, error) {
return toECDSA(d, true)
}

// ToECDSAUnsafe blidly converts a binary blob to a private key. It should almost
// never be used unless you are sure the input is valid and want to avoid hitting
// errors due to bad origin encoding (0 prefixes cut off).
func ToECDSAUnsafe(d []byte) *ecdsa.PrivateKey {
priv, _ := toECDSA(d, false)
return priv
}

// toECDSA creates a private key with the given D value. The strict parameter
// controls whether the key's length should be enforced at the curve size or
// it can also accept legacy encodings (0 prefixes).
func toECDSA(d []byte, strict bool) (*ecdsa.PrivateKey, error) {
priv := new(ecdsa.PrivateKey)
```

```
priv.PublicKey.Curve = S256()
if strict && 8*len(d) != priv.Params().BitSize {
return nil, fmt.Errorf("invalid length, need %d bits", priv.Params().BitSize)
}
priv.D = new(big.Int).SetBytes(d)
priv.PublicKey.X, priv.PublicKey.Y = priv.PublicKey.Curve.ScalarBaseMult(d)
return priv, nil
}

// FromECDSA exports a private key into a binary dump.
func FromECDSA(priv *ecdsa.PrivateKey) []byte {
if priv == nil {
return nil
}
return math.PaddedBigBytes(priv.D, priv.Params().BitSize/8)
}

func ToECDSAPub(pub []byte) *ecdsa.PublicKey {
if len(pub) == 0 {
return nil
}
x, y := elliptic.Unmarshal(S256(), pub)
return &ecdsa.PublicKey{Curve: S256(), X: x, Y: y}
}

func FromECDSAPub(pub *ecdsa.PublicKey) []byte {
if pub == nil || pub.X == nil || pub.Y == nil {
return nil
}
return elliptic.Marshal(S256(), pub.X, pub.Y)
}

// HexToECDSA parses a secp256k1 private key.
func HexToECDSA(hexkey string) (*ecdsa.PrivateKey, error) {
b, err := hex.DecodeString(hexkey)
if err != nil {
return nil, errors.New("invalid hex string")
}
return ToECDSA(b)
}

// LoadECDSA loads a secp256k1 private key from the given file.
```

```
func LoadECDSA(file string) (*ecdsa.PrivateKey, error) {
buf := make([]byte, 64)
fd, err := os.Open(file)
if err != nil {
return nil, err
}
defer fd.Close()
if _, err := io.ReadFull(fd, buf); err != nil {
return nil, err
}

key, err := hex.DecodeString(string(buf))
if err != nil {
return nil, err
}
return ToECDSA(key)
}

// SaveECDSA saves a secp256k1 private key to the given file with
// restrictive permissions. The key data is saved hex-encoded.
func SaveECDSA(file string, key *ecdsa.PrivateKey) error {
k := hex.EncodeToString(FromECDSA(key))
return ioutil.WriteFile(file, []byte(k), 0600)
}

func GenerateKey() (*ecdsa.PrivateKey, error) {
return ecdsa.GenerateKey(S256(), rand.Reader)
}

// ValidateSignatureValues verifies whether the signature values are valid with
// the given chain rules. The v value is assumed to be either 0 or 1.
func ValidateSignatureValues(v byte, r, s *big.Int, homestead bool) bool {
if r.Cmp(common.Big1) < 0 || s.Cmp(common.Big1) < 0 {
return false
}
// reject upper range of s values (ECDSA malleability)
// see discussion in secp256k1/libsecp256k1/include/secp256k1.h
if homestead && s.Cmp(secp256k1_halfN) > 0 {
return false
}
// Frontier: allow s to be in full N range
return r.Cmp(secp256k1_N) < 0 && s.Cmp(secp256k1_N) < 0 && (v == 0 || v == 1)
```

```
}

func PubkeyToAddress(p ecdsa.PublicKey) common.Address {
pubBytes := FromECDSAPub(&p)
return common.BytesToAddress(Keccak256(pubBytes[1:])[12:])
}

func zeroBytes(bytes []byte) {
for i := range bytes {
bytes[i] = 0
}
}

70:F:\git\coin\ethereum\go-ethereum\crypto\crypto_test.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package crypto

import (
"bytes"
"crypto/ecdsa"
"encoding/hex"
"fmt"
"io/ioutil"
"math/big"
"os"
"testing"
"time"

"github.com/ethereum/go-ethereum/common"
)

var testAddrHex = "970e8128ab834e8eac17ab8e3812f010678cf791"
var testPrivHex = "289c2857d4598e37fb9647507e47a309d6133539bf21a8b9cb6df88fd5232032"

// These tests are sanity checks.
// They should ensure that we don't e.g. use Sha3-224 instead of Sha3-256
// and that the sha3 library uses keccak-f permutation.
func TestKeccak256Hash(t *testing.T) {
msg := []byte("abc")
exp, _ :=
hex.DecodeString("4e03657aea45a94fc7d47ba826c8d667c0d1e6e33a64a036ec44f58fa12d6c45"
```

```
)
checkhash(t, "Sha3-256-array", func(in []byte) []byte { h := Keccak256Hash(in); return h[:] }, msg,
exp)
}

func BenchmarkSha3(b *testing.B) {
a := []byte("hello world")
amount := 1000000
start := time.Now()
for i := 0; i < amount; i++ {
Keccak256(a)
}

fmt.Println(amount, ":", time.Since(start))
}

func TestSign(t *testing.T) {
key, _ := HexToECDSA(testPrivHex)
addr := common.HexToAddress(testAddrHex)

msg := Keccak256([]byte("foo"))
sig, err := Sign(msg, key)
if err != nil {
t.Errorf("Sign error: %s", err)
}
recoveredPub, err := Ecrecover(msg, sig)
if err != nil {
t.Errorf("ECRecover error: %s", err)
}
pubKey := ToECDSAPub(recoveredPub)
recoveredAddr := PubkeyToAddress(*pubKey)
if addr != recoveredAddr {
t.Errorf("Address mismatch: want: %x have: %x", addr, recoveredAddr)
}

// should be equal to SigToPub
recoveredPub2, err := SigToPub(msg, sig)
if err != nil {
t.Errorf("ECRecover error: %s", err)
}
recoveredAddr2 := PubkeyToAddress(*recoveredPub2)
if addr != recoveredAddr2 {
```

```
t.Errorf("Address mismatch: want: %x have: %x", addr, recoveredAddr2)
}
}

func TestInvalidSign(t *testing.T) {
if _, err := Sign(make([]byte, 1), nil); err == nil {
t.Errorf("expected sign with hash 1 byte to error")
}
if _, err := Sign(make([]byte, 33), nil); err == nil {
t.Errorf("expected sign with hash 33 byte to error")
}
}

func TestNewContractAddress(t *testing.T) {
key, _ := HexToECDSA(testPrivHex)
addr := common.HexToAddress(testAddrHex)
genAddr := PubkeyToAddress(key.PublicKey)
// sanity check before using addr to create contract address
checkAddr(t, genAddr, addr)

caddr0 := CreateAddress(addr, 0)
caddr1 := CreateAddress(addr, 1)
caddr2 := CreateAddress(addr, 2)
checkAddr(t, common.HexToAddress("333c3310824b7c685133f2bedb2ca4b8b4df633d"), caddr0)
checkAddr(t, common.HexToAddress("8bda78331c916a08481428e4b07c96d3e916d165"),
caddr1)
checkAddr(t, common.HexToAddress("c9ddedf451bc62ce88bf9292afb13df35b670699"), caddr2)
}

func TestLoadECDSAFile(t *testing.T) {
keyBytes := common.FromHex(testPrivHex)
fileName0 := "test_key0"
fileName1 := "test_key1"
checkKey := func(k *ecdsa.PrivateKey) {
checkAddr(t, PubkeyToAddress(k.PublicKey), common.HexToAddress(testAddrHex))
loadedKeyBytes := FromECDSA(k)
if !bytes.Equal(loadedKeyBytes, keyBytes) {
t.Fatalf("private key mismatch: want: %x have: %x", keyBytes, loadedKeyBytes)
}
}

ioutil.WriteFile(fileName0, []byte(testPrivHex), 0600)
```

```
defer os.Remove(fileName0)

key0, err := LoadECDSA(fileName0)
if err != nil {
t.Fatal(err)
}
checkKey(key0)

// again, this time with SaveECDSA instead of manual save:
err = SaveECDSA(fileName1, key0)
if err != nil {
t.Fatal(err)
}
defer os.Remove(fileName1)

key1, err := LoadECDSA(fileName1)
if err != nil {
t.Fatal(err)
}
checkKey(key1)
}

func TestValidateSignatureValues(t *testing.T) {
check := func(expected bool, v byte, r, s *big.Int) {
if ValidateSignatureValues(v, r, s, false) != expected {
t.Errorf("mismatch for v: %d r: %d s: %d want: %v", v, r, s, expected)
}
}
minusOne := big.NewInt(-1)
one := common.Big1
zero := common.Big0
secp256k1nMinus1 := new(big.Int).Sub(secp256k1_N, common.Big1)

// correct v,r,s
check(true, 0, one, one)
check(true, 1, one, one)
// incorrect v, correct r,s,
check(false, 2, one, one)
check(false, 3, one, one)

// incorrect v, combinations of incorrect/correct r,s at lower limit
check(false, 2, zero, zero)
```

```
check(false, 2, zero, one)
check(false, 2, one, zero)
check(false, 2, one, one)

// correct v for any combination of incorrect r,s
check(false, 0, zero, zero)
check(false, 0, zero, one)
check(false, 0, one, zero)

check(false, 1, zero, zero)
check(false, 1, zero, one)
check(false, 1, one, zero)

// correct sig with max r,s
check(true, 0, secp256k1nMinus1, secp256k1nMinus1)
// correct v, combinations of incorrect r,s at upper limit
check(false, 0, secp256k1_N, secp256k1nMinus1)
check(false, 0, secp256k1nMinus1, secp256k1_N)
check(false, 0, secp256k1_N, secp256k1_N)

// current callers ensures r,s cannot be negative, but let's test for that too
// as crypto package could be used stand-alone
check(false, 0, minusOne, one)
check(false, 0, one, minusOne)
}

func checkhash(t *testing.T, name string, f func([]byte) []byte, msg, exp []byte) {
sum := f(msg)
if !bytes.Equal(exp, sum) {
t.Fatalf("hash %s mismatch: want: %x have: %x", name, exp, sum)
}
}

func checkAddr(t *testing.T, addr0, addr1 common.Address) {
if addr0 != addr1 {
t.Fatalf("address mismatch: want: %x have: %x", addr0, addr1)
}
}

// test to help Python team with integration of libsecp256k1
// skip but keep it after they are done
func TestPythonIntegration(t *testing.T) {
```

```
kh := "289c2857d4598e37fb9647507e47a309d6133539bf21a8b9cb6df88fd5232032"
k0, _ := HexToECDSA(kh)

msg0 := Keccak256([]byte("foo"))
sig0, _ := Sign(msg0, k0)

msg1 := common.FromHex("00000000000000000000000000000000")
sig1, _ := Sign(msg0, k0)

t.Logf("msg: %x, privkey: %s sig: %x\n", msg0, kh, sig0)
t.Logf("msg: %x, privkey: %s sig: %x\n", msg1, kh, sig1)
}

71:F:\git\coin\ethereum\go-ethereum\crypto\ecies\asn1.go
// contributors may be used to endorse or promote products derived from
// this software without specific prior written permission.
//
// THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
// "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
// LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
// A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
// OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
// SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
// LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
// DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
// THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
// (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
// OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

package ecies

import (
"bytes"
"crypto"
"crypto/elliptic"
"crypto/sha1"
"crypto/sha256"
"crypto/sha512"
"encoding/asn1"
"encoding/pem"
"fmt"
"hash"
```

```
"math/big"

ethcrypto "github.com/ethereum/go-ethereum/crypto"
)

var (
secgScheme     = []int{1, 3, 132, 1}
shaScheme      = []int{2, 16, 840, 1, 101, 3, 4, 2}
ansiX962Scheme = []int{1, 2, 840, 10045}
x963Scheme     = []int{1, 2, 840, 63, 0}
)

var ErrInvalidPrivateKey = fmt.Errorf("ecies: invalid private key")

func doScheme(base, v []int) asn1.ObjectIdentifier {
var oidInts asn1.ObjectIdentifier
oidInts = append(oidInts, base...)
return append(oidInts, v...)
}

// curve OID code taken from crypto/x509, including
//- oidNameCurve*
//- namedCurveFromOID
//- oidFromNamedCurve
// RFC 5480, 2.1.1.1. Named Curve
//
// secp224r1 OBJECT IDENTIFIER ::= {
//   iso(1) identified-organization(3) certicom(132) curve(0) 33 }
//
// secp256r1 OBJECT IDENTIFIER ::= {
//   iso(1) member-body(2) us(840) ansi-X9-62(10045) curves(3)
//   prime(1) 7 }
//
// secp384r1 OBJECT IDENTIFIER ::= {
//   iso(1) identified-organization(3) certicom(132) curve(0) 34 }
//
// secp521r1 OBJECT IDENTIFIER ::= {
//   iso(1) identified-organization(3) certicom(132) curve(0) 35 }
//
// NB: secp256r1 is equivalent to prime256v1
type secgNamedCurve asn1.ObjectIdentifier
```

```
var (
secgNamedCurveS256 = secgNamedCurve{1, 3, 132, 0, 10}
secgNamedCurveP256 = secgNamedCurve{1, 2, 840, 10045, 3, 1, 7}
secgNamedCurveP384 = secgNamedCurve{1, 3, 132, 0, 34}
secgNamedCurveP521 = secgNamedCurve{1, 3, 132, 0, 35}
rawCurveP256       = []byte{6, 8, 4, 2, 1, 3, 4, 7, 2, 2, 0, 6, 6, 1, 3, 1, 7}
rawCurveP384       = []byte{6, 5, 4, 3, 1, 2, 9, 4, 0, 3, 4}
rawCurveP521       = []byte{6, 5, 4, 3, 1, 2, 9, 4, 0, 3, 5}
)

func rawCurve(curve elliptic.Curve) []byte {
switch curve {
case elliptic.P256():
return rawCurveP256
case elliptic.P384():
return rawCurveP384
case elliptic.P521():
return rawCurveP521
default:
return nil
}
}

func (curve secgNamedCurve) Equal(curve2 secgNamedCurve) bool {
if len(curve) != len(curve2) {
return false
}
for i := range curve {
if curve[i] != curve2[i] {
return false
}
}
return true
}

func namedCurveFromOID(curve secgNamedCurve) elliptic.Curve {
switch {
case curve.Equal(secgNamedCurveS256):
return ethcrypto.S256()
case curve.Equal(secgNamedCurveP256):
return elliptic.P256()
case curve.Equal(secgNamedCurveP384):
```

```
return elliptic.P384()
case curve.Equal(secgNamedCurveP521):
return elliptic.P521()
}
return nil
}

func oidFromNamedCurve(curve elliptic.Curve) (secgNamedCurve, bool) {
switch curve {
case elliptic.P256():
return secgNamedCurveP256, true
case elliptic.P384():
return secgNamedCurveP384, true
case elliptic.P521():
return secgNamedCurveP521, true
case ethcrypto.S256():
return secgNamedCurveS256, true
}

return nil, false
}

// asnAlgorithmIdentifier represents the ASN.1 structure of the same name. See RFC
// 5280, section 4.1.1.2.
type asnAlgorithmIdentifier struct {
Algorithm  asn1.ObjectIdentifier
Parameters asn1.RawValue `asn1:"optional"`
}

func (a asnAlgorithmIdentifier) Cmp(b asnAlgorithmIdentifier) bool {
if len(a.Algorithm) != len(b.Algorithm) {
return false
}
for i := range a.Algorithm {
if a.Algorithm[i] != b.Algorithm[i] {
return false
}
}
return true
}

type asnHashFunction asnAlgorithmIdentifier
```

```
var (
oidSHA1   = asn1.ObjectIdentifier{1, 3, 14, 3, 2, 26}
oidSHA224 = doScheme(shaScheme, []int{4})
oidSHA256 = doScheme(shaScheme, []int{1})
oidSHA384 = doScheme(shaScheme, []int{2})
oidSHA512 = doScheme(shaScheme, []int{3})
)

func hashFromOID(oid asn1.ObjectIdentifier) func() hash.Hash {
switch {
case oid.Equal(oidSHA1):
return sha1.New
case oid.Equal(oidSHA224):
return sha256.New224
case oid.Equal(oidSHA256):
return sha256.New
case oid.Equal(oidSHA384):
return sha512.New384
case oid.Equal(oidSHA512):
return sha512.New
}
return nil
}

func oidFromHash(hash crypto.Hash) (asn1.ObjectIdentifier, bool) {
switch hash {
case crypto.SHA1:
return oidSHA1, true
case crypto.SHA224:
return oidSHA224, true
case crypto.SHA256:
return oidSHA256, true
case crypto.SHA384:
return oidSHA384, true
case crypto.SHA512:
return oidSHA512, true
default:
return nil, false
}
}
```

```
var (
asnAlgoSHA1 = asnHashFunction{
Algorithm: oidSHA1,
}
asnAlgoSHA224 = asnHashFunction{
Algorithm: oidSHA224,
}
asnAlgoSHA256 = asnHashFunction{
Algorithm: oidSHA256,
}
asnAlgoSHA384 = asnHashFunction{
Algorithm: oidSHA384,
}
asnAlgoSHA512 = asnHashFunction{
Algorithm: oidSHA512,
}
)

// type ASNasnSubjectPublicKeyInfo struct {
//
// }
//

type asnSubjectPublicKeyInfo struct {
Algorithm   asn1.ObjectIdentifier
PublicKey   asn1.BitString
Supplements ecpksSupplements `asn1:"optional"`
}

type asnECPKAlgorithms struct {
Type asn1.ObjectIdentifier
}

var idPublicKeyType = doScheme(ansiX962Scheme, []int{2})
var idEcPublicKey = doScheme(idPublicKeyType, []int{1})
var idEcPublicKeySupplemented = doScheme(idPublicKeyType, []int{0})

func curveToRaw(curve elliptic.Curve) (rv asn1.RawValue, ok bool) {
switch curve {
case elliptic.P256(), elliptic.P384(), elliptic.P521():
raw := rawCurve(curve)
return asn1.RawValue{
```

```
Tag:       30,
Bytes:     raw[2:],
FullBytes: raw,
}, true
default:
return rv, false
}
}

func asnECPublicKeyType(curve elliptic.Curve) (algo asnAlgorithmIdentifier, ok bool) {
raw, ok := curveToRaw(curve)
if !ok {
return
} else {
return asnAlgorithmIdentifier{Algorithm: idEcPublicKey,
Parameters: raw}, true
}
}

type asnECPrivKeyVer int

var asnECPrivKeyVer1 asnECPrivKeyVer = 1

type asnPrivateKey struct {
Version asnECPrivKeyVer
Private []byte
Curve   secgNamedCurve `asn1:"optional"`
Public  asn1.BitString
}

var asnECDH = doScheme(secgScheme, []int{12})

type asnECDHAlgorithm asnAlgorithmIdentifier

var (
dhSinglePass_stdDH_sha1kdf = asnECDHAlgorithm{
Algorithm: doScheme(x963Scheme, []int{2}),
}
dhSinglePass_stdDH_sha256kdf = asnECDHAlgorithm{
Algorithm: doScheme(secgScheme, []int{11, 1}),
}
dhSinglePass_stdDH_sha384kdf = asnECDHAlgorithm{
```

```
Algorithm: doScheme(secgScheme, []int{11, 2}),
}
dhSinglePass_stdDH_sha224kdf = asnECDHAlgorithm{
Algorithm: doScheme(secgScheme, []int{11, 0}),
}
dhSinglePass_stdDH_sha512kdf = asnECDHAlgorithm{
Algorithm: doScheme(secgScheme, []int{11, 3}),
}
)

func (a asnECDHAlgorithm) Cmp(b asnECDHAlgorithm) bool {
if len(a.Algorithm) != len(b.Algorithm) {
return false
}
for i := range a.Algorithm {
if a.Algorithm[i] != b.Algorithm[i] {
return false
}
}
return true
}

// asnNISTConcatenation is the only supported KDF at this time.
type asnKeyDerivationFunction asnAlgorithmIdentifier

var asnNISTConcatenationKDF = asnKeyDerivationFunction{
Algorithm: doScheme(secgScheme, []int{17, 1}),
}

func (a asnKeyDerivationFunction) Cmp(b asnKeyDerivationFunction) bool {
if len(a.Algorithm) != len(b.Algorithm) {
return false
}
for i := range a.Algorithm {
if a.Algorithm[i] != b.Algorithm[i] {
return false
}
}
return true
}

var eciesRecommendedParameters = doScheme(secgScheme, []int{7})
```

```
var eciesSpecifiedParameters = doScheme(secgScheme, []int{8})

type asnECIESParameters struct {
KDF asnKeyDerivationFunction     `asn1:"optional"`
Sym asnSymmetricEncryption       `asn1:"optional"`
MAC asnMessageAuthenticationCode `asn1:"optional"`
}

type asnSymmetricEncryption asnAlgorithmIdentifier

var (
aes128CTRinECIES = asnSymmetricEncryption{
Algorithm: doScheme(secgScheme, []int{21, 0}),
}
aes192CTRinECIES = asnSymmetricEncryption{
Algorithm: doScheme(secgScheme, []int{21, 1}),
}
aes256CTRinECIES = asnSymmetricEncryption{
Algorithm: doScheme(secgScheme, []int{21, 2}),
}
)

func (a asnSymmetricEncryption) Cmp(b asnSymmetricEncryption) bool {
if len(a.Algorithm) != len(b.Algorithm) {
return false
}
for i := range a.Algorithm {
if a.Algorithm[i] != b.Algorithm[i] {
return false
}
}
return true
}

type asnMessageAuthenticationCode asnAlgorithmIdentifier

var (
hmacFull = asnMessageAuthenticationCode{
Algorithm: doScheme(secgScheme, []int{22}),
}
)
```

```
func (a asnMessageAuthenticationCode) Cmp(b asnMessageAuthenticationCode) bool {
if len(a.Algorithm) != len(b.Algorithm) {
return false
}
for i := range a.Algorithm {
if a.Algorithm[i] != b.Algorithm[i] {
return false
}
}
return true
}

type ecpksSupplements struct {
ECDomain      secgNamedCurve
ECCAlgorithms eccAlgorithmSet
}

type eccAlgorithmSet struct {
ECDH  asnECDHAlgorithm   `asn1:"optional"`
ECIES asnECIESParameters `asn1:"optional"`
}

func marshalSubjectPublicKeyInfo(pub *PublicKey) (subj asnSubjectPublicKeyInfo, err error) {
subj.Algorithm = idEcPublicKeySupplemented
curve, ok := oidFromNamedCurve(pub.Curve)
if !ok {
err = ErrInvalidPublicKey
return
}
subj.Supplements.ECDomain = curve
if pub.Params != nil {
subj.Supplements.ECCAlgorithms.ECDH = paramsToASNECDH(pub.Params)
subj.Supplements.ECCAlgorithms.ECIES = paramsToASNECIES(pub.Params)
}
pubkey := elliptic.Marshal(pub.Curve, pub.X, pub.Y)
subj.PublicKey = asn1.BitString{
BitLength: len(pubkey) * 8,
Bytes:     pubkey,
}
return
}
```

```
// Encode a public key to DER format.
func MarshalPublic(pub *PublicKey) ([]byte, error) {
subj, err := marshalSubjectPublicKeyInfo(pub)
if err != nil {
return nil, err
}
return asn1.Marshal(subj)
}

// Decode a DER-encoded public key.
func UnmarshalPublic(in []byte) (pub *PublicKey, err error) {
var subj asnSubjectPublicKeyInfo

if _, err = asn1.Unmarshal(in, &subj); err != nil {
return
}
if !subj.Algorithm.Equal(idEcPublicKeySupplemented) {
err = ErrInvalidPublicKey
return
}
pub = new(PublicKey)
pub.Curve = namedCurveFromOID(subj.Supplements.ECDomain)
x, y := elliptic.Unmarshal(pub.Curve, subj.PublicKey.Bytes)
if x == nil {
err = ErrInvalidPublicKey
return
}
pub.X = x
pub.Y = y
pub.Params = new(ECIESParams)
asnECIEStoParams(subj.Supplements.ECCAlgorithms.ECIES, pub.Params)
asnECDHtoParams(subj.Supplements.ECCAlgorithms.ECDH, pub.Params)
if pub.Params == nil {
if pub.Params = ParamsFromCurve(pub.Curve); pub.Params == nil {
err = ErrInvalidPublicKey
}
}
return
}

func marshalPrivateKey(prv *PrivateKey) (ecprv asnPrivateKey, err error) {
ecprv.Version = asnECPrivKeyVer1
```

```
ecprv.Private = prv.D.Bytes()

var ok bool
ecprv.Curve, ok = oidFromNamedCurve(prv.PublicKey.Curve)
if !ok {
err = ErrInvalidPrivateKey
return
}

var pub []byte
if pub, err = MarshalPublic(&prv.PublicKey); err != nil {
return
} else {
ecprv.Public = asn1.BitString{
BitLength: len(pub) * 8,
Bytes:     pub,
}
}
return
}

// Encode a private key to DER format.
func MarshalPrivate(prv *PrivateKey) ([]byte, error) {
ecprv, err := marshalPrivateKey(prv)
if err != nil {
return nil, err
}
return asn1.Marshal(ecprv)
}

