F:\git\java\mar3\filemonitor\target\contract-module\contract-module-2.doc

0:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\main\java\org\ethereum\datasource\JournalSource.java

```
import org.ethereum.util.RLPList;

import java.util.ArrayList;
import java.util.List;

/**
 * The JournalSource records all the changes which were made before each commitUpdate
 * Unlike 'put' deletes are not propagated to the backing Source immediately but are
 * delayed until {@link Pruner} accepts and persists changes for the corresponding hash.
 * <p>
 * Normally this class is used together with State pruning: we need all the state nodes for last N
 * blocks to be able to get back to previous state for applying fork block
 * however we would like to delete 'zombie' nodes which are not referenced anymore by
 * persisting update for the block CurrentBlockNumber - N and we would
 * also like to remove the updates made by the blocks which weren't too lucky
 * to remain on the main chain by reverting update for such blocks
 *
 * @see Pruner
 * <p>
 * Created by Anton Nashatyrev on 08.11.2016.
 */
public class JournalSource<V> extends AbstractChainedSource<byte[], V, byte[], V>
        implements HashedKeySource<byte[], V> {

    public static class Update {
        byte[] updateHash;
        List<byte[]> insertedKeys = new ArrayList<>();
        List<byte[]> deletedKeys = new ArrayList<>();

        public Update() {
        }

        public Update(byte[] bytes) {
            parse(bytes);
        }

        public byte[] serialize() {
            byte[][] insertedBytes = new byte[insertedKeys.size()][];
        for (int i = 0; i < insertedBytes.length; i++) {
            insertedBytes[i] = RLP.encodeElement(insertedKeys.get(i));
        }
        byte[][] deletedBytes = new byte[deletedKeys.size()][];
        for (int i = 0; i < deletedBytes.length; i++) {
            deletedBytes[i] = RLP.encodeElement(deletedKeys.get(i));
        }
        return RLP.encodeList(RLP.encodeElement(updateHash),
                RLP.encodeList(insertedBytes), RLP.encodeList(deletedBytes));
    }

    private void parse(byte[] encoded) {
        RLPList l = (RLPList) RLP.decode2(encoded).get(0);
        updateHash = l.get(0).getRLPData();

        for (RLPElement aRInserted : (RLPList) l.get(1)) {
            insertedKeys.add(aRInserted.getRLPData());
        }
        for (RLPElement aRDeleted : (RLPList) l.get(2)) {
            deletedKeys.add(aRDeleted.getRLPData());
        }
    }

    public List<byte[]> getInsertedKeys() {
        return insertedKeys;
    }

    public List<byte[]> getDeletedKeys() {
        return deletedKeys;
    }
}

private Update currentUpdate = new Update();

Source<byte[], Update> journal = new HashMapDB<>();

/**
 * Constructs instance with the underlying backing Source
 */
public JournalSource(Source<byte[], V> src) {
    super(src);
}
```

```
public void setJournalStore(Source<byte[], byte[]> journalSource) {
    journal = new SourceCodec.BytesKey<>(journalSource,
            new Serializer<Update, byte[]>() {
                @Override
                public byte[] serialize(Update object) {
                    return object.serialize();
                }

                @Override
                public Update deserialize(byte[] stream) {
                    return stream == null ? null : new Update(stream);
                }
            });
}

/**
 * Inserts are immediately propagated to the backing Source
 * though are still recorded to the current update
 * The insert might later be reverted by {@link Pruner}
 */
@Override
public synchronized void put(byte[] key, V val) {
    if (val == null) {
        delete(key);
        return;
    }

    getSource().put(key, val);
    currentUpdate.insertedKeys.add(key);
}

/**
 * Deletes are not propagated to the backing Source immediately
 * but instead they are recorded to the current Update and
 * might be later persisted
 */
@Override
public synchronized void delete(byte[] key) {
    currentUpdate.deletedKeys.add(key);
}
```

```
    @Override
    public synchronized V get(byte[] key) {
        return getSource().get(key);
    }

    /**
     * Records all the changes made prior to this call to a single chunk
     * with supplied hash.
     * Later those updates could be either persisted to backing Source (deletes only)
     * or reverted from the backing Source (inserts only)
     */
    public synchronized Update commitUpdates(byte[] updateHash) {
        currentUpdate.updateHash = updateHash;
        journal.put(updateHash, currentUpdate);
        Update committed = currentUpdate;
        currentUpdate = new Update();
        return committed;
    }

    public Source<byte[], Update> getJournal() {
        return journal;
    }

    @Override
    public synchronized boolean flushImpl() {
        journal.flush();
        return false;
    }
}
```

1:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\main\java\org\ethereum\datasource\leveldb\LevelDbDataSource.java

```
import org.ethereum.datasource.DbSettings;
import org.ethereum.datasource.DbSource;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.util.Map;
import java.util.Set;
import java.util.concurrent.locks.ReadWriteLock;
import java.util.concurrent.locks.ReentrantReadWriteLock;
import static org.ethereum.util.ByteUtil.toHexString;

/**
 * @author Roman Mandeleil
 * @since 18.01.2015
 */
public class LevelDbDataSource implements DbSource<byte[]> {

    private static final Logger logger = LoggerFactory.getLogger("db");

    public static final String AREA = "contract";

    public static DBService dbService;

    SystemProperties config = SystemProperties.getDefault(); // initialized for standalone test

    String name;
    boolean alive;

    DbSettings settings = DbSettings.DEFAULT;

    // The native LevelDB insert/update/delete are normally thread-safe
    // However close operation is not thread-safe and may lead to a native crash when
    // accessing a closed DB.
    // The leveldbJNI lib has a protection over accessing closed DB but it is not synchronized
    // This ReadWriteLock still permits concurrent execution of insert/delete/update operations
    // however blocks them on init/close/delete operations
    private ReadWriteLock resetDbLock = new ReentrantReadWriteLock();

    public LevelDbDataSource() {
    }

    public LevelDbDataSource(String name) {
        this.name = name;
        logger.debug("New LevelDbDataSource: " + name);
    }

    @Override
    public void init() {
        init(DbSettings.DEFAULT);
    }
```

```
@Override
public void init(DbSettings settings) {
    this.settings = settings;
    resetDbLock.writeLock().lock();
    try {
        logger.debug("~> LevelDbDataSource.init(): " + name);

        if (isAlive()) {
            return;
        }

        if (name == null) {
            throw new NullPointerException("no name set to the db");
        }

        if (dbService == null) {
            throw new NullPointerException("dbService is null");
        }

        String[] areas = dbService.listArea();
        if (!ArrayUtils.contains(areas, AREA)) {
            dbService.createArea(AREA);
        }

        alive = true;

        logger.debug("<~ LevelDbDataSource.init(): " + name);
    } finally {
        resetDbLock.writeLock().unlock();
    }
}

@Override
public void reset() {
}

@Override
public byte[] prefixLookup(byte[] key, int prefixBytes) {
    throw new RuntimeException("LevelDbDataSource.prefixLookup() is not supported");
}

@Override
```

```
    public boolean isAlive() {
        return alive;
    }

    @Override
    public void setName(String name) {
        this.name = name;
    }

    @Override
    public String getName() {
        return name;
    }

    @Override
    public byte[] get(byte[] key) {
        resetDbLock.readLock().lock();
        try {
            if (logger.isTraceEnabled()) {
                logger.trace("~> LevelDbDataSource.get(): " + name + ", key: " + toHexString(key));
            }
            try {
                byte[] ret = dbService.get(AREA, key);
                if (logger.isTraceEnabled()) {
                    logger.trace("<~ LevelDbDataSource.get(): " + name + ", key: " + toHexString(key) +
", " + (ret == null ? "null" : ret.length));
                }
                return ret;
            } catch (Exception e) {
                logger.warn("Exception. Retrying again...", e);
                byte[] ret = dbService.get(AREA, key);
                if (logger.isTraceEnabled()) {
                    logger.trace("<~ LevelDbDataSource.get(): " + name + ", key: " + toHexString(key) +
", " + (ret == null ? "null" : ret.length));
                }
                return ret;
            }
        } finally {
            resetDbLock.readLock().unlock();
        }
    }
    @Override
    public void put(byte[] key, byte[] value) {
        resetDbLock.readLock().lock();
        try {
            if (logger.isTraceEnabled()) {
                logger.trace("~> LevelDbDataSource.put(): " + name + ", key: " + toHexString(key) + ", "
+ (value == null ? "null" : value.length));
            }
            dbService.put(AREA, key, value);
            if (logger.isTraceEnabled()) {
                logger.trace("<~ LevelDbDataSource.put(): " + name + ", key: " + toHexString(key) + ", "
+ (value == null ? "null" : value.length));
            }
        } finally {
            resetDbLock.readLock().unlock();
        }
    }

    @Override
    public void delete(byte[] key) {
        resetDbLock.readLock().lock();
        try {
            if (logger.isTraceEnabled()) {
                logger.trace("~> LevelDbDataSource.delete(): " + name + ", key: " + toHexString(key));
            }
            dbService.delete(AREA, key);
            if (logger.isTraceEnabled()) {
                logger.trace("<~ LevelDbDataSource.delete(): " + name + ", key: " + toHexString(key));
            }
        } finally {
            resetDbLock.readLock().unlock();
        }
    }

    @Override
    public Set<byte[]> keys() {
        return null;
    }

    private void updateBatchInternal(Map<byte[], byte[]> rows) {
        BatchOperation batchOperation = dbService.createWriteBatch(AREA);
        for (Map.Entry<byte[], byte[]> entry : rows.entrySet()) {
```

```
        if (entry.getValue() == null) {
            batchOperation.delete(entry.getKey());
        } else {
            batchOperation.put(entry.getKey(), entry.getValue());
        }
    }
    batchOperation.executeBatch();
}

@Override
public void updateBatch(Map<byte[], byte[]> rows) {
    resetDbLock.readLock().lock();
    try {
        if (logger.isTraceEnabled()) {
            logger.trace("~> LevelDbDataSource.updateBatch(): " + name + ", " + rows.size());
        }
        try {
            updateBatchInternal(rows);
            if (logger.isTraceEnabled()) {
                logger.trace("<~ LevelDbDataSource.updateBatch(): " + name + ", " + rows.size());
            }
        } catch (Exception e) {
            logger.error("Error, retrying one more time...", e);
            // try one more time
            try {
                updateBatchInternal(rows);
                if (logger.isTraceEnabled()) {
                    logger.trace("<~ LevelDbDataSource.updateBatch(): " + name + ", " + rows.size());
                }
            } catch (Exception e1) {
                logger.error("Error", e);
                throw new RuntimeException(e);
            }
        }
    } finally {
        resetDbLock.readLock().unlock();
    }
}

@Override
public boolean flush() {
    return false;
    }

    @Override
    public void close() {
    }

}
```

2:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\main\java\org\ethereum\datasource\MemSizeEstimator.java

```
 */
public interface MemSizeEstimator<E> {

    long estimateSize(E e);

    /**
     * byte[] type size estimator
     */
    MemSizeEstimator<byte[]> ByteArrayEstimator = bytes -> {
        return bytes == null ? 0 : bytes.length + 16; // 4 - compressed ref size, 12 - Object header
    };


}
```

3:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\main\java\org\ethereum\datasource\MultiCache.java

```
 * via create() method
 * <p>
 * When flushing children, each child is just flushed if it has backing Source or the whole
 * child cache is put to the MultiCache backing source
 * <p>
 * The primary goal if for caching contract storages in the child repositories (tracks)
 * <p>
 * Created by Anton Nashatyrev on 07.10.2016.
 */
public abstract class MultiCache<V extends CachedSource> extends
ReadWriteCache.BytesKey<V> {

    public MultiCache(Source<byte[], V> src) {
        super(src, WriteCache.CacheType.SIMPLE);
    }
```

```
/**
 * When a child cache is not found in the local cache it is looked up in the backing Source
 * Based on this child backing cache (or null if not found) the new local cache is created
 * via create() method
 */
@Override
public synchronized V get(byte[] key) {
    AbstractCachedSource.Entry<V> ownCacheEntry = getCached(key);
    V ownCache = ownCacheEntry == null ? null : ownCacheEntry.value();
    if (ownCache == null) {
        V v = getSource() != null ? super.get(key) : null;
        ownCache = create(key, v);
        put(key, ownCache);
    }
    return ownCache;
}

/**
 * each child is just flushed if it has backing Source or the whole
 * child cache is put to the MultiCache backing source
 */
@Override
public synchronized boolean flushImpl() {
    boolean ret = false;
    for (byte[] key : writeCache.getModified()) {
        V value = super.get(key);
        if (value == null) {
            // cache was deleted
            ret |= flushChild(key, value);
            if (getSource() != null) {
                getSource().delete(key);
            }
        } else if (value.getSource() != null) {
            ret |= flushChild(key, value);
        } else {
            getSource().put(key, value);
            ret = true;
        }
    }
    return ret;
}
```

```
    /**
     * Is invoked to flush child cache if it has backing Source
     * Some additional tasks may be performed by subclasses here
     */
    protected boolean flushChild(byte[] key, V childCache) {
        return childCache != null ? childCache.flush() : true;
    }

    /**
     * Creates a local child cache instance based on the child cache instance
     * (or null) from the MultiCache backing Source
     */
    protected abstract V create(byte[] key, V srcCache);
}
```

4:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\main\java\org\ethereum\datasource\NodeKeyCompositor.java

```
 * This mechanism is a part of flat storage source which is free from reference counting
 *
 * @author Mikhail Kalinin
 * @see CommonConfig#trieNodeSource()
 * @see RepositoryRoot#RepositoryRoot(Source, byte[])
 * @since 05.12.2017
 */
public class NodeKeyCompositor implements Serializer<byte[], byte[]> {

    public static final int HASH_LEN = 32;
    public static final int PREFIX_BYTES = 16;
    private byte[] addrHash;

    public NodeKeyCompositor(byte[] addrOrHash) {
        this.addrHash = addrHash(addrOrHash);
    }

    @Override
    public byte[] serialize(byte[] key) {
        return composeInner(key, addrHash);
    }

    @Override
    public byte[] deserialize(byte[] stream) {
        return stream;
    }

    public static byte[] compose(byte[] key, byte[] addrOrHash) {
        return composeInner(key, addrHash(addrOrHash));
    }

    private static byte[] composeInner(byte[] key, byte[] addrHash) {

        validateKey(key);

        byte[] derivative = new byte[key.length];
        arraycopy(key, 0, derivative, 0, PREFIX_BYTES);
        arraycopy(addrHash, 0, derivative, PREFIX_BYTES, PREFIX_BYTES);

        return derivative;
    }

    private static void validateKey(byte[] key) {
        if (key.length != HASH_LEN) {
            throw new IllegalArgumentException("Key is not a hash code");
        }
    }

    private static byte[] addrHash(byte[] addrOrHash) {
        return addrOrHash.length == HASH_LEN ? addrOrHash : sha3(addrOrHash);
    }
}
```

5:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\main\java\org\ethereum\datasource\NoDeleteSource.java

```
 * <p>
 * Created by Anton Nashatyrev on 03.11.2016.
 */
public class NoDeleteSource<Key, Value> extends AbstractChainedSource<Key, Value, Key,
Value> {

    public NoDeleteSource(Source<Key, Value> src) {
        super(src);
        setFlushSource(true);
    }
    @Override
    public void delete(Key key) {
    }

    @Override
    public void put(Key key, Value val) {
        if (val != null) {
            getSource().put(key, val);
        }
    }

    @Override
    public Value get(Key key) {
        return getSource().get(key);
    }

    @Override
    protected boolean flushImpl() {
        return false;
    }
}
```

6:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\main\java\org\ethereum\datasource\ObjectDataSource.java

```
 * <p>
 * Created by Anton Nashatyrev on 06.12.2016.
 */
public class ObjectDataSource<V> extends SourceChainBox<byte[], V, byte[], byte[]> {
    ReadCache<byte[], V> cache;
    SourceCodec<byte[], V, byte[], byte[]> codec;
    Source<byte[], byte[]> byteSource;

    /**
     * Creates new instance
     *
     * @param byteSource       baking store
     * @param serializer       for encode/decode byte[] <=> V
     * @param readCacheEntries number of entries to cache
     */
    public ObjectDataSource(Source<byte[], byte[]> byteSource, Serializer<V, byte[]> serializer, int
readCacheEntries) {
        super(byteSource);
        this.byteSource = byteSource;
        add(codec = new SourceCodec<>(byteSource, new Serializers.Identity<byte[]>(), serializer));
        if (readCacheEntries > 0) {
            add(cache = new ReadCache.BytesKey<>(codec).withMaxCapacity(readCacheEntries));
        }
    }
}
```

7:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\main\java\org\ethereum\datasource\PrefixLookupSource.java

```
 * Other operations are simply propagated to backing {@link DbSource}.
 *
 * @author Mikhail Kalinin
 * @since 01.12.2017
 */
public class PrefixLookupSource<V> implements Source<byte[], V> {

    // prefix length in bytes
    private int prefixBytes;
    private DbSource<V> source;

    public PrefixLookupSource(DbSource<V> source, int prefixBytes) {
        this.source = source;
        this.prefixBytes = prefixBytes;
    }

    @Override
    public V get(byte[] key) {
        return source.prefixLookup(key, prefixBytes);
    }

    @Override
    public void put(byte[] key, V val) {
        source.put(key, val);
    }

    @Override
    public void delete(byte[] key) {
        source.delete(key);
    }

    @Override
    public boolean flush() {
        return source.flush();
    }
}
```

8:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\main\java\org\ethereum\datasource\QuotientFilter.java

```
OTHER
    LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
    FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
    DEALINGS IN THE SOFTWARE.
*/
package org.ethereum.datasource;

import com.google.common.base.Preconditions;
import com.google.common.math.LongMath;
import com.google.common.primitives.Ints;

import java.util.Arrays;
import java.util.Iterator;
import java.util.NoSuchElementException;

import static java.lang.System.arraycopy;
import static java.util.Arrays.copyOfRange;
import static org.ethereum.util.ByteUtil.byteArrayToLong;
import static org.ethereum.util.ByteUtil.longToBytes;

//import net.jpountz.xxhash.XXHashFactory;

public class QuotientFilter implements Iterable<Long> {
  //    static final XXHashFactory hashFactory = XXHashFactory.fastestInstance();
  byte QUOTIENT_BITS;
  byte REMAINDER_BITS;
  byte ELEMENT_BITS;
  long INDEX_MASK;
  long REMAINDER_MASK;
  long ELEMENT_MASK;
  long MAX_SIZE;
  long MAX_INSERTIONS;
  int MAX_DUPLICATES = 2;
  long[] table;

  boolean overflowed = false;
  long entries;

  public static QuotientFilter deserialize(byte[] bytes) {
    QuotientFilter ret = new QuotientFilter();
    ret.QUOTIENT_BITS = bytes[0];
    ret.REMAINDER_BITS = bytes[1];
    ret.ELEMENT_BITS = bytes[2];
    ret.INDEX_MASK = byteArrayToLong(copyOfRange(bytes, 3, 11));
    ret.REMAINDER_MASK = byteArrayToLong(copyOfRange(bytes, 11, 19));
    ret.ELEMENT_MASK = byteArrayToLong(copyOfRange(bytes, 19, 27));
    ret.MAX_SIZE = byteArrayToLong(copyOfRange(bytes, 27, 35));
    ret.MAX_INSERTIONS = byteArrayToLong(copyOfRange(bytes, 35, 43));
    ret.overflowed = bytes[43] > 0;
    ret.entries = byteArrayToLong(copyOfRange(bytes, 44, 52));
    ret.table = new long[(bytes.length - 52) / 8];
    for (int i = 0; i < ret.table.length; i++) {
        ret.table[i] = byteArrayToLong(copyOfRange(bytes, 52 + i * 8, 52 + i * 8 + 8));
    }
    return ret;
}

public synchronized byte[] serialize() {
    byte[] ret = new byte[1 + 1 + 1 + 8 + 8 + 8 + 8 + 8 + 1 + 8 + table.length * 8];
    ret[0] = QUOTIENT_BITS;
    ret[1] = REMAINDER_BITS;
    ret[2] = ELEMENT_BITS;
    arraycopy(longToBytes(INDEX_MASK), 0, ret, 3, 8);
    arraycopy(longToBytes(REMAINDER_MASK), 0, ret, 11, 8);
    arraycopy(longToBytes(ELEMENT_MASK), 0, ret, 19, 8);
    arraycopy(longToBytes(MAX_SIZE), 0, ret, 27, 8);
    arraycopy(longToBytes(MAX_INSERTIONS), 0, ret, 35, 8);
    ret[43] = (byte) (overflowed ? 1 : 0);
    arraycopy(longToBytes(entries), 0, ret, 44, 8);
    for (int i = 0; i < table.length; i++) {
        arraycopy(longToBytes(table[i]), 0, ret, 52 + i * 8, 8);
    }
    return ret;
}

static long LOW_MASK(long n) {
    return (1L << n) - 1L;
}

static int TABLE_SIZE(int quotientBits, int remainderBits) {
    long bits = (1L << quotientBits) * (remainderBits + 3);
    long longs = bits / 64;
    return Ints.checkedCast((bits % 64) > 0 ? (longs + 1) : longs);
}
```

```
    static int bitsForNumElementsWithLoadFactor(long numElements) {
        if (numElements == 0) {
            return 1;
        }

        int candidateBits = Long.bitCount(numElements) == 1 ?
                Math.max(1, Long.numberOfTrailingZeros(numElements)) :
                Long.numberOfTrailingZeros(Long.highestOneBit(numElements) << 1L);

        //May need an extra bit due to load factor
        if (((long) (LongMath.pow(2, candidateBits) * 0.75)) < numElements) {
            candidateBits++;
        }

        return candidateBits;
    }

    public static QuotientFilter create(long largestNumberOfElements, long startingElements) {
        Preconditions.checkArgument(largestNumberOfElements >= startingElements);
        Preconditions.checkArgument(startingElements > 0);
        Preconditions.checkArgument(largestNumberOfElements > 0);

        /**
         * The way sizing a quotient filter works is that the quotient bits + remainder bits
         * is the maximum number of elements the filter can store before it runs out of fingerprint bits
         * and can no longer be resized.
         */
        int quotientBits = bitsForNumElementsWithLoadFactor(startingElements);
        int remainderBits = bitsForNumElementsWithLoadFactor(largestNumberOfElements);

        //I am pretty sure that even when completely full you want a non-zero number of remainder
bits
        //This also gives some emergency slack where even if you guess largest number of elements
wrong it will
        //keep working even if you are very wrong.
        remainderBits += 8;
        remainderBits -= quotientBits;

        return new QuotientFilter(quotientBits, remainderBits);
    }
```

```
private QuotientFilter() {
}

public QuotientFilter(int quotientBits, int remainderBits) {
    Preconditions.checkArgument(quotientBits > 0);
    Preconditions.checkArgument(remainderBits > 0);
    Preconditions.checkArgument(quotientBits + remainderBits <= 64);

    QUOTIENT_BITS = (byte) quotientBits;
    REMAINDER_BITS = (byte) remainderBits;
    ELEMENT_BITS = (byte) (REMAINDER_BITS + 3);
    INDEX_MASK = LOW_MASK(QUOTIENT_BITS);
    REMAINDER_MASK = LOW_MASK(REMAINDER_BITS);
    ELEMENT_MASK = LOW_MASK(ELEMENT_BITS);
    MAX_SIZE = 1L << QUOTIENT_BITS;
    MAX_INSERTIONS = (long) (MAX_SIZE * .75);
    table = new long[TABLE_SIZE(QUOTIENT_BITS, REMAINDER_BITS)];
    entries = 0;
}

public QuotientFilter withMaxDuplicates(int maxDuplicates) {
    MAX_DUPLICATES = maxDuplicates;
    return this;
}

/* Return QF[idx] in the lower bits. */
long getElement(long idx) {
    long elt = 0;
    long bitpos = ELEMENT_BITS * idx;
    int tabpos = Ints.checkedCast(bitpos / 64);
    long slotpos = bitpos % 64;
    long spillbits = (slotpos + ELEMENT_BITS) - 64;
    elt = (table[tabpos] >>> slotpos) & ELEMENT_MASK;
    if (spillbits > 0) {
        ++tabpos;
        long x = table[tabpos] & LOW_MASK(spillbits);
        elt |= x << (ELEMENT_BITS - spillbits);
    }
    return elt;
}

/* Store the lower bits of elt into QF[idx]. */
```

```
void setElement(long idx, long elt) {
    long bitpos = ELEMENT_BITS * idx;
    int tabpos = Ints.checkedCast(bitpos / 64);
    long slotpos = bitpos % 64;
    long spillbits = (slotpos + ELEMENT_BITS) - 64;
    elt &= ELEMENT_MASK;
    table[tabpos] &= ~(ELEMENT_MASK << slotpos);
    table[tabpos] |= elt << slotpos;
    if (spillbits > 0) {
        ++tabpos;
        table[tabpos] &= ~LOW_MASK(spillbits);
        table[tabpos] |= elt >>> (ELEMENT_BITS - spillbits);
    }
}

long incrementIndex(long idx) {
    return (idx + 1) & INDEX_MASK;
}

long decrementIndex(long idx) {
    return (idx - 1) & INDEX_MASK;
}

static boolean isElementOccupied(long elt) {
    return (elt & 1) != 0;
}

static long setElementOccupied(long elt) {
    return elt | 1;
}

static long clearElementOccupied(long elt) {
    return elt & ~1;
}

static boolean isElementContinuation(long elt) {
    return (elt & 2) != 0;
}

static long setElementContinuation(long elt) {
    return elt | 2;
}
```

```
static long clearElementContinuation(long elt) {
    return elt & ~2;
}

static boolean isElementShifted(long elt) {
    return (elt & 4) != 0;
}

static long setElementShifted(long elt) {
    return elt | 4;
}

static long clearElementShifted(long elt) {
    return elt & ~4;
}

static long getElementRemainder(long elt) {
    return elt >>> 3;
}

static boolean isElementEmpty(long elt) {
    return (elt & 7) == 0;
}

static boolean isElementClusterStart(long elt) {
    return isElementOccupied(elt) & !isElementContinuation(elt) & !isElementShifted(elt);
}

static boolean isElementRunStart(long elt) {
    return !isElementContinuation(elt) & (isElementOccupied(elt) | isElementShifted(elt));
}

long hashToQuotient(long hash) {
    return (hash >>> REMAINDER_BITS) & INDEX_MASK;
}

long hashToRemainder(long hash) {
    return hash & REMAINDER_MASK;
}

/* Find the start index of the run for fq (given that the run exists). */
```

```
long findRunIndex(long fq) {
    /* Find the start of the cluster. */
    long b = fq;
    while (isElementShifted(getElement(b))) {
        b = decrementIndex(b);
    }

    /* Find the start of the run for fq. */
    long s = b;
    while (b != fq) {
        do {
            s = incrementIndex(s);
        }
        while (isElementContinuation(getElement(s)));

        do {
            b = incrementIndex(b);
        }
        while (!isElementOccupied(getElement(b)));
    }
    return s;
}

/* Insert elt into QF[s], shifting over elements as necessary. */
void insertInto(long s, long elt) {
    long prev;
    long curr = elt;
    boolean empty;

    do {
        prev = getElement(s);
        empty = isElementEmpty(prev);
        if (!empty) {
            /* Fix up `is_shifted' and `is_occupied'. */
            prev = setElementShifted(prev);
            if (isElementOccupied(prev)) {
                curr = setElementOccupied(curr);
                prev = clearElementOccupied(prev);
            }
        }
        setElement(s, curr);
        curr = prev;
```

```
            s = incrementIndex(s);
        }
        while (!empty);
    }

    public boolean overflowed() {
        return overflowed;
    }

//    public void insert(byte[] data)
//    {
//        insert(data, 0, data.length);
//    }
//
//    public void insert(byte[] data, int offset, int length) {
//        insert(hashFactory.hash64().hash(data, offset, length, 0));
//    }

    protected long hash(byte[] bytes) {
        return (bytes[0] & 0xFFL) << 56 |
                (bytes[1] & 0xFFL) << 48 |
                (bytes[2] & 0xFFL) << 40 |
                (bytes[3] & 0xFFL) << 32 |
                (bytes[4] & 0xFFL) << 24 |
                (bytes[5] & 0xFFL) << 16 |
                (bytes[6] & 0xFFL) << 8 |
                (bytes[7] & 0xFFL);
    }

    public synchronized void insert(byte[] hash) {
        insert(hash(hash));
    }

    public synchronized void insert(long hash) {
        if (maybeContainsXTimes(hash, MAX_DUPLICATES)) {
            return;
        }
        if (entries >= MAX_INSERTIONS | overflowed) {
            //Can't safely process an after overflow
            //Only a buggy program would attempt it
            if (overflowed) {
                throw new OverflowedError();
    }

    //Can still resize if we have enough remainder bits
    if (REMAINDER_BITS > 1) {
        selfResizeDouble();
    } else {
        //The filter can't accept more inserts and is effectively broken
        overflowed = true;
        throw new OverflowedError();
    }
}

long fq = hashToQuotient(hash);
long fr = hashToRemainder(hash);
long T_fq = getElement(fq);
long entry = (fr << 3) & ~7;

/* Special-case filling canonical slots to simplify insert_into(). */
if (isElementEmpty(T_fq)) {
    setElement(fq, setElementOccupied(entry));
    ++entries;
    return;
}

if (!isElementOccupied(T_fq)) {
    setElement(fq, setElementOccupied(T_fq));
}

long start = findRunIndex(fq);
long s = start;

if (isElementOccupied(T_fq)) {
    /* Move the cursor to the insert position in the fq run. */
    do {
        long rem = getElementRemainder(getElement(s));
        if (rem >= fr) {
            break;
        }
        s = incrementIndex(s);
    }
    while (isElementContinuation(getElement(s)));
```

```
        if (s == start) {
            /* The old start-of-run becomes a continuation. */
            long old_head = getElement(start);
            setElement(start, setElementContinuation(old_head));
        } else {
            /* The new element becomes a continuation. */
            entry = setElementContinuation(entry);
        }
    }

    /* Set the shifted bit if we can't use the canonical slot. */
    if (s != fq) {
        entry = setElementShifted(entry);
    }

    insertInto(s, entry);
    ++entries;
    return;
}

private void selfResizeDouble() {
    QuotientFilter qf = resize(MAX_INSERTIONS * 2);
    QUOTIENT_BITS = qf.QUOTIENT_BITS;
    REMAINDER_BITS = qf.REMAINDER_BITS;
    ELEMENT_BITS = qf.ELEMENT_BITS;
    INDEX_MASK = qf.INDEX_MASK;
    REMAINDER_MASK = qf.REMAINDER_MASK;
    ELEMENT_MASK = qf.ELEMENT_MASK;
    MAX_SIZE = qf.MAX_SIZE;
    MAX_INSERTIONS = qf.MAX_INSERTIONS;
    table = qf.table;
    if (qf.entries != entries) {
        throw new AssertionError();
    }
}

public boolean maybeContains(byte[] hash) {
    return maybeContains(hash(hash));
}

public synchronized boolean maybeContains(long hash) {
    if (overflowed) {
        //Can't check for existence after overflow occurred
        //and things are missing
        throw new OverflowedError();
    }

    long fq = hashToQuotient(hash);
    long fr = hashToRemainder(hash);
    long T_fq = getElement(fq);

    /* If this quotient has no run, give up. */
    if (!isElementOccupied(T_fq)) {
        return false;
    }

    /* Scan the sorted run for the target remainder. */
    long s = findRunIndex(fq);
    do {
        long rem = getElementRemainder(getElement(s));
        if (rem == fr) {
            return true;
        } else if (rem > fr) {
            return false;
        }
        s = incrementIndex(s);
    }
    while (isElementContinuation(getElement(s)));
    return false;
}

public synchronized boolean maybeContainsXTimes(long hash, int num) {
    if (overflowed) {
        //Can't check for existence after overflow occurred
        //and things are missing
        throw new OverflowedError();
    }

    long fq = hashToQuotient(hash);
    long fr = hashToRemainder(hash);
    long T_fq = getElement(fq);

    /* If this quotient has no run, give up. */
    if (!isElementOccupied(T_fq)) {
```

```
        return false;
    }

    /* Scan the sorted run for the target remainder. */
    long s = findRunIndex(fq);
    int counter = 0;
    do {
        long rem = getElementRemainder(getElement(s));
        if (rem == fr) {
            counter++;
        } else if (rem > fr) {
            break;
        }
        s = incrementIndex(s);
    }
    while (isElementContinuation(getElement(s)));
    return counter >= num;
}

/* Remove the entry in QF[s] and slide the rest of the cluster forward. */
void deleteEntry(long s, long quot) {
    long next;
    long curr = getElement(s);
    long sp = incrementIndex(s);
    long orig = s;

    /*
     * FIXME(vsk): This loop looks ugly. Rewrite.
     */
    while (true) {
        next = getElement(sp);
        boolean curr_occupied = isElementOccupied(curr);

        if (isElementEmpty(next) | isElementClusterStart(next) | sp == orig) {
            setElement(s, 0);
            return;
        } else {
            /* Fix entries which slide into canonical slots. */
            long updated_next = next;
            if (isElementRunStart(next)) {
                do {
                    quot = incrementIndex(quot);
                }
                while (!isElementOccupied(getElement(quot)));

                if (curr_occupied && quot == s) {
                    updated_next = clearElementShifted(next);
                }
            }

            setElement(s, curr_occupied ?
                    setElementOccupied(updated_next) :
                    clearElementOccupied(updated_next));
            s = sp;
            sp = incrementIndex(sp);
            curr = next;
        }
    }
}

public void remove(byte[] hash) {
    remove(hash(hash));
}

public synchronized void remove(long hash) {
    if (maybeContainsXTimes(hash, MAX_DUPLICATES)) {
        return;
    }
    //Can't safely process a remove after overflow
    //Only a buggy program would attempt it
    if (overflowed) {
        throw new OverflowedError();
    }

    long fq = hashToQuotient(hash);
    long fr = hashToRemainder(hash);
    long T_fq = getElement(fq);

    if (!isElementOccupied(T_fq) | entries == 0) {
        //If you remove things that don't exist it's possible you will clobber
        //somethign on a collision, your program is buggy
        throw new NoSuchElementError();
    }
```

```
long start = findRunIndex(fq);
long s = start;
long rem;

/* Find the offending table index (or give up). */
do {
    rem = getElementRemainder(getElement(s));
    if (rem == fr) {
        break;
    } else if (rem > fr) {
        return;
    }
    s = incrementIndex(s);
} while (isElementContinuation(getElement(s)));
if (rem != fr) {
    //If you remove things that don't exist it's possible you will clobber
    //somethign on a collision, your program is buggy
    throw new NoSuchElementError();
}

long kill = (s == fq) ? T_fq : getElement(s);
boolean replace_run_start = isElementRunStart(kill);

/* If we're deleting the last entry in a run, clear `is_occupied'. */
if (isElementRunStart(kill)) {
    long next = getElement(incrementIndex(s));
    if (!isElementContinuation(next)) {
        T_fq = clearElementOccupied(T_fq);
        setElement(fq, T_fq);
    }
}

deleteEntry(s, fq);

if (replace_run_start) {
    long next = getElement(s);
    long updated_next = next;
    if (isElementContinuation(next)) {
        /* The new start-of-run is no longer a continuation. */
        updated_next = clearElementContinuation(next);
    }
    if (s == fq && isElementRunStart(updated_next)) {
```

```
                /* The new start-of-run is in the canonical slot. */
                updated_next = clearElementShifted(updated_next);
            }
            if (updated_next != next) {
                setElement(s, updated_next);
            }
        }

        --entries;
    }

//    public static QuotientFilter merge(Collection<QuotientFilter> filters) {
//        if (filters.stream().map(filter -> filter.REMAINDER_BITS +
filter.QUOTIENT_BITS).distinct().count() != 1) {
//            throw new IllegalArgumentException("All filters must have the same size fingerprint");
//        }
//
//        long totalEntries = filters.stream().collect(Collectors.summingLong(filter -> filter.entries));
//        int requiredQuotientBits = bitsForNumElementsWithLoadFactor(totalEntries);
//        int fingerprintBits = filters.iterator().next().QUOTIENT_BITS +
filters.iterator().next().REMAINDER_BITS;
//        int remainderBits = fingerprintBits - requiredQuotientBits;
//
//        if (remainderBits < 1) {
//            throw new IllegalArgumentException("Impossible to merge not enough fingerprint bits");
//        }
//
//        QuotientFilter resultFilter = new QuotientFilter(requiredQuotientBits, remainderBits);
//
//        Iterable<QFIterator> iterators = (Iterable) filters.stream().map(filter ->
filter.iterator()).collect(Collectors.toList());
//        Iterator<Long> mergeQFIterator = Iterators.mergeSorted(iterators, Ordering.natural());
//        while (mergeQFIterator.hasNext()) {
//            resultFilter.insert(mergeQFIterator.next());
//        }
//        return resultFilter;
//    }

//    public QuotientFilter merge(QuotientFilter other) {
//        return merge(ImmutableList.of(this, other));
//    }
//
```

```
//    public QuotientFilter merge(QuotientFilter... filters) {
//        return merge(Arrays.asList(filters));
//    }

    /*
     * Resizes the filter return a filter with the same contents and space for the minimum specified
number
     * of entries. This may allocate a new filter or return the existing filter.
     */
    public QuotientFilter resize(long minimumEntries) {
        if (minimumEntries <= MAX_INSERTIONS) {
            return this;
        }

        int newQuotientBits = bitsForNumElementsWithLoadFactor(minimumEntries);
        int newRemainderBits = QUOTIENT_BITS + REMAINDER_BITS - newQuotientBits;

        if (newRemainderBits < 1) {
            throw new IllegalArgumentException("Not enough fingerprint bits to resize");
        }

        QuotientFilter qf = new QuotientFilter(newQuotientBits, newRemainderBits);
        QFIterator i = new QFIterator();
        while (i.hasNext()) {
            qf.insert(i.nextPrimitive());
        }
        return qf;
    }

    public int getAllocatedBytes() {
        return table.length << 3;
    }

    public void clear() {
        entries = 0;
        Arrays.fill(table, 0L);
    }

    @Override
    public QFIterator iterator() {
        return new QFIterator();
    }
```

```
class QFIterator implements LongIterator {
    long index;
    long quotient;
    long visited;

    QFIterator() {
        /* Mark the iterator as done. */
        visited = entries;

        if (entries == 0) {
            return;
        }

        /* Find the start of a cluster. */
        long start;
        for (start = 0; start < MAX_SIZE; ++start) {
            if (isElementClusterStart(getElement(start))) {
                break;
            }
        }

        visited = 0;
        index = start;
    }

    @Override
    public boolean hasNext() {
        return entries != visited;
    }

    @Override
    public Long next() {
        return nextPrimitive();
    }

    @Override
    public void remove() {

    }

    @Override
```

```
        public long nextPrimitive() {
            while (hasNext()) {
                long elt = getElement(index);

                /* Keep track of the current run. */
                if (isElementClusterStart(elt)) {
                    quotient = index;
                } else {
                    if (isElementRunStart(elt)) {
                        long quot = quotient;
                        do {
                            quot = incrementIndex(quot);
                        }
                        while (!isElementOccupied(getElement(quot)));
                        quotient = quot;
                    }
                }

                index = incrementIndex(index);

                if (!isElementEmpty(elt)) {
                    long quot = quotient;
                    long rem = getElementRemainder(elt);
                    long hash = (quot << REMAINDER_BITS) | rem;
                    ++visited;
                    return hash;
                }
            }

            throw new NoSuchElementException();
        }
    }

//    @Override
//    public String toString() {
//        StringBuilder sb = new StringBuilder();
//
//        int pad = ((int) (Math.ceil(QUOTIENT_BITS / Math.log(10.0)))) + 1;
//
//        for (int i = 0; i < pad; ++i) {
//            sb.append(' ');
//        }
```

```
//
//        sb.append(String.format("| is_shifted | is_continuation | is_occupied | remainder"
//                + " nel=%d\n", entries));
//
//        for (long idx = 0; idx < MAX_SIZE; ++idx) {
//            String idxString = Long.toString(idx);
//            sb.append(idx);
//
//            int fillspace = pad - idxString.length();
//            for (int i = 0; i < fillspace; ++i) {
//                sb.append(' ');
//            }
//            sb.append("| ");
//
//            long elt = getElement(idx);
//            sb.append(String.format("%d          | ", isElementShifted(elt) == false ? 0 : 1));
//            sb.append(String.format("%d               | ", isElementContinuation(elt) == false ? 0 : 1));
//            sb.append(String.format("%d           | ", isElementOccupied(elt) == false ? 0 : 1));
//            sb.append(getElementRemainder(elt)).append(System.lineSeparator());
//        }
//        return sb.toString();
//    }

    public class OverflowedError extends AssertionError {

    }

    public class NoSuchElementError extends AssertionError {

    }

    public interface LongIterator extends Iterator<Long> {


        long nextPrimitive();

        @Override
        Long next();
    }
}
```

9:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-

```
vm\src\main\java\org\ethereum\datasource\ReadCache.java
import java.util.Collection;
import java.util.Collections;
import java.util.HashMap;
import java.util.Map;

/**
 * Caches entries get/updated and use LRU algo to purge them if the number
 * of entries exceeds threshold.
 * <p>
 * This implementation could extended to estimate cached data size for
 * more accurate size restriction, but if entries are more or less
 * of the same size the entries count would be good enough
 * <p>
 * Another implementation idea is heap sensitive read cache based on
 * SoftReferences, when the cache occupies all the available heap
 * but get shrink when low heap
 * <p>
 * Created by Anton Nashatyrev on 05.10.2016.
 */
public class ReadCache<Key, Value> extends AbstractCachedSource<Key, Value> {

    private final Value NULL = (Value) new Object();

    private Map<Key, Value> cache;
    private boolean byteKeyMap;

    public ReadCache(Source<Key, Value> src) {
        super(src);
        withCache(new HashMap<Key, Value>());
    }

    /**
     * Installs the specific cache Map implementation
     */
    public ReadCache<Key, Value> withCache(Map<Key, Value> cache) {
        byteKeyMap = cache instanceof ByteArrayMap;
        this.cache = Collections.synchronizedMap(cache);
        return this;
    }

    /**
     * Sets the max number of entries to cache
     */
    public ReadCache<Key, Value> withMaxCapacity(int maxCapacity) {
        return withCache(new LRUMap<Key, Value>(maxCapacity) {
            @Override
            protected boolean removeLRU(LinkEntry<Key, Value> entry) {
                cacheRemoved(entry.getKey(), entry.getValue());
                return super.removeLRU(entry);
            }
        });
    }

    // the guard against incorrect Map implementation for byte[] keys
    private boolean checked = false;

    private void checkByteArrKey(Key key) {
        if (checked) {
            return;
        }

        if (key instanceof byte[]) {
            if (!byteKeyMap) {
                throw new RuntimeException("Wrong map/set for byte[] key");
            }
        }
        checked = true;
    }

    @Override
    public void put(Key key, Value val) {
        checkByteArrKey(key);
        if (val == null) {
            delete(key);
        } else {
            cache.put(key, val);
            cacheAdded(key, val);
            getSource().put(key, val);
        }
    }

    @Override
    public Value get(Key key) {
        checkByteArrKey(key);
        Value ret = cache.get(key);
        if (ret == NULL) {
            return null;
        }
        if (ret == null) {
            ret = getSource().get(key);
            cache.put(key, ret == null ? NULL : ret);
            cacheAdded(key, ret);
        }
        return ret;
    }

    @Override
    public void delete(Key key) {
        checkByteArrKey(key);
        Value value = cache.remove(key);
        cacheRemoved(key, value);
        getSource().delete(key);
    }

    @Override
    protected boolean flushImpl() {
        return false;
    }

    @Override
    public synchronized Collection<Key> getModified() {
        return Collections.emptyList();
    }

    @Override
    public boolean hasModified() {
        return false;
    }

    @Override
    public synchronized Entry<Value> getCached(Key key) {
        Value value = cache.get(key);
        return value == null ? null : new SimpleEntry<>(value == NULL ? null : value);
    }
```

```
    /**
     * Shortcut for ReadCache with byte[] keys. Also prevents accidental
     * usage of regular Map implementation (non byte[])
     */
    public static class BytesKey<V> extends ReadCache<byte[], V> implements
CachedSource.BytesKey<V> {

        public BytesKey(Source<byte[], V> src) {
            super(src);
            withCache(new ByteArrayMap<V>());
        }

        @Override
        public ReadCache.BytesKey<V> withMaxCapacity(int maxCapacity) {
            withCache(new ByteArrayMap<V>(new LRUMap<ByteArrayWrapper, V>(maxCapacity) {
                @Override
                protected boolean removeLRU(LinkEntry<ByteArrayWrapper, V> entry) {
                    cacheRemoved(entry.getKey().getData(), entry.getValue());
                    return super.removeLRU(entry);
                }
            }));
            return this;
        }
    }
}
```

10:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\main\java\org\ethereum\datasource\ReadWriteCache.java

```
 * <p>
 * Created by Anton Nashatyrev on 29.11.2016.
 */
public class ReadWriteCache<Key, Value>
        extends SourceChainBox<Key, Value, Key, Value>
        implements CachedSource<Key, Value> {

    protected ReadCache<Key, Value> readCache;
    protected WriteCache<Key, Value> writeCache;

    protected ReadWriteCache(Source<Key, Value> source) {
        super(source);
    }
    public ReadWriteCache(Source<Key, Value> src, WriteCache.CacheType cacheType) {
        super(src);
        add(writeCache = new WriteCache<>(src, cacheType));
        add(readCache = new ReadCache<>(writeCache));
        readCache.setFlushSource(true);
    }

    @Override
    public synchronized Collection<Key> getModified() {
        return writeCache.getModified();
    }

    @Override
    public boolean hasModified() {
        return writeCache.hasModified();
    }

    protected synchronized AbstractCachedSource.Entry<Value> getCached(Key key) {
        AbstractCachedSource.Entry<Value> v = readCache.getCached(key);
        if (v == null) {
            v = writeCache.getCached(key);
        }
        return v;
    }

    @Override
    public synchronized long estimateCacheSize() {
        return readCache.estimateCacheSize() + writeCache.estimateCacheSize();
    }

    public static class BytesKey<V> extends ReadWriteCache<byte[], V> {
        public BytesKey(Source<byte[], V> src, WriteCache.CacheType cacheType) {
            super(src);
            add(this.writeCache = new WriteCache.BytesKey<>(src, cacheType));
            add(this.readCache = new ReadCache.BytesKey<>(writeCache));
            readCache.setFlushSource(true);
        }
    }
}
```

11:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\main\java\org\ethereum\datasource\Serializer.java

```
 */
public interface Serializer<T, S> {
    /**
     * Converts T ==> S
     * Should correctly handle null parameter
     */
    S serialize(T object);

    /**
     * Converts S ==> T
     * Should correctly handle null parameter
     */
    T deserialize(S stream);
}
```

12:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\main\java\org\ethereum\datasource\Serializers.java

```
import org.ethereum.vm.DataWord;

/**
 * Collection of common Serializers
 * Created by Anton Nashatyrev on 08.11.2016.
 */
public class Serializers {

    /**
     * No conversion
     */
    public static class Identity<T> implements Serializer<T, T> {
        @Override
        public T serialize(T object) {
            return object;
        }

        @Override
        public T deserialize(T stream) {
            return stream;
        }
    }

    /**
     * Serializes/Deserializes AccountState instances from the State Trie (part of Ethereum spec)
     */
    public final static Serializer<AccountState, byte[]> AccountStateSerializer = new
Serializer<AccountState, byte[]>() {
        @Override
        public byte[] serialize(AccountState object) {
            return object.getEncoded();
        }

        @Override
        public AccountState deserialize(byte[] stream) {
            return stream == null || stream.length == 0 ? null : new AccountState(stream);
        }
    };

    /**
     * Contract storage key serializer
     */
    public final static Serializer<DataWord, byte[]> StorageKeySerializer = new
Serializer<DataWord, byte[]>() {
        @Override
        public byte[] serialize(DataWord object) {
            return object.getData();
        }

        @Override
        public DataWord deserialize(byte[] stream) {
            return DataWord.of(stream);
        }
    };

    /**
     * Contract storage value serializer (part of Ethereum spec)
     */
    public final static Serializer<DataWord, byte[]> StorageValueSerializer = new
Serializer<DataWord, byte[]>() {
        @Override
        public byte[] serialize(DataWord object) {
            return RLP.encodeElement(object.getNoLeadZeroesData());
        }

        @Override
        public DataWord deserialize(byte[] stream) {
            if (stream == null || stream.length == 0) {
                return null;
            }
            byte[] dataDecoded = RLP.decode2(stream).get(0).getRLPData();
            return DataWord.of(dataDecoded);
        }
    };

    /**
     * Trie node serializer (part of Ethereum spec)
     */
    public final static Serializer<Value, byte[]> TrieNodeSerializer = new Serializer<Value, byte[]>()
{
        @Override
        public byte[] serialize(Value object) {
            return object.asBytes();
        }

        @Override
        public Value deserialize(byte[] stream) {
            return new Value(stream);
        }
    };

    /**
     * Trie node serializer (part of Ethereum spec)
     */
    public final static Serializer<BlockHeader, byte[]> BlockHeaderSerializer = new
Serializer<BlockHeader, byte[]>() {
        @Override
        public byte[] serialize(BlockHeader object) {
            return object == null ? null : object.getEncoded();
        }

        @Override
        public BlockHeader deserialize(byte[] stream) {
            return stream == null ? null : new BlockHeader(stream);
        }
    };

    /**
     * AS IS serializer (doesn't change anything)
     */
    public final static Serializer<byte[], byte[]> AsIsSerializer = new Serializer<byte[], byte[]>() {
        @Override
        public byte[] serialize(byte[] object) {
            return object;
        }

        @Override
        public byte[] deserialize(byte[] stream) {
            return stream;
        }
    };
}
```

13:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\main\java\org\ethereum\datasource\Source.java

```
 */
public interface Source<K, V> {

    /**
     * Puts key-value pair into source
     */
    void put(K key, V val);

    /**
     * Gets a value by its key
     *
     * @return value or <null/> if no such key in the source
     */
    V get(K key);

    /**
     * Deletes the key-value pair from the source
     */
    void delete(K key);

    /**
     * If this source has underlying level source then all
     * changes collected in this source are flushed into the
     * underlying source.
     * The implementation may do 'cascading' flush, i.e. call
     * flush() on the underlying Source
     *
     * @return true if any changes we flushed, false if the underlying
     * Source didn't change
     */
    boolean flush();

}
```

14:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\main\java\org\ethereum\datasource\SourceChainBox.java

```
 * Represents a chain of Sources as a single Source
 * All calls to this Source are delegated to the last Source in the chain
 * On flush all Sources in chain are flushed in reverse order
 * <p>
 * Created by Anton Nashatyrev on 07.12.2016.
 */
public class SourceChainBox<Key, Value, SourceKey, SourceValue>
        extends AbstractChainedSource<Key, Value, SourceKey, SourceValue> {

    List<Source> chain = new ArrayList<>();
    Source<Key, Value> lastSource;

    public SourceChainBox(Source<SourceKey, SourceValue> source) {
        super(source);
    }

    /**
     * Adds next Source in the chain to the collection
     * Sources should be added from most bottom (connected to the backing Source)
     * All calls to the SourceChainBox will be delegated to the last added
     * Source
     */
    public void add(Source src) {
        chain.add(src);
        lastSource = src;
    }

    @Override
    public void put(Key key, Value val) {
        lastSource.put(key, val);
    }
    @Override
    public Value get(Key key) {
        return lastSource.get(key);
    }

    @Override
    public void delete(Key key) {
        lastSource.delete(key);
    }

//    @Override
//    public boolean flush() {
////        boolean ret = false;
////        for (int i = chain.size() - 1; i >= 0 ; i--) {
////            ret |= chain.get(i).flush();
////        }
//        return lastSource.flush();
//    }

    @Override
    protected boolean flushImpl() {
        return lastSource.flush();
    }
}
```

15:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\main\java\org\ethereum\datasource\SourceCodec.java

```
 * <p>
 * Created by Anton Nashatyrev on 03.11.2016.
 */
public class SourceCodec<Key, Value, SourceKey, SourceValue>
        extends AbstractChainedSource<Key, Value, SourceKey, SourceValue> {

    protected Serializer<Key, SourceKey> keySerializer;
    protected Serializer<Value, SourceValue> valSerializer;

    /**
     * Instantiates class
     *
     * @param src           Backing Source
     * @param keySerializer Key codec Key <=> SourceKey
     * @param valSerializer Value codec Value <=> SourceValue
     */
    public SourceCodec(Source<SourceKey, SourceValue> src, Serializer<Key, SourceKey>
keySerializer, Serializer<Value, SourceValue> valSerializer) {
        super(src);
        this.keySerializer = keySerializer;
        this.valSerializer = valSerializer;
        setFlushSource(true);
    }

    @Override
    public void put(Key key, Value val) {
        getSource().put(keySerializer.serialize(key), valSerializer.serialize(val));
    }

    @Override
    public Value get(Key key) {
        return valSerializer.deserialize(getSource().get(keySerializer.serialize(key)));
    }

    @Override
    public void delete(Key key) {
        getSource().delete(keySerializer.serialize(key));
    }

    @Override
    public boolean flushImpl() {
        return false;
    }

    /**
     * Shortcut class when only value conversion is required
     */
    public static class ValueOnly<Key, Value, SourceValue> extends SourceCodec<Key, Value,
Key, SourceValue> {
        public ValueOnly(Source<Key, SourceValue> src, Serializer<Value, SourceValue>
valSerializer) {
            super(src, new Serializers.Identity<Key>(), valSerializer);
        }
    }

    /**
     * Shortcut class when only key conversion is required
     */
    public static class KeyOnly<Key, Value, SourceKey> extends SourceCodec<Key, Value,
SourceKey, Value> {
        public KeyOnly(Source<SourceKey, Value> src, Serializer<Key, SourceKey> keySerializer) {
            super(src, keySerializer, new Serializers.Identity<Value>());
        }
    }

    /**
     * Shortcut class when only value conversion is required and keys are of byte[] type
     */
    public static class BytesKey<Value, SourceValue> extends ValueOnly<byte[], Value,
SourceValue> {
        public BytesKey(Source<byte[], SourceValue> src, Serializer<Value, SourceValue>
valSerializer) {
            super(src, valSerializer);
        }
    }
}
```

16:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\main\java\org\ethereum\datasource\WriteCache.java

```
import java.util.Collection;
import java.util.HashMap;
import java.util.Map;

/**
 * Collects changes and propagate them to the backing Source when flush() is called
 * <p>
 * The WriteCache can be of two types: Simple and Counting
 * <p>
 * Simple acts as regular Map: single and double adding of the same entry has the same effect
 * Source entries (key/value pairs) may have arbitrary nature
 * <p>
 * Counting counts the resulting number of inserts (+1) and deletes (-1) and when flushed
 * does the resulting number of inserts (if sum > 0) or deletes (if sum < 0)
 * Counting Source acts like {@link HashedKeySource} and makes sense only for data
 * where a single key always corresponds to a single value
 * Counting cache normally used as backing store for Trie data structure
 * <p>
 * Created by Anton Nashatyrev on 11.11.2016.
 */
public class WriteCache<Key, Value> extends AbstractCachedSource<Key, Value> {

    /**
     * Type of the write cache
     */
    public enum CacheType {
        /**
         * Simple acts as regular Map: single and double adding of the same entry has the same
effect
         * Source entries (key/value pairs) may have arbitrary nature
         */
        SIMPLE,
        /**
         * Counting counts the resulting number of inserts (+1) and deletes (-1) and when flushed
         * does the resulting number of inserts (if sum > 0) or deletes (if sum < 0)
         * Counting Source acts like {@link HashedKeySource} and makes sense only for data
         * where a single key always corresponds to a single value
         * Counting cache normally used as backing store for Trie data structure
         */
        COUNTING
    }

    private static abstract class CacheEntry<V> implements Entry<V> {
        // dedicated value instance which indicates that the entry was deleted
        // (ref counter decremented) but we don't know actual value behind it
        static final Object UNKNOWN_VALUE = new Object();

        V value;
        int counter = 0;

        protected CacheEntry(V value) {
            this.value = value;
        }

        protected abstract void deleted();

        protected abstract void added();

        protected abstract V getValue();

        @Override
        public V value() {
            V v = getValue();
            return v == UNKNOWN_VALUE ? null : v;
        }
    }

    private static final class SimpleCacheEntry<V> extends CacheEntry<V> {
        public SimpleCacheEntry(V value) {
            super(value);
        }

        @Override
        public void deleted() {
            counter = -1;
        }

        @Override
        public void added() {
            counter = 1;
        }

        @Override
        public V getValue() {
            return counter < 0 ? null : value;
        }
    }

    private static final class CountCacheEntry<V> extends CacheEntry<V> {
        public CountCacheEntry(V value) {
            super(value);
        }

        @Override
        public void deleted() {
            counter--;
        }

        @Override
        public void added() {
            counter++;
        }
    @Override
    public V getValue() {
        // for counting cache we return the cached value even if
        // it was deleted (once or several times) as we don't know
        // how many 'instances' are left behind
        return value;
    }
}

private final boolean isCounting;

protected volatile Map<Key, CacheEntry<Value>> cache = new HashMap<>();

protected ReadWriteUpdateLock rwuLock = new ReentrantReadWriteUpdateLock();
protected ALock readLock = new ALock(rwuLock.readLock());
protected ALock writeLock = new ALock(rwuLock.writeLock());
protected ALock updateLock = new ALock(rwuLock.updateLock());

private boolean checked = false;

public WriteCache(Source<Key, Value> src, CacheType cacheType) {
    super(src);
    this.isCounting = cacheType == CacheType.COUNTING;
}

public WriteCache<Key, Value> withCache(Map<Key, CacheEntry<Value>> cache) {
    this.cache = cache;
    return this;
}

@Override
public Collection<Key> getModified() {
    try (ALock l = readLock.lock()) {
        return cache.keySet();
    }
}

@Override
public boolean hasModified() {
    return !cache.isEmpty();
}
```

```
private CacheEntry<Value> createCacheEntry(Value val) {
    if (isCounting) {
        return new CountCacheEntry<>(val);
    } else {
        return new SimpleCacheEntry<>(val);
    }
}

@Override
public void put(Key key, Value val) {
    checkByteArrKey(key);
    if (val == null) {
        delete(key);
        return;
    }



    try (ALock l = writeLock.lock()) {
        CacheEntry<Value> curVal = cache.get(key);
        if (curVal == null) {
            curVal = createCacheEntry(val);
            CacheEntry<Value> oldVal = cache.put(key, curVal);
            if (oldVal != null) {
                cacheRemoved(key, oldVal.value == unknownValue() ? null : oldVal.value);
            }
            cacheAdded(key, curVal.value);
        }
        // assigning for non-counting cache only
        // for counting cache the value should be immutable (see HashedKeySource)
        curVal.value = val;
        curVal.added();
    }
}

@Override
public Value get(Key key) {
    checkByteArrKey(key);
    try (ALock l = readLock.lock()) {
        CacheEntry<Value> curVal = cache.get(key);
        if (curVal == null) {
            return getSource() == null ? null : getSource().get(key);
        } else {
            Value value = curVal.getValue();
            if (value == unknownValue()) {
                return getSource() == null ? null : getSource().get(key);
            } else {
                return value;
            }
        }
    }
}

@Override
public void delete(Key key) {
    checkByteArrKey(key);
    try (ALock l = writeLock.lock()) {
        CacheEntry<Value> curVal = cache.get(key);
        if (curVal == null) {
            curVal = createCacheEntry(getSource() == null ? null : unknownValue());
            CacheEntry<Value> oldVal = cache.put(key, curVal);
            if (oldVal != null) {
                cacheRemoved(key, oldVal.value);
            }
            cacheAdded(key, curVal.value == unknownValue() ? null : curVal.value);
        }
        curVal.deleted();
    }
}

@Override
public boolean flush() {
    boolean ret = false;
    try (ALock l = updateLock.lock()) {
        for (Map.Entry<Key, CacheEntry<Value>> entry : cache.entrySet()) {
            if (entry.getValue().counter > 0) {
                for (int i = 0; i < entry.getValue().counter; i++) {
                    getSource().put(entry.getKey(), entry.getValue().value);
                }
                ret = true;
            } else if (entry.getValue().counter < 0) {
                for (int i = 0; i > entry.getValue().counter; i--) {
                    getSource().delete(entry.getKey());
                }
                ret = true;
            }
        }
        if (flushSource) {
            getSource().flush();
        }
        try (ALock l1 = writeLock.lock()) {
            cache.clear();
            cacheCleared();
        }
        return ret;
    }
}

@Override
protected boolean flushImpl() {
    return false;
}

private Value unknownValue() {
    return (Value) CacheEntry.UNKNOWN_VALUE;
}

@Override
public Entry<Value> getCached(Key key) {
    try (ALock l = readLock.lock()) {
        CacheEntry<Value> entry = cache.get(key);
        if (entry == null || entry.value == unknownValue()) {
            return null;
        } else {
            return entry;
        }
    }
}

// Guard against wrong cache Map
// if a regular Map is accidentally used for byte[] type keys
// the situation might be tricky to debug
private void checkByteArrKey(Key key) {
    if (checked) {
        return;
    }
```

```
        if (key instanceof byte[]) {
            if (!(cache instanceof ByteArrayMap)) {
                throw new RuntimeException("Wrong map/set for byte[] key");
            }
        }
        checked = true;
    }

    public long debugCacheSize() {
        long ret = 0;
        for (Map.Entry<Key, CacheEntry<Value>> entry : cache.entrySet()) {
            ret += keySizeEstimator.estimateSize(entry.getKey());
            ret += valueSizeEstimator.estimateSize(entry.getValue().value());
        }
        return ret;
    }

    /**
     * Shortcut for WriteCache with byte[] keys. Also prevents accidental
     * usage of regular Map implementation (non byte[])
     */
    public static class BytesKey<V> extends WriteCache<byte[], V> implements
CachedSource.BytesKey<V> {

        public BytesKey(Source<byte[], V> src, CacheType cacheType) {
            super(src, cacheType);
            withCache(new ByteArrayMap<CacheEntry<V>>());
        }
    }
}
```

17:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\main\java\org\ethereum\datasource\XorDataSource.java

```
 * with the specified value
 * <p>
 * May be useful for merging several Sources into a single
 * <p>
 * Created by Anton Nashatyrev on 18.02.2016.
 */
public class XorDataSource<V> extends AbstractChainedSource<byte[], V, byte[], V> {
    private byte[] subKey;
    /**
     * Creates instance with a value all keys are XORed with
     */
    public XorDataSource(Source<byte[], V> source, byte[] subKey) {
        super(source);
        this.subKey = subKey;
    }

    private byte[] convertKey(byte[] key) {
        return ByteUtil.xorAlignRight(key, subKey);
    }

    @Override
    public V get(byte[] key) {
        return getSource().get(convertKey(key));
    }

    @Override
    public void put(byte[] key, V value) {
        getSource().put(convertKey(key), value);
    }

    @Override
    public void delete(byte[] key) {
        getSource().delete(convertKey(key));
    }

    @Override
    protected boolean flushImpl() {
        return false;
    }
}
```

18:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\main\java\org\ethereum\db\AbstractBlockstore.java

```
 */
public abstract class AbstractBlockstore implements BlockStore {

    @Override
    public byte[] getBlockHashByNumber(long blockNumber, byte[] branchBlockHash) {
        Block branchBlock = getBlockByHash(branchBlockHash);
        if (branchBlock.getNumber() < blockNumber) {
            throw new IllegalArgumentException("Requested block number > branch hash number: " +
blockNumber + " < " + branchBlock.getNumber());
        }
        while (branchBlock.getNumber() > blockNumber) {
            branchBlock = getBlockByHash(branchBlock.getParentHash());
        }
        return branchBlock.getHash();
    }
}
```

19:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\main\java\org\ethereum\db\BlockStore.java

```
import java.util.List;

/**
 * @author Roman Mandeleil
 * @since 08.01.2015
 */
public interface BlockStore {

    byte[] getBlockHashByNumber(long blockNumber);

    /**
     * Gets the block hash by its index.
     * When more than one block with the specified index exists (forks)
     * the select the block which is ancestor of the branchBlockHash
     */
    byte[] getBlockHashByNumber(long blockNumber, byte[] branchBlockHash);

    Block getChainBlockByNumber(long blockNumber);

    Block getBlockByHash(byte[] hash);

    boolean isBlockExist(byte[] hash);

    List<byte[]> getListHashesEndWith(byte[] hash, long qty);

    List<BlockHeader> getListHeadersEndWith(byte[] hash, long qty);

    List<Block> getListBlocksEndWith(byte[] hash, long qty);

    void saveBlock(Block block, BigInteger totalDifficulty, boolean mainChain);
    BigInteger getTotalDifficultyForHash(byte[] hash);

    BigInteger getTotalDifficulty();

    Block getBestBlock();

    long getMaxNumber();


    void flush();

    void reBranch(Block forkBlock);

    void load();

    void close();
}
```

20:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\main\java\org\ethereum\db\BlockStoreDummy.java

```
import java.math.BigInteger;
import java.util.List;

/**
 * @author Roman Mandeleil
 * @since 10.02.2015
 */
public class BlockStoreDummy implements BlockStore {

    @Override
    public byte[] getBlockHashByNumber(long blockNumber) {

        byte[] data = String.valueOf(blockNumber).getBytes();
        return HashUtil.sha3(data);
    }

    @Override
    public byte[] getBlockHashByNumber(long blockNumber, byte[] branchBlockHash) {
        return getBlockHashByNumber(blockNumber);
    }
```

```
@Override
public Block getChainBlockByNumber(long blockNumber) {
    return null;
}

@Override
public Block getBlockByHash(byte[] hash) {
    return null;
}

@Override
public boolean isBlockExist(byte[] hash) {
    return false;
}

@Override
public List<byte[]> getListHashesEndWith(byte[] hash, long qty) {
    return null;
}

@Override
public List<BlockHeader> getListHeadersEndWith(byte[] hash, long qty) {
    return null;
}

@Override
public List<Block> getListBlocksEndWith(byte[] hash, long qty) {
    return null;
}

@Override
public void saveBlock(Block block, BigInteger totalDifficulty, boolean mainChain) {

}



@Override
public BigInteger getTotalDifficulty() {
    return null;
}

@Override
```

```
    public Block getBestBlock() {
        return null;
    }



    @Override
    public void flush() {
    }

    @Override
    public void load() {
    }

    @Override
    public long getMaxNumber() {
        return 0;
    }



    @Override
    public void reBranch(Block forkBlock) {

    }

    @Override
    public BigInteger getTotalDifficultyForHash(byte[] hash) {
        return null;
    }

    @Override
    public void close() {
    }
}
```

21:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\main\java\org\ethereum\db\ByteArrayWrapper.java

```
import static org.ethereum.util.ByteUtil.toHexString;

/**
 * @author Roman Mandeleil
 * @since 11.06.2014
 */
public class ByteArrayWrapper implements Comparable<ByteArrayWrapper>, Serializable {

    private final byte[] data;
    private int hashCode = 0;

    public ByteArrayWrapper(byte[] data) {
        if (data == null) {
            throw new NullPointerException("Data must not be null");
        }
        this.data = data;
        this.hashCode = Arrays.hashCode(data);
    }

    @Override
    public boolean equals(Object other) {
        if (!(other instanceof ByteArrayWrapper)) {
            return false;
        }
        byte[] otherData = ((ByteArrayWrapper) other).getData();
        return FastByteComparisons.compareTo(
                data, 0, data.length,
                otherData, 0, otherData.length) == 0;
    }

    @Override
    public int hashCode() {
        return hashCode;
    }

    @Override
    public int compareTo(ByteArrayWrapper o) {
        return FastByteComparisons.compareTo(
                data, 0, data.length,
                o.getData(), 0, o.getData().length);
    }

    public byte[] getData() {
        return data;
    }

    @Override
    public String toString() {
        return toHexString(data);
    }
}
```

22:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\main\java\org\ethereum\db\ContractDetails.java

```
import java.util.List;
import java.util.Map;
import java.util.Set;

public interface ContractDetails {

    void put(DataWord key, DataWord value);

    DataWord get(DataWord key);

    byte[] getCode();

    byte[] getCode(byte[] codeHash);

    void setCode(byte[] code);

    byte[] getStorageHash();

    void decode(byte[] rlpCode);

    void setDirty(boolean dirty);

    void setDeleted(boolean deleted);

    boolean isDirty();

    boolean isDeleted();

    byte[] getEncoded();

    int getStorageSize();

    Set<DataWord> getStorageKeys();

    Map<DataWord, DataWord> getStorage(@Nullable Collection<DataWord> keys);
    Map<DataWord, DataWord> getStorage();

    void setStorage(List<DataWord> storageKeys, List<DataWord> storageValues);

    void setStorage(Map<DataWord, DataWord> storage);

    byte[] getAddress();

    void setAddress(byte[] address);

    ContractDetails clone();

    @Override
    String toString();

    void syncStorage();

    ContractDetails getSnapshotTo(byte[] hash);
}
```

23:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\main\java\org\ethereum\db\DbFlushManager.java

```
import org.ethereum.datasource.AsyncFlushable;
import org.ethereum.datasource.DbSource;
import org.ethereum.datasource.Source;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.util.HashSet;
import java.util.List;
import java.util.Set;
import java.util.concurrent.*;

/**
 * Created by Anton Nashatyrev on 01.12.2016.
 */
public class DbFlushManager {
    private static final Logger logger = LoggerFactory.getLogger("db");

    List<AbstractCachedSource<byte[], ?>> writeCaches = new CopyOnWriteArrayList<>();
    List<Source<byte[], ?>> sources = new CopyOnWriteArrayList<>();
    Set<DbSource> dbSources = new HashSet<>();
    AbstractCachedSource<byte[], byte[]> stateDbCache;

    long sizeThreshold;
    int commitsCountThreshold;
    boolean syncDone = false;
    boolean flushAfterSyncDone;

    SystemProperties config;

    int commitCount = 0;

    private final BlockingQueue<Runnable> executorQueue = new ArrayBlockingQueue<>(1);
    private final ExecutorService flushThread = new ThreadPoolExecutor(1, 1, 0L, 
TimeUnit.MILLISECONDS,
            executorQueue, new ThreadFactoryBuilder().setNameFormat("DbFlushManagerThread-
%d").build());
    Future<Boolean> lastFlush = Futures.immediateFuture(false);

    public DbFlushManager(SystemProperties config, Set<DbSource> dbSources, 
AbstractCachedSource<byte[], byte[]> stateDbCache) {
        this.config = config;
        this.dbSources = dbSources;
        sizeThreshold = config.getConfig().getInt("cache.flush.writeCacheSize") * 1024 * 1024;
        commitsCountThreshold = config.getConfig().getInt("cache.flush.blocks");
        flushAfterSyncDone = config.getConfig().getBoolean("cache.flush.shortSyncFlush");
        this.stateDbCache = stateDbCache;
    }

    public void setSizeThreshold(long sizeThreshold) {
        this.sizeThreshold = sizeThreshold;
    }

    public void addCache(AbstractCachedSource<byte[], ?> cache) {
        writeCaches.add(cache);
    }

    public void addSource(Source<byte[], ?> src) {
        sources.add(src);
    }

    public long getCacheSize() {
        long ret = 0;
        for (AbstractCachedSource<byte[], ?> writeCache : writeCaches) {
            ret += writeCache.estimateCacheSize();
        }
        return ret;
    }

    public synchronized void commit(Runnable atomicUpdate) {
        atomicUpdate.run();
        commit();
    }

    public synchronized void commit() {
        long cacheSize = getCacheSize();
        if (sizeThreshold >= 0 && cacheSize >= sizeThreshold) {
            logger.debug("DbFlushManager: flushing db due to write cache size (" + cacheSize + ")
reached threshold (" + sizeThreshold + ")");
            flush();
        } else if (commitsCountThreshold > 0 && commitCount >= commitsCountThreshold) {
            logger.debug("DbFlushManager: flushing db due to commits (" + commitCount + ")
reached threshold (" + commitsCountThreshold + ")");
            flush();
            commitCount = 0;
        } else if (flushAfterSyncDone && syncDone) {
            logger.debug("DbFlushManager: flushing db due to short sync");
            flush();
        }
        commitCount++;
    }

    public synchronized void flushSync() {
        try {
            flush().get();
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    public synchronized Future<Boolean> flush() {
        if (!lastFlush.isDone()) {
            logger.debug("Waiting for previous flush to complete...");
            try {
                lastFlush.get();
            } catch (Exception e) {
                logger.error("Error during last flush", e);
            }
        }
        logger.debug("Flipping async storages");
        for (AbstractCachedSource<byte[], ?> writeCache : writeCaches) {
            try {
                if (writeCache instanceof AsyncFlushable) {
                    ((AsyncFlushable) writeCache).flipStorage();
                }
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
        }

        logger.debug("Submitting flush task");
        return lastFlush = flushThread.submit(() -> {
            boolean ret = false;
            long s = System.nanoTime();
            logger.debug("Flush started");

            sources.forEach(Source::flush);

            for (AbstractCachedSource<byte[], ?> writeCache : writeCaches) {
                if (writeCache instanceof AsyncFlushable) {
                    try {
                        ret |= ((AsyncFlushable) writeCache).flushAsync().get();
                    } catch (InterruptedException e) {
                        throw new RuntimeException(e);
                    }
                } else {
                    ret |= writeCache.flush();
                }
            }
            if (stateDbCache != null) {
                logger.debug("Flushing to DB");
                stateDbCache.flush();
            }
            logger.debug("Flush completed in " + (System.nanoTime() - s) / 1000000 + " ms");

            return ret;
        });
    }

    /**
     * Flushes all caches and closes all databases
     */
    public synchronized void close() {
        logger.debug("Flushing DBs...");
        flushSync();
        logger.debug("Flush done.");
        for (DbSource dbSource : dbSources) {
            logger.debug("Closing DB: {}", dbSource.getName());
            try {
                dbSource.close();
            } catch (Exception ex) {
                logger.error(String.format("Caught error while closing DB: %s", dbSource.getName()),
ex);
            }
        }
    }
}
```

24:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\main\java\org\ethereum\db\HeaderStore.java

```
import org.ethereum.datasource.Source;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;


/**
 * BlockHeaders store
 * Assumes one chain
 * Uses indexes by header hash and block number
 */
public class HeaderStore {

    private static final Logger logger = LoggerFactory.getLogger("general");

    Source<byte[], byte[]> indexDS;
    DataSourceArray<byte[]> index;
    Source<byte[], byte[]> headersDS;
```

```
ObjectDataSource<BlockHeader> headers;

public HeaderStore() {
}

public void init(Source<byte[], byte[]> index, Source<byte[], byte[]> headers) {
    indexDS = index;
    this.index = new DataSourceArray<>(
            new ObjectDataSource<>(index, Serializers.AsIsSerializer, 2048));
    this.headersDS = headers;
    this.headers = new ObjectDataSource<>(headers, Serializers.BlockHeaderSerializer, 512);
}


public synchronized BlockHeader getBestHeader() {

    long maxNumber = getMaxNumber();
    if (maxNumber < 0) {
        return null;
    }

    return getHeaderByNumber(maxNumber);
}

public synchronized void flush() {
    headers.flush();
    index.flush();
    headersDS.flush();
    indexDS.flush();
}

public synchronized void saveHeader(BlockHeader header) {
    index.set((int) header.getNumber(), header.getHash());
    headers.put(header.getHash(), header);
}

public synchronized BlockHeader getHeaderByNumber(long number) {
    if (number < 0 || number >= index.size()) {
        return null;
    }

    byte[] hash = index.get((int) number);
```

```
        if (hash == null) {
            return null;
        }

        return headers.get(hash);
    }

    public synchronized int size() {
        return index.size();
    }

    public synchronized BlockHeader getHeaderByHash(byte[] hash) {
        return headers.get(hash);
    }

    public synchronized long getMaxNumber() {
        if (index.size() > 0) {
            return (long) index.size() - 1;
        } else {
            return -1;
        }
    }
}
```

25:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\main\java\org\ethereum\db\index\ArrayListIndex.java

```
 * @since 28.01.2016
 */
public class ArrayListIndex implements Index {
    private List<Long> index;

    public ArrayListIndex(Collection<Long> numbers) {
        index = new ArrayList<>(numbers);
        sort();
    }

    @Override
    public synchronized void addAll(Collection<Long> nums) {
        index.addAll(nums);
        sort();
    }
```

```
@Override
public synchronized void add(Long num) {
    index.add(num);
    sort();
}

@Override
public synchronized Long peek() {
    return index.get(0);
}

@Override
public synchronized Long poll() {
    Long num = index.get(0);
    index.remove(0);
    return num;
}

@Override
public synchronized boolean contains(Long num) {
    return Collections.binarySearch(index, num) >= 0;
}

@Override
public synchronized boolean isEmpty() {
    return index.isEmpty();
}

@Override
public synchronized int size() {
    return index.size();
}

@Override
public synchronized void clear() {
    index.clear();
}

private void sort() {
    Collections.sort(index);
}
```

```
    @Override
    public synchronized Iterator<Long> iterator() {
        return new ArrayList<>(index).iterator();
    }

    @Override
    public synchronized void removeAll(Collection<Long> indexes) {
        index.removeAll(indexes);
    }

    @Override
    public synchronized Long peekLast() {

        if (index.isEmpty()) {
            return null;
        }
        return index.get(index.size() - 1);
    }

    @Override
    public synchronized void remove(Long num) {
        index.remove(num);
    }
}
```

26:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\main\java\org\ethereum\db\index\Index.java

```
 * @since 28.01.2016
 */
public interface Index extends Iterable<Long> {

    void addAll(Collection<Long> nums);

    void add(Long num);

    Long peek();

    Long poll();

    boolean contains(Long num);

    boolean isEmpty();
    int size();

    void clear();

    void removeAll(Collection<Long> indexes);

    Long peekLast();

    void remove(Long num);
}
```

27:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\main\java\org\ethereum\db\IndexedBlockStore.java

```
import org.ethereum.datasource.Serializer;
import org.ethereum.datasource.Source;
import org.ethereum.util.*;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.io.Serializable;
import java.math.BigInteger;
import java.util.ArrayList;
import java.util.List;

import static java.math.BigInteger.ZERO;
import static org.ethereum.crypto.HashUtil.shortHash;
import static org.spongycastle.util.Arrays.areEqual;

public class IndexedBlockStore extends AbstractBlockstore {

    private static final Logger logger = LoggerFactory.getLogger("general");

    Source<byte[], byte[]> indexDS;
    DataSourceArray<List<BlockInfo>> index;
    Source<byte[], byte[]> blocksDS;
    ObjectDataSource<Block> blocks;

    public IndexedBlockStore() {
    }

    public void init(Source<byte[], byte[]> index, Source<byte[], byte[]> blocks) {
```

```
    indexDS = index;
    this.index = new DataSourceArray<>(
            new ObjectDataSource<>(index, BLOCK_INFO_SERIALIZER, 512));
    this.blocksDS = blocks;
    this.blocks = new ObjectDataSource<>(blocks, new Serializer<Block, byte[]>() {
        @Override
        public byte[] serialize(Block block) {
            return block.getEncoded();
        }

        @Override
        public Block deserialize(byte[] bytes) {
            return bytes == null ? null : new Block(bytes);
        }
    }, 256);
}

@Override
public synchronized Block getBestBlock() {

    Long maxLevel = getMaxNumber();
    if (maxLevel < 0) {
        return null;
    }

    Block bestBlock = getChainBlockByNumber(maxLevel);
    if (bestBlock != null) {
        return bestBlock;
    }

    // That scenario can happen
    // if there is a fork branch that is
    // higher than main branch but has
    // less TD than the main branch TD
    while (bestBlock == null) {
        --maxLevel;
        bestBlock = getChainBlockByNumber(maxLevel);
    }

    return bestBlock;
}
```

```
    @Override
    public synchronized byte[] getBlockHashByNumber(long blockNumber) {
        Block chainBlock = getChainBlockByNumber(blockNumber);
        return chainBlock == null ? null : chainBlock.getHash(); // FIXME: can be improved by
accessing the hash directly in the index
    }



    @Override
    public synchronized void flush() {
        blocks.flush();
        index.flush();
        blocksDS.flush();
        indexDS.flush();
    }



    @Override
    public synchronized void saveBlock(Block block, BigInteger totalDifficulty, boolean mainChain) {
        addInternalBlock(block, totalDifficulty, mainChain);
    }

    private void addInternalBlock(Block block, BigInteger totalDifficulty, boolean mainChain) {

        List<BlockInfo> blockInfos = block.getNumber() >= index.size() ? null : index.get((int)
block.getNumber());
        blockInfos = blockInfos == null ? new ArrayList<BlockInfo>() : blockInfos;

        BlockInfo blockInfo = new BlockInfo();
        blockInfo.setTotalDifficulty(totalDifficulty);
        blockInfo.setHash(block.getHash());
        blockInfo.setMainChain(mainChain); // FIXME:maybe here I should force reset main chain for
all uncles on that level

        putBlockInfo(blockInfos, blockInfo);
        index.set((int) block.getNumber(), blockInfos);

        blocks.put(block.getHash(), block);
    }

    private void putBlockInfo(List<BlockInfo> blockInfos, BlockInfo blockInfo) {
        for (int i = 0; i < blockInfos.size(); i++) {
            BlockInfo curBlockInfo = blockInfos.get(i);
            if (FastByteComparisons.equal(curBlockInfo.getHash(), blockInfo.getHash())) {
                blockInfos.set(i, blockInfo);
                return;
            }
        }
        blockInfos.add(blockInfo);
    }


    public synchronized List<Block> getBlocksByNumber(long number) {

        List<Block> result = new ArrayList<>();

        if (number >= index.size()) {
            return result;
        }

        List<BlockInfo> blockInfos = index.get((int) number);

        if (blockInfos == null) {
            return result;
        }

        for (BlockInfo blockInfo : blockInfos) {

            byte[] hash = blockInfo.getHash();
            Block block = blocks.get(hash);

            result.add(block);
        }
        return result;
    }

    @Override
    public synchronized Block getChainBlockByNumber(long number) {
        if (number >= index.size()) {
            return null;
        }

        List<BlockInfo> blockInfos = index.get((int) number);
        if (blockInfos == null) {
            return null;
        }

        for (BlockInfo blockInfo : blockInfos) {

            if (blockInfo.isMainChain()) {

                byte[] hash = blockInfo.getHash();
                return blocks.get(hash);
            }
        }

        return null;
    }

    @Override
    public synchronized Block getBlockByHash(byte[] hash) {
        return blocks.get(hash);
    }

    @Override
    public synchronized boolean isBlockExist(byte[] hash) {
        return blocks.get(hash) != null;
    }


    @Override
    public synchronized BigInteger getTotalDifficultyForHash(byte[] hash) {
        Block block = this.getBlockByHash(hash);
        if (block == null) {
            return ZERO;
        }

        Long level = block.getNumber();
        List<BlockInfo> blockInfos = index.get(level.intValue());
        for (BlockInfo blockInfo : blockInfos) {
            if (areEqual(blockInfo.getHash(), hash)) {
                return blockInfo.totalDifficulty;
            }
        }
        return ZERO;
    }


    @Override
    public synchronized BigInteger getTotalDifficulty() {
        long maxNumber = getMaxNumber();

        List<BlockInfo> blockInfos = index.get((int) maxNumber);
        for (BlockInfo blockInfo : blockInfos) {
            if (blockInfo.isMainChain()) {
                return blockInfo.getTotalDifficulty();
            }
        }

        while (true) {
            --maxNumber;
            List<BlockInfo> infos = getBlockInfoForLevel(maxNumber);

            for (BlockInfo blockInfo : infos) {
                if (blockInfo.isMainChain()) {
                    return blockInfo.getTotalDifficulty();
                }
            }
        }
    }

    @Override
    public synchronized long getMaxNumber() {

        Long bestIndex = 0L;

        if (index.size() > 0) {
            bestIndex = (long) index.size();
        }

        return bestIndex - 1L;
    }

    @Override
    public synchronized List<byte[]> getListHashesEndWith(byte[] hash, long number) {
        List<Block> blocks = getListBlocksEndWith(hash, number);
        List<byte[]> hashes = new ArrayList<>(blocks.size());

        for (Block b : blocks) {
            hashes.add(b.getHash());
        }

        return hashes;
    }

    @Override
    public synchronized List<BlockHeader> getListHeadersEndWith(byte[] hash, long qty) {

        List<Block> blocks = getListBlocksEndWith(hash, qty);
        List<BlockHeader> headers = new ArrayList<>(blocks.size());

        for (Block b : blocks) {
            headers.add(b.getHeader());
        }

        return headers;
    }

    @Override
    public synchronized List<Block> getListBlocksEndWith(byte[] hash, long qty) {
        return getListBlocksEndWithInner(hash, qty);
    }

    private List<Block> getListBlocksEndWithInner(byte[] hash, long qty) {

        Block block = this.blocks.get(hash);

        if (block == null) {
            return new ArrayList<>();
        }

        List<Block> blocks = new ArrayList<>((int) qty);

        for (int i = 0; i < qty; ++i) {
            blocks.add(block);
            block = this.blocks.get(block.getParentHash());
            if (block == null) {
```

```
            break;
        }
    }

    return blocks;
}

@Override
public synchronized void reBranch(Block forkBlock) {

    Block bestBlock = getBestBlock();

    long maxLevel = Math.max(bestBlock.getNumber(), forkBlock.getNumber());

    // 1. First ensure that you are one the save level
    long currentLevel = maxLevel;
    Block forkLine = forkBlock;
    if (forkBlock.getNumber() > bestBlock.getNumber()) {

        while (currentLevel > bestBlock.getNumber()) {
            List<BlockInfo> blocks = getBlockInfoForLevel(currentLevel);
            BlockInfo blockInfo = getBlockInfoForHash(blocks, forkLine.getHash());
            if (blockInfo != null) {
                blockInfo.setMainChain(true);
                setBlockInfoForLevel(currentLevel, blocks);
            }
            forkLine = getBlockByHash(forkLine.getParentHash());
            --currentLevel;
        }
    }

    Block bestLine = bestBlock;
    if (bestBlock.getNumber() > forkBlock.getNumber()) {

        while (currentLevel > forkBlock.getNumber()) {

            List<BlockInfo> blocks = getBlockInfoForLevel(currentLevel);
            BlockInfo blockInfo = getBlockInfoForHash(blocks, bestLine.getHash());
            if (blockInfo != null) {
                blockInfo.setMainChain(false);
                setBlockInfoForLevel(currentLevel, blocks);
            }
            bestLine = getBlockByHash(bestLine.getParentHash());
            --currentLevel;
        }
    }


    // 2. Loop back on each level until common block
    while (!bestLine.isEqual(forkLine)) {

        List<BlockInfo> levelBlocks = getBlockInfoForLevel(currentLevel);
        BlockInfo bestInfo = getBlockInfoForHash(levelBlocks, bestLine.getHash());
        if (bestInfo != null) {
            bestInfo.setMainChain(false);
            setBlockInfoForLevel(currentLevel, levelBlocks);
        }

        BlockInfo forkInfo = getBlockInfoForHash(levelBlocks, forkLine.getHash());
        if (forkInfo != null) {
            forkInfo.setMainChain(true);
            setBlockInfoForLevel(currentLevel, levelBlocks);
        }


        bestLine = getBlockByHash(bestLine.getParentHash());
        forkLine = getBlockByHash(forkLine.getParentHash());

        --currentLevel;
    }


}


public synchronized List<byte[]> getListHashesStartWith(long number, long maxBlocks) {

    List<byte[]> result = new ArrayList<>();

    int i;
    for (i = 0; i < maxBlocks; ++i) {
        List<BlockInfo> blockInfos = index.get((int) number);
        if (blockInfos == null) {
            break;
            }

            for (BlockInfo blockInfo : blockInfos) {
                if (blockInfo.isMainChain()) {
                    result.add(blockInfo.getHash());
                    break;
                }
            }

            ++number;
        }
        maxBlocks -= i;

        return result;
    }

    public static class BlockInfo implements Serializable {
        byte[] hash;
        BigInteger totalDifficulty;
        boolean mainChain;

        public byte[] getHash() {
            return hash;
        }

        public void setHash(byte[] hash) {
            this.hash = hash;
        }

        public BigInteger getTotalDifficulty() {
            return totalDifficulty;
        }

        public void setTotalDifficulty(BigInteger totalDifficulty) {
            this.totalDifficulty = totalDifficulty;
        }

        public boolean isMainChain() {
            return mainChain;
        }

        public void setMainChain(boolean mainChain) {
            this.mainChain = mainChain;
        }
    }


    public static final Serializer<List<BlockInfo>, byte[]> BLOCK_INFO_SERIALIZER = new
Serializer<List<BlockInfo>, byte[]>() {

        @Override
        public byte[] serialize(List<BlockInfo> value) {
            List<byte[]> rlpBlockInfoList = new ArrayList<>();
            for (BlockInfo blockInfo : value) {
                byte[] hash = RLP.encodeElement(blockInfo.getHash());
                // Encoding works correctly only with positive BigIntegers
                if (blockInfo.getTotalDifficulty() == null ||
blockInfo.getTotalDifficulty().compareTo(BigInteger.ZERO) < 0) {
                    throw new RuntimeException("BlockInfo totalDifficulty should be positive BigInteger");
                }
                byte[] totalDiff = RLP.encodeBigInteger(blockInfo.getTotalDifficulty());
                byte[] isMainChain = RLP.encodeInt(blockInfo.isMainChain() ? 1 : 0);
                rlpBlockInfoList.add(RLP.encodeList(hash, totalDiff, isMainChain));
            }
            byte[][] elements = rlpBlockInfoList.toArray(new byte[rlpBlockInfoList.size()][]);

            return RLP.encodeList(elements);
        }

        @Override
        public List<BlockInfo> deserialize(byte[] bytes) {
            if (bytes == null) {
                return null;
            }

            List<BlockInfo> blockInfoList = new ArrayList<>();
            RLPList list = (RLPList) RLP.decode2(bytes).get(0);
            for (RLPElement element : list) {
                RLPList rlpBlock = (RLPList) element;
                BlockInfo blockInfo = new BlockInfo();
                byte[] rlpHash = rlpBlock.get(0).getRLPData();
                blockInfo.setHash(rlpHash == null ? new byte[0] : rlpHash);
                byte[] rlpTotalDiff = rlpBlock.get(1).getRLPData();
                blockInfo.setTotalDifficulty(rlpTotalDiff == null ? BigInteger.ZERO :
```

```
ByteUtil.bytesToBigInteger(rlpTotalDiff));
                blockInfo.setMainChain(ByteUtil.byteArrayToInt(rlpBlock.get(2).getRLPData()) == 1);
                blockInfoList.add(blockInfo);
            }

            return blockInfoList;
        }
    };


    public synchronized void printChain() {

        Long number = getMaxNumber();

        for (int i = 0; i < number; ++i) {
            List<BlockInfo> levelInfos = index.get(i);

            if (levelInfos != null) {
                System.out.print(i);
                for (BlockInfo blockInfo : levelInfos) {
                    if (blockInfo.isMainChain()) {
                        System.out.print(" [" + shortHash(blockInfo.getHash()) + "] ");
                    } else {
                        System.out.print(" " + shortHash(blockInfo.getHash()) + " ");
                    }
                }
                System.out.println();
            }

        }

    }

    private synchronized List<BlockInfo> getBlockInfoForLevel(long level) {
        return index.get((int) level);
    }

    private synchronized void setBlockInfoForLevel(long level, List<BlockInfo> infos) {
        index.set((int) level, infos);
    }

    private static BlockInfo getBlockInfoForHash(List<BlockInfo> blocks, byte[] hash) {
        for (BlockInfo blockInfo : blocks) {
            if (areEqual(hash, blockInfo.getHash())) {
                return blockInfo;
            }
        }

        return null;
    }

    @Override
    public synchronized void load() {
    }

    @Override
    public synchronized void close() {
//        logger.info("Closing IndexedBlockStore...");
//        try {
//            indexDS.close();
//        } catch (Exception e) {
//            logger.warn("Problems closing indexDS", e);
//        }
//        try {
//            blocksDS.close();
//        } catch (Exception e) {
//            logger.warn("Problems closing blocksDS", e);
//        }
    }
}
```

28:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\main\java\org\ethereum\db\prune\Chain.java

```
        }
    };

    List<ChainItem> items = new ArrayList<>();

    public List<byte[]> getHashes() {
        return items.stream().map(item -> item.hash).collect(Collectors.toList());
    }

    private Chain() {
    }

    Chain(ChainItem item) {
        this.items.add(item);
    }

    ChainItem top() {
        return items.size() > 0 ? items.get(items.size() - 1) : null;
    }

    long topNumber() {
        return top() != null ? top().number : 0;
    }

    long startNumber() {
        return items.isEmpty() ? 0 : items.get(0).number;
    }

    boolean isHigher(Chain other) {
        return other.topNumber() < this.topNumber();
    }

    boolean contains(ChainItem other) {
        for (ChainItem item : items) {
            if (item.equals(other)) {
                return true;
            }
        }
        return false;
    }

    boolean connect(ChainItem item) {
        if (top().isParentOf(item)) {
            items.add(item);
            return true;
        }

        return false;
    }

    static Chain fromItems(ChainItem... items) {
        if (items.length == 0) {
            return NULL;
        }

        Chain chain = null;
        for (ChainItem item : items) {
            if (chain == null) {
                chain = new Chain(item);
            } else {
                chain.connect(item);
            }
        }

        return chain;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) {
            return true;
        }
        if (o == null || getClass() != o.getClass()) {
            return false;
        }

        Chain chain = (Chain) o;

        return !(items != null ? !items.equals(chain.items) : chain.items != null);
    }

    @Override
    public String toString() {
        if (items.isEmpty()) {
            return "(empty)";
        }
        return "[" + items.get(0) +
                " ~> " + items.get(items.size() - 1) +
                ']';
    }

}
```

29:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-

vm\src\main\java\org\ethereum\db\prune\ChainItem.java

```
    ChainItem(long number, byte[] hash, byte[] parentHash) {
        this.number = number;
        this.hash = hash;
        this.parentHash = parentHash;
    }

    boolean isParentOf(ChainItem that) {
        return FastByteComparisons.equal(hash, that.parentHash);
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) {
            return true;
        }
        if (o == null || getClass() != o.getClass()) {
            return false;
        }
        ChainItem that = (ChainItem) o;
        return FastByteComparisons.equal(hash, that.hash);
    }

    @Override
    public int hashCode() {
        return hash != null ? Arrays.hashCode(hash) : 0;
    }

    @Override
    public String toString() {
        return String.valueOf(number);
    }
}
```

30:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\main\java\org\ethereum\db\prune\Pruner.java

```
 * Taking the information supplied by {@link #journal} (check {@link JournalSource} for details)
 * removes unused nodes from the {@link #storage}.
 * There are two types of unused nodes:
 * nodes not references in the trie after N blocks from the current one and
 * nodes which were inserted in the forks that finally were not accepted
 *
 * <p>
 * Each prune session uses a certain chain {@link Segment}
 * which is going to be 'pruned'. To be confident that live nodes won't be removed,
 * pruner must be initialized with the top of the chain, see {@link #init(List, int)}}.
 * And after that it must be fed with each newly processed block, see {@link
#feed(JournalSource.Update)}.
 * {@link QuotientFilter} ({@link CountingQuotientFilter} implementation in particular) instance is
used to
 * efficiently keep upcoming inserts in memory and protect newly inserted nodes from being
deleted during
 * prune session. The filter is constantly recycled in {@link #prune(Segment)} method.
 *
 * <p>
 * When 'prune.maxDepth' param is quite big, it becomes not efficient to keep reverted nodes until
prune block number has come.
 * Hence Pruner has two step mode to mitigate memory consumption, second step is initiated by
{@link #withSecondStep(List, int)}.
 * In that mode nodes from not accepted forks are deleted from storage immediately but main
chain deletions are
 * postponed for the second step.
 * Second step uses another one instance of QuotientFilter with less memory impact, check {@link
#instantiateFilter(int, int)}.
 *
 * <p>
 * Basically, prune session initiated by {@link #prune(Segment)} method
 * consists of 3 steps: first, it reverts forks, then it persists main chain,
 * after that it recycles {@link #journal} by removing processed updates from it.
 * During the session reverted and deleted nodes are propagated to the {@link #storage}
immediately.
 *
 * @author Mikhail Kalinin
 * @since 25.01.2018
 */
public class Pruner {

    private static final Logger logger = LoggerFactory.getLogger("prune");

    Source<byte[], JournalSource.Update> journal;
    Source<byte[], ?> storage;
    QuotientFilter filter;
    QuotientFilter distantFilter;
```

```
boolean ready = false;

private static class Stats {
    int collisions = 0;
    int deleted = 0;
    double load = 0;

    @Override
    public String toString() {
        return String.format("load %.4f, collisions %d, deleted %d", load, collisions, deleted);
    }
}

Stats maxLoad = new Stats();
Stats maxCollisions = new Stats();
int maxKeysInMemory = 0;
int statsTracker = 0;

Stats distantMaxLoad = new Stats();
Stats distantMaxCollisions = new Stats();

public Pruner(Source<byte[], JournalSource.Update> journal, Source<byte[], ?> storage) {
    this.storage = storage;
    this.journal = journal;
}

public boolean isReady() {
    return ready;
}

public boolean init(List<byte[]> forkWindow, int sizeInBlocks) {
    if (ready) {
        return true;
    }

    if (!forkWindow.isEmpty() && journal.get(forkWindow.get(0)) == null) {
        logger.debug("pruner init aborted: can't fetch update " + toHexString(forkWindow.get(0)));
        return false;
    }

    QuotientFilter filter = instantiateFilter(sizeInBlocks, FILTER_ENTRIES_FORK);
    for (byte[] hash : forkWindow) {
            JournalSource.Update update = journal.get(hash);
            if (update == null) {
                logger.debug("pruner init aborted: can't fetch update " + toHexString(hash));
                //return false;
                continue;
            }
            update.getInsertedKeys().forEach(filter::insert);
        }

        this.filter = filter;
        return ready = true;
    }

    public boolean withSecondStep() {
        return distantFilter != null;
    }

    public void withSecondStep(List<byte[]> mainChainWindow, int sizeInBlocks) {
        if (!ready) {
            return;
        }

        QuotientFilter filter = instantiateFilter(sizeInBlocks, FILTER_ENTRIES_DISTANT);

        if (!mainChainWindow.isEmpty()) {
            int i = mainChainWindow.size() - 1;
            for (; i >= 0; i--) {
                byte[] hash = mainChainWindow.get(i);
                JournalSource.Update update = journal.get(hash);
                if (update == null) {
                    break;
                }
                update.getInsertedKeys().forEach(filter::insert);
            }
            logger.debug("distant filter initialized with set of " + (i < 0 ? mainChainWindow.size() :
mainChainWindow.size() - i) +
                    " hashes, last hash " + toHexString(mainChainWindow.get(i < 0 ? 0 : i)));
        } else {
            logger.debug("distant filter initialized with empty set");
        }

        this.distantFilter = filter;
    }

    private static final int FILTER_ENTRIES_FORK = 1 << 13; // approximate number of nodes per
block
    private static final int FILTER_ENTRIES_DISTANT = 1 << 11;
    private static final int FILTER_MAX_SIZE = Integer.MAX_VALUE >> 1; // that filter will consume
~3g of mem

    private QuotientFilter instantiateFilter(int blocksCnt, int entries) {
        int size = Math.min(entries * blocksCnt, FILTER_MAX_SIZE);
        return CountingQuotientFilter.create(size, size);
    }

    public boolean init(byte[]... upcoming) {
        return init(Arrays.asList(upcoming), 192);
    }

    public void feed(JournalSource.Update update) {
        if (ready) {
            update.getInsertedKeys().forEach(filter::insert);
        }
    }

    public void prune(Segment segment) {
        if (!ready) {
            return;
        }
        assert segment.isComplete();

        logger.trace("prune " + segment);

        long t = System.currentTimeMillis();
        Pruning pruning = new Pruning();
        // important for fork management, check Pruning#insertedInMainChain and
Pruning#insertedInForks for details
        segment.forks.sort((f1, f2) -> Long.compare(f1.startNumber(), f2.startNumber()));
        segment.forks.forEach(pruning::revert);

        // delete updates
        for (Chain chain : segment.forks) {
            chain.getHashes().forEach(journal::delete);
        }
        int nodesPostponed = 0;
        if (withSecondStep()) {
            nodesPostponed = postpone(segment.main);
        } else {
            pruning.nodesDeleted += persist(segment.main);
            segment.main.getHashes().forEach(journal::delete);
        }

        if (logger.isTraceEnabled()) {
            logger.trace("nodes {}, keys in mem: {}, filter load: {}/{}: {}, distinct collisions: {}",
                    (withSecondStep() ? "postponed: " + nodesPostponed : "deleted: " +
pruning.nodesDeleted),
                    pruning.insertedInForks.size() + pruning.insertedInMainChain.size(),
                    ((CountingQuotientFilter) filter).getEntryNumber(), ((CountingQuotientFilter)
filter).getMaxInsertions(),
                    String.format("%.4f", (double) ((CountingQuotientFilter) filter).getEntryNumber() /
                            ((CountingQuotientFilter) filter).getMaxInsertions()),
                    ((CountingQuotientFilter) filter).getCollisionNumber());
        }

        if (logger.isDebugEnabled()) {
            int collisions = ((CountingQuotientFilter) filter).getCollisionNumber();
            double load = (double) ((CountingQuotientFilter) filter).getEntryNumber() /
                    ((CountingQuotientFilter) filter).getMaxInsertions();
            if (collisions > maxCollisions.collisions) {
                maxCollisions.collisions = collisions;
                maxCollisions.load = load;
                maxCollisions.deleted = pruning.nodesDeleted;
            }
            if (load > maxLoad.load) {
                maxLoad.load = load;
                maxLoad.collisions = collisions;
                maxLoad.deleted = pruning.nodesDeleted;
            }
            maxKeysInMemory = Math.max(maxKeysInMemory, pruning.insertedInForks.size() +
pruning.insertedInMainChain.size());

            if (++statsTracker % 100 == 0) {
                logger.debug("fork filter: max load: " + maxLoad);
                logger.debug("fork filter: max collisions: " + maxCollisions);
                logger.debug("fork filter: max keys in mem: " + maxKeysInMemory);
            }
        }

        logger.trace(segment + " pruned in {}ms", System.currentTimeMillis() - t);
    }

    public void persist(byte[] hash) {
        if (!ready || !withSecondStep()) {
            return;
        }

        logger.trace("persist [{}]", toHexString(hash));

        long t = System.currentTimeMillis();
        JournalSource.Update update = journal.get(hash);
        if (update == null) {
            logger.debug("skip [{}]: can't fetch update", HashUtil.shortHash(hash));
            return;
        }

        // persist deleted keys
        int nodesDeleted = 0;
        for (byte[] key : update.getDeletedKeys()) {
            if (!filter.maybeContains(key) && !distantFilter.maybeContains(key)) {
                ++nodesDeleted;
                storage.delete(key);
            }
        }
        // clean up filter
        update.getInsertedKeys().forEach(distantFilter::remove);
        // delete update
        journal.delete(hash);

        if (logger.isDebugEnabled()) {
            int collisions = ((CountingQuotientFilter) distantFilter).getCollisionNumber();
            double load = (double) ((CountingQuotientFilter) distantFilter).getEntryNumber() /
                    ((CountingQuotientFilter) distantFilter).getMaxInsertions();
            if (collisions > distantMaxCollisions.collisions) {
                distantMaxCollisions.collisions = collisions;
                distantMaxCollisions.load = load;
                distantMaxCollisions.deleted = nodesDeleted;
            }
```

```
            if (load > distantMaxLoad.load) {
                distantMaxLoad.load = load;
                distantMaxLoad.collisions = collisions;
                distantMaxLoad.deleted = nodesDeleted;
            }
            if (statsTracker % 100 == 0) {
                logger.debug("distant filter: max load: " + distantMaxLoad);
                logger.debug("distant filter: max collisions: " + distantMaxCollisions);
            }
        }

        if (logger.isTraceEnabled()) {
            logger.trace("[{}] persisted in {}ms: {}/{} ({}%) nodes deleted, filter load: {}/{}: {}, distinct
collisions: {}",
                    HashUtil.shortHash(hash), System.currentTimeMillis() - t, nodesDeleted,
update.getDeletedKeys().size(),
                    nodesDeleted * 100 / update.getDeletedKeys().size(),
                    ((CountingQuotientFilter) distantFilter).getEntryNumber(),
                    ((CountingQuotientFilter) distantFilter).getMaxInsertions(),
                    String.format("%.4f", (double) ((CountingQuotientFilter)
distantFilter).getEntryNumber() /
                            ((CountingQuotientFilter) distantFilter).getMaxInsertions()),
                    ((CountingQuotientFilter) distantFilter).getCollisionNumber());
        }
    }

    private int postpone(Chain chain) {
        if (logger.isTraceEnabled()) {
            logger.trace("<~ postponing " + chain + ": " + strSample(chain.getHashes()));
        }

        int nodesPostponed = 0;
        for (byte[] hash : chain.getHashes()) {
            JournalSource.Update update = journal.get(hash);
            if (update == null) {
                logger.debug("postponing: can't fetch update " + toHexString(hash));
                continue;
            }
            // feed distant filter
            update.getInsertedKeys().forEach(distantFilter::insert);
            // clean up fork filter
            update.getInsertedKeys().forEach(filter::remove);
        nodesPostponed += update.getDeletedKeys().size();
    }

    return nodesPostponed;
}

private int persist(Chain chain) {
    if (logger.isTraceEnabled()) {
        logger.trace("<~ persisting " + chain + ": " + strSample(chain.getHashes()));
    }

    int nodesDeleted = 0;
    for (byte[] hash : chain.getHashes()) {
        JournalSource.Update update = journal.get(hash);
        if (update == null) {
            logger.debug("pruning aborted: can't fetch update of main chain " + toHexString(hash));
            return 0;
        }
        // persist deleted keys
        for (byte[] key : update.getDeletedKeys()) {
            if (!filter.maybeContains(key)) {
                ++nodesDeleted;
                storage.delete(key);
            }
        }
        // clean up filter
        update.getInsertedKeys().forEach(filter::remove);
    }

    return nodesDeleted;
}

private String strSample(Collection<byte[]> hashes) {
    String sample = hashes.stream().limit(3)
            .map(HashUtil::shortHash).collect(Collectors.joining(", "));
    if (hashes.size() > 3) {
        sample += ", ... (" + hashes.size() + " total)";
    }
    return sample;
}
```

```
    private class Pruning {

        // track nodes inserted and deleted in forks
        // to avoid deletion of those nodes which were originally inserted in the main chain
        Set<byte[]> insertedInMainChain = new ByteArraySet();
        Set<byte[]> insertedInForks = new ByteArraySet();
        int nodesDeleted = 0;

        private void revert(Chain chain) {
            if (logger.isTraceEnabled()) {
                logger.trace("<~ reverting " + chain + ": " + strSample(chain.getHashes()));
            }

            for (byte[] hash : chain.getHashes()) {
                JournalSource.Update update = journal.get(hash);
                if (update == null) {
                    logger.debug("reverting chain " + chain + " aborted: can't fetch update " +
toHexString(hash));
                    return;
                }
                // clean up filter
                update.getInsertedKeys().forEach(filter::remove);

                // node that was deleted in fork considered as a node that had earlier been inserted in
main chain
                update.getDeletedKeys().forEach(key -> {
                    if (!insertedInForks.contains(key)) {
                        insertedInMainChain.add(key);
                    }
                });
                update.getInsertedKeys().forEach(key -> {
                    if (!insertedInMainChain.contains(key)) {
                        insertedInForks.add(key);
                    }
                });

                // revert inserted keys
                for (byte[] key : update.getInsertedKeys()) {
                    if (!filter.maybeContains(key) && !insertedInMainChain.contains(key)) {
                        ++nodesDeleted;
                        storage.delete(key);
                    }
                }
            }
        }
    }
}

31:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-
vm\src\main\java\org\ethereum\db\prune\Segment.java
 * add main chain blocks with {@link Tracker#addMain(Block)},
 * then when all blocks are added {@link Tracker#commit()} should be fired
 * to connect added blocks to the segment
 *
 * @author Mikhail Kalinin
 * @see Chain
 * @see ChainItem
 * @since 24.01.2018
 */
public class Segment {

    List<Chain> forks = new ArrayList<>();
    Chain main = Chain.NULL;
    ChainItem root;

    public Segment(Block root) {
        this.root = new ChainItem(root);
    }

    public Segment(long number, byte[] hash, byte[] parentHash) {
        this.root = new ChainItem(number, hash, parentHash);
    }

    public boolean isComplete() {
        if (main == Chain.NULL) {
            return false;
        }

        for (Chain fork : forks) {
            if (!main.isHigher(fork)) {
                return false;
            }
        }
        return true;
```

```
}

public long getRootNumber() {
    return root.number;
}

public long getMaxNumber() {
    return main.topNumber();
}

public Tracker startTracking() {
    return new Tracker(this);
}

public int size() {
    return main.items.size();
}

private void branch(ChainItem item) {
    forks.add(new Chain(item));
}

private void connectMain(ChainItem item) {
    if (main == Chain.NULL) {
        if (root.isParentOf(item)) {
            main = new Chain(item); // start new
        }
    } else {
        main.connect(item);
    }
}

private void connectFork(ChainItem item) {

    for (Chain fork : forks) {
        if (fork.contains(item)) {
            return;
        }
    }

    if (root.isParentOf(item)) {
        branch(item);
    } else {
        for (ChainItem mainItem : main.items) {
            if (mainItem.isParentOf(item)) {
                branch(item);
            }
        }

        for (Chain fork : forks) {
            if (fork.connect(item)) {
                return;
            }
        }

        List<Chain> branchedForks = new ArrayList<>();
        for (Chain fork : forks) {
            for (ChainItem forkItem : fork.items) {
                if (forkItem.isParentOf(item)) {
                    branchedForks.add(new Chain(item));
                }
            }
        }
        forks.addAll(branchedForks);
    }
}

@Override
public String toString() {
    return "" + main;
}

public static final class Tracker {

    Segment segment;
    List<ChainItem> main = new ArrayList<>();
    List<ChainItem> items = new ArrayList<>();

    Tracker(Segment segment) {
        this.segment = segment;
    }

    public void addMain(Block block) {
        main.add(new ChainItem(block));
    }

    public void addAll(List<Block> blocks) {
        items.addAll(blocks.stream()
                .map(ChainItem::new)
                .collect(Collectors.toList()));
    }

    public Tracker addMain(long number, byte[] hash, byte[] parentHash) {
        main.add(new ChainItem(number, hash, parentHash));
        return this;
    }

    public Tracker addItem(long number, byte[] hash, byte[] parentHash) {
        items.add(new ChainItem(number, hash, parentHash));
        return this;
    }

    public void commit() {

        items.removeAll(main);

        main.sort((i1, i2) -> Long.compare(i1.number, i2.number));
        items.sort((i1, i2) -> Long.compare(i1.number, i2.number));

        main.forEach(segment::connectMain);
        items.forEach(segment::connectFork);
    }
}
}
```

32:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\main\java\org\ethereum\db\PruneManager.java

```
import org.ethereum.datasource.Source;
import org.ethereum.db.prune.Pruner;
import org.ethereum.db.prune.Segment;

import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;

/**
```

```
 * Manages state pruning part of block processing.
 *
 * <p>
 * Constructs chain segments and prune them when they are complete
 * <p>
 * Created by Anton Nashatyrev on 10.11.2016.
 *
 * @see Segment
 * @see Pruner
 */
public class PruneManager {

    private static final int LONGEST_CHAIN = 192;

    private JournalSource<?> journalSource;

    private IndexedBlockStore blockStore;

    private int pruneBlocksCnt;

    private Segment segment;
    private Pruner pruner;

    private PruneManager(SystemProperties config) {
        pruneBlocksCnt = config.databasePruneDepth();
    }

    public PruneManager(IndexedBlockStore blockStore, JournalSource<?> journalSource,
                        Source<byte[], ?> pruneStorage, int pruneBlocksCnt) {
        this.blockStore = blockStore;
        this.journalSource = journalSource;
        this.pruneBlocksCnt = pruneBlocksCnt;

        if (journalSource != null && pruneStorage != null) {
            this.pruner = new Pruner(journalSource.getJournal(), pruneStorage);
        }
    }

    public void setStateSource(StateSource stateSource) {
        journalSource = stateSource.getJournalSource();
        if (journalSource != null) {
            pruner = new Pruner(journalSource.getJournal(), stateSource.getNoJournalSource());
        }
    }

    public void blockCommitted(BlockHeader block) {
        if (pruneBlocksCnt < 0) {
            return; // pruning disabled
        }

        JournalSource.Update update = journalSource.commitUpdates(block.getHash());
        pruner.feed(update);

        long forkBlockNum = block.getNumber() - getForkBlocksCnt();
        if (forkBlockNum < 0) {
            return;
        }

        List<Block> pruneBlocks = blockStore.getBlocksByNumber(forkBlockNum);
        Block chainBlock = blockStore.getChainBlockByNumber(forkBlockNum);

        // reset segment and return
        // if chainBlock is accidentally null
        if (chainBlock == null) {
            segment = null;
            return;
        }

        if (segment == null) {
            if (pruneBlocks.size() == 1)    // wait for a single chain
            {
                segment = new Segment(chainBlock);
            }
            return;
        }

        Segment.Tracker tracker = segment.startTracking();
        tracker.addMain(chainBlock);
        tracker.addAll(pruneBlocks);
        tracker.commit();

        if (segment.isComplete()) {
            if (!pruner.isReady()) {
                List<byte[]> forkWindow = getAllChainsHashes(segment.getRootNumber() + 1,
blockStore.getMaxNumber());
                pruner.init(forkWindow, getForkBlocksCnt());

                int mainChainWindowSize = pruneBlocksCnt - getForkBlocksCnt();
                if (mainChainWindowSize > 0) {
                    List<byte[]> mainChainWindow = getMainChainHashes(Math.max(1,
segment.getRootNumber() - mainChainWindowSize + 1),
                            segment.getRootNumber());
                    pruner.withSecondStep(mainChainWindow, mainChainWindowSize);
                }
            }
            pruner.prune(segment);
            segment = new Segment(chainBlock);
        }

        long mainBlockNum = block.getNumber() - getMainBlocksCnt();
        if (mainBlockNum < 0) {
            return;
        }

        byte[] hash = blockStore.getBlockHashByNumber(mainBlockNum);
        pruner.persist(hash);
    }

    private int getForkBlocksCnt() {
        return Math.min(pruneBlocksCnt, 2 * LONGEST_CHAIN);
    }

    private int getMainBlocksCnt() {
        if (pruneBlocksCnt <= 2 * LONGEST_CHAIN) {
            return Integer.MAX_VALUE;
        } else {
            return pruneBlocksCnt;
        }
    }

    private List<byte[]> getAllChainsHashes(long fromBlock, long toBlock) {
        List<byte[]> ret = new ArrayList<>();
        for (long num = fromBlock; num <= toBlock; num++) {
            List<Block> blocks = blockStore.getBlocksByNumber(num);
            List<byte[]> hashes = blocks.stream().map(Block::getHash).collect(Collectors.toList());
            ret.addAll(hashes);
        }
        return ret;
    }

    private List<byte[]> getMainChainHashes(long fromBlock, long toBlock) {
        List<byte[]> ret = new ArrayList<>();
        for (long num = fromBlock; num <= toBlock; num++) {
            byte[] hash = blockStore.getBlockHashByNumber(num);
            ret.add(hash);
        }
        return ret;
    }
}
```

33:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\main\java\org\ethereum\db\RepositoryImpl.java

```
import org.ethereum.crypto.HashUtil;
import org.ethereum.datasource.*;
import org.ethereum.util.ByteUtil;
import org.ethereum.util.FastByteComparisons;
import org.ethereum.vm.DataWord;

import javax.annotation.Nullable;
import java.math.BigInteger;
import java.util.*;

/**
 * Created by Anton Nashatyrev on 07.10.2016.
 */
public class RepositoryImpl implements Repository, org.ethereum.facade.Repository {

    protected RepositoryImpl parent;

    protected Source<byte[], AccountState> accountStateCache;
    protected Source<byte[], byte[]> codeCache;
    protected MultiCache<? extends CachedSource<DataWord, DataWord>> storageCache;

    protected SystemProperties config = SystemProperties.getDefault();

    protected RepositoryImpl() {
    }
    public RepositoryImpl(Source<byte[], AccountState> accountStateCache, Source<byte[],
byte[]> codeCache,
                          MultiCache<? extends CachedSource<DataWord, DataWord>> storageCache) {
        init(accountStateCache, codeCache, storageCache);
    }

    protected void init(Source<byte[], AccountState> accountStateCache, Source<byte[], byte[]>
codeCache,
                        MultiCache<? extends CachedSource<DataWord, DataWord>> storageCache) {
        this.accountStateCache = accountStateCache;
        this.codeCache = codeCache;
        this.storageCache = storageCache;
    }

    @Override
    public synchronized AccountState createAccount(byte[] addr, byte[] creater) {
        AccountState state = new
AccountState(config.getBlockchainConfig().getCommonConstants().getInitialNonce(),
                BigInteger.ZERO, creater);
        accountStateCache.put(addr, state);
        return state;
    }

    @Override
    public synchronized boolean isExist(byte[] addr) {
        return getAccountState(addr) != null;
    }

    @Override
    public synchronized AccountState getAccountState(byte[] addr) {
        return accountStateCache.get(addr);
    }

    synchronized AccountState getOrCreateAccountState(byte[] addr) {
        AccountState ret = accountStateCache.get(addr);
        if (ret == null) {
            ret = createAccount(addr, HashUtil.EMPTY_DATA_HASH);
        }
        return ret;
    }

    @Override
    public synchronized void delete(byte[] addr) {
        accountStateCache.delete(addr);
        storageCache.delete(addr);
    }

    @Override
    public synchronized BigInteger increaseNonce(byte[] addr) {
        AccountState accountState = getOrCreateAccountState(addr);
        accountStateCache.put(addr, accountState.withIncrementedNonce());
        return accountState.getNonce();
    }

    @Override
    public synchronized BigInteger setNonce(byte[] addr, BigInteger nonce) {
        AccountState accountState = getOrCreateAccountState(addr);
        accountStateCache.put(addr, accountState.withNonce(nonce));
        return accountState.getNonce();
    }

    @Override
    public synchronized BigInteger getNonce(byte[] addr) {
        AccountState accountState = getAccountState(addr);
        return accountState == null ?
config.getBlockchainConfig().getCommonConstants().getInitialNonce() :
                accountState.getNonce();
    }

    @Override
    public synchronized ContractDetails getContractDetails(byte[] addr) {
        return new ContractDetailsImpl(addr);
    }

    @Override
    public synchronized boolean hasContractDetails(byte[] addr) {
        return getContractDetails(addr) != null;
    }

    @Override
    public synchronized void saveCode(byte[] addr, byte[] code) {
        byte[] codeHash = HashUtil.sha3(code);
        codeCache.put(codeKey(codeHash, addr), code);
        AccountState accountState = getOrCreateAccountState(addr);
        accountStateCache.put(addr, accountState.withCodeHash(codeHash));
    }

    @Override
    public synchronized byte[] getCode(byte[] addr) {
        byte[] codeHash = getCodeHash(addr);
        return codeHash == null || FastByteComparisons.equal(codeHash,
HashUtil.EMPTY_DATA_HASH) ?
                ByteUtil.EMPTY_BYTE_ARRAY : codeCache.get(codeKey(codeHash, addr));
    }

    // composing a key as there can be several contracts with the same code
    private byte[] codeKey(byte[] codeHash, byte[] addr) {
        return NodeKeyCompositor.compose(codeHash, addr);
    }

    @Override
    public byte[] getCodeHash(byte[] addr) {
        AccountState accountState = getAccountState(addr);
        return accountState != null ? accountState.getCodeHash() : null;
    }

    @Override
    public synchronized void addStorageRow(byte[] addr, DataWord key, DataWord value) {
        getOrCreateAccountState(addr);

        Source<DataWord, DataWord> contractStorage = storageCache.get(addr);
        contractStorage.put(key, value.isZero() ? null : value);
    }

    @Override
    public synchronized DataWord getStorageValue(byte[] addr, DataWord key) {
        AccountState accountState = getAccountState(addr);
        return accountState == null ? null : storageCache.get(addr).get(key);
    }

    @Override
    public synchronized BigInteger getBalance(byte[] addr) {
        AccountState accountState = getAccountState(addr);
        return accountState == null ? BigInteger.ZERO : accountState.getBalance();
    }
    @Override
    public synchronized BigInteger addBalance(byte[] addr, BigInteger value) {
        AccountState accountState = getOrCreateAccountState(addr);
        accountStateCache.put(addr, accountState.withBalanceIncrement(value));
        return accountState.getBalance();
    }

    @Override
    public synchronized RepositoryImpl startTracking() {
        Source<byte[], AccountState> trackAccountStateCache = new
WriteCache.BytesKey<>(accountStateCache,
                WriteCache.CacheType.SIMPLE);
        Source<byte[], byte[]> trackCodeCache = new WriteCache.BytesKey<>(codeCache,
WriteCache.CacheType.SIMPLE);
        MultiCache<CachedSource<DataWord, DataWord>> trackStorageCache = new
MultiCache(storageCache) {
            @Override
            protected CachedSource create(byte[] key, CachedSource srcCache) {
                return new WriteCache<>(srcCache, WriteCache.CacheType.SIMPLE);
            }
        };

        RepositoryImpl ret = new RepositoryImpl(trackAccountStateCache, trackCodeCache,
trackStorageCache);
        ret.parent = this;
        return ret;
    }

    @Override
    public synchronized Repository getSnapshotTo(byte[] root) {
        return parent.getSnapshotTo(root);
    }

    @Override
    public synchronized void commit() {
        Repository parentSync = parent == null ? this : parent;
        // need to synchronize on parent since between different caches flush
        // the parent repo would not be in consistent state
        // when no parent just take this instance as a mock
        synchronized (parentSync) {
            storageCache.flush();
            codeCache.flush();
            accountStateCache.flush();
        }
    }

    @Override
    public synchronized void rollback() {
        // nothing to do, will be GCed
    }

    @Override
    public byte[] getRoot() {
        throw new RuntimeException("Not supported");
    }

    public synchronized String getTrieDump() {
        return dumpStateTrie();
    }

    public String dumpStateTrie() {
        throw new RuntimeException("Not supported");
    }

    /**
     * As tests only implementation this hack is pretty sufficient
     */
    @Override
    public Repository clone() {
        return parent.startTracking();
    }

    class ContractDetailsImpl implements ContractDetails {
        private byte[] address;

        public ContractDetailsImpl(byte[] address) {
            this.address = address;
        }

        @Override
        public void put(DataWord key, DataWord value) {
            RepositoryImpl.this.addStorageRow(address, key, value);
        }
```

```
@Override
public DataWord get(DataWord key) {
   return RepositoryImpl.this.getStorageValue(address, key);
}

@Override
public byte[] getCode() {
   return RepositoryImpl.this.getCode(address);
}

@Override
public byte[] getCode(byte[] codeHash) {
   throw new RuntimeException("Not supported");
}

@Override
public void setCode(byte[] code) {
   RepositoryImpl.this.saveCode(address, code);
}

@Override
public byte[] getStorageHash() {
   throw new RuntimeException("Not supported");
}

@Override
public void decode(byte[] rlpCode) {
   throw new RuntimeException("Not supported");
}

@Override
public void setDirty(boolean dirty) {
   throw new RuntimeException("Not supported");
}

@Override
public void setDeleted(boolean deleted) {
   RepositoryImpl.this.delete(address);
}

@Override
public boolean isDirty() {
    throw new RuntimeException("Not supported");
}

@Override
public boolean isDeleted() {
    throw new RuntimeException("Not supported");
}

@Override
public byte[] getEncoded() {
    throw new RuntimeException("Not supported");
}

@Override
public int getStorageSize() {
    throw new RuntimeException("Not supported");
}

@Override
public Set<DataWord> getStorageKeys() {
    throw new RuntimeException("Not supported");
}

@Override
public Map<DataWord, DataWord> getStorage(@Nullable Collection<DataWord> keys) {
    throw new RuntimeException("Not supported");
}

@Override
public Map<DataWord, DataWord> getStorage() {
    throw new RuntimeException("Not supported");
}

@Override
public void setStorage(List<DataWord> storageKeys, List<DataWord> storageValues) {
    throw new RuntimeException("Not supported");
}

@Override
public void setStorage(Map<DataWord, DataWord> storage) {
    throw new RuntimeException("Not supported");
}
```

```
        @Override
        public byte[] getAddress() {
            return address;
        }

        @Override
        public void setAddress(byte[] address) {
            throw new RuntimeException("Not supported");
        }

        @Override
        public ContractDetails clone() {
            throw new RuntimeException("Not supported");
        }

        @Override
        public void syncStorage() {
            throw new RuntimeException("Not supported");
        }

        @Override
        public ContractDetails getSnapshotTo(byte[] hash) {
            throw new RuntimeException("Not supported");
        }
    }


    @Override
    public Set<byte[]> getAccountsKeys() {
        throw new RuntimeException("Not supported");
    }

    @Override
    public void dumpState(Block block, long gasUsed, int txNumber, byte[] txHash) {
        throw new RuntimeException("Not supported");
    }

    @Override
    public void flush() {
        throw new RuntimeException("Not supported");
    }
    @Override
    public void flushNoReconnect() {
        throw new RuntimeException("Not supported");
    }

    @Override
    public void syncToRoot(byte[] root) {
        throw new RuntimeException("Not supported");
    }

    @Override
    public boolean isClosed() {
        throw new RuntimeException("Not supported");
    }

    @Override
    public void close() {
    }

    @Override
    public void reset() {
        throw new RuntimeException("Not supported");
    }

    @Override
    public int getStorageSize(byte[] addr) {
        throw new RuntimeException("Not supported");
    }

    @Override
    public Set<DataWord> getStorageKeys(byte[] addr) {
        throw new RuntimeException("Not supported");
    }

    @Override
    public Map<DataWord, DataWord> getStorage(byte[] addr, @Nullable Collection<DataWord>
keys) {
        throw new RuntimeException("Not supported");
    }
    @Override
    public void updateBatch(HashMap<ByteArrayWrapper, AccountState> accountStates,
HashMap<ByteArrayWrapper, ContractDetails> contractDetailes) {
        for (Map.Entry<ByteArrayWrapper, AccountState> entry : accountStates.entrySet()) {
            accountStateCache.put(entry.getKey().getData(), entry.getValue());
        }
        for (Map.Entry<ByteArrayWrapper, ContractDetails> entry : contractDetailes.entrySet()) {
            ContractDetails details = getContractDetails(entry.getKey().getData());
            for (DataWord key : entry.getValue().getStorageKeys()) {
                details.put(key, entry.getValue().get(key));
            }
            byte[] code = entry.getValue().getCode();
            if (code != null && code.length > 0) {
                details.setCode(code);
            }
        }
    }

    @Override
    public void loadAccount(byte[] addr, HashMap<ByteArrayWrapper, AccountState>
cacheAccounts, HashMap<ByteArrayWrapper, ContractDetails> cacheDetails) {
        throw new RuntimeException("Not supported");
    }

}
```

34:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\main\java\org\ethereum\db\RepositoryRoot.java

```
import org.ethereum.trie.Trie;
import org.ethereum.trie.TrieImpl;
import org.ethereum.vm.DataWord;

/**
 * Created by Anton Nashatyrev on 07.10.2016.
 */
public class RepositoryRoot extends RepositoryImpl {

    private static class StorageCache extends ReadWriteCache<DataWord, DataWord> {
        Trie<byte[]> trie;

        public StorageCache(Trie<byte[]> trie) {
            super(new SourceCodec<>(trie, Serializers.StorageKeySerializer,
Serializers.StorageValueSerializer), WriteCache.CacheType.SIMPLE);
            this.trie = trie;
        }
    }

    private class MultiStorageCache extends MultiCache<StorageCache> {
        public MultiStorageCache() {
            super(null);
        }

        @Override
        protected synchronized StorageCache create(byte[] key, StorageCache srcCache) {
            AccountState accountState = accountStateCache.get(key);
            Serializer<byte[], byte[]> keyCompositor = new NodeKeyCompositor(key);
            Source<byte[], byte[]> composingSrc = new SourceCodec.KeyOnly<>(trieCache,
keyCompositor);
            TrieImpl storageTrie = createTrie(composingSrc, accountState == null ? null :
accountState.getStateRoot());
            return new StorageCache(storageTrie);
        }

        @Override
        protected synchronized boolean flushChild(byte[] key, StorageCache childCache) {
            if (super.flushChild(key, childCache)) {
                if (childCache != null) {
                    AccountState storageOwnerAcct = accountStateCache.get(key);
                    // need to update account storage root
                    childCache.trie.flush();
                    byte[] rootHash = childCache.trie.getRootHash();
                    accountStateCache.put(key, storageOwnerAcct.withStateRoot(rootHash));
                    return true;
                } else {
                    // account was deleted
                    return true;
                }
            } else {
                // no storage changes
                return false;
            }
        }
    }
    private Source<byte[], byte[]> stateDS;
    private CachedSource.BytesKey<byte[]> trieCache;
    private Trie<byte[]> stateTrie;

    public RepositoryRoot(Source<byte[], byte[]> stateDS) {
        this(stateDS, null);
    }

    /**
     * Building the following structure for snapshot Repository:
     * <p>
     * stateDS --> trieCache --> stateTrie --> accountStateCodec --> accountStateCache
     * \                 \
     * \                 \-->>> storageKeyCompositor --> contractStorageTrie --> storageCodec -->
storageCache
     * \--> codeCache
     *
     * @param stateDS
     * @param root
     */
    public RepositoryRoot(final Source<byte[], byte[]> stateDS, byte[] root) {
        this.stateDS = stateDS;

        trieCache = new WriteCache.BytesKey<>(stateDS, WriteCache.CacheType.COUNTING);
        stateTrie = new SecureTrie(trieCache, root);

        SourceCodec.BytesKey<AccountState, byte[]> accountStateCodec = new
SourceCodec.BytesKey<>(stateTrie, Serializers.AccountStateSerializer);
        final ReadWriteCache.BytesKey<AccountState> accountStateCache = new
ReadWriteCache.BytesKey<>(accountStateCodec, WriteCache.CacheType.SIMPLE);

        final MultiCache<StorageCache> storageCache = new MultiStorageCache();

        // counting as there can be 2 contracts with the same code, 1 can suicide
        Source<byte[], byte[]> codeCache = new WriteCache.BytesKey<>(stateDS,
WriteCache.CacheType.COUNTING);

        init(accountStateCache, codeCache, storageCache);
    }

    @Override
    public synchronized void commit() {
```

```
    super.commit();

    stateTrie.flush();
    trieCache.flush();
}

@Override
public synchronized byte[] getRoot() {
    storageCache.flush();
    accountStateCache.flush();

    return stateTrie.getRootHash();
}

@Override
public synchronized void flush() {
    commit();
}

@Override
public Repository getSnapshotTo(byte[] root) {
    return new RepositoryRoot(stateDS, root);
}

@Override
public Repository clone() {
    return getSnapshotTo(getRoot());
}

@Override
public synchronized String dumpStateTrie() {
    return ((TrieImpl) stateTrie).dumpTrie();
}

@Override
public synchronized void syncToRoot(byte[] root) {
    stateTrie.setRoot(root);
}

protected TrieImpl createTrie(Source<byte[], byte[]> trieCache, byte[] root) {
    return new SecureTrie(trieCache, root);
}
```

```
}

35:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-
vm\src\main\java\org\ethereum\db\RepositoryWrapper.java

import javax.annotation.Nullable;
import java.math.BigInteger;
import java.util.Collection;
import java.util.HashMap;
import java.util.Map;
import java.util.Set;

/**
 * Repository delegating all calls to the last Repository
 * <p>
 * Created by Anton Nashatyrev on 22.12.2016.
 */
public class RepositoryWrapper implements Repository {

    private Repository repository;

    public RepositoryWrapper(Repository repository) {
        this.repository = repository;
    }

    public Repository getRepository() {
        return repository;
    }

    @Override
    public AccountState createAccount(byte[] addr, byte[] creater) {
        return getRepository().createAccount(addr, creater);
    }

    @Override
    public boolean isExist(byte[] addr) {
        return getRepository().isExist(addr);
    }

    @Override
    public AccountState getAccountState(byte[] addr) {
```

```
    return getRepository().getAccountState(addr);
}

@Override
public void delete(byte[] addr) {
    getRepository().delete(addr);
}

@Override
public BigInteger increaseNonce(byte[] addr) {
    return getRepository().increaseNonce(addr);
}

@Override
public BigInteger setNonce(byte[] addr, BigInteger nonce) {
    return getRepository().setNonce(addr, nonce);
}

@Override
public BigInteger getNonce(byte[] addr) {
    return getRepository().getNonce(addr);
}

@Override
public ContractDetails getContractDetails(byte[] addr) {
    return getRepository().getContractDetails(addr);
}

@Override
public boolean hasContractDetails(byte[] addr) {
    return getRepository().hasContractDetails(addr);
}

@Override
public void saveCode(byte[] addr, byte[] code) {
    getRepository().saveCode(addr, code);
}

@Override
public byte[] getCode(byte[] addr) {
    return getRepository().getCode(addr);
}
```

```
@Override
public byte[] getCodeHash(byte[] addr) {
    return getRepository().getCodeHash(addr);
}

@Override
public void addStorageRow(byte[] addr, DataWord key, DataWord value) {
    getRepository().addStorageRow(addr, key, value);
}

@Override
public DataWord getStorageValue(byte[] addr, DataWord key) {
    return getRepository().getStorageValue(addr, key);
}

@Override
public BigInteger getBalance(byte[] addr) {
    return getRepository().getBalance(addr);
}

@Override
public BigInteger addBalance(byte[] addr, BigInteger value) {
    return getRepository().addBalance(addr, value);
}

@Override
public Set<byte[]> getAccountsKeys() {
    return getRepository().getAccountsKeys();
}

@Override
public void dumpState(Block block, long gasUsed, int txNumber, byte[] txHash) {
    getRepository().dumpState(block, gasUsed, txNumber, txHash);
}

@Override
public Repository startTracking() {
    return getRepository().startTracking();
}

@Override
```

```
    public void flush() {
        getRepository().flush();
    }

    @Override
    public void flushNoReconnect() {
        getRepository().flushNoReconnect();
    }

    @Override
    public void commit() {
        getRepository().commit();
    }

    @Override
    public void rollback() {
        getRepository().rollback();
    }

    @Override
    public void syncToRoot(byte[] root) {
        getRepository().syncToRoot(root);
    }

    @Override
    public boolean isClosed() {
        return getRepository().isClosed();
    }

    @Override
    public void close() {
        getRepository().close();
    }

    @Override
    public void reset() {
        getRepository().reset();
    }

    @Override
    public void updateBatch(HashMap<ByteArrayWrapper, AccountState> accountStates,
HashMap<ByteArrayWrapper, ContractDetails> contractDetailes) {
        getRepository().updateBatch(accountStates, contractDetailes);
    }

    @Override
    public byte[] getRoot() {
        return getRepository().getRoot();
    }

    @Override
    public void loadAccount(byte[] addr, HashMap<ByteArrayWrapper, AccountState>
cacheAccounts, HashMap<ByteArrayWrapper, ContractDetails> cacheDetails) {
        getRepository().loadAccount(addr, cacheAccounts, cacheDetails);
    }

    @Override
    public Repository getSnapshotTo(byte[] root) {
        return getRepository().getSnapshotTo(root);
    }

    @Override
    public Repository clone() {
        return getSnapshotTo(getRoot());
    }

    @Override
    public int getStorageSize(byte[] addr) {
        return getRepository().getStorageSize(addr);
    }

    @Override
    public Set<DataWord> getStorageKeys(byte[] addr) {
        return getRepository().getStorageKeys(addr);
    }

    @Override
    public Map<DataWord, DataWord> getStorage(byte[] addr, @Nullable Collection<DataWord>
keys) {
        return getRepository().getStorage(addr, keys);
    }
}
```

36:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-

vm\src\main\java\org\ethereum\db\StateSource.java

```
/**
 * Created by Anton Nashatyrev on 29.11.2016.
 */
public class StateSource extends SourceChainBox<byte[], byte[], byte[], byte[]>
        implements HashedKeySource<byte[], byte[]> {

    // for debug purposes
    public static StateSource INST;

    JournalSource<byte[]> journalSource;
    NoDeleteSource<byte[], byte[]> noDeleteSource;

    ReadCache<byte[], byte[]> readCache;
    AbstractCachedSource<byte[], byte[]> writeCache;

    public StateSource(Source<byte[], byte[]> src, boolean pruningEnabled) {
        super(src);
        INST = this;
        add(readCache = new ReadCache.BytesKey<>(src).withMaxCapacity(16 * 1024 * 1024 / 
512)); // 512 - approx size of a node
        readCache.setFlushSource(true);
        writeCache = new AsyncWriteCache<byte[], byte[]>(readCache) {
            @Override
            protected WriteCache<byte[], byte[]> createCache(Source<byte[], byte[]> source) {
                WriteCache.BytesKey<byte[]> ret = new WriteCache.BytesKey<byte[]>(source, 
WriteCache.CacheType.SIMPLE);
                ret.withSizeEstimators(MemSizeEstimator.ByteArrayEstimator, 
MemSizeEstimator.ByteArrayEstimator);
                ret.setFlushSource(true);
                return ret;
            }
        }.withName("state");

        add(writeCache);

        if (pruningEnabled) {
            add(journalSource = new JournalSource<>(writeCache));
        } else {
            add(noDeleteSource = new NoDeleteSource<>(writeCache));
        }
    }
    public void setConfig(SystemProperties config) {
        int size = config.getConfig().getInt("cache.stateCacheSize");
        readCache.withMaxCapacity(size * 1024 * 1024 / 512); // 512 - approx size of a node
    }

    public void setCommonConfig(CommonConfig commonConfig) {
        if (journalSource != null) {
            journalSource.setJournalStore(commonConfig.cachedDbSource("journal"));
        }
    }

    public JournalSource<byte[]> getJournalSource() {
        return journalSource;
    }

    /**
     * Returns the source behind JournalSource
     */
    public Source<byte[], byte[]> getNoJournalSource() {
        return writeCache;
    }

    public AbstractCachedSource<byte[], byte[]> getWriteCache() {
        return writeCache;
    }

    public ReadCache<byte[], byte[]> getReadCache() {
        return readCache;
    }
}
```

37:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\main\java\org\ethereum\facade\Repository.java

```
import java.util.Collection;
import java.util.Map;
import java.util.Set;

public interface Repository {

    /**
     * @param addr - account to check
```

```
 * @return - true if account exist,
 * false otherwise
 */
boolean isExist(byte[] addr);


/**
 * Retrieve balance of an account
 *
 * @param addr of the account
 * @return balance of the account as a <code>BigInteger</code> value
 */
BigInteger getBalance(byte[] addr);


/**
 * Get current nonce of a given account
 *
 * @param addr of the account
 * @return value of the nonce
 */
BigInteger getNonce(byte[] addr);


/**
 * Retrieve the code associated with an account
 *
 * @param addr of the account
 * @return code in byte-array format
 */
byte[] getCode(byte[] addr);


/**
 * Retrieve storage value from an account for a given key
 *
 * @param addr of the account
 * @param key  associated with this value
 * @return data in the form of a <code>DataWord</code>
 */
DataWord getStorageValue(byte[] addr, DataWord key);
```

```
    /**
     * Retrieve storage size for a given account
     *
     * @param addr of the account
     * @return storage entries count
     */
    int getStorageSize(byte[] addr);

    /**
     * Retrieve all storage keys for a given account
     *
     * @param addr of the account
     * @return set of storage keys or empty set if account with specified address not exists
     */
    Set<DataWord> getStorageKeys(byte[] addr);

    /**
     * Retrieve storage entries from an account for given keys
     *
     * @param addr of the account
     * @param keys
     * @return storage entries for specified keys, or full storage if keys parameter is
<code>null</code>
     */
    Map<DataWord, DataWord> getStorage(byte[] addr, @Nullable Collection<DataWord> keys);
}
```

38:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\main\java\org\ethereum\trie\CollectFullSetOfNodes.java

```
/**
 * @author Roman Mandeleil
 * @since 29.08.2014
 */
public class CollectFullSetOfNodes implements TrieImpl.ScanAction {
    Set<ByteArrayWrapper> nodes = new HashSet<>();

    @Override
    public void doOnNode(byte[] hash, TrieImpl.Node node) {
        nodes.add(new ByteArrayWrapper(hash));
    }
    @Override
    public void doOnValue(byte[] nodeHash, TrieImpl.Node node, byte[] key, byte[] value) {
    }

    public Set<ByteArrayWrapper> getCollectedHashes() {
        return nodes;
    }
}
```

39:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\main\java\org\ethereum\trie\CountAllNodes.java

```
public class CountAllNodes implements TrieImpl.ScanAction {

    int counted = 0;

    @Override
    public void doOnNode(byte[] hash, TrieImpl.Node node) {
        ++counted;
    }

    @Override
    public void doOnValue(byte[] nodeHash, TrieImpl.Node node, byte[] key, byte[] value) {
    }

    public int getCounted() {
        return counted;
    }
}
```

40:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\main\java\org\ethereum\trie\Node.java

```
 * A Node in a Merkle Patricia Tree is one of the following:
 * <p>
 * - NULL (represented as the empty string)
 * - A two-item array [ key, value ] (1 key for 2-item array)
 * - A 17-item array [ v0 ... v15, vt ] (16 keys for 17-item array)
 * <p>
 * The idea is that in the event that there is a long path of nodes
 * each with only one element, we shortcut the descent by setting up
 * a [ key, value ] node, where the key gives the hexadecimal path
 * to descend, in the compact encoding described above, and the value
 * is just the hash of the node like in the standard radix tree.
 * <p>
 * R
 * / \
 * /   \
 * N     N
 * / \   / \
 * L   L L   L
 * <p>
 * Also, we add another conceptual change: internal nodes can no longer
 * have values, only leaves with no children of their own can; however,
 * since to be fully generic we want the key/value store to be able
 * store keys like 'dog' and 'doge' at the same time, we simply add
 * a terminator symbol (16) to the alphabet so there is never a value
 * "en-route" to another value.
 * <p>
 * Where a node is referenced inside a node, what is included is:
 * <p>
 * H(rlp.encode(x)) where H(x) = keccak(x) if len(x) &gt;= 32 else x
 * <p>
 * Note that when updating a trie, you will need to store the key/value pair (keccak(x), x)
 * in a persistent lookup table when you create a node with length &gt;= 32,
 * but if the node is shorter than that then you do not need to store anything
 * when length &lt; 32 for the obvious reason that the function f(x) = x is reversible.
 *
 * @author Nick Savers
 * @since 20.05.2014
 */
public class Node {

    /* RLP encoded value of the Trie-node */
    private final Value value;
    private boolean dirty;

    public Node(Value val) {
        this(val, false);
    }

    public Node(Value val, boolean dirty) {
        this.value = val;
        this.dirty = dirty;
    }
    public Node copy() {
        return new Node(this.value, this.dirty);
    }

    public boolean isDirty() {
        return dirty;
    }

    public void setDirty(boolean dirty) {
        this.dirty = dirty;
    }

    public Value getValue() {
        return value;
    }

    @Override
    public String toString() {
        return "[" + dirty + ", " + value + "]";
    }
}
```

41:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\main\java\org\ethereum\trie\SecureTrie.java

```
public class SecureTrie extends TrieImpl {

    public SecureTrie(byte[] root) {
        super(root);
    }

    public SecureTrie(Source<byte[], byte[]> cache) {
        super(cache, null);
    }

    public SecureTrie(Source<byte[], byte[]> cache, byte[] root) {
        super(cache, root);
    }

    @Override
    public byte[] get(byte[] key) {
        return super.get(sha3(key));
    }

    @Override
    public void put(byte[] key, byte[] value) {
        super.put(sha3(key), value);
    }

    @Override
    public void delete(byte[] key) {
        put(key, EMPTY_BYTE_ARRAY);
    }
}
```

42:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\main\java\org\ethereum\trie\TraceAllNodes.java

```
 * @since 29.08.2014
 */
public class TraceAllNodes implements TrieImpl.ScanAction {

    StringBuilder output = new StringBuilder();

    @Override
    public void doOnNode(byte[] hash, TrieImpl.Node node) {

        output.append(Hex.toHexString(hash)).append(" ==>
").append(node.toString()).append("\n");
    }

    @Override
    public void doOnValue(byte[] nodeHash, TrieImpl.Node node, byte[] key, byte[] value) {
    }

    public String getOutput() {
        return output.toString();
    }
}
```

43:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\main\java\org\ethereum\trie\Trie.java

```
 */
public interface Trie<V> extends Source<byte[], V> {
    byte[] getRootHash();

    void setRoot(byte[] root);

    /**
     * Recursively delete all nodes from root
     */
    void clear();
}
```

44:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\main\java\org\ethereum\trie\TrieImpl.java

```
import org.ethereum.datasource.inmem.HashMapDB;
import org.ethereum.util.FastByteComparisons;
import org.ethereum.util.RLP;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.spongycastle.util.encoders.Hex;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

import static org.apache.commons.lang3.concurrent.ConcurrentUtils.constantFuture;
import static org.ethereum.crypto.HashUtil.EMPTY_TRIE_HASH;
import static org.ethereum.util.ByteUtil.EMPTY_BYTE_ARRAY;
import static org.ethereum.util.ByteUtil.toHexString;
import static org.ethereum.util.RLP.*;

/**
 * Created by Anton Nashatyrev on 07.02.2017.
 */
public class TrieImpl implements Trie<byte[]> {
    private final static Object NULL_NODE = new Object();
    private final static int MIN_BRANCHES_CONCURRENTLY = 3;
    private static ExecutorService executor;

    private static final Logger logger = LoggerFactory.getLogger("state");
```

```
public static ExecutorService getExecutor() {
    if (executor == null) {
        executor = Executors.newFixedThreadPool(4,
                new ThreadFactoryBuilder().setNameFormat("trie-calc-thread-%d").build());
    }
    return executor;
}

public enum NodeType {
    BranchNode,
    KVNodeValue,
    KVNodeNode
}

public final class Node {
    private byte[] hash = null;
    private byte[] rlp = null;
    private RLP.LList parsedRlp = null;
    private boolean dirty = false;

    private Object[] children = null;

    // new empty BranchNode
    public Node() {
        children = new Object[17];
        dirty = true;
    }

    // new KVNode with key and (value or node)
    public Node(TrieKey key, Object valueOrNode) {
        this(new Object[]{key, valueOrNode});
        dirty = true;
    }

    // new Node with hash or RLP
    public Node(byte[] hashOrRlp) {
        if (hashOrRlp.length == 32) {
            this.hash = hashOrRlp;
        } else {
            this.rlp = hashOrRlp;
        }
    }
    private Node(RLP.LList parsedRlp) {
        this.parsedRlp = parsedRlp;
        this.rlp = parsedRlp.getEncoded();
    }

    private Node(Object[] children) {
        this.children = children;
    }

    public boolean resolveCheck() {
        if (rlp != null || parsedRlp != null || hash == null) {
            return true;
        }
        rlp = getHash(hash);
        return rlp != null;
    }

    private void resolve() {
        if (!resolveCheck()) {
            logger.error("Invalid Trie state, can't resolve hash " + toHexString(hash));
            throw new RuntimeException("Invalid Trie state, can't resolve hash " +
toHexString(hash));
        }
    }

    public byte[] encode() {
        return encode(1, true);
    }

    private byte[] encode(final int depth, boolean forceHash) {
        if (!dirty) {
            return hash != null ? encodeElement(hash) : rlp;
        } else {
            NodeType type = getType();
            byte[] ret;
            if (type == NodeType.BranchNode) {
                if (depth == 1 && async) {
                    // parallelize encode() on the first trie level only and if there are at least
                    // MIN_BRANCHES_CONCURRENTLY branches are modified
                    final Object[] encoded = new Object[17];
                    int encodeCnt = 0;
                    for (int i = 0; i < 16; i++) {
                        final Node child = branchNodeGetChild(i);
                        if (child == null) {
                            encoded[i] = EMPTY_ELEMENT_RLP;
                        } else if (!child.dirty) {
                            encoded[i] = child.encode(depth + 1, false);
                        } else {
                            encodeCnt++;
                        }
                    }
                    for (int i = 0; i < 16; i++) {
                        if (encoded[i] == null) {
                            final Node child = branchNodeGetChild(i);
                            if (encodeCnt >= MIN_BRANCHES_CONCURRENTLY) {
                                encoded[i] = getExecutor().submit(() -> child.encode(depth + 1, false));
                            } else {
                                encoded[i] = child.encode(depth + 1, false);
                            }
                        }
                    }
                    byte[] value = branchNodeGetValue();
                    encoded[16] = constantFuture(encodeElement(value));
                    try {
                        ret = encodeRlpListFutures(encoded);
                    } catch (Exception e) {
                        throw new RuntimeException(e);
                    }
                } else {
                    byte[][] encoded = new byte[17][];
                    for (int i = 0; i < 16; i++) {
                        Node child = branchNodeGetChild(i);
                        encoded[i] = child == null ? EMPTY_ELEMENT_RLP : child.encode(depth + 1,
false);
                    }
                    byte[] value = branchNodeGetValue();
                    encoded[16] = encodeElement(value);
                    ret = encodeList(encoded);
                }
            } else if (type == NodeType.KVNodeNode) {
                ret = encodeList(encodeElement(kvNodeGetKey().toPacked()),
kvNodeGetChildNode().encode(depth + 1, false));
            } else {
                byte[] value = kvNodeGetValue();
                ret = encodeList(encodeElement(kvNodeGetKey().toPacked()),
                        encodeElement(value == null ? EMPTY_BYTE_ARRAY : value));
            }
            if (hash != null) {
                deleteHash(hash);
            }
            dirty = false;
            if (ret.length < 32 && !forceHash) {
                rlp = ret;
                return ret;
            } else {
                hash = HashUtil.sha3(ret);
                addHash(hash, ret);
                return encodeElement(hash);
            }
        }
    }

    @SafeVarargs
    private final byte[] encodeRlpListFutures(Object... list) throws ExecutionException,
InterruptedException {
        byte[][] vals = new byte[list.length][];
        for (int i = 0; i < list.length; i++) {
            if (list[i] instanceof Future) {
                vals[i] = ((Future<byte[]>) list[i]).get();
            } else {
                vals[i] = (byte[]) list[i];
            }
        }
        return encodeList(vals);
    }

    private void parse() {
        if (children != null) {
            return;
        }
        resolve();

        RLP.LList list = parsedRlp == null ? RLP.decodeLazyList(rlp) : parsedRlp;

        if (list.size() == 2) {
```

```
        children = new Object[2];
        TrieKey key = TrieKey.fromPacked(list.getBytes(0));
        children[0] = key;
        if (key.isTerminal()) {
            children[1] = list.getBytes(1);
        } else {
            children[1] = list.isList(1) ? new Node(list.getList(1)) : new Node(list.getBytes(1));
        }
    } else {
        children = new Object[17];
        parsedRlp = list;
    }
}

public Node branchNodeGetChild(int hex) {
    parse();
    assert getType() == NodeType.BranchNode;
    Object n = children[hex];
    if (n == null && parsedRlp != null) {
        if (parsedRlp.isList(hex)) {
            n = new Node(parsedRlp.getList(hex));
        } else {
            byte[] bytes = parsedRlp.getBytes(hex);
            if (bytes.length == 0) {
                n = NULL_NODE;
            } else {
                n = new Node(bytes);
            }
        }
        children[hex] = n;
    }
    return n == NULL_NODE ? null : (Node) n;
}

public Node branchNodeSetChild(int hex, Node node) {
    parse();
    assert getType() == NodeType.BranchNode;
    children[hex] = node == null ? NULL_NODE : node;
    dirty = true;
    return this;
}
```

```
public byte[] branchNodeGetValue() {
    parse();
    assert getType() == NodeType.BranchNode;
    Object n = children[16];
    if (n == null && parsedRlp != null) {
        byte[] bytes = parsedRlp.getBytes(16);
        if (bytes.length == 0) {
            n = NULL_NODE;
        } else {
            n = bytes;
        }
        children[16] = n;
    }
    return n == NULL_NODE ? null : (byte[]) n;
}

public Node branchNodeSetValue(byte[] val) {
    parse();
    assert getType() == NodeType.BranchNode;
    children[16] = val == null ? NULL_NODE : val;
    dirty = true;
    return this;
}

public int branchNodeCompactIdx() {
    parse();
    assert getType() == NodeType.BranchNode;
    int cnt = 0;
    int idx = -1;
    for (int i = 0; i < 16; i++) {
        if (branchNodeGetChild(i) != null) {
            cnt++;
            idx = i;
            if (cnt > 1) {
                return -1;
            }
        }
    }
    return cnt > 0 ? idx : (branchNodeGetValue() == null ? -1 : 16);
}

public boolean branchNodeCanCompact() {
    parse();
    assert getType() == NodeType.BranchNode;
    int cnt = 0;
    for (int i = 0; i < 16; i++) {
        cnt += branchNodeGetChild(i) == null ? 0 : 1;
        if (cnt > 1) {
            return false;
        }
    }
    return cnt == 0 || branchNodeGetValue() == null;
}

public TrieKey kvNodeGetKey() {
    parse();
    assert getType() != NodeType.BranchNode;
    return (TrieKey) children[0];
}

public Node kvNodeGetChildNode() {
    parse();
    assert getType() == NodeType.KVNodeNode;
    return (Node) children[1];
}

public byte[] kvNodeGetValue() {
    parse();
    assert getType() == NodeType.KVNodeValue;
    return (byte[]) children[1];
}

public Node kvNodeSetValue(byte[] value) {
    parse();
    assert getType() == NodeType.KVNodeValue;
    children[1] = value;
    dirty = true;
    return this;
}

public Object kvNodeGetValueOrNode() {
    parse();
    assert getType() != NodeType.BranchNode;
    return children[1];
```

```
}

public Node kvNodeSetValueOrNode(Object valueOrNode) {
    parse();
    assert getType() != NodeType.BranchNode;
    children[1] = valueOrNode;
    dirty = true;
    return this;
}

public NodeType getType() {
    parse();

    return children.length == 17 ? NodeType.BranchNode :
            (children[1] instanceof Node ? NodeType.KVNodeNode : NodeType.KVNodeValue);
}

public void dispose() {
    if (hash != null) {
        deleteHash(hash);
    }
}

public Node invalidate() {
    dirty = true;
    return this;
}

/***********  Dump methods  ************/

public String dumpStruct(String indent, String prefix) {
    String ret = indent + prefix + getType() + (dirty ? " *" : "") +
            (hash == null ? "" : "(hash: " + Hex.toHexString(hash).substring(0, 6) + ")");
    if (getType() == NodeType.BranchNode) {
        byte[] value = branchNodeGetValue();
        ret += (value == null ? "" : " [T] = " + Hex.toHexString(value)) + "\n";
        for (int i = 0; i < 16; i++) {
            Node child = branchNodeGetChild(i);
            if (child != null) {
                ret += child.dumpStruct(indent + "  ", "[" + i + "] ");
            }
        }
    } else if (getType() == NodeType.KVNodeNode) {
        ret += " [" + kvNodeGetKey() + "]\n";
        ret += kvNodeGetChildNode().dumpStruct(indent + "  ", "");
    } else {
        ret += " [" + kvNodeGetKey() + "] = " + Hex.toHexString(kvNodeGetValue()) + "\n";
    }
    return ret;
}

public List<String> dumpTrieNode(boolean compact) {
    List<String> ret = new ArrayList<>();
    if (hash != null) {
        ret.add(hash2str(hash, compact) + " ==> " + dumpContent(false, compact));
    }

    if (getType() == NodeType.BranchNode) {
        for (int i = 0; i < 16; i++) {
            Node child = branchNodeGetChild(i);
            if (child != null) {
                ret.addAll(child.dumpTrieNode(compact));
            }
        }
    } else if (getType() == NodeType.KVNodeNode) {
        ret.addAll(kvNodeGetChildNode().dumpTrieNode(compact));
    }
    return ret;
}

private String dumpContent(boolean recursion, boolean compact) {
    if (recursion && hash != null) {
        return hash2str(hash, compact);
    }
    String ret;
    if (getType() == NodeType.BranchNode) {
        ret = "[";
        for (int i = 0; i < 16; i++) {
            Node child = branchNodeGetChild(i);
            ret += i == 0 ? "" : ",";
            ret += child == null ? "" : child.dumpContent(true, compact);
        }
        byte[] value = branchNodeGetValue();
```

```
                ret += value == null ? "" : ", " + val2str(value, compact);
                ret += "]";
            } else if (getType() == NodeType.KVNodeNode) {
                ret = "[<" + kvNodeGetKey() + ">, " + kvNodeGetChildNode().dumpContent(true,
compact) + "]";
            } else {
                ret = "[<" + kvNodeGetKey() + ">, " + val2str(kvNodeGetValue(), compact) + "]";
            }
            return ret;
        }

        @Override
        public String toString() {
            return getType() + (dirty ? " *" : "") + (hash == null ? "" : "(hash: " + toHexString(hash) + "
)");
        }
    }

    public interface ScanAction {

        void doOnNode(byte[] hash, Node node);

        void doOnValue(byte[] nodeHash, Node node, byte[] key, byte[] value);
    }

    private Source<byte[], byte[]> cache;
    private Node root;
    private boolean async = true;

    public TrieImpl() {
        this((byte[]) null);
    }

    public TrieImpl(byte[] root) {
        this(new HashMapDB<byte[]>(), root);
    }

    public TrieImpl(Source<byte[], byte[]> cache) {
        this(cache, null);
    }

    public TrieImpl(Source<byte[], byte[]> cache, byte[] root) {
```

```
    this.cache = cache;
    setRoot(root);
}

public void setAsync(boolean async) {
    this.async = async;
}

private void encode() {
    if (root != null) {
        root.encode();
    }
}

@Override
public void setRoot(byte[] root) {
    if (root != null && !FastByteComparisons.equal(root, EMPTY_TRIE_HASH)) {
        this.root = new Node(root);
    } else {
        this.root = null;
    }

}

private boolean hasRoot() {
    return root != null && root.resolveCheck();
}

public Source<byte[], byte[]> getCache() {
    return cache;
}

private byte[] getHash(byte[] hash) {
    return cache.get(hash);
}

private void addHash(byte[] hash, byte[] ret) {
    cache.put(hash, ret);
}

private void deleteHash(byte[] hash) {
    cache.delete(hash);
```

```
}


@Override
public byte[] get(byte[] key) {
    if (!hasRoot()) {
        return null; // treating unknown root hash as empty trie
    }
    TrieKey k = TrieKey.fromNormal(key);
    return get(root, k);
}

private byte[] get(Node n, TrieKey k) {
    if (n == null) {
        return null;
    }

    NodeType type = n.getType();
    if (type == NodeType.BranchNode) {
        if (k.isEmpty()) {
            return n.branchNodeGetValue();
        }
        Node childNode = n.branchNodeGetChild(k.getHex(0));
        return get(childNode, k.shift(1));
    } else {
        TrieKey k1 = k.matchAndShift(n.kvNodeGetKey());
        if (k1 == null) {
            return null;
        }
        if (type == NodeType.KVNodeValue) {
            return k1.isEmpty() ? n.kvNodeGetValue() : null;
        } else {
            return get(n.kvNodeGetChildNode(), k1);
        }
    }
}

@Override
public void put(byte[] key, byte[] value) {
    TrieKey k = TrieKey.fromNormal(key);
    if (root == null) {
        if (value != null && value.length > 0) {
            root = new Node(k, value);
        }
    } else {
        if (value == null || value.length == 0) {
            root = delete(root, k);
        } else {
            root = insert(root, k, value);
        }
    }
}

private Node insert(Node n, TrieKey k, Object nodeOrValue) {
    NodeType type = n.getType();
    if (type == NodeType.BranchNode) {
        if (k.isEmpty()) {
            return n.branchNodeSetValue((byte[]) nodeOrValue);
        }
        Node childNode = n.branchNodeGetChild(k.getHex(0));
        if (childNode != null) {
            return n.branchNodeSetChild(k.getHex(0), insert(childNode, k.shift(1), nodeOrValue));
        } else {
            TrieKey childKey = k.shift(1);
            Node newChildNode;
            if (!childKey.isEmpty()) {
                newChildNode = new Node(childKey, nodeOrValue);
            } else {
                newChildNode = nodeOrValue instanceof Node ?
                        (Node) nodeOrValue : new Node(childKey, nodeOrValue);
            }
            return n.branchNodeSetChild(k.getHex(0), newChildNode);
        }
    } else {
        TrieKey currentNodeKey = n.kvNodeGetKey();
        TrieKey commonPrefix = k.getCommonPrefix(currentNodeKey);
        if (commonPrefix.isEmpty()) {
            Node newBranchNode = new Node();
            insert(newBranchNode, currentNodeKey, n.kvNodeGetValueOrNode());
            insert(newBranchNode, k, nodeOrValue);
            n.dispose();
            return newBranchNode;
        } else if (commonPrefix.equals(k)) {
            return n.kvNodeSetValueOrNode(nodeOrValue);
        } else if (commonPrefix.equals(currentNodeKey)) {
            insert(n.kvNodeGetChildNode(), k.shift(commonPrefix.getLength()), nodeOrValue);
            return n.invalidate();
        } else {
            Node newBranchNode = new Node();
            Node newKvNode = new Node(commonPrefix, newBranchNode);
            // TODO can be optimized
            insert(newKvNode, currentNodeKey, n.kvNodeGetValueOrNode());
            insert(newKvNode, k, nodeOrValue);
            n.dispose();
            return newKvNode;
        }
    }
}

@Override
public void delete(byte[] key) {
    TrieKey k = TrieKey.fromNormal(key);
    if (root != null) {
        root = delete(root, k);
    }
}

private Node delete(Node n, TrieKey k) {
    NodeType type = n.getType();
    Node newKvNode;
    if (type == NodeType.BranchNode) {
        if (k.isEmpty()) {
            n.branchNodeSetValue(null);
        } else {
            int idx = k.getHex(0);
            Node child = n.branchNodeGetChild(idx);
            if (child == null) {
                return n; // no key found
            }

            Node newNode = delete(child, k.shift(1));
            n.branchNodeSetChild(idx, newNode);
            if (newNode != null) {
                return n; // newNode != null thus number of children didn't decrease
            }
        }
```

```
            // child node or value was deleted and the branch node may need to be compacted
            int compactIdx = n.branchNodeCompactIdx();
            if (compactIdx < 0) {
                return n; // no compaction is required
            }

            // only value or a single child left - compact branch node to kvNode
            n.dispose();
            if (compactIdx == 16) { // only value left
                return new Node(TrieKey.empty(true), n.branchNodeGetValue());
            } else { // only single child left
                newKvNode = new Node(TrieKey.singleHex(compactIdx),
n.branchNodeGetChild(compactIdx));
            }
        } else { // n - kvNode
            TrieKey k1 = k.matchAndShift(n.kvNodeGetKey());
            if (k1 == null) {
                // no key found
                return n;
            } else if (type == NodeType.KVNodeValue) {
                if (k1.isEmpty()) {
                    // delete this kvNode
                    n.dispose();
                    return null;
                } else {
                    // else no key found
                    return n;
                }
            } else {
                Node newChild = delete(n.kvNodeGetChildNode(), k1);
                if (newChild == null) {
                    throw new RuntimeException("Shouldn't happen");
                }
                newKvNode = n.kvNodeSetValueOrNode(newChild);
            }
        }

        // if we get here a new kvNode was created, now need to check
        // if it should be compacted with child kvNode
        Node newChild = newKvNode.kvNodeGetChildNode();
        if (newChild.getType() != NodeType.BranchNode) {
            // two kvNodes should be compacted into a single one
            TrieKey newKey = newKvNode.kvNodeGetKey().concat(newChild.kvNodeGetKey());
            Node newNode = new Node(newKey, newChild.kvNodeGetValueOrNode());
            newChild.dispose();
            newKvNode.dispose();
            return newNode;
        } else {
            // no compaction needed
            return newKvNode;
        }
    }

    @Override
    public byte[] getRootHash() {
        encode();
        return root != null ? root.hash : EMPTY_TRIE_HASH;
    }

    @Override
    public void clear() {
        throw new RuntimeException("Not implemented yet");
    }

    @Override
    public boolean flush() {
        if (root != null && root.dirty) {
            // persist all dirty nodes to underlying Source
            encode();
            // release all Trie Node instances for GC
            root = new Node(root.hash);
            return true;
        } else {
            return false;
        }
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) {
            return true;
        }
        if (o == null || getClass() != o.getClass()) {
            return false;
        }

        TrieImpl trieImpl1 = (TrieImpl) o;

        return FastByteComparisons.equal(getRootHash(), trieImpl1.getRootHash());

    }

    public String dumpStructure() {
        return root == null ? "<empty>" : root.dumpStruct("", "");
    }

    public String dumpTrie() {
        return dumpTrie(true);
    }

    public String dumpTrie(boolean compact) {
        if (root == null) {
            return "<empty>";
        }
        encode();
        StrBuilder ret = new StrBuilder();
        List<String> strings = root.dumpTrieNode(compact);
        ret.append("Root: " + hash2str(getRootHash(), compact) + "\n");
        for (String s : strings) {
            ret.append(s).append('\n');
        }
        return ret.toString();
    }

    public void scanTree(ScanAction scanAction) {
        scanTree(root, TrieKey.empty(false), scanAction);
    }

    public void scanTree(Node node, TrieKey k, ScanAction scanAction) {
        if (node == null) {
            return;
        }
        if (node.hash != null) {
            scanAction.doOnNode(node.hash, node);
        }
        if (node.getType() == NodeType.BranchNode) {
            if (node.branchNodeGetValue() != null) {
                scanAction.doOnValue(node.hash, node, k.toNormal(), node.branchNodeGetValue());
            }
            for (int i = 0; i < 16; i++) {
                scanTree(node.branchNodeGetChild(i), k.concat(TrieKey.singleHex(i)), scanAction);
            }
        } else if (node.getType() == NodeType.KVNodeNode) {
            scanTree(node.kvNodeGetChildNode(), k.concat(node.kvNodeGetKey()), scanAction);
        } else {
            scanAction.doOnValue(node.hash, node, k.concat(node.kvNodeGetKey()).toNormal(),
node.kvNodeGetValue());
        }
    }


    private static String hash2str(byte[] hash, boolean shortHash) {
        String ret = Hex.toHexString(hash);
        return "0x" + (shortHash ? ret.substring(0, 8) : ret);
    }

    private static String val2str(byte[] val, boolean shortHash) {
        String ret = Hex.toHexString(val);
        if (val.length > 16) {
            ret = ret.substring(0, 10) + "... len " + val.length;
        }
        return "\"" + ret + "\"";
    }
}
```

45:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\main\java\org\ethereum\trie\TrieKey.java

```
 * Created by Anton Nashatyrev on 13.02.2017.
 */
public final class TrieKey {
    public static final int ODD_OFFSET_FLAG = 0x1;
    public static final int TERMINATOR_FLAG = 0x2;
    private final byte[] key;
    private final int off;
    private final boolean terminal;

    public static TrieKey fromNormal(byte[] key) {
        return new TrieKey(key);
    }

    public static TrieKey fromPacked(byte[] key) {
        return new TrieKey(key, ((key[0] >> 4) & ODD_OFFSET_FLAG) != 0 ? 1 : 2, ((key[0] >> 4) &
TERMINATOR_FLAG) != 0);
    }

    public static TrieKey empty(boolean terminal) {
        return new TrieKey(EMPTY_BYTE_ARRAY, 0, terminal);
    }

    public static TrieKey singleHex(int hex) {
        TrieKey ret = new TrieKey(new byte[1], 1, false);
        ret.setHex(0, hex);
        return ret;
    }

    public TrieKey(byte[] key, int off, boolean terminal) {
        this.terminal = terminal;
        this.off = off;
        this.key = key;
    }

    private TrieKey(byte[] key) {
        this(key, 0, true);
    }

    public byte[] toPacked() {
        int flags = ((off & 1) != 0 ? ODD_OFFSET_FLAG : 0) | (terminal ? TERMINATOR_FLAG : 0);
        byte[] ret = new byte[getLength() / 2 + 1];
        int toCopy = (flags & ODD_OFFSET_FLAG) != 0 ? ret.length : ret.length - 1;
        System.arraycopy(key, key.length - toCopy, ret, ret.length - toCopy, toCopy);
        ret[0] &= 0x0F;
        ret[0] |= flags << 4;
        return ret;
    }

    public byte[] toNormal() {
        if ((off & 1) != 0) {
            throw new RuntimeException("Can't convert a key with odd number of hexes to normal: " +
this);
```

```
        }
        int arrLen = key.length - off / 2;
        byte[] ret = new byte[arrLen];
        System.arraycopy(key, key.length - arrLen, ret, 0, arrLen);
        return ret;
    }

    public boolean isTerminal() {
        return terminal;
    }

    public boolean isEmpty() {
        return getLength() == 0;
    }

    public TrieKey shift(int hexCnt) {
        return new TrieKey(this.key, off + hexCnt, terminal);
    }

    public TrieKey getCommonPrefix(TrieKey k) {
        // TODO can be optimized
        int prefixLen = 0;
        int thisLength = getLength();
        int kLength = k.getLength();
        while (prefixLen < thisLength && prefixLen < kLength && getHex(prefixLen) ==
k.getHex(prefixLen)) {
            prefixLen++;
        }
        byte[] prefixKey = new byte[(prefixLen + 1) >> 1];
        TrieKey ret = new TrieKey(prefixKey, (prefixLen & 1) == 0 ? 0 : 1,
                prefixLen == getLength() && prefixLen == k.getLength() && terminal && k.isTerminal());
        for (int i = 0; i < prefixLen; i++) {
            ret.setHex(i, k.getHex(i));
        }
        return ret;
    }

    public TrieKey matchAndShift(TrieKey k) {
        int len = getLength();
        int kLen = k.getLength();
        if (len < kLen) {
            return null;
        }

        if ((off & 1) == (k.off & 1)) {
            // optimization to compare whole keys bytes
            if ((off & 1) == 1) {
                if (getHex(0) != k.getHex(0)) {
                    return null;
                }
            }
            int idx1 = (off + 1) >> 1;
            int idx2 = (k.off + 1) >> 1;
            int l = kLen >> 1;
            for (int i = 0; i < l; i++, idx1++, idx2++) {
                if (key[idx1] != k.key[idx2]) {
                    return null;
                }
            }
        } else {
            for (int i = 0; i < kLen; i++) {
                if (getHex(i) != k.getHex(i)) {
                    return null;
                }
            }
        }
        return shift(kLen);
    }

    public int getLength() {
        return (key.length << 1) - off;
    }

    private void setHex(int idx, int hex) {
        int byteIdx = (off + idx) >> 1;
        if (((off + idx) & 1) == 0) {
            key[byteIdx] &= 0x0F;
            key[byteIdx] |= hex << 4;
        } else {
            key[byteIdx] &= 0xF0;
            key[byteIdx] |= hex;
        }
    }
```

```
public int getHex(int idx) {
    byte b = key[(off + idx) >> 1];
    return (((off + idx) & 1) == 0 ? (b >> 4) : b) & 0xF;
}

public TrieKey concat(TrieKey k) {
    if (isTerminal()) {
        throw new RuntimeException("Can' append to terminal key: " + this + " + " + k);
    }
    int len = getLength();
    int kLen = k.getLength();
    int newLen = len + kLen;
    byte[] newKeyBytes = new byte[(newLen + 1) >> 1];
    TrieKey ret = new TrieKey(newKeyBytes, newLen & 1, k.isTerminal());
    for (int i = 0; i < len; i++) {
        ret.setHex(i, getHex(i));
    }
    for (int i = 0; i < kLen; i++) {
        ret.setHex(len + i, k.getHex(i));
    }
    return ret;
}

@Override
public boolean equals(Object obj) {
    TrieKey k = (TrieKey) obj;
    int len = getLength();

    if (len != k.getLength()) {
        return false;
    }
    // TODO can be optimized
    for (int i = 0; i < len; i++) {
        if (getHex(i) != k.getHex(i)) {
            return false;
        }
    }
    return isTerminal() == k.isTerminal();
}

@Override
public String toString() {
        return toHexString(key).substring(off) + (isTerminal() ? "T" : "");
    }
}
```

46:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\main\java\org\ethereum\util\ALock.java

```
 * <p>
 * try (ALock l = wLock.lock()) {
 * // do smth under lock
 * }
 * <p>
 * Created by Anton Nashatyrev on 27.01.2017.
 */
public final class ALock implements AutoCloseable {
    private final Lock lock;

    public ALock(Lock l) {
        this.lock = l;
    }

    public final ALock lock() {
        this.lock.lock();
        return this;
    }

    @Override
    public final void close() {
        this.lock.unlock();
    }
}
```

47:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\main\java\org\ethereum\util\BIUtil.java

```
public class BIUtil {



    /**
     * @param value - not null
     * @return true - if the param is zero
     */
    public static boolean isZero(BigInteger value) {
        return value.compareTo(BigInteger.ZERO) == 0;
}

/**
 * @param valueA - not null
 * @param valueB - not null
 * @return true - if the valueA is equal to valueB is zero
 */
public static boolean isEqual(BigInteger valueA, BigInteger valueB) {
    return valueA.compareTo(valueB) == 0;
}

/**
 * @param valueA - not null
 * @param valueB - not null
 * @return true - if the valueA is not equal to valueB is zero
 */
public static boolean isNotEqual(BigInteger valueA, BigInteger valueB) {
    return !isEqual(valueA, valueB);
}

/**
 * @param valueA - not null
 * @param valueB - not null
 * @return true - if the valueA is less than valueB is zero
 */
public static boolean isLessThan(BigInteger valueA, BigInteger valueB) {
    return valueA.compareTo(valueB) < 0;
}

/**
 * @param valueA - not null
 * @param valueB - not null
 * @return true - if the valueA is more than valueB is zero
 */
public static boolean isMoreThan(BigInteger valueA, BigInteger valueB) {
    return valueA.compareTo(valueB) > 0;
}


/**
 * @param valueA - not null
 * @param valueB - not null
```

```
     * @return sum - valueA + valueB
     */
    public static BigInteger sum(BigInteger valueA, BigInteger valueB) {
        return valueA.add(valueB);
    }



    /**
     * @param data = not null
     * @return new positive BigInteger
     */
    public static BigInteger toBI(byte[] data) {
        return new BigInteger(1, data);
    }

    /**
     * @param data = not null
     * @return new positive BigInteger
     */
    public static BigInteger toBI(long data) {
        return BigInteger.valueOf(data);
    }



    public static boolean isPositive(BigInteger value) {
        return value.signum() > 0;
    }

    public static boolean isCovers(BigInteger covers, BigInteger value) {
        return !isNotCovers(covers, value);
    }

    public static boolean isNotCovers(BigInteger covers, BigInteger value) {
        return covers.compareTo(value) < 0;
    }



    public static void transfer(Repository repository, byte[] fromAddr, byte[] toAddr, BigInteger
value) {
        repository.addBalance(fromAddr, value.negate());
        repository.addBalance(toAddr, value);
    }
    public static boolean exitLong(BigInteger value) {

        return (value.compareTo(new BigInteger(Long.MAX_VALUE + ""))) > -1;
    }

    public static boolean isIn20PercentRange(BigInteger first, BigInteger second) {
        BigInteger five = BigInteger.valueOf(5);
        BigInteger limit = first.add(first.divide(five));
        return !isMoreThan(second, limit);
    }

    public static BigInteger max(BigInteger first, BigInteger second) {
        return first.compareTo(second) < 0 ? second : first;
    }

    /**
     * Returns a result of safe addition of two {@code int} values
     * {@code Integer.MAX_VALUE} is returned if overflow occurs
     */
    public static int addSafely(int a, int b) {
        long res = (long) a + (long) b;
        return res > Integer.MAX_VALUE ? Integer.MAX_VALUE : (int) res;
    }
}

48:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-
vm\src\main\java\org\ethereum\util\blockchain\EtherUtil.java
 */
public class EtherUtil {
    public enum Unit {
        WEI(BigInteger.valueOf(1)),
        GWEI(BigInteger.valueOf(1_000_000_000)),
        SZABO(BigInteger.valueOf(1_000_000_000_000L)),
        FINNEY(BigInteger.valueOf(1_000_000_000_000_000L)),
        ETHER(BigInteger.valueOf(1_000_000_000_000_000_000L));

        BigInteger i;

        Unit(BigInteger i) {
            this.i = i;
        }
    }

    public static BigInteger convert(long amount, Unit unit) {
        return BigInteger.valueOf(amount).multiply(unit.i);
    }
}
```

49:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\main\java\org\ethereum\util\BuildInfo.java

```
import java.io.InputStream;
import java.util.Properties;

public class BuildInfo {

    private static final Logger logger = LoggerFactory.getLogger("general");

    public static String buildHash;
    public static String buildTime;
    public static String buildBranch;

    static {
        try {
            Properties props = new Properties();
            InputStream is = BuildInfo.class.getResourceAsStream("/build-info.properties");

            if (is != null) {
                props.load(is);

                buildHash = props.getProperty("build.hash");
                buildTime = props.getProperty("build.time");
                buildBranch = props.getProperty("build.branch");
            } else {
                logger.warn("File not found `build-info.properties`. Run `gradle build` to generate it");
            }
        } catch (IOException e) {
            logger.error("Error reading /build-info.properties", e);
        }
    }

    public static void printInfo() {
        logger.debug("git.hash: [{}]", buildHash);
        logger.debug("build.time: {}", buildTime);
    }
}
```

50:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\main\java\org\ethereum\util\ByteArrayMap.java

```
/**
 * Created by Anton Nashatyrev on 06.10.2016.
 */
public class ByteArrayMap<V> implements Map<byte[], V> {
    private final Map<ByteArrayWrapper, V> delegate;

    public ByteArrayMap() {
        this(new HashMap<ByteArrayWrapper, V>());
    }

    public ByteArrayMap(Map<ByteArrayWrapper, V> delegate) {
        this.delegate = delegate;
    }

    @Override
    public int size() {
        return delegate.size();
    }

    @Override
    public boolean isEmpty() {
        return delegate.isEmpty();
    }

    @Override
    public boolean containsKey(Object key) {
        return delegate.containsKey(new ByteArrayWrapper((byte[]) key));
    }

    @Override
    public boolean containsValue(Object value) {
        return delegate.containsValue(value);
    }

    @Override
    public V get(Object key) {
        return delegate.get(new ByteArrayWrapper((byte[]) key));
```

```
}

@Override
public V put(byte[] key, V value) {
    return delegate.put(new ByteArrayWrapper(key), value);
}

@Override
public V remove(Object key) {
    return delegate.remove(new ByteArrayWrapper((byte[]) key));
}

@Override
public void putAll(Map<? extends byte[], ? extends V> m) {
    for (Entry<? extends byte[], ? extends V> entry : m.entrySet()) {
        delegate.put(new ByteArrayWrapper(entry.getKey()), entry.getValue());
    }
}

@Override
public void clear() {
    delegate.clear();
}

@Override
public Set<byte[]> keySet() {
    return new ByteArraySet(new SetAdapter<>(delegate));
}

@Override
public Collection<V> values() {
    return delegate.values();
}

@Override
public Set<Entry<byte[], V>> entrySet() {
    return new MapEntrySet(delegate.entrySet());
}

@Override
public boolean equals(Object o) {
    return delegate.equals(o);
```

```
}

@Override
public int hashCode() {
    return delegate.hashCode();
}

@Override
public String toString() {
    return delegate.toString();
}

private class MapEntrySet implements Set<Map.Entry<byte[], V>> {
    private final Set<Map.Entry<ByteArrayWrapper, V>> delegate;

    private MapEntrySet(Set<Entry<ByteArrayWrapper, V>> delegate) {
        this.delegate = delegate;
    }

    @Override
    public int size() {
        return delegate.size();
    }

    @Override
    public boolean isEmpty() {
        return delegate.isEmpty();
    }

    @Override
    public boolean contains(Object o) {
        throw new RuntimeException("Not implemented");
    }

    @Override
    public Iterator<Entry<byte[], V>> iterator() {
        final Iterator<Entry<ByteArrayWrapper, V>> it = delegate.iterator();
        return new Iterator<Entry<byte[], V>>() {

            @Override
            public boolean hasNext() {
                return it.hasNext();
            }

            @Override
            public Entry<byte[], V> next() {
                Entry<ByteArrayWrapper, V> next = it.next();
                return new AbstractMap.SimpleImmutableEntry(next.getKey().getData(),
next.getValue());
            }

            @Override
            public void remove() {
                it.remove();
            }
        };
    }

    @Override
    public Object[] toArray() {
        throw new RuntimeException("Not implemented");
    }

    @Override
    public <T> T[] toArray(T[] a) {
        throw new RuntimeException("Not implemented");
    }

    @Override
    public boolean add(Entry<byte[], V> vEntry) {
        throw new RuntimeException("Not implemented");
    }

    @Override
    public boolean remove(Object o) {
        throw new RuntimeException("Not implemented");
    }

    @Override
    public boolean containsAll(Collection<?> c) {
        throw new RuntimeException("Not implemented");
    }

    @Override
```

```
        public boolean addAll(Collection<? extends Entry<byte[], V>> c) {
            throw new RuntimeException("Not implemented");
        }

        @Override
        public boolean retainAll(Collection<?> c) {
            throw new RuntimeException("Not implemented");
        }

        @Override
        public boolean removeAll(Collection<?> c) {
            throw new RuntimeException("Not implemented");
        }

        @Override
        public void clear() {
            throw new RuntimeException("Not implemented");

        }
    }
}
```

51:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\main\java\org\ethereum\util\ByteArraySet.java

```
import java.util.Iterator;
import java.util.Set;

/**
 * Created by Anton Nashatyrev on 06.10.2016.
 */
public class ByteArraySet implements Set<byte[]> {
    Set<ByteArrayWrapper> delegate;

    public ByteArraySet() {
        this(new HashSet<ByteArrayWrapper>());
    }

    ByteArraySet(Set<ByteArrayWrapper> delegate) {
        this.delegate = delegate;
    }

    @Override
```

```
public int size() {
    return delegate.size();
}

@Override
public boolean isEmpty() {
    return delegate.isEmpty();
}

@Override
public boolean contains(Object o) {
    return delegate.contains(new ByteArrayWrapper((byte[]) o));
}

@Override
public Iterator<byte[]> iterator() {
    return new Iterator<byte[]>() {

        Iterator<ByteArrayWrapper> it = delegate.iterator();

        @Override
        public boolean hasNext() {
            return it.hasNext();
        }

        @Override
        public byte[] next() {
            return it.next().getData();
        }

        @Override
        public void remove() {
            it.remove();
        }
    };
}

@Override
public Object[] toArray() {
    byte[][] ret = new byte[size()][];

    ByteArrayWrapper[] arr = delegate.toArray(new ByteArrayWrapper[size()]);
    for (int i = 0; i < arr.length; i++) {
        ret[i] = arr[i].getData();
    }
    return ret;
}

@Override
public <T> T[] toArray(T[] a) {
    return (T[]) toArray();
}

@Override
public boolean add(byte[] bytes) {
    return delegate.add(new ByteArrayWrapper(bytes));
}

@Override
public boolean remove(Object o) {
    return delegate.remove(new ByteArrayWrapper((byte[]) o));
}

@Override
public boolean containsAll(Collection<?> c) {
    throw new RuntimeException("Not implemented");
}

@Override
public boolean addAll(Collection<? extends byte[]> c) {
    boolean ret = false;
    for (byte[] bytes : c) {
        ret |= add(bytes);
    }
    return ret;
}

@Override
public boolean retainAll(Collection<?> c) {
    throw new RuntimeException("Not implemented");
}

@Override
public boolean removeAll(Collection<?> c) {
        boolean changed = false;
        for (Object el : c) {
            changed |= remove(el);
        }
        return changed;
    }

    @Override
    public void clear() {
        delegate.clear();
    }

    @Override
    public boolean equals(Object o) {
        throw new RuntimeException("Not implemented");
    }

    @Override
    public int hashCode() {
        throw new RuntimeException("Not implemented");
    }
}
```

52:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\main\java\org\ethereum\util\ByteUtil.java

```
import java.io.IOException;
import java.math.BigInteger;
import java.net.InetAddress;
import java.net.UnknownHostException;
import java.nio.ByteBuffer;
import java.util.Arrays;
import java.util.HashSet;
import java.util.Set;

public class ByteUtil {

    public static final byte[] EMPTY_BYTE_ARRAY = new byte[0];
    public static final byte[] ZERO_BYTE_ARRAY = new byte[]{0};

    /**
     * Creates a copy of bytes and appends b to the end of it
     */
    public static byte[] appendByte(byte[] bytes, byte b) {
        byte[] result = Arrays.copyOf(bytes, bytes.length + 1);
        result[result.length - 1] = b;
        return result;
    }

    /**
     * The regular {@link java.math.BigInteger#toByteArray()} method isn't quite what we often
need:
     * it appends a leading zero to indicate that the number is positive and may need padding.
     *
     * @param b        the integer to format into a byte array
     * @param numBytes the desired size of the resulting byte array
     * @return numBytes byte long array.
     */
    public static byte[] bigIntegerToBytes(BigInteger b, int numBytes) {
        if (b == null) {
            return null;
        }
        byte[] bytes = new byte[numBytes];
        byte[] biBytes = b.toByteArray();
        int start = (biBytes.length == numBytes + 1) ? 1 : 0;
        int length = Math.min(biBytes.length, numBytes);
        System.arraycopy(biBytes, start, bytes, numBytes - length, length);
        return bytes;
    }

    public static byte[] bigIntegerToBytesSigned(BigInteger b, int numBytes) {
        if (b == null) {
            return null;
        }
        byte[] bytes = new byte[numBytes];
        Arrays.fill(bytes, b.signum() < 0 ? (byte) 0xFF : 0x00);
        byte[] biBytes = b.toByteArray();
        int start = (biBytes.length == numBytes + 1) ? 1 : 0;
        int length = Math.min(biBytes.length, numBytes);
        System.arraycopy(biBytes, start, bytes, numBytes - length, length);
        return bytes;
    }

    /**
     * Omitting sign indication byte.
     * <br><br>
     * Instead of {@link org.spongycastle.util.BigIntegers#asUnsignedByteArray(BigInteger)}
     * <br>we use this custom method to avoid an empty array in case of BigInteger.ZERO
     *
     * @param value - any big integer number. A <code>null</code>-value will return
<code>null</code>
     * @return A byte array without a leading zero byte if present in the signed encoding.
     * BigInteger.ZERO will return an array with length 1 and byte-value 0.
     */
    public static byte[] bigIntegerToBytes(BigInteger value) {
        if (value == null) {
            return null;
        }

        byte[] data = value.toByteArray();

        if (data.length != 1 && data[0] == 0) {
            byte[] tmp = new byte[data.length - 1];
            System.arraycopy(data, 1, tmp, 0, tmp.length);
            data = tmp;
        }
        return data;
    }

    /**
     * Cast hex encoded value from byte[] to BigInteger
     * null is parsed like byte[0]
     *
     * @param bb byte array contains the values
     * @return unsigned positive BigInteger value.
     */
    public static BigInteger bytesToBigInteger(byte[] bb) {
        return (bb == null || bb.length == 0) ? BigInteger.ZERO : new BigInteger(1, bb);
    }

    /**
     * Returns the amount of nibbles that match each other from 0 ...
     * amount will never be larger than smallest input
     *
     * @param a - first input
     * @param b - second input
     * @return Number of bytes that match
```

```
 */
public static int matchingNibbleLength(byte[] a, byte[] b) {
    int i = 0;
    int length = a.length < b.length ? a.length : b.length;
    while (i < length) {
        if (a[i] != b[i]) {
            return i;
        }
        i++;
    }
    return i;
}

/**
 * Converts a long value into a byte array.
 *
 * @param val - long value to convert
 * @return <code>byte[]</code> of length 8, representing the long value
 */
public static byte[] longToBytes(long val) {
    return ByteBuffer.allocate(Long.BYTES).putLong(val).array();
}

/**
 * Converts a long value into a byte array.
 *
 * @param val - long value to convert
 * @return decimal value with leading byte that are zeroes striped
 */
public static byte[] longToBytesNoLeadZeroes(long val) {

    // todo: improve performance by while strip numbers until (long >> 8 == 0)
    if (val == 0) {
        return EMPTY_BYTE_ARRAY;
    }

    byte[] data = ByteBuffer.allocate(Long.BYTES).putLong(val).array();

    return stripLeadingZeroes(data);
}

/**
```

```
 * Converts int value into a byte array.
 *
 * @param val - int value to convert
 * @return <code>byte[]</code> of length 4, representing the int value
 */
public static byte[] intToBytes(int val) {
    return ByteBuffer.allocate(Integer.BYTES).putInt(val).array();
}

/**
 * Converts a int value into a byte array.
 *
 * @param val - int value to convert
 * @return value with leading byte that are zeroes striped
 */
public static byte[] intToBytesNoLeadZeroes(int val) {

    if (val == 0) {
        return EMPTY_BYTE_ARRAY;
    }

    int lenght = 0;

    int tmpVal = val;
    while (tmpVal != 0) {
        tmpVal = tmpVal >>> 8;
        ++lenght;
    }

    byte[] result = new byte[lenght];

    int index = result.length - 1;
    while (val != 0) {

        result[index] = (byte) (val & 0xFF);
        val = val >>> 8;
        index -= 1;
    }

    return result;
}
```

```
/**
 * Convert a byte-array into a hex String.<br>
 * Works similar to {@link Hex#toHexString}
 * but allows for <code>null</code>
 *
 * @param data - byte-array to convert to a hex-string
 * @return hex representation of the data.<br>
 * Returns an empty String if the input is <code>null</code>
 * @see Hex#toHexString
 */
public static String toHexString(byte[] data) {
    return data == null ? "" : Hex.toHexString(data);
}

/**
 * Calculate packet length
 *
 * @param msg byte[]
 * @return byte-array with 4 elements
 */
public static byte[] calcPacketLength(byte[] msg) {
    int msgLen = msg.length;
    return new byte[]{
            (byte) ((msgLen >> 24) & 0xFF),
            (byte) ((msgLen >> 16) & 0xFF),
            (byte) ((msgLen >> 8) & 0xFF),
            (byte) ((msgLen) & 0xFF)};
}

/**
 * Cast hex encoded value from byte[] to int
 * null is parsed like byte[0]
 * <p>
 * Limited to Integer.MAX_VALUE: 2^32-1 (4 bytes)
 *
 * @param b array contains the values
 * @return unsigned positive int value.
 */
public static int byteArrayToInt(byte[] b) {
    if (b == null || b.length == 0) {
        return 0;
    }
    return new BigInteger(1, b).intValue();
}

/**
 * Cast hex encoded value from byte[] to long
 * null is parsed like byte[0]
 * <p>
 * Limited to Long.MAX_VALUE: 2<sup>63</sup>-1 (8 bytes)
 *
 * @param b array contains the values
 * @return unsigned positive long value.
 */
public static long byteArrayToLong(byte[] b) {
    if (b == null || b.length == 0) {
        return 0;
    }
    return new BigInteger(1, b).longValue();
}


/**
 * Turn nibbles to a pretty looking output string
 * <p>
 * Example. [ 1, 2, 3, 4, 5 ] becomes '\x11\x23\x45'
 *
 * @param nibbles - getting byte of data [ 04 ] and turning
 *                it to a '\x04' representation
 * @return pretty string of nibbles
 */
public static String nibblesToPrettyString(byte[] nibbles) {
    StringBuilder builder = new StringBuilder();
    for (byte nibble : nibbles) {
        final String nibbleString = oneByteToHexString(nibble);
        builder.append("\\x").append(nibbleString);
    }
    return builder.toString();
}

public static String oneByteToHexString(byte value) {
    String retVal = Integer.toString(value & 0xFF, 16);
    if (retVal.length() == 1) {
        retVal = "0" + retVal;
    }
    return retVal;
}

/**
 * Calculate the number of bytes need
 * to encode the number
 *
 * @param val - number
 * @return number of min bytes used to encode the number
 */
public static int numBytes(String val) {

    BigInteger bInt = new BigInteger(val);
    int bytes = 0;

    while (!bInt.equals(BigInteger.ZERO)) {
        bInt = bInt.shiftRight(8);
        ++bytes;
    }
    if (bytes == 0) {
        ++bytes;
    }
    return bytes;
}

/**
 * @param arg - not more that 32 bits
 * @return - bytes of the value pad with complete to 32 zeroes
 */
public static byte[] encodeValFor32Bits(Object arg) {

    byte[] data;

    // check if the string is numeric
    if (arg.toString().trim().matches("-?\\d+(\\.\\d+)?")) {
        data = new BigInteger(arg.toString().trim()).toByteArray();
    }
    // check if it's hex number
    else if (arg.toString().trim().matches("0[xX][0-9a-fA-F]+")) {
        data = new BigInteger(arg.toString().trim().substring(2), 16).toByteArray();
    } else {
        data = arg.toString().trim().getBytes();
    }



    if (data.length > 32) {
        throw new RuntimeException("values can't be more than 32 byte");
    }

    byte[] val = new byte[32];

    int j = 0;
    for (int i = data.length; i > 0; --i) {
        val[31 - j] = data[i - 1];
        ++j;
    }
    return val;
}

/**
 * encode the values and concatenate together
 *
 * @param args Object
 * @return byte[]
 */
public static byte[] encodeDataList(Object... args) {
    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    for (Object arg : args) {
        byte[] val = encodeValFor32Bits(arg);
        try {
            baos.write(val);
        } catch (IOException e) {
            throw new Error("Happen something that should never happen ", e);
        }
    }
    return baos.toByteArray();
}

public static int firstNonZeroByte(byte[] data) {
    for (int i = 0; i < data.length; ++i) {
        if (data[i] != 0) {
            return i;
        }
    }
    return -1;
}

public static byte[] stripLeadingZeroes(byte[] data) {

    if (data == null) {
        return null;
    }

    final int firstNonZero = firstNonZeroByte(data);
    switch (firstNonZero) {
        case -1:
            return ZERO_BYTE_ARRAY;

        case 0:
            return data;

        default:
            byte[] result = new byte[data.length - firstNonZero];
            System.arraycopy(data, firstNonZero, result, 0, data.length - firstNonZero);

            return result;
    }
}

/**
 * increment byte array as a number until max is reached
 *
 * @param bytes byte[]
 * @return boolean
 */
public static boolean increment(byte[] bytes) {
    final int startIndex = 0;
    int i;
    for (i = bytes.length - 1; i >= startIndex; i--) {
        bytes[i]++;
        if (bytes[i] != 0) {
            break;
        }
    }
    // we return false when all bytes are 0 again
    return (i >= startIndex || bytes[startIndex] != 0);
}

/**
 * Utility function to copy a byte array into a new byte array with given size.
 * If the src length is smaller than the given size, the result will be left-padded
 * with zeros.
 *
 * @param value - a BigInteger with a maximum value of 2^256-1
 * @return Byte array of given size with a copy of the <code>src</code>
 */
public static byte[] copyToArray(BigInteger value) {
    byte[] src = ByteUtil.bigIntegerToBytes(value);
    byte[] dest = ByteBuffer.allocate(32).array();
    System.arraycopy(src, 0, dest, dest.length - src.length, src.length);
    return dest;
}


public static ByteArrayWrapper wrap(byte[] data) {
    return new ByteArrayWrapper(data);
}

public static byte[] setBit(byte[] data, int pos, int val) {

    if ((data.length * 8) - 1 < pos) {
        throw new Error("outside byte array limit, pos: " + pos);
    }

    int posByte = data.length - 1 - (pos) / 8;
    int posBit = (pos) % 8;
    byte setter = (byte) (1 << (posBit));
    byte toBeSet = data[posByte];
    byte result;
    if (val == 1) {
        result = (byte) (toBeSet | setter);
    } else {
        result = (byte) (toBeSet & ~setter);
    }

    data[posByte] = result;
    return data;
}

public static int getBit(byte[] data, int pos) {

    if ((data.length * 8) - 1 < pos) {
        throw new Error("outside byte array limit, pos: " + pos);
    }

    int posByte = data.length - 1 - pos / 8;
    int posBit = pos % 8;
    byte dataByte = data[posByte];
    return Math.min(1, (dataByte & (1 << (posBit))));
}

public static byte[] and(byte[] b1, byte[] b2) {
    if (b1.length != b2.length) {
        throw new RuntimeException("Array sizes differ");
    }
    byte[] ret = new byte[b1.length];
    for (int i = 0; i < ret.length; i++) {
        ret[i] = (byte) (b1[i] & b2[i]);
    }
    return ret;
}

public static byte[] or(byte[] b1, byte[] b2) {
    if (b1.length != b2.length) {
        throw new RuntimeException("Array sizes differ");
    }
    byte[] ret = new byte[b1.length];
    for (int i = 0; i < ret.length; i++) {
        ret[i] = (byte) (b1[i] | b2[i]);
    }
    return ret;
}

public static byte[] xor(byte[] b1, byte[] b2) {
    if (b1.length != b2.length) {
        throw new RuntimeException("Array sizes differ");
    }
    byte[] ret = new byte[b1.length];
```

```
        for (int i = 0; i < ret.length; i++) {
            ret[i] = (byte) (b1[i] ^ b2[i]);
        }
        return ret;
    }

    /**
     * XORs byte arrays of different lengths by aligning length of the shortest via adding zeros at
beginning
     */
    public static byte[] xorAlignRight(byte[] b1, byte[] b2) {
        if (b1.length > b2.length) {
            byte[] b2_ = new byte[b1.length];
            System.arraycopy(b2, 0, b2_, b1.length - b2.length, b2.length);
            b2 = b2_;
        } else if (b2.length > b1.length) {
            byte[] b1_ = new byte[b2.length];
            System.arraycopy(b1, 0, b1_, b2.length - b1.length, b1.length);
            b1 = b1_;
        }

        return xor(b1, b2);
    }

    /**
     * @param arrays - arrays to merge
     * @return - merged array
     */
    public static byte[] merge(byte[]... arrays) {
        int count = 0;
        for (byte[] array : arrays) {
            count += array.length;
        }

        // Create new array and copy all array contents
        byte[] mergedArray = new byte[count];
        int start = 0;
        for (byte[] array : arrays) {
            System.arraycopy(array, 0, mergedArray, start, array.length);
            start += array.length;
        }
        return mergedArray;
```

```
}

public static boolean isNullOrZeroArray(byte[] array) {
    return (array == null) || (array.length == 0);
}

public static boolean isSingleZero(byte[] array) {
    return (array.length == 1 && array[0] == 0);
}



public static Set<byte[]> difference(Set<byte[]> setA, Set<byte[]> setB) {

    Set<byte[]> result = new HashSet<>();

    for (byte[] elementA : setA) {
        boolean found = false;
        for (byte[] elementB : setB) {

            if (Arrays.equals(elementA, elementB)) {
                found = true;
                break;
            }
        }
        if (!found) {
            result.add(elementA);
        }
    }

    return result;
}

public static int length(byte[]... bytes) {
    int result = 0;
    for (byte[] array : bytes) {
        result += (array == null) ? 0 : array.length;
    }
    return result;
}

public static byte[] intsToBytes(int[] arr, boolean bigEndian) {
    byte[] ret = new byte[arr.length * 4];
```

```
        intsToBytes(arr, ret, bigEndian);
        return ret;
    }

    public static int[] bytesToInts(byte[] arr, boolean bigEndian) {
        int[] ret = new int[arr.length / 4];
        bytesToInts(arr, ret, bigEndian);
        return ret;
    }

    public static void bytesToInts(byte[] b, int[] arr, boolean bigEndian) {
        if (!bigEndian) {
            int off = 0;
            for (int i = 0; i < arr.length; i++) {
                int ii = b[off++] & 0x000000FF;
                ii |= (b[off++] << 8) & 0x0000FF00;
                ii |= (b[off++] << 16) & 0x00FF0000;
                ii |= (b[off++] << 24);
                arr[i] = ii;
            }
        } else {
            int off = 0;
            for (int i = 0; i < arr.length; i++) {
                int ii = b[off++] << 24;
                ii |= (b[off++] << 16) & 0x00FF0000;
                ii |= (b[off++] << 8) & 0x0000FF00;
                ii |= b[off++] & 0x000000FF;
                arr[i] = ii;
            }
        }
    }

    public static void intsToBytes(int[] arr, byte[] b, boolean bigEndian) {
        if (!bigEndian) {
            int off = 0;
            for (int i = 0; i < arr.length; i++) {
                int ii = arr[i];
                b[off++] = (byte) (ii & 0xFF);
                b[off++] = (byte) ((ii >> 8) & 0xFF);
                b[off++] = (byte) ((ii >> 16) & 0xFF);
                b[off++] = (byte) ((ii >> 24) & 0xFF);
            }
        } else {
            int off = 0;
            for (int i = 0; i < arr.length; i++) {
                int ii = arr[i];
                b[off++] = (byte) ((ii >> 24) & 0xFF);
                b[off++] = (byte) ((ii >> 16) & 0xFF);
                b[off++] = (byte) ((ii >> 8) & 0xFF);
                b[off++] = (byte) (ii & 0xFF);
            }
        }
    }

    public static short bigEndianToShort(byte[] bs) {
        return bigEndianToShort(bs, 0);
    }

    public static short bigEndianToShort(byte[] bs, int off) {
        int n = bs[off] << 8;
        ++off;
        n |= bs[off] & 0xFF;
        return (short) n;
    }

    public static byte[] shortToBytes(short n) {
        return ByteBuffer.allocate(2).putShort(n).array();
    }

    /**
     * Converts string hex representation to data bytes
     * Accepts following hex:
     * - with or without 0x prefix
     * - with no leading 0, like 0xabc -> 0x0abc
     *
     * @param data String like '0xa5e..' or just 'a5e..'
     * @return decoded bytes array
     */
    public static byte[] hexStringToBytes(String data) {
        if (data == null) {
            return EMPTY_BYTE_ARRAY;
        }
        if (data.startsWith("0x")) {
            data = data.substring(2);
        }
        if (data.length() % 2 == 1) {
            data = "0" + data;
        }
        return Hex.decode(data);
    }

    /**
     * Converts string representation of host/ip to 4-bytes byte[] IPv4
     */
    public static byte[] hostToBytes(String ip) {
        byte[] bytesIp;
        try {
            bytesIp = InetAddress.getByName(ip).getAddress();
        } catch (UnknownHostException e) {
            bytesIp = new byte[4];  // fall back to invalid 0.0.0.0 address
        }

        return bytesIp;
    }

    /**
     * Converts 4 bytes IPv4 IP to String representation
     */
    public static String bytesToIp(byte[] bytesIp) {

        StringBuilder sb = new StringBuilder();
        sb.append(bytesIp[0] & 0xFF);
        sb.append(".");
        sb.append(bytesIp[1] & 0xFF);
        sb.append(".");
        sb.append(bytesIp[2] & 0xFF);
        sb.append(".");
        sb.append(bytesIp[3] & 0xFF);

        String ip = sb.toString();
        return ip;
    }

    /**
     * Returns a number of zero bits preceding the highest-order ("leftmost") one-bit
     * interpreting input array as a big-endian integer value
```

```
     */
    public static int numberOfLeadingZeros(byte[] bytes) {

        int i = firstNonZeroByte(bytes);

        if (i == -1) {
            return bytes.length * 8;
        } else {
            int byteLeadingZeros = Integer.numberOfLeadingZeros((int) bytes[i] & 0xff) - 24;
            return i * 8 + byteLeadingZeros;
        }
    }

    /**
     * Parses fixed number of bytes starting from {@code offset} in {@code input} array.
     * If {@code input} has not enough bytes return array will be right padded with zero bytes.
     * I.e. if {@code offset} is higher than {@code input.length} then zero byte array of length
{@code len} will be returned
     */
    public static byte[] parseBytes(byte[] input, int offset, int len) {

        if (offset >= input.length || len == 0) {
            return EMPTY_BYTE_ARRAY;
        }

        byte[] bytes = new byte[len];
        System.arraycopy(input, offset, bytes, 0, Math.min(input.length - offset, len));
        return bytes;
    }

    /**
     * Parses 32-bytes word from given input.
     * Uses {@link #parseBytes(byte[], int, int)} method,
     * thus, result will be right-padded with zero bytes if there is not enough bytes in {@code input}
     *
     * @param idx an index of the word starting from {@code 0}
     */
    public static byte[] parseWord(byte[] input, int idx) {
        return parseBytes(input, 32 * idx, 32);
    }

    /**
```

```
     * Parses 32-bytes word from given input.
     * Uses {@link #parseBytes(byte[], int, int)} method,
     * thus, result will be right-padded with zero bytes if there is not enough bytes in {@code input}
     *
     * @param idx    an index of the word starting from {@code 0}
     * @param offset an offset in {@code input} array to start parsing from
     */
    public static byte[] parseWord(byte[] input, int offset, int idx) {
        return parseBytes(input, offset + 32 * idx, 32);
    }
}
```

53:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\main\java\org\ethereum\util\CollectionUtils.java

```
 * @author Mikhail Kalinin
 * @since 14.07.2015
 */
public class CollectionUtils {
    public static <T> List<T> truncate(final List<T> items, int limit) {
        if (limit > items.size()) {
            return new ArrayList<>(items);
        }
        List<T> truncated = new ArrayList<>(limit);
        for (T item : items) {
            truncated.add(item);
            if (truncated.size() == limit) {
                break;
            }
        }
        return truncated;
    }

    public static <T> List<T> truncateRand(final List<T> items, int limit) {
        if (limit > items.size()) {
            return new ArrayList<>(items);
        }
        List<T> truncated = new ArrayList<>(limit);

        LinkedList<Integer> index = new LinkedList<>();
        for (int i = 0; i < items.size(); ++i) {
            index.add(i);
        }
        if (limit * 2 < items.size()) {
            // Limit is very small comparing to items.size()
            Set<Integer> smallIndex = new HashSet<>();
            for (int i = 0; i < limit; ++i) {
                int randomNum = ThreadLocalRandom.current().nextInt(0, index.size());
                smallIndex.add(index.remove(randomNum));
            }
            smallIndex.forEach(i -> truncated.add(items.get(i)));
        } else {
            // Limit is compared to items.size()
            for (int i = 0; i < items.size() - limit; ++i) {
                int randomNum = ThreadLocalRandom.current().nextInt(0, index.size());
                index.remove(randomNum);
            }
            index.forEach(i -> truncated.add(items.get(i)));
        }

        return truncated;
    }
}
```

54:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\main\java\org\ethereum\util\CompactEncoder.java

```
import static java.util.Arrays.copyOf;
import static java.util.Arrays.copyOfRange;
import static org.ethereum.util.ByteUtil.appendByte;
import static org.spongycastle.util.Arrays.concatenate;
import static org.spongycastle.util.encoders.Hex.encode;

/**
 * Compact encoding of hex sequence with optional terminator
 * <p>
 * The traditional compact way of encoding a hex string is to convert it into binary
 * - that is, a string like 0f1248 would become three bytes 15, 18, 72. However,
 * this approach has one slight problem: what if the length of the hex string is odd?
 * In that case, there is no way to distinguish between, say, 0f1248 and f1248.
 * <p>
 * Additionally, our application in the Merkle Patricia tree requires the additional feature
 * that a hex string can also have a special "terminator symbol" at the end (denoted by the 'T').
 * A terminator symbol can occur only once, and only at the end.
 * <p>
 * An alternative way of thinking about this to not think of there being a terminator symbol,
 * but instead treat bit specifying the existence of the terminator symbol as a bit specifying
 * that the given node encodes a final node, where the value is an actual value, rather than
 * the hash of yet another node.
 * <p>
 * To solve both of these issues, we force the first nibble of the final byte-stream to encode
 * two flags, specifying oddness of length (ignoring the 'T' symbol) and terminator status;
 * these are placed, respectively, into the two lowest significant bits of the first nibble.
 * In the case of an even-length hex string, we must introduce a second nibble (of value zero)
 * to ensure the hex-string is even in length and thus is representable by a whole number of bytes.
 * <p>
 * Examples:
 * &gt; [ 1, 2, 3, 4, 5 ]
 * '\x11\x23\x45'
 * &gt; [ 0, 1, 2, 3, 4, 5 ]
 * '\x00\x01\x23\x45'
 * &gt; [ 0, 15, 1, 12, 11, 8, T ]
 * '\x20\x0f\x1c\xb8'
 * &gt; [ 15, 1, 12, 11, 8, T ]
 * '\x3f\x1c\xb8'
 */
public class CompactEncoder {

    private final static byte TERMINATOR = 16;
    private final static Map<Character, Byte> hexMap = new HashMap<>();

    static {
        hexMap.put('0', (byte) 0x0);
        hexMap.put('1', (byte) 0x1);
        hexMap.put('2', (byte) 0x2);
        hexMap.put('3', (byte) 0x3);
        hexMap.put('4', (byte) 0x4);
        hexMap.put('5', (byte) 0x5);
        hexMap.put('6', (byte) 0x6);
        hexMap.put('7', (byte) 0x7);
        hexMap.put('8', (byte) 0x8);
        hexMap.put('9', (byte) 0x9);
        hexMap.put('a', (byte) 0xa);
        hexMap.put('b', (byte) 0xb);
        hexMap.put('c', (byte) 0xc);
        hexMap.put('d', (byte) 0xd);
        hexMap.put('e', (byte) 0xe);
        hexMap.put('f', (byte) 0xf);
    }

    /**
     * Pack nibbles to binary
     *
     * @param nibbles sequence. may have a terminator
     * @return hex-encoded byte array
     */
    public static byte[] packNibbles(byte[] nibbles) {
        int terminator = 0;

        if (nibbles[nibbles.length - 1] == TERMINATOR) {
            terminator = 1;
            nibbles = copyOf(nibbles, nibbles.length - 1);
        }
        int oddlen = nibbles.length % 2;
        int flag = 2 * terminator + oddlen;
        if (oddlen != 0) {
            byte[] flags = new byte[]{(byte) flag};
            nibbles = concatenate(flags, nibbles);
        } else {
            byte[] flags = new byte[]{(byte) flag, 0};
            nibbles = concatenate(flags, nibbles);
        }
        ByteArrayOutputStream buffer = new ByteArrayOutputStream();
        for (int i = 0; i < nibbles.length; i += 2) {
            buffer.write(16 * nibbles[i] + nibbles[i + 1]);
        }
        return buffer.toByteArray();
    }

    public static boolean hasTerminator(byte[] packedKey) {
        return ((packedKey[0] >> 4) & 2) != 0;
    }

    /**
     * Unpack a binary string to its nibbles equivalent
     *
     * @param str of binary data
     * @return array of nibbles in byte-format
     */
    public static byte[] unpackToNibbles(byte[] str) {
        byte[] base = binToNibbles(str);
        base = copyOf(base, base.length - 1);
        if (base[0] >= 2) {
            base = appendByte(base, TERMINATOR);
        }
        if (base[0] % 2 == 1) {
            base = copyOfRange(base, 1, base.length);
        } else {
            base = copyOfRange(base, 2, base.length);
        }
        return base;
    }

    /**
     * Transforms a binary array to hexadecimal format + terminator
     *
     * @param str byte[]
     * @return array with each individual nibble adding a terminator at the end
     */
    public static byte[] binToNibbles(byte[] str) {

        byte[] hexEncoded = encode(str);
        byte[] hexEncodedTerminated = Arrays.copyOf(hexEncoded, hexEncoded.length + 1);

        for (int i = 0; i < hexEncoded.length; ++i) {
            byte b = hexEncodedTerminated[i];
            hexEncodedTerminated[i] = hexMap.get((char) b);
        }

        hexEncodedTerminated[hexEncodedTerminated.length - 1] = TERMINATOR;
        return hexEncodedTerminated;
    }


    public static byte[] binToNibblesNoTerminator(byte[] str) {

        byte[] hexEncoded = encode(str);

        for (int i = 0; i < hexEncoded.length; ++i) {
            byte b = hexEncoded[i];
```

```
            hexEncoded[i] = hexMap.get((char) b);
        }

        return hexEncoded;
    }
}
```

55:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\main\java\org\ethereum\util\CopyOnWriteMap.java

```
 *  "License"); you may not use this file except in compliance
 *  with the License.  You may obtain a copy of the License at
 *
 *    http://www.apache.org/licenses/LICENSE-2.0
 *
 *  Unless required by applicable law or agreed to in writing,
 *  software distributed under the License is distributed on an
 *  "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY
 *  KIND, either express or implied.  See the License for the
 *  specific language governing permissions and limitations
 *  under the License.
 *
 */
package org.ethereum.util;

import java.util.Collection;
import java.util.HashMap;
import java.util.Map;
import java.util.Set;

/**
 * A thread-safe version of {@link Map} in which all operations that change the
 * Map are implemented by making a new copy of the underlying Map.
 * <p>
 * While the creation of a new Map can be expensive, this class is designed for
 * cases in which the primary function is to read data from the Map, not to
 * modify the Map.  Therefore the operations that do not cause a change to this
 * class happen quickly and concurrently.
 *
 * @author The Apache MINA Project (dev@mina.apache.org)
 * @version $Rev$, $Date$
 */
public class CopyOnWriteMap<K, V> implements Map<K, V>, Cloneable {
```

```
private volatile Map<K, V> internalMap;

/**
 * Creates a new instance of CopyOnWriteMap.
 */
public CopyOnWriteMap() {
    internalMap = new HashMap<K, V>();
}

/**
 * Creates a new instance of CopyOnWriteMap with the specified initial size
 *
 * @param initialCapacity The initial size of the Map.
 */
public CopyOnWriteMap(int initialCapacity) {
    internalMap = new HashMap<K, V>(initialCapacity);
}

/**
 * Creates a new instance of CopyOnWriteMap in which the
 * initial data being held by this map is contained in
 * the supplied map.
 *
 * @param data A Map containing the initial contents to be placed into
 *             this class.
 */
public CopyOnWriteMap(Map<K, V> data) {
    internalMap = new HashMap<K, V>(data);
}

/**
 * Adds the provided key and value to this map.
 *
 * @see java.util.Map#put(java.lang.Object, java.lang.Object)
 */
@Override
public V put(K key, V value) {
    synchronized (this) {
        Map<K, V> newMap = new HashMap<K, V>(internalMap);
        V val = newMap.put(key, value);
        internalMap = newMap;
        return val;
    }
}

/**
 * Removed the value and key from this map based on the
 * provided key.
 *
 * @see java.util.Map#remove(java.lang.Object)
 */
@Override
public V remove(Object key) {
    synchronized (this) {
        Map<K, V> newMap = new HashMap<K, V>(internalMap);
        V val = newMap.remove(key);
        internalMap = newMap;
        return val;
    }
}

/**
 * Inserts all the keys and values contained in the
 * provided map to this map.
 *
 * @see java.util.Map#putAll(java.util.Map)
 */
@Override
public void putAll(Map<? extends K, ? extends V> newData) {
    synchronized (this) {
        Map<K, V> newMap = new HashMap<K, V>(internalMap);
        newMap.putAll(newData);
        internalMap = newMap;
    }
}

/**
 * Removes all entries in this map.
 *
 * @see java.util.Map#clear()
 */
@Override
public void clear() {
    synchronized (this) {
        internalMap = new HashMap<K, V>();
    }
}

//
//  Below are methods that do not modify
//          the internal Maps

/**
 * Returns the number of key/value pairs in this map.
 *
 * @see java.util.Map#size()
 */
@Override
public int size() {
    return internalMap.size();
}

/**
 * Returns true if this map is empty, otherwise false.
 *
 * @see java.util.Map#isEmpty()
 */
@Override
public boolean isEmpty() {
    return internalMap.isEmpty();
}

/**
 * Returns true if this map contains the provided key, otherwise
 * this method return false.
 *
 * @see java.util.Map#containsKey(java.lang.Object)
 */
@Override
public boolean containsKey(Object key) {
    return internalMap.containsKey(key);
}

/**
 * Returns true if this map contains the provided value, otherwise
 * this method returns false.
 *
 * @see java.util.Map#containsValue(java.lang.Object)
 */
@Override
public boolean containsValue(Object value) {
    return internalMap.containsValue(value);
}

/**
 * Returns the value associated with the provided key from this
 * map.
 *
 * @see java.util.Map#get(java.lang.Object)
 */
@Override
public V get(Object key) {
    return internalMap.get(key);
}

/**
 * This method will return a read-only {@link Set}.
 */
@Override
public Set<K> keySet() {
    return internalMap.keySet();
}

/**
 * This method will return a read-only {@link Collection}.
 */
@Override
public Collection<V> values() {
    return internalMap.values();
}

/**
 * This method will return a read-only {@link Set}.
 */
@Override
public Set<Entry<K, V>> entrySet() {
    return internalMap.entrySet();
}
```

```
    @Override
    public Object clone() {
        try {
            return super.clone();
        } catch (CloneNotSupportedException e) {
            throw new InternalError();
        }
    }
}
```

56:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\main\java\org\ethereum\util\DecodeResult.java

```
@SuppressWarnings("serial")
public class DecodeResult implements Serializable {

    private int pos;
    private Object decoded;

    public DecodeResult(int pos, Object decoded) {
        this.pos = pos;
        this.decoded = decoded;
    }

    public int getPos() {
        return pos;
    }

    public Object getDecoded() {
        return decoded;
    }

    @Override
    public String toString() {
        return asString(this.decoded);
    }

    private String asString(Object decoded) {
        if (decoded instanceof String) {
            return (String) decoded;
        } else if (decoded instanceof byte[]) {
            return Hex.toHexString((byte[]) decoded);
        } else if (decoded instanceof Object[]) {
            String result = "";
            for (Object item : (Object[]) decoded) {
                result += asString(item);
            }
            return result;
        }
        throw new RuntimeException("Not a valid type. Should not occur");
    }
}
```

57:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\main\java\org\ethereum\util\ExecutorPipeline.java

```
import java.util.concurrent.LinkedBlockingQueue;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicInteger;
import java.util.concurrent.atomic.AtomicLong;
import java.util.concurrent.locks.ReentrantLock;
import java.util.function.Consumer;
import java.util.function.Function;

/**
 * Queues execution tasks into a single pipeline where some tasks can be executed in parallel
 * but preserve 'messages' order so the next task process messages on a single thread in
 * the same order they were added to the previous executor
 * <p>
 * Created by Anton Nashatyrev on 23.02.2016.
 */
public class ExecutorPipeline<In, Out> {

    private BlockingQueue<Runnable> queue;
    private ThreadPoolExecutor exec;
    private boolean preserveOrder = false;
    private Function<In, Out> processor;
    private Consumer<Throwable> exceptionHandler;
    private ExecutorPipeline<Out, ?> next;

    private AtomicLong orderCounter = new AtomicLong();
    private long nextOutTaskNumber = 0;
    private Map<Long, Out> orderMap = new HashMap<>();
    private ReentrantLock lock = new ReentrantLock();
    private String threadPoolName;

    private static AtomicInteger pipeNumber = new AtomicInteger(1);
    private AtomicInteger threadNumber = new AtomicInteger(1);

    public ExecutorPipeline(int threads, int queueSize, boolean preserveOrder, Function<In, Out>
processor,
                            Consumer<Throwable> exceptionHandler) {
        queue = new LimitedQueue<>(queueSize);
        exec = new ThreadPoolExecutor(threads, threads, 0L, TimeUnit.MILLISECONDS, queue, r -
>
                new Thread(r, threadPoolName + "-" + threadNumber.getAndIncrement())
        );
        this.preserveOrder = preserveOrder;
        this.processor = processor;
        this.exceptionHandler = exceptionHandler;
        this.threadPoolName = "pipe-" + pipeNumber.getAndIncrement();
    }

    public ExecutorPipeline<Out, Void> add(int threads, int queueSize, final Consumer<Out>
consumer) {
        return add(threads, queueSize, false, out -> {
            consumer.accept(out);
            return null;
        });
    }

    public <NextOut> ExecutorPipeline<Out, NextOut> add(int threads, int queueSize, boolean
preserveOrder,
                                                        Function<Out, NextOut> processor) {
        ExecutorPipeline<Out, NextOut> ret = new ExecutorPipeline<>(threads, queueSize,
preserveOrder, processor, exceptionHandler);
        next = ret;
        return ret;
    }

    private void pushNext(long order, Out res) {
        if (next != null) {
            if (!preserveOrder) {
                next.push(res);
            } else {
                lock.lock();
                try {
                    if (order == nextOutTaskNumber) {
                        next.push(res);
                        while (true) {
                            nextOutTaskNumber++;
                            Out out = orderMap.remove(nextOutTaskNumber);
                            if (out == null) {
                                break;
                            }
                            next.push(out);
                        }
                    } else {
                        orderMap.put(order, res);
                    }
                } finally {
                    lock.unlock();
                }
            }
        }
    }

    public void push(final In in) {
        final long order = orderCounter.getAndIncrement();
        exec.execute(() -> {
            try {
                pushNext(order, processor.apply(in));
            } catch (Throwable e) {
                exceptionHandler.accept(e);
            }
        });
    }

    public void pushAll(final List<In> list) {
        for (In in : list) {
            push(in);
        }
    }

    public ExecutorPipeline<In, Out> setThreadPoolName(String threadPoolName) {
        this.threadPoolName = threadPoolName;
        return this;
    }
```

```
public BlockingQueue<Runnable> getQueue() {
    return queue;
}

public Map<Long, Out> getOrderMap() {
    return orderMap;
}

public void shutdown() {
    try {
        exec.shutdown();
    } catch (Exception e) {
    }
    if (next != null) {
        exec.shutdown();
    }
}

public boolean isShutdown() {
    return exec.isShutdown();
}

/**
 * Shutdowns executors and waits until all pipeline
 * submitted tasks complete
 *
 * @throws InterruptedException
 */
public void join() throws InterruptedException {
    exec.shutdown();
    exec.awaitTermination(10, TimeUnit.MINUTES);
    if (next != null) {
        next.join();
    }
}

private static class LimitedQueue<E> extends LinkedBlockingQueue<E> {
    public LimitedQueue(int maxSize) {
        super(maxSize);
    }
```

```
        @Override
        public boolean offer(E e) {
            // turn offer() and add() into a blocking calls (unless interrupted)
            try {
                put(e);
                return true;
            } catch (InterruptedException ie) {
                Thread.currentThread().interrupt();
            }
            return false;
        }
    }
}
```

58:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\main\java\org\ethereum\util\FastByteComparisons.java

```
 * "License"); you may not use this file except in compliance
 * with the License.  You may obtain a copy of the License at
 * <p>
 * http://www.apache.org/licenses/LICENSE-2.0
 * <p>
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */
package org.ethereum.util;

import com.google.common.primitives.UnsignedBytes;



/**
 * Utility code to do optimized byte-array comparison.
 * This is borrowed and slightly modified from Guava's {@link UnsignedBytes}
 * class to be able to compare arrays that start at non-zero offsets.
 */
@SuppressWarnings("restriction")
public abstract class FastByteComparisons {

    public static boolean equal(byte[] b1, byte[] b2) {
        return b1.length == b2.length && compareTo(b1, 0, b1.length, b2, 0, b2.length) == 0;
```

```
}

/**
 * Lexicographically compare two byte arrays.
 *
 * @param b1 buffer1
 * @param s1 offset1
 * @param l1 length1
 * @param b2 buffer2
 * @param s2 offset2
 * @param l2 length2
 * @return int
 */
public static int compareTo(byte[] b1, int s1, int l1, byte[] b2, int s2, int l2) {
    return LexicographicalComparerHolder.BEST_COMPARER.compareTo(
            b1, s1, l1, b2, s2, l2);
}

private interface Comparer<T> {
    int compareTo(T buffer1, int offset1, int length1,
                  T buffer2, int offset2, int length2);
}

private static Comparer<byte[]> lexicographicalComparerJavaImpl() {
    return LexicographicalComparerHolder.PureJavaComparer.INSTANCE;
}


/**
 * <p>Uses reflection to gracefully fall back to the Java implementation if
 * {@code Unsafe} isn't available.
 */
private static class LexicographicalComparerHolder {
    static final String UNSAFE_COMPARER_NAME =
            LexicographicalComparerHolder.class.getName() + "$UnsafeComparer";

    static final Comparer<byte[]> BEST_COMPARER = getBestComparer();

    /**
     * Returns the Unsafe-using Comparer, or falls back to the pure-Java
     * implementation if unable to do so.
     */
    static Comparer<byte[]> getBestComparer() {
      try {
            Class<?> theClass = Class.forName(UNSAFE_COMPARER_NAME);

            // yes, UnsafeComparer does implement Comparer<byte[]>
            @SuppressWarnings("unchecked")
            Comparer<byte[]> comparer =
                (Comparer<byte[]>) theClass.getEnumConstants()[0];
            return comparer;
      } catch (Throwable t) { // ensure we really catch *everything*
            return lexicographicalComparerJavaImpl();
      }
    }

    private enum PureJavaComparer implements Comparer<byte[]> {
      INSTANCE;

      @Override
      public int compareTo(byte[] buffer1, int offset1, int length1,
                 byte[] buffer2, int offset2, int length2) {
            // Short circuit equal case
            if (buffer1 == buffer2 &&
                offset1 == offset2 &&
                length1 == length2) {
              return 0;
            }
            int end1 = offset1 + length1;
            int end2 = offset2 + length2;
            for (int i = offset1, j = offset2; i < end1 && j < end2; i++, j++) {
              int a = (buffer1[i] & 0xff);
              int b = (buffer2[j] & 0xff);
              if (a != b) {
                return a - b;
              }
            }
            return length1 - length2;
      }
    }


}
}
```

59:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\main\java\org\ethereum\util\FileUtil.java

```
import java.util.ArrayList;
import java.util.List;

public class FileUtil {

    public static List<String> recursiveList(String path) throws IOException {

        final List<String> files = new ArrayList<>();

        Files.walkFileTree(Paths.get(path), new FileVisitor<Path>() {
            @Override
            public FileVisitResult preVisitDirectory(Path dir, BasicFileAttributes attrs) throws
IOException {
                return FileVisitResult.CONTINUE;
            }

            @Override
            public FileVisitResult visitFile(Path file, BasicFileAttributes attrs) throws IOException {
                files.add(file.toString());
                return FileVisitResult.CONTINUE;
            }

            @Override
            public FileVisitResult visitFileFailed(Path file, IOException exc) throws IOException {
                return FileVisitResult.CONTINUE;
            }

            @Override
            public FileVisitResult postVisitDirectory(Path dir, IOException exc) throws IOException {
                return FileVisitResult.CONTINUE;
            }
        });

        return files;
    }

    public static boolean recursiveDelete(String fileName) {
        File file = new File(fileName);
        if (file.exists()) {
            //check if the file is a directory
            if (file.isDirectory()) {
                if ((file.list()).length > 0) {
                    for (String s : file.list()) {
                        //call deletion of file individually
                        recursiveDelete(fileName + System.getProperty("file.separator") + s);
                    }
                }
            }

            file.setWritable(true);
            boolean result = file.delete();
            return result;
        } else {
            return false;
        }
    }

}
```

60:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\main\java\org\ethereum\util\MinMaxMap.java

```
 */
public class MinMaxMap<V> extends TreeMap<Long, V> {

    public void clearAllAfter(long key) {
        if (isEmpty()) {
            return;
        }
        navigableKeySet().subSet(key, false, getMax(), true).clear();
    }

    public void clearAllBefore(long key) {
        if (isEmpty()) {
            return;
        }
        descendingKeySet().subSet(key, false, getMin(), true).clear();
    }

    public Long getMin() {
        return isEmpty() ? null : firstKey();
    }
    public Long getMax() {
        return isEmpty() ? null : lastKey();
    }
}
```

61:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\main\java\org\ethereum\util\RLP.java

```
import java.math.BigInteger;
import java.util.*;

import static java.util.Arrays.copyOfRange;
import static org.ethereum.util.ByteUtil.*;
import static org.spongycastle.util.Arrays.concatenate;
import static org.spongycastle.util.BigIntegers.asUnsignedByteArray;

/**
 * Recursive Length Prefix (RLP) encoding.
 * <p>
 * The purpose of RLP is to encode arbitrarily nested arrays of binary data, and
 * RLP is the main encoding method used to serialize objects in Ethereum. The
 * only purpose of RLP is to encode structure; encoding specific atomic data
 * types (eg. strings, integers, floats) is left up to higher-order protocols; in
 * Ethereum the standard is that integers are represented in big endian binary
 * form. If one wishes to use RLP to encode a dictionary, the two suggested
 * canonical forms are to either use [[k1,v1],[k2,v2]...] with keys in
 * lexicographic order or to use the higher-level Patricia Tree encoding as
 * Ethereum does.
 * <p>
 * The RLP encoding function takes in an item. An item is defined as follows:
 * <p>
 * - A string (ie. byte array) is an item - A list of items is an item
 * <p>
 * For example, an empty string is an item, as is the string containing the word
 * "cat", a list containing any number of strings, as well as more complex data
 * structures like ["cat",["puppy","cow"],"horse",[[]],"pig",[""],"sheep"]. Note
 * that in the context of the rest of this article, "string" will be used as a
 * synonym for "a certain number of bytes of binary data"; no special encodings
 * are used and no knowledge about the content of the strings is implied.
 * <p>
 * See: https://github.com/ethereum/wiki/wiki/%5BEnglish%5D-RLP
 *
 * @author Roman Mandeleil
 * @since 01.04.2014
 */
public class RLP {

    private static final Logger logger = LoggerFactory.getLogger("rlp");



    public static final byte[] EMPTY_ELEMENT_RLP = encodeElement(new byte[0]);

    private static final int MAX_DEPTH = 16;

    /**
     * Allow for content up to size of 2^64 bytes *
     */
    private static final double MAX_ITEM_LENGTH = Math.pow(256, 8);

    /**
     * Reason for threshold according to Vitalik Buterin:
     * - 56 bytes maximizes the benefit of both options
     * - if we went with 60 then we would have only had 4 slots for long strings
     * so RLP would not have been able to store objects above 4gb
     * - if we went with 48 then RLP would be fine for 2^128 space, but that's way too much
     * - so 56 and 2^64 space seems like the right place to put the cutoff
     * - also, that's where Bitcoin's varint does the cutof
     */
    private static final int SIZE_THRESHOLD = 56;

    /** RLP encoding rules are defined as follows: */

    /*
     * For a single byte whose value is in the [0x00, 0x7f] range, that byte is
     * its own RLP encoding.
     */

    /**
     * [0x80]
     * If a string is 0-55 bytes long, the RLP encoding consists of a single
     * byte with value 0x80 plus the length of the string followed by the
     * string. The range of the first byte is thus [0x80, 0xb7].
     */
```

```
private static final int OFFSET_SHORT_ITEM = 0x80;

/**
 * [0xb7]
 * If a string is more than 55 bytes long, the RLP encoding consists of a
 * single byte with value 0xb7 plus the length of the length of the string
 * in binary form, followed by the length of the string, followed by the
 * string. For example, a length-1024 string would be encoded as
 * \xb9\x04\x00 followed by the string. The range of the first byte is thus
 * [0xb8, 0xbf].
 */
private static final int OFFSET_LONG_ITEM = 0xb7;

/**
 * [0xc0]
 * If the total payload of a list (i.e. the combined length of all its
 * items) is 0-55 bytes long, the RLP encoding consists of a single byte
 * with value 0xc0 plus the length of the list followed by the concatenation
 * of the RLP encodings of the items. The range of the first byte is thus
 * [0xc0, 0xf7].
 */
private static final int OFFSET_SHORT_LIST = 0xc0;

/**
 * [0xf7]
 * If the total payload of a list is more than 55 bytes long, the RLP
 * encoding consists of a single byte with value 0xf7 plus the length of the
 * length of the list in binary form, followed by the length of the list,
 * followed by the concatenation of the RLP encodings of the items. The
 * range of the first byte is thus [0xf8, 0xff].
 */
private static final int OFFSET_LONG_LIST = 0xf7;


/* ******************************************************
 *                    DECODING                    *
 * ******************************************************/

private static byte decodeOneByteItem(byte[] data, int index) {
    // null item
    if ((data[index] & 0xFF) == OFFSET_SHORT_ITEM) {
        return (byte) (data[index] - OFFSET_SHORT_ITEM);
        }
        // single byte item
        if ((data[index] & 0xFF) < OFFSET_SHORT_ITEM) {
            return data[index];
        }
        // single byte item
        if ((data[index] & 0xFF) == OFFSET_SHORT_ITEM + 1) {
            return data[index + 1];
        }
        return 0;
    }

    public static int decodeInt(byte[] data, int index) {

        int value = 0;
        // NOTE: From RLP doc:
        // Ethereum integers must be represented in big endian binary form
        // with no leading zeroes (thus making the integer value zero be
        // equivalent to the empty byte array)

        if (data[index] == 0x00) {
            throw new RuntimeException("not a number");
        } else if ((data[index] & 0xFF) < OFFSET_SHORT_ITEM) {

            return data[index];

        } else if ((data[index] & 0xFF) <= OFFSET_SHORT_ITEM + Integer.BYTES) {

            byte length = (byte) (data[index] - OFFSET_SHORT_ITEM);
            byte pow = (byte) (length - 1);
            for (int i = 1; i <= length; ++i) {
                // << (8 * pow) == bit shift to 0 (*1), 8 (*256) , 16 (*65..)..
                value += (data[index + i] & 0xFF) << (8 * pow);
                pow--;
            }
        } else {

            // If there are more than 4 bytes, it is not going
            // to decode properly into an int.
            throw new RuntimeException("wrong decode attempt");
        }
        return value;
```

```
}

static short decodeShort(byte[] data, int index) {

    short value = 0;

    if (data[index] == 0x00) {
        throw new RuntimeException("not a number");
    } else if ((data[index] & 0xFF) < OFFSET_SHORT_ITEM) {

        return data[index];

    } else if ((data[index] & 0xFF) <= OFFSET_SHORT_ITEM + Short.BYTES) {

        byte length = (byte) (data[index] - OFFSET_SHORT_ITEM);
        byte pow = (byte) (length - 1);
        for (int i = 1; i <= length; ++i) {
            // << (8 * pow) == bit shift to 0 (*1), 8 (*256) , 16 (*65..)
            value += (data[index + i] & 0xFF) << (8 * pow);
            pow--;
        }
    } else {

        // If there are more than 2 bytes, it is not going
        // to decode properly into a short.
        throw new RuntimeException("wrong decode attempt");
    }
    return value;
}

public static long decodeLong(byte[] data, int index) {

    long value = 0;

    if (data[index] == 0x00) {
        throw new RuntimeException("not a number");
    } else if ((data[index] & 0xFF) < OFFSET_SHORT_ITEM) {

        return data[index];

    } else if ((data[index] & 0xFF) <= OFFSET_SHORT_ITEM + Long.BYTES) {
        byte length = (byte) (data[index] - OFFSET_SHORT_ITEM);
        byte pow = (byte) (length - 1);
        for (int i = 1; i <= length; ++i) {
            // << (8 * pow) == bit shift to 0 (*1), 8 (*256) , 16 (*65..)..
            value += (long) (data[index + i] & 0xFF) << (8 * pow);
            pow--;
        }
    } else {

        // If there are more than 8 bytes, it is not going
        // to decode properly into a long.
        throw new RuntimeException("wrong decode attempt");
    }
    return value;
}

private static String decodeStringItem(byte[] data, int index) {

    final byte[] valueBytes = decodeItemBytes(data, index);

    if (valueBytes.length == 0) {
        // shortcut
        return "";
    } else {
        return new String(valueBytes);
    }
}

public static BigInteger decodeBigInteger(byte[] data, int index) {

    final byte[] valueBytes = decodeItemBytes(data, index);

    if (valueBytes.length == 0) {
        // shortcut
        return BigInteger.ZERO;
    } else {
        BigInteger res = new BigInteger(1, valueBytes);
        return res;
    }
}

private static byte[] decodeByteArray(byte[] data, int index) {
```

```
    return decodeItemBytes(data, index);
}

private static int nextItemLength(byte[] data, int index) {

    if (index >= data.length) {
        return -1;
    }
    // [0xf8, 0xff]
    if ((data[index] & 0xFF) > OFFSET_LONG_LIST) {
        byte lengthOfLength = (byte) (data[index] - OFFSET_LONG_LIST);

        return calcLength(lengthOfLength, data, index);
    }
    // [0xc0, 0xf7]
    if ((data[index] & 0xFF) >= OFFSET_SHORT_LIST
            && (data[index] & 0xFF) <= OFFSET_LONG_LIST) {

        return (byte) ((data[index] & 0xFF) - OFFSET_SHORT_LIST);
    }
    // [0xb8, 0xbf]
    if ((data[index] & 0xFF) > OFFSET_LONG_ITEM
            && (data[index] & 0xFF) < OFFSET_SHORT_LIST) {

        byte lengthOfLength = (byte) (data[index] - OFFSET_LONG_ITEM);
        return calcLength(lengthOfLength, data, index);
    }
    // [0x81, 0xb7]
    if ((data[index] & 0xFF) > OFFSET_SHORT_ITEM
            && (data[index] & 0xFF) <= OFFSET_LONG_ITEM) {
        return (byte) ((data[index] & 0xFF) - OFFSET_SHORT_ITEM);
    }
    // [0x00, 0x80]
    if ((data[index] & 0xFF) <= OFFSET_SHORT_ITEM) {
        return 1;
    }
    return -1;
}

public static byte[] decodeIP4Bytes(byte[] data, int index) {
    int offset = 1;

    final byte[] result = new byte[4];
    for (int i = 0; i < 4; i++) {
        result[i] = decodeOneByteItem(data, index + offset);
        if ((data[index + offset] & 0xFF) > OFFSET_SHORT_ITEM) {
            offset += 2;
        } else {
            offset += 1;
        }
    }

    // return IP address
    return result;
}

public static int getFirstListElement(byte[] payload, int pos) {

    if (pos >= payload.length) {
        return -1;
    }

    // [0xf8, 0xff]
    if ((payload[pos] & 0xFF) > OFFSET_LONG_LIST) {
        byte lengthOfLength = (byte) (payload[pos] - OFFSET_LONG_LIST);
        return pos + lengthOfLength + 1;
    }
    // [0xc0, 0xf7]
    if ((payload[pos] & 0xFF) >= OFFSET_SHORT_LIST
            && (payload[pos] & 0xFF) <= OFFSET_LONG_LIST) {
        return pos + 1;
    }
    // [0xb8, 0xbf]
    if ((payload[pos] & 0xFF) > OFFSET_LONG_ITEM
            && (payload[pos] & 0xFF) < OFFSET_SHORT_LIST) {
        byte lengthOfLength = (byte) (payload[pos] - OFFSET_LONG_ITEM);
        return pos + lengthOfLength + 1;
    }
    return -1;
}

public static int getNextElementIndex(byte[] payload, int pos) {
```

```
if (pos >= payload.length) {
   return -1;
}

// [0xf8, 0xff]
if ((payload[pos] & 0xFF) > OFFSET_LONG_LIST) {
   byte lengthOfLength = (byte) (payload[pos] - OFFSET_LONG_LIST);
   int length = calcLength(lengthOfLength, payload, pos);
   return pos + lengthOfLength + length + 1;
}
// [0xc0, 0xf7]
if ((payload[pos] & 0xFF) >= OFFSET_SHORT_LIST
      && (payload[pos] & 0xFF) <= OFFSET_LONG_LIST) {

   byte length = (byte) ((payload[pos] & 0xFF) - OFFSET_SHORT_LIST);
   return pos + 1 + length;
}
// [0xb8, 0xbf]
if ((payload[pos] & 0xFF) > OFFSET_LONG_ITEM
      && (payload[pos] & 0xFF) < OFFSET_SHORT_LIST) {

   byte lengthOfLength = (byte) (payload[pos] - OFFSET_LONG_ITEM);
   int length = calcLength(lengthOfLength, payload, pos);
   return pos + lengthOfLength + length + 1;
}
// [0x81, 0xb7]
if ((payload[pos] & 0xFF) > OFFSET_SHORT_ITEM
      && (payload[pos] & 0xFF) <= OFFSET_LONG_ITEM) {

   byte length = (byte) ((payload[pos] & 0xFF) - OFFSET_SHORT_ITEM);
   return pos + 1 + length;
}
// []0x80]
if ((payload[pos] & 0xFF) == OFFSET_SHORT_ITEM) {
   return pos + 1;
}
// [0x00, 0x7f]
if ((payload[pos] & 0xFF) < OFFSET_SHORT_ITEM) {
   return pos + 1;
}
return -1;
```

```
}

/**
 * Parse length of long item or list.
 * RLP supports lengths with up to 8 bytes long,
 * but due to java limitation it returns either encoded length
 * or {@link Integer#MAX_VALUE} in case if encoded length is greater
 *
 * @param lengthOfLength length of length in bytes
 * @param msgData        message
 * @param pos            position to parse from
 * @return calculated length
 */
private static int calcLength(int lengthOfLength, byte[] msgData, int pos) {
    byte pow = (byte) (lengthOfLength - 1);
    int length = 0;
    for (int i = 1; i <= lengthOfLength; ++i) {

        int bt = msgData[pos + i] & 0xFF;
        int shift = 8 * pow;

        // no leading zeros are acceptable
        if (bt == 0 && length == 0) {
            throw new RuntimeException("RLP length contains leading zeros");
        }

        // return MAX_VALUE if index of highest bit is more than 31
        if (32 - Integer.numberOfLeadingZeros(bt) + shift > 31) {
            return Integer.MAX_VALUE;
        }

        length += bt << shift;
        pow--;
    }

    // check that length is in payload bounds
    verifyLength(length, msgData.length - pos - lengthOfLength);

    return length;
}

public static byte getCommandCode(byte[] data) {
    int index = getFirstListElement(data, 0);
    final byte command = data[index];
    return ((command & 0xFF) == OFFSET_SHORT_ITEM) ? 0 : command;
}

/**
 * Parse wire byte[] message into RLP elements
 *
 * @param msgData    - raw RLP data
 * @param depthLimit - limits depth of decoding
 * @return rlpList
 * - outcome of recursive RLP structure
 */
public static RLPList decode2(byte[] msgData, int depthLimit) {
    if (depthLimit < 1) {
        throw new RuntimeException("Depth limit should be 1 or higher");
    }
    RLPList rlpList = new RLPList();
    fullTraverse(msgData, 0, 0, msgData.length, rlpList, depthLimit);
    return rlpList;
}

/**
 * Parse wire byte[] message into RLP elements
 *
 * @param msgData - raw RLP data
 * @return rlpList
 * - outcome of recursive RLP structure
 */
public static RLPList decode2(byte[] msgData) {
    RLPList rlpList = new RLPList();
    fullTraverse(msgData, 0, 0, msgData.length, rlpList, Integer.MAX_VALUE);
    return rlpList;
}

/**
 * Decodes RLP with list without going deep after 1st level list
 * (actually, 2nd as 1st level is wrap only)
 * <p>
 * So assuming you've packed several byte[] with {@link #encodeList(byte[]...)},
 * you could use this method to unpack them,
 * getting RLPList with RLPItem's holding byte[] inside
```

```
     *
     * @param msgData rlp data
     * @return list of RLPItems
     */
    public static RLPList unwrapList(byte[] msgData) {
        return (RLPList) decode2(msgData, 2).get(0);
    }

    public static RLPElement decode2OneItem(byte[] msgData, int startPos) {
        RLPList rlpList = new RLPList();
        fullTraverse(msgData, 0, startPos, startPos + 1, rlpList, Integer.MAX_VALUE);
        return rlpList.get(0);
    }

    /**
     * Get exactly one message payload
     */
    static void fullTraverse(byte[] msgData, int level, int startPos,
                             int endPos, RLPList rlpList, int depth) {
        if (level > MAX_DEPTH) {
            throw new RuntimeException(String.format("Error: Traversing over max RLP depth (%s)",
MAX_DEPTH));
        }

        try {
            if (msgData == null || msgData.length == 0) {
                return;
            }
            int pos = startPos;

            while (pos < endPos) {

                logger.debug("fullTraverse: level: " + level + " startPos: " + pos + " endPos: " + endPos);


                // It's a list with a payload more than 55 bytes
                // data[0] - 0xF7 = how many next bytes allocated
                // for the length of the list
                if ((msgData[pos] & 0xFF) > OFFSET_LONG_LIST) {

                    byte lengthOfLength = (byte) (msgData[pos] - OFFSET_LONG_LIST);
                    int length = calcLength(lengthOfLength, msgData, pos);
        if (length < SIZE_THRESHOLD) {
            throw new RuntimeException("Short list has been encoded as long list");
        }

        // check that length is in payload bounds
        verifyLength(length, msgData.length - pos - lengthOfLength);

        byte[] rlpData = new byte[lengthOfLength + length + 1];
        System.arraycopy(msgData, pos, rlpData, 0, lengthOfLength
                + length + 1);

        if (level + 1 < depth) {
            RLPList newLevelList = new RLPList();
            newLevelList.setRLPData(rlpData);

            fullTraverse(msgData, level + 1, pos + lengthOfLength + 1,
                    pos + lengthOfLength + length + 1, newLevelList, depth);
            rlpList.add(newLevelList);
        } else {
            rlpList.add(new RLPItem(rlpData));
        }

        pos += lengthOfLength + length + 1;
        continue;
    }
    // It's a list with a payload less than 55 bytes
    if ((msgData[pos] & 0xFF) >= OFFSET_SHORT_LIST
            && (msgData[pos] & 0xFF) <= OFFSET_LONG_LIST) {

        byte length = (byte) ((msgData[pos] & 0xFF) - OFFSET_SHORT_LIST);

        byte[] rlpData = new byte[length + 1];
        System.arraycopy(msgData, pos, rlpData, 0, length + 1);

        if (level + 1 < depth) {
            RLPList newLevelList = new RLPList();
            newLevelList.setRLPData(rlpData);

            if (length > 0) {
                fullTraverse(msgData, level + 1, pos + 1, pos + length + 1, newLevelList, depth);
            }
            rlpList.add(newLevelList);
        } else {
            rlpList.add(new RLPItem(rlpData));
        }

        pos += 1 + length;
        continue;
    }
    // It's an item with a payload more than 55 bytes
    // data[0] - 0xB7 = how much next bytes allocated for
    // the length of the string
    if ((msgData[pos] & 0xFF) > OFFSET_LONG_ITEM
            && (msgData[pos] & 0xFF) < OFFSET_SHORT_LIST) {

        byte lengthOfLength = (byte) (msgData[pos] - OFFSET_LONG_ITEM);
        int length = calcLength(lengthOfLength, msgData, pos);

        if (length < SIZE_THRESHOLD) {
            throw new RuntimeException("Short item has been encoded as long item");
        }

        // check that length is in payload bounds
        verifyLength(length, msgData.length - pos - lengthOfLength);

        // now we can parse an item for data[1]..data[length]
        byte[] item = new byte[length];
        System.arraycopy(msgData, pos + lengthOfLength + 1, item,
                0, length);

        RLPItem rlpItem = new RLPItem(item);
        rlpList.add(rlpItem);
        pos += lengthOfLength + length + 1;

        continue;
    }
    // It's an item less than 55 bytes long,
    // data[0] - 0x80 == length of the item
    if ((msgData[pos] & 0xFF) > OFFSET_SHORT_ITEM
            && (msgData[pos] & 0xFF) <= OFFSET_LONG_ITEM) {

        byte length = (byte) ((msgData[pos] & 0xFF) - OFFSET_SHORT_ITEM);
```

```
                byte[] item = new byte[length];
                System.arraycopy(msgData, pos + 1, item, 0, length);

                if (length == 1 && (item[0] & 0xFF) < OFFSET_SHORT_ITEM) {
                    throw new RuntimeException("Single byte has been encoded as byte string");
                }

                RLPItem rlpItem = new RLPItem(item);
                rlpList.add(rlpItem);
                pos += 1 + length;

                continue;
            }
            // null item
            if ((msgData[pos] & 0xFF) == OFFSET_SHORT_ITEM) {
                byte[] item = ByteUtil.EMPTY_BYTE_ARRAY;
                RLPItem rlpItem = new RLPItem(item);
                rlpList.add(rlpItem);
                pos += 1;
                continue;
            }
            // single byte item
            if ((msgData[pos] & 0xFF) < OFFSET_SHORT_ITEM) {

                byte[] item = {(byte) (msgData[pos] & 0xFF)};

                RLPItem rlpItem = new RLPItem(item);
                rlpList.add(rlpItem);
                pos += 1;
            }
        }
    } catch (Exception e) {
        throw new RuntimeException("RLP wrong encoding (" + Hex.toHexString(msgData,
startPos, endPos - startPos) + ")", e);
    } catch (OutOfMemoryError e) {
        throw new RuntimeException("Invalid RLP (excessive mem allocation while parsing) (" +
Hex.toHexString(msgData, startPos, endPos - startPos) + ")", e);
    }
  }

  /**
   * Compares supplied length information with maximum possible
```

```
     *
     * @param suppliedLength  Length info from header
     * @param availableLength Length of remaining object
     * @throws RuntimeException if supplied length is bigger than available
     */
    private static void verifyLength(int suppliedLength, int availableLength) {
        if (suppliedLength > availableLength) {
            throw new RuntimeException(String.format("Length parsed from RLP (%s bytes) is greater
" +
                    "than possible size of data (%s bytes)", suppliedLength, availableLength));
        }
    }

    /**
     * Reads any RLP encoded byte-array and returns all objects as byte-array or list of byte-arrays
     *
     * @param data RLP encoded byte-array
     * @param pos  position in the array to start reading
     * @return DecodeResult encapsulates the decoded items as a single Object and the final read
position
     */
    public static DecodeResult decode(byte[] data, int pos) {
        if (data == null || data.length < 1) {
            return null;
        }
        int prefix = data[pos] & 0xFF;
        if (prefix == OFFSET_SHORT_ITEM) {  // 0x80
            return new DecodeResult(pos + 1, ""); // means no length or 0
        } else if (prefix < OFFSET_SHORT_ITEM) {  // [0x00, 0x7f]
            return new DecodeResult(pos + 1, new byte[]{data[pos]}); // byte is its own RLP encoding
        } else if (prefix <= OFFSET_LONG_ITEM) {  // [0x81, 0xb7]
            int len = prefix - OFFSET_SHORT_ITEM; // length of the encoded bytes
            return new DecodeResult(pos + 1 + len, copyOfRange(data, pos + 1, pos + 1 + len));
        } else if (prefix < OFFSET_SHORT_LIST) {  // [0xb8, 0xbf]
            int lenlen = prefix - OFFSET_LONG_ITEM; // length of length the encoded bytes
            int lenbytes = byteArrayToInt(copyOfRange(data, pos + 1, pos + 1 + lenlen)); // length of
encoded bytes
            // check that length is in payload bounds
            verifyLength(lenbytes, data.length - pos - 1 - lenlen);
            return new DecodeResult(pos + 1 + lenlen + lenbytes, copyOfRange(data, pos + 1 +
lenlen, pos + 1 + lenlen
                    + lenbytes));
        } else if (prefix <= OFFSET_LONG_LIST) {  // [0xc0, 0xf7]
            int len = prefix - OFFSET_SHORT_LIST; // length of the encoded list
            int prevPos = pos;
            pos++;
            return decodeList(data, pos, prevPos, len);
        } else if (prefix <= 0xFF) {  // [0xf8, 0xff]
            int lenlen = prefix - OFFSET_LONG_LIST; // length of length the encoded list
            int lenlist = byteArrayToInt(copyOfRange(data, pos + 1, pos + 1 + lenlen)); // length of
encoded bytes
            pos = pos + lenlen + 1; // start at position of first element in list
            int prevPos = lenlist;
            return decodeList(data, pos, prevPos, lenlist);
        } else {
            throw new RuntimeException("Only byte values between 0x00 and 0xFF are supported,
but got: " + prefix);
        }
    }

    public static final class LList {
        private final byte[] rlp;
        private final int[] offsets = new int[32];
        private final int[] lens = new int[32];
        private int cnt;

        public LList(byte[] rlp) {
            this.rlp = rlp;
        }

        public byte[] getEncoded() {
            byte encoded[][] = new byte[cnt][];
            for (int i = 0; i < cnt; i++) {
                encoded[i] = encodeElement(getBytes(i));
            }
            return encodeList(encoded);
        }

        public void add(int off, int len, boolean isList) {
            offsets[cnt] = off;
            lens[cnt] = isList ? (-1 - len) : len;
            cnt++;
        }
        public byte[] getBytes(int idx) {
            int len = lens[idx];
            len = len < 0 ? (-len - 1) : len;
            byte[] ret = new byte[len];
            System.arraycopy(rlp, offsets[idx], ret, 0, len);
            return ret;
        }

        public LList getList(int idx) {
            return decodeLazyList(rlp, offsets[idx], -lens[idx] - 1);
        }

        public boolean isList(int idx) {
            return lens[idx] < 0;
        }

        public int size() {
            return cnt;
        }
    }

    public static LList decodeLazyList(byte[] data) {
        return decodeLazyList(data, 0, data.length).getList(0);
    }

    public static LList decodeLazyList(byte[] data, int pos, int length) {
        if (data == null || data.length < 1) {
            return null;
        }
        LList ret = new LList(data);
        int end = pos + length;

        while (pos < end) {
            int prefix = data[pos] & 0xFF;
            if (prefix == OFFSET_SHORT_ITEM) {  // 0x80
                ret.add(pos, 0, false); // means no length or 0
                pos++;
            } else if (prefix < OFFSET_SHORT_ITEM) {  // [0x00, 0x7f]
                ret.add(pos, 1, false); // means no length or 0
                pos++;
            } else if (prefix <= OFFSET_LONG_ITEM) {  // [0x81, 0xb7]
                int len = prefix - OFFSET_SHORT_ITEM; // length of the encoded bytes
```

```
                ret.add(pos + 1, len, false);
                pos += len + 1;
            } else if (prefix < OFFSET_SHORT_LIST) {  // [0xb8, 0xbf]
                int lenlen = prefix - OFFSET_LONG_ITEM; // length of length the encoded bytes
                int lenbytes = byteArrayToInt(copyOfRange(data, pos + 1, pos + 1 + lenlen)); // length of
encoded bytes
                // check that length is in payload bounds
                verifyLength(lenbytes, data.length - pos - 1 - lenlen);
                ret.add(pos + 1 + lenlen, lenbytes, false);
                pos += 1 + lenlen + lenbytes;
            } else if (prefix <= OFFSET_LONG_LIST) {  // [0xc0, 0xf7]
                int len = prefix - OFFSET_SHORT_LIST; // length of the encoded list
                ret.add(pos + 1, len, true);
                pos += 1 + len;
            } else if (prefix <= 0xFF) {  // [0xf8, 0xff]
                int lenlen = prefix - OFFSET_LONG_LIST; // length of length the encoded list
                int lenlist = byteArrayToInt(copyOfRange(data, pos + 1, pos + 1 + lenlen)); // length of
encoded bytes
                // check that length is in payload bounds
                verifyLength(lenlist, data.length - pos - 1 - lenlen);
                ret.add(pos + 1 + lenlen, lenlist, true);
                pos += 1 + lenlen + lenlist; // start at position of first element in list
            } else {
                throw new RuntimeException("Only byte values between 0x00 and 0xFF are supported,
but got: " + prefix);
            }
        }
        return ret;
    }


    private static DecodeResult decodeList(byte[] data, int pos, int prevPos, int len) {
        // check that length is in payload bounds
        verifyLength(len, data.length - pos);

        List<Object> slice = new ArrayList<>();
        for (int i = 0; i < len; ) {
            // Get the next item in the data list and append it
            DecodeResult result = decode(data, pos);
            slice.add(result.getDecoded());
            // Increment pos by the amount bytes in the previous read
            prevPos = result.getPos();
        i += (prevPos - pos);
        pos = prevPos;
    }
    return new DecodeResult(pos, slice.toArray());
}

/* ******************************************************
 *                  ENCODING                  *
 * ******************************************************/

/**
 * Turn Object into its RLP encoded equivalent of a byte-array
 * Support for String, Integer, BigInteger and Lists of any of these types.
 *
 * @param input as object or List of objects
 * @return byte[] RLP encoded
 */
public static byte[] encode(Object input) {
    Value val = new Value(input);
    if (val.isList()) {
        List<Object> inputArray = val.asList();
        if (inputArray.isEmpty()) {
            return encodeLength(inputArray.size(), OFFSET_SHORT_LIST);
        }
        byte[] output = ByteUtil.EMPTY_BYTE_ARRAY;
        for (Object object : inputArray) {
            output = concatenate(output, encode(object));
        }
        byte[] prefix = encodeLength(output.length, OFFSET_SHORT_LIST);
        return concatenate(prefix, output);
    } else {
        byte[] inputAsBytes = toBytes(input);
        if (inputAsBytes.length == 1 && (inputAsBytes[0] & 0xff) <= 0x80) {
            return inputAsBytes;
        } else {
            byte[] firstByte = encodeLength(inputAsBytes.length, OFFSET_SHORT_ITEM);
            return concatenate(firstByte, inputAsBytes);
        }
    }
}

/**
```

```
 * Integer limitation goes up to 2^31-1 so length can never be bigger than MAX_ITEM_LENGTH
 */
public static byte[] encodeLength(int length, int offset) {
    if (length < SIZE_THRESHOLD) {
        byte firstByte = (byte) (length + offset);
        return new byte[]{firstByte};
    } else if (length < MAX_ITEM_LENGTH) {
        byte[] binaryLength;
        if (length > 0xFF) {
            binaryLength = intToBytesNoLeadZeroes(length);
        } else {
            binaryLength = new byte[]{(byte) length};
        }
        byte firstByte = (byte) (binaryLength.length + offset + SIZE_THRESHOLD - 1);
        return concatenate(new byte[]{firstByte}, binaryLength);
    } else {
        throw new RuntimeException("Input too long");
    }
}

public static byte[] encodeByte(byte singleByte) {
    if ((singleByte & 0xFF) == 0) {
        return new byte[]{(byte) OFFSET_SHORT_ITEM};
    } else if ((singleByte & 0xFF) <= 0x7F) {
        return new byte[]{singleByte};
    } else {
        return new byte[]{(byte) (OFFSET_SHORT_ITEM + 1), singleByte};
    }
}

public static byte[] encodeShort(short singleShort) {

    if ((singleShort & 0xFF) == singleShort) {
        return encodeByte((byte) singleShort);
    } else {
        return new byte[]{(byte) (OFFSET_SHORT_ITEM + 2),
                (byte) (singleShort >> 8 & 0xFF),
                (byte) (singleShort >> 0 & 0xFF)};
    }
}

public static byte[] encodeInt(int singleInt) {
    if ((singleInt & 0xFF) == singleInt) {
        return encodeByte((byte) singleInt);
    } else if ((singleInt & 0xFFFF) == singleInt) {
        return encodeShort((short) singleInt);
    } else if ((singleInt & 0xFFFFFF) == singleInt) {
        return new byte[]{(byte) (OFFSET_SHORT_ITEM + 3),
                (byte) (singleInt >>> 16),
                (byte) (singleInt >>> 8),
                (byte) singleInt};
    } else {
        return new byte[]{(byte) (OFFSET_SHORT_ITEM + 4),
                (byte) (singleInt >>> 24),
                (byte) (singleInt >>> 16),
                (byte) (singleInt >>> 8),
                (byte) singleInt};
    }
}

public static byte[] encodeString(String srcString) {
    return encodeElement(srcString.getBytes());
}

public static byte[] encodeBigInteger(BigInteger srcBigInteger) {
    if (srcBigInteger.compareTo(BigInteger.ZERO) < 0) {
        throw new RuntimeException("negative numbers are not allowed");
    }

    if (srcBigInteger.equals(BigInteger.ZERO)) {
        return encodeByte((byte) 0);
    } else {
        return encodeElement(asUnsignedByteArray(srcBigInteger));
    }
}

public static byte[] encodeElement(byte[] srcData) {

    // [0x80]
    if (isNullOrZeroArray(srcData)) {
        return new byte[]{(byte) OFFSET_SHORT_ITEM};

        // [0x00]
```

```
} else if (isSingleZero(srcData)) {
    return srcData;

    // [0x01, 0x7f] - single byte, that byte is its own RLP encoding
} else if (srcData.length == 1 && (srcData[0] & 0xFF) < 0x80) {
    return srcData;

    // [0x80, 0xb7], 0 - 55 bytes
} else if (srcData.length < SIZE_THRESHOLD) {
    // length = 8X
    byte length = (byte) (OFFSET_SHORT_ITEM + srcData.length);
    byte[] data = Arrays.copyOf(srcData, srcData.length + 1);
    System.arraycopy(data, 0, data, 1, srcData.length);
    data[0] = length;

    return data;
    // [0xb8, 0xbf], 56+ bytes
} else {
    // length of length = BX
    // prefix = [BX, [length]]
    int tmpLength = srcData.length;
    byte lengthOfLength = 0;
    while (tmpLength != 0) {
        ++lengthOfLength;
        tmpLength = tmpLength >> 8;
    }

    // set length Of length at first byte
    byte[] data = new byte[1 + lengthOfLength + srcData.length];
    data[0] = (byte) (OFFSET_LONG_ITEM + lengthOfLength);

    // copy length after first byte
    tmpLength = srcData.length;
    for (int i = lengthOfLength; i > 0; --i) {
        data[i] = (byte) (tmpLength & 0xFF);
        tmpLength = tmpLength >> 8;
    }

    // at last copy the number bytes after its length
    System.arraycopy(srcData, 0, data, 1 + lengthOfLength, srcData.length);

    return data;
    }
}

public static int calcElementPrefixSize(byte[] srcData) {

    if (isNullOrZeroArray(srcData)) {
        return 0;
    } else if (isSingleZero(srcData)) {
        return 0;
    } else if (srcData.length == 1 && (srcData[0] & 0xFF) < 0x80) {
        return 0;
    } else if (srcData.length < SIZE_THRESHOLD) {
        return 1;
    } else {
        // length of length = BX
        // prefix = [BX, [length]]
        int tmpLength = srcData.length;
        byte byteNum = 0;
        while (tmpLength != 0) {
            ++byteNum;
            tmpLength = tmpLength >> 8;
        }

        return 1 + byteNum;
    }
}


public static byte[] encodeListHeader(int size) {

    if (size == 0) {
        return new byte[]{(byte) OFFSET_SHORT_LIST};
    }

    int totalLength = size;

    byte[] header;
    if (totalLength < SIZE_THRESHOLD) {

        header = new byte[1];
        header[0] = (byte) (OFFSET_SHORT_LIST + totalLength);
    } else {
        // length of length = BX
        // prefix = [BX, [length]]
        int tmpLength = totalLength;
        byte byteNum = 0;
        while (tmpLength != 0) {
            ++byteNum;
            tmpLength = tmpLength >> 8;
        }
        tmpLength = totalLength;

        byte[] lenBytes = new byte[byteNum];
        for (int i = 0; i < byteNum; ++i) {
            lenBytes[byteNum - 1 - i] = (byte) ((tmpLength >> (8 * i)) & 0xFF);
        }
        // first byte = F7 + bytes.length
        header = new byte[1 + lenBytes.length];
        header[0] = (byte) (OFFSET_LONG_LIST + byteNum);
        System.arraycopy(lenBytes, 0, header, 1, lenBytes.length);

    }

    return header;
}


public static byte[] encodeLongElementHeader(int length) {

    if (length < SIZE_THRESHOLD) {

        if (length == 0) {
            return new byte[]{(byte) 0x80};
        } else {
            return new byte[]{(byte) (0x80 + length)};
        }

    } else {

        // length of length = BX
        // prefix = [BX, [length]]
        int tmpLength = length;
        byte byteNum = 0;
        while (tmpLength != 0) {
```

```
            ++byteNum;
            tmpLength = tmpLength >> 8;
        }

        byte[] lenBytes = new byte[byteNum];
        for (int i = 0; i < byteNum; ++i) {
            lenBytes[byteNum - 1 - i] = (byte) ((length >> (8 * i)) & 0xFF);
        }

        // first byte = F7 + bytes.length
        byte[] header = new byte[1 + lenBytes.length];
        header[0] = (byte) (OFFSET_LONG_ITEM + byteNum);
        System.arraycopy(lenBytes, 0, header, 1, lenBytes.length);

        return header;
    }
}

public static byte[] encodeSet(Set<ByteArrayWrapper> data) {

    int dataLength = 0;
    Set<byte[]> encodedElements = new HashSet<>();
    for (ByteArrayWrapper element : data) {

        byte[] encodedElement = RLP.encodeElement(element.getData());
        dataLength += encodedElement.length;
        encodedElements.add(encodedElement);
    }

    byte[] listHeader = encodeListHeader(dataLength);

    byte[] output = new byte[listHeader.length + dataLength];

    System.arraycopy(listHeader, 0, output, 0, listHeader.length);

    int cummStart = listHeader.length;
    for (byte[] element : encodedElements) {
        System.arraycopy(element, 0, output, cummStart, element.length);
        cummStart += element.length;
    }

    return output;
```

```
}

/**
 * A handy shortcut for {@link #encodeElement(byte[])} + {@link #encodeList(byte[]...)}
 * <p>
 * Encodes each data element and wraps them all into a list.
 */
public static byte[] wrapList(byte[]... data) {
    byte[][] elements = new byte[data.length][];
    for (int i = 0; i < data.length; i++) {
        elements[i] = encodeElement(data[i]);
    }
    return encodeList(elements);
}

public static byte[] encodeList(byte[]... elements) {

    if (elements == null) {
        return new byte[]{(byte) OFFSET_SHORT_LIST};
    }

    int totalLength = 0;
    for (byte[] element1 : elements) {
        totalLength += element1.length;
    }

    byte[] data;
    int copyPos;
    if (totalLength < SIZE_THRESHOLD) {

        data = new byte[1 + totalLength];
        data[0] = (byte) (OFFSET_SHORT_LIST + totalLength);
        copyPos = 1;
    } else {
        // length of length = BX
        // prefix = [BX, [length]]
        int tmpLength = totalLength;
        byte byteNum = 0;
        while (tmpLength != 0) {
            ++byteNum;
            tmpLength = tmpLength >> 8;
        }
```

```
            tmpLength = totalLength;
            byte[] lenBytes = new byte[byteNum];
            for (int i = 0; i < byteNum; ++i) {
                lenBytes[byteNum - 1 - i] = (byte) ((tmpLength >> (8 * i)) & 0xFF);
            }
            // first byte = F7 + bytes.length
            data = new byte[1 + lenBytes.length + totalLength];
            data[0] = (byte) (OFFSET_LONG_LIST + byteNum);
            System.arraycopy(lenBytes, 0, data, 1, lenBytes.length);

            copyPos = lenBytes.length + 1;
        }
        for (byte[] element : elements) {
            System.arraycopy(element, 0, data, copyPos, element.length);
            copyPos += element.length;
        }
        return data;
    }

    /*
     *  Utility function to convert Objects into byte arrays
     */
    private static byte[] toBytes(Object input) {
        if (input instanceof byte[]) {
            return (byte[]) input;
        } else if (input instanceof String) {
            String inputString = (String) input;
            return inputString.getBytes();
        } else if (input instanceof Long) {
            Long inputLong = (Long) input;
            return (inputLong == 0) ? ByteUtil.EMPTY_BYTE_ARRAY :
asUnsignedByteArray(BigInteger.valueOf(inputLong));
        } else if (input instanceof Integer) {
            Integer inputInt = (Integer) input;
            return (inputInt == 0) ? ByteUtil.EMPTY_BYTE_ARRAY :
asUnsignedByteArray(BigInteger.valueOf(inputInt));
        } else if (input instanceof BigInteger) {
            BigInteger inputBigInt = (BigInteger) input;
            return (inputBigInt.equals(BigInteger.ZERO)) ? ByteUtil.EMPTY_BYTE_ARRAY :
asUnsignedByteArray(inputBigInt);
        } else if (input instanceof Value) {
            Value val = (Value) input;
            return toBytes(val.asObj());
        }
        throw new RuntimeException("Unsupported type: Only accepting String, Integer and
BigInteger for now");
    }


    private static byte[] decodeItemBytes(byte[] data, int index) {

        final int length = calculateItemLength(data, index);
        // [0x80]
        if (length == 0) {

            return new byte[0];

            // [0x00, 0x7f] - single byte with item
        } else if ((data[index] & 0xFF) < OFFSET_SHORT_ITEM) {

            byte[] valueBytes = new byte[1];
            System.arraycopy(data, index, valueBytes, 0, 1);
            return valueBytes;

            // [0x01, 0xb7] - 1-55 bytes item
        } else if ((data[index] & 0xFF) <= OFFSET_LONG_ITEM) {

            byte[] valueBytes = new byte[length];
            System.arraycopy(data, index + 1, valueBytes, 0, length);
            return valueBytes;

            // [0xb8, 0xbf] - 56+ bytes item
        } else if ((data[index] & 0xFF) > OFFSET_LONG_ITEM
                && (data[index] & 0xFF) < OFFSET_SHORT_LIST) {

            byte lengthOfLength = (byte) (data[index] - OFFSET_LONG_ITEM);
            byte[] valueBytes = new byte[length];
            System.arraycopy(data, index + 1 + lengthOfLength, valueBytes, 0, length);
            return valueBytes;
        } else {
            throw new RuntimeException("wrong decode attempt");
        }
    }
    private static int calculateItemLength(byte[] data, int index) {

        // [0xb8, 0xbf] - 56+ bytes item
        if ((data[index] & 0xFF) > OFFSET_LONG_ITEM
                && (data[index] & 0xFF) < OFFSET_SHORT_LIST) {

            byte lengthOfLength = (byte) (data[index] - OFFSET_LONG_ITEM);
            return calcLength(lengthOfLength, data, index);

            // [0x81, 0xb7] - 0-55 bytes item
        } else if ((data[index] & 0xFF) > OFFSET_SHORT_ITEM
                && (data[index] & 0xFF) <= OFFSET_LONG_ITEM) {

            return (byte) (data[index] - OFFSET_SHORT_ITEM);

            // [0x80] - item = 0 itself
        } else if ((data[index] & 0xFF) == OFFSET_SHORT_ITEM) {

            return (byte) 0;

            // [0x00, 0x7f] - 1 byte item, no separate length representation
        } else if ((data[index] & 0xFF) < OFFSET_SHORT_ITEM) {

            return (byte) 1;

        } else {
            throw new RuntimeException("wrong decode attempt");
        }
    }
}
```

62:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\main\java\org\ethereum\util\RLPElement.java

```
 *
 * @author Roman Mandeleil
 * @since 01.04.2014
 */
public interface RLPElement extends Serializable {

    byte[] getRLPData();
}
```

63:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\main\java\org\ethereum\util\RLPItem.java

```
public class RLPItem implements RLPElement {

    private final byte[] rlpData;

    public RLPItem(byte[] rlpData) {
        this.rlpData = rlpData;
    }

    @Override
    public byte[] getRLPData() {
        if (rlpData.length == 0) {
            return null;
        }
        return rlpData;
    }
}
```

64:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\main\java\org\ethereum\util\RLPList.java

```
 * @since 21.04.14
 */
public class RLPList extends ArrayList<RLPElement> implements RLPElement {

    byte[] rlpData;

    public void setRLPData(byte[] rlpData) {
        this.rlpData = rlpData;
    }

    @Override
    public byte[] getRLPData() {
        return rlpData;
    }

    public static void recursivePrint(RLPElement element) {

        if (element == null) {
            throw new RuntimeException("RLPElement object can't be null");
        }
        if (element instanceof RLPList) {

            RLPList rlpList = (RLPList) element;
            System.out.print("[");
            for (RLPElement singleElement : rlpList) {
                recursivePrint(singleElement);
            }
            System.out.print("]");
        } else {
            String hex = ByteUtil.toHexString(element.getRLPData());
            System.out.print(hex + ", ");
        }
    }
}
```

65:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\main\java\org\ethereum\util\SetAdapter.java

```
/**
 * Created by Anton Nashatyrev on 06.10.2016.
 */
public class SetAdapter<E> implements Set<E> {
    private static final Object DummyValue = new Object();
    Map<E, Object> delegate;

    public SetAdapter(Map<E, ?> delegate) {
        this.delegate = (Map<E, Object>) delegate;
    }

    @Override
    public int size() {
        return delegate.size();
    }

    @Override
    public boolean isEmpty() {
        return delegate.isEmpty();
    }

    @Override
    public boolean contains(Object o) {
        return delegate.containsKey(o);
    }

    @Override
    public Iterator<E> iterator() {
        return delegate.keySet().iterator();
    }

    @Override
    public Object[] toArray() {
        return delegate.keySet().toArray();
    }

    @Override
    public <T> T[] toArray(T[] a) {
        return delegate.keySet().toArray(a);
    }

    @Override
    public boolean add(E e) {
        return delegate.put(e, DummyValue) == null;
    }

    @Override
    public boolean remove(Object o) {
        return delegate.remove(o) != null;
    }

    @Override
    public boolean containsAll(Collection<?> c) {
        return delegate.keySet().containsAll(c);
    }

    @Override
    public boolean addAll(Collection<? extends E> c) {
        boolean ret = false;
        for (E e : c) {
            ret |= add(e);
        }
        return ret;
    }

    @Override
```

```
    public boolean retainAll(Collection<?> c) {
        throw new RuntimeException("Not implemented"); // TODO add later if required
    }

    @Override
    public boolean removeAll(Collection<?> c) {
        boolean ret = false;
        for (Object e : c) {
            ret |= remove(e);
        }
        return ret;
    }

    @Override
    public void clear() {
        delegate.clear();
    }
}
```

66:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\main\java\org\ethereum\util\TimeUtils.java

```
public class TimeUtils {

    /**
     * Converts minutes to millis
     *
     * @param minutes time in minutes
     * @return corresponding millis value
     */
    public static long minutesToMillis(long minutes) {
        return minutes * 60 * 1000;
    }

    /**
     * Converts seconds to millis
     *
     * @param seconds time in seconds
     * @return corresponding millis value
     */
    public static long secondsToMillis(long seconds) {
        return seconds * 1000;
    }
    /**
     * Converts millis to minutes
     *
     * @param millis time in millis
     * @return time in minutes
     */
    public static long millisToMinutes(long millis) {
        return Math.round(millis / 60.0 / 1000.0);
    }

    /**
     * Converts millis to seconds
     *
     * @param millis time in millis
     * @return time in seconds
     */
    public static long millisToSeconds(long millis) {
        return Math.round(millis / 1000.0);
    }

    /**
     * Returns timestamp in the future after some millis passed from now
     *
     * @param millis millis count
     * @return future timestamp
     */
    public static long timeAfterMillis(long millis) {
        return System.currentTimeMillis() + millis;
    }
}
```

67:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\main\java\org\ethereum\util\Utils.java

```
import org.spongycastle.util.encoders.DecoderException;
import org.spongycastle.util.encoders.Hex;

import javax.swing.*;
import java.lang.reflect.Array;
import java.math.BigInteger;
import java.net.URL;
import java.security.SecureRandom;
import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.util.*;
import java.util.regex.Pattern;

public class Utils {
    private static final DataWord DIVISOR = DataWord.of(64);

    private static SecureRandom random = new SecureRandom();

    /**
     * @param number should be in form '0x34fabd34....'
     * @return String
     */
    public static BigInteger unifiedNumericToBigInteger(String number) {

        boolean match = Pattern.matches("0[xX][0-9a-fA-F]+", number);
        if (!match) {
            return (new BigInteger(number));
        } else {
            number = number.substring(2);
            number = number.length() % 2 != 0 ? "0".concat(number) : number;
            byte[] numberBytes = Hex.decode(number);
            return (new BigInteger(1, numberBytes));
        }
    }

    /**
     * Return formatted Date String: yyyy.MM.dd HH:mm:ss
     * Based on Unix's time() input in seconds
     *
     * @param timestamp seconds since start of Unix-time
     * @return String formatted as - yyyy.MM.dd HH:mm:ss
     */
    public static String longToDateTime(long timestamp) {
        Date date = new Date(timestamp * 1000);
        DateFormat formatter = new SimpleDateFormat("yyyy.MM.dd HH:mm:ss");
        return formatter.format(date);
    }

    public static String longToTimePeriod(long msec) {
        if (msec < 1000) {
```

```
        return msec + "ms";
    }
    if (msec < 3000) {
        return String.format("%.2fs", msec / 1000d);
    }
    if (msec < 60 * 1000) {
        return (msec / 1000) + "s";
    }
    long sec = msec / 1000;
    if (sec < 5 * 60) {
        return (sec / 60) + "m" + (sec % 60) + "s";
    }
    long min = sec / 60;
    if (min < 60) {
        return min + "m";
    }
    long hour = min / 60;
    if (min < 24 * 60) {
        return hour + "h" + (min % 60) + "m";
    }
    long day = hour / 24;
    return day + "d" + (hour % 24) + "h";
}

public static ImageIcon getImageIcon(String resource) {
    URL imageURL = ClassLoader.getSystemResource(resource);
    ImageIcon image = new ImageIcon(imageURL);
    return image;
}

static BigInteger _1000_ = new BigInteger("1000");

public static String getValueShortString(BigInteger number) {
    BigInteger result = number;
    int pow = 0;
    while (result.compareTo(_1000_) == 1 || result.compareTo(_1000_) == 0) {
        result = result.divide(_1000_);
        pow += 3;
    }
    return result.toString() + "\u00b7(" + "10^" + pow + ")";
}
```

```
    /**
     * Decodes a hex string to address bytes and checks validity
     *
     * @param hex - a hex string of the address, e.g.,
6c386a4b26f73c802f34673f7248bb118f97424a
     * @return - decode and validated address byte[]
     */
    public static byte[] addressStringToBytes(String hex) {
        final byte[] addr;
        try {
            addr = Hex.decode(hex);
        } catch (DecoderException addressIsNotValid) {
            return null;
        }

        if (isValidAddress(addr)) {
            return addr;
        }
        return null;
    }

    public static boolean isValidAddress(byte[] addr) {
        return addr != null && addr.length == 20;
    }

    /**
     * @param addr length should be 20
     * @return short string represent 1f21c...
     */
    public static String getAddressShortString(byte[] addr) {

        if (!isValidAddress(addr)) {
            throw new Error("not an address");
        }

        String addrShort = Hex.toHexString(addr, 0, 3);

        StringBuffer sb = new StringBuffer();
        sb.append(addrShort);
        sb.append("...");

        return sb.toString();
```

```
}

public static SecureRandom getRandom() {
    return random;
}

public static double JAVA_VERSION = getJavaVersion();

static double getJavaVersion() {
    String version = System.getProperty("java.version");

    // on android this property equals to 0
    if ("0".equals(version)) {
        return 0;
    }

    int pos = 0, count = 0;
    for (; pos < version.length() && count < 2; pos++) {
        if (version.charAt(pos) == '.') {
            count++;
        }
    }
    return Double.parseDouble(version.substring(0, pos - 1));
}

public static String getHashListShort(List<byte[]> blockHashes) {
    if (blockHashes.isEmpty()) {
        return "[]";
    }

    StringBuilder sb = new StringBuilder();
    String firstHash = Hex.toHexString(blockHashes.get(0));
    String lastHash = Hex.toHexString(blockHashes.get(blockHashes.size() - 1));
    return sb.append(" ").append(firstHash).append("...").append(lastHash).toString();
}

public static String getNodeIdShort(String nodeId) {
    return nodeId == null ? "<null>" : nodeId.substring(0, 8);
}

public static long toUnixTime(long javaTime) {
    return javaTime / 1000;
```

```
}

public static long fromUnixTime(long unixTime) {
    return unixTime * 1000;
}

public static <T> T[] mergeArrays(T[]... arr) {
    int size = 0;
    for (T[] ts : arr) {
        size += ts.length;
    }
    T[] ret = (T[]) Array.newInstance(arr[0].getClass().getComponentType(), size);
    int off = 0;
    for (T[] ts : arr) {
        System.arraycopy(ts, 0, ret, off, ts.length);
        off += ts.length;
    }
    return ret;
}

public static String align(String s, char fillChar, int targetLen, boolean alignRight) {
    if (targetLen <= s.length()) {
        return s;
    }
    String alignString = repeat("" + fillChar, targetLen - s.length());
    return alignRight ? alignString + s : s + alignString;

}

public static String repeat(String s, int n) {
    if (s.length() == 1) {
        byte[] bb = new byte[n];
        Arrays.fill(bb, s.getBytes()[0]);
        return new String(bb);
    } else {
        StringBuilder ret = new StringBuilder();
        for (int i = 0; i < n; i++) {
            ret.append(s);
        }
        return ret.toString();
    }
}
```

```
public static List<ByteArrayWrapper> dumpKeys(DbSource<byte[]> ds) {

    ArrayList<ByteArrayWrapper> keys = new ArrayList<>();

    for (byte[] key : ds.keys()) {
        keys.add(ByteUtil.wrap(key));
    }
    Collections.sort(keys);
    return keys;
}

public static DataWord allButOne64th(DataWord dw) {
    DataWord divResult = dw.div(DIVISOR);
    return dw.sub(divResult);
}

/**
 * Show std err messages in red and throw RuntimeException to stop execution.
 */
public static void showErrorAndExit(String message, String... messages) {
    LoggerFactory.getLogger("general").error(message);
    final String ANSI_RED = "\u001B[31m";
    final String ANSI_RESET = "\u001B[0m";

    System.err.println(ANSI_RED);
    System.err.println("");
    System.err.println("        " + message);
    for (String msg : messages) {
        System.err.println("        " + msg);
    }
    System.err.println("");
    System.err.println(ANSI_RESET);

    throw new RuntimeException(message);
}

/**
 * Show std warning messages in red.
 */
public static void showWarn(String message, String... messages) {
    LoggerFactory.getLogger("general").warn(message);
```

```
        final String ANSI_RED = "\u001B[31m";
        final String ANSI_RESET = "\u001B[0m";

        System.err.println(ANSI_RED);
        System.err.println("");
        System.err.println("        " + message);
        for (String msg : messages) {
            System.err.println("        " + msg);
        }
        System.err.println("");
        System.err.println(ANSI_RESET);
    }

    public static String sizeToStr(long size) {
        if (size < 2 * (1L << 10)) {
            return size + "b";
        }
        if (size < 2 * (1L << 20)) {
            return String.format("%dKb", size / (1L << 10));
        }
        if (size < 2 * (1L << 30)) {
            return String.format("%dMb", size / (1L << 20));
        }
        return String.format("%dGb", size / (1L << 30));
    }

    public static void sleep(long ms) {
        try {
            Thread.sleep(ms);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }

    public static boolean isHexEncoded(String value) {
        if (value == null) {
            return false;
        }
        if ("".equals(value)) {
            return true;
        }
        try {
            //noinspection ResultOfMethodCallIgnored
            new BigInteger(value, 16);
            return true;
        } catch (NumberFormatException e) {
            return false;
        }
    }
}
```

68:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\main\java\org\ethereum\util\Value.java

```
import java.util.Arrays;
import java.util.List;

import static org.ethereum.util.ByteUtil.toHexString;

/**
 * Class to encapsulate an object and provide utilities for conversion
 */
public class Value {

    private Object value;
    private byte[] rlp;
    private byte[] sha3;

    private boolean decoded = false;

    public static Value fromRlpEncoded(byte[] data) {

        if (data != null && data.length != 0) {
            Value v = new Value();
            v.init(data);
            return v;
        }
        return null;
    }

    public Value() {
    }

    public void init(byte[] rlp) {
        this.rlp = rlp;
    }

    public Value(Object obj) {

        this.decoded = true;
        if (obj == null) {
            return;
        }

        if (obj instanceof Value) {
            this.value = ((Value) obj).asObj();
        } else {
            this.value = obj;
        }
    }

    public Value withHash(byte[] hash) {
        sha3 = hash;
        return this;
    }

    /* *****************
     *      Convert
     * *****************/

    public Object asObj() {
        decode();
        return value;
    }

    public List<Object> asList() {
        decode();
        Object[] valueArray = (Object[]) value;
        return Arrays.asList(valueArray);
    }

    public int asInt() {
        decode();
        if (isInt()) {
            return (Integer) value;
        } else if (isBytes()) {
            return new BigInteger(1, asBytes()).intValue();
        }
        return 0;
    }

    public long asLong() {
        decode();
        if (isLong()) {
            return (Long) value;
        } else if (isBytes()) {
            return new BigInteger(1, asBytes()).longValue();
        }
        return 0;
    }

    public BigInteger asBigInt() {
        decode();
        return (BigInteger) value;
    }

    public String asString() {
        decode();
        if (isBytes()) {
            return new String((byte[]) value);
        } else if (isString()) {
            return (String) value;
        }
        return "";
    }

    public byte[] asBytes() {
        decode();
        if (isBytes()) {
            return (byte[]) value;
        } else if (isString()) {
            return asString().getBytes();
        }
        return ByteUtil.EMPTY_BYTE_ARRAY;
    }

    public String getHex() {
        return Hex.toHexString(this.encode());
```

```
}

public byte[] getData() {
    return this.encode();
}



public int[] asSlice() {
    return (int[]) value;
}

public Value get(int index) {
    if (isList()) {
        // Guard for OutOfBounds
        if (asList().size() <= index) {
            return new Value(null);
        }
        if (index < 0) {
            throw new RuntimeException("Negative index not allowed");
        }
        return new Value(asList().get(index));
    }
    // If this wasn't a slice you probably shouldn't be using this function
    return new Value(null);
}

/* *****************
 *      Utility
 * *****************/

public void decode() {
    if (!this.decoded) {
        this.value = RLP.decode(rlp, 0).getDecoded();
        this.decoded = true;
    }
}

public byte[] encode() {
    if (rlp == null) {
        rlp = RLP.encode(value);
    }
    return rlp;
    }

    public byte[] hash() {
        if (sha3 == null) {
            sha3 = HashUtil.sha3(encode());
        }
        return sha3;
    }

    /* *****************
     *      Checks
     * *****************/

    public boolean isList() {
        decode();
        return value != null && value.getClass().isArray() &&
!value.getClass().getComponentType().isPrimitive();
    }

    public boolean isString() {
        decode();
        return value instanceof String;
    }

    public boolean isInt() {
        decode();
        return value instanceof Integer;
    }

    public boolean isLong() {
        decode();
        return value instanceof Long;
    }

    public boolean isBigInt() {
        decode();
        return value instanceof BigInteger;
    }

    public boolean isBytes() {
        decode();
        return value instanceof byte[];
```

```
}

// it's only if the isBytes() = true;
public boolean isReadableString() {

    decode();
    int readableChars = 0;
    byte[] data = (byte[]) value;

    if (data.length == 1 && data[0] > 31 && data[0] < 126) {
        return true;
    }

    for (byte aData : data) {
        if (aData > 32 && aData < 126) {
            ++readableChars;
        }
    }

    return (double) readableChars / (double) data.length > 0.55;
}

// it's only if the isBytes() = true;
public boolean isHexString() {

    decode();
    int hexChars = 0;
    byte[] data = (byte[]) value;

    for (byte aData : data) {

        if ((aData >= 48 && aData <= 57)
                || (aData >= 97 && aData <= 102)) {
            ++hexChars;
        }
    }

    return (double) hexChars / (double) data.length > 0.9;
}

public boolean isHashCode() {
    decode();
    return this.asBytes().length == 32;
}

public boolean isNull() {
    decode();
    return value == null;
}

public boolean isEmpty() {
    decode();
    if (isNull()) {
        return true;
    }
    if (isBytes() && asBytes().length == 0) {
        return true;
    }
    if (isList() && asList().isEmpty()) {
        return true;
    }
    if (isString() && asString().isEmpty()) {
        return true;
    }

    return false;
}

public int length() {
    decode();
    if (isList()) {
        return asList().size();
    } else if (isBytes()) {
        return asBytes().length;
    } else if (isString()) {
        return asString().length();
    }
    return 0;
}

@Override
public String toString() {

    decode();
```

```
StringBuilder stringBuilder = new StringBuilder();

if (isList()) {

    Object[] list = (Object[]) value;

    // special case - key/value node
    if (list.length == 2) {

        stringBuilder.append("[ ");

        Value key = new Value(list[0]);

        byte[] keyNibbles = CompactEncoder.binToNibblesNoTerminator(key.asBytes());
        String keyString = ByteUtil.nibblesToPrettyString(keyNibbles);
        stringBuilder.append(keyString);

        stringBuilder.append(",");

        Value val = new Value(list[1]);
        stringBuilder.append(val.toString());

        stringBuilder.append(" ]");
        return stringBuilder.toString();
    }
    stringBuilder.append(" [");

    for (int i = 0; i < list.length; ++i) {
        Value val = new Value(list[i]);
        if (val.isString() || val.isEmpty()) {
            stringBuilder.append("'").append(val.toString()).append("'");
        } else {
            stringBuilder.append(val.toString());
        }
        if (i < list.length - 1) {
            stringBuilder.append(", ");
        }
    }
    stringBuilder.append("] ");

    return stringBuilder.toString();
} else if (isEmpty()) {
```

```
            return "";
        } else if (isBytes()) {

            StringBuilder output = new StringBuilder();
            if (isHashCode()) {
                output.append(toHexString(asBytes()));
            } else if (isReadableString()) {
                output.append("'");
                for (byte oneByte : asBytes()) {
                    if (oneByte < 16) {
                        output.append("\\x").append(ByteUtil.oneByteToHexString(oneByte));
                    } else {
                        output.append(Character.valueOf((char) oneByte));
                    }
                }
                output.append("'");
                return output.toString();
            }
            return toHexString(this.asBytes());
        } else if (isString()) {
            return asString();
        }
        return "Unexpected type";
    }

    public int countBranchNodes() {
        decode();
        if (this.isList()) {
            List<Object> objList = this.asList();
            int i = 0;
            for (Object obj : objList) {
                i += (new Value(obj)).countBranchNodes();
            }
            return i;
        } else if (this.isBytes()) {
            this.asBytes();
        }
        return 0;
    }
}
```

69:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-

vm\src\main\java\org\ethereum\vm\DataWord.java

```
import org.ethereum.util.FastByteComparisons;
import org.spongycastle.util.encoders.Hex;

import java.math.BigInteger;
import java.util.Arrays;

import static org.ethereum.util.ByteUtil.*;

/**
 * DataWord is the 32-byte array representation of a 256-bit number
 * Calculations can be done on this word with other DataWords
 * DataWord is immutable. Use one of `of` factories for instance creation.
 *
 * @author Roman Mandeleil
 * @since 01.06.2014
 */
public final class DataWord implements Comparable<DataWord> {

    /* Maximum value of the DataWord */
    public static final int MAX_POW = 256;
    public static final BigInteger _2_256 = BigInteger.valueOf(2).pow(MAX_POW);
    public static final BigInteger MAX_VALUE = _2_256.subtract(BigInteger.ONE);
    public static final DataWord ZERO = new DataWord(new byte[32]);
    public static final DataWord ONE = DataWord.of((byte) 1);

    public static final long MEM_SIZE = 32 + 16 + 16;

    private final byte[] data;

    /**
     * Unsafe private constructor
     * Doesn't guarantee immutability if byte[] contents are changed later
     * Use one of factory methods instead:
     * - {@link #of(byte[])}
     * - {@link #of(ByteArrayWrapper)}
     * - {@link #of(String)}
     * - {@link #of(long)}
     * - {@link #of(int)}
     *
     * @param data Byte Array[32] which is guaranteed to be immutable
     */
    private DataWord(byte[] data) {
        if (data == null) {
            throw new RuntimeException("Input byte array should have 32 bytes in it!");
        }
        this.data = data;
    }

    public static DataWord of(byte[] data) {
        if (data == null || data.length == 0) {
            return DataWord.ZERO;
        }

        int leadingZeroBits = numberOfLeadingZeros(data);
        int valueBits = 8 * data.length - leadingZeroBits;
        if (valueBits <= 8) {
            if (data[data.length - 1] == 0) {
                return DataWord.ZERO;
            }
            if (data[data.length - 1] == 1) {
                return DataWord.ONE;
            }
        }

        if (data.length >= 32) {
            return new DataWord(Arrays.copyOf(data, data.length));
        } else if (data.length <= 32) {
            byte[] bytes = new byte[32];
            System.arraycopy(data, 0, bytes, 32 - data.length, data.length);
            return new DataWord(bytes);
        } else {
            throw new RuntimeException(String.format("Data word can't exceed 32 bytes: 0x%s",
ByteUtil.toHexString(data)));
        }
    }

    private String sData;

    public DataWord(String data) {
        this(data.getBytes());
        this.sData = data;
    }
```

```
public static DataWord of(ByteArrayWrapper wrappedData) {
    return of(wrappedData.getData());
}

@JsonCreator
public static DataWord of(String data) {
    return of(Hex.decode(data));
}

public static DataWord of(byte num) {
    byte[] bb = new byte[32];
    bb[31] = num;
    return new DataWord(bb);

}

public static DataWord of(int num) {
    return of(intToBytes(num));
}

public static DataWord of(long num) {
    return of(longToBytes(num));
}

/**
 * Returns instance data
 * Actually copy of internal byte array is provided
 * in order to protect DataWord immutability
 *
 * @return instance data
 */
public byte[] getData() {
    return Arrays.copyOf(data, data.length);
}

/**
 * Returns copy of instance data
 *
 * @return copy of instance data
 */
private byte[] copyData() {
    return Arrays.copyOf(data, data.length);
```

```
}

public byte[] getNoLeadZeroesData() {
    return ByteUtil.stripLeadingZeroes(copyData());
}

public byte[] getLast20Bytes() {
    return Arrays.copyOfRange(data, 12, data.length);
}

public BigInteger value() {
    return new BigInteger(1, data);
}

/**
 * Converts this DataWord to an int, checking for lost information.
 * If this DataWord is out of the possible range for an int result
 * then an ArithmeticException is thrown.
 *
 * @return this DataWord converted to an int.
 * @throws ArithmeticException - if this will not fit in an int.
 */
public int intValue() {
    int intVal = 0;

    for (byte aData : data) {
        intVal = (intVal << 8) + (aData & 0xff);
    }

    return intVal;
}

/**
 * In case of int overflow returns Integer.MAX_VALUE
 * otherwise works as #intValue()
 */
public int intValueSafe() {
    int bytesOccupied = bytesOccupied();
    int intValue = intValue();
    if (bytesOccupied > 4 || intValue < 0) {
        return Integer.MAX_VALUE;
    }
    return intValue;
}

/**
 * Converts this DataWord to a long, checking for lost information.
 * If this DataWord is out of the possible range for a long result
 * then an ArithmeticException is thrown.
 *
 * @return this DataWord converted to a long.
 * @throws ArithmeticException - if this will not fit in a long.
 */
public long longValue() {

    long longVal = 0;
    for (byte aData : data) {
        longVal = (longVal << 8) + (aData & 0xff);
    }

    return longVal;
}

/**
 * In case of long overflow returns Long.MAX_VALUE
 * otherwise works as #longValue()
 */
public long longValueSafe() {
    int bytesOccupied = bytesOccupied();
    long longValue = longValue();
    if (bytesOccupied > 8 || longValue < 0) {
        return Long.MAX_VALUE;
    }
    return longValue;
}

public BigInteger sValue() {
    return new BigInteger(data);
}

public String bigIntValue() {
    return new BigInteger(data).toString();
}
```

```
public boolean isZero() {
    if (this == ZERO) {
        return true;
    }
    return this.compareTo(ZERO) == 0;
}

// only in case of signed operation
// when the number is explicit defined
// as negative
public boolean isNegative() {
    int result = data[0] & 0x80;
    return result == 0x80;
}

public DataWord and(DataWord word) {
    byte[] newData = this.copyData();
    for (int i = 0; i < this.data.length; ++i) {
        newData[i] &= word.data[i];
    }
    return new DataWord(newData);
}

public DataWord or(DataWord word) {
    byte[] newData = this.copyData();
    for (int i = 0; i < this.data.length; ++i) {
        newData[i] |= word.data[i];
    }
    return new DataWord(newData);
}

public DataWord xor(DataWord word) {
    byte[] newData = this.copyData();
    for (int i = 0; i < this.data.length; ++i) {
        newData[i] ^= word.data[i];
    }
    return new DataWord(newData);
}

public DataWord negate() {
    if (this.isZero()) {
        return ZERO;
    }
    return bnot().add(DataWord.ONE);
}

public DataWord bnot() {
    if (this.isZero()) {
        return new DataWord(ByteUtil.copyToArray(MAX_VALUE));
    }
    return new DataWord(ByteUtil.copyToArray(MAX_VALUE.subtract(this.value())));
}

// By   : Holger
// From : http://stackoverflow.com/a/24023466/459349
public DataWord add(DataWord word) {
    byte[] newData = new byte[32];
    for (int i = 31, overflow = 0; i >= 0; i--) {
        int v = (this.data[i] & 0xff) + (word.data[i] & 0xff) + overflow;
        newData[i] = (byte) v;
        overflow = v >>> 8;
    }
    return new DataWord(newData);
}

// old add-method with BigInteger quick hack
public DataWord add2(DataWord word) {
    BigInteger result = value().add(word.value());
    return new DataWord(ByteUtil.copyToArray(result.and(MAX_VALUE)));
}

// TODO: mul can be done in more efficient way
// TODO:     with shift left shift right trick
// TODO      without BigInteger quick hack
public DataWord mul(DataWord word) {
    BigInteger result = value().multiply(word.value());
    return new DataWord(ByteUtil.copyToArray(result.and(MAX_VALUE)));
}

// TODO: improve with no BigInteger
public DataWord div(DataWord word) {

    if (word.isZero()) {
        return ZERO;
    }

    BigInteger result = value().divide(word.value());
    return new DataWord(ByteUtil.copyToArray(result.and(MAX_VALUE)));
}

// TODO: improve with no BigInteger
public DataWord sDiv(DataWord word) {

    if (word.isZero()) {
        return ZERO;
    }

    BigInteger result = sValue().divide(word.sValue());
    return new DataWord(ByteUtil.copyToArray(result.and(MAX_VALUE)));
}

// TODO: improve with no BigInteger
public DataWord sub(DataWord word) {
    BigInteger result = value().subtract(word.value());
    return new DataWord(ByteUtil.copyToArray(result.and(MAX_VALUE)));
}

// TODO: improve with no BigInteger
public DataWord exp(DataWord word) {
    BigInteger newData = value().modPow(word.value(), _2_256);
    return new DataWord(ByteUtil.copyToArray(newData));
}

// TODO: improve with no BigInteger
public DataWord mod(DataWord word) {

    if (word.isZero()) {
        return ZERO;
    }

    BigInteger result = value().mod(word.value());
    return new DataWord(ByteUtil.copyToArray(result.and(MAX_VALUE)));
}

public DataWord sMod(DataWord word) {
    if (word.isZero()) {
        return ZERO;
    }

    BigInteger result = sValue().abs().mod(word.sValue().abs());
    result = (sValue().signum() == -1) ? result.negate() : result;

    return new DataWord(ByteUtil.copyToArray(result.and(MAX_VALUE)));
}

public DataWord addmod(DataWord word1, DataWord word2) {
    if (word2.isZero()) {
        return ZERO;
    }

    BigInteger result = value().add(word1.value()).mod(word2.value());
    return new DataWord(ByteUtil.copyToArray(result.and(MAX_VALUE)));
}

public DataWord mulmod(DataWord word1, DataWord word2) {

    if (this.isZero() || word1.isZero() || word2.isZero()) {
        return ZERO;
    }

    BigInteger result = value().multiply(word1.value()).mod(word2.value());
    return new DataWord(ByteUtil.copyToArray(result.and(MAX_VALUE)));
}

/**
 * Shift left, both this and input arg are treated as unsigned
 *
 * @param arg
 * @return this << arg
 */
public DataWord shiftLeft(DataWord arg) {
    if (arg.value().compareTo(BigInteger.valueOf(MAX_POW)) >= 0) {
        return DataWord.ZERO;
    }

    BigInteger result = value().shiftLeft(arg.intValueSafe());
    return new DataWord(ByteUtil.copyToArray(result.and(MAX_VALUE)));
```

```
}

/**
 * Shift right, both this and input arg are treated as unsigned
 *
 * @param arg
 * @return this >> arg
 */
public DataWord shiftRight(DataWord arg) {
    if (arg.value().compareTo(BigInteger.valueOf(MAX_POW)) >= 0) {
        return DataWord.ZERO;
    }

    BigInteger result = value().shiftRight(arg.intValueSafe());
    return new DataWord(ByteUtil.copyToArray(result.and(MAX_VALUE)));
}

/**
 * Shift right, this is signed, while input arg is treated as unsigned
 *
 * @param arg
 * @return this >> arg
 */
public DataWord shiftRightSigned(DataWord arg) {
    if (arg.value().compareTo(BigInteger.valueOf(MAX_POW)) >= 0) {
        if (this.isNegative()) {
            return DataWord.ONE.negate();
        } else {
            return DataWord.ZERO;
        }
    }

    BigInteger result = sValue().shiftRight(arg.intValueSafe());
    return new DataWord(ByteUtil.copyToArray(result.and(MAX_VALUE)));
}

@JsonValue
@Override
public String toString() {
    return sData != null ? sData : toHexString(data);
}
```

```
public String toPrefixString() {

    byte[] pref = getNoLeadZeroesData();
    if (pref.length == 0) {
        return "";
    }

    if (pref.length < 7) {
        return Hex.toHexString(pref);
    }

    return Hex.toHexString(pref).substring(0, 6);
}

public String shortHex() {
    String hexValue = Hex.toHexString(getNoLeadZeroesData()).toUpperCase();
    return "0x" + hexValue.replaceFirst("^0+(?!$)", "");
}

@Override
public boolean equals(Object o) {
    if (this == o) {
        return true;
    }
    if (o == null || getClass() != o.getClass()) {
        return false;
    }

    DataWord that = (DataWord) o;

    return java.util.Arrays.equals(this.data, that.data);
}

@Override
public int hashCode() {
    return java.util.Arrays.hashCode(data);
}

@Override
public int compareTo(DataWord o) {
    if (o == null) {
        return -1;
    }
    int result = FastByteComparisons.compareTo(
            data, 0, data.length,
            o.data, 0, o.data.length);
    // Convert result into -1, 0 or 1 as is the convention
    return (int) Math.signum(result);
}

public DataWord signExtend(byte k) {
    if (0 > k || k > 31) {
        throw new IndexOutOfBoundsException();
    }
    byte mask = this.sValue().testBit((k * 8) + 7) ? (byte) 0xff : 0;
    byte[] newData = this.copyData();
    for (int i = 31; i > k; i--) {
        newData[31 - i] = mask;
    }
    return new DataWord(newData);
}

public int bytesOccupied() {
    int firstNonZero = ByteUtil.firstNonZeroByte(data);
    if (firstNonZero == -1) {
        return 0;
    }
    return 31 - firstNonZero + 1;
}

public boolean isHex(String hex) {
    return Hex.toHexString(data).equals(hex);
}

public String asString() {
    return new String(getNoLeadZeroesData());
}

public DataWord(BigInteger bigInteger) {
    this(bigInteger.toString());
}

public BigInteger toBigInteger() {
    return new BigInteger(asString());
    }

}
```

70:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\test\java\contracts\crowdsale\Crowdsale.java

```
    @View
    public Address getToken() {
        return token;
    }

    @View
    public Address getWallet() {
        return wallet;
    }

    @View
    public BigInteger getRate() {
        return rate;
    }

    @View
    public BigInteger getWeiRaised() {
        return weiRaised;
    }

    class TokenPurchaseEvent implements Event {

        private Address purchaser;

        private Address beneficiary;

        private BigInteger value;

        private BigInteger amount;

        public TokenPurchaseEvent(Address purchaser, Address beneficiary, BigInteger value,
BigInteger amount) {
            this.purchaser = purchaser;
            this.beneficiary = beneficiary;
            this.value = value;
            this.amount = amount;
```

```
}

public Address getPurchaser() {
    return purchaser;
}

public void setPurchaser(Address purchaser) {
    this.purchaser = purchaser;
}

public Address getBeneficiary() {
    return beneficiary;
}

public void setBeneficiary(Address beneficiary) {
    this.beneficiary = beneficiary;
}

public BigInteger getValue() {
    return value;
}

public void setValue(BigInteger value) {
    this.value = value;
}

public BigInteger getAmount() {
    return amount;
}

public void setAmount(BigInteger amount) {
    this.amount = amount;
}

@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;

    TokenPurchaseEvent that = (TokenPurchaseEvent) o;

    if (purchaser != null ? !purchaser.equals(that.purchaser) : that.purchaser != null) return
```

```
false;
            if (beneficiary != null ? !beneficiary.equals(that.beneficiary) : that.beneficiary != null) return
false;
            if (value != null ? !value.equals(that.value) : that.value != null) return false;
            return amount != null ? amount.equals(that.amount) : that.amount == null;
        }

        @Override
        public int hashCode() {
            int result = purchaser != null ? purchaser.hashCode() : 0;
            result = 31 * result + (beneficiary != null ? beneficiary.hashCode() : 0);
            result = 31 * result + (value != null ? value.hashCode() : 0);
            result = 31 * result + (amount != null ? amount.hashCode() : 0);
            return result;
        }

        @Override
        public String toString() {
            return "TokenPurchaseEvent{" +
                    "purchaser=" + purchaser +
                    ", beneficiary=" + beneficiary +
                    ", value=" + value +
                    ", amount=" + amount +
                    '}';
        }

    }

    public Crowdsale(BigInteger rate, Address wallet, Address token) {
        require(rate != null && rate.compareTo(BigInteger.ZERO) > 0);
        require(wallet != null);
        require(token != null);

        this.rate = rate;
        this.wallet = wallet;
        this.token = token;
    }

//    function () external payable {
//        buyTokens(msg.sender);
//    }
```

```
@Payable
public void buyTokens(Address beneficiary) {

    BigInteger weiAmount = Msg.value();
    preValidatePurchase(beneficiary, weiAmount);

    BigInteger tokens = getTokenAmount(weiAmount);

    weiRaised = weiRaised.add(weiAmount);

    processPurchase(beneficiary, tokens);
    emit(new TokenPurchaseEvent(Msg.sender(), beneficiary, weiAmount, tokens));

    updatePurchasingState(beneficiary, weiAmount);

    forwardFunds();
    postValidatePurchase(beneficiary, weiAmount);
}

protected void preValidatePurchase(Address beneficiary, BigInteger weiAmount) {
    require(beneficiary != null);
    require(weiAmount != null && weiAmount.compareTo(BigInteger.ZERO) > 0);
}

protected void postValidatePurchase(Address beneficiary, BigInteger weiAmount) {
    // optional override
}

protected void deliverTokens(Address beneficiary, BigInteger tokenAmount) {
    String[][] args = new String[][]{{beneficiary.toString()}, {tokenAmount.toString()}};
    token.call("transfer", null, args, null);
}

protected void processPurchase(Address beneficiary, BigInteger tokenAmount) {
    deliverTokens(beneficiary, tokenAmount);
}

protected void updatePurchasingState(Address beneficiary, BigInteger weiAmount) {
    // optional override
}

protected BigInteger getTokenAmount(BigInteger weiAmount) {
        return weiAmount.multiply(rate);
    }

    protected void forwardFunds() {
        wallet.transfer(Msg.value());
    }

}
```

71:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\test\java\contracts\crowdsale\distribution\FinalizableCrowdsale.java

```
    }

    public void finalized() {
        onlyOwner();
        require(!isFinalized);
        require(hasClosed());

        finalization();
        emit(new Finalized());

        isFinalized = true;
    }

    protected void finalization() {
    }

    public FinalizableCrowdsale(long openingTime, long closingTime, BigInteger rate, Address wallet, Address token) {
        super(openingTime, closingTime, rate, wallet, token);
    }

}
```

72:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\test\java\contracts\crowdsale\distribution\RefundableCrowdsale.java

```
        return vault;
    }

    public RefundableCrowdsale(long openingTime, long closingTime, BigInteger rate, Address wallet, Address token, BigInteger goal) {
        super(openingTime, closingTime, rate, wallet, token);
        require(goal.compareTo(BigInteger.ZERO) > 0);
        vault = new RefundVault(wallet);
        this.goal = goal;
    }

    public void claimRefund() {
        require(isFinalized());
        require(!goalReached());

        vault.refund(Msg.sender());
    }

    @View
    public boolean goalReached() {
        return getWeiRaised().compareTo(goal) >= 0;
    }

    @Override
    protected void finalization() {
        if (goalReached()) {
            vault.close();
        } else {
            vault.enableRefunds();
        }

        super.finalization();
    }

    @Override
    protected void forwardFunds() {
        vault.deposit(Msg.sender());
    }

}
```

73:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\test\java\contracts\crowdsale\distribution\utils\RefundVault.java

```
    private Address wallet;
    private int state;

    @View
```

```
public BigInteger getDeposited(Address address) {
    BigInteger value = deposited.get(address);
    if (value == null) {
        value = BigInteger.ZERO;
    }
    return value;
}

@View
public Address getWallet() {
    return wallet;
}

@View
public int getState() {
    return state;
}

class Closed implements Event {

}

class RefundsEnabled implements Event {

}

class Refunded implements Event {

    private Address beneficiary;
    private BigInteger weiAmount;

    public Refunded(Address beneficiary, BigInteger weiAmount) {
        this.beneficiary = beneficiary;
        this.weiAmount = weiAmount;
    }

    @Override
    public String toString() {
        return "Refunded{" +
                "beneficiary=" + beneficiary +
                ", weiAmount=" + weiAmount +
                '}';
    }

}

public RefundVault(Address wallet) {
    super();
    this.wallet = wallet;
    this.state = Active;
}

@Payable
public void deposit(Address investor) {
    onlyOwner();
    require(state == Active);
    BigInteger value = deposited.get(investor);
    if (value == null) {
        value = BigInteger.ZERO;
    }
    deposited.put(investor, value.add(Msg.value()));
}

public void close() {
    onlyOwner();
    require(state == Active);
    state = Closed;
    emit(new Closed());
    wallet.transfer(Msg.address().balance());
}

public void enableRefunds() {
    onlyOwner();
    require(state == Active);
    state = Refunding;
    emit(new RefundsEnabled());
}

public void refund(Address investor) {
    require(state == Refunding);
    BigInteger depositedValue = deposited.get(investor);
    if (depositedValue == null) {
        depositedValue = BigInteger.ZERO;
    }
```

```
        deposited.put(investor, BigInteger.ZERO);
        investor.transfer(depositedValue);
        emit(new Refunded(investor, depositedValue));
    }

}
```

74:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\test\java\contracts\crowdsale\emission\MintedCrowdsale.java

```
    public MintedCrowdsale(long openingTime, long closingTime, BigInteger rate, Address wallet,
BigInteger cap, Address token, BigInteger goal) {
        super(openingTime, closingTime, rate, wallet, token, goal);
        this.owner = Msg.sender();
        require(cap.compareTo(BigInteger.ZERO) > 0);
        this.cap = cap;
    }

    public boolean capReached() {
        return getWeiRaised().compareTo(cap) >= 0;
    }

    @Override
    protected void preValidatePurchase(Address beneficiary, BigInteger weiAmount) {
        super.preValidatePurchase(beneficiary, weiAmount);
        require(getWeiRaised().add(weiAmount).compareTo(cap) <= 0);
    }

    @Override
    protected void deliverTokens(Address beneficiary, BigInteger tokenAmount) {
        String[][] args = new String[][]{{beneficiary.toString()}, {tokenAmount.toString()}};
        getToken().call("mint", null, args, null);
    }

    @Override
    @View
    public Address getOwner() {
        return owner;
    }

    @Override
    public void onlyOwner() {
        require(Msg.sender().equals(owner));
    }

    @Override
    public void transferOwnership(Address newOwner) {
        onlyOwner();
        emit(new OwnershipTransferredEvent(owner, newOwner));
        owner = newOwner;
    }

    @Override
    public void renounceOwnership() {
        onlyOwner();
        emit(new OwnershipRenouncedEvent(owner));
        owner = null;
    }

}
```

75:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\test\java\contracts\crowdsale\validation\CappedCrowdsale.java

```
    }

    @View
    public boolean capReached() {
        return getWeiRaised().compareTo(cap) >= 0;
    }

    protected void preValidatePurchase(Address beneficiary, BigInteger weiAmount) {
        super.preValidatePurchase(beneficiary, weiAmount);
        require(getWeiRaised().add(weiAmount).compareTo(cap) <= 0);
    }

}
```

76:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\test\java\contracts\crowdsale\validation\TimedCrowdsale.java

```
        return closingTime;
    }

    public void onlyWhileOpen() {
        require(Block.timestamp() >= openingTime && Block.timestamp() <= closingTime);
    }
    public TimedCrowdsale(long openingTime, long closingTime, BigInteger rate, Address wallet,
Address token) {
        super(rate, wallet, token);
        require(openingTime >= Block.timestamp());
        require(closingTime >= openingTime);

        this.openingTime = openingTime;
        this.closingTime = closingTime;
    }

    @View
    public boolean hasClosed() {
        return Block.timestamp() > closingTime;
    }

    @Override
    protected void preValidatePurchase(Address beneficiary, BigInteger weiAmount) {
        onlyWhileOpen();
        super.preValidatePurchase(beneficiary, weiAmount);
    }

}
```

77:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\test\java\contracts\examples\SampleCrowdsale.java

78:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\test\java\contracts\examples\SimpleToken.java

```
    public String getSymbol() {
        return symbol;
    }

    @View
    public int getDecimals() {
        return decimals;
    }

    @View
    public BigInteger getInitialSupply() {
        return initialSupply;
    }
    public SimpleToken() {
        totalSupply = initialSupply;
        balances.put(Msg.sender(), initialSupply);
        emit(new TransferEvent(null, Msg.sender(), initialSupply));
    }

}
```

79:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\test\java\contracts\examples\TestCrowdsale.java

80:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\test\java\contracts\examples\TestToken.java

```
    @View
    public String getSymbol() {
        return symbol;
    }

    @View
    public int getDecimals() {
        return decimals;
    }

    @View
    public BigInteger getInitialSupply() {
        return initialSupply;
    }

    public TestToken() {
        totalSupply = initialSupply;
        balances.put(Msg.sender(), initialSupply);
        emit(new TransferEvent(null, Msg.sender(), initialSupply));
    }

    @Payable
    @Override
    public void _payable() {

    }
```

```
}

81:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-
vm\src\test\java\contracts\ownership\Ownable.java
        public Address getPreviousOwner() {
            return previousOwner;
        }

        public void setPreviousOwner(Address previousOwner) {
            this.previousOwner = previousOwner;
        }

        @Override
        public boolean equals(Object o) {
            if (this == o) return true;
            if (o == null || getClass() != o.getClass()) return false;

            OwnershipRenouncedEvent that = (OwnershipRenouncedEvent) o;

            return previousOwner != null ? previousOwner.equals(that.previousOwner) :
that.previousOwner == null;
        }

        @Override
        public int hashCode() {
            return previousOwner != null ? previousOwner.hashCode() : 0;
        }

        @Override
        public String toString() {
            return "OwnershipRenouncedEvent{" +
                    "previousOwner=" + previousOwner +
                    '}';
        }

    }

    class OwnershipTransferredEvent implements Event {

        private Address previousOwner;

        private Address newOwner;
        public OwnershipTransferredEvent(Address previousOwner, Address newOwner) {
            this.previousOwner = previousOwner;
            this.newOwner = newOwner;
        }

        public Address getPreviousOwner() {
            return previousOwner;
        }

        public void setPreviousOwner(Address previousOwner) {
            this.previousOwner = previousOwner;
        }

        public Address getNewOwner() {
            return newOwner;
        }

        public void setNewOwner(Address newOwner) {
            this.newOwner = newOwner;
        }

        @Override
        public boolean equals(Object o) {
            if (this == o) return true;
            if (o == null || getClass() != o.getClass()) return false;

            OwnershipTransferredEvent that = (OwnershipTransferredEvent) o;

            if (previousOwner != null ? !previousOwner.equals(that.previousOwner) :
that.previousOwner != null)
            return false;
            return newOwner != null ? newOwner.equals(that.newOwner) : that.newOwner == null;
        }

        @Override
        public int hashCode() {
            int result = previousOwner != null ? previousOwner.hashCode() : 0;
            result = 31 * result + (newOwner != null ? newOwner.hashCode() : 0);
            return result;
        }
```

```
        @Override
        public String toString() {
            return "OwnershipTransferredEvent{" +
                    "previousOwner=" + previousOwner +
                    ", newOwner=" + newOwner +
                    '}';
        }

    }

}
```

82:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\test\java\contracts\ownership\OwnableImpl.java

```
    @Override
    public void onlyOwner() {
        require(Msg.sender().equals(owner));
    }

    @Override
    public void transferOwnership(Address newOwner) {
        onlyOwner();
        emit(new OwnershipTransferredEvent(owner, newOwner));
        owner = newOwner;
    }

    @Override
    public void renounceOwnership() {
        onlyOwner();
        emit(new OwnershipRenouncedEvent(owner));
        owner = null;
    }

}
```

83:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\test\java\contracts\token\ERC20\BasicToken.java

```
    }

    @Override
    @View
    public BigInteger balanceOf(Address owner) {
        require(owner != null);
        BigInteger balance = balances.get(owner);
        if (balance == null) {
            balance = BigInteger.ZERO;
        }
        return balance;
    }

    @Override
    public boolean transfer(Address to, BigInteger value) {
        subtractBalance(Msg.sender(), value);
        addBalance(to, value);
        emit(new TransferEvent(Msg.sender(), to, value));
        return true;
    }

    protected void addBalance(Address address, BigInteger value) {
        BigInteger balance = balanceOf(address);
        check(value);
        check(balance);
        balances.put(address, balance.add(value));
    }

    protected void subtractBalance(Address address, BigInteger value) {
        BigInteger balance = balanceOf(address);
        check(balance, value);
        balances.put(address, balance.subtract(value));
    }

    protected void check(BigInteger value) {
        require(value != null && value.compareTo(BigInteger.ZERO) >= 0);
    }

    protected void check(BigInteger value1, BigInteger value2) {
        check(value1);
        check(value2);
        require(value1.compareTo(value2) >= 0);
    }

}
```

84:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-

```
        private BigInteger value;

        public BurnEvent(Address burner, BigInteger value) {
            this.burner = burner;
            this.value = value;
        }

        public Address getBurner() {
            return burner;
        }

        public void setBurner(Address burner) {
            this.burner = burner;
        }

        public BigInteger getValue() {
            return value;
        }

        public void setValue(BigInteger value) {
            this.value = value;
        }

        @Override
        public boolean equals(Object o) {
            if (this == o) return true;
            if (o == null || getClass() != o.getClass()) return false;

            BurnEvent burnEvent = (BurnEvent) o;

            if (burner != null ? !burner.equals(burnEvent.burner) : burnEvent.burner != null) return
false;
            return value != null ? value.equals(burnEvent.value) : burnEvent.value == null;
        }

        @Override
        public int hashCode() {
            int result = burner != null ? burner.hashCode() : 0;
            result = 31 * result + (value != null ? value.hashCode() : 0);
            return result;
        }

        @Override
        public String toString() {
            return "BurnEvent{" +
                    "burner=" + burner +
                    ", value=" + value +
                    '}';
        }

    }

}
```

85:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\test\java\contracts\token\ERC20\ERC20.java

```
        private BigInteger value;

        public ApprovalEvent(Address owner, Address spender, BigInteger value) {
            this.owner = owner;
            this.spender = spender;
            this.value = value;
        }

        public Address getOwner() {
            return owner;
        }

        public void setOwner(Address owner) {
            this.owner = owner;
        }

        public Address getSpender() {
            return spender;
        }

        public void setSpender(Address spender) {
            this.spender = spender;
        }

        public BigInteger getValue() {
            return value;
        }

        public void setValue(BigInteger value) {
            this.value = value;
        }

        @Override
        public boolean equals(Object o) {
            if (this == o) return true;
            if (o == null || getClass() != o.getClass()) return false;

            ApprovalEvent that = (ApprovalEvent) o;

            if (owner != null ? !owner.equals(that.owner) : that.owner != null) return false;
            if (spender != null ? !spender.equals(that.spender) : that.spender != null) return false;
            return value != null ? value.equals(that.value) : that.value == null;
        }

        @Override
        public int hashCode() {
            int result = owner != null ? owner.hashCode() : 0;
            result = 31 * result + (spender != null ? spender.hashCode() : 0);
            result = 31 * result + (value != null ? value.hashCode() : 0);
            return result;
        }

        @Override
        public String toString() {
            return "ApprovalEvent{" +
                    "owner=" + owner +
                    ", spender=" + spender +
                    ", value=" + value +
                    '}';
        }

    }

}
```

86:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\test\java\contracts\token\ERC20\ERC20Basic.java

```
private BigInteger value;

public TransferEvent(Address from, Address to, BigInteger value) {
    this.from = from;
    this.to = to;
    this.value = value;
}

public Address getFrom() {
    return from;
}

public void setFrom(Address from) {
    this.from = from;
}

public Address getTo() {
    return to;
}

public void setTo(Address to) {
    this.to = to;
}

public BigInteger getValue() {
    return value;
}

public void setValue(BigInteger value) {
    this.value = value;
}

@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;

    TransferEvent that = (TransferEvent) o;

    if (from != null ? !from.equals(that.from) : that.from != null) return false;
    if (to != null ? !to.equals(that.to) : that.to != null) return false;
    return value != null ? value.equals(that.value) : that.value == null;
```

```
        }

        @Override
        public int hashCode() {
            int result = from != null ? from.hashCode() : 0;
            result = 31 * result + (to != null ? to.hashCode() : 0);
            result = 31 * result + (value != null ? value.hashCode() : 0);
            return result;
        }

        @Override
        public String toString() {
            return "TransferEvent{" +
                    "from=" + from +
                    ", to=" + to +
                    ", value=" + value +
                    '}';
        }

    }

}
```

87:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\test\java\contracts\token\ERC20\MintableToken.java

```
    @View
    public boolean isMintingFinished() {
        return mintingFinished;
    }

    @Override
    @View
    public Address getOwner() {
        return owner;
    }

    @Override
    public void onlyOwner() {
        require(Msg.sender().equals(owner));
    }

    @Override
```

```
public void transferOwnership(Address newOwner) {
    onlyOwner();
    emit(new OwnershipTransferredEvent(owner, newOwner));
    owner = newOwner;
}

@Override
public void renounceOwnership() {
    onlyOwner();
    emit(new OwnershipRenouncedEvent(owner));
    owner = null;
}

public void canMint() {
    require(!mintingFinished);
}

public void hasMintPermission() {
    require(Msg.sender().equals(owner));
}

public boolean mint(Address to, BigInteger amount) {
    hasMintPermission();
    canMint();
    check(amount);
    totalSupply = totalSupply.add(amount);
    addBalance(to, amount);
    emit(new MintEvent(to, amount));
    emit(new TransferEvent(null, to, amount));
    return true;
}

public boolean finishMinting() {
    onlyOwner();
    canMint();
    mintingFinished = true;
    emit(new MintFinishedEvent());
    return true;
}

class MintEvent implements Event {
```

```
private Address to;

private BigInteger amount;

public MintEvent(Address to, BigInteger amount) {
    this.to = to;
    this.amount = amount;
}

public Address getTo() {
    return to;
}

public void setTo(Address to) {
    this.to = to;
}

public BigInteger getAmount() {
    return amount;
}

public void setAmount(BigInteger amount) {
    this.amount = amount;
}

@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;

    MintEvent mintEvent = (MintEvent) o;

    if (to != null ? !to.equals(mintEvent.to) : mintEvent.to != null) return false;
    return amount != null ? amount.equals(mintEvent.amount) : mintEvent.amount == null;
}

@Override
public int hashCode() {
    int result = to != null ? to.hashCode() : 0;
    result = 31 * result + (amount != null ? amount.hashCode() : 0);
    return result;
}
```

```
        @Override
        public String toString() {
            return "MintEvent{" +
                    "to=" + to +
                    ", amount=" + amount +
                    '}';
        }

    }

    class MintFinishedEvent implements Event {

    }

}
```

88:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\test\java\contracts\token\ERC20\StandardToken.java

```
        BigInteger value = ownerAllowed.get(spender);
        if (value == null) {
            value = BigInteger.ZERO;
        }
        return value;
    }

    @Override
    public boolean transferFrom(Address from, Address to, BigInteger value) {
        subtractBalance(from, value);
        addBalance(to, value);
        subtractAllowed(from, Msg.sender(), value);
        emit(new TransferEvent(from, to, value));
        return true;
    }

    @Override
    public boolean approve(Address spender, BigInteger value) {
        setAllowed(Msg.sender(), spender, value);
        emit(new ApprovalEvent(Msg.sender(), spender, value));
        return true;
    }
    public boolean increaseApproval(Address spender, BigInteger addedValue) {
        addAllowed(Msg.sender(), spender, addedValue);
        emit(new ApprovalEvent(Msg.sender(), spender, allowance(Msg.sender(), spender)));
        return true;
    }

    public boolean decreaseApproval(Address spender, BigInteger subtractedValue) {
        check(subtractedValue);
        BigInteger oldValue = allowance(Msg.sender(), spender);
        if (subtractedValue.compareTo(oldValue) > 0) {
            setAllowed(Msg.sender(), spender, BigInteger.ZERO);
        } else {
            subtractAllowed(Msg.sender(), spender, subtractedValue);
        }
        emit(new ApprovalEvent(Msg.sender(), spender, allowance(Msg.sender(), spender)));
        return true;
    }

    protected void addAllowed(Address address1, Address address2, BigInteger value) {
        BigInteger allowance = allowance(address1, address2);
        check(allowance);
        check(value);
        setAllowed(address1, address2, allowance.add(value));
    }

    protected void subtractAllowed(Address address1, Address address2, BigInteger value) {
        BigInteger allowance = allowance(address1, address2);
        check(allowance, value);
        setAllowed(address1, address2, allowance.subtract(value));
    }

    protected void setAllowed(Address address1, Address address2, BigInteger value) {
        check(value);
        Map<Address, BigInteger> address1Allowed = allowed.get(address1);
        if (address1Allowed == null) {
            address1Allowed = new HashMap<Address, BigInteger>();
            allowed.put(address1, address1Allowed);
        }
        address1Allowed.put(address2, value);
    }

}
```

```
89:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-
vm\src\test\java\io\nuls\contract\ContractTest.java
 */
package io.nuls.contract;

import io.nuls.contract.util.VMContext;
import io.nuls.contract.vm.natives.io.nuls.contract.sdk.NativeAddress;
import io.nuls.contract.vm.program.*;
import io.nuls.contract.vm.program.impl.ProgramExecutorImpl;
import io.nuls.db.service.DBService;
import io.nuls.db.service.impl.LevelDBServiceImpl;
import org.apache.commons.io.IOUtils;
import org.junit.Before;
import org.junit.Test;
import org.spongycastle.util.encoders.Hex;

import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.math.BigInteger;
import java.util.ArrayList;
import java.util.List;

public class ContractTest {

    private VMContext vmContext;
    private DBService dbService;
    private ProgramExecutor programExecutor;

    private static final String ADDRESS = "TTavpNMqB5XnrzmypowtGaSQ7Gw9u63m";
    private static final String SENDER = "TTaqTVJSPgw3RU9cgjQ5WdhpufmRT343";
    private static final String BUYER = "TTapY7gpBm1DHEgwguSFFtuK3JvGZVKK";

    @Before
    public void setUp() {
        dbService = new LevelDBServiceImpl();
        programExecutor = new ProgramExecutorImpl(vmContext, dbService);
    }

    @Test
    public void testCreate() throws IOException {
        InputStream in = new
FileInputStream(ContractTest.class.getResource("/token_contract").getFile());
        //InputStream in = new FileInputStream(ContractTest.class.getResource("/").getFile() +
"../contract.jar");
        byte[] contractCode = IOUtils.toByteArray(in);

        ProgramCreate programCreate = new ProgramCreate();
        programCreate.setContractAddress(NativeAddress.toBytes(ADDRESS));
        programCreate.setSender(NativeAddress.toBytes(SENDER));
        programCreate.setPrice(1);
        programCreate.setGasLimit(1000000);
        programCreate.setNumber(1);
        programCreate.setContractCode(contractCode);
        //programCreate.args();
        System.out.println(programCreate);

        byte[] prevStateRoot =
Hex.decode("56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc001622fb5e363b421");

        ProgramExecutor track = programExecutor.begin(prevStateRoot);
        ProgramResult programResult = track.create(programCreate);
        track.commit();

        System.out.println(programResult);
        System.out.println("stateRoot: " + Hex.toHexString(track.getRoot()));
        System.out.println();
        sleep();
    }

    @Test
    public void testCall() throws IOException {
        ProgramCall programCall = new ProgramCall();
        programCall.setContractAddress(NativeAddress.toBytes(ADDRESS));
        programCall.setSender(NativeAddress.toBytes(SENDER));
        programCall.setPrice(1);
        programCall.setGasLimit(1000000);
        programCall.setNumber(1);
        programCall.setMethodName("mint");
        programCall.setMethodDesc("");
        programCall.args(BUYER, "1000");
        System.out.println(programCall);
        byte[] prevStateRoot =
Hex.decode("9433b3dd7d6647b57294b72f26ebaf3e614a49647e3fb5599e2d8d6adb14e073");

        ProgramExecutor track = programExecutor.begin(prevStateRoot);
        ProgramResult programResult = track.call(programCall);
        track.commit();

        System.out.println(programResult);
        System.out.println("pierre - stateRoot: " + Hex.toHexString(track.getRoot()));
        System.out.println();

        programCall.setMethodName("balanceOf");
        programCall.setMethodDesc("");
        programCall.args(BUYER);
        System.out.println(programCall);

        track = programExecutor.begin(track.getRoot());
        programResult = track.call(programCall);
        track.commit();

        System.out.println(programResult);
        System.out.println("pierre - stateRoot: " + Hex.toHexString(track.getRoot()));
        System.out.println();
        sleep();
    }

    @Test
    public void testAddBalance() throws IOException {
        ProgramCall programCall = new ProgramCall();
        programCall.setContractAddress(NativeAddress.toBytes(ADDRESS));
        programCall.setSender(NativeAddress.toBytes(SENDER));
        programCall.setPrice(1);
        programCall.setGasLimit(1000000);
        programCall.setNumber(1);
        programCall.setMethodName("_payable");
        programCall.setMethodDesc("()V");
        programCall.setValue(new BigInteger("100"));
        System.out.println(programCall);

        byte[] prevStateRoot =
Hex.decode("fc089b5c5ebd949f308169bff9d963b5905edc3d0d38fd16bb16cd1d1029c73e");
```

```
        ProgramExecutor track = programExecutor.begin(prevStateRoot);
        ProgramResult programResult = track.call(programCall);
        track.commit();

        System.out.println(programResult);
        System.out.println("pierre - stateRoot: " + Hex.toHexString(track.getRoot()));
        System.out.println();
        sleep();
    }

    @Test
    public void testStop() throws IOException {
        byte[] prevStateRoot =
Hex.decode("fc089b5c5ebd949f308169bff9d963b5905edc3d0d38fd16bb16cd1d1029c73e");
        byte[] address = NativeAddress.toBytes(ADDRESS);
        byte[] sender = NativeAddress.toBytes(SENDER);

        ProgramExecutor track = programExecutor.begin(prevStateRoot);
        ProgramResult programResult = track.stop(address, sender);
        track.commit();

        System.out.println(programResult);
        System.out.println("stateRoot: " + Hex.toHexString(track.getRoot()));
        System.out.println();
        sleep();
    }

    @Test
    public void testStatus() throws IOException {
        byte[] prevStateRoot =
Hex.decode("fc089b5c5ebd949f308169bff9d963b5905edc3d0d38fd16bb16cd1d1029c73e");
        byte[] address = NativeAddress.toBytes(ADDRESS);
        byte[] sender = NativeAddress.toBytes(SENDER);

        ProgramExecutor track = programExecutor.begin(prevStateRoot);
        ProgramStatus programStatus = track.status(address);

        System.out.println(programStatus);
        System.out.println("stateRoot: " + Hex.toHexString(track.getRoot()));
        System.out.println();
        sleep();
    }
    @Test
    public void testTransactions() {
        List<ProgramCall> transactions = new ArrayList<>();

        ProgramCall programCall = new ProgramCall();
        programCall.setContractAddress(NativeAddress.toBytes(ADDRESS));
        programCall.setSender(NativeAddress.toBytes(SENDER));
        programCall.setPrice(1);
        programCall.setGasLimit(1000000);
        programCall.setNumber(1);
        programCall.setMethodName("balanceOf");
        programCall.setMethodDesc("");
        programCall.args(ADDRESS);
        System.out.println(programCall);
        transactions.add(programCall);

        ProgramCall programCall1 = new ProgramCall();
        programCall1.setContractAddress(NativeAddress.toBytes(ADDRESS));
        programCall1.setSender(NativeAddress.toBytes(SENDER));
        programCall1.setPrice(0);
        programCall1.setGasLimit(1000000);
        programCall1.setNumber(1);
        programCall1.setMethodName("balanceOf");
        programCall1.setMethodDesc("");
        programCall1.args(ADDRESS);
        System.out.println(programCall1);
        transactions.add(programCall1);

        byte[] prevStateRoot =
Hex.decode("fc089b5c5ebd949f308169bff9d963b5905edc3d0d38fd16bb16cd1d1029c73e");

        for (ProgramCall transaction : transactions) {
            ProgramExecutor track = programExecutor.begin(prevStateRoot);
            ProgramResult programResult = track.call(transaction);
            track.commit();

            prevStateRoot = track.getRoot();

            System.out.println("stateRoot: " + Hex.toHexString(track.getRoot()));
            System.out.println();
        }
        sleep();
    }

    @Test
    public void testMethod() throws IOException {
        byte[] prevStateRoot =
Hex.decode("fc089b5c5ebd949f308169bff9d963b5905edc3d0d38fd16bb16cd1d1029c73e");
        byte[] address = NativeAddress.toBytes(ADDRESS);
        byte[] sender = NativeAddress.toBytes(SENDER);

        ProgramExecutor track = programExecutor.begin(prevStateRoot);
        List<ProgramMethod> methods = track.method(address);
        //track.commit();

        for (ProgramMethod method : methods) {
            System.out.println(method);
        }
        sleep();
    }

    @Test
    public void testJarMethod() throws IOException {
        InputStream in = new
FileInputStream(ContractTest.class.getResource("/token_contract").getFile());
        byte[] contractCode = IOUtils.toByteArray(in);

        List<ProgramMethod> methods = programExecutor.jarMethod(contractCode);

        for (ProgramMethod method : methods) {
            System.out.println(method);
        }
        sleep();
    }

    @Test
    public void testTransfer() throws IOException {
        ProgramCall programCall = new ProgramCall();
        programCall.setContractAddress(NativeAddress.toBytes(ADDRESS));
        programCall.setSender(NativeAddress.toBytes(SENDER));
        programCall.setPrice(1);
        programCall.setGasLimit(1000000);
        programCall.setNumber(1);
        programCall.setMethodName("transfer");
        programCall.setMethodDesc("");
        programCall.args(BUYER, "-1000");
        System.out.println(programCall);

        byte[] prevStateRoot =
Hex.decode("fc089b5c5ebd949f308169bff9d963b5905edc3d0d38fd16bb16cd1d1029c73e");

        ProgramExecutor track = programExecutor.begin(prevStateRoot);
        ProgramResult programResult = track.call(programCall);
        track.commit();

        System.out.println(programResult);
        System.out.println("pierre - stateRoot: " + Hex.toHexString(track.getRoot()));
        System.out.println();

        programCall.setMethodName("balanceOf");
        programCall.setMethodDesc("");
        programCall.args(BUYER);
        System.out.println(programCall);

        track = programExecutor.begin(track.getRoot());
        programResult = track.call(programCall);
        track.commit();

        System.out.println(programResult);
        System.out.println("pierre - stateRoot: " + Hex.toHexString(track.getRoot()));
        System.out.println();
        sleep();
    }

    @Test
    public void testGetter() throws IOException {
        ProgramCall programCall = new ProgramCall();
        programCall.setContractAddress(NativeAddress.toBytes(ADDRESS));
        //programCall.setSender(NativeAddress.toBytes(SENDER));
        programCall.setPrice(1);
        programCall.setGasLimit(1000000);
        programCall.setNumber(1);
        programCall.setMethodName("getName");
        programCall.setMethodDesc("");
        programCall.setValue(new BigInteger("0"));
        System.out.println(programCall);

        byte[] prevStateRoot =
Hex.decode("fc089b5c5ebd949f308169bff9d963b5905edc3d0d38fd16bb16cd1d1029c73e");

        ProgramExecutor track = programExecutor.begin(prevStateRoot);
        ProgramResult programResult = track.call(programCall);
        track.commit();

        System.out.println(programResult);
        System.out.println("pierre - stateRoot: " + Hex.toHexString(track.getRoot()));
        System.out.println();
        sleep();
    }

    @Test
    public void testThread() {
        byte[] prevStateRoot =
Hex.decode("56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc001622fb5e363b421");
        //ProgramExecutor track = programExecutor.begin(prevStateRoot);
        new Thread(new Runnable() {
            @Override
            public void run() {
                ProgramExecutor track = programExecutor.begin(prevStateRoot);
                track.commit();
            }
        }).start();
        try {
            Thread.sleep(10 * 1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    public static void sleep() {
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
```

```
}

90:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-
vm\src\test\java\io\nuls\contract\CrowdsaleTest.java
 */
package io.nuls.contract;

import io.nuls.contract.util.VMContext;
import io.nuls.contract.vm.natives.io.nuls.contract.sdk.NativeAddress;
import io.nuls.contract.vm.program.ProgramCall;
import io.nuls.contract.vm.program.ProgramCreate;
import io.nuls.contract.vm.program.ProgramExecutor;
import io.nuls.contract.vm.program.ProgramResult;
import io.nuls.contract.vm.program.impl.ProgramExecutorImpl;
import io.nuls.db.service.DBService;
import io.nuls.db.service.impl.LevelDBServiceImpl;
import org.apache.commons.io.IOUtils;
import org.junit.Before;
import org.junit.Test;
import org.spongycastle.util.encoders.Hex;

import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.math.BigInteger;

import static io.nuls.contract.ContractTest.sleep;

public class CrowdsaleTest {

    private VMContext vmContext;
    private DBService dbService;
    private ProgramExecutor programExecutor;

    private static final String CROWDSALE_ADDRESS =
"TTaqTVJSPgw3RU9cgjQ5WdhpufmRT343";
    private static final String TOKEN_ADDRESS = "TTavpNMqB5XnrzmypowtGaSQ7Gw9u63m";
    private static final String WALLET_ADDRESS = "TTan6QCd5jeWRLomTyfauAHEQkbWQDTw";
    private static final String SENDER = "TTanAZ7fAK6Y6ziuXGw5pQvqWhHnRQsQ";
    private static final String BUYER = "TTapY7gpBm1DHEgwguSFFtuK3JvGZVKK";

    @Before
    public void setUp() {
        dbService = new LevelDBServiceImpl();
        programExecutor = new ProgramExecutorImpl(vmContext, dbService);
    }

    @Test
    public void testCreate() throws IOException {
        InputStream in = new
FileInputStream(ContractTest.class.getResource("/crowdsale_contract").getFile());
        byte[] contractCode = IOUtils.toByteArray(in);

        ProgramCreate programCreate = new ProgramCreate();
        programCreate.setContractAddress(NativeAddress.toBytes(CROWDSALE_ADDRESS));
        programCreate.setSender(NativeAddress.toBytes(SENDER));
        programCreate.setPrice(1);
        programCreate.setGasLimit(1000000);
        programCreate.setNumber(1);
        programCreate.setContractCode(contractCode);
        programCreate.args("1535012808001", "1635012808001", "10", WALLET_ADDRESS,
"20000000", TOKEN_ADDRESS, "10000000");
        System.out.println(programCreate);

        byte[] prevStateRoot =
Hex.decode("fc089b5c5ebd949f308169bff9d963b5905edc3d0d38fd16bb16cd1d1029c73e");

        ProgramExecutor track = programExecutor.begin(prevStateRoot);
        ProgramResult programResult = track.create(programCreate);
        track.commit();

        System.out.println(programResult);
        System.out.println("stateRoot: " + Hex.toHexString(track.getRoot()));
        System.out.println();
        sleep();
    }

    @Test
    public void testBuyTokens() throws IOException {
        byte[] prevStateRoot =
Hex.decode("87b890ccdbcae7ec892e94f4ad29250ca8ba3b0288515a9a742ee3bb8d7238b4");

        balanceOf(prevStateRoot);
        ProgramCall programCall = new ProgramCall();
        programCall.setContractAddress(NativeAddress.toBytes(CROWDSALE_ADDRESS));
        programCall.setSender(NativeAddress.toBytes(SENDER));
        programCall.setValue(new BigInteger("1000"));
        programCall.setPrice(1);
        programCall.setGasLimit(1000000);
        programCall.setNumber(1);
        programCall.setMethodName("buyTokens");
        programCall.setMethodDesc("");
        programCall.args(BUYER);
        System.out.println(programCall);

        ProgramExecutor track = programExecutor.begin(prevStateRoot);
        ProgramResult programResult = track.call(programCall);
        track.commit();

        System.out.println(programResult);
        System.out.println("stateRoot: " + Hex.toHexString(track.getRoot()));
        System.out.println();

        balanceOf(track.getRoot());

        sleep();
    }

    public void balanceOf(byte[] prevStateRoot) throws IOException {
        ProgramCall programCall = new ProgramCall();
        programCall.setContractAddress(NativeAddress.toBytes(TOKEN_ADDRESS));
        programCall.setSender(NativeAddress.toBytes(SENDER));
        programCall.setPrice(1);
        programCall.setGasLimit(1000000);
        programCall.setNumber(1);
        programCall.setMethodName("balanceOf");
        programCall.setMethodDesc("");
        programCall.args(BUYER);

        ProgramExecutor track = programExecutor.begin(prevStateRoot);
        ProgramResult programResult = track.call(programCall);
        track.commit();

        System.out.println(programResult);
        System.out.println("stateRoot: " + Hex.toHexString(track.getRoot()));
        System.out.println();
        sleep();
    }

}

91:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-
vm\src\test\java\io\nuls\contract\VoteTest.java
 */
package io.nuls.contract;

import io.nuls.contract.util.VMContext;
import io.nuls.contract.vm.natives.io.nuls.contract.sdk.NativeAddress;
import io.nuls.contract.vm.program.ProgramCall;
import io.nuls.contract.vm.program.ProgramCreate;
import io.nuls.contract.vm.program.ProgramExecutor;
import io.nuls.contract.vm.program.ProgramResult;
import io.nuls.contract.vm.program.impl.ProgramExecutorImpl;
import io.nuls.db.service.DBService;
import io.nuls.db.service.impl.LevelDBServiceImpl;
import org.apache.commons.io.IOUtils;
import org.junit.Before;
import org.junit.Test;
import org.spongycastle.util.encoders.Hex;

import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.math.BigInteger;
import java.util.ArrayList;
import java.util.List;

public class VoteTest {

    private VMContext vmContext;
    private DBService dbService;
    private ProgramExecutor programExecutor;

    private static final String ADDRESS = "Nse5j2pZLWHH9iF4GW9Cja2sFyVXQtDX";
    private static final String SENDER = "Nse5gVscugsS8C1S1svh6CMcpoVWewpa";
    private static final String BUYER = "NsdwCuCKs2AXFfUT7PxXXJPm2XxybX6H";
    @Before
    public void setUp() {
        dbService = new LevelDBServiceImpl();
        programExecutor = new ProgramExecutorImpl(vmContext, dbService);
    }

    @Test
    public void testBatchBlock() throws IOException {
        byte[] prevStateRoot =
Hex.decode("56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc001622fb5e363b421");
        List transactions = createTransactions();
        prevStateRoot = testBatch(prevStateRoot, transactions);
        System.out.println("1");
        for (int i = 0; i < 100; i++) {
            transactions = transactions();
            prevStateRoot = testBatch(prevStateRoot, transactions);
            System.out.println("" + (i + 2) + "");
        }
    }

    @Test
    public void testOneBlock() throws IOException {
        byte[] prevStateRoot =
Hex.decode("56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc001622fb5e363b421");
        List transactions = createTransactions();
        prevStateRoot = testOne(prevStateRoot, transactions);
        System.out.println("1");
        for (int i = 0; i < 100; i++) {
            transactions = transactions();
            prevStateRoot = testOne(prevStateRoot, transactions);
            System.out.println("" + (i + 2) + "");
        }
    }

    public byte[] testBatch(byte[] prevStateRoot, List transactions) throws IOException {

        long start = System.currentTimeMillis();

        ProgramExecutor track = programExecutor.begin(prevStateRoot);

        for (int i = 0; i < transactions.size(); i++) {
            long transactionStart = System.currentTimeMillis();
            Object transaction = transactions.get(i);

            if (transaction instanceof ProgramCreate) {

                ProgramCreate create = (ProgramCreate) transaction;
                ProgramExecutor txTrack = track.startTracking();
                ProgramResult programResult = txTrack.create(create);
                txTrack.commit();

                System.out.println(programResult);
            } else if (transaction instanceof ProgramCall) {

                ProgramCall call = (ProgramCall) transaction;
                ProgramExecutor txTrack = track.startTracking();
                ProgramResult programResult = txTrack.call(call);
                txTrack.commit();

                System.out.println(programResult);
            }
            System.out.println("" + i + ", " + (System.currentTimeMillis() - transactionStart) + "ms");
        }

        long time = System.currentTimeMillis() - start;
        System.out.println("" + transactions.size() + " " + time + "ms" + (time / transactions.size()) +
"ms");

        System.out.println("");
        long commitStart = System.currentTimeMillis();
        track.commit();
        System.out.println("" + (System.currentTimeMillis() - commitStart) + "ms");

        byte[] root = track.getRoot();

        System.out.println("stateRoot: " + Hex.toHexString(root));
        System.out.println("" + (System.currentTimeMillis() - start) + "ms");
        return root;
    }

    public byte[] testOne(byte[] prevStateRoot, List transactions) throws IOException {

        long start = System.currentTimeMillis();
        //ProgramExecutor track = programExecutor.begin(prevStateRoot);

        for (int i = 0; i < transactions.size(); i++) {
            long transactionStart = System.currentTimeMillis();
            Object transaction = transactions.get(i);

            if (transaction instanceof ProgramCreate) {

                ProgramCreate create = (ProgramCreate) transaction;
                //ProgramExecutor txTrack = track.startTracking();
                ProgramExecutor track = programExecutor.begin(prevStateRoot);
                ProgramResult programResult = track.create(create);
                track.commit();

                System.out.println(programResult);
                prevStateRoot = track.getRoot();
                System.out.println("stateRoot: " + Hex.toHexString(prevStateRoot));
            } else if (transaction instanceof ProgramCall) {

                ProgramCall call = (ProgramCall) transaction;
                //ProgramExecutor txTrack = track.startTracking();
                ProgramExecutor track = programExecutor.begin(prevStateRoot);
                ProgramResult programResult = track.call(call);
                track.commit();

                System.out.println(programResult);
                prevStateRoot = track.getRoot();
                System.out.println("stateRoot: " + Hex.toHexString(prevStateRoot));
            }
            System.out.println("" + i + ", " + (System.currentTimeMillis() - transactionStart) + "ms");
        }

        long time = System.currentTimeMillis() - start;
        System.out.println("" + transactions.size() + " " + time + "ms" + (time / transactions.size()) +
"ms");

//        System.out.println("");
//        long commitStart = System.currentTimeMillis();
//        track.commit();
//        System.out.println("" + (System.currentTimeMillis() - commitStart) + "ms");

        //System.out.println("stateRoot: " + Hex.toHexString(track.getRoot()));
        //System.out.println("" + (System.currentTimeMillis() - start) + "ms");
        return prevStateRoot;
    }

    public List createTransactions() throws IOException {
        List transactions = new ArrayList();

        ProgramCreate programCreate = new ProgramCreate();
        programCreate.setContractAddress(NativeAddress.toBytes(ADDRESS));
        programCreate.setSender(NativeAddress.toBytes(SENDER));
        programCreate.setPrice(1);
        programCreate.setGasLimit(1000000);
        programCreate.setNumber(1);
        InputStream in = new
FileInputStream(ContractTest.class.getResource("/vote_contract").getFile());
        //InputStream in = new FileInputStream("C:\\workspace\\nuls-
vote\\out\\artifacts\\contract\\contract.jar");
        byte[] contractCode = IOUtils.toByteArray(in);
        programCreate.setContractCode(contractCode);
        programCreate.args("10000");

        transactions.add(programCreate);

        return transactions;
    }

    public List transactions() throws IOException {
        List transactions = new ArrayList();

        for (int i = 0; i < 10; i++) {
            ProgramCall programCall = new ProgramCall();
            programCall.setContractAddress(NativeAddress.toBytes(ADDRESS));
            programCall.setSender(NativeAddress.toBytes(SENDER));
            programCall.setPrice(1);
            programCall.setGasLimit(200000);
            programCall.setNumber(1);
            programCall.setMethodName("create");
            programCall.setMethodDesc("");
            programCall.setValue(new BigInteger("10000"));
            String[][] args = new String[][]{
                    {""}, {""}, {"1", "2", "3"},
                    {"1535012808000"}, {"1545197500000"}, {"true"}, {"3"}, {"true"}
            };
            programCall.setArgs(args);

            transactions.add(programCall);
        }

        ProgramCall programCall = new ProgramCall();
        programCall.setContractAddress(NativeAddress.toBytes(ADDRESS));
        programCall.setSender(NativeAddress.toBytes(SENDER));
        programCall.setPrice(1);
        programCall.setGasLimit(200000);
        programCall.setNumber(1);
        programCall.setMethodName("redemption");
        programCall.setMethodDesc("");
        programCall.setValue(new BigInteger("0"));
        String[][] args = new String[][]{
                {"1"}
        };
        programCall.setArgs(args);

        transactions.add(programCall);

        return transactions;
    }

}
```

92:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\test\java\testcontract\balancecheck\TestBalanceCheck.java

```
package testcontract.balancecheck;

import io.nuls.contract.sdk.Address;
import io.nuls.contract.sdk.Contract;
import io.nuls.contract.sdk.Msg;

import java.math.BigInteger;

/**
 * @author: PierreLuo
 * @date: 2018/7/31
 */
public class TestBalanceCheck implements Contract {
    private String name = "TestBalanceCheck";
    private final String symbol = "TT";
    private final int decimals = 18;

    public String getName() {
        return name;
    }

    public String getSymbol() {
        return symbol;
    }

    public int getDecimals() {
        return decimals;
    }

    public String balance() {
        return Msg.address().balance().toString();
    }

    public String transfer() {
        Address address = new Address("NsdtgQGAxXPh53oZxAWjVpnHnK7mu4wY");
        if(Msg.address().balance().longValue() > 10000000L) {
            address.transfer(BigInteger.valueOf(10000000));
        }
        return Msg.address().balance().toString();
    }

    public TestBalanceCheck() {
        name += " - AW158U";
    }

}
```

93:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\test\java\testcontract\contractcallcontract\ContractCallContract.java

```
package testcontract.contractcallcontract;

import io.nuls.contract.sdk.Address;
import io.nuls.contract.sdk.Contract;
import io.nuls.contract.sdk.Msg;
import io.nuls.contract.sdk.Utils;
import io.nuls.contract.sdk.annotation.Payable;

import java.math.BigInteger;

import static io.nuls.contract.sdk.Utils.require;

/**
 * @author: PierreLuo
 * @date: 2018/10/6
 */
public class ContractCallContract implements Contract {

    @Override
    @Payable
    public void _payable() {

    }

    @Payable
    public String callContract(Address contract, String methodName, String[] args, BigInteger value)
{
        try {
            String[][] args2 = null;
            if(args != null) {
                args2 = new String[args.length][];
                int i = 0;
                for (String arg : args) {
                    args2[i++] = new String[]{arg};
                }
            }
            contract.call(methodName, null, args2, value);
            return "success";
        } catch (Exception e) {
            Utils.revert("exception: " + e.getMessage());
            return e.getMessage();
        }
    }

    //@Payable
    //public String callContractWithReturnValue(Address contract, String methodName, String[]
args, BigInteger value) {
    //    try {
```

```
//        String[][] args2 = null;
//        if(args != null) {
//            args2 = new String[args.length][];
//            int i = 0;
//            for (String arg : args) {
//                args2[i++] = new String[]{arg};
//            }
//        }
//        String returnValue = contract.callWithReturnValue(methodName, null, args2, value);
//        return "success, inner contract call return value: " + returnValue;
//    } catch (Exception e) {
//        Utils.revert("exception: " + e.getMessage());
//        return e.getMessage();
//    }
//}
//
//@Payable
//public String multyForAddressAndcallContractWithReturnValue(Address add1, BigInteger
add1_na, String add3ForString, BigInteger add3_na,
//                                                    Address contract, String methodName, String[] args,
BigInteger value) {
//    try {
//        String multy = multyForAddress(add1, add1_na, add3ForString, add3_na);
//        String[][] args2 = null;
//        if(args != null) {
//            args2 = new String[args.length][];
//            int i = 0;
//            for (String arg : args) {
//                args2[i++] = new String[]{arg};
//            }
//        }
//        String returnValue = contract.callWithReturnValue(methodName, null, args2, value);
//        return "success, multy:" + multy + ", inner contract call return value: " + returnValue;
//    } catch (Exception e) {
//        Utils.revert("exception: " + e.getMessage());
//        return e.getMessage();
//    }
//}

private String multyForAddress(Address add1, BigInteger add1_na, String add3ForString,
BigInteger add3_na) {
    require(add1 != null && add3ForString != null, "Address cannot be empty.");
```

```
        add1.transfer(add1_na);
        Address address_3 = new Address(add3ForString);
        address_3.transfer(add3_na);
        return "multyForAddress: " + Msg.address().balance().toString();
    }
}
```

94:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\test\java\testcontract\incorrectaddress\TestIncorrectAddress.java

```
package testcontract.incorrectaddress;

import io.nuls.contract.sdk.Address;
import io.nuls.contract.sdk.Contract;
import io.nuls.contract.sdk.Msg;

import java.math.BigInteger;

/**
 * @desription:
 * @author: PierreLuo
 * @date: 2018/8/4
 */
public class TestIncorrectAddress implements Contract {

    private String name = "TestIncorrectAddress";

    public String getName() {
        return name;
    }

    public String setName(String name) {
        this.name = name;
        return name;
    }

    public String getBalance() {
        return Msg.address().balance().toString();
    }


    public String multy() {
        Address address_1 = new Address("NsdtgQGAxXPh53oZxAWjVpnHnK7mu4wY");
        address_1.transfer(BigInteger.valueOf(10000001));
        Address address_2 = new Address("Nse5ZkiQs1Q1pdEtRKwXk6Nk6ooYu2L2");
        address_2.transfer(BigInteger.valueOf(20000000));
        // incorrect address
        Address address_3 = new Address("NadtydTVWskMc7GkZzbsq2Fommmmmmmm");
        address_3.transfer(BigInteger.valueOf(30000000));
        return "balance: " + Msg.address().balance().toString();
    }
}
```

95:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\test\java\testcontract\multytransfer\TestMultyTransfer.java

```
package testcontract.multytransfer;

import io.nuls.contract.sdk.*;
import io.nuls.contract.sdk.annotation.Payable;

import java.math.BigInteger;

import static io.nuls.contract.sdk.Utils.require;

/**
 * @desription:
 * @author: PierreLuo
 * @date: 2018/7/26
 */
public class TestMultyTransfer implements Contract {
    private String name = "TestMultyTransfer";
    private final String symbol = "TT";
    private final int decimals = 18;

    @Override
    @Payable
    public void _payable() {

    }

    public TestMultyTransfer() {
        name += " - AW158U";
        allInfo = "";
    }
    public String getName() {
        return name;
    }

    public String getSymbol() {
        return symbol;
    }

    public int getDecimals() {
        return decimals;
    }

    public String balance() {
        return Msg.address().balance().toString();
    }

    public String single() {
        Address address_1 = new Address("NsdtgQGAxXPh53oZxAWjVpnHnK7mu4wY");
        address_1.transfer(BigInteger.valueOf(10000000));
        return "balance: " + Msg.address().balance().toString();
    }

    @Payable
    public String multy() {
        //Utils.revert();
        Address address_1 = new Address("NsdtgQGAxXPh53oZxAWjVpnHnK7mu4wY");
        address_1.transfer(BigInteger.valueOf(10000001));
        //address_1.transfer(BigInteger.valueOf(10000000));
        Address address_2 = new Address("Nse5ZkiQs1Q1pdEtRKwXk6Nk6ooYu2L2");
        address_2.transfer(BigInteger.valueOf(20000000));
        Address address_3 = new Address("NsdtydTVWskMc7GkZzbsq2FoChqKFwMf");
        address_3.transfer(BigInteger.valueOf(30000000));
        //address_3.transfer(BigInteger.valueOf(30000000));
        return "balance: " + Msg.address().balance().toString();
    }

    @Payable
    public String multyForAddress(Address add1, BigInteger add1_na, Address add2, BigInteger
add2_na, String add3ForString, BigInteger add3_na) {
        require(add1 != null && add2 != null && add3ForString != null, "Address cannot be empty.");
        add1.transfer(add1_na);
        add2.transfer(add2_na);
        Address address_3 = new Address(add3ForString);
        address_3.transfer(add3_na);
        return "balance: " + Msg.address().balance().toString();
    }

    private String allInfo;
    public String allInfo() {
        append("\nsender: " + Msg.sender().toString());
        append("\naddress: " + Msg.address().toString());
        append("\nvalue: " + Msg.value());
        append("\ngasleft: " + Msg.gasleft());
        append("\ngasprice: " + Msg.gasprice());
        append("\n\n\n");
        append("\nbestBlockHash: " + Block.blockhash(Block.number()));
        append("\ncoinbase: " + Block.coinbase());
        append("\nheight: " + Block.number());
        append("\ntimestamp: " + Block.timestamp());
        append("\n");
        return allInfo;
    }

    private void append(String appender) {
        allInfo += appender;
    }

}
```

96:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\test\java\testcontract\nrc20\NRC20.java

```
    @View
    BigInteger balanceOf(@Required Address owner);

    boolean transfer(@Required Address to, @Required BigInteger value);

    boolean transferFrom(@Required Address from, @Required Address to, @Required BigInteger
value);

    boolean approve(@Required Address spender, @Required BigInteger value);

    @View
    BigInteger allowance(@Required Address owner, @Required Address spender);
```

```
class TransferEvent implements Event {

    private Address from;

    private Address to;

    private BigInteger value;

    public TransferEvent(Address from, @Required Address to, @Required BigInteger value) {
        this.from = from;
        this.to = to;
        this.value = value;
    }

    public Address getFrom() {
        return from;
    }

    public void setFrom(Address from) {
        this.from = from;
    }

    public Address getTo() {
        return to;
    }

    public void setTo(Address to) {
        this.to = to;
    }

    public BigInteger getValue() {
        return value;
    }

    public void setValue(BigInteger value) {
        this.value = value;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
```

```
            TransferEvent that = (TransferEvent) o;

            if (from != null ? !from.equals(that.from) : that.from != null) return false;
            if (to != null ? !to.equals(that.to) : that.to != null) return false;
            return value != null ? value.equals(that.value) : that.value == null;
        }

        @Override
        public int hashCode() {
            int result = from != null ? from.hashCode() : 0;
            result = 31 * result + (to != null ? to.hashCode() : 0);
            result = 31 * result + (value != null ? value.hashCode() : 0);
            return result;
        }

        @Override
        public String toString() {
            return "TransferEvent{" +
                    "from=" + from +
                    ", to=" + to +
                    ", value=" + value +
                    '}';
        }

    }

    class ApprovalEvent implements Event {

        private Address owner;

        private Address spender;

        private BigInteger value;

        public ApprovalEvent(@Required Address owner, @Required Address spender, @Required
BigInteger value) {
            this.owner = owner;
            this.spender = spender;
            this.value = value;
        }
```

```
public Address getOwner() {
    return owner;
}

public void setOwner(Address owner) {
    this.owner = owner;
}

public Address getSpender() {
    return spender;
}

public void setSpender(Address spender) {
    this.spender = spender;
}

public BigInteger getValue() {
    return value;
}

public void setValue(BigInteger value) {
    this.value = value;
}

@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;

    ApprovalEvent that = (ApprovalEvent) o;

    if (owner != null ? !owner.equals(that.owner) : that.owner != null) return false;
    if (spender != null ? !spender.equals(that.spender) : that.spender != null) return false;
    return value != null ? value.equals(that.value) : that.value == null;
}

@Override
public int hashCode() {
    int result = owner != null ? owner.hashCode() : 0;
    result = 31 * result + (spender != null ? spender.hashCode() : 0);
    result = 31 * result + (value != null ? value.hashCode() : 0);
    return result;
```

```
        }

        @Override
        public String toString() {
            return "ApprovalEvent{" +
                    "owner=" + owner +
                    ", spender=" + spender +
                    ", value=" + value +
                    '}';
        }

    }

}
```

97:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\test\java\testcontract\nrc20\SimpleToken.java

```
    private Map<Address, Map<Address, BigInteger>> allowed = new HashMap<Address,
Map<Address, BigInteger>>();

    @Override
    @View
    public String name() {
        return name;
    }

    @Override
    @View
    public String symbol() {
        return symbol;
    }

    @Override
    @View
    public int decimals() {
        return decimals;
    }

    @Override
    @View
    public BigInteger totalSupply() {
        return totalSupply;
    }

    public SimpleToken(@Required String name, @Required String symbol, @Required BigInteger
initialAmount, @Required int decimals) {
        this.name = name;
        this.symbol = symbol;
        this.decimals = decimals;
        totalSupply = initialAmount.multiply(BigInteger.TEN.pow(decimals));;
        balances.put(Msg.sender(), totalSupply);
        emit(new TransferEvent(null, Msg.sender(), totalSupply));
    }

    @Override
    @View
    public BigInteger allowance(@Required Address owner, @Required Address spender) {
        Map<Address, BigInteger> ownerAllowed = allowed.get(owner);
        if (ownerAllowed == null) {
            return BigInteger.ZERO;
        }
        BigInteger value = ownerAllowed.get(spender);
        if (value == null) {
            value = BigInteger.ZERO;
        }
        return value;
    }

    @Override
    public boolean transferFrom(@Required Address from, @Required Address to, @Required
BigInteger value) {
        subtractAllowed(from, Msg.sender(), value);
        subtractBalance(from, value);
        addBalance(to, value);
        emit(new TransferEvent(from, to, value));
        return true;
    }

    @Override
    @View
    public BigInteger balanceOf(@Required Address owner) {
        require(owner != null);
        BigInteger balance = balances.get(owner);
        if (balance == null) {
            balance = BigInteger.ZERO;
        }
        return balance;
    }

    @Override
    public boolean transfer(@Required Address to, @Required BigInteger value) {
        subtractBalance(Msg.sender(), value);
        addBalance(to, value);
        emit(new TransferEvent(Msg.sender(), to, value));
        return true;
    }

    @Override
    public boolean approve(@Required Address spender, @Required BigInteger value) {
        setAllowed(Msg.sender(), spender, value);
        emit(new ApprovalEvent(Msg.sender(), spender, value));
        return true;
    }

    public boolean increaseApproval(@Required Address spender, @Required BigInteger
addedValue) {
        addAllowed(Msg.sender(), spender, addedValue);
        emit(new ApprovalEvent(Msg.sender(), spender, allowance(Msg.sender(), spender)));
        return true;
    }

    public boolean decreaseApproval(@Required Address spender, @Required BigInteger
subtractedValue) {
        check(subtractedValue);
        BigInteger oldValue = allowance(Msg.sender(), spender);
        if (subtractedValue.compareTo(oldValue) > 0) {
            setAllowed(Msg.sender(), spender, BigInteger.ZERO);
        } else {
            subtractAllowed(Msg.sender(), spender, subtractedValue);
        }
        emit(new ApprovalEvent(Msg.sender(), spender, allowance(Msg.sender(), spender)));
        return true;
    }

    private void addAllowed(Address address1, Address address2, BigInteger value) {
        BigInteger allowance = allowance(address1, address2);
        check(allowance);
        check(value);
        setAllowed(address1, address2, allowance.add(value));
    }

    private void subtractAllowed(Address address1, Address address2, BigInteger value) {
        BigInteger allowance = allowance(address1, address2);
        check(allowance, value, "Insufficient approved token");
        setAllowed(address1, address2, allowance.subtract(value));
    }

    private void setAllowed(Address address1, Address address2, BigInteger value) {
        check(value);
        Map<Address, BigInteger> address1Allowed = allowed.get(address1);
        if (address1Allowed == null) {
            address1Allowed = new HashMap<Address, BigInteger>();
            allowed.put(address1, address1Allowed);
        }
        address1Allowed.put(address2, value);
    }

    private void addBalance(Address address, BigInteger value) {
        BigInteger balance = balanceOf(address);
        check(value, "The value must be greater than or equal to 0.");
        check(balance);
        balances.put(address, balance.add(value));
    }

    private void subtractBalance(Address address, BigInteger value) {
        BigInteger balance = balanceOf(address);
        check(balance, value, "Insufficient balance of token.");
        balances.put(address, balance.subtract(value));
    }

    private void check(BigInteger value) {
        require(value != null && value.compareTo(BigInteger.ZERO) >= 0);
    }

    private void check(BigInteger value1, BigInteger value2) {
        check(value1);
        check(value2);
        require(value1.compareTo(value2) >= 0);
    }

    private void check(BigInteger value, String msg) {
        require(value != null && value.compareTo(BigInteger.ZERO) >= 0, msg);
    }

    private void check(BigInteger value1, BigInteger value2, String msg) {
        check(value1);
        check(value2);
        require(value1.compareTo(value2) >= 0, msg);
    }



}
```

98:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\test\java\testcontract\simple\ContractCodeHexString.java

```
package testcontract.simple;

import org.apache.commons.io.IOUtils;
import org.spongycastle.util.encoders.Hex;

import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;

/**
 * @desription:
 * @author: PierreLuo
 * @date: 2018/7/4
 */
public class ContractCodeHexString {
    public static void main(String[] args) throws IOException {
        InputStream in = new
FileInputStream(ContractCodeHexString.class.getResource("/contract.jar").getFile());
        byte[] contractCode = IOUtils.toByteArray(in);
        System.out.println(Hex.toHexString(contractCode));
    }
}
```

99:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-

vm\src\test\java\testcontract\simple\TestContract.java

```java
package testcontract.simple;

import io.nuls.contract.sdk.Address;
import io.nuls.contract.sdk.Contract;
import io.nuls.contract.sdk.Msg;
import io.nuls.contract.sdk.Utils;
import io.nuls.contract.sdk.annotation.View;

import java.math.BigInteger;
import java.util.HashMap;
import java.util.Map;

/**
 * @desription:
 * @author: PierreLuo
 * @date: 2018/7/4
 */
public class TestContract implements Contract {
    private String name = "TestToken";
    private final String symbol = "TT";
    private final int decimals = 18;

    private Map<String, String> map = new HashMap<>();

    @View
    public String getName() {
        return name;
    }

    public String getSymbol() {
        return symbol;
    }

    public int getDecimals() {
        return decimals;
    }

    public String balance() {
        return Msg.address().balance().toString();
    }
```

```
public String transfer() {
    Address address = new Address("NsdtgQGAxXPh53oZxAWjVpnHnK7mu4wY");
    address.transfer(BigInteger.valueOf(10000000));
    return "AW158U";
}

//@View
//public int randomNumberInt() {
//    return Utils.pseudoRandom(1, 16);
//}
//
//@View
//public int randomNumberString() {
//    return Utils.pseudoRandom("a", 16);
//}
//
//@View
//public int randomNumberWithIntAndCap(int cap) {
//    return Utils.pseudoRandom(1, cap);
//}
//
//@View
//public int randomNumberWithStringAndCap(int cap) {
//    return Utils.pseudoRandom("a", cap);
//}

public TestContract() {
    name += " - AW158U";
    map.put("123", "123a");
    map.put("124", "124a");
    map.put("125", "125a");
    map.put("126", "126a");
    map.put("127", "127a");
}

//public String setName(String name, int cap) {
//    this.name += name + Utils.pseudoRandom("a", cap);
//    return this.name;
//}

@View
public String map() {
        String result = "";
        //for(Map.Entry<String, String> entry : map.entrySet()) {
        //    result += entry.getKey() + ": " + entry.getValue() + " \n";
        //}
        for(String key : map.keySet()) {
            result += key + ": " + map.get(key) + " \n";
        }
        return result;
    }

    @View
    public String asd(String b, String[] a) {
        String result = "";
        if(a != null) {
            for(String aaa : a) {
                result += aaa;
            }
        }
        return result + b;
    }

    @View
    public String qwe(String b, String a, String c) {
        String result = "";
        return result + b + a +c;
    }

}
```

100:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\base\contract-vm\src\test\java\testcontract\testlotoftransfer\TestLotOfTransfer.java

```
package testcontract.testlotoftransfer;

import io.nuls.contract.sdk.Address;
import io.nuls.contract.sdk.Contract;
import io.nuls.contract.sdk.Msg;

import java.math.BigInteger;

/**
 * @desription:
 * @author: PierreLuo
 * @date: 2018/8/3
 */
public class TestLotOfTransfer implements Contract {
    private String name = "TestLotOfTransfer";

    public String getName() {
        return name;
    }

    public String getBalance() {
        return Msg.address().balance().toString();
    }

    public String transfer(int loop) {
        for (int i = 0; i < loop; i++) {
            Address address = new Address("NsdtgQGAxXPh53oZxAWjVpnHnK7mu4wY");
            address.transfer(BigInteger.valueOf(10000));
        }
        return name;
    }

    public String setName(String name) {
        this.name = name;
        return name;
    }
}
```

101:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\contract\src\main\java\io\nuls\contract\constant\ContractConstant.java

```
package io.nuls.contract.constant;

import io.nuls.kernel.constant.NulsConstant;

public interface ContractConstant extends NulsConstant {

    short MODULE_ID_CONTRACT = 10;

    /**
     * CONTRACT
     */
    int TX_TYPE_CREATE_CONTRACT = 100;
    int TX_TYPE_CALL_CONTRACT = 101;
    int TX_TYPE_DELETE_CONTRACT = 102;

    /**
     * CONTRACT STATUS
     */
    int NOT_FOUND = 0;

    int NORMAL = 1;

    int STOP = 2;

    /**
     * contract transfer
     */
    int TX_TYPE_CONTRACT_TRANSFER = 103;
    long CONTRACT_TRANSFER_GAS_COST = 1000;

    String BALANCE_TRIGGER_METHOD_NAME = "_payable";
    String BALANCE_TRIGGER_METHOD_DESC = "() return void";

    String CONTRACT_CONSTRUCTOR = "<init>";




    String CALL = "call";
    String CREATE = "create";
    String DELETE = "delete";

    String GET = "get";

    String SEND_BACK_REMARK = "Contract execution failed, return funds.";

    long CONTRACT_CONSTANT_GASLIMIT = 10000000;
    long CONTRACT_CONSTANT_PRICE = 1;

    long MAX_GASLIMIT = 10000000;

    long CONTRACT_MINIMUM_PRICE = 25;

    int MAX_PACKAGE_GAS = 5000000;
```

```
    /**
     *
     */
    String CONTRACT_EVENT = "event";
    String CONTRACT_EVENT_ADDRESS = "contractAddress";
    String CONTRACT_EVENT_DATA = "payload";

    /**
     * NRC20
     */
    String NRC20_METHOD_NAME = "name";
    String NRC20_METHOD_SYMBOL = "symbol";
    String NRC20_METHOD_DECIMALS = "decimals";
    String NRC20_METHOD_TOTAL_SUPPLY = "totalSupply";
    String NRC20_METHOD_BALANCE_OF = "balanceOf";
    String NRC20_METHOD_TRANSFER = "transfer";
    String NRC20_METHOD_TRANSFER_FROM = "transferFrom";
    String NRC20_METHOD_APPROVE = "approve";
    String NRC20_METHOD_ALLOWANCE = "allowance";
    String NRC20_EVENT_TRANSFER = "TransferEvent";
    String NRC20_EVENT_APPROVAL = "ApprovalEvent";

}

102:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-
module\contract\src\main\java\io\nuls\contract\constant\ContractErrorCode.java
 */
package io.nuls.contract.constant;

import io.nuls.kernel.constant.ErrorCode;
import io.nuls.kernel.constant.KernelErrorCode;
import io.nuls.kernel.constant.TransactionErrorCode;

/**
 * @desription:
 * @author: PierreLuo
 * @date: 2018/6/17
 */
public interface ContractErrorCode extends TransactionErrorCode, KernelErrorCode {

    ErrorCode CONTRACT_EXECUTE_ERROR = ErrorCode.init("100001");
    ErrorCode CONTRACT_ADDRESS_NOT_EXIST = ErrorCode.init("100002");

    ErrorCode CONTRACT_TX_CREATE_ERROR = ErrorCode.init("100003");

    ErrorCode ILLEGAL_CONTRACT_ADDRESS = ErrorCode.init("100004");

    ErrorCode NON_CONTRACTUAL_TRANSACTION = ErrorCode.init("100005");

    ErrorCode NON_CONTRACTUAL_TRANSACTION_NO_TRANSFER = 
ErrorCode.init("100006");

    ErrorCode CONTRACT_NAME_FORMAT_INCORRECT = ErrorCode.init("100007");

    ErrorCode CONTRACT_NOT_NRC20 = ErrorCode.init("100008");

    ErrorCode CONTRACT_NON_VIEW_METHOD = ErrorCode.init("100009");

    ErrorCode ILLEGAL_CONTRACT = ErrorCode.init("100010");

    ErrorCode CONTRACT_DUPLICATE_TOKEN_NAME = ErrorCode.init("100011");

    ErrorCode CONTRACT_NRC20_SYMBOL_FORMAT_INCORRECT = 
ErrorCode.init("100012");

    ErrorCode CONTRACT_LOCK = ErrorCode.init("100013");

    ErrorCode CONTRACT_NRC20_MAXIMUM_DECIMALS = ErrorCode.init("100014");

    ErrorCode CONTRACT_NRC20_MAXIMUM_TOTAL_SUPPLY = ErrorCode.init("100015");

    ErrorCode CONTRACT_MINIMUM_PRICE = ErrorCode.init("100016");

    ErrorCode CONTRACT_DELETE_BALANCE = ErrorCode.init("100017");

    ErrorCode CONTRACT_DELETE_CREATER = ErrorCode.init("100018");

    ErrorCode CONTRACT_DELETED = ErrorCode.init("100019");
}
```

103:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\contract\src\main\java\io\nuls\contract\dto\ContractResult.java

```
 */
```

```
package io.nuls.contract.dto;

import com.fasterxml.jackson.annotation.JsonIgnore;
import io.nuls.core.tools.crypto.Hex;
import io.nuls.kernel.utils.AddressTool;

import java.math.BigInteger;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class ContractResult {

    /**
     *
     */
    private byte[] sender;

    /**
     *
     */
    private byte[] contractAddress;

    /**
     *
     */
    private String result;
    /**
     * Gas
     */
    private long gasUsed;
    /**
     *
     */
    private long price;
    /**
     *
     */
    private byte[] stateRoot;

    /**
```

```
 *
 */
private long value;

/**
 *
 */
private boolean revert;

/**
 *
 */
private boolean error;

/**
 *
 */
private String errorMessage;

/**
 *
 */
private String stackTrace;

/**
 *
 */
private BigInteger balance;

private BigInteger preBalance;

/**
 *
 */
private BigInteger nonce;

private boolean acceptDirectTransfer;

private boolean isNrc20;

/**
 * ()
 */
private List<ContractTransfer> transfers = new ArrayList<>();

/**
 *
 */
private List<String> events = new ArrayList<>();

private String remark;

private transient Object txTrack;

@JsonIgnore
public Object getTxTrack() {
    return txTrack;
}

public void setTxTrack(Object txTrack) {
    this.txTrack = txTrack;
}

public byte[] getSender() {
    return sender;
}

public void setSender(byte[] sender) {
    this.sender = sender;
}

public boolean isSuccess() {
    return !error && !revert;
}

public byte[] getContractAddress() {
    return contractAddress;
}

public void setContractAddress(byte[] contractAddress) {
    this.contractAddress = contractAddress;
}

public String getResult() {
    return result;
}

public void setResult(String result) {
    this.result = result;
}

public byte[] getStateRoot() {
    return stateRoot;
}

public void setStateRoot(byte[] stateRoot) {
    this.stateRoot = stateRoot;
}

public long getGasUsed() {
    return gasUsed;
}

public void setGasUsed(long gasUsed) {
    this.gasUsed = gasUsed;
}

public long getPrice() {
    return price;
}

public void setPrice(long price) {
    this.price = price;
}

public List<String> getEvents() {
    return events;
}

public void setEvents(List<String> events) {
    this.events = events;
}

public List<ContractTransfer> getTransfers() {
    return transfers;
}
```

```
public void setTransfers(List<ContractTransfer> transfers) {
    this.transfers = transfers;
}

public boolean isRevert() {
    return revert;
}

public void setRevert(boolean revert) {
    this.revert = revert;
}

public boolean isError() {
    return error;
}

public void setError(boolean error) {
    this.error = error;
}

public String getErrorMessage() {
    return errorMessage;
}

public void setErrorMessage(String errorMessage) {
    this.errorMessage = errorMessage;
}

public String getStackTrace() {
    return stackTrace;
}

public void setStackTrace(String stackTrace) {
    this.stackTrace = stackTrace;
}

public BigInteger getBalance() {
    return balance;
}

public void setBalance(BigInteger balance) {
    this.balance = balance;
}

public BigInteger getPreBalance() {
    return preBalance;
}

public void setPreBalance(BigInteger preBalance) {
    this.preBalance = preBalance;
}

public BigInteger getNonce() {
    return nonce;
}

public void setNonce(BigInteger nonce) {
    this.nonce = nonce;
}

public long getValue() {
    return value;
}

public void setValue(long value) {
    this.value = value;
}

public String getRemark() {
    return remark;
}

public void setRemark(String remark) {
    this.remark = remark;
}

public boolean isAcceptDirectTransfer() {
    return acceptDirectTransfer;
}

public void setAcceptDirectTransfer(boolean acceptDirectTransfer) {
    this.acceptDirectTransfer = acceptDirectTransfer;
}
```

```
    public boolean isNrc20() {
        return isNrc20;
    }

    public void setNrc20(boolean nrc20) {
        isNrc20 = nrc20;
    }

    @Override
    public String toString() {
        return "ContractResult{" +
                "contractAddress=" + AddressTool.getStringAddressByBytes(contractAddress) +
                ", result='" + result + '\'' +
                ", gasUsed=" + gasUsed +
                ", stateRoot=" + (stateRoot != null ? Hex.encode(stateRoot) : stateRoot ) +
                ", value=" + value +
                ", revert=" + revert +
                ", error=" + error +
                ", errorMessage='" + errorMessage + '\'' +
                ", stackTrace='" + stackTrace + '\'' +
                ", balance=" + (balance != null ? balance.toString() : null) +
                ", nonce=" + nonce +
                ", transfersSize=" + (transfers != null ? transfers.size() : 0) +
                ", eventsSize=" + (events != null ? events.size() : 0) +
                ", remark='" + remark + '\'' +
                '}';
    }
}
```

104:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\contract\src\main\java\io\nuls\contract\dto\ContractTokenInfo.java

```
package io.nuls.contract.dto;

import io.nuls.contract.util.ContractUtil;

import java.math.BigInteger;

/**
 * @desription:
 * @author: PierreLuo
 * @date: 2018/8/19
 */
public class ContractTokenInfo {

    private String contractAddress;
    private String name;
    private String symbol;
    private BigInteger amount;
    private long decimals;
    private long blockHeight;
    private int status;

    public ContractTokenInfo() {
    }

    public ContractTokenInfo(String contractAddress, String name, long decimals, BigInteger
amount, String symbol, long blockHeight) {
        this.name = name;
        this.amount = amount;
        this.contractAddress = contractAddress;
        this.decimals = decimals;
        this.symbol = symbol;
        this.blockHeight = blockHeight;
    }

    public String getSymbol() {
        return symbol;
    }

    public void setSymbol(String symbol) {
        this.symbol = symbol;
    }

    public String getContractAddress() {
        return contractAddress;
    }

    public void setContractAddress(String contractAddress) {
        this.contractAddress = contractAddress;
    }

    public String getName() {
        return name;
```

```
}

public ContractTokenInfo setName(String name) {
    this.name = name;
    return this;
}

public BigInteger getAmount() {
    return amount;
}

public ContractTokenInfo setAmount(BigInteger amount) {
    this.amount = amount;
    return this;
}

public long getDecimals() {
    return decimals;
}

public void setDecimals(long decimals) {
    this.decimals = decimals;
}

public long getBlockHeight() {
    return blockHeight;
}

public void setBlockHeight(long blockHeight) {
    this.blockHeight = blockHeight;
}

public boolean isLock() {
    return ContractUtil.isLockContract(this.blockHeight);
}

public boolean isStop() {
    return ContractUtil.isTerminatedContract(this.status);
}

public int getStatus() {
    return status;
    }

    public void setStatus(int status) {
        this.status = status;
    }
}

105:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-
module\contract\src\main\java\io\nuls\contract\dto\ContractTokenTransferInfoPo.java
 */

package io.nuls.contract.dto;

import java.io.Serializable;
import java.math.BigDecimal;
import java.math.BigInteger;
import java.util.Arrays;

/**
 * @author: PierreLuo
 * @date: 2018/7/23
 */
public class ContractTokenTransferInfoPo implements Serializable {

    private byte[] from;
    private byte[] to;
    private BigInteger value;
    private String contractAddress;
    private String name;
    private String symbol;
    private long decimals;
    private long time;
    private byte status;
    private byte[] txHash;
    private long blockHeight;

    public byte[] getFrom() {
        return from;
    }

    public void setFrom(byte[] from) {
        this.from = from;
```

```
}

public byte[] getTo() {
    return to;
}

public void setTo(byte[] to) {
    this.to = to;
}

public BigInteger getValue() {
    return value;
}

public void setValue(BigInteger value) {
    this.value = value;
}

public String getContractAddress() {
    return contractAddress;
}

public void setContractAddress(String contractAddress) {
    this.contractAddress = contractAddress;
}

public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}

public String getSymbol() {
    return symbol;
}

public void setSymbol(String symbol) {
    this.symbol = symbol;
}
```

```
public long getDecimals() {
    return decimals;
}

public void setDecimals(long decimals) {
    this.decimals = decimals;
}

public long getTime() {
    return time;
}

public void setTime(long time) {
    this.time = time;
}

public byte getStatus() {
    return status;
}

public void setStatus(byte status) {
    this.status = status;
}

public byte[] getTxHash() {
    return txHash;
}

public void setTxHash(byte[] txHash) {
    this.txHash = txHash;
}

public long getBlockHeight() {
    return blockHeight;
}

public void setBlockHeight(long blockHeight) {
    this.blockHeight = blockHeight;
}

public String getInfo(byte[] address) {
    BigDecimal result = BigDecimal.ZERO;
```

```
        if(this.status == 2) {
            return result.toPlainString();
        }

        if(Arrays.equals(from, address)) {
            result = result.subtract(new BigDecimal(value).divide(BigDecimal.TEN.pow((int)
decimals)));
        }
        if(Arrays.equals(to, address)) {
            result = result.add(new BigDecimal(value).divide(BigDecimal.TEN.pow((int) decimals)));
        }
        if(result.compareTo(BigDecimal.ZERO) > 0) {
            return "+" + result.toPlainString();
        } else {
            return result.toPlainString();
        }
    }

}
```

106:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\contract\src\main\java\io\nuls\contract\dto\ContractTransfer.java

```
package io.nuls.contract.dto;

import io.nuls.kernel.model.Na;
import io.nuls.kernel.model.NulsDigestData;
import io.nuls.kernel.utils.AddressTool;

/**
 * @author: PierreLuo
 */
public class ContractTransfer {

    private byte[] from;

    private byte[] to;

    private Na value;

    private Na fee;

    private boolean isSendBack;
```

```
/**
 *  hash
 */
private NulsDigestData orginHash;

/**
 *  ()hash
 */
private NulsDigestData hash;

public ContractTransfer(){

}

public ContractTransfer(byte[] from, byte[] to, Na value, Na fee) {
    this.from = from;
    this.to = to;
    this.value = value;
    this.fee = fee;
    this.isSendBack = false;
}

public ContractTransfer(byte[] from, byte[] to, Na value, Na fee, boolean isSendBack) {
    this.from = from;
    this.to = to;
    this.value = value;
    this.fee = fee;
    this.isSendBack = isSendBack;
}

public byte[] getFrom() {
    return from;
}

public void setFrom(byte[] from) {
    this.from = from;
}

public byte[] getTo() {
    return to;
}
```

```
public void setTo(byte[] to) {
    this.to = to;
}

public Na getValue() {
    return value;
}

public void setValue(Na value) {
    this.value = value;
}

public Na getFee() {
    return fee;
}

public void setFee(Na fee) {
    this.fee = fee;
}

public boolean isSendBack() {
    return isSendBack;
}

public void setSendBack(boolean sendBack) {
    isSendBack = sendBack;
}

public NulsDigestData getOrginHash() {
    return orginHash;
}

public void setOrginHash(NulsDigestData orginHash) {
    this.orginHash = orginHash;
}

public NulsDigestData getHash() {
    return hash;
}

public void setHash(NulsDigestData hash) {
        this.hash = hash;
    }

}
```

107:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\contract\src\main\java\io\nuls\contract\entity\tx\CallContractTransaction.java

```
package io.nuls.contract.entity.tx;

import io.nuls.contract.constant.ContractConstant;
import io.nuls.contract.dto.ContractResult;
import io.nuls.contract.dto.ContractTransfer;
import io.nuls.contract.entity.txdata.CallContractData;
import io.nuls.contract.service.ContractService;
import io.nuls.core.tools.json.JSONUtils;
import io.nuls.core.tools.log.Log;
import io.nuls.core.tools.map.MapUtil;
import io.nuls.kernel.context.NulsContext;
import io.nuls.kernel.exception.NulsException;
import io.nuls.kernel.model.BlockHeader;
import io.nuls.kernel.model.Coin;
import io.nuls.kernel.model.Na;
import io.nuls.kernel.model.Transaction;
import io.nuls.kernel.utils.AddressTool;
import io.nuls.kernel.utils.ByteArrayWrapper;
import io.nuls.kernel.utils.NulsByteBuffer;

import java.util.*;

import static io.nuls.contract.constant.ContractConstant.CONTRACT_EVENT_ADDRESS;

/**
 * @Desription:
 * @Author: PierreLuo
 * @Date: 2018/4/20
 */
public class CallContractTransaction extends Transaction<CallContractData> implements
ContractTransaction{

    private transient ContractService contractService =
NulsContext.getServiceBean(ContractService.class);
```

```
/**
 *
 */
private ContractResult contractResult;

private transient Collection<ContractTransferTransaction> contractTransferTxs;

private transient Na returnNa;

private transient BlockHeader blockHeader;

public CallContractTransaction() {
    super(ContractConstant.TX_TYPE_CALL_CONTRACT);
}

@Override
protected CallContractData parseTxData(NulsByteBuffer byteBuffer) throws NulsException {
    return byteBuffer.readNulsData(new CallContractData());
}

/**
 *
 *
 * : 1.  2.
 * `getInfo`
 * toListCoin
 * toListCoin -  -
 *      ()`to`Token`from`,`to`
 *       `0`
 * @param address
 * @return
 */
@Override
public String getInfo(byte[] address) {
    List<Coin> toList = coinData.getTo();
    int size = toList.size();
    if(size == 1) {
        if (Arrays.equals(address, toList.get(0).getAddress())) {
            return "-" + getFee().toCoinString();
        } else {
            return "0";
        }
    } else if(size == 2) {
        Coin to1 = toList.get(0);
        Coin to2 = toList.get(1);
        boolean equals1 = Arrays.equals(address, to1.getAddress());
        boolean equals2 = Arrays.equals(address, to2.getAddress());
        if (!equals1 && !equals2) {
            return "0";
        } else if (!equals1) {
            return "-" + to1.getNa().add(getFee()).toCoinString();
        } else if (!equals2){
            return "-" + to2.getNa().add(getFee()).toCoinString();
        } else {
            return "--";
        }
    } else {
        return "--";
    }
}

@Override
public ContractResult getContractResult() {
    return contractResult;
}

@Override
public void setContractResult(ContractResult contractResult) {
    this.contractResult = contractResult;
}

@Override
public void setReturnNa(Na returnNa) {
    this.returnNa = returnNa;
}

@Override
public Na getFee() {
    Na resultFee = super.getFee();
    if(returnNa != null) {
        resultFee = resultFee.minus(returnNa);
    }
    return resultFee;
}
```

```
    @Override
    public BlockHeader getBlockHeader() {
        return blockHeader;
    }

    @Override
    public void setBlockHeader(BlockHeader blockHeader) {
        this.blockHeader = blockHeader;
    }

    @Override
    public List<byte[]> getAllRelativeAddress() {
        if(contractResult == null && contractService != null) {
            contractResult = contractService.getContractExecuteResult(this.hash);
        }
        if(contractResult != null) {
            List<byte[]> relativeAddress = super.getAllRelativeAddress();
            if(relativeAddress == null) {
                return new ArrayList<>();
            }
            if(relativeAddress.size() == 0) {
                return relativeAddress;
            }
            HashSet<ByteArrayWrapper> addressesSet =
MapUtil.createHashSet(relativeAddress.size());
            relativeAddress.stream().forEach(address -> addressesSet.add(new
ByteArrayWrapper(address)));

            // ()
            List<ContractTransfer> transfers = contractResult.getTransfers();
            if(transfers != null && transfers.size() > 0) {
                for(ContractTransfer transfer : transfers) {
                    addressesSet.add(new ByteArrayWrapper(transfer.getFrom()));
                    addressesSet.add(new ByteArrayWrapper(transfer.getTo()));
                }
            }
            //
            List<byte[]> parseEventContractList =
this.parseEventContract(contractResult.getEvents());
            for(byte[] contract : parseEventContractList) {
                addressesSet.add(new ByteArrayWrapper(contract));
            }


            List<byte[]> resultList = new ArrayList<>();
            for(ByteArrayWrapper wrapper : addressesSet) {
                resultList.add(wrapper.getBytes());
            }
            return resultList;
        } else {
            return super.getAllRelativeAddress();
        }
    }

    private List<byte[]> parseEventContract(List<String> events) {
        List<byte[]> result = new ArrayList<>();
        if(events == null || events.isEmpty()) {
            return result;
        }
        for(String event : events) {
            try {
                Map<String, Object> eventMap = JSONUtils.json2map(event);
                String contractAddress = (String) eventMap.get(CONTRACT_EVENT_ADDRESS);
                result.add(AddressTool.getAddress(contractAddress));
            } catch (Exception e) {
                Log.error(e);
            }
        }
        return result;
    }

    public void setContractTransferTxs(Collection<ContractTransferTransaction>
contractTransferTxs) {
        this.contractTransferTxs = contractTransferTxs;
    }

    public Collection<ContractTransferTransaction> getContractTransferTxs() {
        return contractTransferTxs;
    }
}
```

108:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\contract\src\main\java\io\nuls\contract\entity\tx\ContractTransaction.java

```java
package io.nuls.contract.entity.tx;

import io.nuls.contract.dto.ContractResult;
import io.nuls.kernel.model.BlockHeader;
import io.nuls.kernel.model.Na;

/**
 * @desription:
 * @author: PierreLuo
 * @date: 2018/7/19
 */
public interface ContractTransaction {

    ContractResult getContractResult();

    void setContractResult(ContractResult contractResult);

    void setReturnNa(Na returnNa);

    BlockHeader getBlockHeader();

    void setBlockHeader(BlockHeader blockHeader);
}
```

109:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\contract\src\main\java\io\nuls\contract\entity\tx\ContractTransferTransaction.java

```java
package io.nuls.contract.entity.tx;

import io.nuls.contract.constant.ContractConstant;
import io.nuls.contract.dto.ContractTransfer;
import io.nuls.contract.entity.txdata.ContractTransferData;
import io.nuls.kernel.exception.NulsException;
import io.nuls.kernel.model.Coin;
import io.nuls.kernel.model.Na;
import io.nuls.kernel.model.Transaction;
import io.nuls.kernel.utils.NulsByteBuffer;

import java.util.Arrays;

/**
 * @desription:
 * @author: PierreLuo
 * @date: 2018/6/7
 */
public class ContractTransferTransaction extends Transaction<ContractTransferData> {

    private transient ContractTransfer transfer;

    public ContractTransferTransaction() {
        super(ContractConstant.TX_TYPE_CONTRACT_TRANSFER);
    }

    @Override
    protected ContractTransferData parseTxData(NulsByteBuffer byteBuffer) throws NulsException
{
        return byteBuffer.readNulsData(new ContractTransferData());
    }

    @Override
    public String getInfo(byte[] address) {
        Coin to = coinData.getTo().get(0);
        return "+" + to.getNa().toCoinString();
    }

    @Override
    public boolean isSystemTx() {
        return true;
    }

    @Override
    public boolean needVerifySignature() {
        return false;
    }

    public ContractTransfer getTransfer() {
        return transfer;
    }

    public ContractTransferTransaction setTransfer(ContractTransfer transfer) {
        this.transfer = transfer;
        return this;
    }

    @Override
    public Na getFee() {
        ContractTransferData data = this.txData;
        byte success = data.getSuccess();
        // Gas
        if(success == 1) {
            return Na.ZERO;
        }
        //
        Na fee = Na.ZERO;
        if (null != coinData) {
            fee = coinData.getFee();
        }
        return fee;
    }
}
```

110:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\contract\src\main\java\io\nuls\contract\entity\tx\CreateContractTransaction.java

```
package io.nuls.contract.entity.tx;

import io.nuls.contract.constant.ContractConstant;
import io.nuls.contract.dto.ContractResult;
import io.nuls.contract.entity.txdata.CreateContractData;
import io.nuls.kernel.exception.NulsException;
import io.nuls.kernel.model.BlockHeader;
import io.nuls.kernel.model.Coin;
import io.nuls.kernel.model.Na;
import io.nuls.kernel.model.Transaction;
import io.nuls.kernel.utils.NulsByteBuffer;

import java.util.Arrays;
import java.util.List;

/**
 * @Desription:
 * @Author: PierreLuo
 * @Date: 2018/4/20
 */
public class CreateContractTransaction extends Transaction<CreateContractData> implements ContractTransaction{

    private ContractResult contractResult;
    private transient Na returnNa;

    private transient BlockHeader blockHeader;

    public CreateContractTransaction() {
        super(ContractConstant.TX_TYPE_CREATE_CONTRACT);
    }

    @Override
    protected CreateContractData parseTxData(NulsByteBuffer byteBuffer) throws NulsException {
        return byteBuffer.readNulsData(new CreateContractData());
    }

    /**
     *
     *
     * : 1.
     * `getInfo`
     * toListCoin()
     *      Token`from`,`to`
     *       `0`
     * @param address
     * @return
     */
    @Override
    public String getInfo(byte[] address) {
        List<Coin> toList = coinData.getTo();
        int size = toList.size();
        if(size == 1) {
            if (Arrays.equals(address, toList.get(0).getAddress())) {
                return "-" + getFee().toCoinString();
            } else {
                return "0";
            }
        } else {
            return "--";
        }
    }

    @Override
    public ContractResult getContractResult() {
        return contractResult;
    }

    @Override
    public void setContractResult(ContractResult contractResult) {
        this.contractResult = contractResult;
    }

    @Override
    public void setReturnNa(Na returnNa) {
        this.returnNa = returnNa;
    }

    @Override
    public BlockHeader getBlockHeader() {
        return blockHeader;
    }

    @Override
    public void setBlockHeader(BlockHeader blockHeader) {
        this.blockHeader = blockHeader;
    }

    @Override
    public Na getFee() {
        Na resultFee = super.getFee();
        if(returnNa != null) {
            resultFee = resultFee.minus(returnNa);
        }
        return resultFee;
    }
}
```

111:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\contract\src\main\java\io\nuls\contract\entity\tx\DeleteContractTransaction.java

```
package io.nuls.contract.entity.tx;

import io.nuls.contract.constant.ContractConstant;
import io.nuls.contract.dto.ContractResult;
import io.nuls.contract.entity.txdata.DeleteContractData;
import io.nuls.kernel.exception.NulsException;
import io.nuls.kernel.model.BlockHeader;
import io.nuls.kernel.model.Na;
import io.nuls.kernel.model.Transaction;
import io.nuls.kernel.utils.NulsByteBuffer;

/**
 * @Desription:
 * @Author: PierreLuo
 * @Date: 2018/4/20
 */
public class DeleteContractTransaction extends Transaction<DeleteContractData> implements
ContractTransaction{

    private ContractResult contractResult;

    private transient Na returnNa;

    private transient BlockHeader blockHeader;

    public DeleteContractTransaction() {
        super(ContractConstant.TX_TYPE_DELETE_CONTRACT);
    }

    @Override
    protected DeleteContractData parseTxData(NulsByteBuffer byteBuffer) throws NulsException {
        return byteBuffer.readNulsData(new DeleteContractData());
    }

    @Override
    public String getInfo(byte[] address) {
        return "-" + getFee().toCoinString();
    }

    @Override
    public ContractResult getContractResult() {
        return contractResult;
    }

    @Override
    public void setContractResult(ContractResult contractResult) {
        this.contractResult = contractResult;
    }
    @Override
    public void setReturnNa(Na returnNa) {
        this.returnNa = returnNa;
    }

    @Override
    public BlockHeader getBlockHeader() {
        return blockHeader;
    }

    @Override
    public void setBlockHeader(BlockHeader blockHeader) {
        this.blockHeader = blockHeader;
    }

    @Override
    public Na getFee() {
        Na resultFee = super.getFee();
        if(returnNa != null) {
            resultFee = resultFee.minus(returnNa);
        }
        return resultFee;
    }

}
```

112:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\contract\src\main\java\io\nuls\contract\entity\txdata\CallContractData.java

```
package io.nuls.contract.entity.txdata;


import io.nuls.kernel.exception.NulsException;
import io.nuls.kernel.model.Address;
import io.nuls.kernel.model.TransactionLogicData;
import io.nuls.kernel.utils.NulsByteBuffer;
import io.nuls.kernel.utils.NulsOutputStreamBuffer;
import io.nuls.kernel.utils.SerializeUtils;

import java.io.IOException;
import java.util.HashSet;
import java.util.Set;
```

```
/**
 * @Author: PierreLuo
 */
public class CallContractData extends TransactionLogicData implements ContractData{

    private byte[] sender;
    private byte[] contractAddress;
    private long value;
    private long gasLimit;
    private long price;
    private String methodName;
    private String methodDesc;
    private byte argsCount;
    private String[][] args;

    @Override
    public int size() {
        int size = 0;
        size += Address.ADDRESS_LENGTH;
        size += Address.ADDRESS_LENGTH;
        size += SerializeUtils.sizeOfInt64();
        size += SerializeUtils.sizeOfInt64();
        size += SerializeUtils.sizeOfInt64();

        size += SerializeUtils.sizeOfString(methodName);
        size += SerializeUtils.sizeOfString(methodDesc);
        size += 1;
        if(args != null) {
            for(String[] arg : args) {
                if(arg == null) {
                    size += 1;
                } else {
                    size += 1;
                    for(String str : arg) {
                        size += SerializeUtils.sizeOfString(str);
                    }
                }
            }
        }
        return size;
    }
```

```
@Override
protected void serializeToStream(NulsOutputStreamBuffer stream) throws IOException {
    stream.write(sender);
    stream.write(contractAddress);
    stream.writeInt64(value);
    stream.writeInt64(gasLimit);
    stream.writeInt64(price);

    stream.writeString(methodName);
    stream.writeString(methodDesc);
    stream.write(argsCount);
    if(args != null) {
        for(String[] arg : args) {
            if(arg == null) {
                stream.write((byte) 0);
            } else {
                stream.write((byte) arg.length);
                for(String str : arg){
                    stream.writeString(str);
                }
            }
        }
    }
}

@Override
public void parse(NulsByteBuffer byteBuffer) throws NulsException {
    this.sender = byteBuffer.readBytes(Address.ADDRESS_LENGTH);
    this.contractAddress = byteBuffer.readBytes(Address.ADDRESS_LENGTH);
    this.value = byteBuffer.readInt64();
    this.gasLimit = byteBuffer.readInt64();
    this.price = byteBuffer.readInt64();

    this.methodName = byteBuffer.readString();
    this.methodDesc = byteBuffer.readString();
    this.argsCount = byteBuffer.readByte();
    byte length = this.argsCount;
    this.args = new String[length][];
    for(byte i = 0; i < length; i++) {
        byte argCount = byteBuffer.readByte();
        if(argCount == 0) {
            args[i] = new String[0];
        } else {
            String[] arg = new String[argCount];
            for(byte k = 0; k < argCount; k++) {
                arg[k] = byteBuffer.readString();
            }
            args[i] = arg;
        }
    }
}

@Override
public byte[] getSender() {
    return sender;
}

public void setSender(byte[] sender) {
    this.sender = sender;
}

@Override
public byte[] getContractAddress() {
    return contractAddress;
}

public void setContractAddress(byte[] contractAddress) {
    this.contractAddress = contractAddress;
}

@Override
public long getValue() {
    return value;
}

public void setValue(long value) {
    this.value = value;
}

@Override
public long getGasLimit() {
    return gasLimit;
}
```

```
public void setGasLimit(long gasLimit) {
    this.gasLimit = gasLimit;
}

@Override
public long getPrice() {
    return price;
}

public void setPrice(long price) {
    this.price = price;
}

public String getMethodName() {
    return methodName;
}

public void setMethodName(String methodName) {
    this.methodName = methodName;
}

public String getMethodDesc() {
    return methodDesc;
}

public void setMethodDesc(String methodDesc) {
    this.methodDesc = methodDesc;
}

public byte getArgsCount() {
    return argsCount;
}

public void setArgsCount(byte argsCount) {
    this.argsCount = argsCount;
}

public String[][] getArgs() {
    return args;
}

public void setArgs(String[][] args) {
        this.args = args;
    }

    @Override
    public Set<byte[]> getAddresses() {
        Set<byte[]> addressSet = new HashSet<>();
        addressSet.add(contractAddress);
        return addressSet;
    }
}
```

113:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\contract\src\main\java\io\nuls\contract\entity\txdata\ContractData.java

```
package io.nuls.contract.entity.txdata;

/**
 * @desription:
 * @author: PierreLuo
 * @date: 2018/7/19
 */
public interface ContractData {

    long getGasLimit();

    byte[] getSender();

    byte[] getContractAddress();

    long getPrice();

    long getValue();
}
```

114:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\contract\src\main\java\io\nuls\contract\entity\txdata\ContractTransferData.java

```
package io.nuls.contract.entity.txdata;


import io.nuls.kernel.exception.NulsException;
import io.nuls.kernel.model.Address;
import io.nuls.kernel.model.NulsDigestData;
import io.nuls.kernel.model.TransactionLogicData;
import io.nuls.kernel.utils.NulsByteBuffer;
import io.nuls.kernel.utils.NulsOutputStreamBuffer;
import io.nuls.kernel.utils.SerializeUtils;

import java.io.IOException;
import java.util.HashSet;
import java.util.Set;

/**
 * @Author: PierreLuo
 */
public class ContractTransferData extends TransactionLogicData implements ContractData{

    private NulsDigestData orginTxHash;
    private byte[] contractAddress;
    private byte success;

    public ContractTransferData(){

    }

    public ContractTransferData(NulsDigestData orginTxHash, byte[] contractAddress, byte
success) {
        this.orginTxHash = orginTxHash;
        this.contractAddress = contractAddress;
        this.success = success;
    }

    @Override
    public int size() {
        int size = 0;
        size += SerializeUtils.sizeOfNulsData(orginTxHash);
        size += Address.ADDRESS_LENGTH;
        size += 1;
        return size;
    }

    @Override
    protected void serializeToStream(NulsOutputStreamBuffer stream) throws IOException {
        stream.writeNulsData(orginTxHash);
        stream.write(contractAddress);
        stream.write(success);
    }

    @Override
    public void parse(NulsByteBuffer byteBuffer) throws NulsException {
        this.orginTxHash = byteBuffer.readHash();
        this.contractAddress = byteBuffer.readBytes(Address.ADDRESS_LENGTH);
        this.success = byteBuffer.readByte();
    }

    @Override
    public Set<byte[]> getAddresses() {
        Set<byte[]> addressSet = new HashSet<>();
        addressSet.add(contractAddress);
        return addressSet;
    }

    public byte getSuccess() {
        return success;
    }

    public void setSuccess(byte success) {
        this.success = success;
    }

    @Override
    public byte[] getContractAddress() {
        return contractAddress;
    }

    public void setContractAddress(byte[] contractAddress) {
        this.contractAddress = contractAddress;
    }

    public NulsDigestData getOrginTxHash() {
        return orginTxHash;
    }

    public void setOrginTxHash(NulsDigestData orginTxHash) {
        this.orginTxHash = orginTxHash;
    }

    @Override
```

```
    public long getGasLimit() {
        return 0L;
    }

    @Override
    public byte[] getSender() {
        return null;
    }

    @Override
    public long getPrice() {
        return 0L;
    }

    @Override
    public long getValue() {
        return 0L;
    }


}
```

115:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\contract\src\main\java\io\nuls\contract\entity\txdata\CreateContractData.java

```
package io.nuls.contract.entity.txdata;


import io.nuls.kernel.exception.NulsException;
import io.nuls.kernel.model.Address;
import io.nuls.kernel.model.TransactionLogicData;
import io.nuls.kernel.utils.NulsByteBuffer;
import io.nuls.kernel.utils.NulsOutputStreamBuffer;
import io.nuls.kernel.utils.SerializeUtils;

import java.io.IOException;
import java.util.HashSet;
import java.util.Set;

/**
 * @Author: PierreLuo
 */
public class CreateContractData extends TransactionLogicData implements ContractData{
```

```
private byte[] sender;
private byte[] contractAddress;
private long value;
private int codeLen;
private byte[] code;
private long gasLimit;
private long price;
private byte argsCount;
private String[][] args;

@Override
public int size() {
    int size = 0;
    size += Address.ADDRESS_LENGTH;
    size += Address.ADDRESS_LENGTH;
    size += SerializeUtils.sizeOfInt64();
    size += SerializeUtils.sizeOfInt32();
    size += SerializeUtils.sizeOfBytes(code);
    size += SerializeUtils.sizeOfInt64();
    size += SerializeUtils.sizeOfInt64();
    size += 1;
    if(args != null) {
        for(String[] arg : args) {
            if(arg == null) {
                size += 1;
            } else {
                size += 1;
                for(String str : arg) {
                    size += SerializeUtils.sizeOfString(str);
                }
            }
        }
    }
    return size;
}

@Override
protected void serializeToStream(NulsOutputStreamBuffer stream) throws IOException {
    stream.write(sender);
    stream.write(contractAddress);
    stream.writeInt64(value);
    stream.writeUint32(codeLen);
    stream.writeBytesWithLength(code);
    stream.writeInt64(gasLimit);
    stream.writeInt64(price);
    stream.write(argsCount);
    if(args != null) {
        for(String[] arg : args) {
            if(arg == null) {
                stream.write((byte) 0);
            } else {
                stream.write((byte) arg.length);
                for(String str : arg){
                    stream.writeString(str);
                }
            }
        }
    }
}

@Override
public void parse(NulsByteBuffer byteBuffer) throws NulsException {
    this.sender = byteBuffer.readBytes(Address.ADDRESS_LENGTH);
    this.contractAddress = byteBuffer.readBytes(Address.ADDRESS_LENGTH);
    this.value = byteBuffer.readInt64();
    this.codeLen = byteBuffer.readInt32();
    this.code = byteBuffer.readByLengthByte();
    this.gasLimit = byteBuffer.readInt64();
    this.price = byteBuffer.readInt64();
    this.argsCount = byteBuffer.readByte();
    byte length = this.argsCount;
    this.args = new String[length][];
    for(byte i = 0; i < length; i++) {
        byte argCount = byteBuffer.readByte();
        if(argCount == 0) {
            args[i] = new String[0];
        } else {
            String[] arg = new String[argCount];
            for(byte k = 0; k < argCount; k++) {
                arg[k] = byteBuffer.readString();
            }
            args[i] = arg;
        }
    }
}

@Override
public long getValue() {
    return value;
}

public void setValue(long value) {
    this.value = value;
}

@Override
public byte[] getSender() {
    return sender;
}

public void setSender(byte[] sender) {
    this.sender = sender;
}

@Override
public byte[] getContractAddress() {
    return contractAddress;
}

public void setContractAddress(byte[] contractAddress) {
    this.contractAddress = contractAddress;
}

public int getCodeLen() {
    return codeLen;
}

public void setCodeLen(int codeLen) {
    this.codeLen = codeLen;
}

public byte[] getCode() {
    return code;
}
```

```
public void setCode(byte[] code) {
    this.code = code;
}

@Override
public long getGasLimit() {
    return gasLimit;
}

public void setGasLimit(long gasLimit) {
    this.gasLimit = gasLimit;
}

@Override
public long getPrice() {
    return price;
}

public void setPrice(long price) {
    this.price = price;
}

public byte getArgsCount() {
    return argsCount;
}

public void setArgsCount(byte argsCount) {
    this.argsCount = argsCount;
}

public String[][] getArgs() {
    return args;
}

public void setArgs(String[][] args) {
    this.args = args;
}

@Override
public Set<byte[]> getAddresses() {
    Set<byte[]> addressSet = new HashSet<>();
    addressSet.add(contractAddress);
```

```
        return addressSet;
    }

}
```

116:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\contract\src\main\java\io\nuls\contract\entity\txdata\DeleteContractData.java

```
package io.nuls.contract.entity.txdata;


import io.nuls.kernel.exception.NulsException;
import io.nuls.kernel.model.Address;
import io.nuls.kernel.model.TransactionLogicData;
import io.nuls.kernel.utils.NulsByteBuffer;
import io.nuls.kernel.utils.NulsOutputStreamBuffer;

import java.io.IOException;
import java.util.HashSet;
import java.util.Set;

/**
 * @Author: PierreLuo
 */
public class DeleteContractData extends TransactionLogicData implements ContractData{

    private byte[] sender;
    private byte[] contractAddress;

    @Override
    public int size() {
        int size = 0;
        size += Address.ADDRESS_LENGTH;
        size += Address.ADDRESS_LENGTH;
        return size;
    }

    @Override
    protected void serializeToStream(NulsOutputStreamBuffer stream) throws IOException {
        stream.write(sender);
        stream.write(contractAddress);
    }
```

```
@Override
public void parse(NulsByteBuffer byteBuffer) throws NulsException {
    this.sender = byteBuffer.readBytes(Address.ADDRESS_LENGTH);
    this.contractAddress = byteBuffer.readBytes(Address.ADDRESS_LENGTH);
}

@Override
public long getGasLimit() {
    return 0L;
}

@Override
public byte[] getSender() {
    return sender;
}

@Override
public long getPrice() {
    return 0L;
}

@Override
public long getValue() {
    return 0L;
}

public void setSender(byte[] sender) {
    this.sender = sender;
}

@Override
public byte[] getContractAddress() {
    return contractAddress;
}

public void setContractAddress(byte[] contractAddress) {
    this.contractAddress = contractAddress;
}

@Override
public Set<byte[]> getAddresses() {
    Set<byte[]> addressSet = new HashSet<>();
```

```
        addressSet.add(contractAddress);
        return addressSet;
    }
}
```

117:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\contract\src\main\java\io\nuls\contract\module\AbstractContractModule.java

```
package io.nuls.contract.module;


import io.nuls.contract.constant.ContractConstant;
import io.nuls.kernel.constant.NulsConstant;
import io.nuls.kernel.module.BaseModuleBootstrap;


public abstract class AbstractContractModule extends BaseModuleBootstrap {
    public AbstractContractModule() {
        super(ContractConstant.MODULE_ID_CONTRACT);
    }
}
```

118:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-module\contract\src\main\java\io\nuls\contract\service\ContractService.java

```
package io.nuls.contract.service;

import io.nuls.contract.dto.ContractResult;
import io.nuls.contract.dto.ContractTokenInfo;
import io.nuls.contract.dto.ContractTokenTransferInfoPo;
import io.nuls.contract.dto.ContractTransfer;
import io.nuls.contract.entity.tx.ContractTransferTransaction;
import io.nuls.kernel.exception.NulsException;
import io.nuls.kernel.model.*;
import io.nuls.kernel.validate.ValidateResult;

import java.util.List;
import java.util.Map;
import java.util.Set;

public interface ContractService {


    /**
```

```
 *
 *
 * @param addressBytes
 * @return
 */
boolean isContractAddress(byte[] addressBytes);


/**
 *  txInfo : key -> contractAddress + txHash, status is confirmed
 *  UTXO : key -> txHash + index
 *
 * @param txs
 * @return
 */
Result<Integer> saveConfirmedTransactionList(List<Transaction> txs);

/**
 *
 *
 * @param txs
 * @return
 */
Result<Integer> rollbackTransactionList(List<Transaction> txs);


/**
 *
 *
 * @param hash
 * @param contractResult
 * @return
 */
Result saveContractExecuteResult(NulsDigestData hash, ContractResult contractResult);

/**
 *
 *
 * @param hash
 * @return
 */
Result deleteContractExecuteResult(NulsDigestData hash);
```

```
/**
 *
 *
 * @param hash
 * @return
 */
ContractResult getContractExecuteResult(NulsDigestData hash);


/**
 *
 *
 */
void createContractTempBalance();

/**
 *
 *
 */
void removeContractTempBalance();

/**
 * token
 *
 * @param address
 * @param contractAddress
 * @return
 */
Result<ContractTokenInfo> getContractTokenViaVm(String address, String contractAddress);

/**
 * token
 *
 * @param address
 * @param contractAddress
 * @return
 */
Result initAllTokensByAccount(String address);

/**
 *
```

```
     *
     * @param address
     * @return
     */
    Result<List<ContractTokenInfo>> getAllTokensByAccount(String address);

    /**
     * Token
     *
     * @param address
     * @return
     */
    boolean isTokenContractAddress(String address);

    /**
     * Token
     *
     * @param address
     * @return
     */
    Result<List<ContractTokenTransferInfoPo>> getTokenTransferInfoList(String address);

    /**
     * Token
     *
     * @param address
     * @param hash
     * @return
     */
    Result<List<ContractTokenTransferInfoPo>> getTokenTransferInfoList(String address,
NulsDigestData hash);

    Result<byte[]> handleContractResult(Transaction tx, ContractResult contractResult, byte[]
stateRoot, long time, Map<String,Coin> toMaps, Map<String, Coin> contractUsedCoinMap);

    Result saveContractTransferTx(ContractTransferTransaction transferTx);
    Result rollbackContractTransferTx(ContractTransferTransaction transferTx);

    Result<byte[]> verifyContractResult(Transaction tx, ContractResult contractResult, byte[]
stateRoot, long time, Map<String,Coin> toMaps, Map<String,Coin> contractUsedCoinMap);
    Result<byte[]> verifyContractResult(Transaction tx, ContractResult contractResult, byte[]
stateRoot, long time, Map<String,Coin> toMaps, Map<String,Coin> contractUsedCoinMap, Long
blockHeight);

    Result<ContractResult> batchPackageTx(Transaction tx, long bestHeight, Block block, byte[]
stateRoot, Map<String, Coin> toMaps, Map<String, Coin> contractUsedCoinMap);
    Result<ContractResult> batchProcessTx(Transaction tx, long bestHeight, Block block, byte[]
stateRoot, Map<String, Coin> toMaps, Map<String, Coin> contractUsedCoinMap, boolean
isForkChain);

    /**
     * ()
     * @return
     */
    Result<List<ContractTransferTransaction>> loadAllContractTransferTxList();

    void createBatchExecute(byte[] stateRoot);

    Result<byte[]> commitBatchExecute();

    void removeBatchExecute();

    void createCurrentBlockHeader(BlockHeader tempHeader);

    void removeCurrentBlockHeader();
}

119:F:\git\coin\nuls\nuls-1.1.3\nuls\contract-
module\contract\src\main\java\io\nuls\contract\util\ContractUtil.java
package io.nuls.contract.util;

import io.nuls.contract.constant.ContractConstant;
import io.nuls.contract.dto.ContractTokenTransferInfoPo;
import io.nuls.core.tools.json.JSONUtils;
import io.nuls.core.tools.log.Log;
import io.nuls.core.tools.str.StringUtils;
import io.nuls.kernel.context.NulsContext;
import io.nuls.kernel.model.BlockHeader;
import io.nuls.kernel.model.Transaction;
import io.nuls.kernel.utils.AddressTool;
import io.nuls.kernel.utils.VarInt;

import java.lang.reflect.Array;
import java.math.BigInteger;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collection;
import java.util.Map;

import static io.nuls.contract.constant.ContractConstant.*;

/**
 * @desription:
 * @author: PierreLuo
 * @date: 2018/8/25
 */
public class ContractUtil {

    /**
     * BlockExtendsData
     */
    private static final int BLOCK_EXTENDS_DATA_FIX_LENGTH = 28;

    private static final String STRING = "String";

    public static String[][] twoDimensionalArray(Object[] args, String[] types) {
        if (args == null) {
            return null;
        } else {
            int length = args.length;
            String[][] two = new String[length][];
            Object arg;
            for (int i = 0; i < length; i++) {
                arg = args[i];
                if(arg == null) {
                    two[i] = new String[0];
                    continue;
                }
                if(arg instanceof String) {
                    String argStr = (String) arg;
                    // String ->
                    if(types != null && StringUtils.isBlank(argStr) &&
!STRING.equalsIgnoreCase(types[i])) {
                        two[i] = new String[0];
                    } else {
                        two[i] = new String[]{argStr};
                    }
                } else if(arg.getClass().isArray()) {
                    int len = Array.getLength(arg);
                    String[] result = new String[len];
                    for(int k = 0; k < len; k++) {
                        result[k] = valueOf(Array.get(arg, k));
                    }
                    two[i] = result;
                } else if(arg instanceof ArrayList) {
                    ArrayList resultArg = (ArrayList) arg;
                    int size = resultArg.size();
                    String[] result = new String[size];
                    for(int k = 0; k < size; k++) {
                        result[k] = valueOf(resultArg.get(k));
                    }
                    two[i] = result;
                } else {
                    two[i] = new String[]{valueOf(arg)};
                }
            }
            return two;
        }
    }

    public static String[][] twoDimensionalArray(Object[] args) {
        return twoDimensionalArray(args, null);
    }

    public static boolean isLegalContractAddress(byte[] addressBytes) {
        if(addressBytes == null) {
            return false;
        }
        return AddressTool.validContractAddress(addressBytes);
    }

    public static String valueOf(Object obj) {
        return (obj == null) ? null : obj.toString();
    }

    public static ContractTokenTransferInfoPo convertJsonToTokenTransferInfoPo(String event) {
        if(StringUtils.isBlank(event)) {
            return null;
```

```
        }
        ContractTokenTransferInfoPo po;
        try {
            Map<String, Object> eventMap = JSONUtils.json2map(event);
            String eventName = (String) eventMap.get(CONTRACT_EVENT);
            String contractAddress = (String) eventMap.get(CONTRACT_EVENT_ADDRESS);
            po = new ContractTokenTransferInfoPo();
            po.setContractAddress(contractAddress);
            if(NRC20_EVENT_TRANSFER.equals(eventName)) {
                Map<String, Object> data = (Map<String, Object>)
eventMap.get(CONTRACT_EVENT_DATA);
                Collection<Object> values = data.values();
                int i = 0;
                String transferEventdata;
                byte[] addressBytes;
                for(Object object : values) {
                    transferEventdata = (String) object;
                    if(i == 0 || i == 1) {
                        if(AddressTool.validAddress(transferEventdata)) {
                            addressBytes = AddressTool.getAddress(transferEventdata);
                            if(i == 0) {
                                po.setFrom(addressBytes);
                            } else {
                                po.setTo(addressBytes);
                            }
                        }
                    }
                    if(i == 2) {
                        po.setValue(StringUtils.isBlank(transferEventdata) ? BigInteger.ZERO : new
BigInteger(transferEventdata));
                        break;
                    }
                    i++;
                }
                return po;
            }
            return null;
        } catch (Exception e) {
            Log.error(e);
            return null;
        }
    }
```

```
public static boolean isContractTransaction(Transaction tx) {
    if (tx == null) {
        return false;
    }
    int txType = tx.getType();
    if(txType == ContractConstant.TX_TYPE_CREATE_CONTRACT
            || txType == ContractConstant.TX_TYPE_CALL_CONTRACT
            || txType == ContractConstant.TX_TYPE_DELETE_CONTRACT
            || txType == ContractConstant.TX_TYPE_CONTRACT_TRANSFER) {
        return true;
    }
    return false;
}

public static boolean isGasCostContractTransaction(Transaction tx) {
    if (tx == null) {
        return false;
    }
    int txType = tx.getType();
    if(txType == ContractConstant.TX_TYPE_CREATE_CONTRACT
            || txType == ContractConstant.TX_TYPE_CALL_CONTRACT) {
        return true;
    }
    return false;
}

public static boolean isLockContract(long blockHeight) {
    if (blockHeight > 0) {
        long bestBlockHeight = NulsContext.getInstance().getBestHeight();
        long confirmCount = bestBlockHeight - blockHeight;
        if (confirmCount < 7) {
            return true;
        }
    }
    return false;
}

public static byte[] getStateRoot(BlockHeader blockHeader) {
    if(blockHeader == null || blockHeader.getExtend() == null) {
        return null;
    }
```

```
        byte[] stateRoot = blockHeader.getStateRoot();
        if(stateRoot != null && stateRoot.length > 0) {
            return stateRoot;
        }
        try {
            byte[] extend = blockHeader.getExtend();
            if(extend.length > BLOCK_EXTENDS_DATA_FIX_LENGTH) {
                VarInt varInt = new VarInt(extend, BLOCK_EXTENDS_DATA_FIX_LENGTH);
                int lengthFieldSize = varInt.getOriginalSizeInBytes();
                int stateRootlength = (int) varInt.value;
                stateRoot = new byte[stateRootlength];
                System.arraycopy(extend, BLOCK_EXTENDS_DATA_FIX_LENGTH + lengthFieldSize,
stateRoot, 0, stateRootlength);
                blockHeader.setStateRoot(stateRoot);
                return stateRoot;
            }
        } catch (Exception e) {
            Log.error("parse stateRoot error.", e);
        }
        return null;
    }

    public static String bigInteger2String(BigInteger bigInteger) {
        if(bigInteger == null) {
            return null;
        }
        return bigInteger.toString();
    }

    public static String simplifyErrorMsg(String errorMsg) {
        String resultMsg = "contract error - ";
        if(StringUtils.isBlank(errorMsg)) {
            return resultMsg;
        }
        if(errorMsg.contains("Exception:")) {
            String[] msgs = errorMsg.split("Exception:", 2);
            return resultMsg + msgs[1].trim();
        }
        return resultMsg + errorMsg;
    }

    public static boolean isTerminatedContract(int status) {
        return ContractConstant.STOP == status;
    }

    public static boolean isTransferMethod(String method) {
        return (ContractConstant.NRC20_METHOD_TRANSFER.equals(method)
                || ContractConstant.NRC20_METHOD_TRANSFER_FROM.equals(method));
    }

    public static String argToString(String[][] args) {
        if(args == null) {
            return "";
        }
        String result = "";
        for(String[] a : args) {
            result += Arrays.toString(a) + "| ";
        }
        return result;
    }

    public static boolean checkPrice(long price) {
        if(price < ContractConstant.CONTRACT_MINIMUM_PRICE) {
            return false;
        }
        return true;
    }
}
```