// Decode a private key from a DER-encoded format.
func UnmarshalPrivate(in []byte) (prv *PrivateKey, err error) {
var ecprv asnPrivateKey

if _, err = asn1.Unmarshal(in, &ecprv); err != nil {
return
} else if ecprv.Version != asnECPrivKeyVer1 {
err = ErrInvalidPrivateKey
return
}

privateCurve := namedCurveFromOID(ecprv.Curve)
```

```
if privateCurve == nil {
err = ErrInvalidPrivateKey
return
}

prv = new(PrivateKey)
prv.D = new(big.Int).SetBytes(ecprv.Private)

if pub, err := UnmarshalPublic(ecprv.Public.Bytes); err != nil {
return nil, err
} else {
prv.PublicKey = *pub
}

return
}

// Export a public key to PEM format.
func ExportPublicPEM(pub *PublicKey) (out []byte, err error) {
der, err := MarshalPublic(pub)
if err != nil {
return
}

var block pem.Block
block.Type = "ELLIPTIC CURVE PUBLIC KEY"
block.Bytes = der

buf := new(bytes.Buffer)
err = pem.Encode(buf, &block)
if err != nil {
return
} else {
out = buf.Bytes()
}
return
}

// Export a private key to PEM format.
func ExportPrivatePEM(prv *PrivateKey) (out []byte, err error) {
der, err := MarshalPrivate(prv)
if err != nil {
```

```
return
}

var block pem.Block
block.Type = "ELLIPTIC CURVE PRIVATE KEY"
block.Bytes = der

buf := new(bytes.Buffer)
err = pem.Encode(buf, &block)
if err != nil {
return
} else {
out = buf.Bytes()
}
return
}

// Import a PEM-encoded public key.
func ImportPublicPEM(in []byte) (pub *PublicKey, err error) {
p, _ := pem.Decode(in)
if p == nil || p.Type != "ELLIPTIC CURVE PUBLIC KEY" {
return nil, ErrInvalidPublicKey
}

pub, err = UnmarshalPublic(p.Bytes)
return
}

// Import a PEM-encoded private key.
func ImportPrivatePEM(in []byte) (prv *PrivateKey, err error) {
p, _ := pem.Decode(in)
if p == nil || p.Type != "ELLIPTIC CURVE PRIVATE KEY" {
return nil, ErrInvalidPrivateKey
}

prv, err = UnmarshalPrivate(p.Bytes)
return
}

72:F:\git\coin\ethereum\go-ethereum\crypto\ecies\ecies.go
// contributors may be used to endorse or promote products derived from
// this software without specific prior written permission.
```

package ecies

import (
"crypto/cipher"
"crypto/ecdsa"
"crypto/elliptic"
"crypto/hmac"
"crypto/subtle"
"fmt"
"hash"
"io"
"math/big"
)

var (
ErrImport   = fmt.Errorf("ecies: failed to import key")
ErrInvalidCurve   = fmt.Errorf("ecies: invalid elliptic curve")
ErrInvalidParams   = fmt.Errorf("ecies: invalid ECIES parameters")
ErrInvalidPublicKey   = fmt.Errorf("ecies: invalid public key")
ErrSharedKeyIsPointAtInfinity = fmt.Errorf("ecies: shared key is point at infinity")
ErrSharedKeyTooBig   = fmt.Errorf("ecies: shared key params are too big")
)

// PublicKey is a representation of an elliptic curve public key.
type PublicKey struct {
X *big.Int
Y *big.Int
elliptic.Curve
Params *ECIESParams

```
}

// Export an ECIES public key as an ECDSA public key.
func (pub *PublicKey) ExportECDSA() *ecdsa.PublicKey {
return &ecdsa.PublicKey{Curve: pub.Curve, X: pub.X, Y: pub.Y}
}

// Import an ECDSA public key as an ECIES public key.
func ImportECDSAPublic(pub *ecdsa.PublicKey) *PublicKey {
return &PublicKey{
X:      pub.X,
Y:      pub.Y,
Curve:  pub.Curve,
Params: ParamsFromCurve(pub.Curve),
}
}

// PrivateKey is a representation of an elliptic curve private key.
type PrivateKey struct {
PublicKey
D *big.Int
}

// Export an ECIES private key as an ECDSA private key.
func (prv *PrivateKey) ExportECDSA() *ecdsa.PrivateKey {
pub := &prv.PublicKey
pubECDSA := pub.ExportECDSA()
return &ecdsa.PrivateKey{PublicKey: *pubECDSA, D: prv.D}
}

// Import an ECDSA private key as an ECIES private key.
func ImportECDSA(prv *ecdsa.PrivateKey) *PrivateKey {
pub := ImportECDSAPublic(&prv.PublicKey)
return &PrivateKey{*pub, prv.D}
}

// Generate an elliptic curve public / private keypair. If params is nil,
// the recommended default parameters for the key will be chosen.
func GenerateKey(rand io.Reader, curve elliptic.Curve, params *ECIESParams) (prv *PrivateKey,
err error) {
pb, x, y, err := elliptic.GenerateKey(curve, rand)
if err != nil {
```

```
return
}
prv = new(PrivateKey)
prv.PublicKey.X = x
prv.PublicKey.Y = y
prv.PublicKey.Curve = curve
prv.D = new(big.Int).SetBytes(pb)
if params == nil {
params = ParamsFromCurve(curve)
}
prv.PublicKey.Params = params
return
}

// MaxSharedKeyLength returns the maximum length of the shared key the
// public key can produce.
func MaxSharedKeyLength(pub *PublicKey) int {
return (pub.Curve.Params().BitSize + 7) / 8
}

// ECDH key agreement method used to establish secret keys for encryption.
func (prv *PrivateKey) GenerateShared(pub *PublicKey, skLen, macLen int) (sk []byte, err error) {
if prv.PublicKey.Curve != pub.Curve {
return nil, ErrInvalidCurve
}
if skLen+macLen > MaxSharedKeyLength(pub) {
return nil, ErrSharedKeyTooBig
}

x, _ := pub.Curve.ScalarMult(pub.X, pub.Y, prv.D.Bytes())
if x == nil {
return nil, ErrSharedKeyIsPointAtInfinity
}

sk = make([]byte, skLen+macLen)
skBytes := x.Bytes()
copy(sk[len(sk)-len(skBytes):], skBytes)
return sk, nil
}

var (
ErrKeyDataTooLong = fmt.Errorf("ecies: can't supply requested key data")
```

```
ErrSharedTooLong  = fmt.Errorf("ecies: shared secret is too long")
ErrInvalidMessage = fmt.Errorf("ecies: invalid message")
)

var (
big2To32   = new(big.Int).Exp(big.NewInt(2), big.NewInt(32), nil)
big2To32M1 = new(big.Int).Sub(big2To32, big.NewInt(1))
)

func incCounter(ctr []byte) {
if ctr[3]++; ctr[3] != 0 {
return
} else if ctr[2]++; ctr[2] != 0 {
return
} else if ctr[1]++; ctr[1] != 0 {
return
} else if ctr[0]++; ctr[0] != 0 {
return
}
return
}

// NIST SP 800-56 Concatenation Key Derivation Function (see section 5.8.1).
func concatKDF(hash hash.Hash, z, s1 []byte, kdLen int) (k []byte, err error) {
if s1 == nil {
s1 = make([]byte, 0)
}

reps := ((kdLen + 7) * 8) / (hash.BlockSize() * 8)
if big.NewInt(int64(reps)).Cmp(big2To32M1) > 0 {
fmt.Println(big2To32M1)
return nil, ErrKeyDataTooLong
}

counter := []byte{0, 0, 0, 1}
k = make([]byte, 0)

for i := 0; i <= reps; i++ {
hash.Write(counter)
hash.Write(z)
hash.Write(s1)
k = append(k, hash.Sum(nil)...)
```

```
hash.Reset()
incCounter(counter)
}

k = k[:kdLen]
return
}

// messageTag computes the MAC of a message (called the tag) as per
// SEC 1, 3.5.
func messageTag(hash func() hash.Hash, km, msg, shared []byte) []byte {
mac := hmac.New(hash, km)
mac.Write(msg)
mac.Write(shared)
tag := mac.Sum(nil)
return tag
}

// Generate an initialisation vector for CTR mode.
func generateIV(params *ECIESParams, rand io.Reader) (iv []byte, err error) {
iv = make([]byte, params.BlockSize)
_, err = io.ReadFull(rand, iv)
return
}

// symEncrypt carries out CTR encryption using the block cipher specified in the
// parameters.
func symEncrypt(rand io.Reader, params *ECIESParams, key, m []byte) (ct []byte, err error) {
c, err := params.Cipher(key)
if err != nil {
return
}

iv, err := generateIV(params, rand)
if err != nil {
return
}
ctr := cipher.NewCTR(c, iv)

ct = make([]byte, len(m)+params.BlockSize)
copy(ct, iv)
ctr.XORKeyStream(ct[params.BlockSize:], m)
```

```
return
}

// symDecrypt carries out CTR decryption using the block cipher specified in
// the parameters
func symDecrypt(rand io.Reader, params *ECIESParams, key, ct []byte) (m []byte, err error) {
c, err := params.Cipher(key)
if err != nil {
return
}

ctr := cipher.NewCTR(c, ct[:params.BlockSize])

m = make([]byte, len(ct)-params.BlockSize)
ctr.XORKeyStream(m, ct[params.BlockSize:])
return
}

// Encrypt encrypts a message using ECIES as specified in SEC 1, 5.1.
//
// s1 and s2 contain shared information that is not part of the resulting
// ciphertext. s1 is fed into key derivation, s2 is fed into the MAC. If the
// shared information parameters aren't being used, they should be nil.
func Encrypt(rand io.Reader, pub *PublicKey, m, s1, s2 []byte) (ct []byte, err error) {
params := pub.Params
if params == nil {
if params = ParamsFromCurve(pub.Curve); params == nil {
err = ErrUnsupportedECIESParameters
return
}
}
R, err := GenerateKey(rand, pub.Curve, params)
if err != nil {
return
}

hash := params.Hash()
z, err := R.GenerateShared(pub, params.KeyLen, params.KeyLen)
if err != nil {
return
}
K, err := concatKDF(hash, z, s1, params.KeyLen+params.KeyLen)
```

```
if err != nil {
return
}
Ke := K[:params.KeyLen]
Km := K[params.KeyLen:]
hash.Write(Km)
Km = hash.Sum(nil)
hash.Reset()

em, err := symEncrypt(rand, params, Ke, m)
if err != nil || len(em) <= params.BlockSize {
return
}

d := messageTag(params.Hash, Km, em, s2)

Rb := elliptic.Marshal(pub.Curve, R.PublicKey.X, R.PublicKey.Y)
ct = make([]byte, len(Rb)+len(em)+len(d))
copy(ct, Rb)
copy(ct[len(Rb):], em)
copy(ct[len(Rb)+len(em):], d)
return
}

// Decrypt decrypts an ECIES ciphertext.
func (prv *PrivateKey) Decrypt(rand io.Reader, c, s1, s2 []byte) (m []byte, err error) {
if len(c) == 0 {
return nil, ErrInvalidMessage
}
params := prv.PublicKey.Params
if params == nil {
if params = ParamsFromCurve(prv.PublicKey.Curve); params == nil {
err = ErrUnsupportedECIESParameters
return
}
}
hash := params.Hash()

var (
rLen   int
hLen   int = hash.Size()
mStart int
```

```
mEnd   int
)

switch c[0] {
case 2, 3, 4:
rLen = ((prv.PublicKey.Curve.Params().BitSize + 7) / 4)
if len(c) < (rLen + hLen + 1) {
err = ErrInvalidMessage
return
}
default:
err = ErrInvalidPublicKey
return
}

mStart = rLen
mEnd = len(c) - hLen

R := new(PublicKey)
R.Curve = prv.PublicKey.Curve
R.X, R.Y = elliptic.Unmarshal(R.Curve, c[:rLen])
if R.X == nil {
err = ErrInvalidPublicKey
return
}
if !R.Curve.IsOnCurve(R.X, R.Y) {
err = ErrInvalidCurve
return
}

z, err := prv.GenerateShared(R, params.KeyLen, params.KeyLen)
if err != nil {
return
}

K, err := concatKDF(hash, z, s1, params.KeyLen+params.KeyLen)
if err != nil {
return
}

Ke := K[:params.KeyLen]
Km := K[params.KeyLen:]
```

```
hash.Write(Km)
Km = hash.Sum(nil)
hash.Reset()

d := messageTag(params.Hash, Km, c[mStart:mEnd], s2)
if subtle.ConstantTimeCompare(c[mEnd:], d) != 1 {
err = ErrInvalidMessage
return
}

m, err = symDecrypt(rand, params, Ke, c[mStart:mEnd])
return
}

73:F:\git\coin\ethereum\go-ethereum\crypto\ecies\ecies_test.go
// contributors may be used to endorse or promote products derived from
// this software without specific prior written permission.
//
// THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
// "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
// LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
// A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
// OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
// SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
// LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
// DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
// THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
// (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
// OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

package ecies

import (
"bytes"
"crypto/elliptic"
"crypto/rand"
"crypto/sha256"
"encoding/hex"
"flag"
"fmt"
"io/ioutil"
"math/big"
```

```
"testing"

"github.com/ethereum/go-ethereum/crypto"
)

var dumpEnc bool

func init() {
flDump := flag.Bool("dump", false, "write encrypted test message to file")
flag.Parse()
dumpEnc = *flDump
}

// Ensure the KDF generates appropriately sized keys.
func TestKDF(t *testing.T) {
msg := []byte("Hello, world")
h := sha256.New()

k, err := concatKDF(h, msg, nil, 64)
if err != nil {
fmt.Println(err.Error())
t.FailNow()
}
if len(k) != 64 {
fmt.Printf("KDF: generated key is the wrong size (%d instead of 64\n",
len(k))
t.FailNow()
}
}

var ErrBadSharedKeys = fmt.Errorf("ecies: shared keys don't match")

// cmpParams compares a set of ECIES parameters. We assume, as per the
// docs, that AES is the only supported symmetric encryption algorithm.
func cmpParams(p1, p2 *ECIESParams) bool {
if p1.hashAlgo != p2.hashAlgo {
return false
} else if p1.KeyLen != p2.KeyLen {
return false
} else if p1.BlockSize != p2.BlockSize {
return false
}
```

```
return true
}

// cmpPublic returns true if the two public keys represent the same pojnt.
func cmpPublic(pub1, pub2 PublicKey) bool {
if pub1.X == nil || pub1.Y == nil {
fmt.Println(ErrInvalidPublicKey.Error())
return false
}
if pub2.X == nil || pub2.Y == nil {
fmt.Println(ErrInvalidPublicKey.Error())
return false
}
pub1Out := elliptic.Marshal(pub1.Curve, pub1.X, pub1.Y)
pub2Out := elliptic.Marshal(pub2.Curve, pub2.X, pub2.Y)

return bytes.Equal(pub1Out, pub2Out)
}

// cmpPrivate returns true if the two private keys are the same.
func cmpPrivate(prv1, prv2 *PrivateKey) bool {
if prv1 == nil || prv1.D == nil {
return false
} else if prv2 == nil || prv2.D == nil {
return false
} else if prv1.D.Cmp(prv2.D) != 0 {
return false
} else {
return cmpPublic(prv1.PublicKey, prv2.PublicKey)
}
}

// Validate the ECDH component.
func TestSharedKey(t *testing.T) {
prv1, err := GenerateKey(rand.Reader, DefaultCurve, nil)
if err != nil {
fmt.Println(err.Error())
t.FailNow()
}
skLen := MaxSharedKeyLength(&prv1.PublicKey) / 2

prv2, err := GenerateKey(rand.Reader, DefaultCurve, nil)
```

```
if err != nil {
fmt.Println(err.Error())
t.FailNow()
}

sk1, err := prv1.GenerateShared(&prv2.PublicKey, skLen, skLen)
if err != nil {
fmt.Println(err.Error())
t.FailNow()
}

sk2, err := prv2.GenerateShared(&prv1.PublicKey, skLen, skLen)
if err != nil {
fmt.Println(err.Error())
t.FailNow()
}

if !bytes.Equal(sk1, sk2) {
fmt.Println(ErrBadSharedKeys.Error())
t.FailNow()
}
}

func TestSharedKeyPadding(t *testing.T) {
// sanity checks
prv0 := hexKey("1adf5c18167d96a1f9a0b1ef63be8aa27eaf6032c233b2b38f7850cf5b859fd9")
prv1 := hexKey("0097a076fc7fcd9208240668e31c9abee952cbb6e375d1b8febc7499d6e16f1a")
x0, _ :=
new(big.Int).SetString("1a8ed022ff7aec59dc1b440446bdda5ff6bcb3509a8b109077282b361efffbd
8", 16)
x1, _ :=
new(big.Int).SetString("6ab3ac374251f638d0abb3ef596d1dc67955b507c104e5f2009724812dc02
7b8", 16)
y0, _ :=
new(big.Int).SetString("e040bd480b1deccc3bc40bd5b1fdcb7bfd352500b477cb9471366dbd4493f
923", 16)
y1, _ :=
new(big.Int).SetString("8ad915f2b503a8be6facab6588731fefeb584fd2dfa9a77a5e0bba1ec439e4f
a", 16)

if prv0.PublicKey.X.Cmp(x0) != 0 {
t.Errorf("mismatched prv0.X:\nhave: %x\nwant: %x\n", prv0.PublicKey.X.Bytes(), x0.Bytes())
```

```
}
if prv0.PublicKey.Y.Cmp(y0) != 0 {
t.Errorf("mismatched prv0.Y:\nhave: %x\nwant: %x\n", prv0.PublicKey.Y.Bytes(), y0.Bytes())
}
if prv1.PublicKey.X.Cmp(x1) != 0 {
t.Errorf("mismatched prv1.X:\nhave: %x\nwant: %x\n", prv1.PublicKey.X.Bytes(), x1.Bytes())
}
if prv1.PublicKey.Y.Cmp(y1) != 0 {
t.Errorf("mismatched prv1.Y:\nhave: %x\nwant: %x\n", prv1.PublicKey.Y.Bytes(), y1.Bytes())
}

// test shared secret generation
sk1, err := prv0.GenerateShared(&prv1.PublicKey, 16, 16)
if err != nil {
fmt.Println(err.Error())
}

sk2, err := prv1.GenerateShared(&prv0.PublicKey, 16, 16)
if err != nil {
t.Fatal(err.Error())
}

if !bytes.Equal(sk1, sk2) {
t.Fatal(ErrBadSharedKeys.Error())
}
}

// Verify that the key generation code fails when too much key data is
// requested.
func TestTooBigSharedKey(t *testing.T) {
prv1, err := GenerateKey(rand.Reader, DefaultCurve, nil)
if err != nil {
fmt.Println(err.Error())
t.FailNow()
}

prv2, err := GenerateKey(rand.Reader, DefaultCurve, nil)
if err != nil {
fmt.Println(err.Error())
t.FailNow()
}
```

```
_, err = prv1.GenerateShared(&prv2.PublicKey, 32, 32)
if err != ErrSharedKeyTooBig {
fmt.Println("ecdh: shared key should be too large for curve")
t.FailNow()
}

_, err = prv2.GenerateShared(&prv1.PublicKey, 32, 32)
if err != ErrSharedKeyTooBig {
fmt.Println("ecdh: shared key should be too large for curve")
t.FailNow()
}
}

// Ensure a public key can be successfully marshalled and unmarshalled, and
// that the decoded key is the same as the original.
func TestMarshalPublic(t *testing.T) {
prv, err := GenerateKey(rand.Reader, DefaultCurve, nil)
if err != nil {
t.Fatalf("GenerateKey error: %s", err)
}

out, err := MarshalPublic(&prv.PublicKey)
if err != nil {
t.Fatalf("MarshalPublic error: %s", err)
}

pub, err := UnmarshalPublic(out)
if err != nil {
t.Fatalf("UnmarshalPublic error: %s", err)
}

if !cmpPublic(prv.PublicKey, *pub) {
t.Fatal("ecies: failed to unmarshal public key")
}
}

// Ensure that a private key can be encoded into DER format, and that
// the resulting key is properly parsed back into a public key.
func TestMarshalPrivate(t *testing.T) {
prv, err := GenerateKey(rand.Reader, DefaultCurve, nil)
if err != nil {
fmt.Println(err.Error())
```

```
t.FailNow()
}

out, err := MarshalPrivate(prv)
if err != nil {
fmt.Println(err.Error())
t.FailNow()
}

if dumpEnc {
ioutil.WriteFile("test.out", out, 0644)
}

prv2, err := UnmarshalPrivate(out)
if err != nil {
fmt.Println(err.Error())
t.FailNow()
}

if !cmpPrivate(prv, prv2) {
fmt.Println("ecdh: private key import failed")
t.FailNow()
}
}

// Ensure that a private key can be successfully encoded to PEM format, and
// the resulting key is properly parsed back in.
func TestPrivatePEM(t *testing.T) {
prv, err := GenerateKey(rand.Reader, DefaultCurve, nil)
if err != nil {
fmt.Println(err.Error())
t.FailNow()
}

out, err := ExportPrivatePEM(prv)
if err != nil {
fmt.Println(err.Error())
t.FailNow()
}

if dumpEnc {
ioutil.WriteFile("test.key", out, 0644)
```

```
}

prv2, err := ImportPrivatePEM(out)
if err != nil {
fmt.Println(err.Error())
t.FailNow()
} else if !cmpPrivate(prv, prv2) {
fmt.Println("ecdh: import from PEM failed")
t.FailNow()
}
}

// Ensure that a public key can be successfully encoded to PEM format, and
// the resulting key is properly parsed back in.
func TestPublicPEM(t *testing.T) {
prv, err := GenerateKey(rand.Reader, DefaultCurve, nil)
if err != nil {
fmt.Println(err.Error())
t.FailNow()
}

out, err := ExportPublicPEM(&prv.PublicKey)
if err != nil {
fmt.Println(err.Error())
t.FailNow()
}

if dumpEnc {
ioutil.WriteFile("test.pem", out, 0644)
}

pub2, err := ImportPublicPEM(out)
if err != nil {
fmt.Println(err.Error())
t.FailNow()
} else if !cmpPublic(prv.PublicKey, *pub2) {
fmt.Println("ecdh: import from PEM failed")
t.FailNow()
}
}

// Benchmark the generation of P256 keys.
```

```
func BenchmarkGenerateKeyP256(b *testing.B) {
for i := 0; i < b.N; i++ {
if _, err := GenerateKey(rand.Reader, elliptic.P256(), nil); err != nil {
fmt.Println(err.Error())
b.FailNow()
}
}
}

// Benchmark the generation of P256 shared keys.
func BenchmarkGenSharedKeyP256(b *testing.B) {
prv, err := GenerateKey(rand.Reader, elliptic.P256(), nil)
if err != nil {
fmt.Println(err.Error())
b.FailNow()
}
b.ResetTimer()
for i := 0; i < b.N; i++ {
_, err := prv.GenerateShared(&prv.PublicKey, 16, 16)
if err != nil {
fmt.Println(err.Error())
b.FailNow()
}
}
}

// Benchmark the generation of S256 shared keys.
func BenchmarkGenSharedKeyS256(b *testing.B) {
prv, err := GenerateKey(rand.Reader, crypto.S256(), nil)
if err != nil {
fmt.Println(err.Error())
b.FailNow()
}
b.ResetTimer()
for i := 0; i < b.N; i++ {
_, err := prv.GenerateShared(&prv.PublicKey, 16, 16)
if err != nil {
fmt.Println(err.Error())
b.FailNow()
}
}
}
```

```
// Verify that an encrypted message can be successfully decrypted.
func TestEncryptDecrypt(t *testing.T) {
prv1, err := GenerateKey(rand.Reader, DefaultCurve, nil)
if err != nil {
fmt.Println(err.Error())
t.FailNow()
}

prv2, err := GenerateKey(rand.Reader, DefaultCurve, nil)
if err != nil {
fmt.Println(err.Error())
t.FailNow()
}

message := []byte("Hello, world.")
ct, err := Encrypt(rand.Reader, &prv2.PublicKey, message, nil, nil)
if err != nil {
fmt.Println(err.Error())
t.FailNow()
}

pt, err := prv2.Decrypt(rand.Reader, ct, nil, nil)
if err != nil {
fmt.Println(err.Error())
t.FailNow()
}

if !bytes.Equal(pt, message) {
fmt.Println("ecies: plaintext doesn't match message")
t.FailNow()
}

_, err = prv1.Decrypt(rand.Reader, ct, nil, nil)
if err == nil {
fmt.Println("ecies: encryption should not have succeeded")
t.FailNow()
}
}

func TestDecryptShared2(t *testing.T) {
prv, err := GenerateKey(rand.Reader, DefaultCurve, nil)
```

```
if err != nil {
t.Fatal(err)
}
message := []byte("Hello, world.")
shared2 := []byte("shared data 2")
ct, err := Encrypt(rand.Reader, &prv.PublicKey, message, nil, shared2)
if err != nil {
t.Fatal(err)
}

// Check that decrypting with correct shared data works.
pt, err := prv.Decrypt(rand.Reader, ct, nil, shared2)
if err != nil {
t.Fatal(err)
}
if !bytes.Equal(pt, message) {
t.Fatal("ecies: plaintext doesn't match message")
}

// Decrypting without shared data or incorrect shared data fails.
if _, err = prv.Decrypt(rand.Reader, ct, nil, nil); err == nil {
t.Fatal("ecies: decrypting without shared data didn't fail")
}
if _, err = prv.Decrypt(rand.Reader, ct, nil, []byte("garbage")); err == nil {
t.Fatal("ecies: decrypting with incorrect shared data didn't fail")
}
}

// TestMarshalEncryption validates the encode/decode produces a valid
// ECIES encryption key.
func TestMarshalEncryption(t *testing.T) {
prv1, err := GenerateKey(rand.Reader, DefaultCurve, nil)
if err != nil {
fmt.Println(err.Error())
t.FailNow()
}

out, err := MarshalPrivate(prv1)
if err != nil {
fmt.Println(err.Error())
t.FailNow()
}
```

```
prv2, err := UnmarshalPrivate(out)
if err != nil {
fmt.Println(err.Error())
t.FailNow()
}

message := []byte("Hello, world.")
ct, err := Encrypt(rand.Reader, &prv2.PublicKey, message, nil, nil)
if err != nil {
fmt.Println(err.Error())
t.FailNow()
}

pt, err := prv2.Decrypt(rand.Reader, ct, nil, nil)
if err != nil {
fmt.Println(err.Error())
t.FailNow()
}

if !bytes.Equal(pt, message) {
fmt.Println("ecies: plaintext doesn't match message")
t.FailNow()
}

_, err = prv1.Decrypt(rand.Reader, ct, nil, nil)
if err != nil {
fmt.Println(err.Error())
t.FailNow()
}

}

type testCase struct {
Curve    elliptic.Curve
Name     string
Expected bool
}

var testCases = []testCase{
{
Curve:    elliptic.P256(),
```

```
Name:     "P256",
Expected: true,
},
{
Curve:    elliptic.P384(),
Name:     "P384",
Expected: true,
},
{
Curve:    elliptic.P521(),
Name:     "P521",
Expected: true,
},
}

// Test parameter selection for each curve, and that P224 fails automatic
// parameter selection (see README for a discussion of P224). Ensures that
// selecting a set of parameters automatically for the given curve works.
func TestParamSelection(t *testing.T) {
for _, c := range testCases {
testParamSelection(t, c)
}
}

func testParamSelection(t *testing.T, c testCase) {
params := ParamsFromCurve(c.Curve)
if params == nil && c.Expected {
fmt.Printf("%s (%s)\n", ErrInvalidParams.Error(), c.Name)
t.FailNow()
} else if params != nil && !c.Expected {
fmt.Printf("ecies: parameters should be invalid (%s)\n",
c.Name)
t.FailNow()
}

prv1, err := GenerateKey(rand.Reader, DefaultCurve, nil)
if err != nil {
fmt.Printf("%s (%s)\n", err.Error(), c.Name)
t.FailNow()
}

prv2, err := GenerateKey(rand.Reader, DefaultCurve, nil)
```

```
if err != nil {
fmt.Printf("%s (%s)\n", err.Error(), c.Name)
t.FailNow()
}

message := []byte("Hello, world.")
ct, err := Encrypt(rand.Reader, &prv2.PublicKey, message, nil, nil)
if err != nil {
fmt.Printf("%s (%s)\n", err.Error(), c.Name)
t.FailNow()
}

pt, err := prv2.Decrypt(rand.Reader, ct, nil, nil)
if err != nil {
fmt.Printf("%s (%s)\n", err.Error(), c.Name)
t.FailNow()
}

if !bytes.Equal(pt, message) {
fmt.Printf("ecies: plaintext doesn't match message (%s)\n",
c.Name)
t.FailNow()
}

_, err = prv1.Decrypt(rand.Reader, ct, nil, nil)
if err == nil {
fmt.Printf("ecies: encryption should not have succeeded (%s)\n",
c.Name)
t.FailNow()
}

}

// Ensure that the basic public key validation in the decryption operation
// works.
func TestBasicKeyValidation(t *testing.T) {
badBytes := []byte{0, 1, 5, 6, 7, 8, 9}

prv, err := GenerateKey(rand.Reader, DefaultCurve, nil)
if err != nil {
fmt.Println(err.Error())
t.FailNow()
```

```
}

message := []byte("Hello, world.")
ct, err := Encrypt(rand.Reader, &prv.PublicKey, message, nil, nil)
if err != nil {
fmt.Println(err.Error())
t.FailNow()
}

for _, b := range badBytes {
ct[0] = b
_, err := prv.Decrypt(rand.Reader, ct, nil, nil)
if err != ErrInvalidPublicKey {
fmt.Println("ecies: validated an invalid key")
t.FailNow()
}
}
}

func TestBox(t *testing.T) {
prv1 := hexKey("4b50fa71f5c3eeb8fdc452224b2395af2fcc3d125e06c32c82e048c0559db03f")
prv2 := hexKey("d0b043b4c5d657670778242d82d68a29d25d7d711127d17b8e299f156dad361a")
pub2 := &prv2.PublicKey

message := []byte("Hello, world.")
ct, err := Encrypt(rand.Reader, pub2, message, nil, nil)
if err != nil {
t.Fatal(err)
}

pt, err := prv2.Decrypt(rand.Reader, ct, nil, nil)
if err != nil {
t.Fatal(err)
}
if !bytes.Equal(pt, message) {
t.Fatal("ecies: plaintext doesn't match message")
}
if _, err = prv1.Decrypt(rand.Reader, ct, nil, nil); err == nil {
t.Fatal("ecies: encryption should not have succeeded")
}
}
```

```
// Verify GenerateShared against static values - useful when
// debugging changes in underlying libs
func TestSharedKeyStatic(t *testing.T) {
prv1 := hexKey("7ebbc6a8358bc76dd73ebc557056702c8cfc34e5cfcd90eb83af0347575fd2ad")
prv2 := hexKey("6a3d6396903245bba5837752b9e0348874e72db0c4e11e9c485a81b4ea4353b9")

skLen := MaxSharedKeyLength(&prv1.PublicKey) / 2

sk1, err := prv1.GenerateShared(&prv2.PublicKey, skLen, skLen)
if err != nil {
fmt.Println(err.Error())
t.FailNow()
}

sk2, err := prv2.GenerateShared(&prv1.PublicKey, skLen, skLen)
if err != nil {
fmt.Println(err.Error())
t.FailNow()
}

if !bytes.Equal(sk1, sk2) {
fmt.Println(ErrBadSharedKeys.Error())
t.FailNow()
}

sk, _ :=
hex.DecodeString("167ccc13ac5e8a26b131c3446030c60fbfac6aa8e31149d0869f93626a4cdf62")
if !bytes.Equal(sk1, sk) {
t.Fatalf("shared secret mismatch: want: %x have: %x", sk, sk1)
}
}

func hexKey(prv string) *PrivateKey {
key, err := crypto.HexToECDSA(prv)
if err != nil {
panic(err)
}
return ImportECDSA(key)
}

74:F:\git\coin\ethereum\go-ethereum\crypto\ecies\params.go
// contributors may be used to endorse or promote products derived from
```

```
// this software without specific prior written permission.
//
// THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
// "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
// LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
// A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
// OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
// SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
// LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
// DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
// THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
// (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
// OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

package ecies

// This file contains parameters for ECIES encryption, specifying the
// symmetric encryption and HMAC parameters.

import (
"crypto"
"crypto/aes"
"crypto/cipher"
"crypto/elliptic"
"crypto/sha256"
"crypto/sha512"
"fmt"
"hash"

ethcrypto "github.com/ethereum/go-ethereum/crypto"
)

var (
DefaultCurve                = ethcrypto.S256()
ErrUnsupportedECDHAlgorithm   = fmt.Errorf("ecies: unsupported ECDH algorithm")
ErrUnsupportedECIESParameters = fmt.Errorf("ecies: unsupported ECIES parameters")
)

type ECIESParams struct {
Hash      func() hash.Hash // hash function
hashAlgo  crypto.Hash
Cipher    func([]byte) (cipher.Block, error) // symmetric cipher
```

```
BlockSize int                       // block size of symmetric cipher
KeyLen    int                       // length of symmetric key
}

// Standard ECIES parameters:
// * ECIES using AES128 and HMAC-SHA-256-16
// * ECIES using AES256 and HMAC-SHA-256-32
// * ECIES using AES256 and HMAC-SHA-384-48
// * ECIES using AES256 and HMAC-SHA-512-64

var (
ECIES_AES128_SHA256 = &ECIESParams{
Hash:      sha256.New,
hashAlgo:  crypto.SHA256,
Cipher:    aes.NewCipher,
BlockSize: aes.BlockSize,
KeyLen:    16,
}

ECIES_AES256_SHA256 = &ECIESParams{
Hash:      sha256.New,
hashAlgo:  crypto.SHA256,
Cipher:    aes.NewCipher,
BlockSize: aes.BlockSize,
KeyLen:    32,
}

ECIES_AES256_SHA384 = &ECIESParams{
Hash:      sha512.New384,
hashAlgo:  crypto.SHA384,
Cipher:    aes.NewCipher,
BlockSize: aes.BlockSize,
KeyLen:    32,
}

ECIES_AES256_SHA512 = &ECIESParams{
Hash:      sha512.New,
hashAlgo:  crypto.SHA512,
Cipher:    aes.NewCipher,
BlockSize: aes.BlockSize,
KeyLen:    32,
}
```

```
)

var paramsFromCurve = map[elliptic.Curve]*ECIESParams{
ethcrypto.S256(): ECIES_AES128_SHA256,
elliptic.P256():  ECIES_AES128_SHA256,
elliptic.P384():  ECIES_AES256_SHA384,
elliptic.P521():  ECIES_AES256_SHA512,
}

func AddParamsForCurve(curve elliptic.Curve, params *ECIESParams) {
paramsFromCurve[curve] = params
}

// ParamsFromCurve selects parameters optimal for the selected elliptic curve.
// Only the curves P256, P384, and P512 are supported.
func ParamsFromCurve(curve elliptic.Curve) (params *ECIESParams) {
return paramsFromCurve[curve]

/*
switch curve {
case elliptic.P256():
return ECIES_AES128_SHA256
case elliptic.P384():
return ECIES_AES256_SHA384
case elliptic.P521():
return ECIES_AES256_SHA512
default:
return nil
}
*/
}

// ASN.1 encode the ECIES parameters relevant to the encryption operations.
func paramsToASNECIES(params *ECIESParams) (asnParams asnECIESParameters) {
if nil == params {
return
}
asnParams.KDF = asnNISTConcatenationKDF
asnParams.MAC = hmacFull
switch params.KeyLen {
case 16:
asnParams.Sym = aes128CTRinECIES
```

```
case 24:
asnParams.Sym = aes192CTRinECIES
case 32:
asnParams.Sym = aes256CTRinECIES
}
return
}

// ASN.1 encode the ECIES parameters relevant to ECDH.
func paramsToASNECDH(params *ECIESParams) (algo asnECDHAlgorithm) {
switch params.hashAlgo {
case crypto.SHA224:
algo = dhSinglePass_stdDH_sha224kdf
case crypto.SHA256:
algo = dhSinglePass_stdDH_sha256kdf
case crypto.SHA384:
algo = dhSinglePass_stdDH_sha384kdf
case crypto.SHA512:
algo = dhSinglePass_stdDH_sha512kdf
}
return
}

// ASN.1 decode the ECIES parameters relevant to the encryption stage.
func asnECIEStoParams(asnParams asnECIESParameters, params *ECIESParams) {
if !asnParams.KDF.Cmp(asnNISTConcatenationKDF) {
params = nil
return
} else if !asnParams.MAC.Cmp(hmacFull) {
params = nil
return
}

switch {
case asnParams.Sym.Cmp(aes128CTRinECIES):
params.KeyLen = 16
params.BlockSize = 16
params.Cipher = aes.NewCipher
case asnParams.Sym.Cmp(aes192CTRinECIES):
params.KeyLen = 24
params.BlockSize = 16
params.Cipher = aes.NewCipher
```

```
case asnParams.Sym.Cmp(aes256CTRinECIES):
params.KeyLen = 32
params.BlockSize = 16
params.Cipher = aes.NewCipher
default:
params = nil
}
}

// ASN.1 decode the ECIES parameters relevant to ECDH.
func asnECDHtoParams(asnParams asnECDHAlgorithm, params *ECIESParams) {
if asnParams.Cmp(dhSinglePass_stdDH_sha224kdf) {
params.hashAlgo = crypto.SHA224
params.Hash = sha256.New224
} else if asnParams.Cmp(dhSinglePass_stdDH_sha256kdf) {
params.hashAlgo = crypto.SHA256
params.Hash = sha256.New
} else if asnParams.Cmp(dhSinglePass_stdDH_sha384kdf) {
params.hashAlgo = crypto.SHA384
params.Hash = sha512.New384
} else if asnParams.Cmp(dhSinglePass_stdDH_sha512kdf) {
params.hashAlgo = crypto.SHA512
params.Hash = sha512.New
} else {
params = nil
}
}

75:F:\git\coin\ethereum\go-ethereum\crypto\randentropy\rand_entropy.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package randentropy

import (
crand "crypto/rand"
"io"
)

var Reader io.Reader = &randEntropy{}

type randEntropy struct {
}
```

```
func (*randEntropy) Read(bytes []byte) (n int, err error) {
readBytes := GetEntropyCSPRNG(len(bytes))
copy(bytes, readBytes)
return len(bytes), nil
}

func GetEntropyCSPRNG(n int) []byte {
mainBuff := make([]byte, n)
_, err := io.ReadFull(crand.Reader, mainBuff)
if err != nil {
panic("reading from crypto/rand failed: " + err.Error())
}
return mainBuff
}

76:F:\git\coin\ethereum\go-ethereum\crypto\secp256k1\curve.go
//   contributors may be used to endorse or promote products derived from
//   this software without specific prior written permission.
// * The name of ThePiachu may not be used to endorse or promote products
//   derived from this software without specific prior written permission.
//
// THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
// "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
// LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
// A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
// OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
// SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
// LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
// DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
// THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
// (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
// OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

package secp256k1

import (
"crypto/elliptic"
"math/big"
"sync"
"unsafe"
```

```
"github.com/ethereum/go-ethereum/common/math"
)

/*
#include "libsecp256k1/include/secp256k1.h"
extern int secp256k1_pubkey_scalar_mul(const secp256k1_context* ctx, const unsigned char
*point, const unsigned char *scalar);
*/
import "C"

// This code is from https://github.com/ThePiachu/GoBit and implements
// several Koblitz elliptic curves over prime fields.
//
// The curve methods, internally, on Jacobian coordinates. For a given
// (x, y) position on the curve, the Jacobian coordinates are (x1, y1,
// z1) where x = x1/z1² and y = y1/z1³. The greatest speedups come
// when the whole calculation can be performed within the transform
// (as in ScalarMult and ScalarBaseMult). But even for Add and Double,
// it's faster to apply and reverse the transform than to operate in
// affine coordinates.

// A BitCurve represents a Koblitz Curve with a=0.
// See http://www.hyperelliptic.org/EFD/g1p/auto-shortw.html
type BitCurve struct {
P       *big.Int // the order of the underlying field
N       *big.Int // the order of the base point
B       *big.Int // the constant of the BitCurve equation
Gx, Gy  *big.Int // (x,y) of the base point
BitSize int      // the size of the underlying field
}

func (BitCurve *BitCurve) Params() *elliptic.CurveParams {
return &elliptic.CurveParams{
P:       BitCurve.P,
N:       BitCurve.N,
B:       BitCurve.B,
Gx:      BitCurve.Gx,
Gy:      BitCurve.Gy,
BitSize: BitCurve.BitSize,
}
}
```

```
// IsOnBitCurve returns true if the given (x,y) lies on the BitCurve.
func (BitCurve *BitCurve) IsOnCurve(x, y *big.Int) bool {
// y² = x³ + b
y2 := new(big.Int).Mul(y, y) //y²
y2.Mod(y2, BitCurve.P)       //y²%P

x3 := new(big.Int).Mul(x, x) //x²
x3.Mul(x3, x)                //x³

x3.Add(x3, BitCurve.B) //x³+B
x3.Mod(x3, BitCurve.P) //(x³+B)%P

return x3.Cmp(y2) == 0
}

//TODO: double check if the function is okay
// affineFromJacobian reverses the Jacobian transform. See the comment at the
// top of the file.
func (BitCurve *BitCurve) affineFromJacobian(x, y, z *big.Int) (xOut, yOut *big.Int) {
zinv := new(big.Int).ModInverse(z, BitCurve.P)
zinvsq := new(big.Int).Mul(zinv, zinv)

xOut = new(big.Int).Mul(x, zinvsq)
xOut.Mod(xOut, BitCurve.P)
zinvsq.Mul(zinvsq, zinv)
yOut = new(big.Int).Mul(y, zinvsq)
yOut.Mod(yOut, BitCurve.P)
return
}

// Add returns the sum of (x1,y1) and (x2,y2)
func (BitCurve *BitCurve) Add(x1, y1, x2, y2 *big.Int) (*big.Int, *big.Int) {
z := new(big.Int).SetInt64(1)
return BitCurve.affineFromJacobian(BitCurve.addJacobian(x1, y1, z, x2, y2, z))
}

// addJacobian takes two points in Jacobian coordinates, (x1, y1, z1) and
// (x2, y2, z2) and returns their sum, also in Jacobian form.
func (BitCurve *BitCurve) addJacobian(x1, y1, z1, x2, y2, z2 *big.Int) (*big.Int, *big.Int, *big.Int) {
// See http://hyperelliptic.org/EFD/g1p/auto-shortw-jacobian-0.html#addition-add-2007-bl
z1z1 := new(big.Int).Mul(z1, z1)
z1z1.Mod(z1z1, BitCurve.P)
```

```
z2z2 := new(big.Int).Mul(z2, z2)
z2z2.Mod(z2z2, BitCurve.P)

u1 := new(big.Int).Mul(x1, z2z2)
u1.Mod(u1, BitCurve.P)
u2 := new(big.Int).Mul(x2, z1z1)
u2.Mod(u2, BitCurve.P)
h := new(big.Int).Sub(u2, u1)
if h.Sign() == -1 {
h.Add(h, BitCurve.P)
}
i := new(big.Int).Lsh(h, 1)
i.Mul(i, i)
j := new(big.Int).Mul(h, i)

s1 := new(big.Int).Mul(y1, z2)
s1.Mul(s1, z2z2)
s1.Mod(s1, BitCurve.P)
s2 := new(big.Int).Mul(y2, z1)
s2.Mul(s2, z1z1)
s2.Mod(s2, BitCurve.P)
r := new(big.Int).Sub(s2, s1)
if r.Sign() == -1 {
r.Add(r, BitCurve.P)
}
r.Lsh(r, 1)
v := new(big.Int).Mul(u1, i)

x3 := new(big.Int).Set(r)
x3.Mul(x3, x3)
x3.Sub(x3, j)
x3.Sub(x3, v)
x3.Sub(x3, v)
x3.Mod(x3, BitCurve.P)

y3 := new(big.Int).Set(r)
v.Sub(v, x3)
y3.Mul(y3, v)
s1.Mul(s1, j)
s1.Lsh(s1, 1)
y3.Sub(y3, s1)
y3.Mod(y3, BitCurve.P)
```

```
z3 := new(big.Int).Add(z1, z2)
z3.Mul(z3, z3)
z3.Sub(z3, z1z1)
if z3.Sign() == -1 {
z3.Add(z3, BitCurve.P)
}
z3.Sub(z3, z2z2)
if z3.Sign() == -1 {
z3.Add(z3, BitCurve.P)
}
z3.Mul(z3, h)
z3.Mod(z3, BitCurve.P)

return x3, y3, z3
}

// Double returns 2*(x,y)
func (BitCurve *BitCurve) Double(x1, y1 *big.Int) (*big.Int, *big.Int) {
z1 := new(big.Int).SetInt64(1)
return BitCurve.affineFromJacobian(BitCurve.doubleJacobian(x1, y1, z1))
}

// doubleJacobian takes a point in Jacobian coordinates, (x, y, z), and
// returns its double, also in Jacobian form.
func (BitCurve *BitCurve) doubleJacobian(x, y, z *big.Int) (*big.Int, *big.Int, *big.Int) {
// See http://hyperelliptic.org/EFD/g1p/auto-shortw-jacobian-0.html#doubling-dbl-2009-l

a := new(big.Int).Mul(x, x) //X1²
b := new(big.Int).Mul(y, y) //Y1²
c := new(big.Int).Mul(b, b) //B²

d := new(big.Int).Add(x, b) //X1+B
d.Mul(d, d)                 //(X1+B)²
d.Sub(d, a)                 //(X1+B)²-A
d.Sub(d, c)                 //(X1+B)²-A-C
d.Mul(d, big.NewInt(2))     //2*((X1+B)²-A-C)

e := new(big.Int).Mul(big.NewInt(3), a) //3*A
f := new(big.Int).Mul(e, e)             //E²

x3 := new(big.Int).Mul(big.NewInt(2), d) //2*D
```

```
x3.Sub(f, x3)                         //F-2*D
x3.Mod(x3, BitCurve.P)

y3 := new(big.Int).Sub(d, x3)                 //D-X3
y3.Mul(e, y3)                                 //E*(D-X3)
y3.Sub(y3, new(big.Int).Mul(big.NewInt(8), c)) //E*(D-X3)-8*C
y3.Mod(y3, BitCurve.P)

z3 := new(big.Int).Mul(y, z) //Y1*Z1
z3.Mul(big.NewInt(2), z3)    //3*Y1*Z1
z3.Mod(z3, BitCurve.P)

return x3, y3, z3
}

func (BitCurve *BitCurve) ScalarMult(Bx, By *big.Int, scalar []byte) (*big.Int, *big.Int) {
// Ensure scalar is exactly 32 bytes. We pad always, even if
// scalar is 32 bytes long, to avoid a timing side channel.
if len(scalar) > 32 {
panic("can't handle scalars > 256 bits")
}
// NOTE: potential timing issue
padded := make([]byte, 32)
copy(padded[32-len(scalar):], scalar)
scalar = padded

// Do the multiplication in C, updating point.
point := make([]byte, 64)
math.ReadBits(Bx, point[:32])
math.ReadBits(By, point[32:])
pointPtr := (*C.uchar)(unsafe.Pointer(&point[0]))
scalarPtr := (*C.uchar)(unsafe.Pointer(&scalar[0]))
res := C.secp256k1_pubkey_scalar_mul(context, pointPtr, scalarPtr)

// Unpack the result and clear temporaries.
x := new(big.Int).SetBytes(point[:32])
y := new(big.Int).SetBytes(point[32:])
for i := range point {
point[i] = 0
}
for i := range padded {
scalar[i] = 0
```

```
}
if res != 1 {
return nil, nil
}
return x, y
}

// ScalarBaseMult returns k*G, where G is the base point of the group and k is
// an integer in big-endian form.
func (BitCurve *BitCurve) ScalarBaseMult(k []byte) (*big.Int, *big.Int) {
return BitCurve.ScalarMult(BitCurve.Gx, BitCurve.Gy, k)
}

// Marshal converts a point into the form specified in section 4.3.6 of ANSI
// X9.62.
func (BitCurve *BitCurve) Marshal(x, y *big.Int) []byte {
byteLen := (BitCurve.BitSize + 7) >> 3

ret := make([]byte, 1+2*byteLen)
ret[0] = 4 // uncompressed point

xBytes := x.Bytes()
copy(ret[1+byteLen-len(xBytes):], xBytes)
yBytes := y.Bytes()
copy(ret[1+2*byteLen-len(yBytes):], yBytes)
return ret
}

// Unmarshal converts a point, serialised by Marshal, into an x, y pair. On
// error, x = nil.
func (BitCurve *BitCurve) Unmarshal(data []byte) (x, y *big.Int) {
byteLen := (BitCurve.BitSize + 7) >> 3
if len(data) != 1+2*byteLen {
return
}
if data[0] != 4 { // uncompressed form
return
}
x = new(big.Int).SetBytes(data[1 : 1+byteLen])
y = new(big.Int).SetBytes(data[1+byteLen:])
return
}
```

```
var (
initonce sync.Once
theCurve *BitCurve
)

// S256 returns a BitCurve which implements secp256k1 (see SEC 2 section 2.7.1)
func S256() *BitCurve {
initonce.Do(func() {
// See SEC 2 section 2.7.1
// curve parameters taken from:
// http://www.secg.org/collateral/sec2_final.pdf
theCurve = new(BitCurve)
theCurve.P, _ =
new(big.Int).SetString("FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFC2F", 16)
theCurve.N, _ =
new(big.Int).SetString("FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD0364141", 16)
theCurve.B, _ =
new(big.Int).SetString("0000000000000000000000000000000000000000000000000000000000000007", 16)
theCurve.Gx, _ =
new(big.Int).SetString("79BE667EF9DCBBAC55A06295CE870B07029BFCDB2DCE28D959F2815B16F81798", 16)
theCurve.Gy, _ =
new(big.Int).SetString("483ADA7726A3C4655DA4FBFC0E1108A8FD17B448A68554199C47D08FFB10D4B8", 16)
theCurve.BitSize = 256
})
return theCurve
}

77:F:\git\coin\ethereum\go-ethereum\crypto\secp256k1\panic_cb.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package secp256k1

import "C"
import "unsafe"

// Callbacks for converting libsecp256k1 internal faults into
```

```
// recoverable Go panics.

//export secp256k1GoPanicIllegal
func secp256k1GoPanicIllegal(msg *C.char, data unsafe.Pointer) {
panic("illegal argument: " + C.GoString(msg))
}

//export secp256k1GoPanicError
func secp256k1GoPanicError(msg *C.char, data unsafe.Pointer) {
panic("internal error: " + C.GoString(msg))
}

78:F:\git\coin\ethereum\go-ethereum\crypto\secp256k1\secp256.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

// Package secp256k1 wraps the bitcoin secp256k1 C library.
package secp256k1

/*
#cgo CFLAGS: -I./libsecp256k1
#cgo CFLAGS: -I./libsecp256k1/src/
#define USE_NUM_NONE
#define USE_FIELD_10X26
#define USE_FIELD_INV_BUILTIN
#define USE_SCALAR_8X32
#define USE_SCALAR_INV_BUILTIN
#define NDEBUG
#include "./libsecp256k1/src/secp256k1.c"
#include "./libsecp256k1/src/modules/recovery/main_impl.h"
#include "ext.h"

typedef void (*callbackFunc) (const char* msg, void* data);
extern void secp256k1GoPanicIllegal(const char* msg, void* data);
extern void secp256k1GoPanicError(const char* msg, void* data);
*/
import "C"

import (
"errors"
"unsafe"
)
```

```
var context *C.secp256k1_context

func init() {
// around 20 ms on a modern CPU.
context = C.secp256k1_context_create_sign_verify()
C.secp256k1_context_set_illegal_callback(context, C.callbackFunc(C.secp256k1GoPanicIllegal),
nil)
C.secp256k1_context_set_error_callback(context, C.callbackFunc(C.secp256k1GoPanicError),
nil)
}

var (
ErrInvalidMsgLen       = errors.New("invalid message length, need 32 bytes")
ErrInvalidSignatureLen = errors.New("invalid signature length")
ErrInvalidRecoveryID   = errors.New("invalid signature recovery id")
ErrInvalidKey          = errors.New("invalid private key")
ErrSignFailed          = errors.New("signing failed")
ErrRecoverFailed       = errors.New("recovery failed")
)

// Sign creates a recoverable ECDSA signature.
// The produced signature is in the 65-byte [R || S || V] format where V is 0 or 1.
//
// The caller is responsible for ensuring that msg cannot be chosen
// directly by an attacker. It is usually preferable to use a cryptographic
// hash function on any input before handing it to this function.
func Sign(msg []byte, seckey []byte) ([]byte, error) {
if len(msg) != 32 {
return nil, ErrInvalidMsgLen
}
if len(seckey) != 32 {
return nil, ErrInvalidKey
}
seckeydata := (*C.uchar)(unsafe.Pointer(&seckey[0]))
if C.secp256k1_ec_seckey_verify(context, seckeydata) != 1 {
return nil, ErrInvalidKey
}

var (
msgdata   = (*C.uchar)(unsafe.Pointer(&msg[0]))
noncefunc = C.secp256k1_nonce_function_rfc6979
sigstruct C.secp256k1_ecdsa_recoverable_signature
```

```
)
if C.secp256k1_ecdsa_sign_recoverable(context, &sigstruct, msgdata, seckeydata, noncefunc, nil)
== 0 {
return nil, ErrSignFailed
}

var (
sig     = make([]byte, 65)
sigdata = (*C.uchar)(unsafe.Pointer(&sig[0]))
recid   C.int
)
C.secp256k1_ecdsa_recoverable_signature_serialize_compact(context, sigdata, &recid,
&sigstruct)
sig[64] = byte(recid) // add back recid to get 65 bytes sig
return sig, nil
}

// RecoverPubkey returns the the public key of the signer.
// msg must be the 32-byte hash of the message to be signed.
// sig must be a 65-byte compact ECDSA signature containing the
// recovery id as the last element.
func RecoverPubkey(msg []byte, sig []byte) ([]byte, error) {
if len(msg) != 32 {
return nil, ErrInvalidMsgLen
}
if err := checkSignature(sig); err != nil {
return nil, err
}

var (
pubkey  = make([]byte, 65)
sigdata = (*C.uchar)(unsafe.Pointer(&sig[0]))
msgdata = (*C.uchar)(unsafe.Pointer(&msg[0]))
)
if C.secp256k1_ecdsa_recover_pubkey(context, (*C.uchar)(unsafe.Pointer(&pubkey[0])), sigdata,
msgdata) == 0 {
return nil, ErrRecoverFailed
}
return pubkey, nil
}

func checkSignature(sig []byte) error {
```

```
if len(sig) != 65 {
return ErrInvalidSignatureLen
}
if sig[64] >= 4 {
return ErrInvalidRecoveryID
}
return nil
}

79:F:\git\coin\ethereum\go-ethereum\crypto\secp256k1\secp256_test.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package secp256k1

import (
"bytes"
"crypto/ecdsa"
"crypto/elliptic"
"crypto/rand"
"encoding/hex"
"testing"

"github.com/ethereum/go-ethereum/common/math"
"github.com/ethereum/go-ethereum/crypto/randentropy"
)

const TestCount = 1000

func generateKeyPair() (pubkey, privkey []byte) {
key, err := ecdsa.GenerateKey(S256(), rand.Reader)
if err != nil {
panic(err)
}
pubkey = elliptic.Marshal(S256(), key.X, key.Y)
return pubkey, math.PaddedBigBytes(key.D, 32)
}

func randSig() []byte {
sig := randentropy.GetEntropyCSPRNG(65)
sig[32] &= 0x70
sig[64] %= 4
return sig
```

```
}

// tests for malleability
// highest bit of signature ECDSA s value must be 0, in the 33th byte
func compactSigCheck(t *testing.T, sig []byte) {
var b int = int(sig[32])
if b < 0 {
t.Errorf("highest bit is negative: %d", b)
}
if ((b >> 7) == 1) != ((b & 0x80) == 0x80) {
t.Errorf("highest bit: %d bit >> 7: %d", b, b>>7)
}
if (b & 0x80) == 0x80 {
t.Errorf("highest bit: %d bit & 0x80: %d", b, b&0x80)
}
}

func TestSignatureValidity(t *testing.T) {
pubkey, seckey := generateKeyPair()
msg := randentropy.GetEntropyCSPRNG(32)
sig, err := Sign(msg, seckey)
if err != nil {
t.Errorf("signature error: %s", err)
}
compactSigCheck(t, sig)
if len(pubkey) != 65 {
t.Errorf("pubkey length mismatch: want: 65 have: %d", len(pubkey))
}
if len(seckey) != 32 {
t.Errorf("seckey length mismatch: want: 32 have: %d", len(seckey))
}
if len(sig) != 65 {
t.Errorf("sig length mismatch: want: 65 have: %d", len(sig))
}
recid := int(sig[64])
if recid > 4 || recid < 0 {
t.Errorf("sig recid mismatch: want: within 0 to 4 have: %d", int(sig[64]))
}
}

func TestInvalidRecoveryID(t *testing.T) {
_, seckey := generateKeyPair()
```

```
msg := randentropy.GetEntropyCSPRNG(32)
sig, _ := Sign(msg, seckey)
sig[64] = 99
_, err := RecoverPubkey(msg, sig)
if err != ErrInvalidRecoveryID {
t.Fatalf("got %q, want %q", err, ErrInvalidRecoveryID)
}
}

func TestSignAndRecover(t *testing.T) {
pubkey1, seckey := generateKeyPair()
msg := randentropy.GetEntropyCSPRNG(32)
sig, err := Sign(msg, seckey)
if err != nil {
t.Errorf("signature error: %s", err)
}
pubkey2, err := RecoverPubkey(msg, sig)
if err != nil {
t.Errorf("recover error: %s", err)
}
if !bytes.Equal(pubkey1, pubkey2) {
t.Errorf("pubkey mismatch: want: %x have: %x", pubkey1, pubkey2)
}
}

func TestSignDeterministic(t *testing.T) {
_, seckey := generateKeyPair()
msg := make([]byte, 32)
copy(msg, "hi there")

sig1, err := Sign(msg, seckey)
if err != nil {
t.Fatal(err)
}
sig2, err := Sign(msg, seckey)
if err != nil {
t.Fatal(err)
}
if !bytes.Equal(sig1, sig2) {
t.Fatal("signatures not equal")
}
}
```

```
func TestRandomMessagesWithSameKey(t *testing.T) {
pubkey, seckey := generateKeyPair()
keys := func() ([]byte, []byte) {
return pubkey, seckey
}
signAndRecoverWithRandomMessages(t, keys)
}

func TestRandomMessagesWithRandomKeys(t *testing.T) {
keys := func() ([]byte, []byte) {
pubkey, seckey := generateKeyPair()
return pubkey, seckey
}
signAndRecoverWithRandomMessages(t, keys)
}

func signAndRecoverWithRandomMessages(t *testing.T, keys func() ([]byte, []byte)) {
for i := 0; i < TestCount; i++ {
pubkey1, seckey := keys()
msg := randentropy.GetEntropyCSPRNG(32)
sig, err := Sign(msg, seckey)
if err != nil {
t.Fatalf("signature error: %s", err)
}
if sig == nil {
t.Fatal("signature is nil")
}
compactSigCheck(t, sig)

// TODO: why do we flip around the recovery id?
sig[len(sig)-1] %= 4

pubkey2, err := RecoverPubkey(msg, sig)
if err != nil {
t.Fatalf("recover error: %s", err)
}
if pubkey2 == nil {
t.Error("pubkey is nil")
}
if !bytes.Equal(pubkey1, pubkey2) {
t.Fatalf("pubkey mismatch: want: %x have: %x", pubkey1, pubkey2)
```

```
}
}
}

func TestRecoveryOfRandomSignature(t *testing.T) {
pubkey1, _ := generateKeyPair()
msg := randentropy.GetEntropyCSPRNG(32)

for i := 0; i < TestCount; i++ {
// recovery can sometimes work, but if so should always give wrong pubkey
pubkey2, _ := RecoverPubkey(msg, randSig())
if bytes.Equal(pubkey1, pubkey2) {
t.Fatalf("iteration: %d: pubkey mismatch: do NOT want %x: ", i, pubkey2)
}
}
}

func TestRandomMessagesAgainstValidSig(t *testing.T) {
pubkey1, seckey := generateKeyPair()
msg := randentropy.GetEntropyCSPRNG(32)
sig, _ := Sign(msg, seckey)

for i := 0; i < TestCount; i++ {
msg = randentropy.GetEntropyCSPRNG(32)
pubkey2, _ := RecoverPubkey(msg, sig)
// recovery can sometimes work, but if so should always give wrong pubkey
if bytes.Equal(pubkey1, pubkey2) {
t.Fatalf("iteration: %d: pubkey mismatch: do NOT want %x: ", i, pubkey2)
}
}
}

// Useful when the underlying libsecp256k1 API changes to quickly
// check only recover function without use of signature function
func TestRecoverSanity(t *testing.T) {
msg, _ :=
hex.DecodeString("ce0677bb30baa8cf067c88db9811f4333d131bf8bcf12fe7065d211dce971008")
sig, _ :=
hex.DecodeString("90f27b8b488db00b00606796d2987f6a5f59ae62ea05effe84fef5b8b0e549984a
691139ad57a3f0b906637673aa2f63d1f55cb1a69199d4009eea23ceaddc9301")
pubkey1, _ :=
hex.DecodeString("04e32df42865e97135acfb65f3bae71bdc86f4d49150ad6a440b6f15878109880
```

```
a0a2b2667f7e725ceea70c673093bf67663e0312623c8e091b13cf2c0f11ef652")
pubkey2, err := RecoverPubkey(msg, sig)
if err != nil {
t.Fatalf("recover error: %s", err)
}
if !bytes.Equal(pubkey1, pubkey2) {
t.Errorf("pubkey mismatch: want: %x have: %x", pubkey1, pubkey2)
}
}

func BenchmarkSign(b *testing.B) {
_, seckey := generateKeyPair()
msg := randentropy.GetEntropyCSPRNG(32)
b.ResetTimer()

for i := 0; i < b.N; i++ {
Sign(msg, seckey)
}
}

func BenchmarkRecover(b *testing.B) {
msg := randentropy.GetEntropyCSPRNG(32)
_, seckey := generateKeyPair()
sig, _ := Sign(msg, seckey)
b.ResetTimer()

for i := 0; i < b.N; i++ {
RecoverPubkey(msg, sig)
}
}

80:F:\git\coin\ethereum\go-ethereum\crypto\sha3\doc.go
// bytes of output. The SHAKE instances are faster than the SHA3 instances;
// the latter have to allocate memory to conform to the hash.Hash interface.
//
// If you need a secret-key MAC (message authentication code), prepend the
// secret key to the input, hash with SHAKE256 and read at least 32 bytes of
// output.
//
//
// Security strengths
//
```

```
// The SHA3-x (x equals 224, 256, 384, or 512) functions have a security
// strength against preimage attacks of x bits. Since they only produce "x"
// bits of output, their collision-resistance is only "x/2" bits.
//
// The SHAKE-256 and -128 functions have a generic security strength of 256 and
// 128 bits against all attacks, provided that at least 2x bits of their output
// is used.  Requesting more than 64 or 32 bytes of output, respectively, does
// not increase the collision-resistance of the SHAKE functions.
//
//
// The sponge construction
//
// A sponge builds a pseudo-random function from a public pseudo-random
// permutation, by applying the permutation to a state of "rate + capacity"
// bytes, but hiding "capacity" of the bytes.
//
// A sponge starts out with a zero state. To hash an input using a sponge, up
// to "rate" bytes of the input are XORed into the sponge's state. The sponge
// is then "full" and the permutation is applied to "empty" it. This process is
// repeated until all the input has been "absorbed". The input is then padded.
// The digest is "squeezed" from the sponge in the same way, except that output
// output is copied out instead of input being XORed in.
//
// A sponge is parameterized by its generic security strength, which is equal
// to half its capacity; capacity + rate is equal to the permutation's width.
// Since the KeccakF-1600 permutation is 1600 bits (200 bytes) wide, this means
// that the security strength of a sponge instance is equal to (1600 - bitrate) / 2.
//
//
// Recommendations
//
// The SHAKE functions are recommended for most new uses. They can produce
// output of arbitrary length. SHAKE256, with an output length of at least
// 64 bytes, provides 256-bit security against all attacks.  The Keccak team
// recommends it for most applications upgrading from SHA2-512. (NIST chose a
// much stronger, but much slower, sponge instance for SHA3-512.)
//
// The SHA-3 functions are "drop-in" replacements for the SHA-2 functions.
// They produce output of the same length, with the same security strengths
// against all attacks. This means, in particular, that SHA3-256 only has
// 128-bit collision resistance, because its output length is 32 bytes.
package sha3
```

81:F:\git\coin\ethereum\go-ethereum\crypto\sha3\hashes.go
// NewKeccak256 creates a new Keccak-256 hash.
func NewKeccak256() hash.Hash { return &state{rate: 136, outputLen: 32, dsbyte: 0x01} }

// NewKeccak512 creates a new Keccak-512 hash.
func NewKeccak512() hash.Hash { return &state{rate: 72, outputLen: 64, dsbyte: 0x01} }

// New224 creates a new SHA3-224 hash.
// Its generic security strength is 224 bits against preimage attacks,
// and 112 bits against collision attacks.
func New224() hash.Hash { return &state{rate: 144, outputLen: 28, dsbyte: 0x06} }

// New256 creates a new SHA3-256 hash.
// Its generic security strength is 256 bits against preimage attacks,
// and 128 bits against collision attacks.
func New256() hash.Hash { return &state{rate: 136, outputLen: 32, dsbyte: 0x06} }

// New384 creates a new SHA3-384 hash.
// Its generic security strength is 384 bits against preimage attacks,
// and 192 bits against collision attacks.
func New384() hash.Hash { return &state{rate: 104, outputLen: 48, dsbyte: 0x06} }

// New512 creates a new SHA3-512 hash.
// Its generic security strength is 512 bits against preimage attacks,
// and 256 bits against collision attacks.
func New512() hash.Hash { return &state{rate: 72, outputLen: 64, dsbyte: 0x06} }

// Sum224 returns the SHA3-224 digest of the data.
func Sum224(data []byte) (digest [28]byte) {
h := New224()
h.Write(data)
h.Sum(digest[:0])
return
}

// Sum256 returns the SHA3-256 digest of the data.
func Sum256(data []byte) (digest [32]byte) {
h := New256()
h.Write(data)
h.Sum(digest[:0])
return

```
}

// Sum384 returns the SHA3-384 digest of the data.
func Sum384(data []byte) (digest [48]byte) {
h := New384()
h.Write(data)
h.Sum(digest[:0])
return
}

// Sum512 returns the SHA3-512 digest of the data.
func Sum512(data []byte) (digest [64]byte) {
h := New512()
h.Write(data)
h.Sum(digest[:0])
return
}

82:F:\git\coin\ethereum\go-ethereum\crypto\sha3\keccakf.go
0x000000000000808B,
0x0000000080000001,
0x8000000080008081,
0x8000000000008009,
0x000000000000008A,
0x0000000000000088,
0x0000000080008009,
0x000000008000000A,
0x000000008000808B,
0x800000000000008B,
0x8000000000008089,
0x8000000000008003,
0x8000000000008002,
0x8000000000000080,
0x000000000000800A,
0x800000008000000A,
0x8000000080008081,
0x8000000000008080,
0x0000000080000001,
0x8000000080008008,
}

// keccakF1600 applies the Keccak permutation to a 1600b-wide
```

```
// state represented as a slice of 25 uint64s.
func keccakF1600(a *[25]uint64) {
// Implementation translated from Keccak-inplace.c
// in the keccak reference code.
var t, bc0, bc1, bc2, bc3, bc4, d0, d1, d2, d3, d4 uint64

for i := 0; i < 24; i += 4 {
// Combines the 5 steps in each round into 2 steps.
// Unrolls 4 rounds per loop and spreads some steps across rounds.

// Round 1
bc0 = a[0] ^ a[5] ^ a[10] ^ a[15] ^ a[20]
bc1 = a[1] ^ a[6] ^ a[11] ^ a[16] ^ a[21]
bc2 = a[2] ^ a[7] ^ a[12] ^ a[17] ^ a[22]
bc3 = a[3] ^ a[8] ^ a[13] ^ a[18] ^ a[23]
bc4 = a[4] ^ a[9] ^ a[14] ^ a[19] ^ a[24]
d0 = bc4 ^ (bc1<<1 | bc1>>63)
d1 = bc0 ^ (bc2<<1 | bc2>>63)
d2 = bc1 ^ (bc3<<1 | bc3>>63)
d3 = bc2 ^ (bc4<<1 | bc4>>63)
d4 = bc3 ^ (bc0<<1 | bc0>>63)

bc0 = a[0] ^ d0
t = a[6] ^ d1
bc1 = t<<44 | t>>(64-44)
t = a[12] ^ d2
bc2 = t<<43 | t>>(64-43)
t = a[18] ^ d3
bc3 = t<<21 | t>>(64-21)
t = a[24] ^ d4
bc4 = t<<14 | t>>(64-14)
a[0] = bc0 ^ (bc2 &^ bc1) ^ rc[i]
a[6] = bc1 ^ (bc3 &^ bc2)
a[12] = bc2 ^ (bc4 &^ bc3)
a[18] = bc3 ^ (bc0 &^ bc4)
a[24] = bc4 ^ (bc1 &^ bc0)

t = a[10] ^ d0
bc2 = t<<3 | t>>(64-3)
t = a[16] ^ d1
bc3 = t<<45 | t>>(64-45)
t = a[22] ^ d2
```

```
bc4 = t<<61 | t>>(64-61)
t = a[3] ^ d3
bc0 = t<<28 | t>>(64-28)
t = a[9] ^ d4
bc1 = t<<20 | t>>(64-20)
a[10] = bc0 ^ (bc2 &^ bc1)
a[16] = bc1 ^ (bc3 &^ bc2)
a[22] = bc2 ^ (bc4 &^ bc3)
a[3] = bc3 ^ (bc0 &^ bc4)
a[9] = bc4 ^ (bc1 &^ bc0)

t = a[20] ^ d0
bc4 = t<<18 | t>>(64-18)
t = a[1] ^ d1
bc0 = t<<1 | t>>(64-1)
t = a[7] ^ d2
bc1 = t<<6 | t>>(64-6)
t = a[13] ^ d3
bc2 = t<<25 | t>>(64-25)
t = a[19] ^ d4
bc3 = t<<8 | t>>(64-8)
a[20] = bc0 ^ (bc2 &^ bc1)
a[1] = bc1 ^ (bc3 &^ bc2)
a[7] = bc2 ^ (bc4 &^ bc3)
a[13] = bc3 ^ (bc0 &^ bc4)
a[19] = bc4 ^ (bc1 &^ bc0)

t = a[5] ^ d0
bc1 = t<<36 | t>>(64-36)
t = a[11] ^ d1
bc2 = t<<10 | t>>(64-10)
t = a[17] ^ d2
bc3 = t<<15 | t>>(64-15)
t = a[23] ^ d3
bc4 = t<<56 | t>>(64-56)
t = a[4] ^ d4
bc0 = t<<27 | t>>(64-27)
a[5] = bc0 ^ (bc2 &^ bc1)
a[11] = bc1 ^ (bc3 &^ bc2)
a[17] = bc2 ^ (bc4 &^ bc3)
a[23] = bc3 ^ (bc0 &^ bc4)
a[4] = bc4 ^ (bc1 &^ bc0)
```

```
t = a[15] ^ d0
bc3 = t<<41 | t>>(64-41)
t = a[21] ^ d1
bc4 = t<<2 | t>>(64-2)
t = a[2] ^ d2
bc0 = t<<62 | t>>(64-62)
t = a[8] ^ d3
bc1 = t<<55 | t>>(64-55)
t = a[14] ^ d4
bc2 = t<<39 | t>>(64-39)
a[15] = bc0 ^ (bc2 &^ bc1)
a[21] = bc1 ^ (bc3 &^ bc2)
a[2] = bc2 ^ (bc4 &^ bc3)
a[8] = bc3 ^ (bc0 &^ bc4)
a[14] = bc4 ^ (bc1 &^ bc0)

// Round 2
bc0 = a[0] ^ a[5] ^ a[10] ^ a[15] ^ a[20]
bc1 = a[1] ^ a[6] ^ a[11] ^ a[16] ^ a[21]
bc2 = a[2] ^ a[7] ^ a[12] ^ a[17] ^ a[22]
bc3 = a[3] ^ a[8] ^ a[13] ^ a[18] ^ a[23]
bc4 = a[4] ^ a[9] ^ a[14] ^ a[19] ^ a[24]
d0 = bc4 ^ (bc1<<1 | bc1>>63)
d1 = bc0 ^ (bc2<<1 | bc2>>63)
d2 = bc1 ^ (bc3<<1 | bc3>>63)
d3 = bc2 ^ (bc4<<1 | bc4>>63)
d4 = bc3 ^ (bc0<<1 | bc0>>63)

bc0 = a[0] ^ d0
t = a[16] ^ d1
bc1 = t<<44 | t>>(64-44)
t = a[7] ^ d2
bc2 = t<<43 | t>>(64-43)
t = a[23] ^ d3
bc3 = t<<21 | t>>(64-21)
t = a[14] ^ d4
bc4 = t<<14 | t>>(64-14)
a[0] = bc0 ^ (bc2 &^ bc1) ^ rc[i+1]
a[16] = bc1 ^ (bc3 &^ bc2)
a[7] = bc2 ^ (bc4 &^ bc3)
a[23] = bc3 ^ (bc0 &^ bc4)
```

```
a[14] = bc4 ^ (bc1 &^ bc0)

t = a[20] ^ d0
bc2 = t<<3 | t>>(64-3)
t = a[11] ^ d1
bc3 = t<<45 | t>>(64-45)
t = a[2] ^ d2
bc4 = t<<61 | t>>(64-61)
t = a[18] ^ d3
bc0 = t<<28 | t>>(64-28)
t = a[9] ^ d4
bc1 = t<<20 | t>>(64-20)
a[20] = bc0 ^ (bc2 &^ bc1)
a[11] = bc1 ^ (bc3 &^ bc2)
a[2] = bc2 ^ (bc4 &^ bc3)
a[18] = bc3 ^ (bc0 &^ bc4)
a[9] = bc4 ^ (bc1 &^ bc0)

t = a[15] ^ d0
bc4 = t<<18 | t>>(64-18)
t = a[6] ^ d1
bc0 = t<<1 | t>>(64-1)
t = a[22] ^ d2
bc1 = t<<6 | t>>(64-6)
t = a[13] ^ d3
bc2 = t<<25 | t>>(64-25)
t = a[4] ^ d4
bc3 = t<<8 | t>>(64-8)
a[15] = bc0 ^ (bc2 &^ bc1)
a[6] = bc1 ^ (bc3 &^ bc2)
a[22] = bc2 ^ (bc4 &^ bc3)
a[13] = bc3 ^ (bc0 &^ bc4)
a[4] = bc4 ^ (bc1 &^ bc0)

t = a[10] ^ d0
bc1 = t<<36 | t>>(64-36)
t = a[1] ^ d1
bc2 = t<<10 | t>>(64-10)
t = a[17] ^ d2
bc3 = t<<15 | t>>(64-15)
t = a[8] ^ d3
bc4 = t<<56 | t>>(64-56)
```

```
t = a[24] ^ d4
bc0 = t<<27 | t>>(64-27)
a[10] = bc0 ^ (bc2 &^ bc1)
a[1] = bc1 ^ (bc3 &^ bc2)
a[17] = bc2 ^ (bc4 &^ bc3)
a[8] = bc3 ^ (bc0 &^ bc4)
a[24] = bc4 ^ (bc1 &^ bc0)

t = a[5] ^ d0
bc3 = t<<41 | t>>(64-41)
t = a[21] ^ d1
bc4 = t<<2 | t>>(64-2)
t = a[12] ^ d2
bc0 = t<<62 | t>>(64-62)
t = a[3] ^ d3
bc1 = t<<55 | t>>(64-55)
t = a[19] ^ d4
bc2 = t<<39 | t>>(64-39)
a[5] = bc0 ^ (bc2 &^ bc1)
a[21] = bc1 ^ (bc3 &^ bc2)
a[12] = bc2 ^ (bc4 &^ bc3)
a[3] = bc3 ^ (bc0 &^ bc4)
a[19] = bc4 ^ (bc1 &^ bc0)

// Round 3
bc0 = a[0] ^ a[5] ^ a[10] ^ a[15] ^ a[20]
bc1 = a[1] ^ a[6] ^ a[11] ^ a[16] ^ a[21]
bc2 = a[2] ^ a[7] ^ a[12] ^ a[17] ^ a[22]
bc3 = a[3] ^ a[8] ^ a[13] ^ a[18] ^ a[23]
bc4 = a[4] ^ a[9] ^ a[14] ^ a[19] ^ a[24]
d0 = bc4 ^ (bc1<<1 | bc1>>63)
d1 = bc0 ^ (bc2<<1 | bc2>>63)
d2 = bc1 ^ (bc3<<1 | bc3>>63)
d3 = bc2 ^ (bc4<<1 | bc4>>63)
d4 = bc3 ^ (bc0<<1 | bc0>>63)

bc0 = a[0] ^ d0
t = a[11] ^ d1
bc1 = t<<44 | t>>(64-44)
t = a[22] ^ d2
bc2 = t<<43 | t>>(64-43)
t = a[8] ^ d3
```

```
bc3 = t<<21 | t>>(64-21)
t = a[19] ^ d4
bc4 = t<<14 | t>>(64-14)
a[0] = bc0 ^ (bc2 &^ bc1) ^ rc[i+2]
a[11] = bc1 ^ (bc3 &^ bc2)
a[22] = bc2 ^ (bc4 &^ bc3)
a[8] = bc3 ^ (bc0 &^ bc4)
a[19] = bc4 ^ (bc1 &^ bc0)

t = a[15] ^ d0
bc2 = t<<3 | t>>(64-3)
t = a[1] ^ d1
bc3 = t<<45 | t>>(64-45)
t = a[12] ^ d2
bc4 = t<<61 | t>>(64-61)
t = a[23] ^ d3
bc0 = t<<28 | t>>(64-28)
t = a[9] ^ d4
bc1 = t<<20 | t>>(64-20)
a[15] = bc0 ^ (bc2 &^ bc1)
a[1] = bc1 ^ (bc3 &^ bc2)
a[12] = bc2 ^ (bc4 &^ bc3)
a[23] = bc3 ^ (bc0 &^ bc4)
a[9] = bc4 ^ (bc1 &^ bc0)

t = a[5] ^ d0
bc4 = t<<18 | t>>(64-18)
t = a[16] ^ d1
bc0 = t<<1 | t>>(64-1)
t = a[2] ^ d2
bc1 = t<<6 | t>>(64-6)
t = a[13] ^ d3
bc2 = t<<25 | t>>(64-25)
t = a[24] ^ d4
bc3 = t<<8 | t>>(64-8)
a[5] = bc0 ^ (bc2 &^ bc1)
a[16] = bc1 ^ (bc3 &^ bc2)
a[2] = bc2 ^ (bc4 &^ bc3)
a[13] = bc3 ^ (bc0 &^ bc4)
a[24] = bc4 ^ (bc1 &^ bc0)

t = a[20] ^ d0
```

```
bc1 = t<<36 | t>>(64-36)
t = a[6] ^ d1
bc2 = t<<10 | t>>(64-10)
t = a[17] ^ d2
bc3 = t<<15 | t>>(64-15)
t = a[3] ^ d3
bc4 = t<<56 | t>>(64-56)
t = a[14] ^ d4
bc0 = t<<27 | t>>(64-27)
a[20] = bc0 ^ (bc2 &^ bc1)
a[6] = bc1 ^ (bc3 &^ bc2)
a[17] = bc2 ^ (bc4 &^ bc3)
a[3] = bc3 ^ (bc0 &^ bc4)
a[14] = bc4 ^ (bc1 &^ bc0)

t = a[10] ^ d0
bc3 = t<<41 | t>>(64-41)
t = a[21] ^ d1
bc4 = t<<2 | t>>(64-2)
t = a[7] ^ d2
bc0 = t<<62 | t>>(64-62)
t = a[18] ^ d3
bc1 = t<<55 | t>>(64-55)
t = a[4] ^ d4
bc2 = t<<39 | t>>(64-39)
a[10] = bc0 ^ (bc2 &^ bc1)
a[21] = bc1 ^ (bc3 &^ bc2)
a[7] = bc2 ^ (bc4 &^ bc3)
a[18] = bc3 ^ (bc0 &^ bc4)
a[4] = bc4 ^ (bc1 &^ bc0)

// Round 4
bc0 = a[0] ^ a[5] ^ a[10] ^ a[15] ^ a[20]
bc1 = a[1] ^ a[6] ^ a[11] ^ a[16] ^ a[21]
bc2 = a[2] ^ a[7] ^ a[12] ^ a[17] ^ a[22]
bc3 = a[3] ^ a[8] ^ a[13] ^ a[18] ^ a[23]
bc4 = a[4] ^ a[9] ^ a[14] ^ a[19] ^ a[24]
d0 = bc4 ^ (bc1<<1 | bc1>>63)
d1 = bc0 ^ (bc2<<1 | bc2>>63)
d2 = bc1 ^ (bc3<<1 | bc3>>63)
d3 = bc2 ^ (bc4<<1 | bc4>>63)
d4 = bc3 ^ (bc0<<1 | bc0>>63)
```

```
bc0 = a[0] ^ d0
t = a[1] ^ d1
bc1 = t<<44 | t>>(64-44)
t = a[2] ^ d2
bc2 = t<<43 | t>>(64-43)
t = a[3] ^ d3
bc3 = t<<21 | t>>(64-21)
t = a[4] ^ d4
bc4 = t<<14 | t>>(64-14)
a[0] = bc0 ^ (bc2 &^ bc1) ^ rc[i+3]
a[1] = bc1 ^ (bc3 &^ bc2)
a[2] = bc2 ^ (bc4 &^ bc3)
a[3] = bc3 ^ (bc0 &^ bc4)
a[4] = bc4 ^ (bc1 &^ bc0)

t = a[5] ^ d0
bc2 = t<<3 | t>>(64-3)
t = a[6] ^ d1
bc3 = t<<45 | t>>(64-45)
t = a[7] ^ d2
bc4 = t<<61 | t>>(64-61)
t = a[8] ^ d3
bc0 = t<<28 | t>>(64-28)
t = a[9] ^ d4
bc1 = t<<20 | t>>(64-20)
a[5] = bc0 ^ (bc2 &^ bc1)
a[6] = bc1 ^ (bc3 &^ bc2)
a[7] = bc2 ^ (bc4 &^ bc3)
a[8] = bc3 ^ (bc0 &^ bc4)
a[9] = bc4 ^ (bc1 &^ bc0)

t = a[10] ^ d0
bc4 = t<<18 | t>>(64-18)
t = a[11] ^ d1
bc0 = t<<1 | t>>(64-1)
t = a[12] ^ d2
bc1 = t<<6 | t>>(64-6)
t = a[13] ^ d3
bc2 = t<<25 | t>>(64-25)
t = a[14] ^ d4
bc3 = t<<8 | t>>(64-8)
```

```
a[10] = bc0 ^ (bc2 &^ bc1)
a[11] = bc1 ^ (bc3 &^ bc2)
a[12] = bc2 ^ (bc4 &^ bc3)
a[13] = bc3 ^ (bc0 &^ bc4)
a[14] = bc4 ^ (bc1 &^ bc0)

t = a[15] ^ d0
bc1 = t<<36 | t>>(64-36)
t = a[16] ^ d1
bc2 = t<<10 | t>>(64-10)
t = a[17] ^ d2
bc3 = t<<15 | t>>(64-15)
t = a[18] ^ d3
bc4 = t<<56 | t>>(64-56)
t = a[19] ^ d4
bc0 = t<<27 | t>>(64-27)
a[15] = bc0 ^ (bc2 &^ bc1)
a[16] = bc1 ^ (bc3 &^ bc2)
a[17] = bc2 ^ (bc4 &^ bc3)
a[18] = bc3 ^ (bc0 &^ bc4)
a[19] = bc4 ^ (bc1 &^ bc0)

t = a[20] ^ d0
bc3 = t<<41 | t>>(64-41)
t = a[21] ^ d1
bc4 = t<<2 | t>>(64-2)
t = a[22] ^ d2
bc0 = t<<62 | t>>(64-62)
t = a[23] ^ d3
bc1 = t<<55 | t>>(64-55)
t = a[24] ^ d4
bc2 = t<<39 | t>>(64-39)
a[20] = bc0 ^ (bc2 &^ bc1)
a[21] = bc1 ^ (bc3 &^ bc2)
a[22] = bc2 ^ (bc4 &^ bc3)
a[23] = bc3 ^ (bc0 &^ bc4)
a[24] = bc4 ^ (bc1 &^ bc0)
}
}
```

83:F:\git\coin\ethereum\go-ethereum\crypto\sha3\keccakf_amd64.go

84:F:\git\coin\ethereum\go-ethereum\crypto\sha3\register.go
crypto.RegisterHash(crypto.SHA3_256, New256)
crypto.RegisterHash(crypto.SHA3_384, New384)
crypto.RegisterHash(crypto.SHA3_512, New512)
}

85:F:\git\coin\ethereum\go-ethereum\crypto\sha3\sha3.go
)

const (
// maxRate is the maximum size of the internal buffer. SHAKE-256
// currently needs the largest buffer.
maxRate = 168
)

type state struct {
// Generic sponge components.
a   [25]uint64 // main state of the hash
buf []byte   // points into storage
rate int   // the number of bytes of state to use

// dsbyte contains the "domain separation" bits and the first bit of
// the padding. Sections 6.1 and 6.2 of [1] separate the outputs of the
// SHA-3 and SHAKE functions by appending bitstrings to the message.
// Using a little-endian bit-ordering convention, these are "01" for SHA-3
// and "1111" for SHAKE, or 00000010b and 00001111b, respectively. Then the
// padding rule from section 5.1 is applied to pad the message to a multiple
// of the rate, which involves adding a "1" bit, zero or more "0" bits, and
// a final "1" bit. We merge the first "1" bit from the padding into dsbyte,
// giving 00000110b (0x06) and 00011111b (0x1f).
// [1] http://csrc.nist.gov/publications/drafts/fips-202/fips_202_draft.pdf
//   "Draft FIPS 202: SHA-3 Standard: Permutation-Based Hash and
//   Extendable-Output Functions (May 2014)"
dsbyte byte
storage [maxRate]byte

// Specific to SHA-3 and SHAKE.
fixedOutput bool   // whether this is a fixed-output-length instance
outputLen   int   // the default output size in bytes
state   spongeDirection // whether the sponge is absorbing or squeezing
}

```
// BlockSize returns the rate of sponge underlying this hash function.
func (d *state) BlockSize() int { return d.rate }

// Size returns the output size of the hash function in bytes.
func (d *state) Size() int { return d.outputLen }

// Reset clears the internal state by zeroing the sponge state and
// the byte buffer, and setting Sponge.state to absorbing.
func (d *state) Reset() {
// Zero the permutation's state.
for i := range d.a {
d.a[i] = 0
}
d.state = spongeAbsorbing
d.buf = d.storage[:0]
}

func (d *state) clone() *state {
ret := *d
if ret.state == spongeAbsorbing {
ret.buf = ret.storage[:len(ret.buf)]
} else {
ret.buf = ret.storage[d.rate-cap(d.buf) : d.rate]
}

return &ret
}

// permute applies the KeccakF-1600 permutation. It handles
// any input-output buffering.
func (d *state) permute() {
switch d.state {
case spongeAbsorbing:
// If we're absorbing, we need to xor the input into the state
// before applying the permutation.
xorIn(d, d.buf)
d.buf = d.storage[:0]
keccakF1600(&d.a)
case spongeSqueezing:
// If we're squeezing, we need to apply the permutatin before
// copying more output.
keccakF1600(&d.a)
```

```
d.buf = d.storage[:d.rate]
copyOut(d, d.buf)
}
}

// pads appends the domain separation bits in dsbyte, applies
// the multi-bitrate 10..1 padding rule, and permutes the state.
func (d *state) padAndPermute(dsbyte byte) {
if d.buf == nil {
d.buf = d.storage[:0]
}
// Pad with this instance's domain-separator bits. We know that there's
// at least one byte of space in d.buf because, if it were full,
// permute would have been called to empty it. dsbyte also contains the
// first one bit for the padding. See the comment in the state struct.
d.buf = append(d.buf, dsbyte)
zerosStart := len(d.buf)
d.buf = d.storage[:d.rate]
for i := zerosStart; i < d.rate; i++ {
d.buf[i] = 0
}
// This adds the final one bit for the padding. Because of the way that
// bits are numbered from the LSB upwards, the final bit is the MSB of
// the last byte.
d.buf[d.rate-1] ^= 0x80
// Apply the permutation
d.permute()
d.state = spongeSqueezing
d.buf = d.storage[:d.rate]
copyOut(d, d.buf)
}

// Write absorbs more data into the hash's state. It produces an error
// if more data is written to the ShakeHash after writing
func (d *state) Write(p []byte) (written int, err error) {
if d.state != spongeAbsorbing {
panic("sha3: write to sponge after read")
}
if d.buf == nil {
d.buf = d.storage[:0]
}
written = len(p)
```

```
for len(p) > 0 {
if len(d.buf) == 0 && len(p) >= d.rate {
// The fast path; absorb a full "rate" bytes of input and apply the permutation.
xorIn(d, p[:d.rate])
p = p[d.rate:]
keccakF1600(&d.a)
} else {
// The slow path; buffer the input until we can fill the sponge, and then xor it in.
todo := d.rate - len(d.buf)
if todo > len(p) {
todo = len(p)
}
d.buf = append(d.buf, p[:todo]...)
p = p[todo:]

// If the sponge is full, apply the permutation.
if len(d.buf) == d.rate {
d.permute()
}
}
}

return
}

// Read squeezes an arbitrary number of bytes from the sponge.
func (d *state) Read(out []byte) (n int, err error) {
// If we're still absorbing, pad and apply the permutation.
if d.state == spongeAbsorbing {
d.padAndPermute(d.dsbyte)
}

n = len(out)

// Now, do the squeezing.
for len(out) > 0 {
n := copy(out, d.buf)
d.buf = d.buf[n:]
out = out[n:]

// Apply the permutation if we've squeezed the sponge dry.
```

```
if len(d.buf) == 0 {
d.permute()
}
}

return
}

// Sum applies padding to the hash state and then squeezes out the desired
// number of output bytes.
func (d *state) Sum(in []byte) []byte {
// Make a copy of the original hash so that caller can keep writing
// and summing.
dup := d.clone()
hash := make([]byte, dup.outputLen)
dup.Read(hash)
return append(in, hash...)
}

86:F:\git\coin\ethereum\go-ethereum\crypto\sha3\sha3_test.go
"compress/flate"
"encoding/hex"
"encoding/json"
"hash"
"os"
"strings"
"testing"
)

const (
testString  = "brekeccakkeccak koax koax"
katFilename = "testdata/keccakKats.json.deflate"
)

// Internal-use instances of SHAKE used to test against KATs.
func newHashShake128() hash.Hash {
return &state{rate: 168, dsbyte: 0x1f, outputLen: 512}
}
func newHashShake256() hash.Hash {
return &state{rate: 136, dsbyte: 0x1f, outputLen: 512}
}
```

```
// testDigests contains functions returning hash.Hash instances
// with output-length equal to the KAT length for both SHA-3 and
// SHAKE instances.
var testDigests = map[string]func() hash.Hash{
"SHA3-224": New224,
"SHA3-256": New256,
"SHA3-384": New384,
"SHA3-512": New512,
"SHAKE128": newHashShake128,
"SHAKE256": newHashShake256,
}

// testShakes contains functions that return ShakeHash instances for
// testing the ShakeHash-specific interface.
var testShakes = map[string]func() ShakeHash{
"SHAKE128": NewShake128,
"SHAKE256": NewShake256,
}

// decodeHex converts a hex-encoded string into a raw byte string.
func decodeHex(s string) []byte {
b, err := hex.DecodeString(s)
if err != nil {
panic(err)
}
return b
}

// structs used to marshal JSON test-cases.
type KeccakKats struct {
Kats map[string][]struct {
Digest  string `json:"digest"`
Length  int64  `json:"length"`
Message string `json:"message"`
}
}

func testUnalignedAndGeneric(t *testing.T, testf func(impl string)) {
xorInOrig, copyOutOrig := xorIn, copyOut
xorIn, copyOut = xorInGeneric, copyOutGeneric
testf("generic")
if xorImplementationUnaligned != "generic" {
```

```
xorIn, copyOut = xorInUnaligned, copyOutUnaligned
testf("unaligned")
}
xorIn, copyOut = xorInOrig, copyOutOrig
}

// TestKeccakKats tests the SHA-3 and Shake implementations against all the
// ShortMsgKATs from https://github.com/gvanas/KeccakCodePackage
// (The testvectors are stored in keccakKats.json.deflate due to their length.)
func TestKeccakKats(t *testing.T) {
testUnalignedAndGeneric(t, func(impl string) {
// Read the KATs.
deflated, err := os.Open(katFilename)
if err != nil {
t.Errorf("error opening %s: %s", katFilename, err)
}
file := flate.NewReader(deflated)
dec := json.NewDecoder(file)
var katSet KeccakKats
err = dec.Decode(&katSet)
if err != nil {
t.Errorf("error decoding KATs: %s", err)
}

// Do the KATs.
for functionName, kats := range katSet.Kats {
d := testDigests[functionName]()
for _, kat := range kats {
d.Reset()
in, err := hex.DecodeString(kat.Message)
if err != nil {
t.Errorf("error decoding KAT: %s", err)
}
d.Write(in[:kat.Length/8])
got := strings.ToUpper(hex.EncodeToString(d.Sum(nil)))
if got != kat.Digest {
t.Errorf("function=%s, implementation=%s, length=%d\nmessage:\n  %s\ngot:\n  %s\nwanted:\n
%s",
functionName, impl, kat.Length, kat.Message, got, kat.Digest)
t.Logf("wanted %+v", kat)
t.FailNow()
}
```

```
continue
}
}
})
}

// TestUnalignedWrite tests that writing data in an arbitrary pattern with
// small input buffers.
func testUnalignedWrite(t *testing.T) {
testUnalignedAndGeneric(t, func(impl string) {
buf := sequentialBytes(0x10000)
for alg, df := range testDigests {
d := df()
d.Reset()
d.Write(buf)
want := d.Sum(nil)
d.Reset()
for i := 0; i < len(buf); {
// Cycle through offsets which make a 137 byte sequence.
// Because 137 is prime this sequence should exercise all corner cases.
offsets := [17]int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 1}
for _, j := range offsets {
if v := len(buf) - i; v < j {
j = v
}
d.Write(buf[i : i+j])
i += j
}
}
got := d.Sum(nil)
if !bytes.Equal(got, want) {
t.Errorf("Unaligned writes, implementation=%s, alg=%s\ngot %q, want %q", impl, alg, got, want)
}
}
})
}

// TestAppend checks that appending works when reallocation is necessary.
func TestAppend(t *testing.T) {
testUnalignedAndGeneric(t, func(impl string) {
d := New224()
```

```
for capacity := 2; capacity <= 66; capacity += 64 {
// The first time around the loop, Sum will have to reallocate.
// The second time, it will not.
buf := make([]byte, 2, capacity)
d.Reset()
d.Write([]byte{0xcc})
buf = d.Sum(buf)
expected := "0000DF70ADC49B2E76EEE3A6931B93FA41841C3AF2CDF5B32A18B5478C39"
if got := strings.ToUpper(hex.EncodeToString(buf)); got != expected {
t.Errorf("got %s, want %s", got, expected)
}
}
})
}

// TestAppendNoRealloc tests that appending works when no reallocation is necessary.
func TestAppendNoRealloc(t *testing.T) {
testUnalignedAndGeneric(t, func(impl string) {
buf := make([]byte, 1, 200)
d := New224()
d.Write([]byte{0xcc})
buf = d.Sum(buf)
expected := "00DF70ADC49B2E76EEE3A6931B93FA41841C3AF2CDF5B32A18B5478C39"
if got := strings.ToUpper(hex.EncodeToString(buf)); got != expected {
t.Errorf("%s: got %s, want %s", impl, got, expected)
}
})
}

// TestSqueezing checks that squeezing the full output a single time produces
// the same output as repeatedly squeezing the instance.
func TestSqueezing(t *testing.T) {
testUnalignedAndGeneric(t, func(impl string) {
for functionName, newShakeHash := range testShakes {
d0 := newShakeHash()
d0.Write([]byte(testString))
ref := make([]byte, 32)
d0.Read(ref)

d1 := newShakeHash()
d1.Write([]byte(testString))
var multiple []byte
```

```
for range ref {
one := make([]byte, 1)
d1.Read(one)
multiple = append(multiple, one...)
}
if !bytes.Equal(ref, multiple) {
t.Errorf("%s (%s): squeezing %d bytes one at a time failed", functionName, impl, len(ref))
}
}
})
}

// sequentialBytes produces a buffer of size consecutive bytes 0x00, 0x01, ..., used for testing.
func sequentialBytes(size int) []byte {
result := make([]byte, size)
for i := range result {
result[i] = byte(i)
}
return result
}

// BenchmarkPermutationFunction measures the speed of the permutation function
// with no input data.
func BenchmarkPermutationFunction(b *testing.B) {
b.SetBytes(int64(200))
var lanes [25]uint64
for i := 0; i < b.N; i++ {
keccakF1600(&lanes)
}
}

// benchmarkHash tests the speed to hash num buffers of buflen each.
func benchmarkHash(b *testing.B, h hash.Hash, size, num int) {
b.StopTimer()
h.Reset()
data := sequentialBytes(size)
b.SetBytes(int64(size * num))
b.StartTimer()

var state []byte
for i := 0; i < b.N; i++ {
for j := 0; j < num; j++ {
```

```
h.Write(data)
}
state = h.Sum(state[:0])
}
b.StopTimer()
h.Reset()
}

// benchmarkShake is specialized to the Shake instances, which don't
// require a copy on reading output.
func benchmarkShake(b *testing.B, h ShakeHash, size, num int) {
b.StopTimer()
h.Reset()
data := sequentialBytes(size)
d := make([]byte, 32)

b.SetBytes(int64(size * num))
b.StartTimer()

for i := 0; i < b.N; i++ {
h.Reset()
for j := 0; j < num; j++ {
h.Write(data)
}
h.Read(d)
}
}

func BenchmarkSha3_512_MTU(b *testing.B) { benchmarkHash(b, New512(), 1350, 1) }
func BenchmarkSha3_384_MTU(b *testing.B) { benchmarkHash(b, New384(), 1350, 1) }
func BenchmarkSha3_256_MTU(b *testing.B) { benchmarkHash(b, New256(), 1350, 1) }
func BenchmarkSha3_224_MTU(b *testing.B) { benchmarkHash(b, New224(), 1350, 1) }

func BenchmarkShake128_MTU(b *testing.B)  { benchmarkShake(b, NewShake128(), 1350, 1) }
func BenchmarkShake256_MTU(b *testing.B)  { benchmarkShake(b, NewShake256(), 1350, 1) }
func BenchmarkShake256_16x(b *testing.B)  { benchmarkShake(b, NewShake256(), 16, 1024) }
func BenchmarkShake256_1MiB(b *testing.B) { benchmarkShake(b, NewShake256(), 1024, 1024)
}

func BenchmarkSha3_512_1MiB(b *testing.B) { benchmarkHash(b, New512(), 1024, 1024) }

func Example_sum() {
```

```
buf := []byte("some data to hash")
// A hash needs to be 64 bytes long to have 256-bit collision resistance.
h := make([]byte, 64)
// Compute a 64-byte hash of buf and put it in h.
ShakeSum256(h, buf)
}

func Example_mac() {
k := []byte("this is a secret key; you should generate a strong random key that's at least 32 bytes long")
buf := []byte("and this is some data to authenticate")
// A MAC with 32 bytes of output has 256-bit security strength -- if you use at least a 32-byte-long key.
h := make([]byte, 32)
d := NewShake256()
// Write the key into the hash.
d.Write(k)
// Now write the data.
d.Write(buf)
// Read 32 bytes of output from the hash into h.
d.Read(h)
}

87:F:\git\coin\ethereum\go-ethereum\crypto\sha3\shake.go
// ShakeHash defines the interface to hash functions that
// support arbitrary-length output.
type ShakeHash interface {
// Write absorbs more data into the hash's state. It panics if input is
// written to it after output has been read from it.
io.Writer

// Read reads more output from the hash; reading affects the hash's
// state. (ShakeHash.Read is thus very different from Hash.Sum)
// It never returns an error.
io.Reader

// Clone returns a copy of the ShakeHash in its current state.
Clone() ShakeHash

// Reset resets the ShakeHash to its initial state.
Reset()
}
```

```
func (d *state) Clone() ShakeHash {
return d.clone()
}

// NewShake128 creates a new SHAKE128 variable-output-length ShakeHash.
// Its generic security strength is 128 bits against all attacks if at
// least 32 bytes of its output are used.
func NewShake128() ShakeHash { return &state{rate: 168, dsbyte: 0x1f} }

// NewShake256 creates a new SHAKE128 variable-output-length ShakeHash.
// Its generic security strength is 256 bits against all attacks if
// at least 64 bytes of its output are used.
func NewShake256() ShakeHash { return &state{rate: 136, dsbyte: 0x1f} }

// ShakeSum128 writes an arbitrary-length digest of data into hash.
func ShakeSum128(hash, data []byte) {
h := NewShake128()
h.Write(data)
h.Read(hash)
}

// ShakeSum256 writes an arbitrary-length digest of data into hash.
func ShakeSum256(hash, data []byte) {
h := NewShake256()
h.Write(data)
h.Read(hash)
}

88:F:\git\coin\ethereum\go-ethereum\crypto\sha3\xor.go

const xorImplementationUnaligned = "generic"

89:F:\git\coin\ethereum\go-ethereum\crypto\sha3\xor_generic.go
for i := 0; i < n; i++ {
a := binary.LittleEndian.Uint64(buf)
d.a[i] ^= a
buf = buf[8:]
}
}

// copyOutGeneric copies ulint64s to a byte buffer.
```

```
func copyOutGeneric(d *state, b []byte) {
for i := 0; len(b) >= 8; i++ {
binary.LittleEndian.PutUint64(b, d.a[i])
b = b[8:]
}
}
```

90:F:\git\coin\ethereum\go-ethereum\crypto\sha3\xor_unaligned.go

```
if n >= 72 {
d.a[0] ^= bw[0]
d.a[1] ^= bw[1]
d.a[2] ^= bw[2]
d.a[3] ^= bw[3]
d.a[4] ^= bw[4]
d.a[5] ^= bw[5]
d.a[6] ^= bw[6]
d.a[7] ^= bw[7]
d.a[8] ^= bw[8]
}
if n >= 104 {
d.a[9] ^= bw[9]
d.a[10] ^= bw[10]
d.a[11] ^= bw[11]
d.a[12] ^= bw[12]
}
if n >= 136 {
d.a[13] ^= bw[13]
d.a[14] ^= bw[14]
d.a[15] ^= bw[15]
d.a[16] ^= bw[16]
}
if n >= 144 {
d.a[17] ^= bw[17]
}
if n >= 168 {
d.a[18] ^= bw[18]
d.a[19] ^= bw[19]
d.a[20] ^= bw[20]
}
}

func copyOutUnaligned(d *state, buf []byte) {
```

```
ab := (*[maxRate]uint8)(unsafe.Pointer(&d.a[0]))
copy(buf, ab[:])
}

var (
xorIn   = xorInUnaligned
copyOut = copyOutUnaligned
)

const xorImplementationUnaligned = "unaligned"

91:F:\git\coin\ethereum\go-ethereum\crypto\signature_cgo.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

// +build !nacl,!js,!nocgo

package crypto

import (
"crypto/ecdsa"
"crypto/elliptic"
"fmt"

"github.com/ethereum/go-ethereum/common/math"
"github.com/ethereum/go-ethereum/crypto/secp256k1"
)

func Ecrecover(hash, sig []byte) ([]byte, error) {
return secp256k1.RecoverPubkey(hash, sig)
}

func SigToPub(hash, sig []byte) (*ecdsa.PublicKey, error) {
s, err := Ecrecover(hash, sig)
if err != nil {
return nil, err
}

x, y := elliptic.Unmarshal(S256(), s)
return &ecdsa.PublicKey{Curve: S256(), X: x, Y: y}, nil
}

// Sign calculates an ECDSA signature.
```

```
//
// This function is susceptible to chosen plaintext attacks that can leak
// information about the private key that is used for signing. Callers must
// be aware that the given hash cannot be chosen by an adversery. Common
// solution is to hash any input before calculating the signature.
//
// The produced signature is in the [R || S || V] format where V is 0 or 1.
func Sign(hash []byte, prv *ecdsa.PrivateKey) (sig []byte, err error) {
if len(hash) != 32 {
return nil, fmt.Errorf("hash is required to be exactly 32 bytes (%d)", len(hash))
}
seckey := math.PaddedBigBytes(prv.D, prv.Params().BitSize/8)
defer zeroBytes(seckey)
return secp256k1.Sign(hash, seckey)
}

// S256 returns an instance of the secp256k1 curve.
func S256() elliptic.Curve {
return secp256k1.S256()
}

92:F:\git\coin\ethereum\go-ethereum\crypto\signature_nocgo.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

// +build nacl js nocgo

package crypto

import (
"crypto/ecdsa"
"crypto/elliptic"
"fmt"

"github.com/btcsuite/btcd/btcec"
)

func Ecrecover(hash, sig []byte) ([]byte, error) {
pub, err := SigToPub(hash, sig)
if err != nil {
return nil, err
}
bytes := (*btcec.PublicKey)(pub).SerializeUncompressed()
```

```
return bytes, err
}

func SigToPub(hash, sig []byte) (*ecdsa.PublicKey, error) {
// Convert to btcec input format with 'recovery id' v at the beginning.
btcsig := make([]byte, 65)
btcsig[0] = sig[64] + 27
copy(btcsig[1:], sig)

pub, _, err := btcec.RecoverCompact(btcec.S256(), btcsig, hash)
return (*ecdsa.PublicKey)(pub), err
}

// Sign calculates an ECDSA signature.
//
// This function is susceptible to chosen plaintext attacks that can leak
// information about the private key that is used for signing. Callers must
// be aware that the given hash cannot be chosen by an adversery. Common
// solution is to hash any input before calculating the signature.
//
// The produced signature is in the [R || S || V] format where V is 0 or 1.
func Sign(hash []byte, prv *ecdsa.PrivateKey) ([]byte, error) {
if len(hash) != 32 {
return nil, fmt.Errorf("hash is required to be exactly 32 bytes (%d)", len(hash))
}
if prv.Curve != btcec.S256() {
return nil, fmt.Errorf("private key curve is not secp256k1")
}
sig, err := btcec.SignCompact(btcec.S256(), (*btcec.PrivateKey)(prv), hash, false)
if err != nil {
return nil, err
}
// Convert to Ethereum signature format with 'recovery id' v at the end.
v := sig[0] - 27
copy(sig, sig[1:])
sig[64] = v
return sig, nil
}

// S256 returns an instance of the secp256k1 curve.
func S256() elliptic.Curve {
return btcec.S256()
```

```
}

93:F:\git\coin\ethereum\go-ethereum\crypto\signature_test.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package crypto

import (
"bytes"
"encoding/hex"
"testing"
)

func TestRecoverSanity(t *testing.T) {
msg, _ :=
hex.DecodeString("ce0677bb30baa8cf067c88db9811f4333d131bf8bcf12fe7065d211dce971008")
sig, _ :=
hex.DecodeString("90f27b8b488db00b00606796d2987f6a5f59ae62ea05effe84fef5b8b0e549984a
691139ad57a3f0b906637673aa2f63d1f55cb1a69199d4009eea23ceaddc9301")
pubkey1, _ :=
hex.DecodeString("04e32df42865e97135acfb65f3bae71bdc86f4d49150ad6a440b6f15878109880
a0a2b2667f7e725ceea70c673093bf67663e0312623c8e091b13cf2c0f11ef652")
pubkey2, err := Ecrecover(msg, sig)
if err != nil {
t.Fatalf("recover error: %s", err)
}
if !bytes.Equal(pubkey1, pubkey2) {
t.Errorf("pubkey mismatch: want: %x have: %x", pubkey1, pubkey2)
}
}

94:F:\git\coin\ethereum\go-ethereum\eth\api.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package eth

import (
"bytes"
"compress/gzip"
"context"
"fmt"
"io"
```

```
"io/ioutil"
"math/big"
"os"
"strings"
"time"

"github.com/ethereum/go-ethereum/common"
"github.com/ethereum/go-ethereum/common/hexutil"
"github.com/ethereum/go-ethereum/core"
"github.com/ethereum/go-ethereum/core/state"
"github.com/ethereum/go-ethereum/core/types"
"github.com/ethereum/go-ethereum/core/vm"
"github.com/ethereum/go-ethereum/internal/ethapi"
"github.com/ethereum/go-ethereum/log"
"github.com/ethereum/go-ethereum/miner"
"github.com/ethereum/go-ethereum/params"
"github.com/ethereum/go-ethereum/rlp"
"github.com/ethereum/go-ethereum/rpc"
"github.com/ethereum/go-ethereum/trie"
)

const defaultTraceTimeout = 5 * time.Second

// PublicEthereumAPI provides an API to access Ethereum full node-related
// information.
type PublicEthereumAPI struct {
e *Ethereum
}

// NewPublicEthereumAPI creates a new Etheruem protocol API for full nodes.
func NewPublicEthereumAPI(e *Ethereum) *PublicEthereumAPI {
return &PublicEthereumAPI{e}
}

// Etherbase is the address that mining rewards will be send to
func (api *PublicEthereumAPI) Etherbase() (common.Address, error) {
return api.e.Etherbase()
}

// Coinbase is the address that mining rewards will be send to (alias for Etherbase)
func (api *PublicEthereumAPI) Coinbase() (common.Address, error) {
return api.Etherbase()
```

```
}

// Hashrate returns the POW hashrate
func (api *PublicEthereumAPI) Hashrate() hexutil.Uint64 {
return hexutil.Uint64(api.e.Miner().HashRate())
}

// PublicMinerAPI provides an API to control the miner.
// It offers only methods that operate on data that pose no security risk when it is publicly accessible.
type PublicMinerAPI struct {
e     *Ethereum
agent *miner.RemoteAgent
}

// NewPublicMinerAPI create a new PublicMinerAPI instance.
func NewPublicMinerAPI(e *Ethereum) *PublicMinerAPI {
agent := miner.NewRemoteAgent(e.BlockChain(), e.Engine())
e.Miner().Register(agent)

return &PublicMinerAPI{e, agent}
}

// Mining returns an indication if this node is currently mining.
func (api *PublicMinerAPI) Mining() bool {
return api.e.IsMining()
}

// SubmitWork can be used by external miner to submit their POW solution. It returns an indication if the work was
// accepted. Note, this is not an indication if the provided work was valid!
func (api *PublicMinerAPI) SubmitWork(nonce types.BlockNonce, solution, digest common.Hash) bool {
return api.agent.SubmitWork(nonce, digest, solution)
}

// GetWork returns a work package for external miner. The work package consists of 3 strings
// result[0], 32 bytes hex encoded current block header pow-hash
// result[1], 32 bytes hex encoded seed hash used for DAG
// result[2], 32 bytes hex encoded boundary condition ("target"), 2^256/difficulty
func (api *PublicMinerAPI) GetWork() ([3]string, error) {
if !api.e.IsMining() {
```

```
if err := api.e.StartMining(false); err != nil {
return [3]string{}, err
}
}
work, err := api.agent.GetWork()
if err != nil {
return work, fmt.Errorf("mining not ready: %v", err)
}
return work, nil
}

// SubmitHashrate can be used for remote miners to submit their hash rate. This enables the node to report the combined
// hash rate of all miners which submit work through this node. It accepts the miner hash rate and an identifier which
// must be unique between nodes.
func (api *PublicMinerAPI) SubmitHashrate(hashrate hexutil.Uint64, id common.Hash) bool {
api.agent.SubmitHashrate(id, uint64(hashrate))
return true
}

// PrivateMinerAPI provides private RPC methods to control the miner.
// These methods can be abused by external users and must be considered insecure for use by untrusted users.
type PrivateMinerAPI struct {
e *Ethereum
}

// NewPrivateMinerAPI create a new RPC service which controls the miner of this node.
func NewPrivateMinerAPI(e *Ethereum) *PrivateMinerAPI {
return &PrivateMinerAPI{e: e}
}

// Start the miner with the given number of threads. If threads is nil the number
// of workers started is equal to the number of logical CPUs that are usable by
// this process. If mining is already running, this method adjust the number of
// threads allowed to use.
func (api *PrivateMinerAPI) Start(threads *int) error {
// Set the number of threads if the seal engine supports it
if threads == nil {
threads = new(int)
} else if *threads == 0 {
```

```
*threads = -1 // Disable the miner from within
}
type threaded interface {
SetThreads(threads int)
}
if th, ok := api.e.engine.(threaded); ok {
log.Info("Updated mining threads", "threads", *threads)
th.SetThreads(*threads)
}
// Start the miner and return
if !api.e.IsMining() {
// Propagate the initial price point to the transaction pool
api.e.lock.RLock()
price := api.e.gasPrice
api.e.lock.RUnlock()

api.e.txPool.SetGasPrice(price)
return api.e.StartMining(true)
}
return nil
}

// Stop the miner
func (api *PrivateMinerAPI) Stop() bool {
type threaded interface {
SetThreads(threads int)
}
if th, ok := api.e.engine.(threaded); ok {
th.SetThreads(-1)
}
api.e.StopMining()
return true
}

// SetExtra sets the extra data string that is included when this miner mines a block.
func (api *PrivateMinerAPI) SetExtra(extra string) (bool, error) {
if err := api.e.Miner().SetExtra([]byte(extra)); err != nil {
return false, err
}
return true, nil
}
```

```
// SetGasPrice sets the minimum accepted gas price for the miner.
func (api *PrivateMinerAPI) SetGasPrice(gasPrice hexutil.Big) bool {
api.e.lock.Lock()
api.e.gasPrice = (*big.Int)(&gasPrice)
api.e.lock.Unlock()

api.e.txPool.SetGasPrice((*big.Int)(&gasPrice))
return true
}

// SetEtherbase sets the etherbase of the miner
func (api *PrivateMinerAPI) SetEtherbase(etherbase common.Address) bool {
api.e.SetEtherbase(etherbase)
return true
}

// GetHashrate returns the current hashrate of the miner.
func (api *PrivateMinerAPI) GetHashrate() uint64 {
return uint64(api.e.miner.HashRate())
}

// PrivateAdminAPI is the collection of Etheruem full node-related APIs
// exposed over the private admin endpoint.
type PrivateAdminAPI struct {
eth *Ethereum
}

// NewPrivateAdminAPI creates a new API definition for the full node private
// admin methods of the Ethereum service.
func NewPrivateAdminAPI(eth *Ethereum) *PrivateAdminAPI {
return &PrivateAdminAPI{eth: eth}
}

// ExportChain exports the current blockchain into a local file.
func (api *PrivateAdminAPI) ExportChain(file string) (bool, error) {
// Make sure we can create the file to export into
out, err := os.OpenFile(file, os.O_CREATE|os.O_WRONLY|os.O_TRUNC, os.ModePerm)
if err != nil {
return false, err
}
defer out.Close()
```

```
var writer io.Writer = out
if strings.HasSuffix(file, ".gz") {
writer = gzip.NewWriter(writer)
defer writer.(*gzip.Writer).Close()
}

// Export the blockchain
if err := api.eth.BlockChain().Export(writer); err != nil {
return false, err
}
return true, nil
}

func hasAllBlocks(chain *core.BlockChain, bs []*types.Block) bool {
for _, b := range bs {
if !chain.HasBlock(b.Hash()) {
return false
}
}

return true
}

// ImportChain imports a blockchain from a local file.
func (api *PrivateAdminAPI) ImportChain(file string) (bool, error) {
// Make sure the can access the file to import
in, err := os.Open(file)
if err != nil {
return false, err
}
defer in.Close()

var reader io.Reader = in
if strings.HasSuffix(file, ".gz") {
if reader, err = gzip.NewReader(reader); err != nil {
return false, err
}
}

// Run actual the import in pre-configured batches
stream := rlp.NewStream(reader, 0)
```

```
blocks, index := make([]*types.Block, 0, 2500), 0
for batch := 0; ; batch++ {
// Load a batch of blocks from the input file
for len(blocks) < cap(blocks) {
block := new(types.Block)
if err := stream.Decode(block); err == io.EOF {
break
} else if err != nil {
return false, fmt.Errorf("block %d: failed to parse: %v", index, err)
}
blocks = append(blocks, block)
index++
}
if len(blocks) == 0 {
break
}

if hasAllBlocks(api.eth.BlockChain(), blocks) {
blocks = blocks[:0]
continue
}
// Import the batch and reset the buffer
if _, err := api.eth.BlockChain().InsertChain(blocks); err != nil {
return false, fmt.Errorf("batch %d: failed to insert: %v", batch, err)
}
blocks = blocks[:0]
}
return true, nil
}

// PublicDebugAPI is the collection of Etheruem full node APIs exposed
// over the public debugging endpoint.
type PublicDebugAPI struct {
eth *Ethereum
}

// NewPublicDebugAPI creates a new API definition for the full node-
// related public debug methods of the Ethereum service.
func NewPublicDebugAPI(eth *Ethereum) *PublicDebugAPI {
return &PublicDebugAPI{eth: eth}
}
```

```
// DumpBlock retrieves the entire state of the database at a given block.
func (api *PublicDebugAPI) DumpBlock(blockNr rpc.BlockNumber) (state.Dump, error) {
if blockNr == rpc.PendingBlockNumber {
// If we're dumping the pending state, we need to request
// both the pending block as well as the pending state from
// the miner and operate on those
_, stateDb := api.eth.miner.Pending()
return stateDb.RawDump(), nil
}
var block *types.Block
if blockNr == rpc.LatestBlockNumber {
block = api.eth.blockchain.CurrentBlock()
} else {
block = api.eth.blockchain.GetBlockByNumber(uint64(blockNr))
}
if block == nil {
return state.Dump{}, fmt.Errorf("block #%d not found", blockNr)
}
stateDb, err := api.eth.BlockChain().StateAt(block.Root())
if err != nil {
return state.Dump{}, err
}
return stateDb.RawDump(), nil
}

// PrivateDebugAPI is the collection of Etheruem full node APIs exposed over
// the private debugging endpoint.
type PrivateDebugAPI struct {
config *params.ChainConfig
eth    *Ethereum
}

// NewPrivateDebugAPI creates a new API definition for the full node-related
// private debug methods of the Ethereum service.
func NewPrivateDebugAPI(config *params.ChainConfig, eth *Ethereum) *PrivateDebugAPI {
return &PrivateDebugAPI{config: config, eth: eth}
}

// BlockTraceResult is the returned value when replaying a block to check for
// consensus results and full VM trace logs for all included transactions.
type BlockTraceResult struct {
Validated  bool                `json:"validated"`
```

```
StructLogs []ethapi.StructLogRes `json:"structLogs"`
Error      string               `json:"error"`
}

// TraceArgs holds extra parameters to trace functions
type TraceArgs struct {
*vm.LogConfig
Tracer  *string
Timeout *string
}

// TraceBlock processes the given block'api RLP but does not import the block in to
// the chain.
func (api *PrivateDebugAPI) TraceBlock(blockRlp []byte, config *vm.LogConfig) BlockTraceResult
{
var block types.Block
err := rlp.Decode(bytes.NewReader(blockRlp), &block)
if err != nil {
return BlockTraceResult{Error: fmt.Sprintf("could not decode block: %v", err)}
}

validated, logs, err := api.traceBlock(&block, config)
return BlockTraceResult{
Validated:  validated,
StructLogs: ethapi.FormatLogs(logs),
Error:      formatError(err),
}
}

// TraceBlockFromFile loads the block'api RLP from the given file name and attempts to
// process it but does not import the block in to the chain.
func (api *PrivateDebugAPI) TraceBlockFromFile(file string, config *vm.LogConfig)
BlockTraceResult {
blockRlp, err := ioutil.ReadFile(file)
if err != nil {
return BlockTraceResult{Error: fmt.Sprintf("could not read file: %v", err)}
}
return api.TraceBlock(blockRlp, config)
}

// TraceBlockByNumber processes the block by canonical block number.
func (api *PrivateDebugAPI) TraceBlockByNumber(blockNr rpc.BlockNumber, config
```

```
*vm.LogConfig) BlockTraceResult {
// Fetch the block that we aim to reprocess
var block *types.Block
switch blockNr {
case rpc.PendingBlockNumber:
// Pending block is only known by the miner
block = api.eth.miner.PendingBlock()
case rpc.LatestBlockNumber:
block = api.eth.blockchain.CurrentBlock()
default:
block = api.eth.blockchain.GetBlockByNumber(uint64(blockNr))
}

if block == nil {
return BlockTraceResult{Error: fmt.Sprintf("block #%d not found", blockNr)}
}

validated, logs, err := api.traceBlock(block, config)
return BlockTraceResult{
Validated:  validated,
StructLogs: ethapi.FormatLogs(logs),
Error:      formatError(err),
}
}

// TraceBlockByHash processes the block by hash.
func (api *PrivateDebugAPI) TraceBlockByHash(hash common.Hash, config *vm.LogConfig)
BlockTraceResult {
// Fetch the block that we aim to reprocess
block := api.eth.BlockChain().GetBlockByHash(hash)
if block == nil {
return BlockTraceResult{Error: fmt.Sprintf("block #%x not found", hash)}
}

validated, logs, err := api.traceBlock(block, config)
return BlockTraceResult{
Validated:  validated,
StructLogs: ethapi.FormatLogs(logs),
Error:      formatError(err),
}
}
```

```
// traceBlock processes the given block but does not save the state.
func (api *PrivateDebugAPI) traceBlock(block *types.Block, logConfig *vm.LogConfig) (bool,
[]vm.StructLog, error) {
// Validate and reprocess the block
var (
blockchain = api.eth.BlockChain()
validator  = blockchain.Validator()
processor  = blockchain.Processor()
)

structLogger := vm.NewStructLogger(logConfig)

config := vm.Config{
Debug:  true,
Tracer: structLogger,
}
if err := api.eth.engine.VerifyHeader(blockchain, block.Header(), true); err != nil {
return false, structLogger.StructLogs(), err
}
statedb, err := blockchain.StateAt(blockchain.GetBlock(block.ParentHash(), block.NumberU64()-
1).Root())
if err != nil {
return false, structLogger.StructLogs(), err
}

receipts, _, usedGas, err := processor.Process(block, statedb, config)
if err != nil {
return false, structLogger.StructLogs(), err
}
if err := validator.ValidateState(block, blockchain.GetBlock(block.ParentHash(),
block.NumberU64()-1), statedb, receipts, usedGas); err != nil {
return false, structLogger.StructLogs(), err
}
return true, structLogger.StructLogs(), nil
}

// callmsg is the message type used for call transitions.
type callmsg struct {
addr          common.Address
to            *common.Address
gas, gasPrice *big.Int
value         *big.Int
```

```
data          []byte
}

// accessor boilerplate to implement core.Message
func (m callmsg) From() (common.Address, error)         { return m.addr, nil }
func (m callmsg) FromFrontier() (common.Address, error) { return m.addr, nil }
func (m callmsg) Nonce() uint64                         { return 0 }
func (m callmsg) CheckNonce() bool                      { return false }
func (m callmsg) To() *common.Address                   { return m.to }
func (m callmsg) GasPrice() *big.Int                    { return m.gasPrice }
func (m callmsg) Gas() *big.Int                         { return m.gas }
func (m callmsg) Value() *big.Int                       { return m.value }
func (m callmsg) Data() []byte                          { return m.data }

// formatError formats a Go error into either an empty string or the data content
// of the error itself.
func formatError(err error) string {
if err == nil {
return ""
}
return err.Error()
}

type timeoutError struct{}

func (t *timeoutError) Error() string {
return "Execution time exceeded"
}

// TraceTransaction returns the structured logs created during the execution of EVM
// and returns them as a JSON object.
func (api *PrivateDebugAPI) TraceTransaction(ctx context.Context, txHash common.Hash, config
*TraceArgs) (interface{}, error) {
var tracer vm.Tracer
if config != nil && config.Tracer != nil {
timeout := defaultTraceTimeout
if config.Timeout != nil {
var err error
if timeout, err = time.ParseDuration(*config.Timeout); err != nil {
return nil, err
}
}
```

```
var err error
if tracer, err = ethapi.NewJavascriptTracer(*config.Tracer); err != nil {
return nil, err
}

// Handle timeouts and RPC cancellations
deadlineCtx, cancel := context.WithTimeout(ctx, timeout)
go func() {
<-deadlineCtx.Done()
tracer.(*ethapi.JavascriptTracer).Stop(&timeoutError{})
}()
defer cancel()
} else if config == nil {
tracer = vm.NewStructLogger(nil)
} else {
tracer = vm.NewStructLogger(config.LogConfig)
}

// Retrieve the tx from the chain and the containing block
tx, blockHash, _, txIndex := core.GetTransaction(api.eth.ChainDb(), txHash)
if tx == nil {
return nil, fmt.Errorf("transaction %x not found", txHash)
}
msg, context, statedb, err := api.computeTxEnv(blockHash, int(txIndex))
if err != nil {
return nil, err
}

// Run the transaction with tracing enabled.
vmenv := vm.NewEVM(context, statedb, api.config, vm.Config{Debug: true, Tracer: tracer})
ret, gas, err := core.ApplyMessage(vmenv, msg, new(core.GasPool).AddGas(tx.Gas()))
if err != nil {
return nil, fmt.Errorf("tracing failed: %v", err)
}
switch tracer := tracer.(type) {
case *vm.StructLogger:
return &ethapi.ExecutionResult{
Gas:         gas,
ReturnValue: fmt.Sprintf("%x", ret),
StructLogs:  ethapi.FormatLogs(tracer.StructLogs()),
}, nil
```

```
case *ethapi.JavascriptTracer:
return tracer.GetResult()
default:
panic(fmt.Sprintf("bad tracer type %T", tracer))
}
}

// computeTxEnv returns the execution environment of a certain transaction.
func (api *PrivateDebugAPI) computeTxEnv(blockHash common.Hash, txIndex int)
(core.Message, vm.Context, *state.StateDB, error) {
// Create the parent state.
block := api.eth.BlockChain().GetBlockByHash(blockHash)
if block == nil {
return nil, vm.Context{}, nil, fmt.Errorf("block %x not found", blockHash)
}
parent := api.eth.BlockChain().GetBlock(block.ParentHash(), block.NumberU64()-1)
if parent == nil {
return nil, vm.Context{}, nil, fmt.Errorf("block parent %x not found", block.ParentHash())
}
statedb, err := api.eth.BlockChain().StateAt(parent.Root())
if err != nil {
return nil, vm.Context{}, nil, err
}
txs := block.Transactions()

// Recompute transactions up to the target index.
signer := types.MakeSigner(api.config, block.Number())
for idx, tx := range txs {
// Assemble the transaction call message
msg, _ := tx.AsMessage(signer)
context := core.NewEVMContext(msg, block.Header(), api.eth.BlockChain(), nil)
if idx == txIndex {
return msg, context, statedb, nil
}

vmenv := vm.NewEVM(context, statedb, api.config, vm.Config{})
gp := new(core.GasPool).AddGas(tx.Gas())
_, _, err := core.ApplyMessage(vmenv, msg, gp)
if err != nil {
return nil, vm.Context{}, nil, fmt.Errorf("tx %x failed: %v", tx.Hash(), err)
}
statedb.DeleteSuicides()
```

```
}
return nil, vm.Context{}, nil, fmt.Errorf("tx index %d out of range for block %x", txIndex, blockHash)
}

// Preimage is a debug API function that returns the preimage for a sha3 hash, if known.
func (api *PrivateDebugAPI) Preimage(ctx context.Context, hash common.Hash) (hexutil.Bytes,
error) {
db := core.PreimageTable(api.eth.ChainDb())
return db.Get(hash.Bytes())
}

// GetBadBLocks returns a list of the last 'bad blocks' that the client has seen on the network
// and returns them as a JSON list of block-hashes
func (api *PrivateDebugAPI) GetBadBlocks(ctx context.Context) ([]core.BadBlockArgs, error) {
return api.eth.BlockChain().BadBlocks()
}

// StorageRangeResult is the result of a debug_storageRangeAt API call.
type StorageRangeResult struct {
Storage storageMap   `json:"storage"`
NextKey *common.Hash `json:"nextKey"` // nil if Storage includes the last key in the trie.
}

type storageMap map[common.Hash]storageEntry

type storageEntry struct {
Key   *common.Hash `json:"key"`
Value common.Hash  `json:"value"`
}

// StorageRangeAt returns the storage at the given block height and transaction index.
func (api *PrivateDebugAPI) StorageRangeAt(ctx context.Context, blockHash common.Hash,
txIndex int, contractAddress common.Address, keyStart hexutil.Bytes, maxResult int)
(StorageRangeResult, error) {
_, _, statedb, err := api.computeTxEnv(blockHash, txIndex)
if err != nil {
return StorageRangeResult{}, err
}
st := statedb.StorageTrie(contractAddress)
if st == nil {
return StorageRangeResult{}, fmt.Errorf("account %x doesn't exist", contractAddress)
}
```

```
return storageRangeAt(st, keyStart, maxResult), nil
}

func storageRangeAt(st state.Trie, start []byte, maxResult int) StorageRangeResult {
it := trie.NewIterator(st.NodeIterator(start))
result := StorageRangeResult{Storage: storageMap{}}
for i := 0; i < maxResult && it.Next(); i++ {
e := storageEntry{Value: common.BytesToHash(it.Value)}
if preimage := st.GetKey(it.Key); preimage != nil {
preimage := common.BytesToHash(preimage)
e.Key = &preimage
}
result.Storage[common.BytesToHash(it.Key)] = e
}
// Add the 'next key' so clients can continue downloading.
if it.Next() {
next := common.BytesToHash(it.Key)
result.NextKey = &next
}
return result
}

95:F:\git\coin\ethereum\go-ethereum\eth\api_backend.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package eth

import (
"context"
"math/big"

"github.com/ethereum/go-ethereum/accounts"
"github.com/ethereum/go-ethereum/common"
"github.com/ethereum/go-ethereum/common/math"
"github.com/ethereum/go-ethereum/core"
"github.com/ethereum/go-ethereum/core/state"
"github.com/ethereum/go-ethereum/core/types"
"github.com/ethereum/go-ethereum/core/vm"
"github.com/ethereum/go-ethereum/eth/downloader"
"github.com/ethereum/go-ethereum/eth/gasprice"
"github.com/ethereum/go-ethereum/ethdb"
"github.com/ethereum/go-ethereum/event"
```

```
"github.com/ethereum/go-ethereum/params"
"github.com/ethereum/go-ethereum/rpc"
)

// EthApiBackend implements ethapi.Backend for full nodes
type EthApiBackend struct {
eth *Ethereum
gpo *gasprice.Oracle
}

func (b *EthApiBackend) ChainConfig() *params.ChainConfig {
return b.eth.chainConfig
}

func (b *EthApiBackend) CurrentBlock() *types.Block {
return b.eth.blockchain.CurrentBlock()
}

func (b *EthApiBackend) SetHead(number uint64) {
b.eth.protocolManager.downloader.Cancel()
b.eth.blockchain.SetHead(number)
}

func (b *EthApiBackend) HeaderByNumber(ctx context.Context, blockNr rpc.BlockNumber)
(*types.Header, error) {
// Pending block is only known by the miner
if blockNr == rpc.PendingBlockNumber {
block := b.eth.miner.PendingBlock()
return block.Header(), nil
}
// Otherwise resolve and return the block
if blockNr == rpc.LatestBlockNumber {
return b.eth.blockchain.CurrentBlock().Header(), nil
}
return b.eth.blockchain.GetHeaderByNumber(uint64(blockNr)), nil
}

func (b *EthApiBackend) BlockByNumber(ctx context.Context, blockNr rpc.BlockNumber)
(*types.Block, error) {
// Pending block is only known by the miner
if blockNr == rpc.PendingBlockNumber {
block := b.eth.miner.PendingBlock()
```

```
return block, nil
}
// Otherwise resolve and return the block
if blockNr == rpc.LatestBlockNumber {
return b.eth.blockchain.CurrentBlock(), nil
}
return b.eth.blockchain.GetBlockByNumber(uint64(blockNr)), nil
}

func (b *EthApiBackend) StateAndHeaderByNumber(ctx context.Context, blockNr
rpc.BlockNumber) (*state.StateDB, *types.Header, error) {
// Pending state is only known by the miner
if blockNr == rpc.PendingBlockNumber {
block, state := b.eth.miner.Pending()
return state, block.Header(), nil
}
// Otherwise resolve the block number and return its state
header, err := b.HeaderByNumber(ctx, blockNr)
if header == nil || err != nil {
return nil, nil, err
}
stateDb, err := b.eth.BlockChain().StateAt(header.Root)
return stateDb, header, err
}

func (b *EthApiBackend) GetBlock(ctx context.Context, blockHash common.Hash) (*types.Block,
error) {
return b.eth.blockchain.GetBlockByHash(blockHash), nil
}

func (b *EthApiBackend) GetReceipts(ctx context.Context, blockHash common.Hash)
(types.Receipts, error) {
return core.GetBlockReceipts(b.eth.chainDb, blockHash, core.GetBlockNumber(b.eth.chainDb,
blockHash)), nil
}

func (b *EthApiBackend) GetTd(blockHash common.Hash) *big.Int {
return b.eth.blockchain.GetTdByHash(blockHash)
}

func (b *EthApiBackend) GetEVM(ctx context.Context, msg core.Message, state *state.StateDB,
header *types.Header, vmCfg vm.Config) (*vm.EVM, func() error, error) {
```

```
state.SetBalance(msg.From(), math.MaxBig256)
vmError := func() error { return nil }

context := core.NewEVMContext(msg, header, b.eth.BlockChain(), nil)
return vm.NewEVM(context, state, b.eth.chainConfig, vmCfg), vmError, nil
}

func (b *EthApiBackend) SendTx(ctx context.Context, signedTx *types.Transaction) error {
b.eth.txMu.Lock()
defer b.eth.txMu.Unlock()

b.eth.txPool.SetLocal(signedTx)
return b.eth.txPool.Add(signedTx)
}

func (b *EthApiBackend) RemoveTx(txHash common.Hash) {
b.eth.txMu.Lock()
defer b.eth.txMu.Unlock()

b.eth.txPool.Remove(txHash)
}

func (b *EthApiBackend) GetPoolTransactions() (types.Transactions, error) {
b.eth.txMu.Lock()
defer b.eth.txMu.Unlock()

pending, err := b.eth.txPool.Pending()
if err != nil {
return nil, err
}

var txs types.Transactions
for _, batch := range pending {
txs = append(txs, batch...)
}
return txs, nil
}

func (b *EthApiBackend) GetPoolTransaction(hash common.Hash) *types.Transaction {
b.eth.txMu.Lock()
defer b.eth.txMu.Unlock()
```

```
return b.eth.txPool.Get(hash)
}

func (b *EthApiBackend) GetPoolNonce(ctx context.Context, addr common.Address) (uint64,
error) {
b.eth.txMu.Lock()
defer b.eth.txMu.Unlock()

return b.eth.txPool.State().GetNonce(addr), nil
}

func (b *EthApiBackend) Stats() (pending int, queued int) {
b.eth.txMu.Lock()
defer b.eth.txMu.Unlock()

return b.eth.txPool.Stats()
}

func (b *EthApiBackend) TxPoolContent() (map[common.Address]types.Transactions,
map[common.Address]types.Transactions) {
b.eth.txMu.Lock()
defer b.eth.txMu.Unlock()

return b.eth.TxPool().Content()
}

func (b *EthApiBackend) Downloader() *downloader.Downloader {
return b.eth.Downloader()
}

func (b *EthApiBackend) ProtocolVersion() int {
return b.eth.EthVersion()
}

func (b *EthApiBackend) SuggestPrice(ctx context.Context) (*big.Int, error) {
return b.gpo.SuggestPrice(ctx)
}

func (b *EthApiBackend) ChainDb() ethdb.Database {
return b.eth.ChainDb()
}
```

```
func (b *EthApiBackend) EventMux() *event.TypeMux {
return b.eth.EventMux()
}

func (b *EthApiBackend) AccountManager() *accounts.Manager {
return b.eth.AccountManager()
}

96:F:\git\coin\ethereum\go-ethereum\eth\api_test.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package eth

import (
"reflect"
"testing"

"github.com/davecgh/go-spew/spew"
"github.com/ethereum/go-ethereum/common"
"github.com/ethereum/go-ethereum/core/state"
"github.com/ethereum/go-ethereum/ethdb"
)

var dumper = spew.ConfigState{Indent: "    "}

func TestStorageRangeAt(t *testing.T) {
// Create a state where account 0x010000... has a few storage entries.
var (
db, _    = ethdb.NewMemDatabase()
state, _ = state.New(common.Hash{}, state.NewDatabase(db))
addr     = common.Address{0x01}
keys     = []common.Hash{ // hashes of Keys of storage
common.HexToHash("340dd630ad21bf010b4e676dbfa9ba9a02175262d1fa356232cfde6cb5b47ef2"),
common.HexToHash("426fcb404ab2d5d8e61a3d918108006bbb0a9be65e92235bb10eefbdb6dcd053"),
common.HexToHash("48078cfed56339ea54962e72c37c7f588fc4f8e5bc173827ba75cb10a63a96a5"),
common.HexToHash("5723d2c3a83af9b735e3b7f21531e5623d183a9095a56604ead41f3582fdfb75"),
}
storage = storageMap{
```

```
keys[0]: {Key: &common.Hash{0x02}, Value: common.Hash{0x01}},
keys[1]: {Key: &common.Hash{0x04}, Value: common.Hash{0x02}},
keys[2]: {Key: &common.Hash{0x01}, Value: common.Hash{0x03}},
keys[3]: {Key: &common.Hash{0x03}, Value: common.Hash{0x04}},
}
)
for _, entry := range storage {
state.SetState(addr, *entry.Key, entry.Value)
}

// Check a few combinations of limit and start/end.
tests := []struct {
start []byte
limit int
want  StorageRangeResult
}{
{
start: []byte{}, limit: 0,
want: StorageRangeResult{storageMap{}, &keys[0]},
},
{
start: []byte{}, limit: 100,
want: StorageRangeResult{storage, nil},
},
{
start: []byte{}, limit: 2,
want: StorageRangeResult{storageMap{keys[0]: storage[keys[0]], keys[1]: storage[keys[1]]},
&keys[2]},
},
{
start: []byte{0x00}, limit: 4,
want: StorageRangeResult{storage, nil},
},
{
start: []byte{0x40}, limit: 2,
want: StorageRangeResult{storageMap{keys[1]: storage[keys[1]], keys[2]: storage[keys[2]]},
&keys[3]},
},
}
for _, test := range tests {
result := storageRangeAt(state.StorageTrie(addr), test.start, test.limit)
if !reflect.DeepEqual(result, test.want) {
```

```
t.Fatalf("wrong result for range 0x%x.., limit %d:\ngot %s\nwant %s",
test.start, test.limit, dumper.Sdump(result), dumper.Sdump(&test.want))
}
}
}
```

97:F:\git\coin\ethereum\go-ethereum\eth\backend.go

```
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

// Package eth implements the Ethereum protocol.
package eth

import (
"errors"
"fmt"
"math/big"
"runtime"
"sync"
"sync/atomic"

"github.com/ethereum/go-ethereum/accounts"
"github.com/ethereum/go-ethereum/common"
"github.com/ethereum/go-ethereum/common/hexutil"
"github.com/ethereum/go-ethereum/consensus"
"github.com/ethereum/go-ethereum/consensus/clique"
"github.com/ethereum/go-ethereum/consensus/ethash"
"github.com/ethereum/go-ethereum/core"
"github.com/ethereum/go-ethereum/core/types"
"github.com/ethereum/go-ethereum/core/vm"
"github.com/ethereum/go-ethereum/eth/downloader"
"github.com/ethereum/go-ethereum/eth/filters"
"github.com/ethereum/go-ethereum/eth/gasprice"
"github.com/ethereum/go-ethereum/ethdb"
"github.com/ethereum/go-ethereum/event"
"github.com/ethereum/go-ethereum/internal/ethapi"
"github.com/ethereum/go-ethereum/log"
"github.com/ethereum/go-ethereum/miner"
"github.com/ethereum/go-ethereum/node"
"github.com/ethereum/go-ethereum/p2p"
"github.com/ethereum/go-ethereum/params"
"github.com/ethereum/go-ethereum/rlp"
"github.com/ethereum/go-ethereum/rpc"
```

```
)

type LesServer interface {
Start(srvr *p2p.Server)
Stop()
Protocols() []p2p.Protocol
}

// Ethereum implements the Ethereum full node service.
type Ethereum struct {
chainConfig *params.ChainConfig
// Channel for shutting down the service
shutdownChan  chan bool // Channel for shutting down the ethereum
stopDbUpgrade func()  // stop chain db sequential key upgrade
// Handlers
txPool        *core.TxPool
txMu          sync.Mutex
blockchain     *core.BlockChain
protocolManager *ProtocolManager
lesServer      LesServer
// DB interfaces
chainDb ethdb.Database // Block chain database

eventMux       *event.TypeMux
engine         consensus.Engine
accountManager *accounts.Manager

ApiBackend *EthApiBackend

miner     *miner.Miner
gasPrice  *big.Int
etherbase common.Address

networkId     uint64
netRPCService *ethapi.PublicNetAPI

lock sync.RWMutex // Protects the variadic fields (e.g. gas price and etherbase)
}

func (s *Ethereum) AddLesServer(ls LesServer) {
s.lesServer = ls
}
```

```
// New creates a new Ethereum object (including the
// initialisation of the common Ethereum object)
func New(ctx *node.ServiceContext, config *Config) (*Ethereum, error) {
if config.SyncMode == downloader.LightSync {
return nil, errors.New("can't run eth.Ethereum in light sync mode, use les.LightEthereum")
}
if !config.SyncMode.IsValid() {
return nil, fmt.Errorf("invalid sync mode %d", config.SyncMode)
}

chainDb, err := CreateDB(ctx, config, "chaindata")
if err != nil {
return nil, err
}
stopDbUpgrade := upgradeSequentialKeys(chainDb)
chainConfig, genesisHash, genesisErr := core.SetupGenesisBlock(chainDb, config.Genesis)
if _, ok := genesisErr.(*params.ConfigCompatError); genesisErr != nil && !ok {
return nil, genesisErr
}
log.Info("Initialised chain configuration", "config", chainConfig)

eth := &Ethereum{
chainDb:        chainDb,
chainConfig:    chainConfig,
eventMux:       ctx.EventMux,
accountManager: ctx.AccountManager,
engine:         CreateConsensusEngine(ctx, config, chainConfig, chainDb),
shutdownChan:   make(chan bool),
stopDbUpgrade:  stopDbUpgrade,
networkId:      config.NetworkId,
gasPrice:       config.GasPrice,
etherbase:      config.Etherbase,
}

if err := addMipmapBloomBins(chainDb); err != nil {
return nil, err
}
log.Info("Initialising Ethereum protocol", "versions", ProtocolVersions, "network", config.NetworkId)

if !config.SkipBcVersionCheck {
bcVersion := core.GetBlockChainVersion(chainDb)
```

```
if bcVersion != core.BlockChainVersion && bcVersion != 0 {
return nil, fmt.Errorf("Blockchain DB version mismatch (%d / %d). Run geth upgradedb.\n",
bcVersion, core.BlockChainVersion)
}
core.WriteBlockChainVersion(chainDb, core.BlockChainVersion)
}

vmConfig := vm.Config{EnablePreimageRecording: config.EnablePreimageRecording}
eth.blockchain, err = core.NewBlockChain(chainDb, eth.chainConfig, eth.engine, eth.eventMux,
vmConfig)
if err != nil {
return nil, err
}
// Rewind the chain in case of an incompatible config upgrade.
if compat, ok := genesisErr.(*params.ConfigCompatError); ok {
log.Warn("Rewinding chain to upgrade configuration", "err", compat)
eth.blockchain.SetHead(compat.RewindTo)
core.WriteChainConfig(chainDb, genesisHash, chainConfig)
}

newPool := core.NewTxPool(config.TxPool, eth.chainConfig, eth.EventMux(),
eth.blockchain.State, eth.blockchain.GasLimit)
eth.txPool = newPool

maxPeers := config.MaxPeers
if config.LightServ > 0 {
// if we are running a light server, limit the number of ETH peers so that we reserve some space
for incoming LES connections
// temporary solution until the new peer connectivity API is finished
halfPeers := maxPeers / 2
maxPeers -= config.LightPeers
if maxPeers < halfPeers {
maxPeers = halfPeers
}
}

if eth.protocolManager, err = NewProtocolManager(eth.chainConfig, config.SyncMode,
config.NetworkId, maxPeers, eth.eventMux, eth.txPool, eth.engine, eth.blockchain, chainDb); err
!= nil {
return nil, err
}
```

```
eth.miner = miner.New(eth, eth.chainConfig, eth.EventMux(), eth.engine)
eth.miner.SetExtra(makeExtraData(config.ExtraData))

eth.ApiBackend = &EthApiBackend{eth, nil}
gpoParams := config.GPO
if gpoParams.Default == nil {
gpoParams.Default = config.GasPrice
}
eth.ApiBackend.gpo = gasprice.NewOracle(eth.ApiBackend, gpoParams)

return eth, nil
}

func makeExtraData(extra []byte) []byte {
if len(extra) == 0 {
// create default extradata
extra, _ = rlp.EncodeToBytes([]interface{}{
uint(params.VersionMajor<<16 | params.VersionMinor<<8 | params.VersionPatch),
"geth",
runtime.Version(),
runtime.GOOS,
})
}
if uint64(len(extra)) > params.MaximumExtraDataSize {
log.Warn("Miner extra data exceed limit", "extra", hexutil.Bytes(extra), "limit",
params.MaximumExtraDataSize)
extra = nil
}
return extra
}

// CreateDB creates the chain database.
func CreateDB(ctx *node.ServiceContext, config *Config, name string) (ethdb.Database, error) {
db, err := ctx.OpenDatabase(name, config.DatabaseCache, config.DatabaseHandles)
if err != nil {
return nil, err
}
if db, ok := db.(*ethdb.LDBDatabase); ok {
db.Meter("eth/db/chaindata/")
}
return db, nil
}
```

```
// CreateConsensusEngine creates the required type of consensus engine instance for an
Ethereum service
func CreateConsensusEngine(ctx *node.ServiceContext, config *Config, chainConfig
*params.ChainConfig, db ethdb.Database) consensus.Engine {
// If proof-of-authority is requested, set it up
if chainConfig.Clique != nil {
return clique.New(chainConfig.Clique, db)
}
// Otherwise assume proof-of-work
switch {
case config.PowFake:
log.Warn("Ethash used in fake mode")
return ethash.NewFaker()
case config.PowTest:
log.Warn("Ethash used in test mode")
return ethash.NewTester()
case config.PowShared:
log.Warn("Ethash used in shared mode")
return ethash.NewShared()
default:
engine := ethash.New(ctx.ResolvePath(config.EthashCacheDir), config.EthashCachesInMem,
config.EthashCachesOnDisk,
config.EthashDatasetDir, config.EthashDatasetsInMem, config.EthashDatasetsOnDisk)
engine.SetThreads(-1) // Disable CPU mining
return engine
}
}

// APIs returns the collection of RPC services the ethereum package offers.
// NOTE, some of these services probably need to be moved to somewhere else.
func (s *Ethereum) APIs() []rpc.API {
apis := ethapi.GetAPIs(s.ApiBackend)

// Append any APIs exposed explicitly by the consensus engine
apis = append(apis, s.engine.APIs(s.BlockChain())...)

// Append all the local APIs and return
return append(apis, []rpc.API{
{
Namespace: "eth",
Version:   "1.0",
```

```
Service:   NewPublicEthereumAPI(s),
Public:    true,
}, {
Namespace: "eth",
Version:   "1.0",
Service:   NewPublicMinerAPI(s),
Public:    true,
}, {
Namespace: "eth",
Version:   "1.0",
Service:   downloader.NewPublicDownloaderAPI(s.protocolManager.downloader, s.eventMux),
Public:    true,
}, {
Namespace: "miner",
Version:   "1.0",
Service:   NewPrivateMinerAPI(s),
Public:    false,
}, {
Namespace: "eth",
Version:   "1.0",
Service:   filters.NewPublicFilterAPI(s.ApiBackend, false),
Public:    true,
}, {
Namespace: "admin",
Version:   "1.0",
Service:   NewPrivateAdminAPI(s),
}, {
Namespace: "debug",
Version:   "1.0",
Service:   NewPublicDebugAPI(s),
Public:    true,
}, {
Namespace: "debug",
Version:   "1.0",
Service:   NewPrivateDebugAPI(s.chainConfig, s),
}, {
Namespace: "net",
Version:   "1.0",
Service:   s.netRPCService,
Public:    true,
},
}...)
```

```
}

func (s *Ethereum) ResetWithGenesisBlock(gb *types.Block) {
s.blockchain.ResetWithGenesisBlock(gb)
}

func (s *Ethereum) Etherbase() (eb common.Address, err error) {
s.lock.RLock()
etherbase := s.etherbase
s.lock.RUnlock()

if etherbase != (common.Address{}) {
return etherbase, nil
}
if wallets := s.AccountManager().Wallets(); len(wallets) > 0 {
if accounts := wallets[0].Accounts(); len(accounts) > 0 {
return accounts[0].Address, nil
}
}
return common.Address{}, fmt.Errorf("etherbase address must be explicitly specified")
}

// set in js console via admin interface or wrapper from cli flags
func (self *Ethereum) SetEtherbase(etherbase common.Address) {
self.lock.Lock()
self.etherbase = etherbase
self.lock.Unlock()

self.miner.SetEtherbase(etherbase)
}

func (s *Ethereum) StartMining(local bool) error {
eb, err := s.Etherbase()
if err != nil {
log.Error("Cannot start mining without etherbase", "err", err)
return fmt.Errorf("etherbase missing: %v", err)
}
if clique, ok := s.engine.(*clique.Clique); ok {
wallet, err := s.accountManager.Find(accounts.Account{Address: eb})
if wallet == nil || err != nil {
log.Error("Etherbase account unavailable locally", "err", err)
return fmt.Errorf("singer missing: %v", err)
```

```
}
clique.Authorize(eb, wallet.SignHash)
}
if local {
// If local (CPU) mining is started, we can disable the transaction rejection
// mechanism introduced to speed sync times. CPU mining on mainnet is ludicrous
// so noone will ever hit this path, whereas marking sync done on CPU mining
// will ensure that private networks work in single miner mode too.
atomic.StoreUint32(&s.protocolManager.acceptTxs, 1)
}
go s.miner.Start(eb)
return nil
}

func (s *Ethereum) StopMining()         { s.miner.Stop() }
func (s *Ethereum) IsMining() bool      { return s.miner.Mining() }
func (s *Ethereum) Miner() *miner.Miner { return s.miner }

func (s *Ethereum) AccountManager() *accounts.Manager  { return s.accountManager }
func (s *Ethereum) BlockChain() *core.BlockChain       { return s.blockchain }
func (s *Ethereum) TxPool() *core.TxPool               { return s.txPool }
func (s *Ethereum) EventMux() *event.TypeMux           { return s.eventMux }
func (s *Ethereum) Engine() consensus.Engine           { return s.engine }
func (s *Ethereum) ChainDb() ethdb.Database            { return s.chainDb }
func (s *Ethereum) IsListening() bool                  { return true } // Always listening
func (s *Ethereum) EthVersion() int                    { return
int(s.protocolManager.SubProtocols[0].Version) }
func (s *Ethereum) NetVersion() uint64                 { return s.networkId }
func (s *Ethereum) Downloader() *downloader.Downloader { return s.protocolManager.downloader
}

// Protocols implements node.Service, returning all the currently configured
// network protocols to start.
func (s *Ethereum) Protocols() []p2p.Protocol {
if s.lesServer == nil {
return s.protocolManager.SubProtocols
} else {
return append(s.protocolManager.SubProtocols, s.lesServer.Protocols()...)
}
}

// Start implements node.Service, starting all internal goroutines needed by the
```

```
// Ethereum protocol implementation.
func (s *Ethereum) Start(srvr *p2p.Server) error {
s.netRPCService = ethapi.NewPublicNetAPI(srvr, s.NetVersion())

s.protocolManager.Start()
if s.lesServer != nil {
s.lesServer.Start(srvr)
}
return nil
}

// Stop implements node.Service, terminating all internal goroutines used by the
// Ethereum protocol.
func (s *Ethereum) Stop() error {
if s.stopDbUpgrade != nil {
s.stopDbUpgrade()
}
s.blockchain.Stop()
s.protocolManager.Stop()
if s.lesServer != nil {
s.lesServer.Stop()
}
s.txPool.Stop()
s.miner.Stop()
s.eventMux.Stop()

s.chainDb.Close()
close(s.shutdownChan)

return nil
}

98:F:\git\coin\ethereum\go-ethereum\eth\backend_test.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package eth

import (
"math/big"
"testing"

"github.com/ethereum/go-ethereum/common"
```

```
"github.com/ethereum/go-ethereum/core"
"github.com/ethereum/go-ethereum/core/types"
"github.com/ethereum/go-ethereum/ethdb"
"github.com/ethereum/go-ethereum/params"
)

func TestMipmapUpgrade(t *testing.T) {
db, _ := ethdb.NewMemDatabase()
addr := common.BytesToAddress([]byte("jeff"))
genesis := new(core.Genesis).MustCommit(db)

chain, receipts := core.GenerateChain(params.TestChainConfig, genesis, db, 10, func(i int, gen
*core.BlockGen) {
var receipts types.Receipts
switch i {
case 1:
receipt := types.NewReceipt(nil, new(big.Int))
receipt.Logs = []*types.Log{{Address: addr}}
gen.AddUncheckedReceipt(receipt)
receipts = types.Receipts{receipt}
case 2:
receipt := types.NewReceipt(nil, new(big.Int))
receipt.Logs = []*types.Log{{Address: addr}}
gen.AddUncheckedReceipt(receipt)
receipts = types.Receipts{receipt}
}

// store the receipts
err := core.WriteReceipts(db, receipts)
if err != nil {
t.Fatal(err)
}
})
for i, block := range chain {
core.WriteBlock(db, block)
if err := core.WriteCanonicalHash(db, block.Hash(), block.NumberU64()); err != nil {
t.Fatalf("failed to insert block number: %v", err)
}
if err := core.WriteHeadBlockHash(db, block.Hash()); err != nil {
t.Fatalf("failed to insert block number: %v", err)
}
if err := core.WriteBlockReceipts(db, block.Hash(), block.NumberU64(), receipts[i]); err != nil {
```

```
t.Fatal("error writing block receipts:", err)
}
}

err := addMipmapBloomBins(db)
if err != nil {
t.Fatal(err)
}

bloom := core.GetMipmapBloom(db, 1, core.MIPMapLevels[0])
if (bloom == types.Bloom{}) {
t.Error("got empty bloom filter")
}

data, _ := db.Get([]byte("setting-mipmap-version"))
if len(data) == 0 {
t.Error("setting-mipmap-version not written to database")
}
}

99:F:\git\coin\ethereum\go-ethereum\eth\bind.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package eth

import (
"context"
"math/big"

"github.com/ethereum/go-ethereum"
"github.com/ethereum/go-ethereum/common"
"github.com/ethereum/go-ethereum/common/hexutil"
"github.com/ethereum/go-ethereum/core/types"
"github.com/ethereum/go-ethereum/internal/ethapi"
"github.com/ethereum/go-ethereum/rlp"
"github.com/ethereum/go-ethereum/rpc"
)

// ContractBackend implements bind.ContractBackend with direct calls to Ethereum
// internals to support operating on contracts within subprotocols like eth and
// swarm.
//
```

```
// Internally this backend uses the already exposed API endpoints of the Ethereum
// object. These should be rewritten to internal Go method calls when the Go API
// is refactored to support a clean library use.
type ContractBackend struct {
eapi  *ethapi.PublicEthereumAPI        // Wrapper around the Ethereum object to access metadata
bcapi *ethapi.PublicBlockChainAPI      // Wrapper around the blockchain to access chain data
txapi *ethapi.PublicTransactionPoolAPI // Wrapper around the transaction pool to access
transaction data
}

// NewContractBackend creates a new native contract backend using an existing
// Etheruem object.
func NewContractBackend(apiBackend ethapi.Backend) *ContractBackend {
return &ContractBackend{
eapi:  ethapi.NewPublicEthereumAPI(apiBackend),
bcapi: ethapi.NewPublicBlockChainAPI(apiBackend),
txapi: ethapi.NewPublicTransactionPoolAPI(apiBackend, new(ethapi.AddrLocker)),
}
}

// CodeAt retrieves any code associated with the contract from the local API.
func (b *ContractBackend) CodeAt(ctx context.Context, contract common.Address, blockNum
*big.Int) ([]byte, error) {
return b.bcapi.GetCode(ctx, contract, toBlockNumber(blockNum))
}

// CodeAt retrieves any code associated with the contract from the local API.
func (b *ContractBackend) PendingCodeAt(ctx context.Context, contract common.Address)
([]byte, error) {
return b.bcapi.GetCode(ctx, contract, rpc.PendingBlockNumber)
}

// ContractCall implements bind.ContractCaller executing an Ethereum contract
// call with the specified data as the input. The pending flag requests execution
// against the pending block, not the stable head of the chain.
func (b *ContractBackend) CallContract(ctx context.Context, msg ethereum.CallMsg, blockNum
*big.Int) ([]byte, error) {
out, err := b.bcapi.Call(ctx, toCallArgs(msg), toBlockNumber(blockNum))
return out, err
}

// ContractCall implements bind.ContractCaller executing an Ethereum contract
```

```
// call with the specified data as the input. The pending flag requests execution
// against the pending block, not the stable head of the chain.
func (b *ContractBackend) PendingCallContract(ctx context.Context, msg ethereum.CallMsg)
([]byte, error) {
out, err := b.bcapi.Call(ctx, toCallArgs(msg), rpc.PendingBlockNumber)
return out, err
}

func toCallArgs(msg ethereum.CallMsg) ethapi.CallArgs {
args := ethapi.CallArgs{
To:   msg.To,
From: msg.From,
Data: msg.Data,
}
if msg.Gas != nil {
args.Gas = hexutil.Big(*msg.Gas)
}
if msg.GasPrice != nil {
args.GasPrice = hexutil.Big(*msg.GasPrice)
}
if msg.Value != nil {
args.Value = hexutil.Big(*msg.Value)
}
return args
}

func toBlockNumber(num *big.Int) rpc.BlockNumber {
if num == nil {
return rpc.LatestBlockNumber
}
return rpc.BlockNumber(num.Int64())
}

// PendingAccountNonce implements bind.ContractTransactor retrieving the current
// pending nonce associated with an account.
func (b *ContractBackend) PendingNonceAt(ctx context.Context, account common.Address)
(nonce uint64, err error) {
out, err := b.txapi.GetTransactionCount(ctx, account, rpc.PendingBlockNumber)
if out != nil {
nonce = uint64(*out)
}
return nonce, err
```

```
}

// SuggestGasPrice implements bind.ContractTransactor retrieving the currently
// suggested gas price to allow a timely execution of a transaction.
func (b *ContractBackend) SuggestGasPrice(ctx context.Context) (*big.Int, error) {
return b.eapi.GasPrice(ctx)
}

// EstimateGasLimit implements bind.ContractTransactor triing to estimate the gas
// needed to execute a specific transaction based on the current pending state of
// the backend blockchain. There is no guarantee that this is the true gas limit
// requirement as other transactions may be added or removed by miners, but it
// should provide a basis for setting a reasonable default.
func (b *ContractBackend) EstimateGas(ctx context.Context, msg ethereum.CallMsg) (*big.Int,
error) {
out, err := b.bcapi.EstimateGas(ctx, toCallArgs(msg))
return out.ToInt(), err
}

// SendTransaction implements bind.ContractTransactor injects the transaction
// into the pending pool for execution.
func (b *ContractBackend) SendTransaction(ctx context.Context, tx *types.Transaction) error {
raw, _ := rlp.EncodeToBytes(tx)
_, err := b.txapi.SendRawTransaction(ctx, raw)
return err
}

100:F:\git\coin\ethereum\go-ethereum\eth\config.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package eth

import (
"math/big"
"os"
"os/user"
"path/filepath"
"runtime"

"github.com/ethereum/go-ethereum/common"
"github.com/ethereum/go-ethereum/common/hexutil"
"github.com/ethereum/go-ethereum/core"
```

```
"github.com/ethereum/go-ethereum/eth/downloader"
"github.com/ethereum/go-ethereum/eth/gasprice"
"github.com/ethereum/go-ethereum/params"
)

// DefaultConfig contains default settings for use on the Ethereum main net.
var DefaultConfig = Config{
SyncMode:            downloader.FastSync,
EthashCacheDir:      "ethash",
EthashCachesInMem:   2,
EthashCachesOnDisk:  3,
EthashDatasetsInMem:  1,
EthashDatasetsOnDisk: 2,
NetworkId:           1,
LightPeers:          20,
DatabaseCache:       128,
GasPrice:            big.NewInt(18 * params.Shannon),

TxPool: core.DefaultTxPoolConfig,
GPO: gasprice.Config{
Blocks:     10,
Percentile: 50,
},
}

func init() {
home := os.Getenv("HOME")
if home == "" {
if user, err := user.Current(); err == nil {
home = user.HomeDir
}
}
if runtime.GOOS == "windows" {
DefaultConfig.EthashDatasetDir = filepath.Join(home, "AppData", "Ethash")
} else {
DefaultConfig.EthashDatasetDir = filepath.Join(home, ".ethash")
}
}

//go:generate gencodec -type Config -field-override configMarshaling -formats toml -out
gen_config.go
```

```
type Config struct {
// The genesis block, which is inserted if the database is empty.
// If nil, the Ethereum main net block is used.
Genesis *core.Genesis `toml:",omitempty"`

// Protocol options
NetworkId uint64 // Network ID to use for selecting peers to connect to
SyncMode  downloader.SyncMode

// Light client options
LightServ  int `toml:",omitempty"` // Maximum percentage of time allowed for serving LES requests
LightPeers int `toml:",omitempty"` // Maximum number of LES client peers
MaxPeers   int `toml:"-"`          // Maximum number of global peers

// Database options
SkipBcVersionCheck bool `toml:"-"`
DatabaseHandles    int  `toml:"-"`
DatabaseCache      int

// Mining-related options
Etherbase    common.Address `toml:",omitempty"`
MinerThreads int            `toml:",omitempty"`
ExtraData    []byte         `toml:",omitempty"`
GasPrice     *big.Int

// Ethash options
EthashCacheDir       string
EthashCachesInMem    int
EthashCachesOnDisk   int
EthashDatasetDir     string
EthashDatasetsInMem  int
EthashDatasetsOnDisk int

// Transaction pool options
TxPool core.TxPoolConfig

// Gas Price Oracle options
GPO gasprice.Config

// Enables tracking of SHA3 preimages in the VM
EnablePreimageRecording bool
```

```
// Miscellaneous options
DocRoot   string `toml:"-"`
PowFake   bool   `toml:"-"`
PowTest   bool   `toml:"-"`
PowShared bool   `toml:"-"`
}

type configMarshaling struct {
ExtraData hexutil.Bytes
}

101:F:\git\coin\ethereum\go-ethereum\eth\db_upgrade.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

// Package eth implements the Ethereum protocol.
package eth

import (
"bytes"
"encoding/binary"
"fmt"
"math/big"
"time"

"github.com/ethereum/go-ethereum/common"
"github.com/ethereum/go-ethereum/core"
"github.com/ethereum/go-ethereum/core/types"
"github.com/ethereum/go-ethereum/ethdb"
"github.com/ethereum/go-ethereum/log"
"github.com/ethereum/go-ethereum/rlp"
)

var useSequentialKeys = []byte("dbUpgrade_20160530sequentialKeys")

// upgradeSequentialKeys checks the chain database version and
// starts a background process to make upgrades if necessary.
// Returns a stop function that blocks until the process has
// been safely stopped.
func upgradeSequentialKeys(db ethdb.Database) (stopFn func()) {
data, _ := db.Get(useSequentialKeys)
if len(data) > 0 && data[0] == 42 {
return nil // already converted
```

```
}

if data, _ := db.Get([]byte("LastHeader")); len(data) == 0 {
db.Put(useSequentialKeys, []byte{42})
return nil // empty database, nothing to do
}

log.Warn("Upgrading chain database to use sequential keys")

stopChn := make(chan struct{})
stoppedChn := make(chan struct{})

go func() {
stopFn := func() bool {
select {
case <-time.After(time.Microsecond * 100): // make sure other processes don't get starved
case <-stopChn:
return true
}
return false
}

err, stopped := upgradeSequentialCanonicalNumbers(db, stopFn)
if err == nil && !stopped {
err, stopped = upgradeSequentialBlocks(db, stopFn)
}
if err == nil && !stopped {
err, stopped = upgradeSequentialOrphanedReceipts(db, stopFn)
}
if err == nil && !stopped {
log.Info("Database conversion successful")
db.Put(useSequentialKeys, []byte{42})
}
if err != nil {
log.Error("Database conversion failed", "err", err)
}
close(stoppedChn)
}()

return func() {
close(stopChn)
<-stoppedChn
```

```
}
}

// upgradeSequentialCanonicalNumbers reads all old format canonical numbers from
// the database, writes them in new format and deletes the old ones if successful.
func upgradeSequentialCanonicalNumbers(db ethdb.Database, stopFn func() bool) (error, bool) {
prefix := []byte("block-num-")
it := db.(*ethdb.LDBDatabase).NewIterator()
defer func() {
it.Release()
}()
it.Seek(prefix)
cnt := 0
for bytes.HasPrefix(it.Key(), prefix) {
keyPtr := it.Key()
if len(keyPtr) < 20 {
cnt++
if cnt%100000 == 0 {
it.Release()
it = db.(*ethdb.LDBDatabase).NewIterator()
it.Seek(keyPtr)
log.Info("Converting canonical numbers", "count", cnt)
}
number := big.NewInt(0).SetBytes(keyPtr[10:]).Uint64()
newKey := []byte("h12345678n")
binary.BigEndian.PutUint64(newKey[1:9], number)
if err := db.Put(newKey, it.Value()); err != nil {
return err, false
}
if err := db.Delete(keyPtr); err != nil {
return err, false
}
}

if stopFn() {
return nil, true
}
it.Next()
}
if cnt > 0 {
log.Info("converted canonical numbers", "count", cnt)
}
```

```
return nil, false
}

// upgradeSequentialBlocks reads all old format block headers, bodies, TDs and block
// receipts from the database, writes them in new format and deletes the old ones
// if successful.
func upgradeSequentialBlocks(db ethdb.Database, stopFn func() bool) (error, bool) {
prefix := []byte("block-")
it := db.(*ethdb.LDBDatabase).NewIterator()
defer func() {
it.Release()
}()
it.Seek(prefix)
cnt := 0
for bytes.HasPrefix(it.Key(), prefix) {
keyPtr := it.Key()
if len(keyPtr) >= 38 {
cnt++
if cnt%10000 == 0 {
it.Release()
it = db.(*ethdb.LDBDatabase).NewIterator()
it.Seek(keyPtr)
log.Info("Converting blocks", "count", cnt)
}
// convert header, body, td and block receipts
var keyPrefix [38]byte
copy(keyPrefix[:], keyPtr[0:38])
hash := keyPrefix[6:38]
if err := upgradeSequentialBlockData(db, hash); err != nil {
return err, false
}
// delete old db entries belonging to this hash
for bytes.HasPrefix(it.Key(), keyPrefix[:]) {
if err := db.Delete(it.Key()); err != nil {
return err, false
}
it.Next()
}
if err := db.Delete(append([]byte("receipts-block-"), hash...)); err != nil {
return err, false
}
} else {
```

```
it.Next()
}

if stopFn() {
return nil, true
}
}
if cnt > 0 {
log.Info("Converted blocks", "count", cnt)
}
return nil, false
}

// upgradeSequentialOrphanedReceipts removes any old format block receipts from the
// database that did not have a corresponding block
func upgradeSequentialOrphanedReceipts(db ethdb.Database, stopFn func() bool) (error, bool) {
prefix := []byte("receipts-block-")
it := db.(*ethdb.LDBDatabase).NewIterator()
defer it.Release()
it.Seek(prefix)
cnt := 0
for bytes.HasPrefix(it.Key(), prefix) {
// phase 2 already converted receipts belonging to existing
// blocks, just remove if there's anything left
cnt++
if err := db.Delete(it.Key()); err != nil {
return err, false
}

if stopFn() {
return nil, true
}
it.Next()
}
if cnt > 0 {
log.Info("Removed orphaned block receipts", "count", cnt)
}
return nil, false
}

// upgradeSequentialBlockData upgrades the header, body, td and block receipts
// database entries belonging to a single hash (doesn't delete old data).
```

```
func upgradeSequentialBlockData(db ethdb.Database, hash []byte) error {
// get old chain data and block number
headerRLP, _ := db.Get(append(append([]byte("block-"), hash...), []byte("-header")...))
if len(headerRLP) == 0 {
return nil
}
header := new(types.Header)
if err := rlp.Decode(bytes.NewReader(headerRLP), header); err != nil {
return err
}
number := header.Number.Uint64()
bodyRLP, _ := db.Get(append(append([]byte("block-"), hash...), []byte("-body")...))
tdRLP, _ := db.Get(append(append([]byte("block-"), hash...), []byte("-td")...))
receiptsRLP, _ := db.Get(append([]byte("receipts-block-"), hash...))
// store new hash -> number association
encNum := make([]byte, 8)
binary.BigEndian.PutUint64(encNum, number)
if err := db.Put(append([]byte("H"), hash...), encNum); err != nil {
return err
}
// store new chain data
if err := db.Put(append(append([]byte("h"), encNum...), hash...), headerRLP); err != nil {
return err
}
if len(tdRLP) != 0 {
if err := db.Put(append(append(append([]byte("h"), encNum...), hash...), []byte("t")...), tdRLP); err
!= nil {
return err
}
}
if len(bodyRLP) != 0 {
if err := db.Put(append(append([]byte("b"), encNum...), hash...), bodyRLP); err != nil {
return err
}
}
if len(receiptsRLP) != 0 {
if err := db.Put(append(append([]byte("r"), encNum...), hash...), receiptsRLP); err != nil {
return err
}
}
return nil
}
```

```
func addMipmapBloomBins(db ethdb.Database) (err error) {
const mipmapVersion uint = 2

// check if the version is set. We ignore data for now since there's
// only one version so we can easily ignore it for now
var data []byte
data, _ = db.Get([]byte("setting-mipmap-version"))
if len(data) > 0 {
var version uint
if err := rlp.DecodeBytes(data, &version); err == nil && version == mipmapVersion {
return nil
}
}

defer func() {
if err == nil {
var val []byte
val, err = rlp.EncodeToBytes(mipmapVersion)
if err == nil {
err = db.Put([]byte("setting-mipmap-version"), val)
}
return
}
}()
latestHash := core.GetHeadBlockHash(db)
latestBlock := core.GetBlock(db, latestHash, core.GetBlockNumber(db, latestHash))
if latestBlock == nil { // clean database
return
}

tstart := time.Now()
log.Warn("Upgrading db log bloom bins")
for i := uint64(0); i <= latestBlock.NumberU64(); i++ {
hash := core.GetCanonicalHash(db, i)
if (hash == common.Hash{}) {
return fmt.Errorf("chain db corrupted. Could not find block %d.", i)
}
core.WriteMipmapBloom(db, i, core.GetBlockReceipts(db, hash, i))
}
log.Info("Bloom-bin upgrade completed", "elapsed", common.PrettyDuration(time.Since(tstart)))
return nil
```

```
}

102:F:\git\coin\ethereum\go-ethereum\eth\downloader\api.go
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.

package downloader

import (
"context"
"sync"

ethereum "github.com/ethereum/go-ethereum"
"github.com/ethereum/go-ethereum/event"
"github.com/ethereum/go-ethereum/rpc"
)

// PublicDownloaderAPI provides an API which gives information about the current synchronisation status.
// It offers only methods that operates on data that can be available to anyone without security risks.
type PublicDownloaderAPI struct {
d                       *Downloader
mux                     *event.TypeMux
installSyncSubscription   chan chan interface{}
uninstallSyncSubscription chan *uninstallSyncSubscriptionRequest
}

// NewPublicDownloaderAPI create a new PublicDownloaderAPI. The API has an internal event loop that
// listens for events from the downloader through the global event mux. In case it receives one of
// these events it broadcasts it to all syncing subscriptions that are installed through the
// installSyncSubscription channel.
func NewPublicDownloaderAPI(d *Downloader, m *event.TypeMux) *PublicDownloaderAPI {
api := &PublicDownloaderAPI{
d:   d,
mux: m,
installSyncSubscription:   make(chan chan interface{}),
uninstallSyncSubscription: make(chan *uninstallSyncSubscriptionRequest),
}

go api.eventLoop()
```

```
return api
}

// eventLoop runs an loop until the event mux closes. It will install and uninstall new
// sync subscriptions and broadcasts sync status updates to the installed sync subscriptions.
func (api *PublicDownloaderAPI) eventLoop() {
var (
sub               = api.mux.Subscribe(StartEvent{}, DoneEvent{}, FailedEvent{})
syncSubscriptions = make(map[chan interface{}]struct{})
)

for {
select {
case i := <-api.installSyncSubscription:
syncSubscriptions[i] = struct{}{}
case u := <-api.uninstallSyncSubscription:
delete(syncSubscriptions, u.c)
close(u.uninstalled)
case event := <-sub.Chan():
if event == nil {
return
}

var notification interface{}
switch event.Data.(type) {
case StartEvent:
notification = &SyncingResult{
Syncing: true,
Status:  api.d.Progress(),
}
case DoneEvent, FailedEvent:
notification = false
}
// broadcast
for c := range syncSubscriptions {
c <- notification
}
}
}
}

// Syncing provides information when this nodes starts synchronising with the Ethereum network
```

```
and when it's finished.
func (api *PublicDownloaderAPI) Syncing(ctx context.Context) (*rpc.Subscription, error) {
notifier, supported := rpc.NotifierFromContext(ctx)
if !supported {
return &rpc.Subscription{}, rpc.ErrNotificationsUnsupported
}

rpcSub := notifier.CreateSubscription()

go func() {
statuses := make(chan interface{})
sub := api.SubscribeSyncStatus(statuses)

for {
select {
case status := <-statuses:
notifier.Notify(rpcSub.ID, status)
case <-rpcSub.Err():
sub.Unsubscribe()
return
case <-notifier.Closed():
sub.Unsubscribe()
return
}
}
}()

return rpcSub, nil
}

// SyncingResult provides information about the current synchronisation status for this node.
type SyncingResult struct {
Syncing bool                `json:"syncing"`
Status  ethereum.SyncProgress `json:"status"`
}

// uninstallSyncSubscriptionRequest uninstalles a syncing subscription in the API event loop.
type uninstallSyncSubscriptionRequest struct {
c           chan interface{}
uninstalled chan interface{}
}
```

```
// SyncStatusSubscription represents a syncing subscription.
type SyncStatusSubscription struct {
api       *PublicDownloaderAPI // register subscription in event loop of this api instance
c         chan interface{}     // channel where events are broadcasted to
unsubOnce sync.Once            // make sure unsubscribe logic is executed once
}

// Unsubscribe uninstalls the subscription from the DownloadAPI event loop.
// The status channel that was passed to subscribeSyncStatus isn't used anymore
// after this method returns.
func (s *SyncStatusSubscription) Unsubscribe() {
s.unsubOnce.Do(func() {
req := uninstallSyncSubscriptionRequest{s.c, make(chan interface{})}
s.api.uninstallSyncSubscription <- &req

for {
select {
case <-s.c:
// drop new status events until uninstall confirmation
continue
case <-req.uninstalled:
return
}
}
})
}

// SubscribeSyncStatus creates a subscription that will broadcast new synchronisation updates.
// The given channel must receive interface values, the result can either
func (api *PublicDownloaderAPI) SubscribeSyncStatus(status chan interface{})
*SyncStatusSubscription {
api.installSyncSubscription <- status
return &SyncStatusSubscription{api: api, c: status}
}
```